5.1 SEARCHING

5.1.1 Linear Search

Linear search or sequential search is a method for finding a particular value in a list that consists of checking every one of its elements, one at a time and in sequence, until the desired one is found.

Linear search is the simplest search algorithm; it is a special case of brute-force search. Its worst case cost is proportional to the number of elements in the list; and so is its expected cost, if all list elements are equally likely to be searched for. Therefore, if the list has more than a few elements, other methods (such as binary search or hashing) will be faster, but they also impose additional requirements.

How Linear Search works

Linear search in an array is usually programmed by stepping up an index variable until it reaches the last index. This normally requires two comparisons for each list item: one to check whether the index has reached the end of the array, and another one to check whether the item has the desired value.

Linear Search Algorithm

```
1. Repeat For J = 1 to N
2. If (ITEM == A[J]) Then
3. Print: ITEM found at location J
4. Return      [End of If]
   [End of For Loop]
5. If (J > N) Then
```

```
6. Print: ITEM doesn’t exist
      [End of If]
7. Exit
```

```
//CODE
   int a[10],i,n,m,c=0, x;

   printf("Enter the size of an array: ");
   scanf("%d",&n);

   printf("Enter the elements of the array: ");
   for(i=0;i<=n-1;i++){
       scanf("%d",&a[i]);
   }

   printf("Enter the number to be search: ");
   scanf("%d",&m);
   for(i=0;i<=n-1;i++){
       if(a[i]==m){
          x=I;
           c=1;
           break;
       }
   }
   if(c==0)
       printf("The number is not in the list");
   else
       printf("The number is found at location %d", x);

}
```

Complexity of linear Search

Linear search on a list of n elements. In the worst case, the search must visit every element once. This happens when the value being searched for is either the last element in the list, or is not in the list. However, on average, assuming the value searched for is in the list and each list element is equally likely to be the value searched for, the search visits only n/2 elements. In best case the array is already sorted i.e O(1)

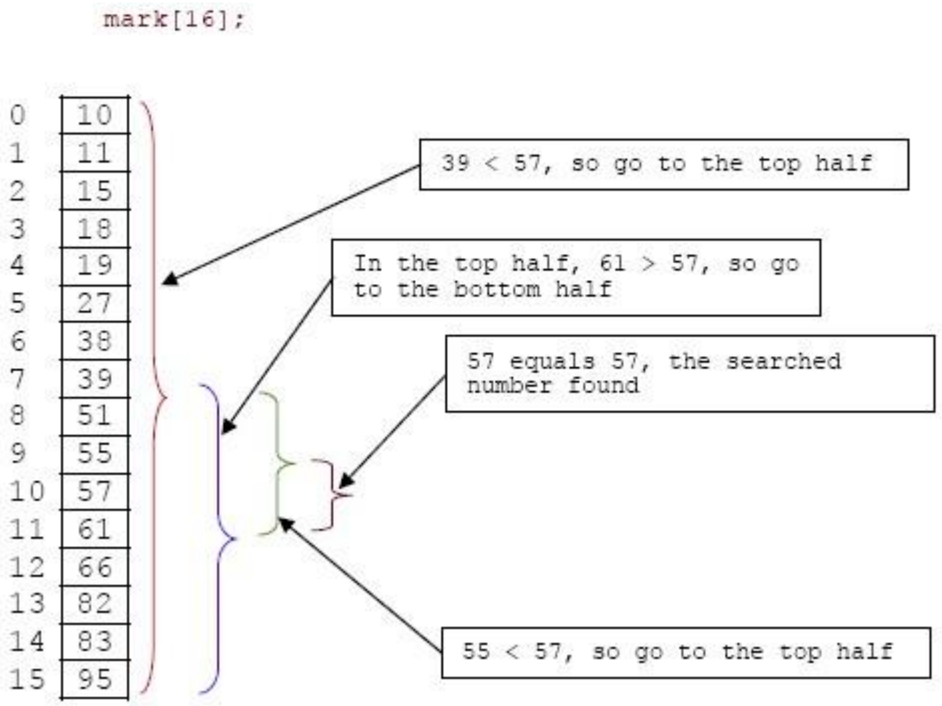
Algorithm	Worst Case	Average Case	Best Case
Linear Search	O(n)	O(n)	O(1)

5.1.2 Binary Search

A binary search or half-interval search algorithm finds the position of a specified input value (the search "key") within an array sorted by key value. For binary search, the array should be arranged in ascending or descending order. In each step, the algorithm compares the search key value with the key value of the middle element of the array. If the keys match, then a matching element has been found and its index is returned. Otherwise, if the search key is less than the middle element's key, then the algorithm repeats its action on the sub-array to the left of the middle element or, if the search key is greater, on the sub-array to the right. If the remaining array to be searched is empty, then the key cannot be found in the array and a special "not found" indication is returned.

