8 Sorting

Sorting means arranging the data according to their values in some specified order, where order can be either ascending or descending. For example if we have a list of numbers {6, 2, 8, 1, 4}, then after sorting them in ascending order we get {1, 2, 4, 6, 8} and after sorting them in descending order we get {8, 6, 4, 2, 1}. Here the data that we sorted consists only of numbers, but it may be anything like strings or records. Generally we have to sort a list of records where each record contains several information fields. Sorting is done with respect to a key where key is a part of the record. Sorting these records means rearranging the records so that the key values are in order. The key on which sorting is performed is also known as the sort key.

Suppose we have several records of employees where each record contains three fields viz. name, age and salary. We can sort the records taking any one of these fields as the sort key. The table below shows the unsorted list of records and the sorted lists having name, age and salary as the sort keys one by one.

Name	Age	Salary
Sumit	25	4500
Amit	37	7800
Deepak	45	6000
Neelam	29	3200
Priya	23	9000
Shaman	50	6500
Kiran	39	5500
Chetna	34	8000

Unsorted List L

Name	Age	Salary
Amit	37	7800
Chetna	34	8000
Deepak	45	6000
Kiran	39	5500
Neelam	29	3200
Priya	23	9000
Shaman	50	6500
Sumit	25	4500

List L sorted by name in ascending order

Name	Age	Salary
Priya	23	9000
Sumit	25	4500
Neelam	29	3200
Chetna	34	8000
Amit	37	7800
Kiran	39	5500
Deepak	45	6000
Shaman	50	6500

List L sorted by age in ascending order

Name	Age	Salary
Priya	23	9000
Chetna	34	8000
Amit	37	7800
Shaman	50	6500
Deepak	45	6000
Kiran	39	5500
Sumit	25	4500
Neelam	29	3200

List L sorted by salary in descending order

Figure 8.1 Sorting on different keys

We can see that sorting the data according to different keys arranges the data in different orders.

In our algorithms we will perform sorting on a list of integer values only, so that we can focus on the logic of the algorithm. The extension of these algorithms to sort a list of records is simple. After the discussion of all sort methods, there are two programs in which we can see how to sort records.

Now let us see what is the requirement of sorting and why is it important to keep our data in sorted order. In our daily life, we can see many places where data is kept in sorted order like dictionary, telephone directory, index of books, bank accounts, merit list, roll numbers etc. Imagine the time taken to search for a word in a dictionary if the words were not arranged alphabetically or consider the case when you have to search for a name in telephone directory and the names are not sorted. Suppose you want to know where the topic "Queue" is given in this book, then obviously you will go to the index of this book to find the page number; you directly go to the words starting with 'Q' and in an instant you find your word. This was possible because words in the index were sorted. If the index was not sorted then you had only one option for searching a particular word i.e. one by one. So we see that it is easier and faster to search for an item in data that is sorted. Similarly in computer applications, sorting helps in faster information retrieval and hence the data processing operations become more efficient if data is arranged in some specific order. So practically there is no data processing application that does not perform sorting.

8.1 Sort Order

We can sort the data either in ascending (increasing) or in descending (decreasing) order. If the order is not mentioned then it is assumed to be ascending order. In this chapter we will use ascending order in our examples and algorithms. By making simple modifications, these algorithms can be made to work for descending order also.

In the case of numbers the ascending or descending order is clear. The alphabetic and nonalphabetic characters are generally ordered according to their ASCII values. The strings can be ordered using `strcmp()` function.

8.2 Types of Sorting

There are two types of sorting, internal sorting and external sorting. If the data to be sorted is small enough to be placed in main memory at a time, then the sorting process can take place in main memory and this sorting is called internal sorting. So in internal sorting all the data to be sorted is kept in the main memory during the sorting process.

If there is large amount of data to be sorted which can't be placed in main memory at a time, then the data that is currently being sorted is brought into main memory and rest is on secondary memory i.e. on external files like disks and tapes. This type of sorting is called external sorting.

In internal sort all the data is in main memory so it is easy to access any element while it is not so in external sort. In this chapter we will discuss internal sorting only.

8.3 Sort Stability

Sort stability comes into picture if the key on which data is being sorted is not unique for each record, i.e. two or more records have identical keys. For example consider a list of records where each record contains name and age of a person. We will take name as the sort key and sort all the records according to the names. The unsorted list of these records is given in the first table while the next three tables have sorted list.

Name	Age
Vineet	25
Amit	37
Deepa	67
Shriya	45
Deepa	20
Kiran	18
Deepa	56

Unsorted list

Name	Age
Amit	37
Deepa	56
Deepa	67
Deepa	20
Kiran	18
Shriya	45
Vineet	25

Sorted list (Unstable Sort)

Name	Age
Amit	37
Deepa	20
Deepa	56
Deepa	67
Kiran	18
Shriya	45
Vineet	25

Sorted list (Unstable Sort)

Name	Age
Amit	37
Deepa	67
Deepa	20
Deepa	56
Kiran	18
Shriya	45
Vineet	25

Sorted list (Stable Sort)

Figure 8.2 Stable and Unstable Sort

Any sorting algorithm would place (Amit,37) in first position, (Kiran,18) in fifth position, (Shriya,45) in sixth position and (Vineet,25) in seventh position. There are three records with identical keys(names), which are (Deepa,67), (Deepa,20) and (Deepa,56) and any sorting algorithm would place them in adjacent locations i.e. second, third and fourth locations but not necessarily in the same relative order.

A sorting algorithm is said to be stable if it maintains the relative order of the duplicate keys in the sorted output i.e. if keys are equal then their relative order in the sorted output is same. For example if records R_i and R_j have equal keys and if record R_i precedes record R_j in the input data then R_i should precede R_j in the sorted output data also if the sort is stable. If the sort is not stable then R_i and R_j may be in any order in the sorted output. So in an unstable sort the duplicate keys may occur in any order in the sorted output.

In our example the first two sorted lists did not maintain the relative order of the duplicate keys while the third one did. So we can say that the first two sorted lists were obtained by unstable sorting algorithms while the last one was obtained by a stable sorting algorithm.

Sometimes we need to sort the records according to different keys at different times i.e. records which are sorted on one key are again sorted on another key. In these types of situations an unstable sort is not desirable. Let us take an example and see why it is so.

Suppose we have a list of all students of a school consisting of their names and classes, and the list is alphabetically sorted on name, i.e. all the names are in alphabetical order. Now suppose we want to sort this list with respect to class. Any sorting algorithm will place the names of all classmates in adjacent locations, but only a stable sort will place the names of students in a particular class alphabetically.

Name	Class
Amit	12
Anuj	11
Chetan	10
Deepak	12
Karan	12
Manav	11
Neelam	10
Raj	11

List L with names in sorted order

Name	Class
Neelam	10
Chetan	10
Manav	11
Raj	11
Anuj	11
Deepak	12
Amit	12
Karan	12

List L sorted on class (Unstable Sort)

Name	Class
Chetan	10
Neelam	10
Anuj	11
Manav	11
Raj	11
Amit	12
Deepak	12
Karan	12

List L sorted on class (Stable Sort)

Figure 8.3

We can see that in stable sort we got the names of students of each class in alphabetical order while unstable sort disturbed the initial order of students who were in same class.

8.4 Sort by Address(Indirect Sort)

Sorting can be done in two ways, by actually moving the records or by maintaining an auxiliary array of pointers and rearranging the pointers in that array. Let us first see how the sorting is done by moving the records.

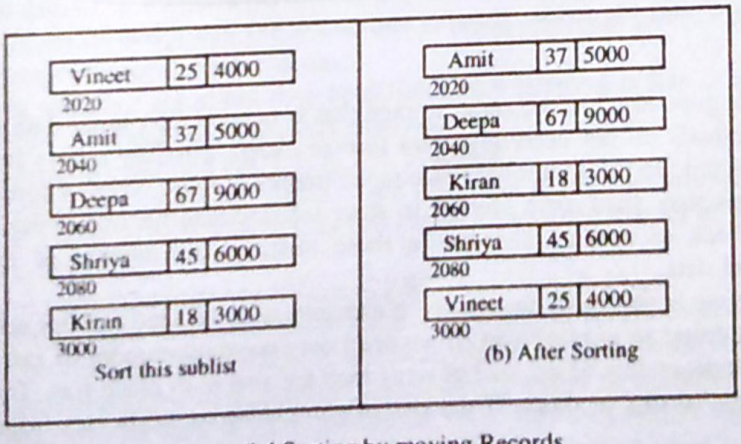


Figure 8.4 Sorting by moving Records

Here we can see that the records are moved from one place to another in the memory, for example the record (Vineet, 25, 4000) was initially stored at address 2020 but after sorting it is at address 3000.

