Chapter - 08 : "Hashing, Sorting & Searching"

HASHING

The searching techniques linear search and binary search are based on the comparison of keys. We need such searching techniques in which there are no unnecessary comparisons of keys. We need search techniques in which there are no unnecessary comparisons. Therefore, A searching technique called hashing or hash addressing is used to compute the location of the desired record in order to retrieve the desired record in a single access. This avoid the unnecessary comparison. In this method, the location of the desired record present in the search table depends only on the given key but not on other keys. For example, if we have a table of n students records each of which is defined by the roll number key. This roll number key takes value from 1 to n inclusive. If the roll number key takes values from 1 to n inclusive. If the roll number is used as an index into the student table, we can directly find the information of student in question. Therefore, array can be used to organize records in such a search table.

Hash Table

A hash table is a data structure where we store a key value after applying the hash function, it is arranged in the form of an array that is addressed via a hash function. The hash table is divided into a number of buckets and each bucket is in turn capable of storing a number of records. Thus we can say that a bucket has number of slots and each slot is capable of holding one record.

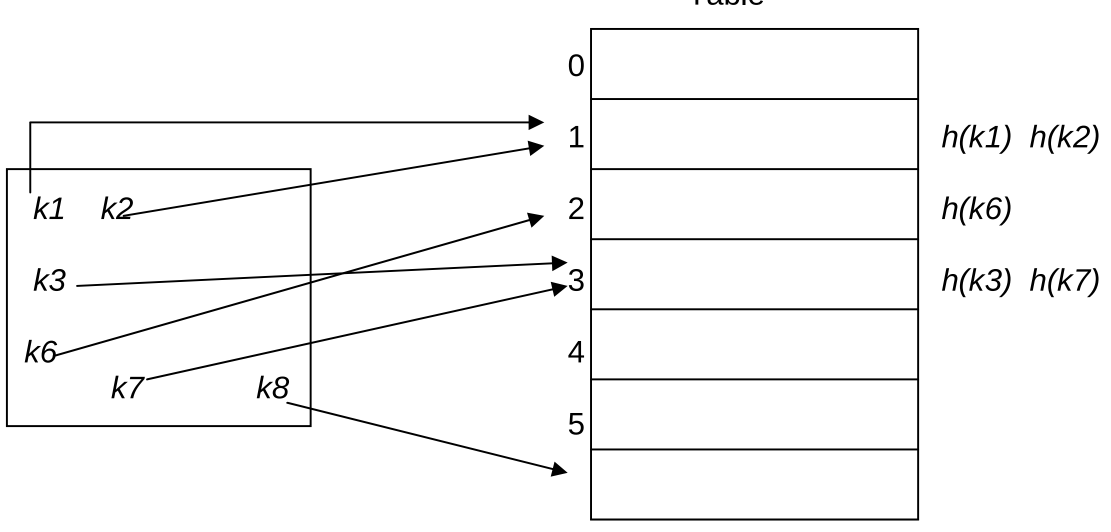


Fig (1)

The time required to locate any element in the hash table is 0 (1). It is constant and it is not depend on the number of data elements stored in the table. Now question is how we map the number of keys to a particular location in the hash table i.e., *h(k).* It is computed using the hash function.

HASH FUNCTION

The basic idea in **hashing** is the transformation of a key into the corresponding location in the hash table. This is done by a hash function. A **hash function** can be defined as a function that take key as input and transforms it into a hash index. It is usually denoted as H.

$$H : K \rightarrow M$$

where	H is a Hash function
	K is a set of keys.
	M is a set of memory addresses.

Sometimes, such a function H may not yield distinct values, it is possible that two different keys k1 and k2 will yield the same hash address. This situation is called **Hash collision**.

There are different types of Hash functions available. To choose the hash function H : K→M there are two things to consider. Firstly the function H should be a very easy and quick to compute. Secondly, the function H should, distribute the keys to the number of locations of hash table with less number of collisions. Different hash functions are :

(i) Division reminder method.
(ii) Mid square method.
(iii) Folding method.

<u>Division Reminder method</u> :

In **Division reminder method**, key k is divided by a number m larger than the number n of keys in k and the reminder of this division is taken as index into a hash table, i.e.,

$$h(k) = k \bmod m$$

The number m is usually chosen to be a prime number or a number without small divisors, since this frequently minimizes the number of collisions.

The above hash function will map the keys in the range 0 to *m -1* and is acceptable in C/C++. But if we want the hash address to range from 1 to m rather than from 0 to m – 1 we use the formula

$$h(k) = k \bmod m + 1$$

<u>Mid Square method</u> :

In the **Mid square method**, the key is first squared. Therefore, the hash function is defined by

$$h(k) = p$$

where p is obtained by deleting digits from both sides of k^2. To properly implement this the same position of k^2 must be used for all the keys.

Folding method :

In **Folding method**, the key, k is partitioned into a number of parts k1, k2,……, kr, where each part, except possibly the last, has the same number of digits as the required address. Then the parts are added together, ignoring the last carry i.e.,

$$h(k) = k_1 + k_2 + \ldots\ldots + k_r$$

where the leading-digits carriers, if any are ignored.

Time complexity of various sorting algorithms

S.No.	Algorithm	Worst Case	Average Case	Best Case
1.	**Selection Sort**	$O(n^2)$	$O(n^2)$	$O(n^2)$
2.	**Bubble Sort**	$O(n^2)$	$O(n^2)$	$O(n^2)$
3.	**Insertion Sort**	$O(n^2)$	$O(n^2)$	$n - 1$
4.	**Quick Sort**	$O(n^2)$	$\log_2 n$	$\log_2 n$
5.	**Heap Sort**	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
6.	**Merge Sort**	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
7.	**Radix Sort**	$O(n^2)$	$O(n \log n)$	$O(n \log n)$

SEARCHING

Searching is an operation which finds the location of a given element in the list. The search is said to be successful or unsuccessful depending on whether the element that is to be searched is found or not. There are mainly two standard searching methods, which are commonly used –

(i) Linear Search
(ii) Binary Search.

(I) Linear Search

This is the simplest method of searching. In this method, the element to be found is sequentially searched in the list. This method can be applied to a sorted or an unsorted list.

Searching in case of a sorted list starts from 0th element and continues until the element is found or an element whose value is greater (assuming the list is sorted in ascending order) than the value being searched is reached. As against this, searching in case of unsorted list starts from the 0th element and continues until the element is found or the end of the list is reached. To understand this, consider the array shown in Fig. (2). In which we are required to search a number 57.

11	2	9	13	57	25	17	1	90	3

$\boxed{57}$

11	**2**	9	13	57	25	17	1	90	3

$\boxed{57}$

11	2	**9**	13	57	25	17	1	90	3

$\boxed{57}$

11	2	9	**13**	57	25	17	1	90	3

$\boxed{57}$

11	2	9	13	**57**	25	17	1	90	3

$\boxed{57}$

Fig (2) : Linear search in an unsorted array

The array shown in Fig. (2) consists of 10 numbers. Suppose the element that is to be searched is **57**. So **57** is compared with all the elements started with 0th element and the searching process ends either when **57** is found or the lists ends.

The performance of linear search algorithm can be measured by counting the comparisons done to find out an element. The number of comparisons is **O (n)**.

In case of sorted list, searching of element starts from the 0th element. Searching ends when the element is found or any element of the list is found to be greater than the element to be searched. This is shown in Fig. (2).

