UNIT-1

Introduction

Data Structures:

A data structures is basically a group of data elements that are put together under one name, and which defines a particular way of storing and organizing data in a computer. So that it can be used efficiently.

Classification of Data structures:

Data structures are generally categorized into two classes

1. Primitive Data Structures
2. Non-primitive Data Structures.

Primitive Data Structures:

Primitive data structures are the fundamental data types which are supported by a programming language. Some basic data types are integer, real, character, Boolean.

Non-primitive Data Structures:

Non-primitive data structures are those data structures which are created using primitive data structures.

Examples: stacks, Queues, linked lists, trees, graphs.

Linear Data Structures:

If the elements of a data structures are stored in a linear or sequential order, then it is called linear data structures.

Ex: stacks, Queues, Linked lists.

Non-linear Data Structures:

If the elements of a data structures are not stored in a linear order, then it is called non-linear data structures.

Ex: Trees, Graphs.

Operations on Data Structures:

Different operations that can be performed on the various data structures

1. ***Traversing:*** to access each data item exactly once. So that it can be processed.
2. ***Searching:*** to find the location of one or more data items that satisfy the given constraints, such a data item may or may not be present in the given collection of data items.
3. ***Inserting:*** to add data items to the given list of data items.

4. ***Deleting:*** to remove (delete) a particular data item from the given collection of data items.
5. ***Sorting***: data items can be arranged in some order like ascending or descending order.
6. ***Merging:*** list of two sorted data items can be combined to form a single list of sorted data items.

Abstract Data Type (ADT):

Abstract datatype is a type for objects whose behavior is defined by a set of value and a set of operations.

The definition of ADT only mention what operations are to be performed but not how these operations will be implemented. It does not specify how data will be organized in memory and what algorithms will be used for implementing operations.

Example: Array ADT

For all A∈Array, i∈index, x∈item, j, size∈integer

Array Create (j, list) := return an array of j dimension

Item Retrieve (A, i) := if(i∈index)

return item associated with index value I in an array A

else return error

Array Store (A, i, x) := if(i∈index)

return an array that is identical to array A except the new pair (i, x) has been inserted.

else return error.

end Array.

Algorithm:

An algorithm is a finite set of instructions. In addition, all algorithms must satisfy the following criteria

1. **Input:** Zero or more quantities are externally supplied.
2. **Output:** At least one quantity is produced.
3. **Definiteness:** Each instruction is clear and Un-ambiguous.
4. **Finiteness:** If we trace out the instructions of an algorithm, then for all cases the algorithm terminates after a finite number of steps.
5. **Effectiveness:** Each instruction must be very basic.

Performance Analysis:

The efficiency of an algorithm can be decided by measuring the performance of an algorithm.

We can measure the performance by computing two factors:

1. Space Complexity.
2. Time Complexity.

Space complexity:

The space complexity of an algorithm is the amount of memory required to run to completion.

To compute this space complexity we use two factors:

1. fixed part
2. variable part

fixed part:

A fixed part that is independent of the characteristics of the inputs and outputs. This part includes the instruction space(space for code), space for simple variables and fixed size component variables, space for constants and so on.

Variable part:

A variable part that consists of the space needed by component variables whose size is dependent on the particular problem instance being solved, the space needed by the reference variables and recursion stack space.

Space requirement s(p) of any algorithm p can be given as

$$s(p) = c + Sp$$

where c is a constant i.e., fixed part Sp is a space depend upon instance characteristics .

Time Complexity:

Time complexity of an algorithm is amount of computer time required by an algorithm to run to completion.

Time T(P) taken by a program P is the sum of the compile time and the run time.

Time complexity Examples :-

1. **Sum of an array elements**

	Statement	s/e	frequency	Total steps
1	Algorithm sum(a, n)	0	0	0
2	{	0	0	0
3	s := 0;	1	1	1
4	for i := 1 to n do	1	n+1	n+1
5	s = s + a[i];	1	n	n
6	return s;	1	1	1

7	}	0	0	0
Total =		2n+3		

2. Matrix addition

	Statement	s/e	frequency	Total steps
1	Algorithm add(a, b, c, m, n)	0	0	0
2	{	0	0	0
3	for i := 1 to m do	1	m+1	m+1
4	for j := 1 to n do	1	m(n+1)	m(n+1)
5	C[i, j] = a[i, j] + b[i, j];	1	mn	mn
6	}	0	0	0
Total =		m+1+mn+m+mn= 2mn+2m+1		

3. Matrix Multiplication

	Statement	s/e	frequency	Total steps
1	Algorithm mul(a, b, c, n)	0	0	0
2	{	0	0	0
3	for i := 1 to n do	1	n+1	n+1
4	for j := 1 to n do	1	n(n+1)	n^2+n
5	{	0	0	0
6	c[i , j] = 0;	1	n^2	n^2
7	for k := 1 to n do	1	$n^2(n+1)$	n^3+n^2
8	c[i, j] = c[i , j] + a[i, k] * b[k, j];	1	n^3	n^3
9	}	0	0	0
10	}	0	0	0
Total =		$2n^3+3n^2+2n+1$		

4. Recursive sum of array elements

	Statement	s/e	frequency n=0	n>0	Total steps n=0	n>0
1	Algorithm Rsum(a, n)	0	0	0	0	0
2	{	0	0	0	0	0
3	if (n <= 0) then	1	1	1	1	1
4	return 0;	1	1	0	1	0
5	else	0	0	0	0	0
6	return Rsum(a, n-1) + a[n];	1+x	0	1	0	1+x
7	}	0	0	0	0	0
Total =					=2	=2+x

where x = Trsum(n-1)

$$Trsum(n) = \begin{cases} 2 & \text{if } n=0 \\ 2+Trsum(n-1) & \text{if } n>0 \end{cases}$$

Trsum(n) = 2 + Trsum(n-1)
=2 + 2 + Trsum(n-2) = 2.2 + Trsum(n-2)
=2.2 + 2 + Trsum(n-3) = 3.2 + Trsum(n-3)

n times

=n.2 + Trsum(0) = n.2 + 2
=2n + 2.