If records to be sorted are very large then this process of moving records can be an expensive task. In this case we can take an array of pointers, which contains addresses of the records in memory. Now instead of rearranging the records, we rearrange the addresses inside the pointer array. In figure 8.5, we have performed sorting on the same records but this time by adjusting pointers in the pointer array.

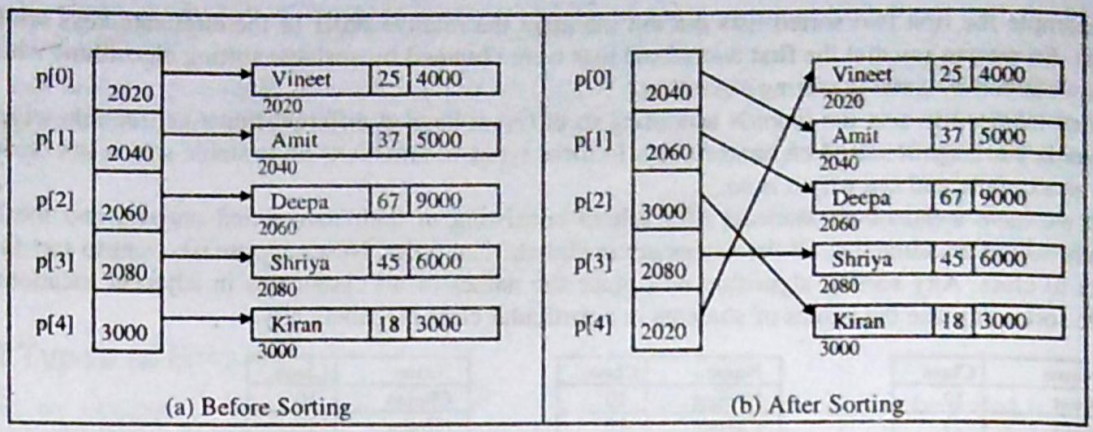


Figure 8.5 Sorting by Rearranging Pointers

Here we can see that records are at the same position but the pointers, which are pointing to records, are in different sequence but still pointing to the same record. For example, the record (Vineet, 25, 4000) was stored at address 2020 before and after the sorting. Before sorting, the first pointer of the array was pointing to first record, second pointer to the second record and so on. After sorting, first pointer will point to the first record of sorted list, second pointer will point to the second record of sorted list and so on.

This process of sorting by adjusting pointers is called sorting by address or indirect sort because we can indirectly refer to the records in sorted order. If the records are elements of an array then we can store the indices of these elements in an auxiliary array instead of storing the addresses and sorting can be done indirectly by rearranging the values of indices contained inside the auxiliary array.

In the programs that we take in this chapter, we will perform the sorting by moving the data and the extension of these programs to perform sorting by address is simple. After the discussion of all sort methods, we have given a program that illustrates sorting by address(P8.16).

Another way to avoid movement of records while sorting is to store the records in a linked list. In that case we will have to only change the pointers instead of moving the records. Here we have overhead of an extra pointer field but if the records are very large then this overhead is negligible.

8.5 In place Sort

In place sorting methods generally use the same storage that is occupied by input data to store the output data while sorting. These methods do not need any extra storage except possibly a very little amount of working storage that can be neglected. So the additional space requirements of these types of sorts are O(1).

Other sorting methods may need extra storage to store intermediate results of the sorting, and at last the sorted data is copied back to original storage. In these methods the amount of extra storage needed is proportional to the size of data.

For example suppose we have to sort an array of n elements and we need another array of size n to perform this sort then we are not doing an in place sort. If we need only constant number of extra variables, i.e. number of variables required is independent of the size of array then the sort is in place sort. The in place sorts are also known as minimal storage sorting methods. If the size of array to be sorted is very large then it is beneficial to use an in place sort.

Merge sort is not an in place sort because it requires an extra array of size n to sort an array of size n. Selection sort, bubble sort, insertion sort methods are in place sort methods.

8.6 Sort pass

The process of sorting requires traversing the given list many times. These traversals may be on the whole list or a part of it depending on the algorithm. This procedure of sequentially traversing the list or a part of it is called a pass. Each pass can be considered as a step in sorting and after the last pass we get the sorted list.

8.7 Sort Efficiency

Sorting is an important and frequent operation in many applications and so the aim is not only to get the sorted data but to get it in most efficient manner. Therefore many algorithms have been developed for sorting and to decide which one to use we need to compare them using some parameters. The choice is made using these three parameters -

1. Coding time
2. Space requirement
3. Run time or execution time

If data is in small quantity and sorting is needed only at few occasions then any simple sorting technique would be adequate. This is because in these cases a simple or less efficient technique would behave at par with the complex techniques developed to minimize run time and space requirements. So there is no point in spending lot of time in searching for best sorting algorithm or implementing a complicated technique.

We have already discussed about the space requirement of a sort, and we've seen that if data to be sorted is in large quantity then it is better to use an in place sort.

The most important parameter is the running time of the algorithm. If the amount of data to be sorted is in large quantity, then it is crucial to minimize run time by choosing an efficient sorting technique.

The two basic operations in any sorting algorithm are comparisons and record movements. The record moves or any other operations are generally a constant factor of the number of comparisons and moreover the record moves can be considerably reduced, so the run time is calculated by measuring the number of comparisons. Calculating the exact number of comparisons may not be always possible so an approximation is given by big-O notation. Thus the run time efficiency of different algorithms is expressed in terms of O notation. The efficiency of most of the sorting algorithms is in between $O(n \log n)$ and $O(n^2)$.

In some sorting algorithms, the time taken to sort depends on the order in which elements appear in the original data i.e. these algorithms behave differently when the data is already sorted or when it is in reverse order. For example if the data to be sorted is {4, 6, 8, 9, 10}, then an intelligent algorithm will immediately find out that the data is already sorted and it will not waste time in doing anything. Some sorting algorithms always take same time to sort, irrespective of the order of data.

The run time of a data sensitive algorithm may be different for different orders of data; hence we need to analyze the sorting algorithms in three different cases which are -

(i) Input data is in sorted order(ascending), e.g. { 1, 2, 3, 4, 5, 6, 7, 8 }

(ii) Input data is in random order, i.e. all the elements are dispersed in the data and there is no specific order among these elements e.g. { 4, 8, 1, 6, 5, 2, 3, 7 }. In this case it is assumed that all n! permutations of data are equally likely where n is size of data.

(iii) Input data is in reverse sorted order(descending) e.g. { 8, 7, 6, 5, 4, 3, 2, 1}.

There are numerous sorting algorithms but none of them can be termed best or most efficient, each algorithm has its advantages and disadvantages. The choice of a sorting algorithm depends on the specific situation. For example if we know in advance that our data is almost sorted then it would be useful to use an algorithm which can identify this order. The size of data is also considered while deciding which algorithm to choose. The amount of space available also determines our choice of algorithm. If we have no extra space then we have to go for in place algorithms. So we can see that the implementation of particular sorting technique depends not only on the order of that technique but also on the situation and type of data. Now after this introduction of sorting we are ready to study various sorting algorithms and their analysis.

8.8 Selection Sort

Suppose that you are given some numbers and asked to arrange them in ascending order. The most intuitive way to do this would be to find the smallest number and put in the first place and then find the second smallest number and put it in the second place and so on. This is the simple technique on which selection sort is based. It is named so because in each pass it selects the smallest element and keeps it in its exact place.

Suppose we have n elements stored in an array `arr`. First we will search the smallest element from `arr[0].....arr[n-1]` and exchange it with `arr[0]`. This will place the smallest element of list at 0th position of array. Now we will search smallest element from remaining elements `arr[1].....arr[n-1]` and exchange it with `arr[1]`. This will place the second smallest element of the list at 1st position of array. This process continues till the whole array is sorted. The whole process is as-

Pass 1 :
1. Search the smallest element from arr[0].....arr[n-1].
2. Exchange this element with arr[0].