1	2	3	9	11	13	17	25	57	90

$\boxed{57}$

1	**2**	3	9	11	13	17	25	57	90

$\boxed{57}$

1	2	**3**	9	11	13	17	25	57	90

$\boxed{57}$

1	2	3	**9**	11	13	17	25	57	90

$\boxed{57}$

1	2	3	9	**11**	13	17	25	57	90

$\boxed{57}$

1	2	3	9	11	**13**	17	25	57	90	57
1	2	3	9	11	13	**17**	25	57	90	57
1	2	3	9	11	13	17	**25**	57	90	57
1	2	3	9	11	13	17	25	**57**	90	57

Fig (3) : Linear Search in a sorted array

ALGORITHM 1 : LINEAR SEARCH

Let A[5] is an array of 5 elements and NUM is the number to be searched in array, I is the index number of each array element.

```
1.  Set F = 0.
2.  Read 5 elements of array A.
3.  Read number NUM to be searched in Array A.
4.  Repeat Step 4 for I = 0 to 4 :
        If NUM = A[ I ] then :
            Set F = 1
            BREAK;
        [End of If Structure]
    [End of Step 4 loop].
5.  If F = 0 then :
      Write 'Number is not present in array'
    Else
      Write 'Number is present in array at location - ',I+1
    [End of If Else Structure].
6.  Exit
```

Program 1 : WAP to search a number in an array using Linear Search method.

```
#include<stdio.h>
#include<conio.h>
void main()
{
clrscr();
int A[5],num,i,f=0;
printf("Enter 5 elements of an array :\n");
for(i=0;i<5;i++)
 scanf("%d",&A[i]);

printf("\nEnter number to be searched in array : ");
scanf("%d",&num);

for(i=0;i<5;i++)
{
 if(num==A[i])
 {
  f=1;
  break;
 }
}
if(f==0)
   printf("\n%d is not present in array",num);
else
   printf("\n%d is present in array at location-%d",num,i+1);
getch();
}
```

(II) Binary Search

Binary search method is very fast and efficient. This method requires that the list of elements be in sorted order.

In this method, to search an element we compare it with the element present at the center of the list. If it matches then the search is successful. Otherwise, the list is divided into two halves: one from **0th** element to the center element (first half), and another from center element to the last element (second half). As a result, all the elements in first half are smaller than the center element, whereas, all the elements in second half are greater than the center element.

The searching will now proceed in either of the two halves depending upon whether the element is greater or smaller than the center element. If the element is smaller than the center element then the searching will be done in the first half, otherwise in the second half.
Same process of comparing the required element with the center element and if not found then dividing the elements into two halves is repeated for the first half or second half. This procedure is repeated till the element is found or the division of half parts gives one element. Let us understand this with the help of Fig. (4).

1	2	3	9	**11**	13	17	25	57	90	57

1	2	3	9	11	13	17	**25**	57	90	57

1	2	3	9	11	13	17	25	**57**	90	57

Fig. (4) : Binary Search

Suppose an array **arr** consists of **10** sorted numbers and **57** is element that is to be searched. The binary search method when applied to this array works as follows :

(a) **57** is compared with the element present at the center of the list (i.e. **11**). Since **57** is greater than **11**, the searching is restricted only to the second half of the array.
(b) Now **57** is compared with the center element of the second half of array (i.e. **25**). Here, again **57** is greater than **25** so the searching now proceed in the elements present between the **25** and the last element **90**.
(c) This process is repeated till **57** is found or no further division of sub-array is possible.

The maximum number of comparisons in binary search is limited to **$\log_2 n$.**

ALGORITHM 2 : BINARY SEARCH

Let A[10] is an array of 10 elements and NUM is the number to be searched in array, L represents the lower bound, U represents the upper bound and M represents the index number of middle element, I represents the index number of each array element.

```
1. Set F = 0, L = 0, U = 9.
2. Read 10 elements of array A in ascending order.
3. Read number NUM to be searched in Array A.
4. Repeat Step 4 while L ≤ U
       Set M = (L + U) / 2.
           If NUM > A[M] : then
              Set L = M +1
           Else If NUM < A[M] : then
              Set U = M – 1
           Else
              Set F = 1
              BREAK.
           [End of If Else If Structure]
   [End of Step 4 loop].
5. If F = 0 : then
      Write ‘Number is not present in Array’.
   Else
      Write ‘Number is present in Array at position–‘, M + 1.
   [End of If Else Structure]
6. Exit
```

Program 2 : WAP to search a number in an array using Binary search

```
#include<stdio.h>
#include<conio.h>
void main()
{
clrscr();
int A[10],num,L=0,U=9,M,f=0,i;
printf("\nEnter 10 elements of an array in ascending order : \n");
for(i=0;i<10;i++)
 scanf("%d",&A[i]);
printf("\nEnter number to be searched in array : ");
scanf("%d",&num);
while(L<=U)
{
 M=(L+U)/2;
 if(num>A[M])
   L=M+1;
 else
  if(num<A[M])
   U=M-1;
  else
  {
   f=1;
   break;
  }
}
if(f==0)
 printf("\n%d is not present in array",num);
else
  printf("\n%d is present in array at location-%d",num,M+1);
getch();
}
```

Comparison of Linear Search and Binary Search :

Consider the following set of elements : 1, 2, 3, 9, 11, 13, 17, 25, 57, 90

Suppose, we want to search 25 in the above set of numbers. Table (1) shows number of comparisons required in both the methods.

Method	Number of comparisons
Linear Search	8
Binary Search	3

Table (1) : Comparison between Linear and Binary search.

The table clearly shows that how fast a binary search algorithm works. The **advantage** of the binary search method is that, in each iteration, it reduces the number of elements to be searched from **n** to **n/2**. On the other hand, linear search method checks sequentially for every element, which makes it inefficient.

The disadvantage of binary search is that it works only on sorted lists. So when searching is to be performed on unsorted list then linear search is the only option.

■ <u>SORTING</u>

Sorting means arranging a set of data in some order. There are different methods that are used to sort the data in ascending or descending order. These methods can be divided into two categories. They are as follows :

External Sorting :

When the data to be sorted is so large that some of the data is present in the memory and some is kept in auxiliary memory (hard disk, floopy, tape etc), then external sorting methods are used. External sorting is applied to the huge amount of data that cannot be accommodate in the memory all at a time. So data from the disk is loaded into memory part by part and each part that is loaded is sorted and the sorted data is stored into some intermediate file. Finally all the sorted parts present in different intermediate file. Finally all the sorted parts present in different intermediate files are merged into one single file.

Internal Sorting :

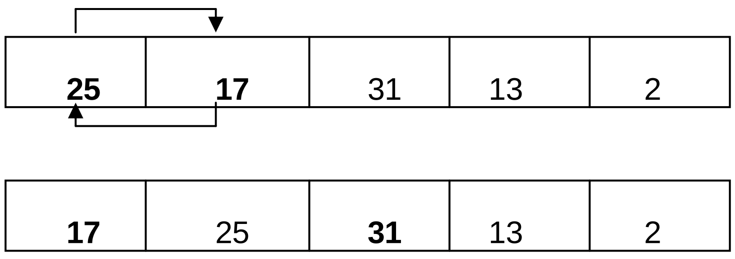
If all the data that is to be sorted can be accommodated at a time in memory then internal sorting methods are used.
There are different types of internal sorting methods. The following methods sort the data in ascending order. With a minor change, we can also sort the data in descending order. Some standard methods are as given below :

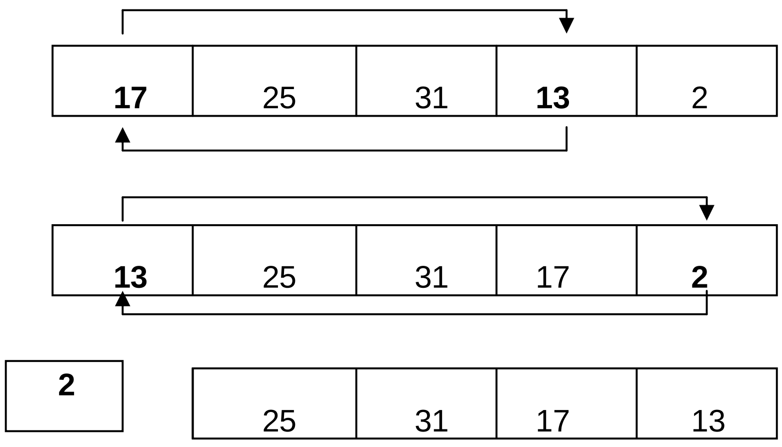
1. Selection Sort
2. Bubble Sort
3. Insertion Sort
4. Quick Sort
5. Merge Sort
6. Radix Sort
7. Heap Sort