<u>How Binary Search Works</u>

Searching a sorted collection is a common task. A dictionary is a sorted list of word definitions. Given a word, one can find its definition. A telephone book is a sorted list of people's names, addresses, and telephone numbers. Knowing someone's name allows one to quickly find their telephone number and address.


mark[16];
0 10
1 11
2 15
3 18
4 19
5 27
6 38
7 39
8 51
9 55
10 57
11 61
12 66
13 82
14 83
15 95
39 < 57, so go to the top half
In the top half, 61 > 57, so go to the bottom half
57 equals 57, the searched number found
55 < 57, so go to the top half

<u>Binary Search Algorithm</u>

1. Set BEG = 1 and END = N

2. Set MID = (BEG + END) / 2

3. Repeat step 4 to 8 While (BEG <= END) and (A[MID] ≠ ITEM)

```
4. If (ITEM < A[MID]) Then
5. Set END = MID - 1
6. Else
7. Set BEG = MID + 1
[End of If]
8. Set MID = (BEG + END) / 2
9. If (A[MID] == ITEM) Then
10. Print: ITEM exists at location MID
11. Else
12. Print: ITEM doesn’t exist
[End of If]
13. Exit
```

```
//CODE
 int    ar[10],val,mid,low,high,size,i;
clrscr();
printf("\nenter the no.s of elements u wanna input in array\n");
scanf("%d",&size);
for(i=0;i<size;i++)
{
printf("input the element no %d\n",i+1);
scanf("%d",&ar[i]);
}
printf("the arry inputed is \n");
for(i=0;i<size;i++)
{
```

```
printf("%d\t",ar[i]);
}
low=0;
high=size-1;
printf("\ninput the no. u wanna search \n");
scanf("%d",&val);
while(val!=ar[mid]&&high>=low)
{
mid=(low+high)/2;
if(ar[mid]==val)
{
printf("value found at %d position",mid+1);
}
if(val>ar[mid])
{
low=mid+1;
}
else
{
high=mid-1;
}}
```

<u>Complexity of Binary Search</u>

A binary search halves the number of items to check with each iteration, so locating an item (or determining its absence) takes logarithmic time.

Algorithm	Worst Case	Average Case	Best Case
Binary Search	O(nlogn)	O(nlogn)	O(1)

5.2 INTRODUCTION TO SORTING

Sorting is nothing but storage of data in sorted order, it can be in ascending or descending order. The term Sorting comes into picture with the term Searching. There are so many things in our real life that we need to search, like a particular record in database, roll numbers in merit list, a particular telephone number, any particular page in a book etc.
Sorting arranges data in a sequence which makes searching easier. Every record which is going to be sorted will contain one key. Based on the key the record will be sorted. For example, suppose we have a record of students, every such record will have the following data:
Roll No.
Name
Age
Here Student roll no. can be taken as key for sorting the records in ascending or descending order. Now suppose we have to search a Student with roll no. 15, we don't need to search the complete record we will simply search between the Students with roll no. 10 to 20.

Sorting Efficiency

There are many techniques for sorting. Implementation of particular sorting technique depends upon situation. Sorting techniques mainly depends on two parameters.
First parameter is the execution time of program, which means time taken for execution of program.
Second is the space, which means space taken by the program.

5.3 TYPES OF SORTING

- An internal sort is any data sorting process that takes place entirely within the main memory of a computer. This is possible whenever the data to be sorted is small enough to all be held in the main memory.
- External sorting is a term for a class of sorting algorithms that can handle massive amounts of data. External sorting is required when the data being sorted do not fit into the main memory of a computing device (usually RAM) and instead they must reside in the slower external memory (usually a hard drive). External sorting typically uses a hybrid sort-merge strategy. In the sorting phase, chunks of data small enough to fit in main memory are read, sorted, and written out to a temporary file. In the merge phase, the sorted sub files are combined into a single larger file.
- We can say a sorting algorithm is stable if two objects with equal keys appear in the same order in sorted output as they appear in the input unsorted array.

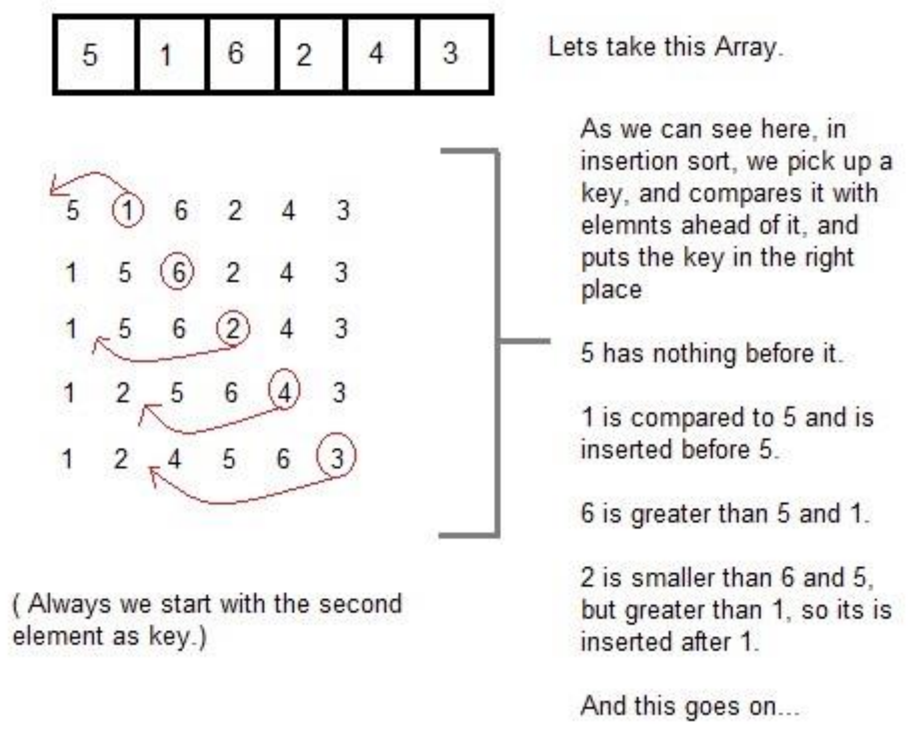
5.3.1 Insertion sort

It is a simple sorting algorithm that builds the final sorted array (or list) one item at a time. This algorithm is less efficient on large lists than more advanced algorithms such as quicksort, heapsort, or merge sort. However, insertion sort provides several advantages:

· Simple implementation

- Efficient for small data sets
- Stable; i.e., does not change the relative order of elements with equal keys
- In-place; i.e., only requires a constant amount O(1) of additional memory space.