Result : arr[0] is sorted.

Pass 2 :
1. Search the smallest element from arr[1].....arr[n-1].
2. Exchange this element with arr[1].

Result : arr[0], arr[1] are sorted.

.....

.....

Pass n-1 :
1. Search the smallest element from arr[n-2] and arr[n-1].
2. Exchange this element with arr[n-2].

Result : arr[0], arr[1].....arr[n-2] are sorted.

Now all the elements except the last one have been put in their proper place. The remaining last element `arr[n-1]` will definitely be the largest of all and so it is automatically at its proper place. So we need only n-1 passes to sort the array. Let us take a list of elements in unsorted order and sort it by applying selection sort.

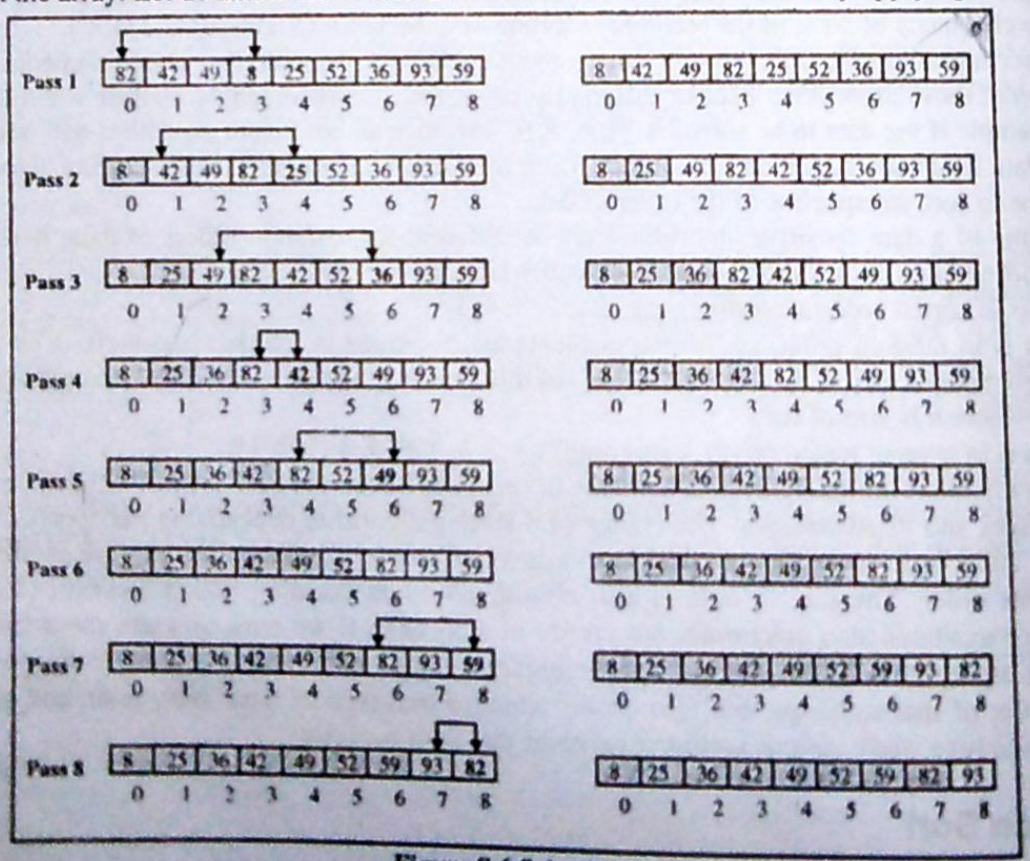


Figure 8.6 Selection Sort

In the first pass, 8 is the smal
i.e. 82. In the second pass, 25 is
42. Similarly other passes also pr
place.

```
/*P8.1 Program of sorting us
#include <stdio.h>
#define MAX 100
main()
{
        int arr[MAX],i,j,n,t
        printf("Enter the nu
        scanf("%d",&n);
        for(i=0; i<n; i++)
        {
                printf("Ente
                scanf("%d",
        }
        /*Selection sort*/
        for(i=0; i<n-1; i++)
        {
                /*Find the i
                min = i;
                for(j=i+1; j
                {
                        if(a

                }
                if(i!=min)
                {
                        temp
                        arr[
                        arr[

                }
        }
        printf("Sorted list
        for(i=0; i<n; i++)
                printf("%d
        printf("\n");
}/*End of main()*/
```

Each iteration of outer for l
exchange `arr[i]` with the sma
index of the smallest element
`arr[min]` is compared with ea
a smaller element, its index is a

After finding the smallest el
a condition to avoid swapping
proper place. In the pass 6 of fi

8.8.1 Analysis of Sele

In selection sort, the number o
sensitive. The number of com
random order. In first pass,
comparisons, in second pass
comparisons. In the last pass,
The total number of compariso

In the first pass, 8 is the smallest element among arr[0].....arr[8], so it is exchanged with arr[0] i.e. 82. In the second pass, 25 is the smallest among arr[1].....arr[8] so it is exchanged with arr[1] i.e. 42. Similarly other passes also proceed. The shaded portion shows the elements that have been put in their final place.

```
/*p8.1 Program of sorting using selection sort*/
#include <stdio.h>
#define MAX 100
main()
{
        int arr[MAX],i,j,n,temp,min;
        printf("Enter the number of elements : ");
        scanf("%d",&n);
        for(i=0; i<n; i++)
        {
                printf("Enter element %d : ",i+1);
                scanf("%d", &arr[i]);
        }
        /*Selection sort*/
        for(i=0; i<n-1; i++)
        {
                /*Find the index of smallest element*/
                min = i;
                for(j=i+1; j<n; j++)
                {
                        if(arr[min] > arr[j])
                                min = j ;
                }
                if(i!=min)
                {
                        temp = arr[i];
                        arr[i] = arr[min];
                        arr[min] = temp ;
                }
        }
        printf("Sorted list is : \n");
        for(i=0; i<n; i++)
                printf("%d ", arr[i]);
        printf("\n");
}/*End of main()*/
```

Each iteration of outer for loop corresponds to a single pass. In each iteration of outer for loop we have to exchange arr[i] with the smallest element among arr[i]...arr[n-1]. The inner for loop is used to find the index of the smallest element and it is stored in min. Initially variable min is initialized with i. After this, arr[min] is compared with each of the elements arr[i+1], arr[i+2].....arr[n-1] and whenever we get a smaller element, its index is assigned to min.

After finding the smallest element, it is exchanged with arr[i]. We have preceded this swap operation with a condition to avoid swapping of an element with itself. This situation arises when an element is already in its proper place. In the pass 6 of figure 8.6, arr[5] has to be swapped with arr[5] which is obviously redundant.

8.8.1 Analysis of Selection Sort

In selection sort, the number of comparisons does not depend on the order of data i.e. selection sort is not data sensitive. The number of comparisons performed is same whether input data is sorted, reverse sorted or in random order. In first pass, arr[0] is compared with arr[1].....arr[n-1] so there will be n-1 comparisons, in second pass arr[1] is compared with arr[2].....arr[n-1] so there will be n-2 comparisons. In the last pass, arr[n-2] is compared with arr[n-1] so there will be only one comparison. The total number of comparisons will be-

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

So the efficiency of selection sort is $O(n^2)$ in all the three cases.

Selection sort is simple to implement and requires only one temporary variable for swapping elements. The main advantage of selection sort is that data movement is very less. If an element is at its proper place then it will not be moved at all, so if many elements are at proper place i.e. the list is almost sorted then there will be very little data movement. Thus we see that the number of swaps depends on the order of data, and it can never be more than `n-1`. The swaps are very less as compared to insertion and bubble sorts. If the records are large then cost of moving data is more than the cost of comparisons, so if the records are large it is better to use selection sort.

Selection sort is not a stable sort. It requires only one temporary variable so it is an in-place sort, space complexity is O(1).

8.9 Bubble Sort

Bubble sort proceeds by scanning the list and exchanging the adjacent elements if they are out of order with respect to each other. It compares each element with its adjacent element and swaps them if they are not in order i.e. `arr[i]` will be compared with `arr[i+1]` and if `arr[i]>arr[i+1]` then they will be swapped.

In selection sort we searched for the smallest element and then performed the swap, while here swap will be performed as soon as we find two adjacent elements out of order. So in selection sort there was only one swap in a pass while in bubble sort there may be many swaps in a single pass. Hence bubble sort is not an efficient sorting technique but it is simple and easy to implement.