(I) Selection Sort

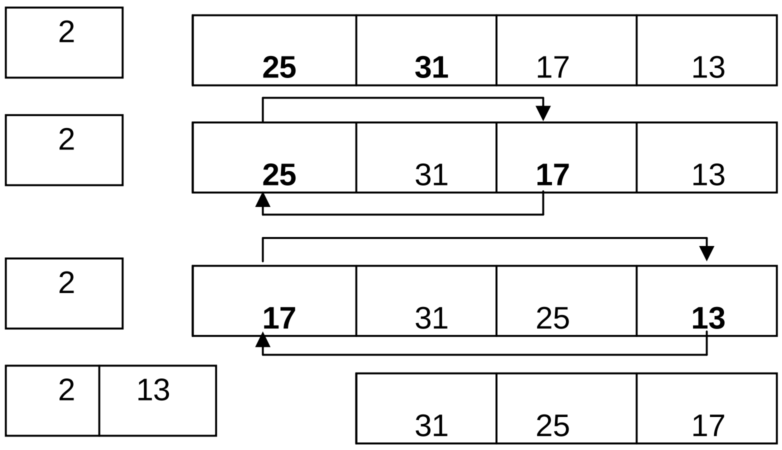
This is the simplest method of sorting. In this method, to sort the data in ascending order, the **0th** element is compared with all other elements. If the **0th** element is found to be greater than the compared element then they are interchanged. So after the first iteration, the smallest element is placed at 0th position. The same procedure is repeated for the 1st element and so on. This can be explained with the help of Fig. (5).

First Iteration

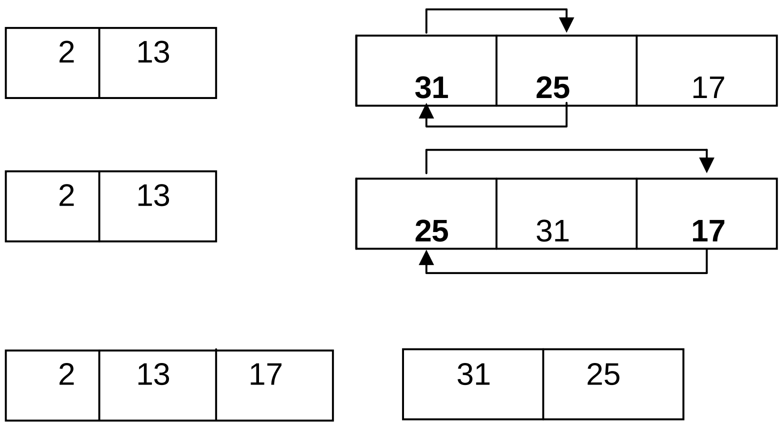




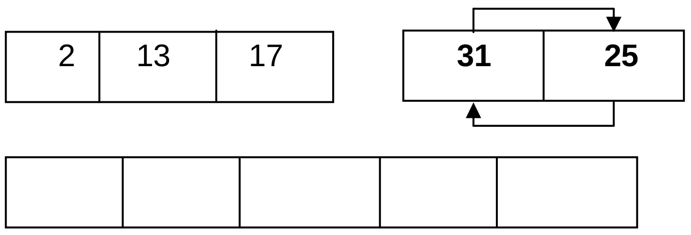
Second Iteration



Third Iteration



Fourth Iteration



2 13 17 25 31

Fig (5) : Selection Sort

Suppose an array **arr** consists of 5 numbers. The selection sort algorithm work as follows :

1. In the first iteration, the **0th** element 25 is compared with **1st** element 17 and since 25 is greater than 17, they are interchanged.
2. Now the **0th** element 17 is compared with **2nd** element 31. But 17 being less than 31, hence they are not interchanged.
3. This process is repeated till **0th** element is compared with rest of the elements. During the comparison if **0th** element is found to be greater than the compared element, then they are interchanged, otherwise not.
4. At the end of the first iteration, the **0th** element holds the smallest number.
5. Now the second iteration starts with the **1st** element 25. The above process of comparison and swapping is repeated.
6. So if there are **n** elements, then after **(n – 1)** iterations, the array is sorted.

ALGORITHM 3 : SELECTION SORT

Let A[5] is an array of 5 elements and TEMP is the variable used for swapping of array elements.

```
1.  Read 5 elements of array A.
2.  Write Original Array A.
3.  Repeat Steps 3 to 4 for I = 0 to 3.
4.  Repeat Step 4 for J = I+1 to 4.
        IF A[I] > A[J] : then              [Interchange A[I] & A[J]]
            Set TEMP = A[J].
            Set A[J] = A[I]
            Set A[I] = TEMP
        [End of IF structure].
    [End of Step 4 loop].
    [End of Step 3 loop].
5.  Write Sorted Array A.
6.  Exit
```

Program 3 : WAP to sort an array using Selection Sort.

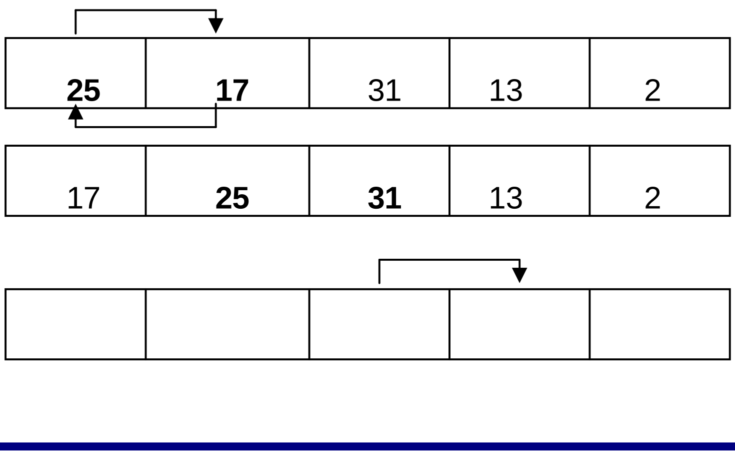
```
#include<stdio.h>
#include<conio.h>
void main()
{
clrscr();
int A[5],i,j,temp;
printf("Enter 5 elements of an array : ");
for(i=0;i<5;i++)
```

```
 scanf("%d",&A[i]);

printf("\nOriginal Array:\n");
for(i=0;i<5;i++)
  printf("%d\t",A[i]);

for(i=0;i<4;i++)
{
  for(j=i+1;j<5;j++)
  {
   if(A[i]>A[j])
   {
     temp=A[j];
     A[j]=A[i];
     A[i]=temp;
   }
  }
}

printf("\nSorted Array using Selection Sort :\n");
for(i=0;i<5;i++)
   printf("%d\t",A[i]);
getch();
}
```

(II) BUBBLE SORT

In this method, to arrange elements in ascending order, to begin with the 0th element is compared with the 1st element. If it is found to be greater than the 1st element then they are interchanged. Then the 1st element is compared with the 2nd element, if it is found to be greater, then they are interchanged. In the same way all the elements(excluding list) are compared with their element and are interchanged if required. This is the first iteration and on completing this iteration the largest element gets placed at the last position. Similarly, in the second iteration the comparisons are made till the last but one element and this time the second largest element gets placed at the second last position in the list. As a result, after all the iterations the list becomes a sorted list. This can be explained with the help of Fig. (6).

