UNIT- I

INTRODUCTION TO ALGORITHMS :-

Defination :- An Algorithm is step by step process to solve a problem, where each step indicates an intermediate task. Algorithm contains finite number of steps that leads to the solution of the problem.

Characteristics of Algorithm :-

→ Input / Output – takes one or more input and produce required output.

→ Finiteness – must terminate in countable steps.

→ Definiteness – must be stated clearly.

→ Effectiveness – Each and every step in an algorithm can be converted into programming language statement.

Categories of Algorithm :-

1. Sequence
2. Selection
3. Iteration

① Sequence :- The steps described in an algorithm are performed successively one by one without skipping any step.

eg: //adding two numbers.

Step 1 : Start.
Step 2 : read a, b
Step 3 : Sum = a+b
Step 4 : write sum
Step 5 : Stop.

② Selection :- In order to solve the problem which involve decision making or option selection, we go for selection type of algorithm.

```
if (condition)
    statement-1;
else
    statement-2;
```

eg:- // biggest among two numbers

Step 1: Start
Step 2: read a, b
Step 3: if a>b then
Step 4: write "a is greater then b"
Step 5: else
Step 6: write "b is greater then a"
Step 7: Stop.

③ Iteration :- Iteration type algorithms are used in solving the problems which involves repetition of statement. This is often called a loop.

eg :- // Sum of N numbers.

Step 1 : Start
Step 2 - Sum = 0
Step 3 - for i = 1 to N do
Sum = Sum + i
Step 4 : Stop.

Performance Analysis an Algorithm :-

The efficiency of an algorithm can be measured by the following metrics.

i) Time Complexity
2) Space complexity.

1) Time Complexity :- The amount of time required for an algorithm to complete its execution is its time complexity. An algorithm is said to be efficient if it takes the minimum amount of time to complete its execution.

(2) Space Complexity :- The amount of space occupied by an algorithm is known as space complexity. An algorithm is said to be efficient if it occupies less space and required the minimum amount of time to complete its execution.

Practice Zone

* Write an algorithm to find the largest among three different numbers.

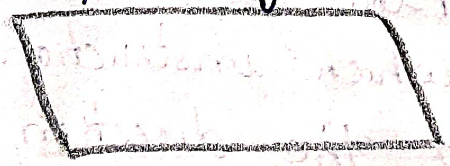
FLOWCHARTS:

Flow charts is a graphical representation of an algorithm. Programmers often use it as a program-planning tool. It makes use of symbols which are connected among them to indicate the flow of information and processing.

Basic Symbols used in flowchart designs —

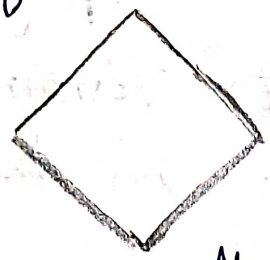
① Terminal :- The oval symbol indicates Start, Stop and Halt in a program's logic flow. ~~# pause / halt~~ Terminal is the first and last symbols in the flowchart.

② Input/Output :- A parallelogram denoted any function of input/output type. Input and output are indicated with parallelogram.



③ Processing :- A box represents arithmetic instructions. All arithmetic processes such as adding, subtracting, multiplication and division are indicated by action or process symbol

(4) Decision :- Diamond symbol represents a decision point. Decision based operations such as yes/no question or true/false are indicated by diamond in flowchart.



(5) Connectors :- Whenever flowchart becomes complex or it spreads over more than one page, it is useful to use connectors to avoid any confusions. It is represented by a circle.

(6) Flow lines :- Flow lines indicate the exact sequence in which instructions are executed. Arrow represent the direction of flow of control and relationship among different symbols of flowchart.