How Insertion Sort Works


5 1 6 2 4 3
Lets take this Array.
5 1 6 2 4 3
1 5 6 2 4 3
1 5 6 2 4 3
1 2 5 6 4 3
1 2 4 5 6 3
As we can see here, in insertion sort, we pick up a key, and compares it with elemnts ahead of it, and puts the key in the right place
5 has nothing before it.
1 is compared to 5 and is inserted before 5.
6 is greater than 5 and 1.
2 is smaller than 6 and 5, but greater than 1, so its is inserted after 1.
And this goes on...
(Always we start with the second element as key.)

Insertion Sort Algorithm

This algorithm sorts the array A with N elements.

```
1. Set A[0]=-12345(infinity i.e. Any large no)
2. Repeat step 3 to 5 for k=2 to n
3. Set key=A[k] And  j=k-1
4. Repeat while key<A[j]
   A) Set A[j+1]=A[j]
   b) j=j-1
5. Set A[j+1]=key
6. Return
```

//CODE

```
int A[6] = {5, 1, 6, 2, 4, 3};
int i, j, key;
for(i=1;  i<6;  i++)
{
key  =  A[i];
j = i-1;
while(j>=0  &&  key  <  A[j])
{
A[j+1]  =  A[j];
j--;
}
A[j+1]  =  key;
}
```

<u>Complexity of Insertion Sort</u>

The number f(n) of comparisons in the insertion sort algorithm can be easily computed. First of all, the worst case occurs when the array A is in reverse order and the inner loop must use the maximum number K-1 of comparisons. Hence

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

Furthermore, One can show that, on the average, there will be approximately (K-1)/2 comparisons in the inner loop. Accordingly, for the average case.

$F(n)=O(n^2)$

Thus the insertion sort algorithm is a very slow algorithm when n is very large.

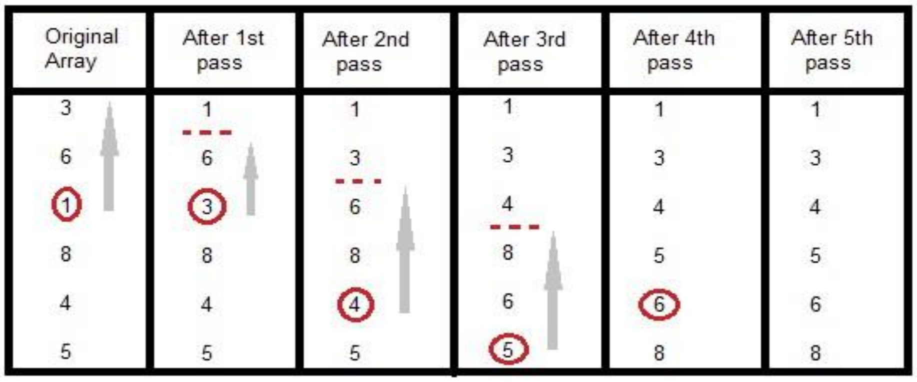
Algorithm	Worst Case	Average Case	Best Case
Insertion Sort	$n(n-1)/2 = O(n^2)$	$n(n-1)/4 = O(n^2)$	O(n)

5.3.2 Selection Sort

Selection sorting is conceptually the simplest sorting algorithm. This algorithm first finds the smallest element in the array and exchanges it with the element in the first position, then find the second smallest element and exchange it with the element in the second position, and continues in this way until the entire array is sorted

<u>How Selection Sort works</u>

In the first pass, the smallest element found is 1, so it is placed at the first position, then leaving first element, smallest element is searched from the rest of the elements, 3 is the smallest, so it is then placed at the second position. Then we leave 1 and 3, from the rest of the elements, we search for the smallest and put it at third position and keep doing this, until array is sorted



Original Array	After 1st pass	After 2nd pass	After 3rd pass	After 4th pass	After 5th pass
3	1	1	1	1	1
6	6	3	3	3	3
1	3	6	4	4	4
8	8	8	8	5	5
4	4	4	6	6	6
5	5	5	5	8	8

Selection Sort Algorithm

```
1. Repeat For J = 0 to N-1
2. Set MIN = J
3. Repeat For K = J+1 to N
4. If (A[K] < A[MIN]) Then
5. Set MIN = K
[End of If]
[End of Step 3 For Loop]
6. Interchange A[J] and A[MIN]
[End of Step 1 For Loop]
7. Exit
```

//CODE

```
void selectionSort(int a[], int size)
{
  int i, j, min, temp;
  for(i=0; i < size-1; i++ )
  {
    min = i;   //setting min as i
    for(j=i+1; j < size; j++)
    {
      if(a[j] < a[min])   //if element at j is less than element at min position
      {
       min = j;    //then set min as j
      }
    }
   temp = a[i];
  a[i] = a[min];
  a[min] = temp;
  }
}
```

