



INTERNAL SORTING

Introduction

16.1

Sorting in English language refers to separating or arranging things according to different classes. However, in computer science, *sorting* also referred to as *ordering* deals with arranging elements of a list or a set or records of a file in the ascending or descending order.

In the case of sorting a list of alphabetical or numerical or alphanumerical elements, the elements are arranged in their ascending or descending order based on their alphabetical or numerical sequence number. The sequence is also referred to as a *collating sequence*. In the case of sorting a file of records, one or more fields of the records are chosen as the key based on which the records are arranged in the ascending or descending order.

Examples of lists before and after sorting are shown below:

Unsorted lists

{ 34, 12, 78, 65, 90, 11, 45}
{ tea, coffee, cocoa, milk, malt, chocolate}
{ n12x, m34b, n24x, a78h, g56v, m12k, k34d}

Sorted lists

{11, 12, 34, 45, 65, 78, 90}
{ chocolate, cocoa, coffee, malt, milk, tea}
{ a78h, g56v, k34d, m12k, m34b, n12x, n24x}

Sorting has acquired immense significance in the discipline of computer science. Several data structures and algorithms display efficient performance when presented with sorted data sets.

Many different sorting algorithms have been invented each having its own advantages and disadvantages. These algorithms may be classified into families such as *sorting by exchange*, *sorting by insertion*, *sorting by distribution*, *sorting by selection* and so on. However in many cases, it is difficult to classify the algorithms as belonging to only a specific family.

A sorting technique is said to be *stable* if keys that are equal retain their relative orders of occurrence even after sorting. In other words, if K_1, K_2 are two keys such that $K_1 = K_2$, and $p(K_1) < p(K_2)$ where $p(K_i)$ is the position index of the keys in the unsorted list, then after sorting, $p'(K_1) < p'(K_2)$ where $p'(K_i)$ is the index positions of the keys in the sorted list.

If the list of data or records to be sorted are small enough to be accommodated in the internal memory of the computer, then it is referred to as *internal sorting*. On the other hand if the data list or records to be sorted are voluminous and are accommodated in external storage devices such as tapes, disks and drums, then the sorting undertaken is referred to as *external sorting*. External sorting methods are quite different from internal sorting methods and are discussed in Chapter 17.

16.1 Introduction

16.2 Bubble Sort

16.3 Insertion Sort

16.4 Selection Sort

16.5 Merge Sort

16.6 Shell Sort

16.7 Quick Sort

16.8 Heap Sort

16.9 Radix Sort

In this chapter we discuss the internal sorting techniques of Bubble Sort, Insertion Sort, Selection sort, Merge Sort, Shell sort, Quick Sort, Heap Sort and Radix Sort.

Bubble Sort

16.2

Bubble sort belongs to the family of *sorting by exchange* or *transposition*, where during the sorting process pairs of elements that are out of order are interchanged until the whole list is ordered. Given an unordered list $L = \{K_1, K_2, K_3, \dots, K_n\}$ bubble sort orders the elements in their ascending order (i.e.), $L = \{K_1, K_2, K_3, \dots, K_n\}, K_1 \leq K_2 \leq \dots \leq K_n$.

Given the unordered list $L = \{K_1, K_2, K_3, \dots, K_n\}$, of keys, bubble sort compares pairs of elements K_i and K_j swapping them if $K_i > K_j$. At the end of the first pass of comparisons, the largest element in the list L moves to the last position in the list. In the next pass, the sublist $\{K_1, K_2, K_3, \dots, K_{n-1}\}$ is considered for sorting. Once again the pair wise comparison of elements in the sub list results in the next largest element floating to the last position of the sublist. Thus in $(n-1)$ passes where n is the number of elements in the list, the list L is sorted. The sorting is called bubble sorting for the reason that, with each pass the next largest element of the list floats or "bubbles" to its appropriate position in the sorted list.

Algorithm 16.1 illustrates the working of bubble sort.