After first pass the largest element will be at its proper position in the array i.e. $(n-1)^{th}$ position, after second pass the second largest element will be placed at its proper position i.e. $(n-2)^{th}$ position. Similarly after each pass the next larger elements will be moved to the end of the list and placed at their proper positions. If there are n elements, then only `n-1` passes are required to sort the array and the procedure is as-

Pass 1 :
Compare arr[0] and arr[1], If arr[0] > arr[1] then exchange them,
Compare arr[1] and arr[2], If arr[1] > arr[2] then exchange them,
Compare arr[2] and arr[3], If arr[2] > arr[3] then exchange them,
.....
.....
Compare arr[n-2] and arr[n-1], If arr[n-2] > arr[n-1] then exchange them.
Result : Largest element is placed at $(n-1)^{th}$ position
arr[n-1] is sorted.

Pass 2 :
Compare arr[0] and arr[1], If arr[0] > arr[1] then exchange them,
Compare arr[1] and arr[2], If arr[1] > arr[2] then exchange them,
Compare arr[2] and arr[3], If arr[2] > arr[3] then exchange them,
.....
Compare arr[n-3] and arr[n-2], If arr[n-3] > arr[n-2] then exchange them.
Result : Second largest element is placed at $(n-2)^{th}$ position
arr[n-2], arr[n-1] are sorted.
.....
.....

Pass n-2 :
Compare arr[0] and arr[1], If arr[0] >arr[1] then exchange them,
Compare arr[1] and arr[2], If arr[1] >arr[2] then exchange them.
Result : arr[2],.....arr[n-2], arr[n-1] are sorted.

Pass n-1 :
Compare arr[0] and arr[1], If arr[0] >arr[1] then exchange them.

for swapping elements. The
t is at its proper place then it
nost sorted then there will be
rder of data, and it can never
sorts. If the records are large
s are large it is better to use

it is an in-place sort, space

if they are out of order with
waps them if they are not in
n they will be swapped.
swap, while here swap will be
sort there was only one swap
bubble sort is not an efficient

y i.e. $(n-1)^{th}$ position, after
$(n-2)^{th}$ position. Similarly
aced at their proper positions.
procedure is as-

Result : arr[1], arr[2],.....arr[n-2], arr[n-1] are sorted.

In the second pass, comparisons will be done only up to $(n-2)^{th}$ position i.e. the last comparison is done between `arr[n-3]` and `arr[n-2]`, because the largest element has already been placed at its proper position i.e. position `(n-1)`. In the third pass, the last comparison is done between `arr[n-4]` and `arr[n-3]`, because the second largest element has already been placed at its proper position i.e. position `(n-2)`. In the last pass there is only one comparison which is in between `arr[0]` and `arr[1]`. The remaining element `arr[0]` will definitely be the smallest element, so the whole list is sorted after `n-1` passes.

Let us take an array in unsorted order and sort it by applying bubble sort.

40 20 50 60 30 10

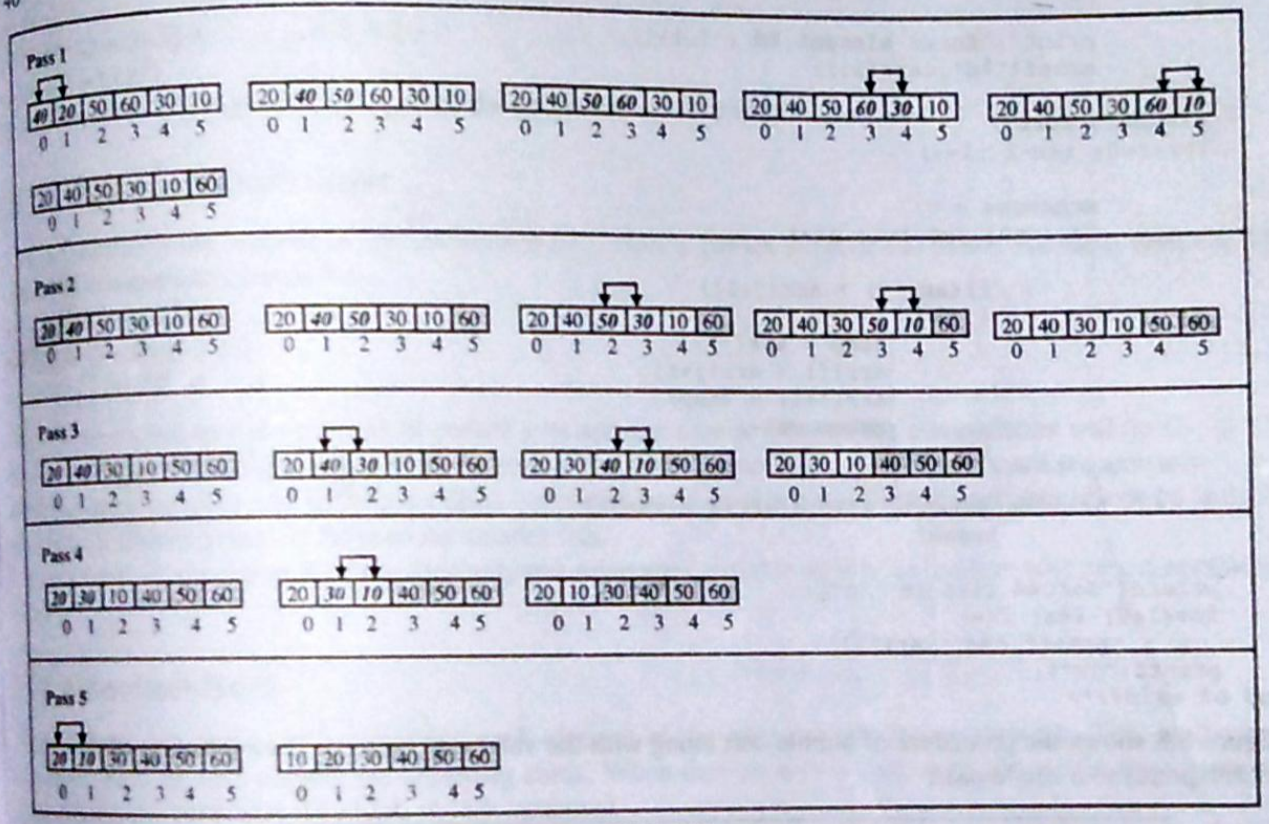


Figure 8.7 Bubble sort

Let us see what happens in the first pass. First `arr[0]` is compared with `arr[1]`, since 40>20 they are swapped. Now `arr[1]` is compared with `arr[2]`, since 40<50 they are not swapped. Now `arr[2]` is compared with `arr[3]`, since 50<60 they are not swapped. Now `arr[3]` is compared with `arr[4]`, since 60>30 they are swapped. Now `arr[4]` is compared with `arr[5]`, since 60>10 they are swapped. At the end of this pass the largest element 60 is placed at the last position. The elements which are being compared are shown in italics and the elements which have been placed in proper place are shaded.

Sometimes it is possible that a list of n elements becomes sorted in less than `n-1` passes. For example consider this list - 40 20 10 30 60 50

After first pass the list is - 20 10 30 40 50 60

After second pass the list is - 10 20 30 40 50 60

The list of 6 elements becomes sorted in only 2 passes. Hence other passes are unnecessary and there is no need to proceed further. Now the question is how we will be able to know that the list has become sorted. If no swaps occur in a pass it means that the list is sorted. For example in the above case there will be no swaps in the

third pass. We can take a variable that keeps record of the number of swaps in a pass and if no swaps occur then we can terminate our procedure.

```
/*P8.2 Program of sorting using bubble sort*/
#include <stdio.h>
#define MAX 100
main()
{
        int arr[MAX],i,j,temp,n,xchanges;
        printf("Enter the number of elements : ");
        scanf("%d",&n);
        for(i=0; i<n; i++)
        {
                printf("Enter element %d : ",i+1);
                scanf("%d",&arr[i]);
        }
        /*Bubble sort*/
        for(i=0; i<n-1 ;i++)
        {
                xchanges = 0;
                for(j=0; j<n-1-i; j++)
                {
                        if(arr[j] > arr[j+1])
                        {
                                temp = arr[j];
                                arr[j] = arr[j+1];
                                arr[j+1] = temp;
                                xchanges++;
                        }
                }
                if(xchanges==0) /*If list is sorted*/
                        break;
        }
        printf("Sorted list is :\n");
        for(i=0; i<n; i++)
                printf("%d ",arr[i]);
        printf("\n");
}/*End of main()*/
```

The figure 8.8 shows the procedure of bubble sort along with the values of `i` and `j`. Each iteration of outer for loop corresponds to a single pass.