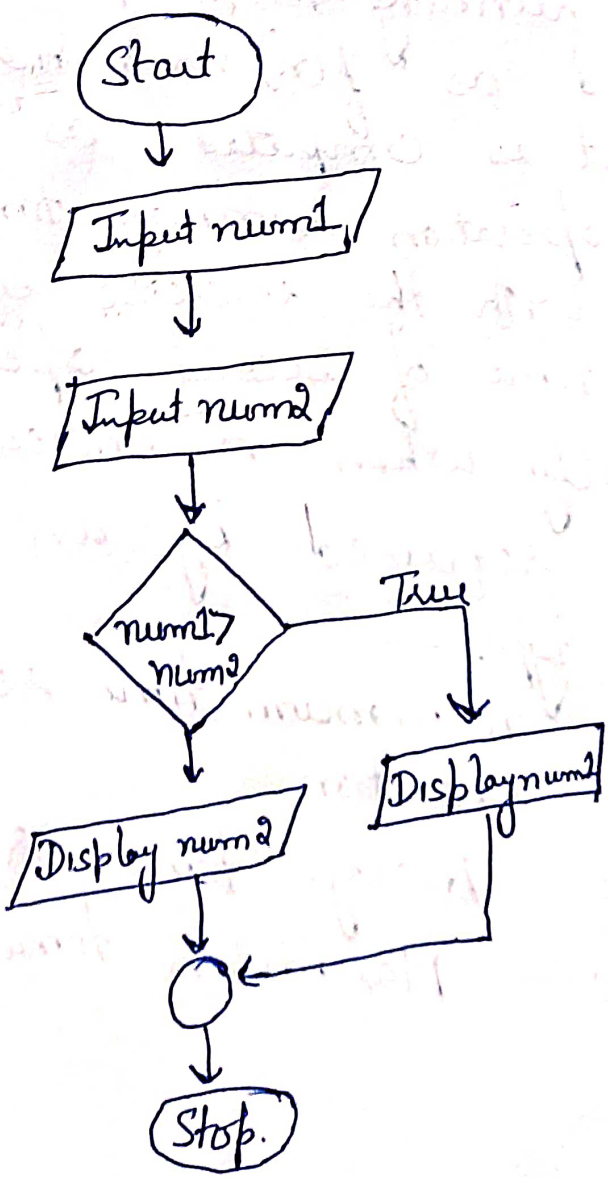
Rules for creating flowchart :

(1) Flowchart opening statement must be 'start' keyword.
(2) Flowchart ending statement must be 'end' keyword.
(3) All symbols in flowchart must be connected with arrow line.
(4) The decision symbol in the flowchart is associated with the arrow line.

Advantages of flowchart :-

1. Flowcharts are a better way of communicating the logic of the system.
2. It act as a guide for blueprint during program designed.
3. It help in debugging process.
4. Easy to understand.

Example : Draw a flowchart to input two numbers from the user and display the largest of two numbers.



Asymptotic Notations:

Asymptotic analysis of an algorithm refers to defining the mathematical boundation of its run time performance. Using asymptotic analysis, we can very well conclude the best case, average case and worst case scenario of an algorithm.

Asymptotic analysis refers to computing the running time of any operation in mathematical units of computation.

eg :- the running time of one operation is computed as $f(n)$ and may be for another operation it is computed as $g(n^2)$. This means the first operation running time will increase linearly with the increase in n and the running time of second operation will increase exponentially when n increases.

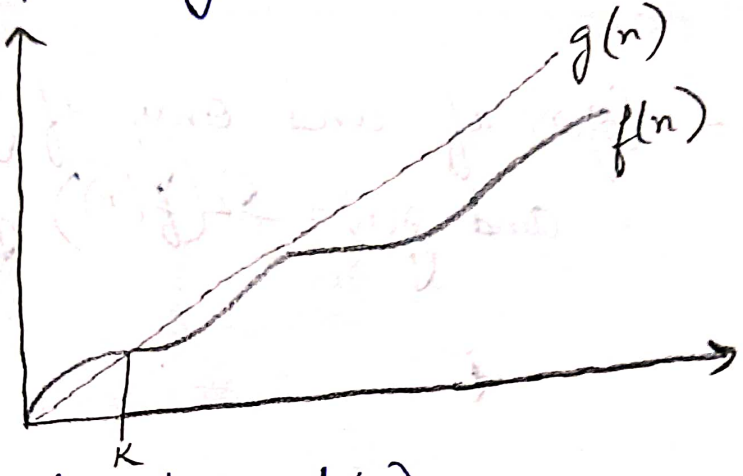
The time required by an algorithm falls under three types —

1. Best case — minimum time required for program execution.
2. Average case — Average time required.
3. Worst case — Maximum time required.

