MODULE-1: DESIGN AND ANALYSIS OF ALGORITHM

– Pearson Publication

Define Algorithm.

- Finite number of steps to solve a problem.

How algorithm works?

- Algorithm can be expressed as natural language or pseudocode.

Q. Write an algorithm to find the sum of two numbers.

step 1: Start
step 2: Input two numbers from user, num1 and num2
step 3: add the two numbers
step 4: print the result
step 5: Stop

* Types of Algorithm

1. Brute force algorithm
   - It is the simplest algorithm that is based on hit and try technique.

2. Divide and conquer algorithm
   - Main problem is divided into sub-program, and the process continues to get the smallest part of the given prob
   Finally, required output is obtained.
3. Greedy Algorithm.
   - Best solution out of all the feasible solution is obtained, and the process continue.
4. Back tracking algorithm.
   - In this desiging technique, final solution is obtained by starting once again from the initial position.

5. Dynamic algorithm
- This algorithm always starts with some initial approach.

6. Recursive algorithm
- In this desiging technique, function that call itself are involved.

* Characteristic of Algorithm

1. Well defined input
- i. Input based on the algorithm that is well defined.
- ii. There should be some input, it can be zero or any other inputs.

2. Well defined Output
- The output generated by the algorithm should be well defined and understood.
-

3. Clear and unambiguous
- The input and output of algorithm should be clear and well defined with no double meaning.

4. Finiteness
- i. The output/input generated should be finite.
- ii. The steps should be countable.

5. Language independent
- The algorithm should be in a language understood by the programmer. i.e natural language or pseudocode.

6. Feasible
- Possible solution should be there.
- The algorithm must be simple, generic and practical such that it can be executed with the available resource.

Advantages of Algorithm:

1. Easy to understand and learn.
2. Simplest way for solving a problem.
3. In an algorithm, the problem is broken into small pieces (modularity) hence, it is easier for the programmers in solving the problem.

Disadvantages:

1. Time taking process.
2. Branching and looping statements are difficult to show in algorithm.
3. Understanding complexity is different.

* Pseudo code:

- Pseudo code is an informal high level description of the operating principle of computer program or other algorithm.
- The purpose of using pseudo code is that it is easier for people to understand than conventional programming language code & that it is an efficient and environment independent description of the key principle of an algorithm.

The main constructs of Pseudo-code

1. Sequence
2. Case
3. While
4. repeat - until
5. for
6. If - then - else.

1. Sequence: Sequence represents linear tasks sequentially performed one after the other.
2. While: While a loop with a condition at its beginning

3. Repeat-until: a loop with a condition at the bottom.

4. For: for another way of looping

5. If-then-else: a conditional statement changing the flow of the algorithm.

6. Case: the generalization form of if-then-else.

Example of Pseudo-code constructs:

1. Sequence:
Input: Read Obtain Get
Ouput: Print Display Show
Compute: Compute,
Calculate Determine
Initialize: Set Init
Add: Increment Bump
Sub: Decrement

2. While
While condition
Sequence
Endwhile

3. Repeat-until
Repeat
Sequence
Until condition

4. FOR
FOR 'iteration bounds
sequence
ENDFOR

5. CASE
CASE expression OF
condition 1: Sequence 1
condition 2: Sequence 2
...
condition n: Sequence n
OTHERS:
default sequence
ENDCASE

6. IF-THEN-ELSE
IF condition THEN
Sequence 1
ELSE
sequence 2
ENDIF

Example 1:
Write a pseudo code to find whether the given year is leap year or not

Soln
```
Leap Year( Year)
IF (Year%4 ==0) THEN
   PRINT Leap Year
ELSE
   PRINT Not a Leap Year
ENDIF
```

imp. * How to write pseudocode

1. Always capitalize the initial word (often one of the main six constructs).
2. Make only one statement per line.
3. Indent to show hierachy, improve readability and show nested constructs.
4. Always end multi-line sections using any of the END keywords (ENDIF, ENDWHILE, etc).
5. Keep your statements programming language independent.
6. All the key-words used in pseudocode should be written in block letter.
7. Keep it simple, concise and readable.

Pseudocode Example 1: Adding Two numbers.

```
BEGIN
NUMBER S1, S2, Sum
OUTPUT ("Input number 1: ")
INPUT S1
OUTPUT ("Input number 2: ")
INPUT S2
Sum S1 + S2
OUTPUT Sum
END
```

Pseudocode Example 2 : Calculate Area and Perimeter of Rectangle

```
BEGIN
NUMBER len, b, area, perimeter
OUTPUT ("Input lenght and breadth")
INPUT len, b
area = len x b
perimeter = 2 x (len + b)
OUTPUT area
OUTPUT perimeter
END
```

Pseudocode Example 3: Find Area and Perimeter of Square

```
BEGIN
NUMBER S1, S2, area, perimeter
OUTPUT ("Input side:")
INPUT S1
INPUT S2
area = S1 x S1
perimeter = 4 x S1
OUTPUT area
OUTPUT perimeter
END
```

Example 4 : Check a number is positive or negative?

```
BEGIN
NUMBER num
OUTPUT "Enter a number"
INPUT num
IF num > 0 THEN
    OUTPUT "Enterned number is positive"
ELSE IF num < 0 THEN
```

```
    OUTPUT "Entered number is negative"
ELSE
    OUTPUT "Entered number is zero"
ENDIF
END
```

Example 5: Find the biggest of three number (with and without logical operators)

With logical operators

```
BEGIN
NUMBER num1, num2, num3
INPUT num1
INPUT num2
INPUT num3

IF num1 > num2 AND num1 > num3 THEN
OUTPUT "num1 is the biggest of three numbers"
ELSEIF num2 > num1 AND num2 > num1 THEN
   OUTPUT "num2 is the biggest of three numbers"
ELSE
   OUTPUT "num3 is the biggest of three numbers"
ENDIF
END
```

Without logical operators.

```
BEGIN
NUMBER num1, num2, num3
INPUT num1
INPUT num2
INPUT num3
```

```
IF num1 > num2 THEN
   IF num1> num3 THEN
   OUTPUT "num1 is the biggest of the three numbers"
   ELSE
   OUTPUT "num3 is the biggest of the three numbers"
   ENDIF
ELSEIF num2 > num3 THEN
    OUTPUT "num2 is the biggest of the three numbers"
    ELSE
   OUTPUT "num3 is the biggest of the three numbers"
 ENDIF
END
```

logical operators

S_1	S_2	AND $S_1 \cdot S_2$	OR $S_1 + S_2$	!OR	!AND
0	0	0	0	1	1
0	1	0	1	0	1
1	0	0	1	0	1
1	1	1	1	0	0

Example 6: Print number from 1 to 100 with the help of FOR loop.

```
BEGIN
~~NUMB~~
FOR COUNTER =1 REPEAT
  ~~REPE~~ COUNTER <=100
  DO
  OUTPUT COUNTER
ENDFOR
END
```

* Define Flowchart
Graphical representation of the algorithm or pseudocode into to understand the code visually.
Visualization

Advantage
- Since a flowchart is pictorial representation of algorithm, it is easy to interpret and understand the process.

