UNIT –III

Searching & Sorting

Syllabus:

Searching: Linear and Binary Search.
Sorting: Bubble sort, Insertion Sort, Heap Sort, Merge Sort & Quick Sort.

Learning Material

Searching:

It is a process of verifying whether the searching element is available in the given set of elements or not.

Types of Searching techniques are:

1. Linear Search
2. Binary Search

1. **Linear Search**:-

In linear search, search process starts from starting index of array i.e. **0^{th}** index of array and end's with ending index of array i.e. **$(n-1)^{th}$** index. Here searching is done in Linear fashion (Sequential fashion).

Algorithm linearsearch(a<array>, n, key)

Input: a is an array with n elements, key is the element to be searched.

Output: key element is found in array, if it is available.

1. flag = 0
2. i =0
3. while(i<n)
    a) if(a[i] = = key)
        i) flag = 1
        ii) break
    b) end if
    c) i = i +1
4. end loop
5. if(flag == 1)
    a) print(key element is found in the array)

else

a) print(key element is not found in the array)

6.end if

End linearsearch

C program to implement linear search:-

```
#include<stdio.h>
#include<conio.h>
main( )
{
        int  a[10],n,key,index,i,flag=0;
        clrscr( );
        printf("\nEnter size of the array");
        scanf("%d",&n);
        printf("\nEnter elements of the array");
        for(i=0;i<n;i++)
        scanf("%d",&a[i]);
        printf("\nEnter key element");
        scanf("%d",&key);
        for(i=0;i<n;i++)
        {
                if(a[i]==key)
                {
                        flag=1;
                        break;
                }

        }
        if(flag==1)
                printf("\nKey element is found");
        else
                printf("\nKey element is not found");
        getch();
}
```

Analysis of linear search:-

1. **Best case time complexity:-** In linear search, the best case will occurs if key element is the first element of the array.
   Best case time complexity T(n)=O(1)
2. **Worst case time complexity:-** In linear search, the worst case will occurs if

key element is the last element of the array.
Worst case time complexity T(n)=O(n)

3. **Average case time complexity:-** In linear search, the average case will occurs if key element is in between of the array.
Average case time complexity T(n)=O(n)

---

2. Binary Search:-

The input to binary search must be in ascending order i.e. set of elements be in ascending order.

Searching process in Binary search as follows:

- First, key element is compared with middle element of array.
- If key element is equal to middle element of array then Successful Search.
- If key element is less than the middle element of array, then search in LEFT part. So update high value therefore high=mid-1.
- If key element is greater than middle element of array, then search in RIGHT part. So update low value therefore low=mid+1.

Algorithm binarysearch(a<array>, n, key)

Input: a is an array with n elements, key is the element to be searched.

Output: key element is found in array, if it is available.

1. flag= 0
2. low = 0
3. high = n-1
4. while(low<=high)
    a) mid=(low+high)/2.
    b) if(key==a[mid])
        i) flag=1
        ii) break
    c) else if(key<a[mid])
        i) high = mid - 1

d) else if(key>a[mid])

i) low= mid + 1

5.end loop

6.if(flag==1)

a) print(key element is found)

else

a) print(key element is not found)

End binarysearch

C Program to implement binary search:-

```
#include<stdio.h>
#include<conio.h>
main( )
{
        int  n,a[10],i,key,low,mid,high,flag;
        clrscr();
        printf("\nEnter size of the array");
        scanf("%d",&n);
        printf("\nEnter elements of the array");
        for(i=0;i<n;i++)
                scanf("%d",&a[i]);
        printf("\nEnter key element");
                scanf("%d",&key);
        flag=0;
        low=0;
        high=n-1;
        while(low<=high)
        {
                mid=(low+high)/2;
                if(key==a[mid])
                {
                        flag=1;
                        break;
                }
                else if(key<a[mid])
                        high=mid-1;
                else
```

```
                low=mid+1;
        }
        if(flag==1)
                printf("key element is found");
        else
                printf("key element is not found");
        getch();
}
```

Analysis of binary search:-

1. **Best Case time complexity:-** In binary search, the best case will occurs if key element is array's middle element.
   So, Best case time complexity T(n)=O(1)

2. **Worst case time complexity:-**In binary search, the worst case will occurs if key element is either first or last element.