Asymptotic Notations :-

(1) Big Oh Notation, O

The notation $O(n)$ is the formal way to express the upper bound of an algorithm's running time. It measures the worst case time complexity or the longest amount of time an algorithm can possibly take to complete.



eg :- for a function $f(n)$

$O(f(n)) \leq \{g(n)$: there exists $c > 0$ and n_0 such that $f(n) < c.g(n)$ for all $n > n_0$

(2) Omega Notation Ω

→ express lower bound
→ measures best case time complexity.

eg :- for a function $f(n)$

$\Omega(f(n)) \geq \{g(n)$: there exists $c > 0$ and n_0 such that $g(n) \leq c\, f(n)$ for all $n > n_0\}$

③ Theta Notation, Θ

→ express both lower bound and upper bound.

→ average case time complexity.

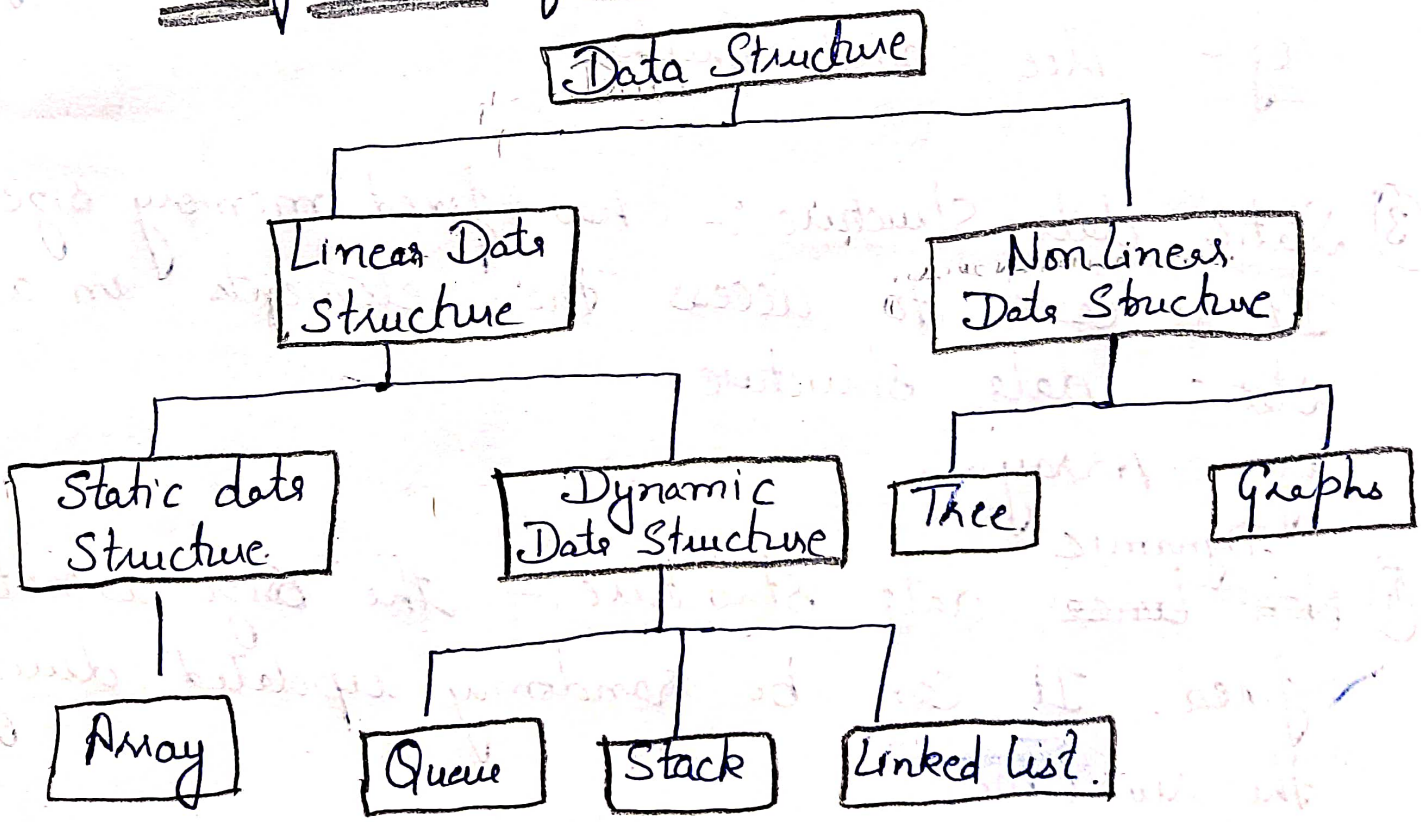
$\Theta(f(n)) = \{g(n)$ if and only if $g(n) = O(f(n))$ and $g(n) = \Omega(f(n))$ for all $n > n_0\}$