- Types of flowchart
1. Process flowchat
2. Data flowchat
3. business modeling flowchat

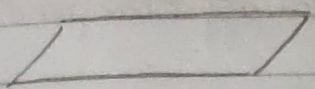
i. Process flowchat
This flowchart involves the activity of processing the data.

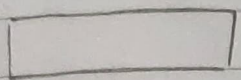
ii. Data flowchart
With this flowchart, one can study and manage the data in order to analyze the information inside and outside the system.

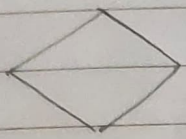
iii. Business process modeling flowchart
With the help of this flowchart any type of business can be represented with the help of notations or diagram

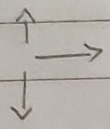
* Symbols, notation or flowchart

1 Represents start or End point -

2.  - Input / Output

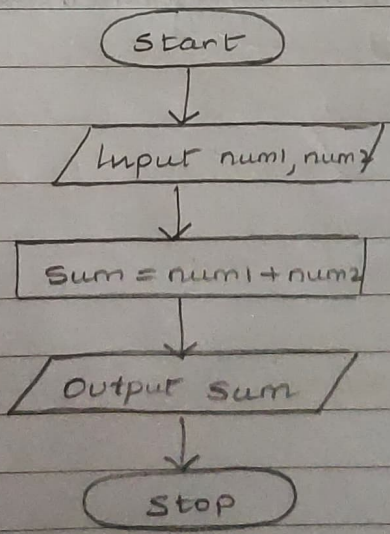
3.  - Processing

4.  - Decision box

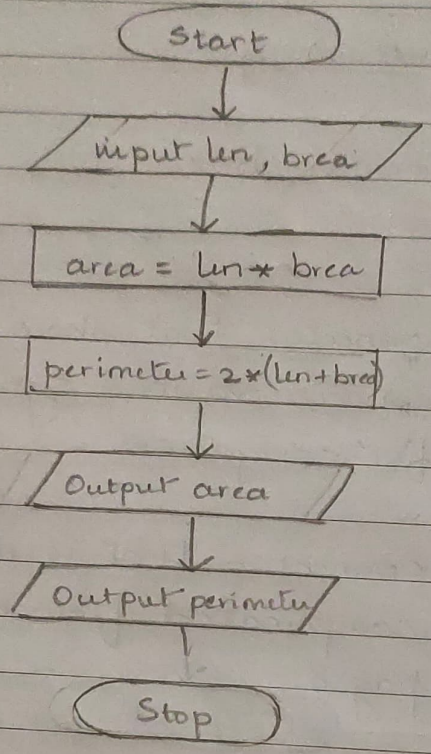
5.  - connector that show the relationship between various symbol & is used for navigation

6.  - connector that is used for drawing complicated flowchart in multiple pages.

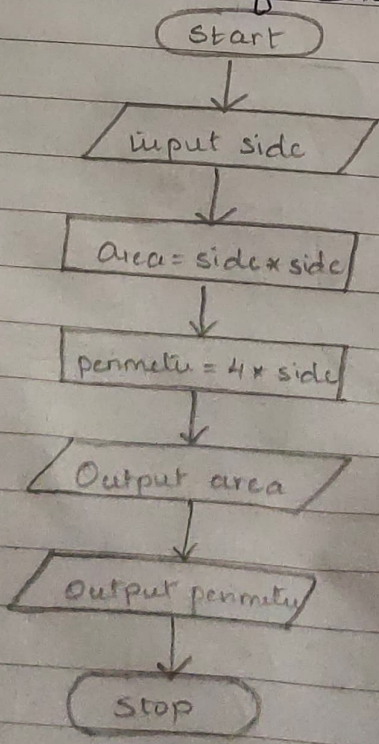
Example 1: Draw the flowchart for adding two numbers.



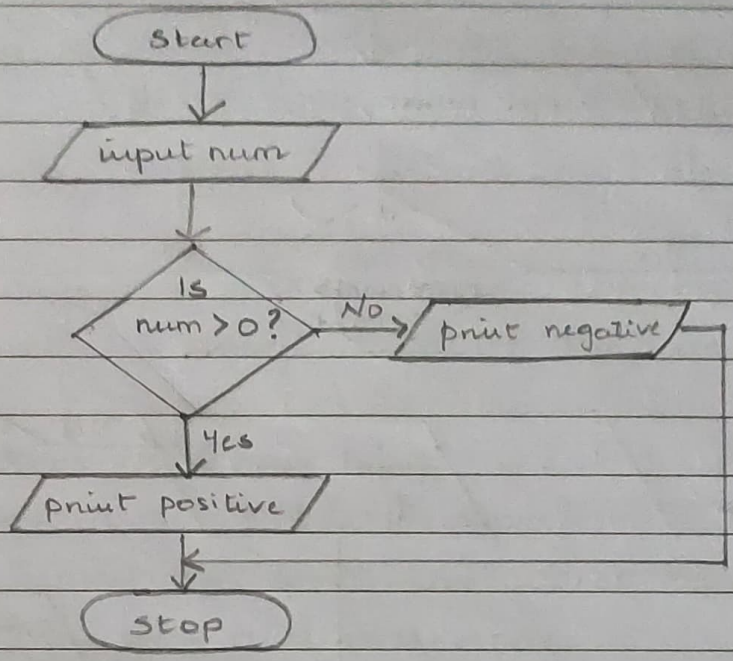
Example 2 : Draw a flowchart for calculating area and perimeter of rectangle