   $T(n)=T(n/2) + 1$

   The given array Is divided into 2 parts — 1 comparison (to compare key element with array's middle element

   $T(n)=T(n/2)+1 \Rightarrow T(n/2^1)+1$
   $T(n)=T(n/4)+1+1 \Rightarrow T(n/2^2)+2$
   $T(n)=T(n/8)+1+1+1 \Rightarrow T(n/2^3)+3$

   .
   .
   .

   $T(n)=T(n/2^k)+k$
   Let $2^k=n$
   Apply log on both sides then $\log 2^k=\log n$
   $k\log 2=\log n$
   $k.1=\log n$
   $k=\log n$

   $T(n)=T(n/2^k)+k$
   $=T(n/n)+\log n$
   $=T(1)+\log n$
   $=\log n$

So, worst case time complexity=O(logn)

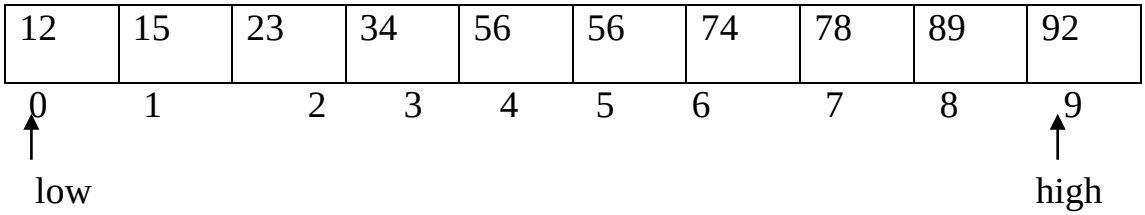
3. **Average case tie complexity:- O(logn)**

Ex:- Using binary search trace for the key element 89 from the given list of elements 34,23,15,89,74,56,92,78,12,56

Ans)We can implement binary search only when the elements are in ascending order.

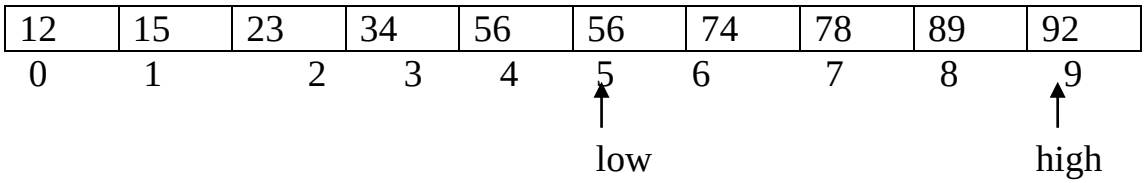
So the given list of elements in ascending order are 12,15,23,34,56,56,74,78,89,92

n=10, low=0, high=n-1
=9



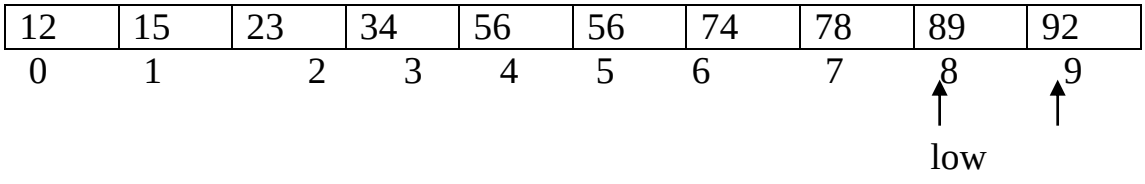
low<=high
0 <= 9 condition is true
mid=(low+high)/2 =>(0+9)/2 =>9/2 =>4 (int/int=int)
so, **mid=4**

Compare key element (89) with array's middle element (a[4] i.e. 56)
89>56 , so low=mid+1
low=4+1
=5



low<=high
5 <= 9 condition is true
mid=(low+high)/2 =>(5+9)/2 =>14/2 =>7
so, **mid=7**

compare key element(89) with array's middle element(a[7] i.e. 78)
89>78 , so low=mid+1
low=7+1
=8



low<=high
8 <= 9 condition is true
mid=(low+high)/2 =>(8+9)/2 =>17/2 =>8
so, **mid=8**

compare key element(89) with array's middle element(a[8] i.e. 89)
89= =89, so key element is found.