DATA STRUCTURE :-

A data structure is a storage that is used to store and organize data. It is a way of arranging data on a computer so that it can be accessed and updated efficiently.

→ Used for organizing the data.
→ Used for processing, retrieving and storing data.

Classification of Data Structure



① Linear Data Structure :- Data structure in which data elements are arranged sequentially or linearly, where each element is attached to its previous and next adjacent elements is called a linear data structure.

eg :- Array, Stack, Queue, linked list etc.

② Non- linear data structure :- where data elements are not placed sequentially or linearly.
In a non- linear data structure, we can't traverse all elements in a single run only.

eg :- Tree and Graphs.

③ Static data structure :- has fixed memory size.
It is easier to access the elements in a static data structure.

eg :- Array.

④ ~~Non- linear~~ Dynamic data structure :- the size is not fixed. It can be randomly updated during the runtime.

eg :- Queue, stack etc.

Data Structure Operations :-

(1) Traversing :- Accessing each record exactly once so that certain items in the record may be processed (Also called visiting)

(2) Searching :- Finding the location of record.

(3) Inserting :- Adding new record to the structure.

(4) Deleting :- Removing the record from the structure.

(5) Sorting :- Arranging the records in some logical order.

(6) Merging :- Combining the records in two different sorted files into a single sorted file.

Static Memory Allocation :- Static memory is allocated for declared variables by the compiler. The address can be found using the address of operator and can be assigned to a pointer. The memory is allocated during Compile time. (It uses stack for managing the static allocation of memory)

Dynamic Memory Allocation :- Memory allocation done at the time of execution is known as DMA. Functions calloc() and malloc() support allocating dynamic memory. (It uses heap for managing the dynamic allocation of memory) It is more efficient that SMA. (Memory reusability)

There are two types of available memories - stack and heap. Static Memory allocation can be done on stack whereas dynamic memory allocation can be done on both stack and heap.

There are 4 library functions provided by C defined under <stdlib.h> header file to dynamic memory allocation in C programming.

1. malloc()
2. calloc()
3. free()
4. realloc()

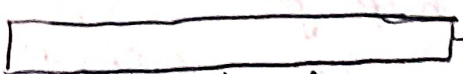
① **malloc()** :- memory allocation is used to dynamically allocate a single large block of memory with specified size.