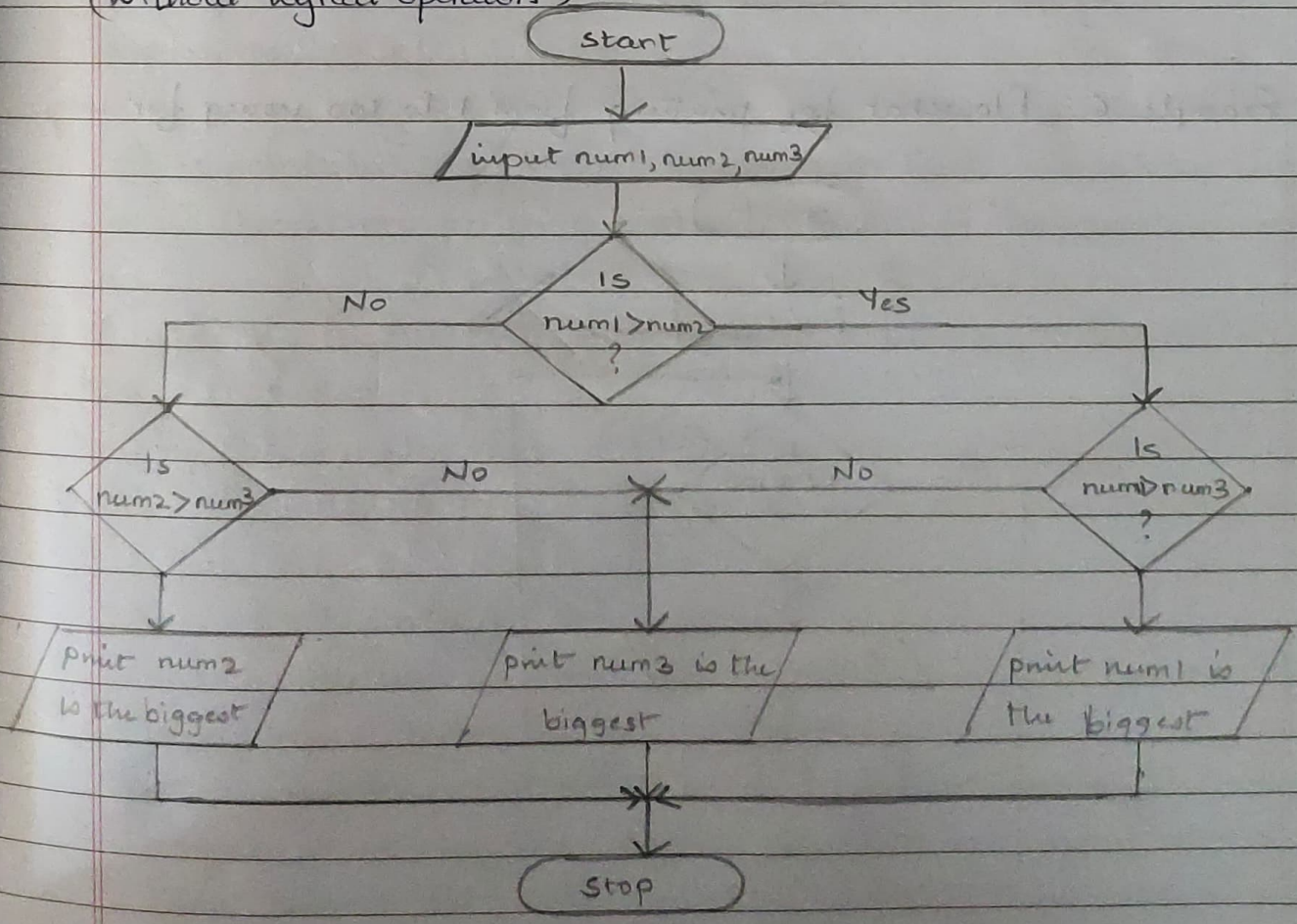
Example 3: Flowchart for calculating area and perimeter of squa



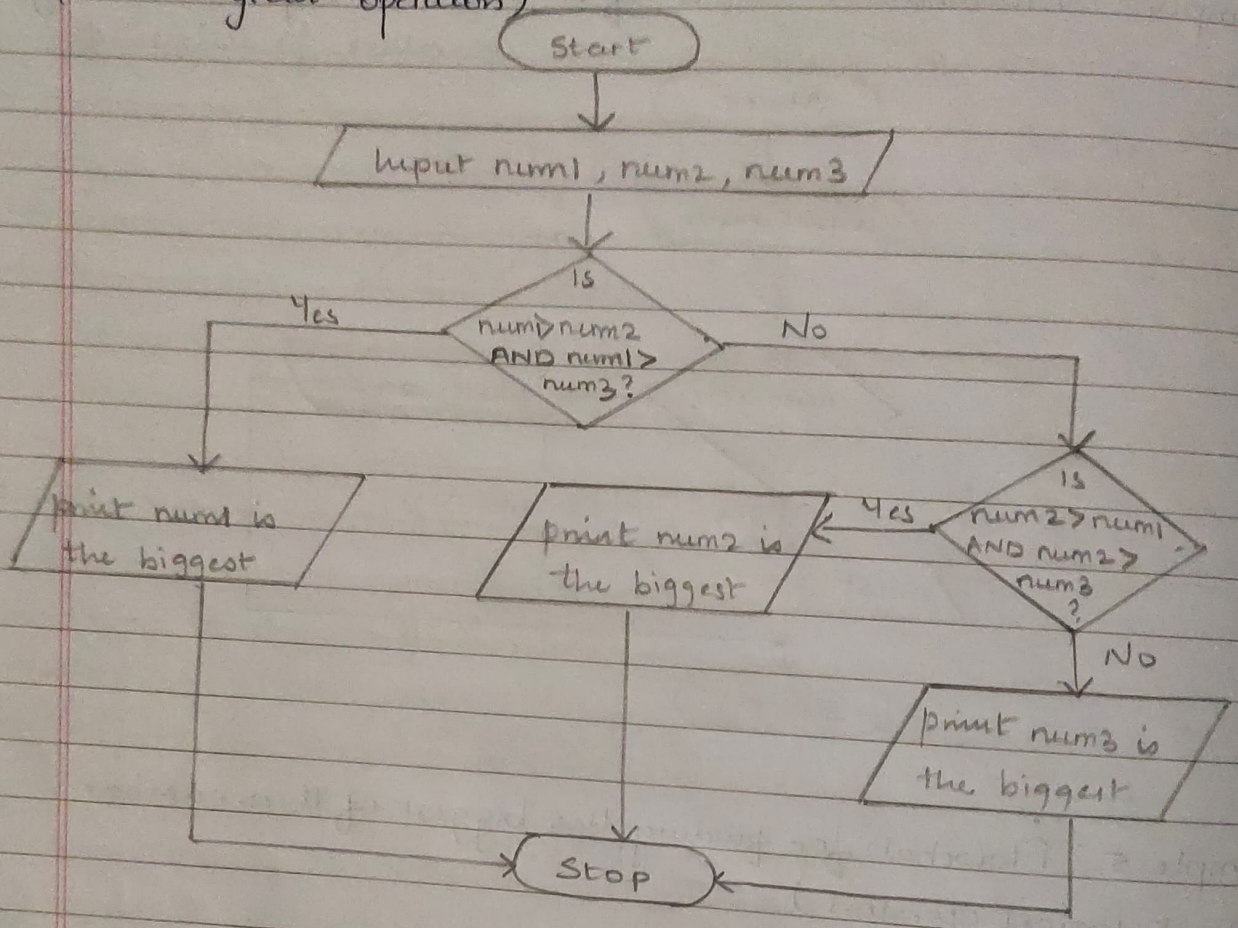
Example 4: Flowchart to check if a number is positive or negative.