Sorting: Sorting means arranging the elements either in ascending or descending order.

There are two types of sorting.

1. Internal Sorting.
2. External Sorting.

1. Internal Sorting: For sorting a set of elements, if we use only primary memory (Main memory), then that sorting process is known as internal sorting. i.e. internal sorting deals with data stored in computer memory.

2. External Sorting: For sorting a set of elements, if we use both primary memory (Main memory) and secondary memory, then that sorting process is known as external sorting. i.e. external sorting deals with data stored in files.

Different types of sorting techniques.

1. Bubble sort
2. Insertion sort
3. Merge sort
4. Quick sort
5. Heap sort

1. Bubble sort:

- In bubble sort, in each iteration we compare adjacent elements i.e. i^{th} index element will be compared with $(i+1)^{th}$ index element, if they are not in ascending order, then swap them.
- After first iteration the biggest element is moved to the last position.
- After second iteration the next biggest element is moved to next last but one position.
- In bubble sort for sorting n elements, we require (n-1) passes (or) iterations

Process:

1. In pass1, a[0] and a[1] are compared, then a[1] is compared with a[2], then a[2] is compared with a[3] and so on. Finally a[n-2] is compared with a[n-1]. Pass1 involves (n-1) comparisons and places the biggest element at the highest index of the array.

2. In pass2, a[0] and a[1] are compared, then a[1] is compared with a[2], then a[2] is compared with a[3] and so on. Finally a[n-3] is compared with a[n-2]. Pass2 involves (n-2) comparisons and places the next biggest element at the next highest index of the array.

3. In pass (n-1), a[0] and a[1] are compared. After this step all the elements of the array are arranged in ascending order.

C Program to implement bubble sort:-

```
#include<stdio.h>
#include<conio.h>
void bubble_sort(int [ ],int);
 main()
{
        int n,a[10],i;
       clrscr();
       printf("\n Enter size of the array");
        scanf("%d",&n);
       printf("\n Enter elements of the array");
       for(i=0;i<n;i++)
              scanf("%d",&a[i]);
       bubble_sort(a,n);
       printf("\n After sorting the elements of the array are");
        for(i=0;i<n;i++)
              printf("%d \t",a[i]);
        getch();
}

void bubble_sort(int a[ ],int n)
{
        int i,j,temp;
       for(i=0;i<n-1;i++)
       {
              for(j=0;j<n-1-i;j++)
              {
                     if(a[j]>a[j+1])
                     {
                            temp=a[j];
```

```
                    a[j]=a[j+1];
                    a[j+1]=temp;
                }
            }
        }
}
```

Analysis of bubble sort:-

1. Best case time complexity:- The best case will occur if array is already in sorted (Ascending) order .

T(n)=O(n)

2. Worst case time complexity:-The worst case will occur if the elements are in descending order.

Iteration	**No. Of comparisions**
1st	n-1
2nd	n-2
3rd	n-3
4th	n-4
.	
.	
.	
.	
Last iteration	1

Worst case time complexity $T(n)=(n-1)+(n-2)+(n-3)+----------+1$

$T(n)=n(n-1)/2$

$\mathbf{T(n)=n^2}$

3.Average case time complexity:- $O(n^2)$

---

2. Insertion sort

1. Step 1: The second element of an array is compared with the elements that appears before it (only first element in this case). If the second element is smaller than first element, second element is inserted in the position of first element**. After first step, first two elements of an array will be sorted.**
2. Step 2: The third element of an array is compared with the elements that appears before it (first and second element). If third element is smaller than first element, it is inserted in the position of first element. If third element is larger than first element but, smaller than second element, it is inserted in the position of second element. If third element is larger than both the elements, it

is kept in the position as it is. **After second step, first three elements of an array will be sorted.**

3. Step 3: Similarly, the fourth element of an array is compared with the elements that appears before it (first, second and third element) and the same procedure is applied and that element is inserted in the proper position. **After third step, first four elements of an array will be sorted.**