Algorithm 16.1: Procedure for Bubble sort

```

procedure BUBBLE_SORT( $L, n$ )
    /*  $L[1:n]$  is an unordered list of data elements to be
       sorted in the ascending order */
    for  $i = 1$  to  $n-1$  do /*  $n-1$  passes*/
        for  $j = 1$  to  $n-i$  do
            if ( $L[j] > L[j+1]$ ) swap( $L[j], L[j+1]$ );
            /* swap pair wise elements*/
        end /* the next largest element "bubbles" to the last position*/
    end
end BUBBLE_SORT.
    
```

Example 16.1 Let $L = \{92, 78, 34, 23, 56, 90, 17, 52, 67, 81, 18\}$ be an unordered list. As the first step in the first pass of bubble sort, 92 is compared with 78. Since $92 > 78$, the elements are swapped yielding the list $\{78, 92, 34, 23, 56, 90, 17, 52, 67, 81, 18\}$. The swapped elements are shown in bold. Now the pair 92 and 34 are compared resulting in a swap which yields the list $\{78, 34, 92, 23, 56, 90, 17, 52, 67, 81, 18\}$. It is easy to see that at the end of pass one, the largest element of the list viz., 92 would have moved to the last position in the list. At the end of pass one, the list would be $\{78, 34, 23, 56, 90, 17, 52, 67, 81, 18, 92\}$.

In the second pass the list considered for sorting discounts the last element viz., 92 since 92 has found its appropriate position in the sorted list. At the end of the second pass, the next largest element viz., 90 would have moved to the end of the list. The partially sorted list at this point would be $\{34, 23, 56, 78, 17, 52, 67, 81, 18, 90, 92\}$. The elements shown in grey indicate elements discounted from the sorting process. In pass 10 the whole list would be completely sorted.

The trace of algorithm BUBBLE_SORT (Algorithm 16.1) over L is shown in Table 16.1. Here i keeps count of the passes and j keeps track of the pair wise element comparisons within a pass. The lower (l) and upper (u) bounds of the loop controlled by j in each pass is shown as $l..u$. Elements shown in grey and underlined in the list L at the end of pass i , indicate those discounted from the sorting process.

Table 16.1 Trace of Algorithm 16.1 over the list $L = \{92, 78, 34, 23, 56, 90, 17, 52, 67, 81, 18\}$

(Pass) i	j	List L at the end of Pass i
1	1..10	[78, 34, 23, 56, 90, 17, 52, 67, 81, 18, 92]
2	1..9	[34, 23, 56, 78, 17, 52, 67, 81, 18, 90, <u>92</u>]
3	1..8	[23, 34, 56, 17, 52, 67, 78, 18, 81, <u>90</u> , <u>92</u>]
4	1..7	[23, 34, 17, 52, 56, 67, 18, 78, <u>81</u> , <u>90</u> , <u>92</u>]
5	1..6	[23, 17, 34, 52, 56, 18, 67, <u>78</u> , <u>81</u> , <u>90</u> , <u>92</u>]
6	1..5	[17, 23, 34, 52, 18, 56, <u>67</u> , <u>78</u> , <u>81</u> , <u>90</u> , <u>92</u>]
7	1..4	[17, 23, 34, 18, 52, <u>56</u> , <u>67</u> , <u>78</u> , <u>81</u> , <u>90</u> , <u>92</u>]
8	1..3	[17, 23, 18, 34, <u>52</u> , <u>56</u> , <u>67</u> , <u>78</u> , <u>81</u> , <u>90</u> , <u>92</u>]
9	1..2	[17, 18, 23, <u>34</u> , <u>52</u> , <u>56</u> , <u>67</u> , <u>78</u> , <u>81</u> , <u>90</u> , <u>92</u>]
10	1..1	[17, 18, <u>23</u> , <u>34</u> , <u>52</u> , <u>56</u> , <u>67</u> , <u>78</u> , <u>81</u> , <u>90</u> , <u>92</u>]

Stability and performance analysis

Bubble sort is a stable sort since equal keys do not undergo swapping, as can be observed in Algorithm 16.1, and this contributes to the keys maintaining their relative orders of occurrence in the sorted list.