ASYMPTOTIC NOTATIONS :

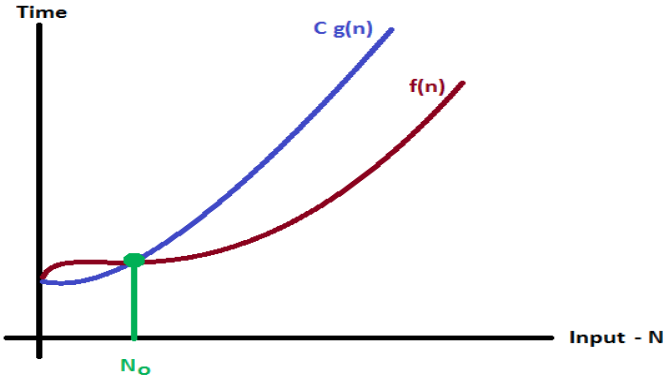
- To choose best algorithm we use space complexity and time complexity.
- There are 3 types . There are

1. Big - Oh (O)
2. Big - Omega (Ω)
3. Big - Theta (Θ)

Big 'O' notation: (Upper bound)

The function f(n)=O(g(n)) if and only if there exists positive constants c and n_0 such that
f(n) <= c*g(n) for all n, n>=n_0

Diagram :-



Examples:

1) Let f(n) = 3n+2
3n + 2 <= 4 * n for all n, n>=2
Here c=4 g(n) = n and n_0 = 2
Therefore, f(n) = O(n)

2) Let f(n) = $10n^2+4n+2$
$10n^2+4n+2$ <= 11 * n^2 for all n, n >= 5
Here c=11 g(n) = n^2 and n_0 = 5
Therefore, f(n) = $O(n^2)$

3)) Let f(n) = $2n^3+4n^2+5n+2$
$2n^3+10n^2+4n+2$ <= 3 * n^3 for all n, n >= 6
Here c= 3 g(n) = n^3 and n_0 = 6
Therefore, f(n) = $O(n^3)$

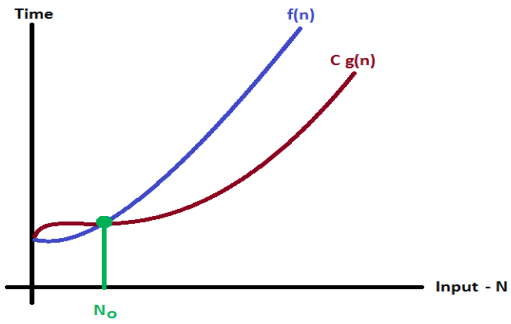
ORDER OF GROWTH IS AS FOLLOWS
$O(1) < O(\log n) < O(n) < O(n^* \log n) < O(n^2) < O(n^3) < O(2^n)$

OMEGA 'Ω' NOTATION: (Lower bound)

The function f(n) = Ω(g(n)) if and only if there exists positive constants c and n_0 such that

f(n)>=c*g(n) for all n, n>=n

Diagram :



Examples:

1) Let f(n) = 3n+2

3n + 2 >= 3 * n for all n, n>=1

Here c=3 g(n) = n and $n_0 = 1$

Therefore, f(n) = Ω(n)

2) Let f(n) = $10n^2+4n+2$

$10n^2+4n+2 >= 10 * n^2$ for all n, n >= 1

Here c=10 $g(n) = n^2$ and $n_0 = 1$

Therefore, $f(n) = \Omega(n^2)$

3)) Let f(n) = $2n^3+4n^2+5n+2$

$2n^3+10n^2+4n+2 >= 2 * n^3$ for all n, n >= 1

Here c= 2 $g(n) = n^3$ and $n_0 = 1$

Therefore, $f(n) = \Omega(n^3)$

THE ORDER OF GROWTH IS AS FOLLOWS

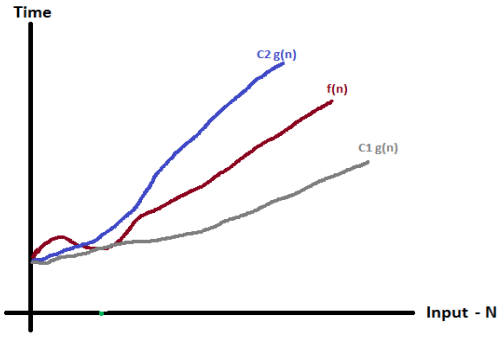
$\Omega(1) < \Omega(\log n) < \Omega(n) < \Omega(n^* \log n) < \Omega(n^2) < \Omega(n^3) < \Omega(2^n)$

THETA 'Θ' NOTATION : (Upper bound and Lower bound)