C Program to implement Insertion Sort:-

```
#include<stdio.h>
#include<conio.h>
int a[10],n;
void insertionsort();
void main()
{
	int i;
	printf("enter size\n");
	scanf("%d",&n);
	printf("enter elements to sort\n");
	for(i=0;i<n;i++)
		scanf("%d",&a[i]);
	insertionsort();

}
void insertionsort()
{
	int i,j,temp;
	for(i=1;i<n;i++)
	{
		temp=a[i];
		for(j=i-1;a[j]>temp && j>=0;j--)
		{
			a[j+1]=a[j];
		}
			a[j+1]=temp;
	}
	for(i=0;i<n;i++)
		printf("%d ",a[i]);
}
```

Analysis of insertion sort:-

1.Best case time complexity:- The best case will occurs when the elements are in ascending order.

Ex:- 10 20 30 40 50

Iteration	**No. Of comparisions**
1st	1
2nd	1
3rd	1
4th	1
.	
.	
.	
.	
Last iteration	1

T(n)=1+1+1+1+------+1

=n-1

T(n)=O(n)

2. Worst case time complexity:-The worst case will occur if the elements are in descending order.

Iteration	**No. Of comparisions**
1st	n-4
2nd	n-3
3rd	n-2
4th	n-1
.	
.	
.	
.	
Last iteration	1

Worst case time complexity T(n)=(n-4)+(n-3)+(n-2)+----------+1

T(n)=n(n-1)/2

$T(n)=n^2$

3.Average case time complexity:- **$O(n^2)$**

---

3. Selection Sort:-

1. In selection sort first find the smallest element in the array and swap with 0^{th} position element. So, after 1^{st} iteration the smallest element will be at 0^{th} position.

2. Then find the second smallest element in the array and swap with 1^{st} position element. So, after 2^{nd} iteration the second smallest element will be at 1^{st} position.

3. Repeat this procedure until array is sorted.

Program for Selection Sort.

```
#include<stdio.h>
#include<conio.h>
void selection_sort(int [ ],int);
void main()
{
        int n,a[10],i;
        clrscr();
        printf("\n Enter size of the array");
        scanf("%d",&n);
        printf("\n Enter elements of the array");
        for(i=0;i<n;i++)
                scanf("%d",&a[i]);
        selection_sort(a,n);
        printf("\n After sorting the elements of the array are");
        for(i=0;i<n;i++)
                printf("%d \t",a[i]);
        getch();
}

void selection_sort(int a[ ],int n)
{
 int i,j,temp;
        for(i=0;i<n-1;i++)
        {
                for(j=i+1;j<n;j++)
                {
                        if(a[i]>a[j])
                        {
                                temp=a[i];
                                a[i]=a[j];
                                a[j]=temp;
                        }
                }
        }
}
```

Analysis of selection sort:-

The time complexity of selection sort in all cases is **$O(n^2)$**

---

4. Merge sort:- Merge sort works on the principle of **Divide and Conquer** technique.

Any **Divide and Conquer algorithm** is implemented using 3 steps.

1.Divide

2.Conquer

3.Combine or merge

1.Divide:- The given array is divided into 2 parts.

2.Conquer:- Each sub array is sorted recursively, so that the first sub array and second sub array are in sorted order.

3.Combine or merge:- Merge the solutions of 2 sub arrays.

The fundamental operation of this algorithm is merging of 2 sorted lists.
If n=1, there is only one element to sort and the answer is at hand otherwise recursively sort the first sub array and the second sub array.
This gives 2 sorted lists, which can be merged together using merging algorithm.

Merging algorithm:-

The basic merging algorithm takes 2 input arrays A and B and an output array C, 3 variables i,j,k which are initially set to the beginning of their respective arrays.
The smallest of A and B is copied to the array C and the appropriate variables are incremented.
When either input list is completed, then the remaining elements of other list are copied to the array C.

Ex:- To sort 8-element array 24 13 26 1 2 27 28 15

1.Divide:- The given array is divided into 2 parts.