Complexity of Selection Sort Algorithm

The number of comparison in the selection sort algorithm is independent of the original order of the element. That is there are n-1 comparison during PASS 1 to find the smallest element, there are n-2 comparisons during PASS 2 to find the second smallest element, and so on. Accordingly

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

Algorithm	Worst Case	Average Case	Best Case
Selection Sort	$n(n-1)/2 = O(n^2)$	$n(n-1)/2 = O(n^2)$	$O(n^2)$

5.3.3 Bubble Sort

Bubble Sort is an algorithm which is used to sort N elements that are given in a memory for eg: an Array with N number of elements. Bubble Sort compares all the element one by one and sort them based on their values.
It is called Bubble sort, because with each iteration the smaller element in the list bubbles up towards the first place, just like a water bubble rises up to the water surface.
Sorting takes place by stepping through all the data items one-by-one in pairs and comparing adjacent data items and swapping each pair that is out of order.

How Bubble Sort Works

Let us take the array of numbers "5 1 4 2 8", and sort the array from lowest number to greatest number using bubble sort. In each step, elements written in bold are being compared. Three passes will be required.

First Pass:
(5 1 4 2 8) → (1 5 4 2 8), Here, algorithm compares the first two elements, and swaps since 5 > 1.
(1 5 4 2 8) → (1 4 5 2 8), Swap since 5 > 4
(1 4 5 2 8) → (1 4 2 5 8), Swap since 5 > 2
(1 4 2 5 8) → (1 4 2 5 8), Now, since these elements are already in order (8 > 5), algorithm does not swap them.

Second Pass:
(1 4 2 5 8) → (1 4 2 5 8)
(1 4 2 5 8) → (1 2 4 5 8), Swap since 4 > 2
(1 2 4 5 8) → (1 2 4 5 8)
(1 2 4 5 8) → (1 2 4 5 8)
Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one whole pass without any swap to know it is sorted.

Third Pass:
(1 2 4 5 8) $\rightarrow$(1 2 4 5 8)
(1 2 4 5 8) $\rightarrow$(1 2 4 5 8)
(1 2 4 5 8) $\rightarrow$(1 2 4 5 8)
(1 2 4 5 8) $\rightarrow$(1 2 4 5 8)

Bubble Sort Algorithm

```
1.  Repeat Step 2 and 3 for k=1 to n

2.  Set ptr=1

3. Repeat while ptr<n-k

       a) If (A[ptr] > A[ptr+1]) Then

          Interchange  A[ptr]  and  A[ptr+1]

          [End of If]

       b) ptr=ptr+1

   [end  of  step  3  loop]

   [end of step 1 loop]

4. Exit
```

//CODE

Let's consider an array with values {5, 1, 6, 2, 4, 3}

```
int a[6] = {5, 1, 6, 2, 4, 3};
int i, j, temp;
for(i=0;   i<6,   i++)
{
  for(j=0;   j<6-i-1;   j++)
  {
    if(  a[j]  >  a[j+1])
    {
      temp = a[j];
    a[j] = a[j+1];
    a[j+1] = temp;
    }
  }
}
```

Above is the algorithm, to sort an array using Bubble Sort. Although the above logic will sort and unsorted array, still the above algorithm isn't efficient and can be enhanced further. Because as per the above logic, the for loop will keep going for six iterations even if the array gets sorted after the second iteration.
Hence we can insert a flag and can keep checking whether swapping of elements is taking place or not. If no swapping is taking place that means the array is sorted and we can jump out of the for loop.

```
int a[6] = {5, 1, 6, 2, 4, 3};
int i, j, temp;
for(i=0;  i<6,  i++)
{
  for(j=0;  j<6-i-1;  j++)
  {
    int flag = 0;        //taking a flag variable
    if(  a[j]  >  a[j+1])
    {
      temp = a[j];
      a[j] = a[j+1];
      a[j+1] = temp;
      flag = 1;         //setting flag as 1, if swapping occurs
    }
  }
  if(!flag)             //breaking out of for loop if no swapping takes place
  {
    break;
  }
}
```

In the above code, if in a complete single cycle of j iteration(inner for loop), no swapping takes place, and flag remains 0, then we will break out of the for loops, because the array has already been sorted.

Complexity of Bubble Sort Algorithm

In Bubble Sort, n-1 comparisons will be done in 1st pass, n-2 in 2nd pass, n-3 in 3rd pass and so on. So the total number of comparisons will be