The function f(n) = Θ(g(n)) if and only if there exists positive constants c_1, c_2 and n_0 such that

$$c_1*g(n) <= f(n) <= c_2*g(n) \text{ for all } n, n>=n0$$

Diagram:



Examples:

1) Let f(n) = 3n+2

$3 * n <= 3n + 2 <= 4 * n$ for all n, n>=2

Here $c_1=3$ $c_2 = 4$ $g(n) = n$ and $n_0 = 2$

Therefore, $f(n) = \Theta(n)$

2) Let $f(n) = 10n^2+4n+2$

$10 * n^2 <= 10n^2+4n+2 <= 11 * n^2$ for all n, n >= 5

Here $c_1= 10$ $c_2 = 11$ $g(n) = n^2$ and $n_0 = 5$

Therefore, $f(n) = \Theta(n^2)$

3)) Let $f(n) = 2n^3+4n^2+5n+2$

$2 * n^3 <= 2n^3+10n^2+4n+2 <= 3 * n^3$ for all n, n >= 6

Here $c_1= 2$ $c_2 = 3$ $g(n) = n^3$ and $n_0 = 6$

Therefore, $f(n) = \Theta(n^3)$

THE ORDER OF GROWTH IS AS FOLLOWS

Θ(1) < Θ(log n) < Θ(n) < Θ(n* log n) < Θ(n^2) < Θ(n^3) < Θ(2^n)

Basic Efficiency Classes

Class	Name
1	Constant
logn	logarithmic
n	linear
nlogn	n-log-n
n^2	Quadratic
n^3	cubic
2^n	exponential

Searching

Linear or sequential search:

This is a straight forward algorithm that searches for a given item(k) in a list of n elements by checking successive elements of the list until either a match with the search element(k) is found or the list is over.

Iterative algorithm:

```
Algorithm seqsrch(A[0…n-1], k)
{
        i=0;
        while((i<n)&&(a[i]≠k) do
                i=i+1;
        if(i<n) return i;
        else return -1;
}
```

Recursive algorithm:

```
Algorithm recseqsrch(int a[20] , int n, int k)
{
        n=n-1;
        if(n>=0)
        {
                if(a[n]= =k)  return n;
                else return recseqsrch(a, n, k);
        }
        else
                return -1;
}
```

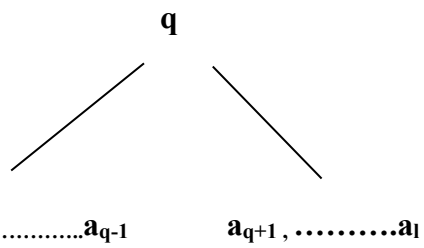
Binary Search:

- Let a[i], 1<= i <= n be a list of elements that are sorted in non-descending order.
- Suppose if we have 'n' number of elements and x is the element to be searched.
- If n=1, small(p) be true then there is no requirement of applying divide and conquer, if n > 1 then it can be divided into new sub problems.
- Pick an index 'q' = (i+l)/2 in the range[i , l] and compare x with a[q] then the following 3 possibilities will occur,

  I. If x=a[q], then search element found,

  II. If x<a[q], then is searched in left sub list a[i], a[i+1],…,a[q-1],

  III. If x>a[q], then x is searched in right sub list a[q+1], a[q+2], …., a[l]



Binary Search (Iterative Algorithm)

```
Algorithm binsearch ( a, n, x )
{
        low := 1; high := n;
        while( low <= high ) do
        {
                mid := (low + high)/2;
                 if ( x < a[mid] ) then high := mid-1;
                else if ( x > a[mid] ) then low := mid + 1;
                else return mid;
        }
        return 0;
}
```

Binary Search (recursive Algorithm)

```
Algorithm Recbinsearch ( a, i, l, x )
{
        if ( i = l ) then //small(p)
        {
                if ( a[i] = x ) then return i;
                else return 0;
        }
        else
        {
                mid = ( i+l )/2;
                if (x < a[mid] ) then return Recbinsearch ( a, i, mid-1, x );
                else if ( x > a[mid] ) then return Recbinsearch ( a, mid+1, l, x );
                else return mid;
        }
}
```

EXAMPLE:

Consider the list n=14 where a={-15, -6, 0, 7, 9, 23, 54, 82, 101, 112, 125, 131, 142, 151}

index	1	2	3	4	5	6	7	8	9	10	11	12	13	14
a[index]	-15	-6	0	7	9	23	54	82	101	112	125	131	142	151

Using binsearch iterative algorithm

X=151(key element which has to be find)

Low	high	mid
1	14	(1+14)/2 = 7
8	14	(8+14)/2 = 11
12	14	(12+14)/2 = 13
14	14	(14+14)/2 = 14 found

X= -14 (key element which has to be find)

Low	high	mid
1	14	(1+14)/2 = 7
1	6	(1+6)/2 = 3
1	2	(1+2)/2 = 1
2	2	(2+2)/2 = 2
2	1	not found

X=9 (key element which has to be find)

Low	high	mid
1	14	(1+14)/2 = 7
1	6	(1+6)/2 = 3
4	6	(4+6)/2 = 5 found

Using Recbinsearch algorithm

1) X=151(key element which has to be find)

Call Recbinsearch (a, 1, 14, 151)

Call Recbinsearch (a, 8, 14, 151)

Call Recbinsearch (a, 12, 14, 151)

Call Recbinsearch (a, 14, 14, 151)

return 14

2) X= -14 (key element which has to be find)