List 1:- 24 13 26 1

List 2:- 2 27 28 15

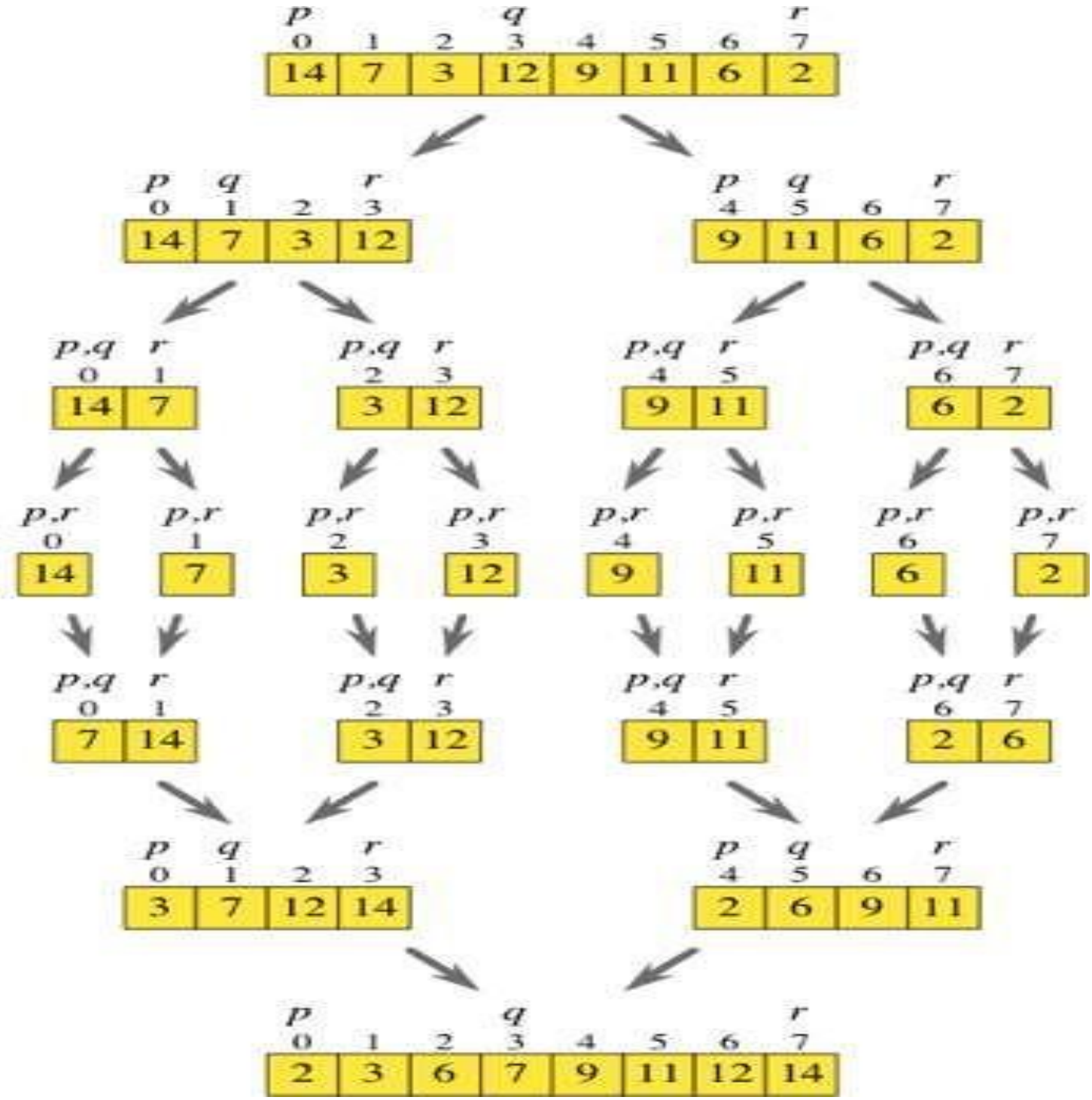
2.Conquer:- Each sub array is sorted recursively, so that the first sub array and second sub array are in sorted order.