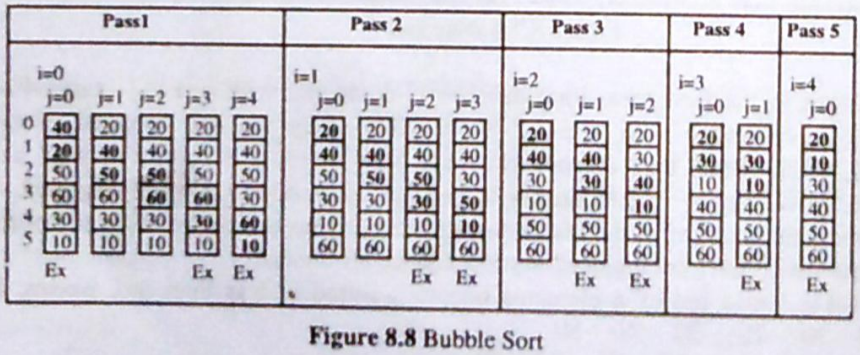


Figure 8.8 Bubble Sort

8.9.1 Analysis of Bubble Sort

The number of passes will depend on the order of data. There can be minimum 1 pass and maximum (n-1) passes.

8.9.1.1 Data in sorted order

If all the elements
loop. The number
swaps. Hence the

8.9.1.2 Data in

If the array elemer
outer for loop. In
comparisons, in th
is
$(n-1) + (n-2) + (n-$
$= n(n-1)/2 = O(n^2)$
The total number

8.9.1.3 Data in

In the i^{th} iteration
number of compar
$(n-1) + (n-2) + (n-$
$= p/2[2(n-1) + (p-$
$= (2pn - p^2 - p])/ 2$

It can be show
main advantage of
one temporary va
large lists; it shoul

Bubble sort is
is O(1).

8.10 Insertio

The insertion sor
technique used by
place among the e

We will consid
only the first ele
element from the
elements in the lis
process of insertin

Pass 1 :
Sorted part : arr[0]
Unsorted part : arr
arr[1] is inserted be
Result : arr[0] and a
Pass 2 :
Sorted part : arr[0].
Unsorted part : arr
arr[2] is inserted be
Result : arr[0], arr
Pass 3 :
Sorted part : arr[0],

If all the elements are sorted then only one pass is required and so there will be only one iteration of outer for loop. The number of comparisons will be (n-1) and all elements are in their proper place so there will be no swaps. Hence the time complexity in this case is O(n).

8.9.1.2 Data in reverse sorted order

If the array elements are in reverse order, (n-1) passes are required and so there will be (n-1) iterations of the outer for loop. In first iteration there will be (n-1) comparisons, in second iteration there will be (n-2) comparisons, in third iteration there will be (n-3) comparisons and so on. So the total number of comparisons is

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

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

The total number of swaps will be equal to the number of comparisons.

8.9.1.3 Data in random order

In the i^{th} iteration the number of comparisons is n-i. So if p passes are required to sort the data, then the total number of comparisons would be -

$(n-1) + (n-2) + (n-3) + \ldots\ldots\ldots (n-p)$

$= p/2[2(n-1) + (p-1)(-1)]$

$= (2pn - p^2 - p])/ 2$

It can be shown that the number of passes p in average case is O(n) so the comparisons will be $O(n^2)$. The main advantage of this algorithm is that it is simple and easy to implement, additional space requirement is only one temporary variable and it behaves O(n) for sorted array of elements. Bubble sort should not be used for large lists; it should generally be used for smaller lists.

Bubble sort is a stable sort. It requires only one temporary variable so it is an in place sort, space complexity is O(1).

10 Insertion Sort

The insertion sort proceeds by inserting each element at the proper place in a sorted list. This is the same technique used by card players for arranging cards. When they receive a card, they place it in the appropriate place among the cards that they have already arranged.

We will consider our list to be divided into two parts - sorted and unsorted. Initially the sorted part contains only the first element of the list and unsorted part contains the rest of the elements. In each pass, the first element from the unsorted part is taken and inserted into the sorted part at appropriate place. If there are n elements in the list, then after n-1 passes the unsorted part disappears and our whole list becomes sorted. The process of inserting each element in proper place is as-

Pass 1 :
Sorted part : arr[0]
Unsorted part : arr[1], arr[2],arr[3], ... ,arr[n-1]
arr[1] is inserted before or after arr[0].
Result : arr[0] and arr[1] are sorted.
Pass 2 :
Sorted part : arr[0], arr[1]
Unsorted part : arr[2],arr[3], ... ,arr[n-1]
arr[2] is inserted before arr[0], or in between arr[0] and arr[1] or after arr[1].
Result : arr[0], arr[1] and arr[2] are sorted.
Pass 3 :
Sorted part : arr[0], arr[1],arr[2]

Unsorted part : arr[3], arr[4],..... ,arr[n-1]
arr[3] is inserted at its proper place among arr[0], arr[1], arr[2]
Result : arr[0], arr[1], arr[2], arr[3] are sorted.
.....
.....

Pass n-1 :
Sorted part : arr[0], arr[1], ...arr[n-2]
Unsorted part : arr[n-1]
arr[n-1] is inserted at its proper place among arr[0], arr[1], ...arr[n-2]
Result : arr[0], arr[1],arr[3],.....arr[n-1] are sorted.

To insert an element in the sorted part we need a vacant position, and this space is created by moving all the larger elements one position to the right. Now let us take a list of elements in unsorted order and sort them by applying insertion sort.

82 42 49 8 25 52 36 93 59

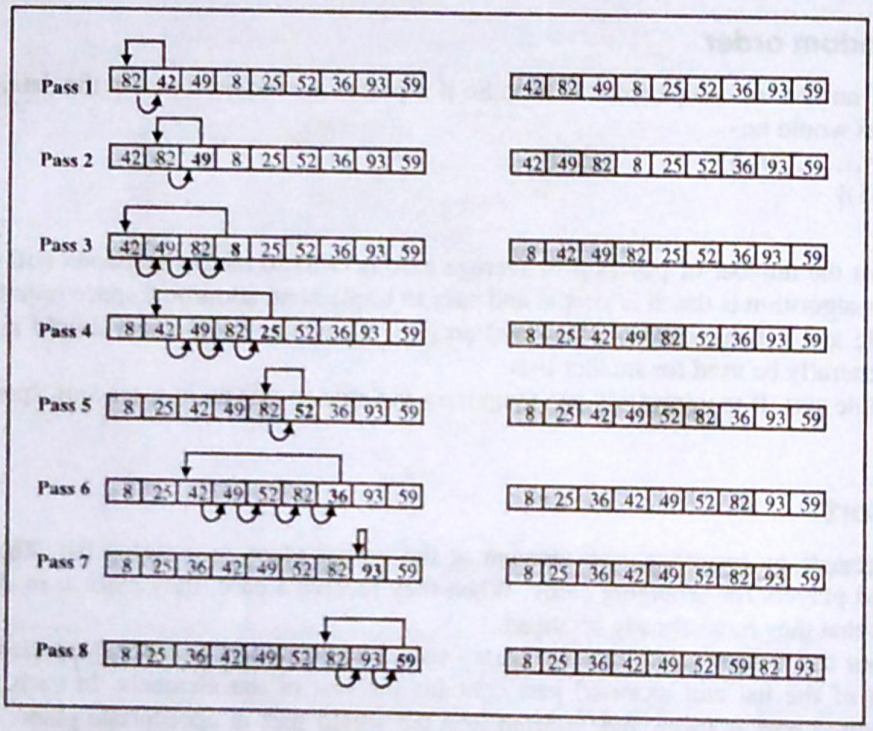


Figure 8.9 Insertion Sort

The shaded portion shows the sorted part of the list and the arrows show the movement of elements. To insert an element we have to scan the sorted part and search for the appropriate place where this element can be inserted. We will use sequential search for this purpose. We also need a vacant position for inserting this element, and for this all the larger elements are moved one position right. For example in pass 6 the elements 42, 49, 52, 82 are moved one position right to make space for insertion of 36.