First Iteration



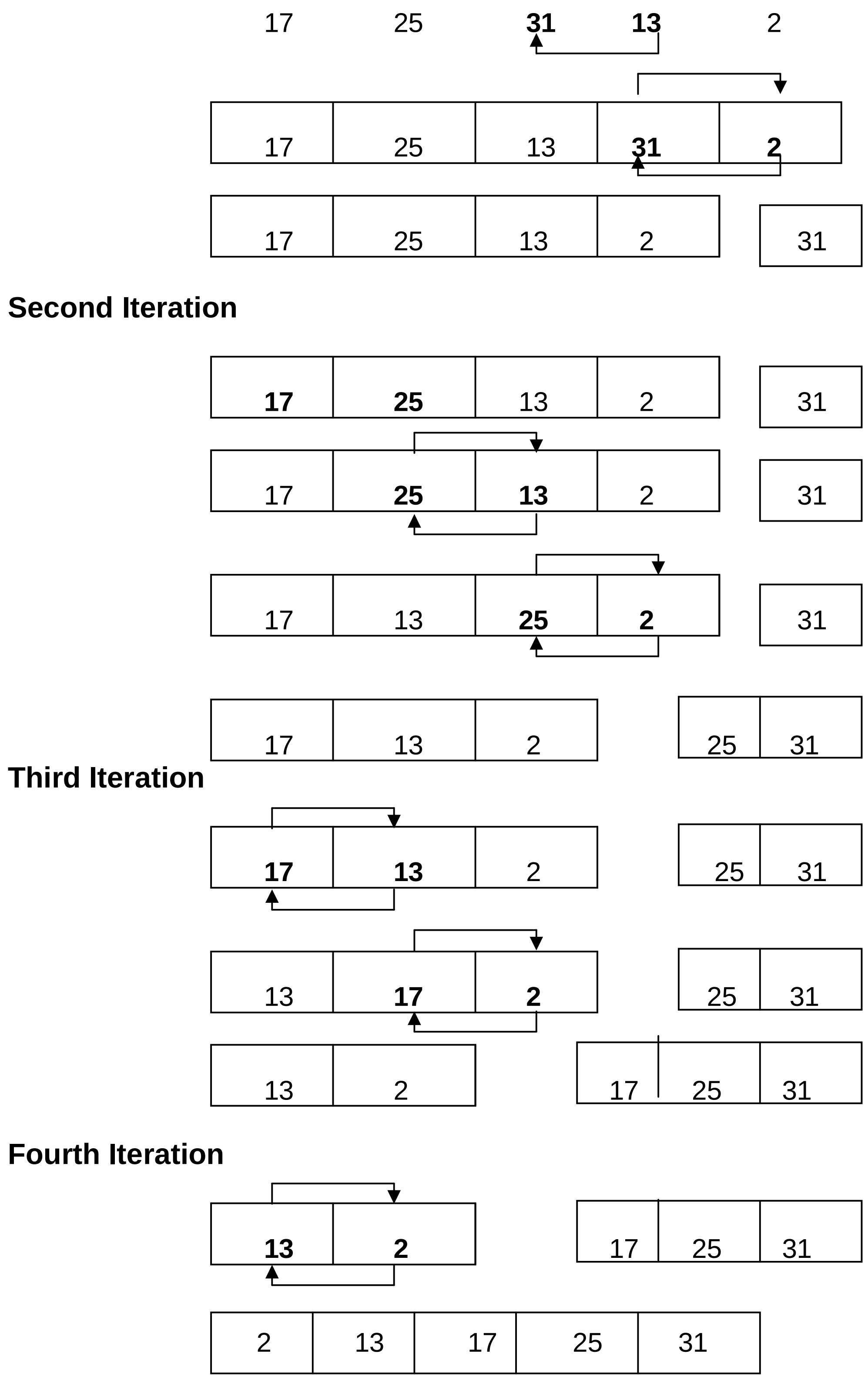


Fig (6) : Bubble Sort

Suppose an array **arr** consists of **5** numbers. The bubble sort algorithm works as follows :

(a) In the first iteration, the **0th** element 25 is compared with **1st** element 17 and since 25 is greater than 17, they are interchanged.
(b) Now the **1st** element 25 is compared with **2nd** element 31. But 25 being less than 31 they are not interchanged.
(c) This process is repeated until **(n – 2)th** element is compared with **(n – 1)th** element. During this comparison, if **(n – 2)th** element is found to be greater than the **(n – 1)th** element, then they are interchanged, otherwise not.
(d) At the end of the first iteration, the **(n – 1)th** element holds the largest number.
(e) Now the second iteration starts with the 0th element 17. The above process of comparison and interchanging is repeated but this time the last comparison is made between **(n – 3)th** and **(n – 2)th** elements.
(f) If there are n numbers of elements then **(n – 1)** iterations need to be performed.

ALGORITHM 4 : BUBBLE SORT

```
        Let A[5] is an array of 5 elements and TEMP is the variable used for swapping of array
elements.

    1. Read 5 elements of array A.
    2. Write Original Array A.
    3. Repeat Steps 3 to 4 for I = 0 to 3.
    4. Repeat Step 4 for J = 0 to 4 – I.
          IF A[J] > A[J+1] : then                [Interchange A[J] & A[J+1]
              Set TEMP = A[J+1].
              Set A[J+1] = A[J]
              Set A[J] = TEMP
          [End of IF structure].
       [End of Step 4 loop].
       [End of Step 3 loop].
    5. Write Sorted array A.
    6. Exit
```

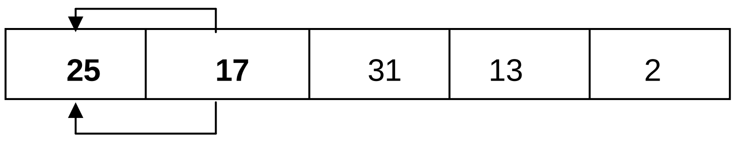
Program 4 : WAP to sort an array using Bubble Sort.

```
#include<stdio.h>
#include<conio.h>
void main()
{
clrscr();
int A[5],i,j,temp;
printf("Enter 5 elements of an array : \n");
for(i=0;i<5;i++)
 scanf("%d",&A[i]);
printf("\nArray before sorting:\n");
for(i=0;i<5;i++)
  printf("%d\t",A[i]);
for(i=0;i<4;i++)
{
  for(j=0;j<4-i;j++)
  {
   if(A[j]>A[j+1])
   {
     temp=A[j+1];
     A[j+1]=A[j];
     A[j]=temp;
   }
  }
}
printf("\nSorted Array using Bubble Sort :\n");
for(i=0;i<5;i++)
   printf("%d\t",A[i]);
getch();
}
```

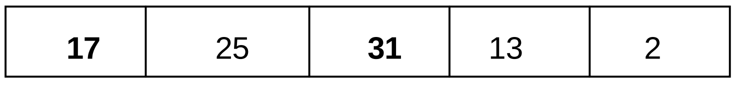
(III) INSERTION SORT

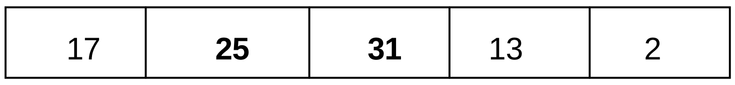
Insertion sort is implemented by inserting a particular element at the appropriate position. In this method, the first iteration starts with comparison of 1st element with the 0th element. In the second iteration, 2nd element is compared with the 0th and 1st element. In general, in every iteration an element is compared with all elements before it. During comparison, if is found that the element in question can be inserted at a suitable position then space is created for it by shifting the other element at the suitable position. This procedure is repeated for all the elements in the array. Let us understand this with the help of Fig. (7).