Call Recbinsearch (a, 1, 14, -14)

Call Recbinsearch (a, 1, 6, -14)

Call Recbinsearch (a, 1, 2, -14)

Call Recbinsearch (a, 2, 2, -14)

return 0

3) X=9 (key element which has to be find)

Call Recbinsearch (a, 1, 14, 9)

Call Recbinsearch (a, 1, 6, 9)

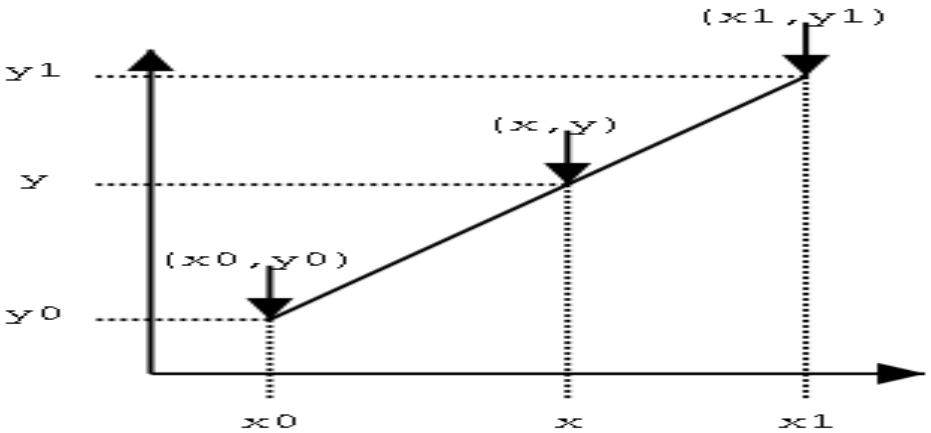
Call Recbinsearch (a, 4, 6, 9)

Call Recbinsearch (a, 5, 5, 9)

return 5

Interpolation Search:

Interpolation search is similar to binary search. First, we take the list of elements in sorting order. The left most element y0 (a[x0]) and the right most element y1(a[x1]). The straight line passing through (x0, y0) and (x1, y1).



The line equation is $\frac{x-x0}{x1-x0} = \frac{y-y0}{y1-y0}$

The search element y is compared with the element whose index is computed as (round off) the X-coordinate of the point on the straight line. So the point (x, y) is on the line.

$$x = x0 + (y - y0) \left(\frac{x1-x0}{y1-y0}\right)$$

```
Algorithm interpolation_search (a[1….n], int y)
{
        low = 1;
        high = n;
        while (low ≤ high) do
        {
                x = x0 + (y - y0) (x1−x0 / y1−y0)
                if (y < a[x])
                        high = x-1;
                else if (y > a[x])
                        low = x+1;
                else return x;
        }
        return -1;
}
```

Fibonacci Search:

Fibonacci search is similar to Binary search and also work on sorted array. The split araay to be searched by using Fibonacci numbers.

```
Algorithm Fibonacci_search ( A, n, key)
{
        low = 0; high = n-1; loc = -1; flag = 0;
        while ((flag != 1) && (low <= high)) do
        {
                //get Fibonacci number < n
                index = get_fibonacci(n);
                if (key = = A[index + low]) then
                {
                        flag = 1;
                        loc = index + low;
                        break;
                }
                else if (key > A[index + low]) then
                        low = low + index + 1;
```

```
            else
                    high = low + index – 1;
            n = high – low +1;
        }
        if (flag = = 1) then
            return loc;
        Else
            return -1;
}
```

Example:

Consider the list n=10

index	0	1	2	3	4	5	6	7	8	9
a[index]	10	34	39	45	53	58	66	75	83	88

index	0	1	2	3	4	5	6	7	8	9
Fibonacci[index]	0	1	1	2	3	5	8	13	21	34

key = 88

low	high	loc	flag	index	n
0	9	-1	0	--	10
9	9	-1	0	8	1
		9	1	0	**return 10**

Key = 10

low	high	loc	flag	index	n
0	9	-1	0	--	10
0	7	-1	0	8	8
0	4	-1	0	5	5
0	2	-1	0	3	3
0	1	-1	0	2	2

0	0	-1	0	1	1
		0	1	0	**return 0**

Key = 35

low	high	loc	flag	index	n
0	9	-1	0	--	10
0	7	-1	0	8	8
0	4	-1	0	5	5
0	2	-1	0	3	3
0	1	-1	0	2	2
2	1	-1	0	1	0
					return -1

sorting

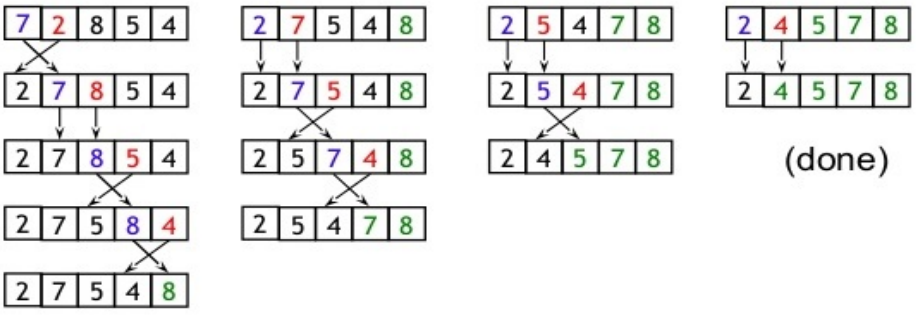
sort:

The given list of elements are arranged in sorting (Ascending) order.

bubble Sort:

In bubble sort to compare adjacent elements of the list and exchange them if they are not in order. By doing it repeatedly, we end up “bubbling up” the largest element to the last position on the list. In the next pass bubbling up the second largest element and so on. After n-1 passes the list is sorted.