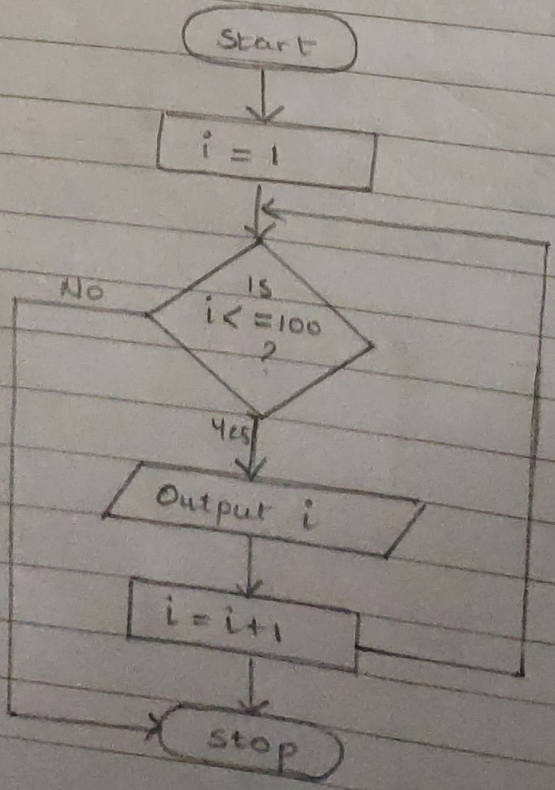
Example 5: Flowchat for finding the biggest of three numbers. (without logical operators)



(With logical operators)



Example 6: Flowchat for printing from 1 to 100 using for loop



What is the Analysis of the Algorithm?

Analysis of algorithms is the process of finding the computational complexity of an algorithm, By computational complexity, we are refferring to the amount of time taken, space and any other resources needed to execute (run) the algorithm.

- The goal of algorithm analysis is to compare different algorithms that are used to solve the same problem.

- Types of algorithm analysis
1. Best case: least time and least space
2. Average case: in-between best and worst case
3. Worst case: max. time and max. space.

- What are Asymptotic notation?

Asymptotic notation of an algorithm is a mathematical representation of its complexity.

- Goals of asymptotic alg.

A-symptotic notation algorithm in general tell us about how good an algorithm performs when compared to another algorithm.

Types of asymptotic notation.
1. Big O Notation (O)
2. Big Omega Notation (Ω)
3. Theta Notation (Θ)

1. Big-Oh Notation (O)

Big-Oh notation is used to define the upper bound of an algorithm in terms of Time complexity.

That means Big-Oh notation always indicates the maximum time required by an algorithm for all input values.

That means Big-Oh notation describes the worst case of

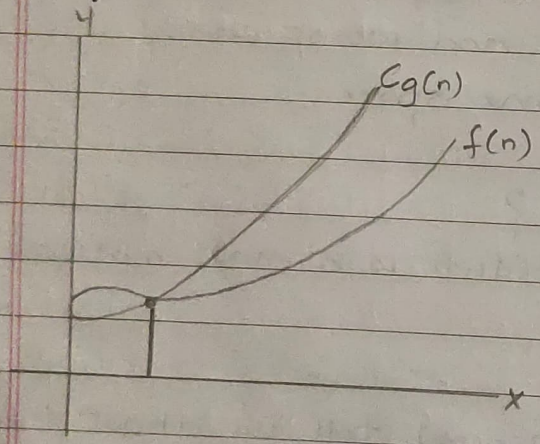
an algorithm time complexity.

Big-Oh Notation can be defined as follows:

Consider function $f(n)$ as time complexity of an algorithm and $g(n)$ is the most significant. If $f(n) <= g(n)$ for all $n >= n_0$, $c > 0$ and $n_0 >= 1$. Then we can represent $f(n)$ as $O(g(n))$.

$$f(n) = O(g(n))$$

Consider the following graph drawn for the values of $f(n)$ and $C(g(n)$ for input (n) value on X-Axis and time required is on Y-axis.



Example:

Consider the following $f(n)$ and $g(n)$.

$f(n) = 3n+2$

$g(n) = n$

If we want to represent $f(n)$ as $O(g(n))$ then it must satisfy $f(n) <= Cg(n)$ for all values of $C > 0$ and $n >= 1$

$f(n) <= Cg(n)$

$\Rightarrow 3n+2 <= Cn$

Above condition is always TRUE for all values of $C = 4$ & $n >= 2$.

by using Big-Oh notation we can represent the time complexity as follow

$$3n+2 = O(n)$$

11. Big Omega Notation (Ω)

Big Omega Notation is used define the lower-bound of an algorithm in terms of time complexity.

That means big-Omega notation always indicates the minimum time required by the algorithm for all input values.

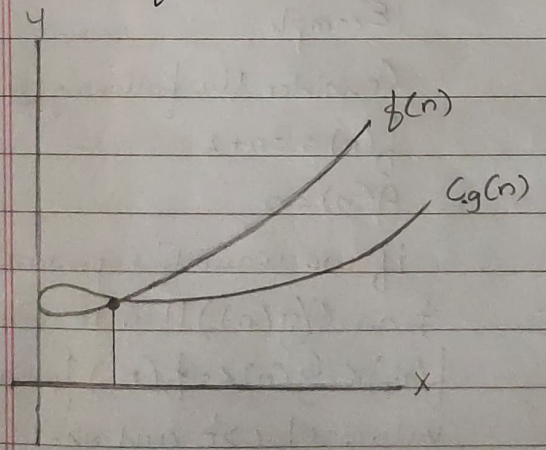
That means big-omega notation describes the best case of an algorithm time complexity.

Big-Omega notation can be defined as follows:

Consider function $f(n)$ as time complexity of an algorithm and $g(n)$ is the most significant. If $f(n) >= g(n)$ for all $n >= n_0$, $c > 0$ and $n_0 >= 1$. Then we can represent $f(n)$ as $\Omega(g(n))$

$$f(n) = \Omega(g(n))$$

Consider the following graph drawn for the values of $f(n)$ and $C(g(n))$ for input (n) value on X-Axis and time required is on Y-axis.



Example:

Consider the following $f(n)$ $g(n)$

$f(n) = 3n + 2$

$g(n) = n$

if we want to represent $f(n)$ as $\Omega(g(n))$ then it must satisfy $f(n) >= cg(n)$ for all values of $c > 0$ and $n >= 1$

$f(n) >= cg(n)$

$\Rightarrow 3n + 2 >= Cn$

Above condition is always TRUE for all values of $c = 4$ & $n >= 2$.

by using Big-Omega notation we can represent the time complexity as follow: $3n + 2 = \Omega(n)$

4. Theta Notation (Θ)

Theta Notation is used to define the bound of an algorithm in terms of time complexity.

That means theta notation always indicates the average time required by the algorithm for all input values.

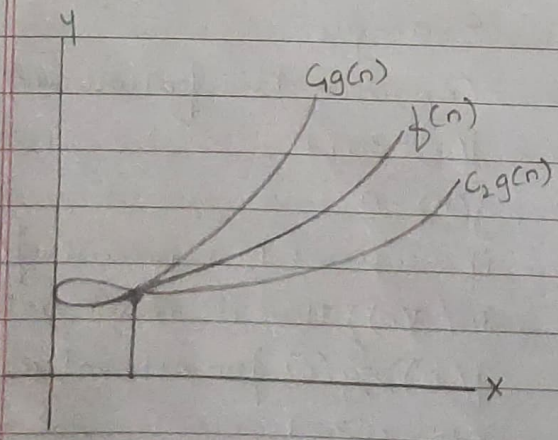
That means theta notation describes the average case of an algorithm time complexity.