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

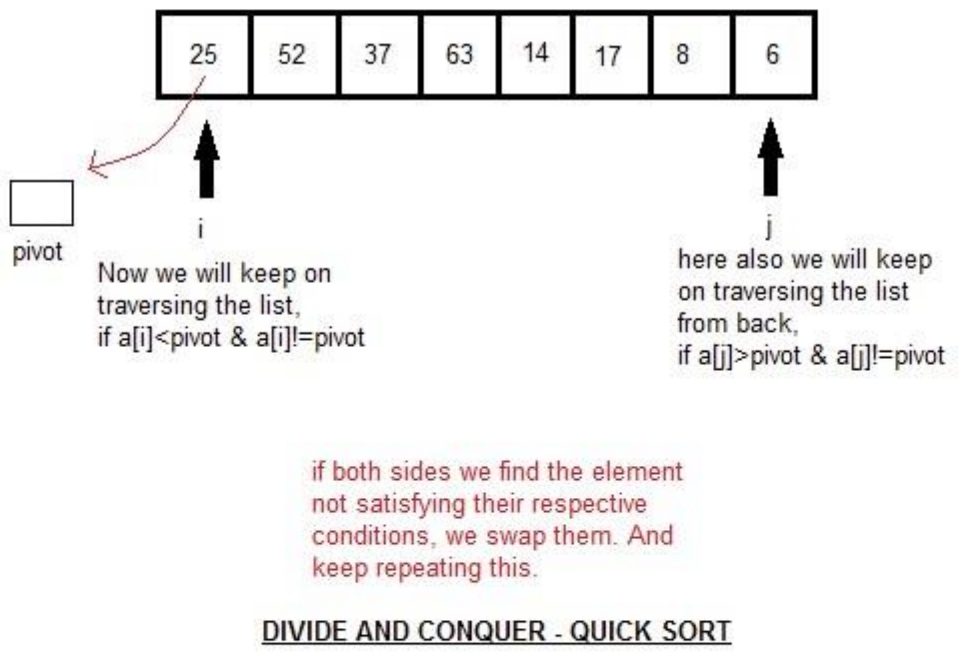
Algorithm	Worst Case	Average Case	Best Case
Bubble Sort	$n(n-1)/2 = O(n^2)$	$n(n-1)/2 = O(n^2)$	$O(n)$

5.3.4 Quick Sort

Quick Sort, as the name suggests, sorts any list very quickly. Quick sort is not stable search, but it is very fast and requires very less additional space. It is based on the rule of Divide and Conquer (also called partition-exchange sort). This algorithm divides the list into three main parts
Elements less than the Pivot element
Pivot element
Elements greater than the pivot element

How Quick Sort Works

In the list of elements, mentioned in below example, we have taken 25 as pivot. So after the first pass, the list will be changed like this.
6 8 17 14 **25** 63 37 52
Hence after the first pass, pivot will be set at its position, with all the elements smaller to it on its left and all the elements larger than it on the right. Now 6 8 17 14 and 63 37 52 are considered as two separate lists, and same logic is applied on them, and we keep doing this until the complete list is sorted.


25
52
37
63
14
17
8
6
pivot
i
Now we will keep on traversing the list, if a[i]<pivot & a[i]!=pivot
j
here also we will keep on traversing the list from back, if a[j]>pivot & a[j]!=pivot
if both sides we find the element not satisfying their respective conditions, we swap them. And keep repeating this.
DIVIDE AND CONQUER - QUICK SORT

Quick Sort Algorithm

```
QUICKSORT (A, p, r)

1 if p < r

2 then q ← PARTITION (A, p, r)
```

```
3 QUICKSORT (A, p, q - 1)

4 QUICKSORT (A, q + 1, r)
```

The key to the algorithm is the PARTITION procedure, which rearranges the subarray A[p r] in place.

```
PARTITION (A, p, r)

1 x ← A[r]

2 i ← p - 1

3 for j ← p to r - 1

4           do if A[j] ≤ x

5                 then i ← i + 1

6                 exchange A[i] ↔ A[j]

7 exchange A[i + 1] ↔ A[r]

8 return i + 1
```

```
//CODE

/*  Sorting using Quick Sort Algorithm
a[] is the array, p is starting index, that is 0,
and r is the last index of array. */

void quicksort(int a[], int p, int r)
{
 if(p  <  r)
 {
   int q;
   q  =  partition(a,  p,  r);
 quicksort(a,    p,    q-1);
 quicksort(a,    q+1,    r);
 }
}

int partition (int a[], int p, int r)
{
 int  i,  j,  pivot,  temp;
 pivot = a[p];
 i  =  p;
 j = r;
 while(1)
 {
  while(a[i] < pivot && a[i] != pivot)
  i++;
  while(a[j] > pivot && a[j] != pivot)
  j--;
  if(i  <  j)
  {
   temp  =  a[i];
  a[i]    =    a[j];
  a[j]   =   temp;
  }
  else
  {
   Return    j;
  }}}
```

Complexity of Quick Sort Algorithm

The Worst Case occurs when the list is sorted. Then the first element will require n comparisons to recognize that it remains in the first position. Furthermore, the first sublist will be empty, but the second sublist will have n-1 elements. Accordingly the second element require n-1 comparisons to recognize that it remains in the second position and so on.

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

Algorithm	Worst Case	Average Case	Best Case
Quick Sort	$n(n+1)/2 = O(n^2)$	O(n logn)	O(n logn)

5.3.5 Merge Sort

Merge Sort follows the rule of Divide and Conquer. But it doesn't divide the list into two halves. In merge sort the unsorted list is divided into N sub lists, each having one element, because a list of one element is considered sorted. Then, it repeatedly merge these sub lists, to produce new sorted sub lists, and at lasts one sorted list is produced.
Merge Sort is quite fast, and has a time complexity of O(n log n). It is also a stable sort, which means the equal elements are ordered in the same order in the sorted list.