Example 16.2 Consider the unordered list $L = \{7^1, 7^2, 7^3, 6\}$. The repeating keys have been distinguished using their orders of occurrence as superscripts. The partially sorted lists at the end of each pass of the bubble sort algorithm are shown below:

Pass 1: $\{7^1, 7^2, 6, 7^3\}$

Pass 2: $\{7^1, 6, 7^2, 7^3\}$

Pass 3: $\{6, 7^1, 7^2, 7^3\}$

Observe how the equal keys $7^1, 7^2, 7^3$ maintain their relative orders of occurrence in the sorted list as well, verifying the stability of bubble sort.

The time complexity of bubble sort in terms of key comparisons is given by $O(n^2)$. It is easy to see this since the procedure involves two loops with their total frequency count given by $O(n^2)$.

Insertion Sort

16.3

Insertion sort as the name indicates belongs to the family of *sorting by insertion* which is based on the principle that a new key K is *inserted* at its appropriate position in an already sorted sub list.

Given an unordered list $L = \{K_1, K_2, K_3, \dots, K_n\}$, insertion sort employs the principle of constructing the list $L = \{K_1, K_2, K_3, \dots, K_i, K, K_{i+1}, \dots, K_n\}$, $K_1 \leq K_2 \leq \dots \leq K_i$ and inserting a key K at its appropriate position by comparing it with its sorted sublist of predecessors $\{K_1, K_2, K_3, \dots, K_i\}$, $K_1 \leq K_2 \leq \dots \leq K_i$ for every key K ($K = K_i = 2, 3, \dots, n$) belonging to the unordered list L .

In the first pass of insertion sort, K_2 is compared with its sorted sublist of predecessors viz., K_1 . K_2 inserts itself at the appropriate position to obtain the sorted sublist $\{K_1, K_2\}$. In the second pass, K_3 compares itself with its sorted sublist of predecessors viz., $\{K_1, K_2\}$ to insert itself at its appropriate position yielding the sorted list $\{K_1, K_2, K_3\}$ and so on. In the $(n-1)^{\text{th}}$ pass, K_n compares itself with its sorted sublist of predecessors $\{K_1, K_2, \dots, K_{n-1}\}$ and having inserted itself at the appropriate position yields the final sorted list $L = \{K_1, K_2, K_3, \dots, K_{i-1}, K, K_{i+1}, \dots, K_n\}$, $K_1 \leq K_2 \leq \dots \leq K_i \leq \dots \leq K_{i-1} \leq K \leq K_{i+1} \leq \dots \leq K_n$. Since each key K finds its appropriate position in the sorted list, such a technique is referred to as *sinking* or *sifting* technique.

Algorithm 16.2 illustrates the working of Insertion sort. The **for** loop in the algorithm keeps count of the passes and the **while** loop implements the comparison of the key key with its sorted sublist of predecessors. So long as the preceding element in the sorted sublist is greater than key the swapping of the element pair is done. If the preceding element in the sorted sublist is less than or equal to key , then key is left at its current position and the current pass terminates.

Example 16.3 Let $L = \{16, 36, 4, 22, 100, 1, 54\}$ be an unordered list of elements. The various passes of the insertion sort procedure are shown below. The snapshots of the list before and after each pass is shown. The key chosen for insertion in each pass is shown in bold and the sorted sublist of predecessors against which the key is compared are shown in brackets.