If we start our searching from the end of the sorted part instead of its beginning then we can simultaneously shift the elements also.

```
/*P8.3 Program of sorting using insertion sort*/
#include<stdio.h>
#define MAX 100
main()
{
        int arr[MAX],i,j,k,n;
```

```
    printf("Enter the number of elements : ");
    scanf("%d",&n);
    for(i=0; i<n; i++)
    {
            printf("Enter element %d : ",i+1);
            scanf("%d", &arr[i]);
    }
    /*Insertion sort*/
    for(i=1; i<n; i++)
    {
            k=arr[i]; /*k is to be inserted at proper place*/
            for(j=i-1; j>=0 && k<arr[j]; j--)
                    arr[j+1]=arr[j];
            arr[j+1]=k;
    }
    printf("Sorted list is :\n");
    for(i=0; i<n; i++)
            printf("%d ",arr[i]);
    printf("\n");
}/*End of main()*/
```

In each iteration of for loop, the first element of the unsorted part is inserted into the sorted part. The element to be inserted (`arr[i]`) is stored in the variable `k`. In the inner for loop, the sorted part is scanned to find the exact location for the insertion of the element `arr[i]`. The search starts from the end of the sorted part so variable `j` is initialized to `i-1`. The search stops when we either reach the beginning of the sorted part or we get an element less than `k`. Inside the inner for loop, the elements are moved right one position, and obviously these are elements which are greater than `k`. At the end `k` is inserted at its proper place.

8.10.1 Analysis of Insertion sort

The outer for loop will always have `n-1` iterations. The iterations of the inner for loop will vary according to the data. For each iteration of outer for loop, the inner for loop executes 0 to `i` times, i.e. the inner for loop can be executed minimum 0 times and maximum `i` times.

arr[i] has to be put in any of these i+1 positions 0,1,2,3……., i-1, i

- If arr[i] is already at proper place i.e. position i is the proper place for arr[i]
  1 comparison, arr[i] compared with arr[i-1]
  No move inside inner for loop and 2 moves for loading and unloading k
- If arr[i] has to be placed at position i-1
  2 comparisons, arr[i] compared with arr[i-1], arr[i-2]
  1 move inside inner for loop and 2 moves for loading and unloading k
- If arr[i] has to be placed at position i-2 ,
  3 comparisons, arr[i] compared with arr[i-1], arr[i-2],arr[i-3]
  2 moves inside inner for loop and 2 moves for loading and unloading k

  .....
  .....

- If arr[i] has to be placed at position 2
  i-1 comparisons, arr[i] compared with arr[i-1], arr[i-2],arr[i-3],……..,arr[1]
  i-2 moves inside inner for loop and 2 moves for loading and unloading k
- If arr[i] has to be placed at position 1,
  i comparisons, arr[i] compared with arr[i-1], arr[i-2],arr[i-3],……..,arr[1],arr[0]
  i-1 moves inside inner for loop and 2 moves for loading and unloading k
- If arr[i] has to be placed at position 0
  i comparisons, arr[i] compared with arr[i-1], arr[i-2],arr[i-3],……..,arr[1],arr[0]
  i moves inside inner for loop and 2 moves for loading and unloading k

8.10.1.1 Data in sorted order

The best case occurs if the list is in sorted order. In each iteration of outer for loop, `arr[i]` is always in proper place. We have seen that if `arr[i]` is at the proper place then there is only one comparison. So there will be only one comparison in each iteration of the outer for loop, and the outer for loop always iterates `n-1` times so there will be total `n-1` comparisons in all, which is O(n).

The body of inner for loop will never be entered, so in each iteration of outer for loop there will be only 2 moves and hence the total number of moves will be `2(n-1)`. All these moves are unnecessary.

8.10.1.2 Data in reverse sorted order

The worst case occurs when the list is sorted in reverse order. In any i^{th} iteration of outer for loop, the element `arr[i]` will be less than each of these elements `arr[0]`, `arr[1]`, ...`arr[i-1]`, so `arr[i]` will always be inserted at 0^{th} position. We have seen above that if the element `arr[i]` has to be placed at 0^{th} position then there will be `i` comparisons. So in first iteration of outer for loop, there will be 1 comparison, in second iteration of outer for loop there will be 2 comparisons and in $(n-1)^{th}$ iteration of outer for loop there will be n-1 comparisons. So the total number of comparisons will be-

$$\sum_{i=1}^{n-1} i = 1 + 2 + 3 + \ldots\ldots + (n-1) = n(n-1)/2 = O(n^2)$$

If `arr[i]` is to be placed at 0^{th} position then the number of moves in i^{th} iteration of for loop is `i+2`, the total number of moves will be-

$$\sum_{i=1}^{n-1} i+2 = n(n-1)/2 + 2(n-1) = O(n^2)$$

There is a variation in the above two cases so now let us analyze the efficiency of insertion sort when data is in random order.

8.10.1.3 Data in random order

Here we assume equal probability of occupying all array locations.

Average number of comparisons in i^{th} iteration of outer for loop

$$\frac{1+2+3+\ldots+(i-1)+i+i}{i+1}$$

$$= [\, i(i+1)] \,/\, 2(i+1) + i/(i+1)$$

$$= i/2 + 1 - 1/(i+1)$$

Total number of comparisons

$$= \sum_{i=1}^{n-1} i/2 + \sum_{i=1}^{n-1} 1 - \sum_{i=1}^{n-1} 1/(i+1)$$

$$= n(n-1)/4 + (n-1) - \sum_{i=1}^{n-1} 1/(i+1)$$

$$= n(n-1)/4 + n - \sum_{j=1}^{n} 1/j$$

$$= n(n-1)/4 + n - \log_e n$$

$$= O(n^2)$$

Average number of moves in i^{th} iteration of outer for loop

$$\frac{0+1+2+\ldots+(i-1)+i}{(i+1)}$$

$= [i(i+1)] / [2(i+1)]$

$= i/2$

Total number of moves

$$= \sum_{i=1}^{n-1} i/2 = n(n-1)/4$$

$= O(n^2)$

The advantage of insertion sort is its simplicity, and it is very efficient when number of elements to be sorted are very less, because for smaller file size n the difference between $O(n^2)$ and $O(n \log n)$ is very less and $O(n \log n)$ generally requires complex sorting technique. Insertion sort is also efficient for lists that are almost sorted.

We can simplify the inner for loop, or reduce a condition in inner for loop, by taking a sentinel value in `arr[0]`. This value is taken to be smaller than the smallest value possible. So we can avoid the test `j>=0`. This condition is true only when the element to be inserted is the smallest and has to be inserted in the beginning of the array. The elements to be sorted can be stored in `arr[1]` to `arr[n]` and a sentinel value in `arr[0]`. So now only single test is sufficient and this will simplify the inner loop.

We can reduce the number of comparisons by using binary search for finding the proper place of insertion. But still the elements have to be shifted right one position which is $O(n^2)$. So the use of binary search would not improve the efficiency of the insertion sort.

A disadvantage of this sorting is the number of movements. The elements of the sorted part also have to move to make place for another element. When the data items to be sorted are large then these movements can prove costly.

Sometimes elements are at their proper place in the list but still they are moved. For example consider this data - 75 19 34 55 12

Here 19, 34 and 55 are at their proper places so their movement is redundant.

Insertion sort is a stable sort. It requires only one temporary variable so it is an in-place sort, space complexity is O(1).

8.11 Shell Sort (Diminishing Increment Sort)

The insertion sort is not efficient because many moves are performed, and the reason for so many moves is that elements are moved only one position at a time. The elements are moved only one position because only adjacent elements are compared. If we can compare elements that are far apart then we can considerably reduce the number of moves thereby making insertion sort more efficient. This is what happens in Shell sort which is an improved version of insertion sort and was given by Donald. L. Shell in 1959. It works by first comparing elements that are far apart and then it compares closer elements, the distance between the elements to be compared reduces with every pass until the last pass where adjacent elements are compared.