Theta notation can be defined as follows:

Consider function $f(n)$ as time complexity of an algorithm and $g(n)$ is the most significant. If $f(n_1) <= g(n) <= f(n_2)$ for all $n >= n_0$, $c = 0$ and $n_0 >= 1$. Then we can represent $f(n)$ as $\Theta(g(n))$

$$f(n) = \Theta(g(n))$$

Consider the following graph drawn for the values of $f(n)$ and $c(g(n))$ for input (n) value on X-Axis and time requi is on Y-axis.



Example:

Consider the following $f(n)$ & g

$f(n) = 3n + 2$

$g(n) = n$

if we want to represent $f(n)$ as $\Theta(g(n))$ then it must satis $f(n_1) <= g(n) <= f(n_2)$ for all values of $c > 0$ and $n >= 1$

$f(n_1) <= g(n) <= f(n_2)$

$\Rightarrow 3n + 2$

Above condition is always TRUE for all values of $c = 4$ & $n >= 2$.

by using Theta notation we can represent the time complexity as follow: $3n + 2 = \Theta(n)$

* Space complexity:

- Space complexity is nothing but the amount of memory space that an algorithm or a problem takes during the execution of that particular problem / alg.
- The space complexity is not only calculated by the space used by the variables in the problem / alg. it also includes and considers the space for input values with it

Space Complexity = Auxiliary Space + Space used for input values.

Example 1: Addition of Numbers.

```
{ int a = x+y+z;
  return (a)
}
```

Hence, the total space complexity: 4 * 4 + 4 = 20 bytes

(no. of variables → 4; space required → 4; return(a) → 4)

Example 2:

```
{
   float x = y*z;
   int a = b+c+d+e;
   return (x);
   return (a);
}
```

(64 bits - 4 int space - 8 float)

total space complexity: 8 * 3 + 5 * 4 + 8 + 4

(space, no. of v., no. of v., space, return)

24 + 20 + 8 + 4 = 56

* What is empirical analysis?

Empirical analysis is an evidence-based approach to the study and interpretation of information. Empirical evidence is information that can be gathered from experience or by the five senses.

- In computer science, empirical algorithmics (or experime alg.) is the practise of using empirical methods to study the behaviour of alg. Alg. are not just designed but also implemented and tested in a variety of situation.

* Design of Algorithm.

Imp 1. Brute force desiging Algorithm

- In Computer Science, brute force is a trial and error methodology.
- Brute force relies on the computational power of the machine to try every possible combination to achieve the target.
- It is also known as Exhaustive search or Generate & Test that yields all possible solution of the problem.

Eg: i. Password Cracking
ii. Rubic cube

Advantages:

i. This algorithm will find all possible solutions and guarantee that it will solve the problem correctly.
ii. This algorithm can be applied to many domains.
iii. It is used primarily to solve small and simple problems.
iv. It is a benchmark that can be used to compare solutions to simple problem and doesn't require domain knowledge.

Disadvantages

i) This algorithm is in-efficient as it must solve every state.
ii) The brute force algorithm does not produce any creative or constructive results when compared to other algorithms

Imp. 1] Bubble sort

i. For each iteration, the bubble sort will compare up to the last unsorted element.

ii. Once all the elements get sorted, in the ascending order, the algorithm will get terminated.

Pseudocode

```
Initialize n = Length of Array
BubbleSort(Array, n)
{
for i=0 to n-2
{
 for j=0 to n-2
{
 if Array [j] > Array [j+1]
{
 swap(Array [j], Array [j+1])
}
ENDIF
}
ENDIFOR
}
ENDFOR
}
```

Eg: Assume an array a = {16, 36, 24, 37, 15}. Sort the elements in ascending order using bubble sort.

Soln Pass 1:

a =

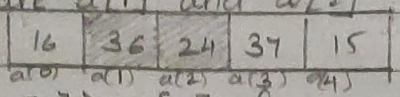
16	36	24	37	15
a[0]	a[1]	a[2]	a[3]	a[4]

Compare a[0] and a[1]

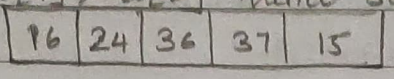
16	36	24	37	15

∴ a[0] < a[1] hence no swap

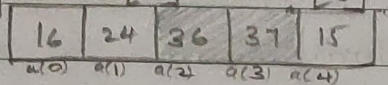
Compare a[1] and a[2]



∴ a[1] > a[2] hence swap

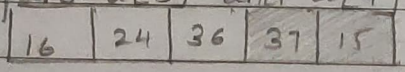


Compare a[2] and a[3]

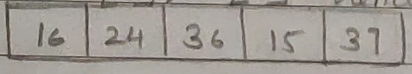


∴ a[2] < a[3] hence no swap

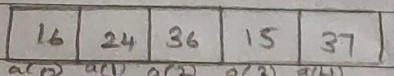
Compare a[3] and a[4]



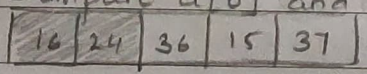
∴ a[3] > a[4] hence swap



Pass 2:

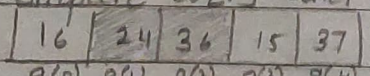


Compare a[0] and a[1]



∴ a[0] < a[1] hence no swap

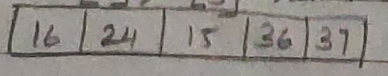
Compare a[1] and a[2]



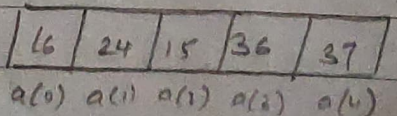
∴ a[1] < a[2] hence no swap

Compare a[2] and a[3]

∴ a[2] > a[3] hence swap



Pass 3:



Compare a[0] and a[1]

16	24	15	36	37
a[0]	a[1]	a[2]	a[3]	a[4]

∴ a[0] < a[1] hence no swap

Compare a[1] and a[2]

16	24	15	36	37
a[0]	a[1]	a[2]	a[3]	a[4]

∴ a[1] > a[2] hence swap

16	15	24	36	37

Pass 4 :

16	15	24	36	37
a[0]	a[1]	a[2]	a[3]	a[4]

Compare a[0] and a[1]

16	15	24	36	37
a[0]	a[1]	a[2]	a[3]	a[4]

∴ a[0] > a[1] hence swap

15	16	24	36	37

Hence the final completed array is : | 15 | 16 | 24 | 36 | 37 |

2] In the given list of array {89, 45, 68, 90, 29, 34, 17} Sort the elements in ascending order with the help of bubble sort method.