Example:



```
Algorithm Bubblesort (A[0…..n-1])
{
        for i = 0 to n-2 do
                for j = 0 to n-2-i
                        if (A[j] > A[j+1]) then
                        {
                                t = A[j];
                                A[j] = A[j+1];
                                A[j+1] = t;
                        }
}
```

Let C(n) = the number of comparisons required for algorithm

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=0}^{n-2-i} 1 = \sum_{i=0}^{n-2} (n-2-i) - 0 + 1$$

$$= \sum_{i=0}^{n-2} (n-1-i)$$

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

$$= \frac{n(n-1)}{2} \in O(n^2)$$

Bubble Sort Program:

```
#include<stdio.h>
void bubblesort();
main()
{
        int n, i, a[100];
        clrscr();
        printf("\nEnter the size of array:");
        scanf("%d",&n);
        printf("\nEnter the elements:\n");
        for(i=0;i<n;i++)
                scanf("%d",&a[i]);
        bubblesort(a,n);
        printf("\nafter sorting elements are:\n");
        for(i=0;i<n;i++)
                printf("%d\t",a[i]);
        getch();
}
void bubblesort(int a[],int n)
{
        int i,j,t;
        for(i=0;i<n-1;i++)
```

```
                for(j=0;j<n-1-i;j++)
                {
                        if(a[j]>a[j+1])
                        {
                                t=a[j];
                                a[j]=a[j+1];
                                a[j+1]=t;
                        }
                }
}
```

Section Sort:

We start the selection sort by scanning the entire given list to find its smallest element and exchange it with the first element, putting the smallest element in its original position in the sorted list. Then we scan the list, starting with the second element, to find the smallest element among the last n-1 elements and exchange it with the second element, after n-1 passes the list is sorted.

Example:

Let us consider the following example with 9 elements to analyze selection Sort:

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*Remarks*
65	70	75	80	50	60	55	85	45	find the first smallest element
i								j	swap a[i] & a[j]
45	70	75	80	50	60	55	85	65	find the second smallest element
	i			j					swap a[i] and a[j]
45	**50**	75	80	70	60	55	85	65	Find the third smallest element
		i				j			swap a[i] and a[j]
45	**50**	**55**	80	70	60	75	85	65	Find the fourth smallest element
			i		j				swap a[i] and a[j]
45	**50**	**55**	**60**	70	80	75	85	65	Find the fifth smallest element
				i				j	swap a[i] and a[j]
45	**50**	**55**	**60**	**65**	80	75	85	70	Find the sixth smallest element
					i			j	swap a[i] and a[j]
45	**50**	**55**	**60**	**65**	**70**	75	85	80	Find the seventh smallest element
						i j			swap a[i] and a[j]
45	**50**	**55**	**60**	**65**	**70**	**75**	85	80	Find the eighth smallest element
							i	J	swap a[i] and a[j]
45	**50**	**55**	**60**	**65**	**70**	**75**	**80**	**85**	The outer loop ends.

```
Algorithm selectionSort (A[0…..n-1])
{
        for i = 0 to n-2 do
        {
                min = i;
                for j = i+1 to n-1 do
                        if (A[j] < A[min]) then
                                min = j;
                t = A[min];
                A[min] = A[i];
                A[i] = t;
        }
}
```

C(n) = the number of comparisons required for algorithm.