Now let us see the procedure of this sorting technique. In each pass we take a number 'h' called the increment. This number decreases in subsequent passes and finally in the last pass it is always 1. In each pass we divide the list into h sublists. Each sublist is created by taking elements that are at a distance of h from each other. For example if we have an array of 17 elements and we take the increments as 5, 3, 1 then the sublists will be created as-

Pass 1: (increment = 5)
5 sublists are created
Sublist 1 : a[0], a[5], a[10], a[15]
Sublist 2 : a[1], a[6], a[11], a[16]
Sublist 3 : a[2], a[7], a[12]
Sublist 4 : a[3], a[8], a[13]

Sublist 5 : a[4], a[9], a[14]
Pass 2 : (increment = 3)
3 sublists are created
Sublist 1 : a[0], a[3], a[6], a[9], a[12], a[15]
Sublist 2 : a[1], a[4], a[7], a[10], a[13], a[16]
Sublist 3 : a[2], a[5], a[8], a[11], a[14]
Pass 3 : (increment =1)
1 sublist is created
Sublist 1 : a[0], a[1], a[2], a[3], a[4], a[5], a[6],a[7], a[8], a[9], a[10], a[11], a[12], a[13], a[14], a[15], a[16]

These sublists are separately sorted among themselves by insertion sort and at the end of the pass these sorted sublists are combined. The list that we get after each pass is partially sorted. In the last pass, increment is 1 so only one sublist is created which consists of all the elements, so insertion sort is performed on the whole list and we get our sorted list.

Since the value of increment continually decreases, this sort is also known as diminishing increment sort.Now we will take an array of 17 elements and see how they can be sorted by shell sort if the increments are taken to be 5, 3, 1. The sorting procedure is shown in figure 8.10.

In the first pass, 5 sublists are created and these sublists are sorted among themselves using insertion sort. For example the first and second sublists are { 19, 10 , 45 ,12 } and { 63, 1, 3, 56 } and after sorting they become { 10, 12, 19 , 45 } and { 1, 3, 56, 63}. Similarly other 3 sublists are also sorted and then all the five sorted sublists are combined. The list that we get after first pass is partially sorted i.e. all the elements that are at a distance of 5 from each other are sorted. This list is also called 5-sorted list and the procedure is called 5-sort. Similarly in the second pass 3-sort is performed and we get 3-sorted list. In the last pass, 1-sort is done and we get our sorted list.

Original list	19	63	2	6	7	10	1	18	9	4	45	3	5	17	16	12	56
Pass 1 : Partiton into 5 sublists	19	63	2	6	7	10	1	18	9	4	45	3	5	17	16	12	56
Sort above 5 sublists	10	1	2	6	4	12	3	5	9	7	19	56	18	17	16	45	63
Combine these 5 sorted sublists	10	1	2	6	4	12	3	5	9	7	19	56	18	17	16	45	63
Pass 2 : Partition into 3 sublists	10	1	2	6	4	12	3	5	9	7	19	56	18	17	16	45	63
Sort above 3 sublists	3	1	2	6	4	9	7	5	12	10	17	16	18	19	56	45	63
Combine these 3 sorted sublists	3	1	2	6	4	9	7	5	12	10	17	16	18	19	56	45	63
Pass 3 : Only one sublist	3	1	2	6	4	9	7	5	12	10	17	16	18	19	56	45	63
	1	2	3	4	5	6	7	9	10	12	16	17	18	19	45	56	63

Figure 8.10 Shell Sort

Initially the elements which are far apart are compared and then the elements which are closer and so on. The distance between elements being compared decreases with each pass and finally in the last pass adjacent elements are compared. This way we have improved over insertion sort because here elements can be moved long distances instead of only one place at a time.

We know that if the list is in almost sorted order then insertion sort proves to be very efficient. Another feature of insertion sort is that it is very efficient on small lists. These two aspects of insertion sort are the basis of Shell sort.

Initially when the value of increment is large, the size of sublists is small so insertion sort is fast. After each pass the elements move closer to their final positions i.e. the list becomes more nearly sorted after each pass. The value of increment decreases with each pass hence leading to larger sublists, and insertion sort is not suitable for large lists. But insertion sort works fast on these larger sublists also because of the fact that they are nearly sorted. When we reach the last pass the list becomes almost sorted i.e. the elements are very much close to their final positions, so the insertion sort on the whole list goes very rapidly. So we can see that in Shell sort the previous passes help in making the list almost sorted so that the insertion sort in the last pass is very fast.

Now let us talk about the choice of increments. There is no restriction on sequence of increments except that it should be 1 in the last pass. The increment sequence suggested by Shell originally was to take first increment equal to half the size of list and divide the increment value by 2 in each pass.

If we have a list of 17 elements then the increment sequence will be 8, 4, 2, 1. The sublists formed in this case are given below (only subscripts are shown).

Increment = 8 { 0, 8, 16 }, { 1, 9 }, { 2, 10 }, { 3, 11 }, { 4, 12 }, { 5, 13 }, {6, 14}, { 7, 15 }
Increment = 4 { 0, 4, 8, 12, 16 }, { 1, 5, 9, 13 }, { 2, 6, 10, 14 }, { 3, 7, 11, 15 }
Increment = 2 { 0, 2, 4, 6, 8, 10, 12, 14, 16 }, { 1, 3, 5, 7, 9, 11, 13,15 }
Increment = 1 { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16 }

We can see that elements in the even and odd positions will not be compared till the last pass so this increment sequence is not a good choice. We can improve this sequence by adding 1 to increment if it is even.

In general it is better not to choose increments which are multiples of each other like 1,3,6,9 or 1,2,4,8 because the elements compared at one pass will be compared again at next pass. We can achieve greater efficiency if the values of increments are relatively prime. Many increment sequences have been suggested but till now none of them has been singled out as perfect. Knuth suggested a sequence

$h_1=1$, $h_{i+1}= 3h_i+1$, stop at h_i when $h_i > (n-1)/9$.

For example if n=10000 then increments would be 1, 4, 13, 40, 121, 364, 1093, 3280.

Now we will see how we can implement shell sort. One method that naturally comes to mind is that in each pass we sort all the sublists one by one using insertion sort. We will use another simple method which will make our code look like insertion sort. We can just take all elements of the list one by one and insert each element among the elements of its sublist. Here the elements will move h positions instead of one position.

```
/*P8.4 Program of sorting using shell sort*/
#include <stdio.h>
#define MAX 100

main()
{
        int arr[MAX],i,j,k,n,incr;
        printf("Enter the number of elements : ");
        scanf("%d",&n);
        for(i=0; i<n; i++)
        {
                printf("Enter element %d : ",i+1);
                scanf("%d",&arr[i]);
        }
```

```
        printf("\nEnter maximum increment (odd value) : ");
        scanf("%d",&incr);

        /*Shell sort*/
        while(incr>=1)
        {
                for(i=incr; i<n; i++)
                {
                        k=arr[i];
                        for(j=i-incr; j>=0 && k<arr[j]; j=j-incr)
                                arr[j+incr]=arr[j];
                        arr[j+incr]=k;
                }
                incr=incr-2; /*Decrease the increment*/
        }/*End of while*/
        printf("Sorted list is :\n");
        for(i=0; i<n; i++)
                printf("%d ",arr[i]);
        printf("\n");
}/*End of main()*/
```

This program is similar to the program of insertion sort, except for a few changes.

8.11.1 Analysis of Shell Sort

The coding of Shell sort technique is simple but its analysis is quite difficult and no one has been able to analyze it mathematically. The only results that are available are based on empirical studies. The running time depends on the number of increments and their value. The empirical studies show that for a particular sequence of increments it is of $O(n(\log n)^2)$ and for another sequence it is $O(n^{1.25})$. Shell sort is not a stable sort. It is an in-place sort and space complexity is O(1).

8.12 Merge Sort

Merge sort was developed by Jon von Neumann in 1945 and it has O(n log n) performance in both worst and average case. The main task in merge sort is merging two sorted lists into a single sorted list, so let's first examine the process of merging two sorted lists.

If there are two sorted arrays, then process of combining these sorted arrays into another sorted array is called merging. A simple approach could be to place the second array after the first and then sort the whole by applying any sorting technique, but by doing this we are not utilizing the fact that both the arrays are sorted. Since both arrays are sorted, we can merge them efficiently by making only one pass through each array. Let us take two arrays `arr1` and `arr2` in sorted order, we will combine them into a third sorted array `arr3`.

arr1- 5 8 9 28 34 arr2- 4 22 25 30 33 40 42

We will take one element from each array, compare them and then take the smaller one in third array. This process will continue until the elements of one array are finished. Then the remaining elements of unfinished array are put in the third array. The whole process for merging is shown in figure 8.11. `arr3` is the merged array, `i`, `j` and `k` are variables used for subscripts of `arr1`, `arr2` and `arr3` respectively.