Pass 1 :

a[] =

89	45	68	90	29	34	17
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]

Compare a[0] and a[1]

89	45	68	90	29	34	17
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]

∴ a[0] > a[1] hence swap

Compare a[1] and a[2]

45	89	68	90	29	34	17
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]

∴ a[1] > a[2] hence swap

Compare a[2] and a[3]

45	68	89	90	29	34	17
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]

∴ a[2] < a[3] hence no swap

Compare a[3] and a[4]

45	68	89	90	29	34	17
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]

∴ a[3] > a[4] hence swap

Compare a[4] and a[5]

45	68	89	29	90	34	17
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]

∴ a[4] > a[5] hence swap

Compare a[5] and a[6]

45	68	89	29	34	90	17
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]

∴ a[5] > a[6] hence swap

45	68	89	29	34	17	90

Pass 2:

45	68	89	29	34	17	90
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]

Compare a[0] and a[1]

45	68	89	29	34	17	90
0	1	2	3	4	5	6

∴ a[0] < a[1] hence no swap

Compare a[1] and a[2]

45	68	89	29	34	17	90
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]

∴ a[1] < a[2] hence no swap

Compare a[2] and a[3]

45	68	89	29	34	17	90
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]

∴ a[2] > a[3] hence swap

Compare a[3] and a[4]

45	68	29	89	34	17	90
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]

∴ a[3] > a[4] hence swap

Compare a[4] and a[5]

45	68	29	34	89	17	90
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]

∴ a[4] > a[5] hence swap

45	68	29	34	17	89	90

Pass 3:

Compare a[0] and a[1]

45	68	29	34	17	89	90
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]

∴ a[0] < a[1] hence no swap

Compare a[1] and a[2]

45	68	29	34	17	89	90
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]

∴ a[1] > a[2] hence swap

Compare a[2] and a[3]

45	29	68	34	17	89	90
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]

∴ a[2] > a[3] hence swap

Compare a[3] and a[4]

45	29	34	68	17	89	90
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]

∴ a[3] > a[4] hence swap

Pass 4:

Compare a[0] and a[1]

45	29	34	17	68	89	90
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]

∴ a[0] > a[1] hence swap

Compare a[1] and a[2]

29	45	34	17	68	89	90
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]

∴ a[1] > a[2] hence swap

Compare a[2] and a[3]

29	34	45	17	68	89	90
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]

∴ a[2] > a[3] hence swap

29	34	17	45	68	89	90

Pass 5: Compare a[0] and a[1]

29	34	17	45	68	89	90
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]

∴ a[0] < a[1] hence no swap

Compare a[1] and a[2]

29	34	17	45	68	89	90
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]

∴ a[1] > a[2] hence swap

29	17	34	45	68	89	90

Pass 6:

Compare a[0] and a[1]

29	17	34	45	68	89	90
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]

∴ a[0] > a[1] hence swap

17	29	34	45	68	89	90

Hence the final computed array is

17	29	34	45	68	89	90

In the given list of array {E, X, A, M, P, L, E}
Sort the elements in ascending order using bubble s

Sol:

a =

E	X	A	M	P	L	E
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]

Pass 1:

Compare a[0] and a[1]

E	X	A	M	P	L	E
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]

∴ a[0] < a[x] hence no swap

Compare a[1] and a[2]

E	X	A	M	P	L	E
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]

∴ a[1] > a[2] hence swap

Compare a[2] and a[3]

E	A	X	M	P	L	E
a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]

∴ a[2] > a[3] hence swap

Compare a[3] and a[4]

E	A	M	X	P	L	E
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]

∴ a[3] > a[4] hence swap

Compare a[4] and a[5]

E	A	M	P	X	L	E
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]

∴ a[4] > a[5] hence swap

Compare a[5] and a[6]

E	A	M	P	L	X	E
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]

∴ a[5] > a[6] hence swap

E	A	M	P	L	E	X

Pass 2: Compare a[0] and a[1]

E	A	M	P	L	E	X
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]

∴ a[0] > a[1] hence swap

Compare a[1] and a[2]

A	E	M	P	L	E	X
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]

∴ a[1] < a[2] hence no swap

Compare a[2] and a[3]

A	E	M	P	L	E	X
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]

∴ a[2] < a[3] hence no swap

Compare a[3] and a[4]

A	E	M	P	L	E	X
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]

∴ a[3] > a[4] hence swap

Compare a[4] and a[5]

A	E	M	L	P	E	X
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]

∴ a[4] > a[5] hence swap

A	E	M	L	E	P	X

Pass 3: Compare a[0] and a[1]

A	E	M	L	E	P	X
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]

$\therefore a[0] < a[1]$ hence no swap

Compare a[1] and a[2]

A	E	M	L	E	P	X
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]

$\therefore a[1] < a[2]$ hence no swap

Compare a[2] and a[3]

A	E	M	L	E	P	X
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]

$\therefore a[2] > a[3]$ hence swap

Compare a[3] and a[4]

A	E	L	M	E	P	X
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]

$\therefore a[3] > a[4]$ hence swap

A	E	L	E	M	P	X

Pass 4 : Compare a[0] and a[1]

A	E	L	E	M	P	X
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]

$\therefore a[0] < a[1]$ hence no swap

Compare a[1] and a[2]

A	E	L	E	M	P	X
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]

$\therefore a[1] < a[2]$ hence no swap

Compare a[2] and a[3]

A	E	L	E	M	P	X
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]

$\therefore a[2] > a[3]$ hence swap

A	E	E	L	M	P	X
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]

Pass 5: Compare a[0] and a[1]

A	E	E	L	M	P	X
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]

∴ a[0] < a[1] hence no swap

Compare a[1] and a[2]

A	E	E	L	M	P	X
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]

∴ a[1] = a[2] hence no swap

A	E	E	L	M	P	X

Pass 6:

Compare a[0] and a[1]

A	E	E	L	M	P	X
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]

∴ a[0] < a[1] hence no swap.

A	E	E	L	M	P	X

Hence the final completed array is | A | E | E | L | M | P | X |

* Complexity of Bubble sort

- Best case complexity: The bubble sort algorithm has a best-case time complexity of O(n) for the already sorted array.
- Average Case complexity: The average case time complexity for the bubble sort algorithm is $O(n^2)$, which happens when 2 or more elements are in jumbled i.e. neither in the ascending order nor in the descending order.
- Worst Case complexity: The worst-case time complexity is also $O(n^2)$, which occurs when we sort the descending order of an array into the ascending order.

Advantages of Bubble Sort.

i) Easily understandable.
ii. Does not neccessitates any extra memory.
iii. The code can be written easily for this algorithm.
iv. Minimal space requirement than that of other sorting algorithm

Disadvantages of Bubble Sort.

i. It does not work well when we have large unsorted list, and it necessitates more resources that end up taking so much of time.
ii. It involves the n^2 order of steps to sort an algorithm.

