Article 1: Sorting

Sorting is a useful technique which is used vastly from program to program. Sorting is a programming technique which is used to sort a list of pre-stored data list in an ascending or descending order according to a preset criterion.

There are several types of sorting, and one must choose a sorting method which best suites its application. One sorting method may be faster than another but slower in some other cases depending on the current situation. Thus, one must make sure the proper characteristics of a particular data items list, such as the number of items and the data type of the subject to be sorted.

Back To Top ⇧

Sorting Methods and the Big-O Notation

There are various sorting methods which are used according to a particular situation. All of the sorting methods are compared with one another using a time unit related measure called the **big-O** notation. The big-O notation is used to measure the efficiency of an algorithm which performs a particular function over a collection of items of size *n*. For example, the big-O complexity of both the Insertion Sort and the Bubble Sort is $O(n^2)$. Although both have got the same time complexity, Bubble Sort is slower. Now let's just go straight to the point of our subject i.e. sorting. There are lots of useful sorting methods, though I will be implementing and discussing only three sorting algorithms. For the rest, I will leave them for you to discover and implement yourself. *Users may feel free to note down and discuss some other interesting sorting algorithms (also on the context that is being discussed) in the user comments below.*

Back To Top ⇧

Sorting Method #1: Bubble Sort

The Bubble Sort algorithm is simple, inefficient sorting algorithm. It is not recommended for use, since its performance at sorting a list of items is terribly slow. It is best at sorting a small list of items, but not for large ones.

The sorting time complexity of the Bubble Sort is $O(n^2)$.

Implementation

```
Procedure BubbleSort(numbers : Array of Integer; size :
Integer);
Var
        i, j, temp : Integer;
```

```
Begin
        For i := size-1 DownTo 1 do
                For j := 2 to i do
                        If (numbers[j-1] > numbers[j]) Then
                        Begin
                                temp := numbers[j-1];
                                numbers[j-1] := numbers[j];
                                numbers[j] := temp;
                        End;

End.
```

This may be freely used in your programs and start off learning sorting algorithms using this simple method.

Back To Top ⇧

Sorting Method #2: Insertion Sort

Insertion Sort algorithm is a bit more efficient sorting algorithm than Bubble Sort. As it name implies, the insertion sort algorithm inserts an unsorted item in an already sorted item list. This makes you think of the use of two seperated arrays - one unsorted and the other sorted. However, to save space one uses the same array and uses a pointer to separate the sorted and unsorted elements of the list.

The sorting time complexity of the Insertion Sort is $O(n^2)$. Although this exactly the same to Bubble Sort's, the Insertion Sort algorithm is twice more efficient, yet inefficient for large lists.

Implementation

```
Procedure InsertionSort(numbers : Array of Integer; size :
Integer);
Var
        i, j, index : Integer;


Begin
        For i := 2 to size-1 do
        Begin
                index := numbers[i];
                j := i;
                While ((j > 1) AND (numbers[j-1] > index)) do
                Begin
                        numbers[j] := numbers[j-1];
                        j := j - 1;
                End;
```

```
                numbers[j] := index;
        End;
End.
```

Back To Top ⇧

Sorting Method #3: Quick Sort

This algorithm seems pretty fast in performance as its name suggests, though it's not easy to implement even if getting the gist of how the sorting algorithm works is not that difficult.

This sorting algorithm uses recursion extensively, so make sure you are quite familiar with recursion, and have used it a lot before trying to understand the algorithm. The quick sort works by using a "pivot". The pivot is an index pointer just like the ones used in previous sorting algorithms. The purpose of the pivot is to divide the list in two halves, one with elements greater than the pivot and the other with elements smaller than the pivot. The pivot is usually chosen to be the left-most element of the list, however it is not necessary and one may choose any random element from the list to be the pivot. Up till now, we have got the array list divided into two halves. Now, we do the same procedure over this two halves just like we did to the whole list - and this is what we call recursion. The longer the list, the more recursion there will be - thus more resources are requested i.e. memory space.

Quick-sort's worst case is when the list is already sorted and choosing the left-most element as the pivot - this will obviously be a very lengthy process which turns out to be inefficient for sorting an already sorted list using a quick sort. One may think of keeping a state variable which keeps track whether a list is already sorted or not and avoid using quick sort to check if an algorithm is sorted or not. Also, if the list to be sorted has got only 1 or less elements, the function returns.

To summurize, the following steps are fundamental in doing a quick sort:

If array has got 1 or less elements, then return.
Choose a pivot from the list.
The array is divided into two sections - one with elements smaller than the pivot, the other with larger elements than the pivot
Use recursion to do the first three steps again, operating on the two divided halves

Implementation

```
Procedure QSort(numbers : Array of Integer; left : Integer;
right : Integer);
Var
        pivot, l_ptr, r_ptr : Integer;

Begin
        l_ptr := left;
        r_ptr := right;
```

```
        pivot := numbers[left];

        While (left < right) do
        Begin
                While ((numbers[right] >= pivot) AND (left <
right)) do
                        right := right - 1;

                If (left <> right) Then
                Begin
                        numbers[left] := numbers[right];
                        left := left + 1;
                End;

                While ((numbers[left] <= pivot) AND (left <
right)) do
                        left := left + 1;

                If (left <> right) Then
                Begin
                        numbers[right] := numbers[left];
                        right := right - 1;
                End;
        End;

        numbers[left] := pivot;
        pivot := left;
        left := l_ptr;
        right := r_ptr;

        If (left < pivot) Then
                QSort(numbers, left, pivot-1);

        If (right > pivot) Then
                QSort(numbers, pivot+1, right);
End;

Procedure QuickSort(numbers : Array of Integer; size : Integer);
Begin
        QSort(numbers, 0, size-1);
End;
```

So far so good, we've completed discussing three sorting algorithms. There are a number of sorting algorithms that were left out of context including Selection Sort, Heap Sort and Merge Sort. These are three other important sorting algorithms which more or less work in the same way like the algorithms we have just discussed. Like for example, the Merge Sort algorithm uses division of the list and recurssion as much as Quick Sort do.

Back To Top ⇧