Initially `i`, `j` and `k` point to the beginning of the three arrays so `i=0`, `j=0`, `k=0`. The elements `arr1[i]` and `arr2[j]` are compared, and the smaller one is copied to the third array `arr3` at position `k`. The variable `k` is incremented and one of the variables `i` or `j` is incremented.


```
/*P8.5 Program of
#include<stdio.h>
#define MAX 100
void merge(int arr
main()
{
        int arr1[MA
        printf("Ent
        scanf("%d"
        printf("Ent
        for(i=0; i<
        {
                pri
                sca
        }
        printf("En
        scanf("%d"
        printf("En
        for(i=0;i<
        {
                pri
                sca
        }
        merge(arr1
        printf("\n
        for(i=0; i
                pri
        printf("\n
}
void merge(int ar
{
        int i,j,k;
```

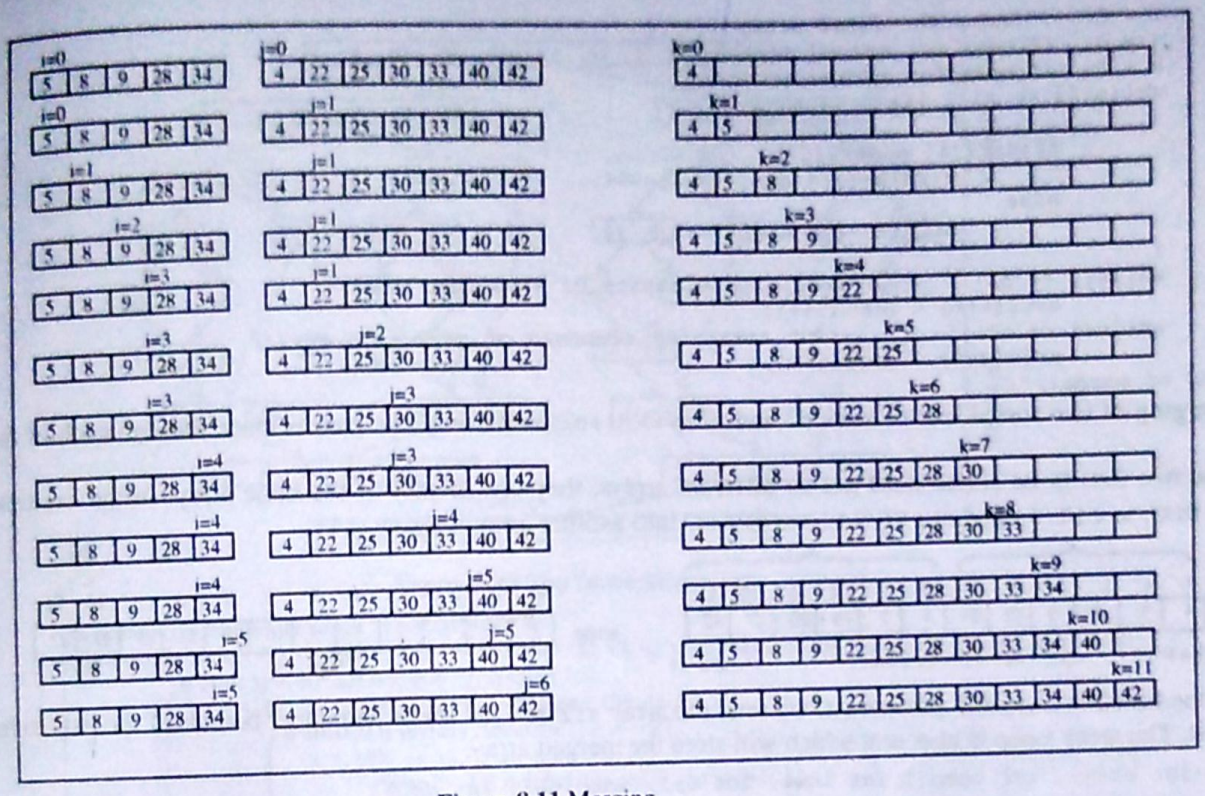


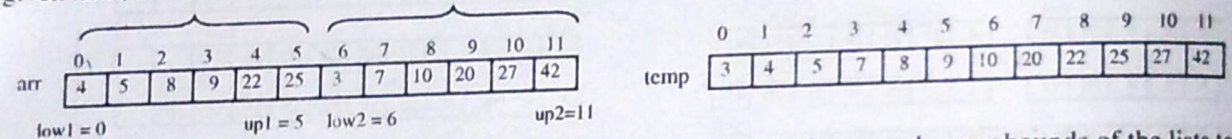
Figure 8.11 Merging

```
/*P8.5 Program of merging two sorted arrays into a third sorted array*/
#include<stdio.h>
#define MAX 100
void merge(int arr1[],int arr2[],int arr3[],int n1,int n2);
main()
{
        int arr1[MAX],arr2[MAX],arr3[2*MAX],n1,n2,i;
        printf("Enter the number of elements in array 1 : ");
        scanf("%d",&n1);
        printf("Enter all the elements in sorted order :\n");
        for(i=0; i<n1; i++)
        {
                printf("Enter element %d : ",i+1);
                scanf("%d",&arr1[i]);
        }
        printf("Enter the number of elements in array 2 : ");
        scanf("%d",&n2);
        printf("Enter all the elements in sorted order :\n");
        for(i=0;i<n2;i++)
        {
                printf("Enter element %d : ",i+1);
                scanf("%d",&arr2[i]);
        }
        merge(arr1,arr2,arr3,n1,n2);
        printf("\nMerged list : ");
        for(i=0; i<n1+n2; i++)
                printf("%d ",arr3[i]);
        printf("\n");
}
void merge(int arr1[],int arr2[],int arr3[],int n1,int n2)
{
        int i,j,k;
```

```
i = 0;  /*Index for first array*/
j = 0;  /*Index for second array*/
k = 0;  /*Index for merged array*/
while((i <= n1-1 )&&(j <= n2-1))
{
        if(arr1[i] < arr2[j])
                arr3[k++] = arr1[i++];
        else
                arr3[k++] = arr2[j++];
}
while(i <= n1-1) /*Put remaining elements of arr1 into arr3*/
        arr3[k++] = arr1[i++];
while(j <= n2-1)        /*Put remaining elements of arr2 into arr3*/
        arr3[k++] = arr2[j++];
}/*End of merge()*/
```

Merging of two sorted lists of sizes n1 and n2 is O(n1+n2) since only one pass is made through each of the lists.

The two lists to be sorted need not be different arrays, they can be part of the same array. In the example given next, `arr[0:5]` and `arr[6:11]` are merged into another array `temp[0:11]`.



In the function `merge()` given next, we send the array `arr` and the lower and upper bounds of the lists to be merged. The array `temp` is also sent which will store the merged array.

```
merge(int arr[], int temp[],int low1, int up1, int low2, int up2)
{
        int i = low1, j = low2, k = low1;
        while(i<=up1 && j<=up2)
        {
                if(arr[i] <= arr[j])
                        temp[k++] = arr[i++];
                else
                        temp[k++] = arr[j++];
        }
        while(i<=up1)
                temp[k++] = arr[i++];
        while(j<=up2)
                temp[k++] = arr[j++];
}
```

If the total number of elements in arr is n, then the performance of the above merging algorithm is O(n).

8.12.1 Top Down Merge Sort (Recursive)

This algorithm is based on the divide and conquer technique. The list is recursively divided till we get single element lists which are obviously sorted, and then the lists are merged repeatedly to get a single sorted list. The procedure for top down merge sort is-

1. Divide the list into two sublists of almost equal size.
2. Sort the left sublist recursively using merge sort.
3. Sort the right sublist recursively using merge sort.
4. Merge the two sorted sublists.

Terminating condition for recursion is when the sublist formed contains only one element. If the list contains odd number of elements then we assume that the left half is bigger. Let us take a list of numbers and sort it through merge sort. List is - 8 5 89 30 42 92 64 4 21 56 3