How Merge Sort Works

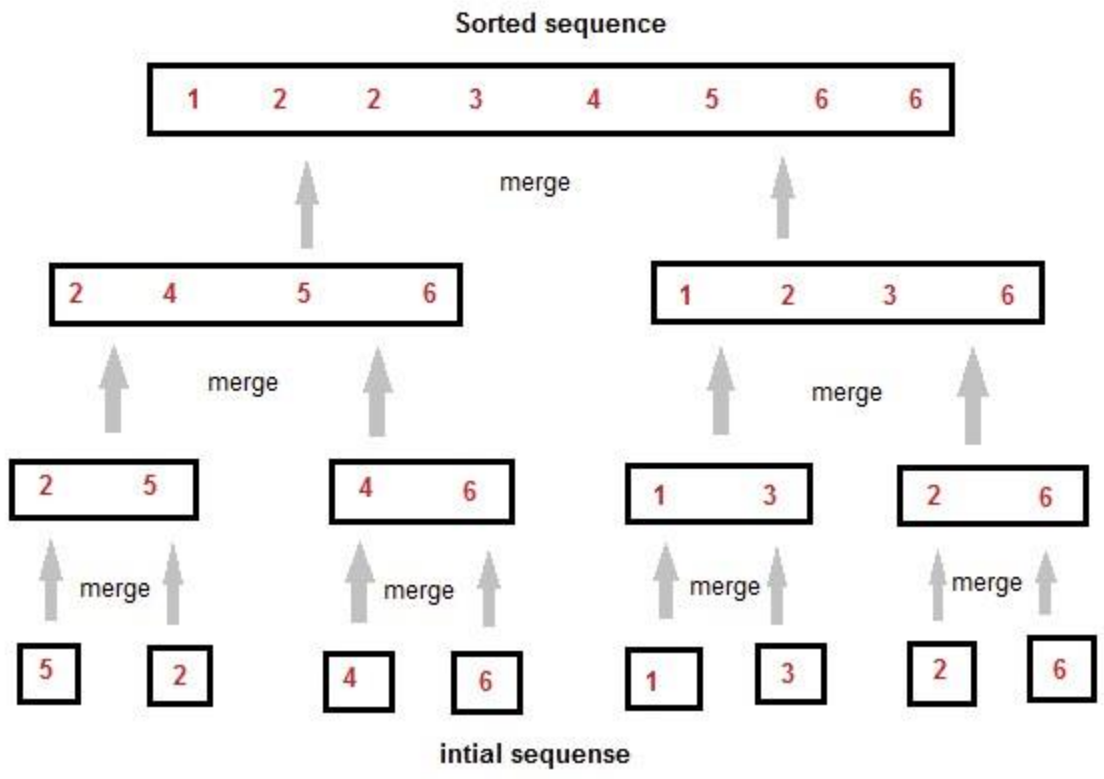
Suppose the array A contains 8 elements, each pass of the merge-sort algorithm will start at the beginning of the array A and merge pairs of sorted subarrays as follows.

PASS 1. Merge each pair of elements to obtain the list of sorted pairs.

PASS 2. Merge each pair of pairs to obtain the list of sorted quadruplets.

PASS 3. Merge each pair of sorted quadruplets to obtain the two sorted subarrays.

PASS 4. Merge the two sorted subarrays to obtain the single sorted array.



Merge Sort Algorithm

```
/* Sorting using Merge Sort Algorithm
 a[] is the array, p is starting index, that is 0,
and r is the last index of array.  */

Lets take a[5] = {32, 45, 67, 2, 7} as the array to be sorted.

void mergesort(int a[], int p, int r)
{
  int q;
  if(p  <  r)
  {
    q  =  floor(  (p+r)  /  2);
    mergesort(a, p, q);
    mergesort(a, q+1, r);
  merge(a, p, q, r);
  }
}

void merge (int a[], int p, int q, int r)
{
  int b[5];     //same size of a[]
  int  i,  j,  k;
  k = 0;
  i = p;
  j = q+1;
  while(i <= q && j <= r)
  {
    if(a[i]  <  a[j])
    {
      b[k++] = a[i++];       // same as b[k]=a[i]; k++; i++;
    }
    else
    {
      b[k++]  =  a[j++];
    }
  }

  while(i  <=  q)
  {
    b[k++]  =  a[i++];
  }

  while(j  <=  r)
  {
    b[k++]  =  a[j++];
  }

  for(i=r;  i  >=  p;  i--)
  {
    a[i] = b[--k];       // copying back the sorted list to a[]
  }
}
```

Complexity of Merge Sort Algorithm

Let f(n) denote the number of comparisons needed to sort an n-element array A using merge-sort algorithm. The algorithm requires at most logn passes. Each pass merges a total of n elements and each pass require at most n comparisons. Thus for both the worst and average case

F(n) ≤ n logn

Thus the time complexity of Merge Sort is O(n Log n) in all 3 cases (worst, average and best) as merge sort always divides the array in two halves and take linear time to merge two halves.