Syntax :- ptr = (cast-type *) malloc (byte-size)

Eg :- ptr = (int *) malloc (100 * sizeof (int));

since size of int is 4 bytes, this statement will allocate 400 bytes of memory. & the pointer ptr holds the address of the first byte.

int *ptr = (int *) malloc (5 * sizeof (int));

ptr = 

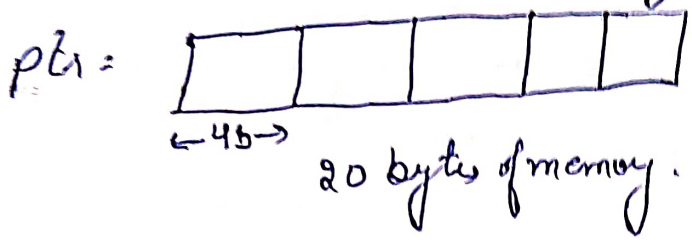
a large 20 bytes memory block is dynamically allocated

20 bytes of memory.

② calloc() method :- Contiguous allocation
→ allocates specified number of blocks of memory
→ It initializes each block with default value '0'
→ It has two parameters or arguments as compare to malloc().

Syntax :- (cast-type *) calloc (n, element-size);
no. of elements.

eg :- ptr = (float*) calloc (5, size of (int));



20 bytes of memory.

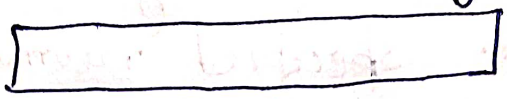
③ free() method :-
"free" method is used to dynamically de-allocate the memory.

syntax :- free(ptr);

④ realloc() method :- reallocation
→ if memory previously allocated with the help of malloc or calloc is insufficient.
→ new blocks will be initialized with default garbage value.

Syntax :- realloc (ptr, newsize);

eg:- int* ptr = (int*) malloc (5 * sizeof (int));

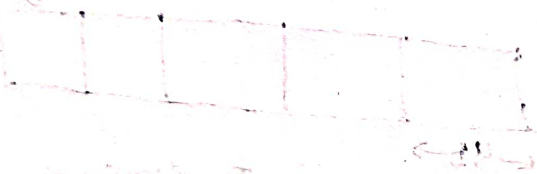
ptr = 

20 bytes.

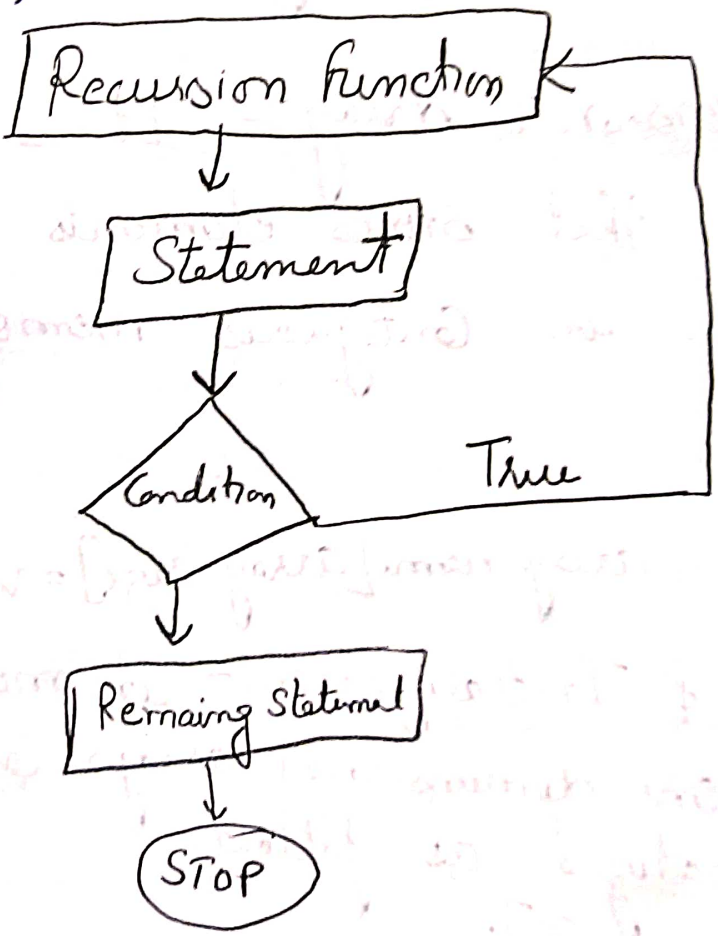
ptr = realloc (ptr, 10 * sizeof (int));

ptr = 

40 bytes.