Q. Write a program for bubble sort.

```
#include<stdio.h>
#include<conio.h>
int main()
{   int arr[50], num, x, y, temp;
    printf("Enter the size of array: ");
    scanf("%d", &num);
    printf("Enter the elements: ");
    for(x=0; x<num; x++)
        scanf("%d", &arr[x]);
    for(x=0; x<num-1; x++){
        for(y=0; y<num-x-1; y++){
            if(arr[y] > arr[y+1]){
                temp = arr[y];
                arr[y] = arr[y+1];
                arr[y+1] = temp;
            }}}
    printf("Array after implementing bubble sort: ");
    for(x=0; x<num; x++){
        printf("%d", arr[x]);
    }return 0;}
```

imp. ii) Selection Sort

Algorithm
SELECTION SORT(arr, n)
Step 1: Repeat Steps 2 and 3 for i=0 to n-1
Step 2: CALL SMALLEST (arr, i, n, pos)
Step 3: SWAP arr[i] with arr[pos];
[END OF LOOP]
Step 4: EXIT

SMALLEST(arr, i, n, pos)
Step 1: [INITIALISZE] SET SMALL = arr[i]
Step 2: [INITIALISZE] SET pos = i
Step 3: Repeat for j = i+1 to n
if (SMALL > arr[j])
SET SMALL = arr[j]
SET pos = j
[END OF if]
[END OF LOOP]
Step 4: RETURN pos

Eg 1: Working of Selection sor Algorithm
Consider the list of elements {12, 29, 25, 8, 32, 17, 40}, Sort the elements accordding to selection sort

Sol:

12	429	25	8	32	17	40

Now, for the first position in the sorted array, the entire array is to be scanned sequentially. At present, 12 is stored at the first position, after searching the entire array, it is found that 8 is the smallest value.

12	29	25	8	32	17	40

So, swap 12 with 8. After the first iteration, 8 will appear at the first position in the sorted array.

| 8 | 29 | 25 | 12 | 32 | 17 | 40 |

For the second position, where 29 is stored presently, we again sequentially scan the rest of the items of unsorted array. After scanning, we find that 12 is the second lowest element in the array that should be appeared at second position.

| 8 | 29 | 25 | 12 | 32 | 17 | 40 |

Now, swap 29 with 12. After the second iteration, 12 will appear at the second position in the sorted array. So, after two iterations, the two smallest values are placed at the beginning in a sorted array.

| 8 | 12 | 25 | 29 | 32 | 17 | 40 |

The same process is applied to rest of the array.

| 8 | 12 | 25 | 29 | 32 | 17 | 40 |

| 8 | 12 | 25 | 29 | 32 | 17 | 40 |

| 8 | 12 | 17 | 29 | 32 | 25 | 40 |

| 8 | 12 | 17 | 29 | 32 | 25 | 40 |

| 8 | 12 | 17 | 29 | 32 | 25 | 40 |

| 8 | 12 | 17 | 25 | 32 | 29 | 40 |

| 8 | 12 | 17 | 25 | 32 | 29 | 40 |

| 8 | 12 | 17 | 25 | 29 | 32 | 40 |

Eg 2: Consider the array {89, 45, 68, 90, 29, 34, 17}. Sort the elements in ascending order using selection sort.

soln

1: | 89 | 45 | 68 | 90 | 29 | 34 | 17 |

| 89 | 45 | 68 | 90 | 29 | 34 | 17 |
S

| 17 | 45 | 68 | 90 | 29 | 34 | 89 |

2: | 17 | 45 | 68 | 90 | 29 | 34 | 89 |

| 17 | 45 | 68 | 90 | 29 | 34 | 89 |
S

| 17 | 29 | 68 | 90 | 45 | 34 | 89 |

3: | 17 | 29 | 68 | 90 | 45 | 34 | 89 |

| 17 | 29 | 68 | 90 | 45 | 34 | 89 |
S

| 17 | 29 | 34 | 90 | 45 | 68 | 89 |

4: | 17 | 29 | 34 | 90 | 45 | 68 | 89 |

| 17 | 29 | 34 | 90 | 45 | 68 | 89 |
S

| 17 | 29 | 34 | 45 | 90 | 68 | 89 |

5: | 17 | 29 | 34 | 45 | 90 | 68 | 89 |

| 17 | 29 | 34 | 45 | 90 | 68 | 89 |
S

| 17 | 29 | 34 | 45 | 68 | 90 | 89 |

6: | 17 | 29 | 34 | 45 | 68 | 90 | 89 |

| 17 | 29 | 34 | 45 | 68 | 90 | 89 |
S

| 17 | 29 | 34 | 45 | 68 | 89 | 90 |

Hence the final sorted array is {17, 29, 34, 45, 68, 89, 9

Complexity

- Best Case Complexity : $O(n)$
- Average Case complexity : $O(n^2)$
- Worst Case complexity : $O(n^2)$

* Sequential Search

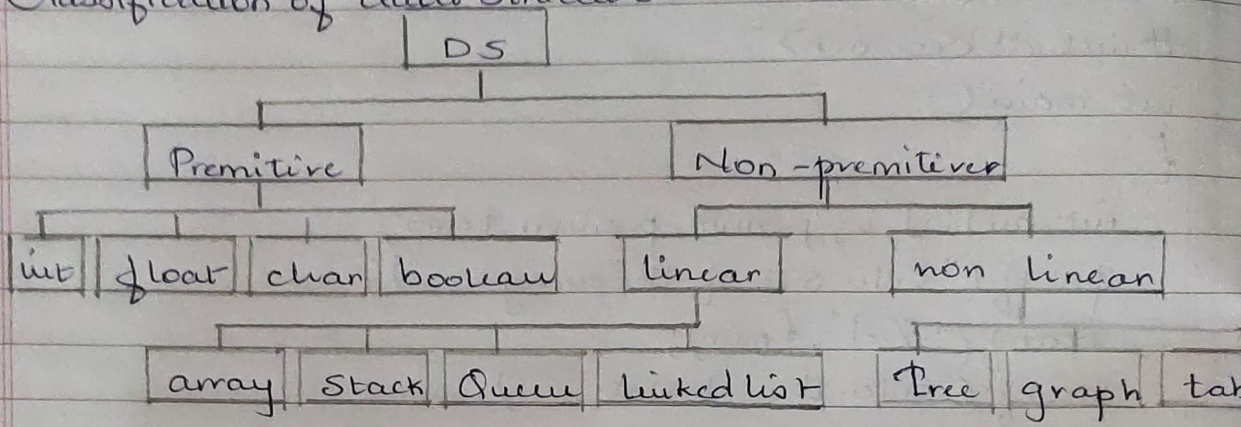
Algorithm SequentialSearch2(A[0...n], K)
// K is the search key and A has n elements
A[n] ← K
while A[i] ≠ K do
i++
if i<n then return i
else return -1

Q] Write a program to implement selection sort.

```
#include<stdio.h>
#include<conio.h>
int main()
{
  int arr[10], i, j, num, position, temp;
  printf("Enter size of the array: ");
  scanf("%d", &num);
  printf("Enter the elements: ");
  for(i=0; i<num; i++)
  {
    scanf("%d", &arr[i]);
  }
  for(i=0; i<num-1; i++)
  {
    position = i;
    for(j=i+1; j<num; j++)
    {
      if(arr[position] > arr[j])
      {
        temp = arr[position];
        arr[position] = arr[j];
        arr[j] = temp;
      }
    }
  }
  printf("After implementing Selection Sort: \n");
  for(i=0; i<num; i++)
  {
    printf("%d ", arr[i]);
  }
  return 0;
}
```

13-2-23 DATA STRUCTURE

* Classification of data structure:



- Data Structure :i) It is a mathematical modeling of data.
  ii) Data structure can be of two types: premitive and non-premitive.
- i) Premitive data structures are the types that are understood by the machine directly.
  ii) Int, float, char, boolean etc are types of premitive data structures.
  i] Int: represented by numbers and holds 4 bytes.
  ii] float: represented by decimal numbers and holds 8 byt
  iii] char: represented by characters and holds 1 byte.
  iv] boolean: represented by true or false i.e 1 or 0.