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} (n-1-(i+1)+1$$

$$= \sum_{i=0}^{n-2} (n-1-i)$$

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

$$= \frac{n(n-1)}{2} \in O(n^2)$$

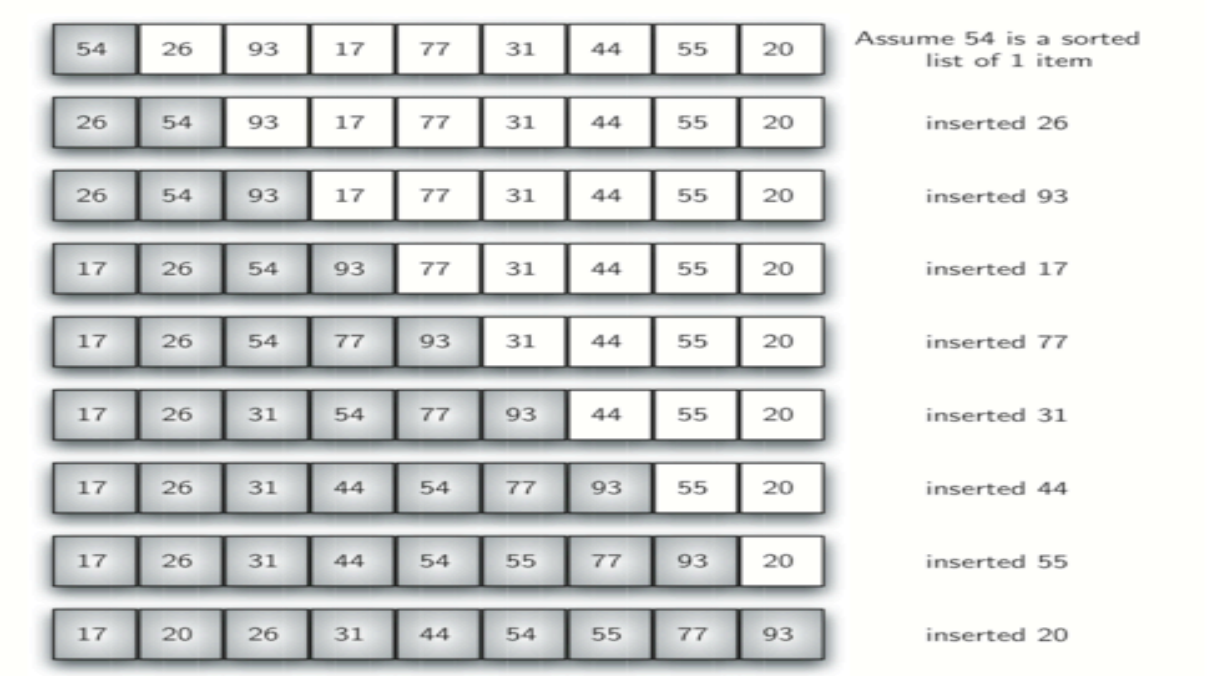
Program:

```
#include<stdio.h>
void selectionsort();
main()
{
        int n,i,a[100];
        clrscr();
        printf("\nEnter the size of array:");
        scanf("%d",&n);
        printf("\nEnter the elements:");
        for(i=0;i<n;i++)
                scanf("%d",&a[i]);
        selectionsort(a,n);
        printf("\nafter sorting elements are:\n");
        for(i=0;i<n;i++)
                printf("%d\t",a[i]);
        getch();
}
void selectionsort(int a[],int n)
{
```

```
        int i,j,t,min;
        for(i=0;i<n-1;i++)
        {
                min=i;
                for(j=i+1;j<n;j++)
                        if(a[j]<a[min])
                                min=j;
                t=a[min];
                a[min]=a[i];
                a[i]=t;
        }
}
```

Insertion Sort:

This is an in-place comparison-based sorting algorithm. Here, a sub-list is maintained which is always sorted. For example, the lower part of an array is maintained to be sorted. An element which is to be 'inserted' in this sorted sub-list, has to find its appropriate place and then it has to be inserted there. Hence the name, **insertion sort.**



```
Algorithm InsertionSort (A[0….n-1])
{
        for i = 1 to n-1 do
        {
                v = a[i];
                j = i-1;
                while ((j >= 0) && (A[j] > v)) do
                {
                        A[j+1] = A[j];
                        j = j-1;
                }
                A[j+1] = v;
        }
}
```

C(n) = The number of comparisons required for algorithm

$C_{worst}(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1 = \sum_{i=1}^{n-1} (i-1) - 0 + 1$

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

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

$= \frac{n(n-1)}{2} \in O(n^2)$

$C_{best}(n) = \sum_{i=1}^{n-1} 1 = (n-1) - 1 + 1$

$= (n-1) \in O(n)$

$C_{avg}(n) = \frac{n2}{4} \in O(n^2)$

Program:

```
#include<stdio.h>
void insertionsort();
main()
{
        int i,n,a[10];
        clrscr();
        printf("\nEnter the size of array:");
        scanf("%d",&n);
        printf("\nEnter the elements:\n");
        for(i=0;i<n;i++)
                scanf("%d",&a[i]);
```

```
        insertionsort(a,n);
        printf("\nafter sorting elements are:\n");
        for(i=0;i<n;i++)
                printf("%d\t",a[i]);
        getch();
}
void insertionsort(int a[],int n)
{
        int t,i,j;
        for(i=1;i<n;i++)
        {
                t=a[i];
                for(j=i-1;j>=0;j--)
                        if(a[j]>t)
                                a[j+1]=a[j];
                        else
                                break;
                a[j+1]=t;
        }
}
```

Quick Sort:

Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays. A large array is partitioned into two arrays one of which holds values smaller than the specified value, say pivot, based on which the partition is made and another array holds values greater than the pivot value.

Quicksort partitions an array and then calls itself recursively twice to sort the two resulting subarrays. This algorithm is quite efficient for large-sized data sets as its average and worst-case complexity are O(n2), respectively.

RULES:

1. Add +∞ to end of the list. That is a[n+1] = +∞
2. Pivot element P = a[1]
3. up = 1 and down = n+1.
4. First up is increased by 1until a[up] is greater than or equal to pivot element. If a[up] is greater than or equal to the pivot element considered index as ‘up’ and First down is decreased by 1 until a[down] is less than or equal to pivot element. If a[down] is less than or equal to the pivot element considered index as ‘down’.
5. if (up < down) then interchange array values a[up] & a[down] and proceed step 4. Otherwise (up >= down) then interchange pivot and a[down] this procedure divides the list into two sub lists a[1].....a[down-1] and a[down+1].....a[n].