Pass 1 (Insert 36)	{ [16] 36, 4, 22, 100, 1, 54}
After Pass 1	{ [16 36] 4, 22, 100, 1, 54}
Pass 2 (Insert 4)	{ [16 36] 4, 22, 100, 1, 54}
After Pass 2	{ [4 16 36] 22, 100, 1, 54}
Pass 3 (Insert 22)	{ [4 16 36] 22, 100, 1, 54}
After Pass 3	{ [4 16 22 36] 100, 1, 54}
Pass 4 (Insert 100)	{ [4 16 22 36] 100, 1, 54}
After Pass 4	{ [4 16 22 36 100] 1, 54}
Pass 5 (Insert 1)	{ [4 16 22 36 100] 1, 54}
After Pass 5	{ [1 4 16 22 36 100] 54}
Pass 6 (Insert 54)	{ [1 4 16 22 36 100] 54}
After Pass 6	{ [1 4 16 22 36 54 100]}

Algorithm 16.2: Procedure for Insertion sort

```

procedure INSERTION_SORT(L, n)
    /* L[1:n] is an unordered list of data elements to be sorted
       in the ascending order */
    for i = 2 to n do          /* n-1 passes */
        key = L[i];           /* key is the key to be inserted
                               and position its location in the
                               unordered list */

```



```

position = 1;
/* compare key with its sorted
sublist of predecessors for insertion
at the appropriate position */
while (position > 1) and (L[position-1] > key) do
    L[position] = L[position-1];
    position = position - 1;
    L[position] = key;
end
end
end INSERTION_SORT.

```

Stability and performance analysis

Insertion sort is a stable sort. It is evident from the algorithm that the insertion of key K at its appropriate position in the sorted sublist affects the position index of the elements in the sublist so long as the elements in the sorted sublist are greater than K . When the elements are less than or equal to the key K , there is no displacement of elements and this contributes to retaining the original order of keys which are equal, in the sorted sublists.

Example 16.4 Consider the list $L = \{3^1, 1, 2^1, 3^2, 3^3, 2^2\}$ where the repeated keys have been superscripted with numbers indicative of their relative orders of occurrence. The keys for insertion are shown in bold and the sorted sublists are bracketed.

The passes of the insertion sort are shown below:

Pass 1 (Insert 1)	$\{ [3^1] 1, 2^1, 3^2, 3^3, 2^2 \}$
After Pass 1	$\{ [1 3^1] 2^1, 3^2, 3^3, 2^2 \}$
Pass 2 (Insert 2)	$\{ [1 3^1] 2^1, 3^2, 3^3, 2^2 \}$
After Pass 2	$\{ [1 2^1 3^1] 3^2, 3^3, 2^2 \}$
Pass 3 (Insert 3)	$\{ [1 2^1 3^1] 3^2, 3^3, 2^2 \}$
After Pass 3	$\{ [1 2^1 3^1 3^2] 3^3, 2^2 \}$
Pass 4 (Insert 3)	$\{ [1 2^1 3^1 3^2] 3^3, 2^2 \}$
After Pass 4	$\{ [1 2^1 3^1 3^2 3^3] 2^2 \}$
Pass 5 (Insert 2)	$\{ [1 2^1 3^1 3^2 3^3] 2^2 \}$
After Pass 5	$\{ [1 2^1 2^2 3^1 3^2 3^3] \}$

The stability of insertion sort can be easily verified on this example. Observe how keys which are equal maintain their original relative orders of occurrence in the sorted list.

The worst case performance of insertion sort occurs when the elements in the list are already sorted in their descending order. It is easy to see that in such a case every key that is to be inserted has to move to the front of the list and therefore undertakes the maximum number of comparisons. Thus if the list $L = \{K_1, K_2, K_3, \dots, K_n\}$, $K_1 \geq K_2 \geq \dots \geq K_n$ is to be insertion sorted then the number of comparisons for the insertion of key K_i would be $(i-1)$ since K_i would swap positions with each of the $(i-1)$ keys occurring before it until it moves to position 1. Therefore the total number of comparisons for inserting each of the keys is given by

$$1 + 2 + 3 + \dots (n-1) = \frac{(n-1)(n)}{2} \approx O(n^2)$$

The best case complexity of insertion sort arises when the list is already sorted in the ascending order. In such a case the complexity in terms of comparisons is given by $O(n)$.
The average case performance of insertion sort reports $O(n^2)$ complexity.

Selection Sort

16.4

Selection sort is built on the principle of repeated *selection* of elements satisfying a specific criterion to aid the sorting process.