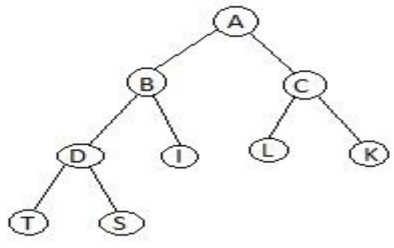
Algorithm	Worst Case	Average Case	Best Case
Merge Sort	O(n logn)	O(n logn)	O(n logn)

5.3.6 Heap Sort

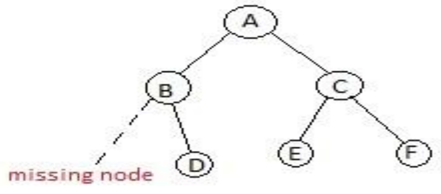
Heap Sort is one of the best sorting methods being in-place and with no quadratic worst-case scenarios. Heap sort algorithm is divided into two basic parts
Creating a Heap of the unsorted list.
Then a sorted array is created by repeatedly removing the largest/smallest element from the heap, and inserting it into the array. The heap is reconstructed after each removal.
What is a Heap?
Heap is a special tree-based data structure that satisfies the following special heap properties
Shape Property: Heap data structure is always a Complete Binary Tree, which means all levels of the tree are fully filled.
Heap Property: All nodes are either greater than or equal to or less than or equal to each of its children. If the parent nodes are greater than their children, heap is called a Max-Heap, and if the parent nodes are smaller than their child nodes, heap is called Min-Heap.

How Heap Sort Works

Initially on receiving an unsorted list, the first step in heap sort is to create a Heap data structure (Max-Heap or Min-Heap). Once heap is built, the first element of the Heap is either largest or smallest (depending upon Max-Heap or Min-Heap), so we put the first element of the heap in our array. Then we again make heap using the remaining elements, to again pick the first element of the heap and put it into the array. We keep on doing the same repeatedly until we have the complete sorted list in our array.



Complete Binary Tree



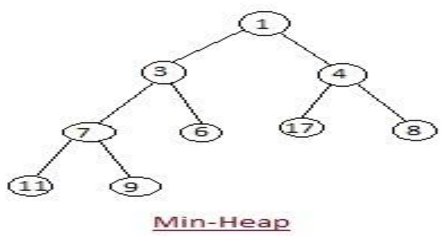
In-Complete Binary Tree

Heap Sort Algorithm

```
• HEAPSORT(A)
1.      BUILD-MAX-HEAP(A)
2.      for i ← length[A] downto 2
3.      do exchange A[1] ↔ A[i ]
4.      heap-size[A] ← heap-size[A] - 1
5.       MAX-HEAPIFY(A, 1)
```

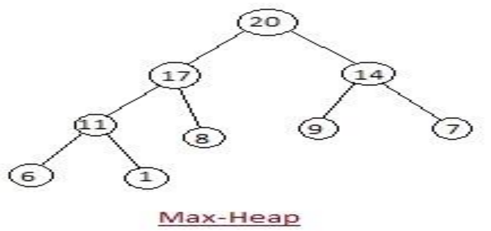
```
• BUILD-MAX-HEAP(A)
1.      heap-size[A] ← length[A]
2.      for i ← length[A]/2 downto 1
3.      do MAX-HEAPIFY(A, i )
```

```
• MAX-HEAPIFY(A, i )
1.      l ← LEFT(i )
2.      r ← RIGHT(i )
3.      if l ≤ heap-size[A] and A[l] > A[i ]
4.      then largest ←l
5.      else largest ←i
6.      if r ≤ heap-size[A] and A[r] > A[largest]
7.      then largest ←r
8.      if largest = i
9.      then exchange A[i ] ↔ A[largest]
10.     MAX-HEAPIFY(A, largest)
```



Min-Heap

In min-heap, first element is the smallest. So when we want to sort a list in ascending order, we create a Min-heap from that list, and picks the first element, as it is the smallest, then we repeat the process with remaining elements.



Max-Heap

In max-heap, the first element is the largest, hence it is used when we need to sort a list in descending order.

//CODE

In the below algorithm, initially heapsort() function is called, which calls buildmaxheap() to build heap, which inturn uses maxheap() to build the heap.

```
void heapsort(int[], int);
void  buildmaxheap(int  [],  int);
void maxheap(int [], int, int);

void main()
{
  int a[10], i, size;
  printf("Enter  size  of  list");    // less than 10, because max size of array is 10
  scanf("%d",&size);
  printf(  "Enter"  elements");
  for( i=0; i < size; i++)
  {
    Scanf("%d",&a[i]);
  }
  heapsort(a,   size);
  getch();
}


void  heapsort  (int  a[],  int  length)
{
  buildmaxheap(a, length);
  int heapsize, i, temp;
  heapsize = length - 1;
  for( i=heapsize; i >= 0; i--)
  {
    temp = a[0];
    a[0]    =    a[heapsize];
    a[heapsize]   =   temp;
    heapsize--;
    maxheap(a,   0,   heapsize);
  }
  for(  i=0;  i  <  length;  i++)
  {
    Printf("\t%d"     ,a[i]);
  }
}

void  buildmaxheap  (int  a[],  int  length)
{
  int i, heapsize;
  heapsize = length - 1;
```

```
 for( i=(length/2); i >= 0; i--)
 {
   maxheap(a,  i,  heapsize);
 }
}

void maxheap(int a[], int i, int heapsize)
{
 int l, r, largest, temp;
 l = 2*i;
 r = 2*i + 1;
 if(l <= heapsize && a[l] > a[i])
 {
   largest  =  l;
 }
 else
 {
   largest  =  i;
 }
 if( r <= heapsize && a[r] > a[largest])
 {
   largest  =  r;
 }
 if(largest  !=  i)
 {
   temp = a[i];
   a[i]   =   a[largest];
   a[largest] = temp;
   maxheap(a,  largest,  heapsize);
 }
}
```

Complexity of Heap Sort Algorithm

The heap sort algorithm is applied to an array A with n elements. The algorithm has two phases, and we analyze the complexity of each phae separately.