List 1:- 1 13 24 26

List 2:- 2 15 27 28

3.Combine or merge:- Merge the solutions of 2 sub arrays. After merging elements are

1 2 13 15 24 26 27 28



C Program to implement Merge Sort:-

```
#include<stdio.h>
#include<conio.h>
int a[20],n;
void merge(int i,int j,int p,int q);
void mergesort(int l,int h);
void main()
{
        int i;
        clrscr();
        printf("enter no array of elements to sort\n");
        scanf("%d",&n);
        printf("enter  array of elements to sort\n");
        for(i=0;i<n;i++)
        scanf("%d",&a[i]);
        mergesort(0,n-1);
        printf("elements after mergesort \n");
        for(i=0;i<n;i++)
        printf("%d ",a[i]);
        getch();

}
void mergesort(int low,int high)
{
        int mid;
        if(low<high)
        {
                mid=(low+high)/2;
                mergesort(low,mid);
                mergesort(mid+1,high);
                merge(low,mid,mid+1,high);
        }
}
void merge(int i,int j,int p,int q)
```

```
{
        int c[100],k=i,r=i;
        while((i<=j) && (p<=q))
        {
                if(a[i]<a[p])
                c[k++]=a[i++];
                else
                c[k++]=a[p++];
        }
        while(i<=j)
        {
                c[k++]=a[i++];
        }
        while(p<=q)
        {
                c[k++]=a[p++];
        }

        for(i=r;i<=q;i++)
        a[i]=c[i];

}
```

Analysis of Merge Sort:- If n=1, then we can get the solution directly.
So, T(n)=1 if n=1
If n>1, then the given array is divided into 2 parts where each array contains n/2 elements and we need minimum 'n' comparisons for merging.

$T(n)=2T(n/2) + n$ if $n>1$

The given array is divided into 2 parts Where each array Contains n/2 elements.

n comparisons for merging

$T(n)=2T(n/2)+n \quad => 2^1 T(n/2^1)+1.n$

$T(n)=2[2T(n/4)+n/2]+n$

$=4T(n/4)+2n \quad => 2^2T(n/2^2)+2.n$

$T(n)=4[2T(n/8)+n/4]+2n$

$=8T(n/8)+3n \quad => 2^3T(n/2^3)+3.n$

.

.

.

$T(n)=2^kT(n/2^k)+k.n$

Let $2^k=n$

Apply log on both sides then $\log 2^k=\log n$

$K\log 2=\log n$

$k.1=\log n$

$k=\log n$

$T(n)=2^kT(n/2^k)+k.n$

$=n.T(n/n)+\log n.n$

$=T(1)+n \log n$

$=1+n \log n$

$=n \log n$

So, the time complexity of Merge sort in all cases is O(nlogn)

5. Quick Sort :- Quick sort is implemented based on the principle of **Divide and Conquer** algorithm.

Divide and conquer algorithm is implemented using 3 steps.

1.Divide 2.Conquer 3.Combine or merge

1.Divide:- Select the first element of an array as pivot element. Divide the given array 2 parts such that the left part contains the elements which are less than the pivot element and right part contains the elements which are greater than the pivot element. This arrangement process is called as **Partitioning.**

2.Conquer:- The left part and right part will be sorted recursively, so that the entire array will be in sorted order.

3.Combine:- There is no need of combine or merge step in quick sort.

Procedure for partitioning:-

Process:

1. Initialize pivot element as first element in the array to be sort.
2. Initialize i as starting index of the array to be sort, j as ending index of the array to be sort.
3. Do the following steps while **i < j**
    1. Repeatedly move the i to right (i.e. increase the i value) while (a[i] <= pivot) i.e. all the elements to Left of pivot element are Less than pivot element.
    2. Repeatedly move the j to left (i.e. decrease the j value) while (a[j] > pivot)

i.e. all the elements to Right of pivot element are Greater than pivot element.

3. If i < j, then swap a[i] and a[j]

4. If i<j is false, then j becomes pivot element position so swap a[j] and a[low].

How to select Pivot element in Quick Sort:-

Quick sort is implemented by selecting an element of an array as pivot element.
There are 3 approaches to select the pivot element.
1. Selecting 1st element as pivot element
2. Selecting a random element as pivot element
3. Selecting Median of 3 elements as pivot element

1. Selecting 1st element as pivot element:- The popular approach is to select first element as pivot element. This is acceptable if the input is random but if the input is either in ascending or descending order then it provides a poor partition.