- i) Non-premitive data structure:

ii) There are two types of non-premitive data structu namely linear and non-linear.
i] linear data structure is based on sequential approach
ii] Array, stack, queue and linked list are examples of linear data structure.
i) Array: Collection of same type of data.
ii) Stack: a linear ds that follows LIFA (last in first out approach. It has a only opening i.e top.

iii) Queue: A linear DS that follows FIFO (first in first out) approach.

* Data structure

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

ii. It is also used for processing, retrieving and storing data.

There are two types of data structure available for the programming purpose:

1] Primitive data structure (that machine can understand)

i. Primitive data structure is a fundamental type of data structure that stores the data of only one type.

ii. Examples of primitive data structure are integer, character, float and pointer.

iii. The size depends on the type of data structure.

The following are the four primitive data structure.

- Integer: The integer data type contains the numeric values. It contains the whole numbers that can be either negative or positive. (4 bytes)
- Float: The float is a data type that can hold decimal values. When the precision of decimal value increases then the double data type is used. (8 bytes)
- Boolean: It is a data type that can hold either a True or False values. It is mainly used for checking the condition.
- Character: It is a data type that can be hold a single character value both uppercase and lowercase such as 'A' and 'a'. (1 byte)

II] Non-primitive (that machine cannot understand)

i) The non-primitive data structure is a kind of data structure that can hold multiple values either in a contiguous or random location.

ii. The non-primitive data structure is further classified into two categories: i.e linear and non-linear data structure.

i) Linear data structure.

- Data structure in which data elements are arranged sequential or linearly, where each element is attached to its previous and next adjacent element, is called a linear data structure.
- Examples of linear data structures are array, stack, queue, linked list etc.

The following are the types of linear data structures:

- Array: An array is a data structure that can hold the elements of same type.

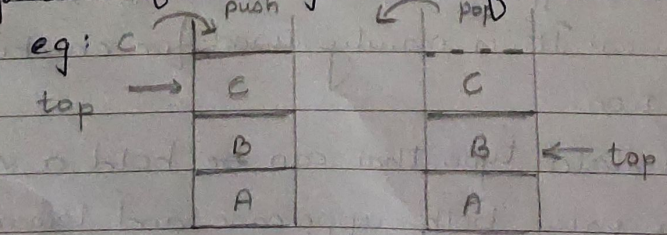
  The commonly used operation in an array is insertion, deletion, traversing, searching.

  eg: int a[6] = {1, 2, 3, 4, 5, 6}

  eg: 1D, 2D & multi D.

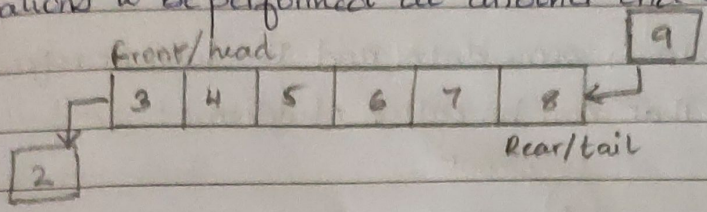
  non primitive linear

- Stack: Stack is a data structure that follows the principle LIFO (Last In First Out). All the operations on the stack are performed from the top of the stack such as PUSH and POP operations. The push operation is the process of inserting elements into the stack while the pop operation is the process of removing element from the stack.

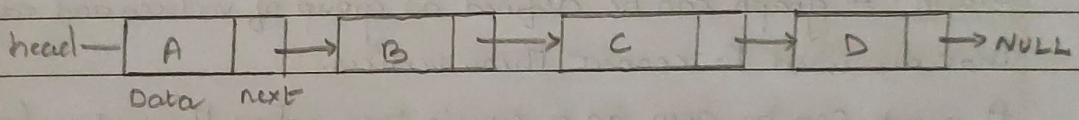


non-premitive linear

- Queue: Queue is a data structure which is referred to be as First In First Out list.
A queue can be defined as an ordered list which enables insert operations to be performed at one end called rear and delete operations to be performed at another end called front.



- Linked link: can be defined as collection of objects called nodes that are randomly stored in the memory.
A node contains two fields i.e data stored at that particular address and the pointer which contains the address of the next node in the memory.
The last node of the list contains pointer to the null.

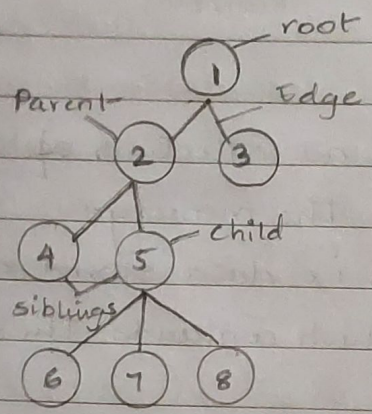


ii) Non-linear data Structure
- Data structures where data elements are not placed sequentially or linearly are called non-linear data structures.
- In a non-linear data structure, we can't traverse all the elements in a single run alone.
- Examples of non-linear data structures are tree, graphs and tables.

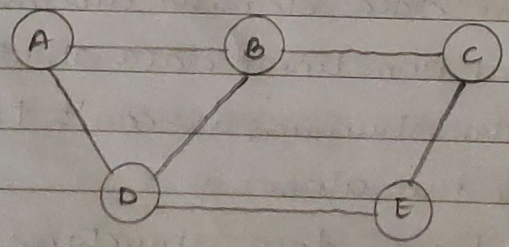
The following are the types of non-linear data structures:
non premitive non linear
- Tree: A tree data structure is defined as a collection of objects or entities known as nodes that are linked together to represent or simulate hierarchy.
- A tree data structure is a non-linear data structure because it does not store in a sequential manner. It is a

hierarchical structure as elements in a Tree are arranged in multiple levels.

- In the tree data structure, the topmost node is known as a root node. Each node contains some data, and data can be of any type.
- Each node contains some data and the link or reference of other nodes that can be called children.



- Graph: A graph can be defined as group of vertices and edges that are used to connect these vertices.
A graph can be seen as a cyclic tree, where the vertices (nodes) maintain any complex relationship among them instead of having parent child relationship. eg: cyclic & acyclic



- Table: Is a non-premitive, non-linear data structure that contains rows and columns.

Questioner

1. Define data & information
2. Define algorithm, charact. of algorithm, adv & disadv. of algorithm, types of algorithm.
3. Asymptotic notation — 8 mks. (V. imp)
4. Pseudocode
5. Brute force technique (imp.)
   a) bubble sort
   b) Selection sort
6. Fundamental of DS