The steps involved in the sorting process are listed below:

- (i) Given an unordered list $L = \{K_1, K_2, K_3, \dots, K_j, \dots, K_n\}$, select the minimum key K
- (ii) Swap K with the element in the first position of the list L , viz., K_1 . By doing so the minimum element of the list has secured its rightful position of number one in the sorted list. This step is termed pass 1.
- (iii) Exclude the first element and select the minimum element K , from amongst the remaining elements of the list L . Swap K with the element in the second position of the list viz., K_2 . This is termed pass 2.
- (iv) Exclude the first two elements which have occupied their rightful positions in the sorted list L . Repeat the process of selecting the next minimum element and swapping it with the appropriate element, until the entire list L gets sorted in the ascending order. The entire sorting gets done in $(n-1)$ passes.

Selection sort can also undertake sorting in the descending order by selecting the *maximum* element instead of the minimum element and swapping it with the element in the *last* position of the list L .

Algorithm 16.3 illustrates the working of selection sort. The procedure `FIND_MINIMUM(L, i, n)` selects the minimum element from the array $L[i:n]$ and returns the position index of the minimum element to procedure `SELECTION_SORT`. The `for` loop in the `SELECTION_SORT` procedure represents the $(n-1)$ passes needed to sort the array $L[1:n]$ in the ascending order. Function `swap` swaps the elements input to it.

Algorithm 16.3: Procedure for Selection sort

```






procedure SELECTION_SORT(L, n)
    /* L[1:n] is an unordered list of data elements to be
       sorted in the ascending order */
    for i = 1 to n-1 do                                /* n-1 passes*/
        minimum_index = FIND_MINIMUM(L, i, n);          /* find minimum element
                                                         of the list L[i:n] and store the position index of
                                                         the element in minimum_index*/
        swap(L[i], L[minimum_index]);
    end
end SELECTION_SORT
  
```

```

procedure FIND_MINIMUM(L, i, n)
    /* the position index of the minimum element in the array
       L[i : n] is returned */
    min_indx = i;
    for j = i + 1 to n do
        if (L[j] < L[min_indx]) min_indx = j;
    end
    return (min_indx);
end FIND_MINIMUM

```

Example 16.5 Let $L = \{71, 17, 86, 100, 54, 27\}$ be an unordered list of elements. Each pass of selection sort is traced below. The minimum element is shown in bold and the arrows indicate the swap of the elements concerned. The elements in gray indicate their exclusion in the passes concerned.

Pass	List L (During Pass)	List L (After Pass)
1	{71, 17, 86, 100, 54, 27} 	{17, 71, 86, 100, 54, 27}
2	{17, 71, 86, 100, 54, 27} 	{17, 27, 86, 100, 54, 71}
3	{17, 27, 86, 100, 54, 71} 	{17, 27, 54, 100, 86, 71}
4	{17, 27, 54, 100, 86, 71} 	{17, 27, 54, 71, 86, 100}
5	{17, 27, 54, 71, 86, 100} 	{17, 27, 54, 71, 86, 100} (Sorted list)

Stability and performance analysis

Selection sort is not stable. Example 16.6 illustrates a case. The computationally expensive portion of selection sort occurs when the minimum element has to be selected in each pass. The time complexity of `FIND_MINIMUM` procedure is $O(n)$. The time complexity of `SELECTION_SORT` procedure is therefore $O(n^2)$.

Example 16.6 Consider the list $L = \{6^1, 6^2, 2\}$. The repeating keys have been superscripted with numbers indicative of their relative orders of occurrence. A trace of the selection sort procedure is shown below. The minimum element is shown in bold and the swapping is indicated by the curved arrow. The elements excluded from the pass are shown in gray.

Pass	List L (During Pass)	List L (After Pass)
1	{ 6 ¹ , 6 ² , 2 } 	{ 2, 6 ² , 6 ¹ }
2	{ 2, 6 ² , 6 ¹ } 	{ 2, 6 ² , 6 ¹ } (Sorted list)

The selection sort on the given list L is therefore not stable.