2.Selecting a random element as pivot element:- Another approach is simply choose the pivot element randomly. On the other hand random number generator is generally an expensive approach and doesn't reduce the average running time of the algorithm.

3. Selecting Median of 3 elements as pivot element:- Another approach is simply choose median of first, last and middle element of the array as pivot element.

C program to implement Quick Sort:-

```
#include<stdio.h>
#include<conio.h>
int a[20],n;
int  partion(int l,int h);
void quicksort(int low,int high);
void main()
{
        int i;
        clrscr();
        printf("enter no array of elements to sort\n");
        scanf("%d",&n);
        printf("enter  array of elements to sort\n");
        for(i=0;i<n;i++)
        scanf("%d",&a[i]);
        quicksort(0,n-1);
        printf("elements after quicksort \n");
        for(i=0;i<n;i++)
```

```
        printf("%d ",a[i]);
        getch();

}
void quicksort(int  low,int high)
{
        int j;
        if(low<high)
        {
                j=partion(low,high);
                quicksort(low,j-1);
                quicksort(j+1,high);

        }
}
int partion(int l,int h)
{
        int i=l,j=h;
        int pivot,temp;
        pivot=a[i];
        while(i<j)
        {
                while(a[i]<=pivot)
                        i++;
                while(a[j]>pivot)
                        j--;
                if(i<j)
                {
                        temp=a[i];
                        a[i]=a[j];
                        a[j]=temp;
                }
        }
        temp=a[j];
        a[j]=pivot;
        a[l]=temp;
        return j;
}
```

Analysis of Quick Sort:-

1.Best Case Time Complexity:- In Quick sort, the best case will occur when we divide the given array into exactly 2 half's.

If n=1, then we can get the solution directly.
So, T(n)=1 if n=1

If n>1, then the given array is divided into 2 parts and we need minimum ‘n’ comparisons

$T(n)=2T(n/2) + n$ if $n>1$

$T(n)=2T(n/2)+n \Rightarrow 2^1 T(n/2^1)+1.n$

$T(n)=2[2T(n/4)+n/2]+n$

$=4T(n/4)+2n \Rightarrow 2^2T(n/2^2)+2.n$

$T(n)=4[2T(n/8)+n/4]+2n$

$=8T(n/8)+3n \Rightarrow 2^3T(n/2^3)+3.n$

.

.

.

$T(n)=2^kT(n/2^k)+k.n$

Let $2^k=n$

Apply log on both sides then $\log 2^k=\log n$

$K\log 2=\log n$

$k.1=\log n$

$k=\log n$

$T(n)=2^kT(n/2^k)+k.n$

$=n.T(n/n)+\log n.n$

$=T(1)+n \log n$

$=1+n \log n$

$=n \log n$

So, the best case time complexity of Quick sort sort =O(nlogn)

2.Averge Case time complexity=O(nlogn)

3.Worst Case time complexity:- The worst case will occurs when pivot element is either **minimum** element or **maximum** element.

Let **‘i’** be the pivot element (smallest element of the array) and assumes that i index location is 1.

$T(n)=T(i-1)+T(n-i)+n$

$=T(1-1)+T(n-1)+n$

$=T(0)+T(n-1)+n$

$T(n)=T(n-1)+n$

$T(n-1)=T(n-2)+n-1$

$T(n-2)=T(n-3)+n-2$

.

.

.

.

1 2 3 4. . . . . $n=n(n+1)/2=n^2$

Worst case time complexity of quick sort=$O(n^2)$

Time Complexities of Different Searching and Sorting Algorithms

Algorithm	Time Complexity		
	Best	Average	Worst
Linear Search	O(1)	O(n)	O(n)
Binary Search	O(1)	O(log n)	O(log n)
Bubble Sort	O(n)	$O(n^2)$	$O(n^2)$
Insertion Sort	O(n)	$O(n^2)$	$O(n^2)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Merge Sort	O(n logn)	O(n logn)	O(n logn)
Quick Sort	O(n logn)	O(n logn)	$O(n^2$
Bucket Sort	O(n+b)	O(n+b)	O(n+b)
Shell Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$