First Iteration

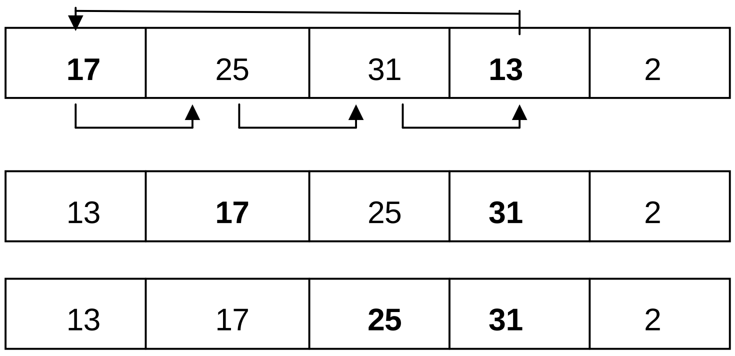


Second Iteration





Third Iteration



Fourth Iteration

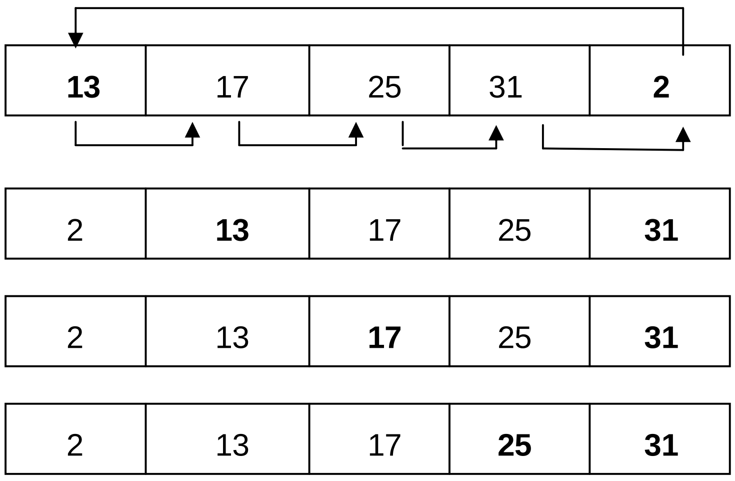


Fig (6) : Insertion Sort

Following steps explain the algorithm of insertion sort for an array **A** of **5** elements.

(a) In the first iteration, the **1st** element **17** is compared with the **0th** element **25**. Since **17** is smaller than **25**, **17** is inserted at **0th** place. The **0th** element **25** is shifted one position to the right.

(b) In the second iteration, the **2nd** element **31** and the **0th** element **17** are compared. Since **31** is greater than **17**, nothing is done. Then the **2nd** element **31** is compared with the **1st** element **25**. Again no action is taken as **25** is less than **31**.

(c) In the third iteration, the **3rd** element **13** is compared with the **0th** element **17**. Since, **13** is smaller than **17**, **13** is inserted at the **0th** place in the array and all the element from **0th** till **2nd** position are shifted to right by one position.

(d) In the fourth iteration, the **4th** element **2** is compared with the **0th** element **13**. Since, **2** is smaller than **13**, the **4th** element is inserted at the **0th** place in the array and all the elements from **0th** till **3rd** are shifted right by one position. As a result, the array now becomes a **sorted** Array.

ALGORITHM 5: INSERTION SORT

Let A [5] is an array of 5 elements and TEMP is the variable used for storing the number to be inserted at a particular position.

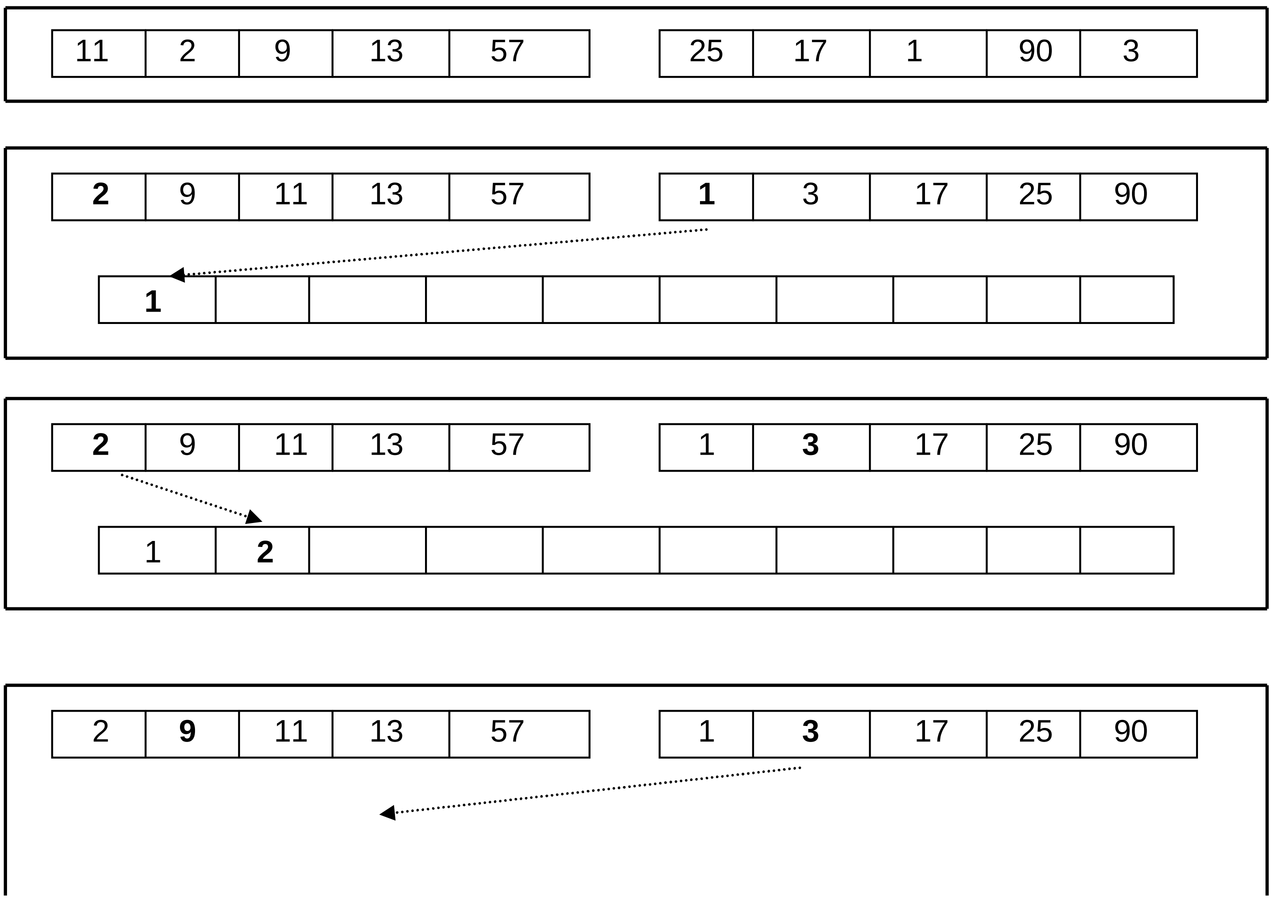
```
1. Read 5 elements of array A.
2. Write Original Array A.
3. Repeat Steps 3 to 5 for I = 1 to 4.
4. Repeat Step 4 to 5 for J = 0 to I – 1.
   IF A[J] > A[I] : then
     Set TEMP = A[I].
5. Repeat Step 5 for K = I to J + 1
           Set A[K] = A[K-1]
   [End of Step 5 loop].
   [End of Step 4 IF structure].
   [End of Step 4 loop].
   [End of Step 3 loop].
6. Write Sorted Array A.
7. Exit
```