6. Apply same procedure for sub lists which produces the list in sorted order.

Example:

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*Remarks*
38	08	16	06	79	57	24	56	02	58	04	70	45	
pivot				up						down			swap up & down
pivot				04						79			
pivot					up			down					swap up & down
pivot					02			57					
pivot						down	up						swap pivot & down
(24	08	16	06	04	02)	**38**	(56	57	58	79	70	45)	
pivot					down	up							swap pivot & down
(02	08	16	06	04)	**24**								
pivot, down	up												swap pivot & down
02	(08	16	06	04)									
	pivot	up		down									swap up & down
	pivot	04		16									
	pivot		down	Up									
	(06	04)	**08**	(16)									swap pivot & down
	pivot	down	up										
	(04)	**06**											swap pivot & down
	04 pivot, down, up												
				16 pivot, down, up									
(02	**04**	**06**	**08**	**16**	**24)**	38							

QUICKSORT ALGORITHM:

```
#include<stdio.h>
#include<conio.h>
void quicksort();
int partition();
int n,a[100];
```

```
void main()
{
	int n,i;
	clrscr();
	printf("\nEnter the size of array:");
	scanf("%d",&n);
	printf("\nEnter the elements:\n");
	for(i=1;i<=n;i++)
		scanf("%d",&a[i]);
	a[i]=999;
	quicksort(1,n);
	printf("\nafter sorting elements are:\n");
	for(i=1;i<=n;i++)
		printf("%d\t",a[i]);
	getch();
}
void quicksort(int p,int q)
{
	int j;
	if(p<q)
	{
		j=partition(p,q+1);
		quicksort(p,j-1);
		quicksort(j+1,q);
	}
}
int partition(int m,int p)
{
	int v,i,j,t;
	v=a[m];
	i=m;
	j=p;
	do
	{
		do
		{
			i++;

		}while(a[i]<v);
		do
```

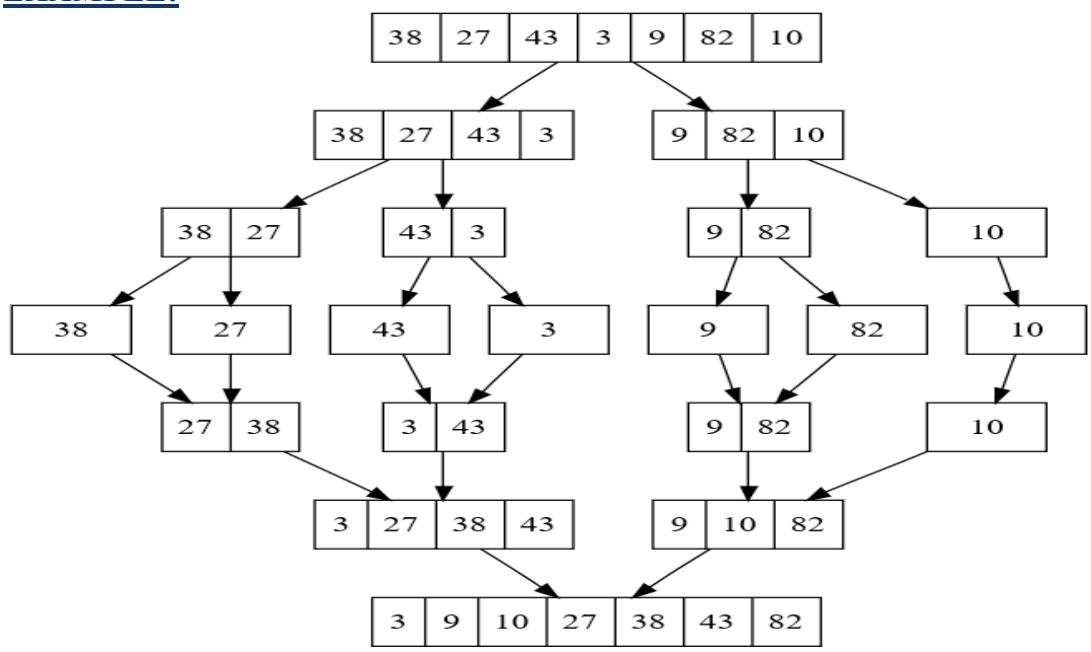
```
                {
                        j--;
                }while((j>=0)&&(a[j]>v));
                if(i<j)
                {
                        t= a[j];
                        a[j]=a[i];
                        a[i]=t;
                }
        }while(i<=j);
        a[m]=a[j];
        a[j]=v;
        return j;
}
```

Merge Sort:

The elements are to be sorted in non-decreasing order. Given a sequence of n elements. The general idea is to imagine them split into two equal size sets. Each set is individually sorted and the resulting sorted sequences are merged to produce a single sorted sequence of n elements.

The divide-and-conquer strategy in which the list is splitting into two equal subsets and sorts them and combines operation is the merging of two sorted sub lists into a single sorted list.

EXAMPLE:



```
Algorithm mergsort ( low, high )
{
        if( low < high )
        {
                mid := ( low + high )/2;
                mergesort ( low, mid );
                mergesort ( mid+1, high );
                merge ( low, mid, high );
        }
}

Algorithm merge( low, mid, high)
{
        i := low;
        j:= mid+1;
        k:= low;
        while((i<=mid) and (j<=high)) do
        {
                if(a[i]<=a[j]) then
                {
                        b[k]:=a[i];
                        i:=i+1;
                }
                else
                {
                        b[k]:=a[j];
                        j:=j+1;
                }
                k:= k+1;
        }
        if (i>mid) then
                for h = j to high do
                {
                        b[k]:= a[h];
                        k:= k+1;
                }
        else
                for h = i to mid do
                {
                        b[k]:= a[h];
                        k:=k+1;
                }
        for h = low to mid do
                a[h]:=b[h];
}
```