Phase 1. Suppose H is a heap. The number of comparisons to find the appropriate place of a new element item in H cannot exceed the depth of H. Since H is complete tree, its depth is bounded by $\log_2 m$ where m is the number of elements in H. Accordingly, the total number g(n) of comparisons to insert the n elements of A into H is bounded as

$$g(n) \leq n \log_2 n$$

Phase 2. If H is a complete tree with m elements, the left and right subtrees of H are heaps and L is the root of H Reheaping uses 4 comparisons to move the node L one step down the tree H. Since the depth cannot exceeds $\log_2 m$, it uses $4\log_2 m$ comparisons to find the appropriate place of L in the tree H.

$$h(n) \leq 4n\log_2 n$$

Thus each phase requires time proportional to $n\log_2 n$, the running time to sort n elements array A would be $n\log_2 n$

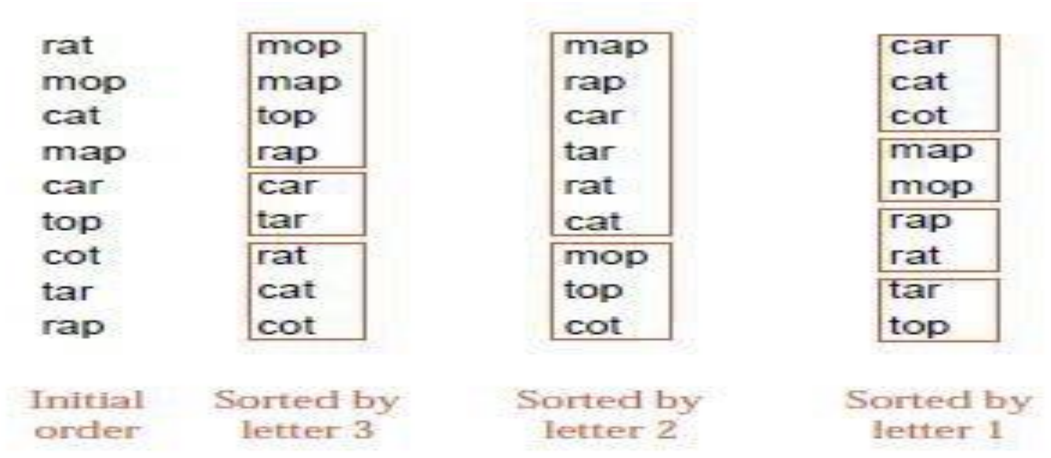
Algorithm	Worst Case	Average Case	Best Case
Heap Sort	O(n logn)	O(n logn)	O(n logn)

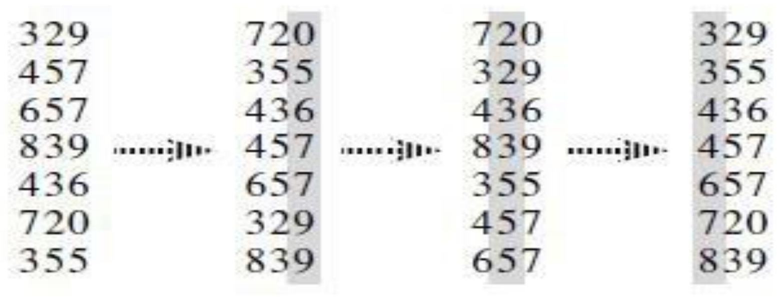
5.3.7 Radix Sort

The idea is to consider the key one character at a time and to divide the entries, not into two sub lists, but into as many sub lists as there are possibilities for the given character from the key. If our keys, for example, are words or other alphabetic strings, then we divide the list into 26 sub lists at each stage. That is, we set up a table of 26 lists and distribute the entries into the lists according to one of the characters in the key.

<u>How Radix Sort Works</u>

A person sorting words by this method might first distribute the words into 26 lists according to the initial letter (or distribute punched cards into 12 piles), then divide each of these sub lists into further sub lists according to the second letter, and so on. The following idea eliminates this multiplicity of sub lists: Partition the items into the table of sub lists first by the least significant position, not the most significant. After this first partition, the sub lists from the table are put back together as a single list, in the order given by the character in the least significant position. The list is then partitioned into the table according to the second least significant position and recombined as one list. When, after repetition of these steps, the list has been partitioned by the most significant place and recombined, it will be completely sorted. This process is illustrated by sorting the list of nine three-letter words below.





Radix Sort Algorithm

Radixsort(A,d)

```
1. For i←1 to d
2. Do use a stable sort to sort array A on digit i
```

Complexity of Radix Sort Algorithm

The list A of n elements A_1, A_2,……………A_n is given. Let d denote the radix(e.g d=10 for
decimal digits, d=26 for letters and d=2 for bits) and each item A_i
is represented by means of s of
the digits:

$$A_i = d_{i1}\, d_{i2} \ldots\ldots\ldots\ldots\ldots\ldots\, d_{is}$$

The radix sort require s passes, the number of digits in each item
. Pass K will compare each d_{ik} with each of the d digits. Hence

$$C(n) \leq d*s*n$$

Algorithm	Worst Case	Average Case	Best Case
Radix Sort	$O(n^2)$	d*s*n	O(n logn)