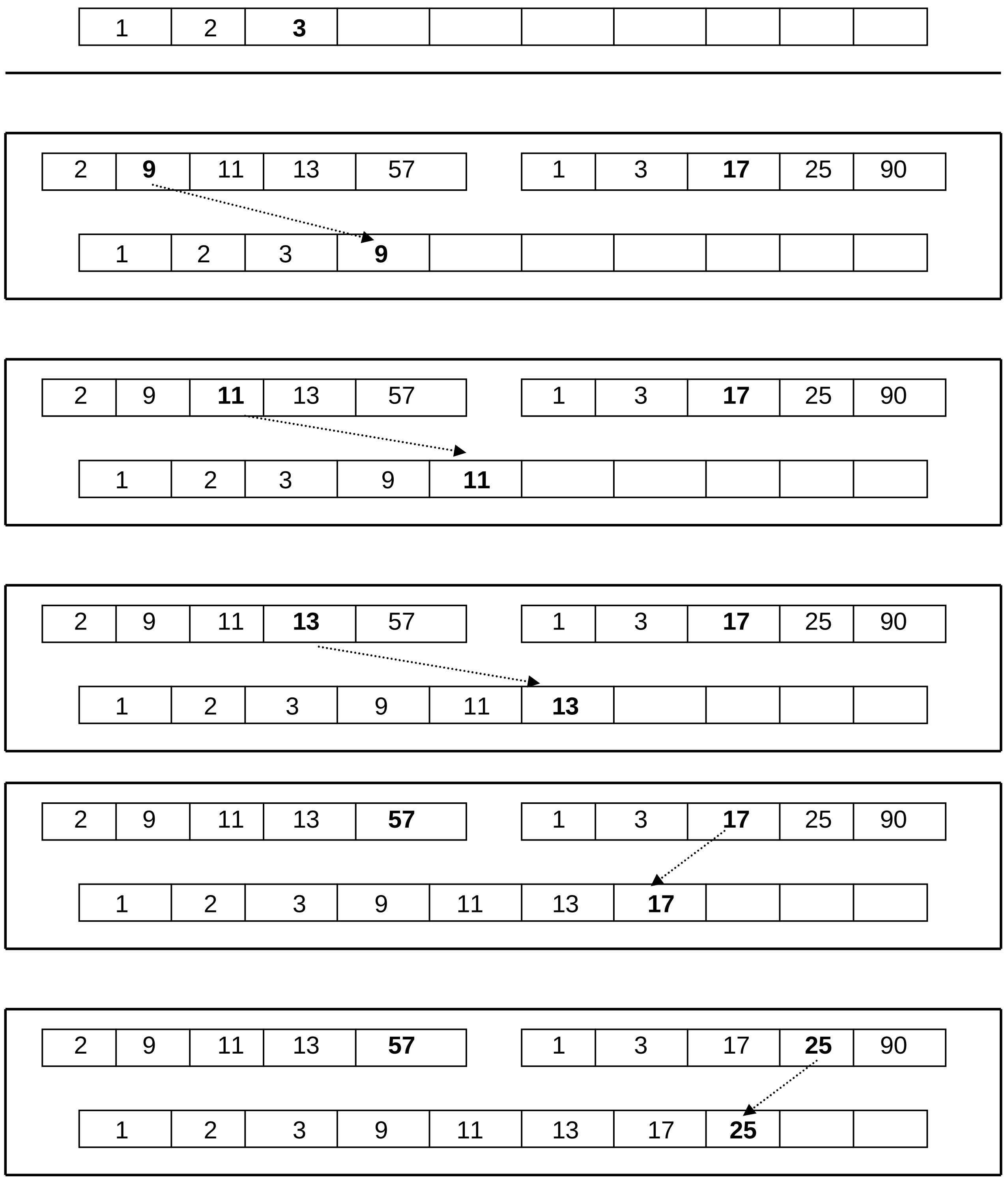
Program 5 : WAP to sort an array using Insertion Sort.

```
#include<stdio.h>
#include<conio.h>

void main ()
{
clrscr ();
int A[5],i,j,k,temp;
printf("Enter 5 elements of an array : ");
for(i=0;i<5;i++)
  scanf("%d",&A[i]);
printf("\nArray before sorting:\n");
for(i=0;i<5;i++)
  printf("%d\t",A[i]);

for(i=1;i<5;i++)
{
 for(j=0;j<i;j++)
  {
    if(A[j]>A[i])
    {
      temp=A[ i ];
       for(k=i;k>j;k- -)
       {
           A[k]=A[k-1];
       }
      A[k]=temp;
    }
  }
}
printf("\nSorted Array using Insertion sort:\n");
for(i=0;i<=4;i++)
  printf("%d\t",A[i]);
getch();
}
```

(IV) MERGE SORT

Merging means combining two sorted lists into one sorted list. For this the elements from both the sorted lists are compared. The smaller of both the elements is then sorted in the third array. The sorting is complete when all the elements from both the arrays are placed in the third array as shown in Fig. (8).




1 2 3
2 9 11 13 57
1 3 17 25 90
1 2 3 9
2 9 11 13 57
1 3 17 25 90
1 2 3 9 11
2 9 11 13 57
1 3 17 25 90
1 2 3 9 11 13
2 9 11 13 57
1 3 17 25 90
1 2 3 9 11 13 17
2 9 11 13 57
1 3 17 25 90
1 2 3 9 11 13 17 25

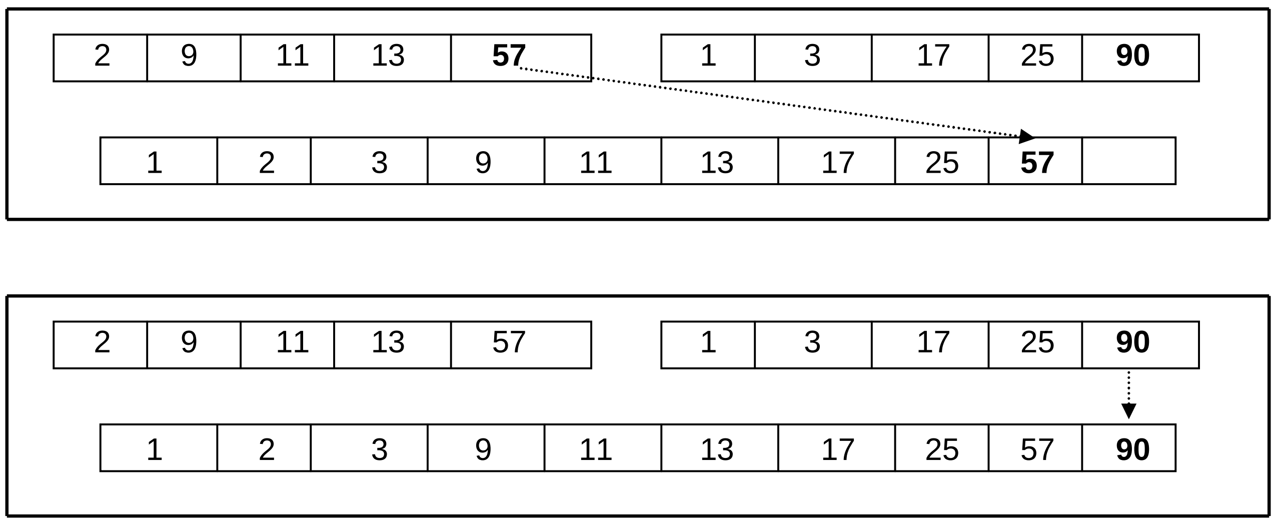


Fig. (8) : Merge Sort

Suppose Arrays **A** and **B** contain **5** elements each. Then Merge sort algorithm works as follows :

(a) The arrays **A** and **B** are sorted using any algorithm.

(b) The **0th** element from the first array, **2**, is compared with the **0th** element of second array, **1**. Since **1** is smaller than **2**, **1** is placed in the third array.

(c) Now, the **0th** element from the first array, **2**, is compared with **1st** element from the second array, **3**. Since **2** is smaller than **3**, **2** is placed in the third array.

(d) Now, the **1st** element from the first array, **9**, is compared with **1st** element from the second array, **3**. Since **3** is smaller than **9**, **3** is placed in the third array.

(e) Now, the **1st** element from the first array, **9**, is compared with **2nd** element from the second array, **17**. Since **9** is smaller than **17**, **9** is placed in the third array.

(f) The same procedure is repeated till end of one of the arrays is reached. Now, the remaining elements from the other array are placed directly into the third list as are already in sorted order.