Recursion :- Recursion is the process in which a function calls itself again and again. There must be a terminating condition to stop such recursive calls. Recursion may also be called the alternative to iteration. Recursion provides us with an elegant way to solve complex problems by breaking them down into smaller problems & with fewer lines of code than iteration.



The recursive function uses the LIFO structure just like the stack data structure.

Applications of Recursion :-

→ Tree Traversals
→ Graph Traversals.
→ Tower of Hanoi.
→ Backtracking Algorithm
→ Binary Search
→ Merge Sort, Quick sort.

Array

One dimensional Array :- It is linear data structure that stores elements of the same data type in contiguous memory location

Syntax :

datatype array-name[array-size] = value 1, value 2, ...;

Declaration & Initialization :- In most programming languages, one dimensional arrays can be declared and initialized as follow :-

```
int myArray[5];
//initialization
myarray[0] = 10;
myarray[1] = 20;
myarray[2] = 30;
myarray[3] = 40;
myarray[4] = 50;
```

Common Operations On One dimensional Array :-

1) Accessing Elements :- Elements in a one-dimensional array are accessed using their respective indices.

eg :- myarray [2] would return the value 30.

Syntax :- element = arrayname[index];

2) Insertion and Deletion :- Inserting and deleting an element in a one-dimensional array may require shifting the subsequent elements accordingly to maintain the sequential order.

Syntax :- arrayname[index] = newElement;

eg :- if we have an array named myArray and we want to insert the element at index 2 :-

```
myArray[2] = 25;
```

Deletion :- Deletion involves shifting elements after the deleted element to fill the gap.

eg :-
```
for(int i = index; i < arraysize - 1; i++)
{
  arrayName[i] = arrayName[i+1];
}
arraysize--;
```

③ Updating Elements :- Elements in an array can be updated by assigning a new value to the desired index.

Syntax :-

```
arrayName[index] = newValue;
```

eg :- `myarray[2] = 35;`

④ Traversing :- Traversing involves visiting each element of the array sequentially, allowing for various operations on each element.

Syntax :-

```
for (int i=0; i<arraysize; i++)
{
  //Access array elements using arrayname[i]
}
```

eg :-

```
printf("Elements of the array :\n");
for (int i=0; i<5; i++)
{
printf("%d", myArray[i]);
}
printf("\n");
```

Two dimensional Array :- A two dimensional array can be defined as collection of elements arranged in rows and columns.

Syntax :-

```
data_type array_name[rows][columns];
```

eg :-

```
int twodimen[4][3];
```

Declaration and Initialization :- The two dimensional array can be declared and defined as shown below :-

```
int matrix[3,4] = {
    {1,2,3,4},
    {5,6,7,8},
    {9,10,11,12}
    };
```

Common operations on Two dimensional Array :-

1) Accessing Elements :- Accessing elements in a two dimensional array involves specifying the row and column indices. To access the element at the second row and third column in the above matrix, you would use ~~matrix[1][2]~~ matrix[1][2]. Considering that array indices typically start from 0.

2) Insertion and Deletion :- Elements can also be modified or inserted using array indices.

3) Updating Elements :- Modifying the value of an existing element is a straightforward operation. Simply access the element using its indices and assign a new value.

4) Traversing :- Traversing involves visiting all rows.

5) Rows & Column operations :- operations on rows or Columns are summing the values in row or col finding the minimum or maximum element or swapping two rows/ Columns.

6) Transpose :- It involves swapping its rows with Columns.

7) Matrix Multiplication :- For two matrices A and B, the product C = A*B is computed as

$$C[i][j] = \Sigma(A[i][k] * B[k][j])$$

for all valid indices i, j and k.

Applications of 2-D Array :-

→ Matrices and Graphs
→ Image Processing
→ Board Games like chess, tic tac toe