Merge Sort Program:

```
#include<stdio.h>
void mergesort();
void merge();
int a[20];
main()
{
        int i,n;
        clrscr();
        printf("\nEnter the size of array:");
        scanf("%d",&n);
        printf("\nEnter the elements:\n");
        for(i=0;i<n;i++)
                scanf("%d",&a[i]);
        mergesort(0,n-1);
        printf("\nAfter sorting elements are:\n");
        for(i=0;i<n;i++)
                printf("%d\t",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,high);
        }
}
void merge(int low,int mid, int high)
{
        int i,j,h,k,b[20];
        h=low;
        i=low;
        j=mid+1;
        while((h<=mid)&&(j<=high))
        {
                if(a[h]<=a[j])
                {
```

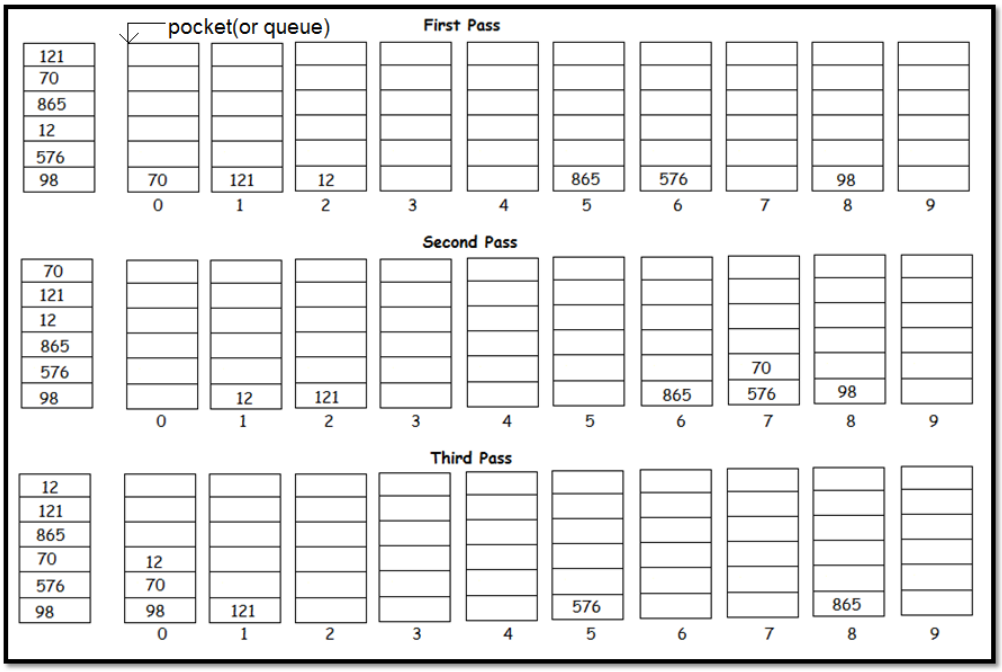
```
                b[i]=a[h];
                h++;
        }
        else
        {
                b[i]=a[j];
                j++;
        }
        i++;
    }
    if(h>mid)
        for(k=j;k<=high;k++)
        {
                b[i]=a[k];
                i++;
        }
        else
        {
                for(k=h;k<=mid;k++)
                {
                        b[i]=a[k];
                        i++;
                }
        }
    for(k=low;k<=high;k++)
        a[k]=b[k];

}
```

Radix Sort:

- Radix sort is one of the sorting algorithms used to sort a list of integer numbers in order.
- In radix sort algorithm, a list of integer numbers will be sorted based on the digits of individual numbers. Sorting is performed from **least significant digit to the most significant digit**.
- Radix sort algorithm requires the number of passes which are equal to the number of digits present in the largest number among the list of numbers.
- For example, if the largest number is a 3 digit number then that list is sorted with 3 passes.

Example:



Program:

```
#include<stdio.h>
#include<conio.h>
int largest();
void radixsort();
main()
{
        int a[20],i,n;
        printf("\nEnter the size of array:");
        scanf("%d",&n);
        printf("\nEnter the elements in array\n");
        for(i=0;i<n;i++)
                scanf("%d",&a[i]);
        radixsort(a,n);
        printf("\nthe sorted array\n");
        for(i=0;i<n;i++)
                printf("%d\t",a[i]);
        getch();
}

int largest(int a[20], int n)
{
        int large,i;
        large=a[0];
```

```
	for(i=1;i<n;i++)
		if(a[i]>large)
			large=a[i];
	return large;
}
void radixsort(int a[20],int n)
{
	int bucket[20][20],buckcoun[20];
	int i,j,k,remainder,nop,div,large,pass;
	nop=0;
	div=1;
	large=largest(a,n);
	while(large>0)
	{
		nop++;
		large=large/10;
	}
	for(pass=0;pass<nop;pass++)
	{
		for(i=0;i<10;i++)
			buckcoun[i]=0;
		for(i=0;i<n;i++)
		{
			remainder=(a[i]/div)%10;
			bucket[remainder][buckcoun[remainder]]=a[i];
			buckcoun[remainder]=buckcoun[remainder]+1;
		}
		i=0;
		for(k=0;k<10;k++)
		{
			for(j=0;j<buckcoun[k];j++)
			{
				a[i]=bucket[k][j];
				i++;
			}
		}
		div=div*10;
	}
}
```