ALGORITHM 6 : MERGE SORT

Let A[5] is first array of 5 elements, B[5] is second array of 5 elements & C[10] is the merged array of 10 elements where we store the elements of Array A and Array B.

```
1.  Read 5 elements of array A.
2.  Read 5 elements of array B.
3.  Repeat Steps 3 to 4 for I = 1 to 4.
4.  Repeat Step 4 for J = I+1 to 4.
       IF A[I] > A[J] : then                [Sort first Array A by using selection sort]
          Set TEMP = A[J].
          Set A[J] = A[I]
          Set A[I] = TEMP
       [End of IF structure].
       IF B[I] > B[J] : then                [Sort second Array B by using selection sort]
          Set TEMP = B[J].
          Set B[J] = B[I]
          Set B[I] = TEMP
       [End of IF structure].
    [End of Step 4 loop].
    [End of Step 3 loop].
5.  Set J = K = 0
6.  Repeat Step 6 for I = 0 to 9
          IF A[J]<=B[K] : then
             Set C[I] = A[J].
             Set J=J+1
          ELSE
             Set C[I]=B[K].
             Set K=K+1
          [End of IF ELSE Structure]
          IF J = 5 OR K = 5 : then
            Set I=I+1.
            BREAK.
          [End of IF Structure]
     [End of Step 6 loop]
7.  Repeat Step 7 for J<5
          Set C[I] = A[J].
          Set I = I + 1
          Set J = J +1
      [End of Step 7 loop]
8.  Repeat Step 8 for K<5
          Set C[I] = B[K].
          Set I = I + 1
          Set K = K +1
      [End of Step 8 loop]
9.  Write Merged & Sorted Array C.
10. Exit
```

Program 6 : WAP to sort an array using Merge Sort.

```
#include<stdio.h>
#include<conio.h>
void main()
{
 clrscr();
 int A[5],B[5],C[10];
 int i,j,k,temp;
 printf("\nEnter 5 elements of first array : \n");
 for(i=0;i<5;i++)
    scanf("%d",&A[i]);
 printf("\nEnter 5 elements of second array : \n");
 for(i=0;i<5;i++)
   scanf("%d",&B[i]);
 for(i=0;i<4;i++)
 {
  for(j=i+1;j<5;j++)
  {
    if(A[i]>A[j])
    {
     temp=A[i];
     A[i]=A[j];
     A[j]=temp;
    }
    if(B[i]>B[j])
    {
     temp=B[i];
     B[i]=B[j];
     B[j]=temp;
    }
  }
 }

 for(i=0,j=0,k=0;i<10;i++)
 {
   if(A[j]<=B[k])   // A[0]<=B[0]
   {
     C[i]=A[j];
     j++;
   }
   else
   {
     C[i]=B[k];
     k++;
   }
   if(j==5 || k==5)
   {
     i++;
     break;
   }
 }

 for(;j<5;)
 {
   C[i]=A[j];
   i++;
   j++;
 }

 for(;k<5;)
 {
   C[i]=B[k];
   i++;
   k++;
 }

 printf("\nSorted Array using Merge Sort : \n");
 for(i=0;i<10;i++)
     printf("%d\t",C[i]);
getch();
}
```

(V) QUICK SORT

Quick sort is a very popular sorting method. The name comes from the fact that , in general, Quick sort can sort a list of data elements significantly faster than any of the common sorting algorithms. This algorithm is based on the fact that it is faster and easier to sort two small arrays than one larger one. The basic strategy of Quick sort is to divide and conquer.

If you were given a large stack of papers bearing the names of the students to sort them by name, you might use the following approach. Pick a splitting value, say L (known as **pivot** element) and divide the stack of papers into two piles, A – L and M – Z (note that the two piles will not necessarily contain the same number of papers. Then take the first pile and sub-divide it into two piles, A – F and G – L. The A – F pile can be further broken down in A – C and D – F. This division process goes on until the piles are small enough to be easily sorted. The same process is applied to the M – Z pile. Finally, all the small sorted piles can be stacked one on top of the other to produce an ordered set of papers.

This strategy is based on recursion – on each attempt to sort the stack of papers, the pile is divided and then the same approach is used to sort each smaller piles (a smaller case).

Quick sort is also known as **Partition Exchange sort**. The quick sort procedure can be explained with the help of Fig. (9). In Fig. (9), the element that is indicated by ‘ * ’ is the pivot element and the element that is indicated by ‘ **–** ‘ is the element whose position is finalized.

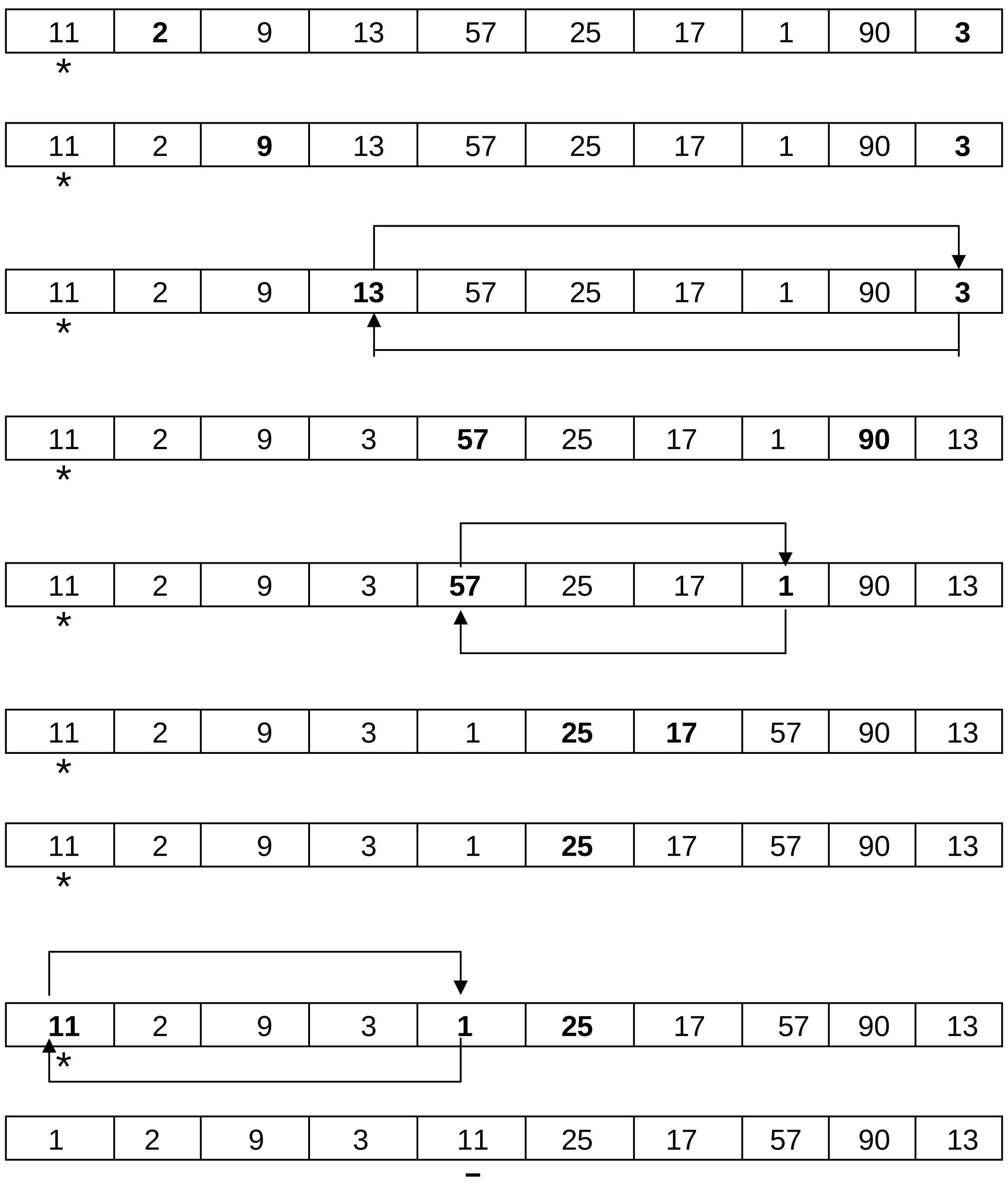


Fig. (9) Quick Sort

Suppose an array **A** consists of **10** distinct elements. The quick sort Algorithm works as follows :

(a) In the first iteration, we will place the **0th** element **11** at its final position and divide the array. Here, **11** is the pivot element. To divide the array, two index variables, **p** and **q**, are taken. The indexes are initialized in such a way that, **p** refers to the **1st** element **2** and **q** refers to the **(n-1)th** element **3**.

(b) The job of index variable **p** is to search an element that is greater than the value of **0th** location. So **p** is incremented by one till the value stored at **p** is greater than **0th** element. In our case it is incremented till **13**, as **13** is greater than **11**.

(c) Similarly, **q** needs to search an element that is smaller than the **0th** element. So **q** is decremented by one till the value stored at q is smaller than the value at **0th** location. In our case **q** is not decremented because **3** is less than **11**.

(d) When these elements are found they are interchanged. Again from the current positions **p** and **q** are incremented and decremented respectively and exchanges are made appropriately if desired.

(e) The process ends whenever the index pointers meet or crossover. In our case, they are crossed at the value **1** and **25** for the indexes **p** and **q** respectively. Finally, the **0th** element **11** is interchanged with the value at index **q** i.e., 1. The position **q** is now the final position of the **pivot** element **11**.

(f) As a result, the whole array is divided into two parts. Where all the elements before **11** are less than **11** and all the elements after **11** are greater than **11**.

(g) Now the same procedure is applied for the two sub – arrays. As a result, at the end when all sub – arrays are left with one element, the original array becomes sorted.

ALGORITHM 7 : QUICK SORT

```
     Let A[10] is an array of 10 elements.
  1. Read 5 elements of array A.
  2. Write Original Array A.
  3. Call QUICKSORT(A,0, 9).
  4. Write Sorted Array A.
  5. Exit
```

Procedure : QUICKSORT(A,LOWER,UPPER)

```
  1. IF UPPER>LOWER : then
             Set  PIVOT= Call SPLIT(A,LOWER,UPPER).
             Call QUICKSORT(A,lower,PIVOT - 1);
             Call QUICKSORT(A,PIVOT + 1,upper);
     [End of IF Structure]
```

Procedure : SPLIT(A,LOWER,UPPER)

```
  1. Set P = LOWER + 1
  2. Set Q = UPPER.
  3. Set PIVOT=A[LOWER]
  4. Repeat Step 4 to 7 for Q>=P
  5. Repeat Step 5 for A[P]<PIVOT
             Set P = P + 1.
  6. Repeat Step 6 for A[Q]>PIVOT
             Set Q = Q – 1.
```

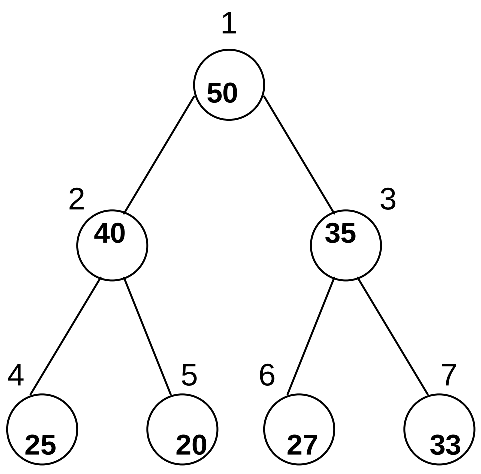
```
7.  IF Q > P : then
              Set T = A[P].
              Set A[P] = A[Q]
              Set A[Q] = T
    [End of IF Structure]
    [End of Step 4 loop]
8.  Set T = A[LOWER].
    Set A[LOWER] = A[Q].
    Set A[Q] = T.
9.  RETURN Q.
```

Program 7 : WAP to sort an array using Quick Sort.

```
#include <stdio.h>
#include <conio.h>
void quicksort(int *,int,int);
int split(int *,int,int);

void main()
{
 clrscr();
 int A[10];
 int i;
 printf("\nEnter 10 elements of an array : \n");
 for(i=0;i<10;i++)
   scanf("%d",&A[i]);

 quicksort(A,0,9);

 printf("\nSorted Array using Quick Sort :\n");
 for(i=0;i<10;i++)
   printf("%d\t",A[i]);
 getch();
}

void quicksort(int A[ ],int lower,int upper)
{
 int pivot;
 if(upper>lower)
 {
   pivot=split(A,lower,upper);
   quicksort(A,lower,pivot - 1);
   quicksort(A,pivot + 1,upper);
 }
}
```

```
int split(int A[ ],int lower,int upper)
{
  int pivot,p,q,t;
  p=lower+1;
  q=upper;
  pivot=A[lower];
  while(q>=p)
  {
    while(A[p]<pivot)
      p++;
    while(A[q]>pivot)
      q--;
    if(q>p)
    {
          t=A[p];
          A[p]=A[q];
          A[q]=t;
    }
  }
  t=A[lower];
  A[lower]=A[q];
  A[q]=t;

  return q;
}
```

(VI) HEAP SORT



1	2	3	4	5	6	7
50	**40**	**35**	**25**	**20**	**27**	**33**

Length[A] = 7

A heap is a binary tree that satisfies following properties :

1. **Shape Property :**
   It means that heap must be a complete binary tree.

2. **Order Property :**
   (i) It means that every node in heap (i.e. a root), the value stored in that node is **greater** or **equal** the value of its children. A heap that satisfies this property is known as a maximum heap.
   (ii) If the order property is such that for each node in heap (i.e. root), the value stored in that node is **less** or **equal** to its children, the heap is known as minimum heap.

Operations on the heap :

1. **Inserting An Element into Heap :**
   The elements are always inserted at the bottom of the original heap. After insertion, the heap remains complete but the order property is not followed so we use an UPHEAP or HEAPIFY operation. This involves moving the elements upward from the last position where it satisfies the order property. If the value of last node is greater than its parent, exchange it's value with it's parent and repeat the process with each parent node until the order property is satisfied.

2. **Deleting an Element from Heap :**
   Elements are always deleted from the root of the heap.

Algorithm for insertion of element :

INHEAP (TREE, N, ITEM)

A heap H with N elements is stored in the array TREE, and an item of information is given. This procedure inserts ITEM as a new element of H. PTR gives the location of ITEM as it rises in the tree, and PAR denotes the location of the parent of ITEM.

```
1. Set N = N +1 and PTR = N            [Add new node to H and initialize PTR].
2. Repeat Steps 3 to 6 while PTR<1     [Find location to insert ITEM].
3. Set PAR = [PTR/2].                  [Location of parent node].
4. If ITEM ≤ TREE[PAR], then :
       Set TREE[PTR] = ITEM, and return.
       [End of If Structure].
5. Set TREE[PTR] = TREE[PAR].          [Moves node down]
6. Set PTR = PAR                       [Updates PTR]
       [End of step 2 loop].
7. Set TREE[1] = ITEM                  [Assign ITEM as a root of H].
8. Exit
```

Algorithm for deletion of element :

DELHEAP(TREE,N,ITEM)

A heap H with N elements is stored in the array TREE. This procedure assigns the root TREE[1] of H to the variable ITEM and then reheaps the remaining elements. The variable LAST saves the value of the original last node of H. The pointers PTR, LEFT and RIGHT give the locations of LAST and its left and right children as LAST sinks in the tree.

```
1.  Set ITEM = TREE[1].                      [Removes root of H].
2.  Set LAST = TREE[N] and N = N – 1         [Removes last node of H]
3.  Set PTR = 1, LEFT = 2 and RIGHT = 3      [Initialize Pointers]
4.  Repeat Steps 5 to 7 while RIGHT ≤ N :
5.     If LAST ≥ TREE[LEFT] and LAST ≥ TREE[RIGHT], then :
          Set TREE[PTR] = LAST and Return.
       [End of If Structure].
    6.  If TREE[RIGHT] ≤ TREE[L
    7.
    8.  EFT] and PTR = LEFT
          Set TREE[PTR] = TREE[LEFT] and PTR = LEFT.
       Else
          Set TREE[PTR] = TREE[RIGHT] and PTR = RIGHT.
          [End of If structure]
7.  Set LEFT = 2 * PTR and RIGHT = LEFT + 1
          [End of Step 4 loop].
8.  If LEFT = N and if LAST M TREE[LEFT], then Set PTR = LEFT.
9.  Set TREE[PTR] = LAST
10. Exit
```