2. INTRODUCTION TO ALGORITHM.

• ALGORITHM :

An algorithm is a set of unambigious instructions, which, when executed performs a task correctly.

• CHARACTERISTICS OF AN ALGORITHM:

1. INPUT : An algorithm has some input values. We can pass zero or some input values to an algorithm.
2. OUTPUT : We will get one or more output at end of algorithm.
3. DEFINITENESS : Each instruction is in clear format.
4. FINITENESS : If we trace out the instructions, the algorithm must terminate after finite steps.
5. EFFECTIVENESS : Every instruction must be in basic format.

• ALGORITHM TYPES :

1. SEQUENCE ALGORITHM : It is a series of steps in sequential order without any break. Here, instructions are executed from top to bottom.
2. Selection ALGORITHM : steps of an algorithm are designed by selecting condition checking like IF, IF-ELSE, elif etc.
3. ITERATION ALGORITHM : Based on certain conditions, & repetedely processed the same statements until the specified condition becomes false while, do-while and for.
4. RECURSIVE ALGORITHM : A function calls itself is known as recursion and the function is called as recursive function. The main advantage of recursion concept is to reduce length of code.
   a. DIRECT RECURSION : A function calls itself directly is known as direct recursion
   b. INDIRECT RECURSION : A function calls another function, which initiates to call of the initial function is known as Indirect recursion.

- APPROCHES IN ALGORITHM:

1. BRUTE FORCE ALGORITHM:

The general logic structure is applied design an algorithm. It is also known as EXHAUSTIVE SEARCH ALGORITM that searches all possible to provide required solution.

TYPES:

1. OPTIMIZING: Finding all solutions of a proble and then taking out the best solution is known then it will terminate if the best solution is known.
2. SACRIFICING: As soon as the best solution is found, then it will stop.

2. DIVIDE AND CONQUER:

This breaks down the algorithm to solve the problem in different methods. It allows you to break down problem into differ methods and valid output is produced for the valid input. This valid output is passed to some other function.

3. GREEDY ALGORITHM:

It is an algorithm paradigm that ma an optional choice on each iteration with the hope of getting best solution. It is easy to implement and has faster execution time. But there are very rare cases in which it provid the optimal solution.

- The major categories of algorithms are given belo

1. Sort: Algorithm developed for sorting the items in a certain order
2. Search: Algorithm developed for searching the items inside a data structure.
3. Delete: Algorithm developed for deleting the existing element from the data structure
4. Insert: Algorithm developed for inserting an item inside a data structure.
5. Update: Algorithm developed for updating the existing element inside a data struct

3. Analysis of Algorithm :

1. What is meant by Algorithm Analysis?

Algorithm analysis is an important part of "computational theory of complexity".

- It provides theoritical estimation for the required resources of an algorithm to solve a computational problem.
- Analysis of algorithm is the determination of the amount of time and space resources required to execute it.

2.
- The Algorithm can be analyzed in two ways (levels). i.e, first is before creating the algorithm and second is after creating the algorithm.
- There are two analysis of an algorithm:

1. PRIORI ANALYSIS :

Here, priori analysis is the theoretical analysis of an algorithm which is done before implementing the algorithm.

2. PASTERIOR ANALYSIS:

Here, pasterior analysis is a practical analysis of an algorithm. The practical analysis is achieved by implementing algorithm using any programming language

3. TYPES OF ALGORITHM ANALYSIS:

1. Best case.
2. Worst case.
3. Average case.

4. Why Analysis of Algorithms is important?

1. To predict the behaviour of an algorithm without implementing it on a specific system.
2. By analyzing different algorithms, we can compare them to determine the best one for our purpose.
3. It is important for evaluating efficiency, comparing options, optimizing performance, guiding design, predicting behaviour and Impacting real-world applications.
4. The analysis is only an approximation, it is not perfect.

1. ·ALGORITHM COMPLEXITY - TIME COMPLEXITY:

The performance of the algorithm is the amount of the time required to compare execution.

- The time complexity of an algorithm is denoted by the 'BIG O' notation.
- Here big O notation is the asymptotic notation to represent time complexity.
- The time complexty is mainly calculated by counting the number of steps to finish execution.

→Example:

```
sum=0
for i in n:
    sum += i
return sum.
```

- In above code, the time complexity of the loop statement will be atleast 'n' and if value of 'n' increases, then ~~com~~ time complexity also increases.
- We generally consider the worst-time complexity as it is the maximum time taken for any ~~in~~ given input size.

2. SPACE COMPLEXITY:

An algorithm's space complexity is the amount of space required to solve a problem and produce an output. Similar to the time complexity, space complexity is also expressed in 'BIG O' NOTATION.

- [AUXILLARY SPACE is extra space or temporary space used by algorithm].
- The space complexity of an algorithm is the total space taken by algorithm with respect to input size.
- Space complexity includes both AUXILLARY SPACE and space used by input.
- Space complexity = Auxillary space + Input Size

3. SEARCH ALGORITHM :

on each day, we search for something in our day to day life.

Similarly, with the case of computer, huge data is stored in a computer that whenever user asks for any data then the computer searches for that data in the memory and provides that data to the user. There are mainly two techniques available to search data in an array:

a. LINEAR SEARCH :

Linear search is a simple search algorithm that sequentially checks each element in a list until the desired element is found or the end of the list is reached.

b. BINARY SEARCH :

Binary search is a fast searching algorithm that works on sorted lists. It follows divide and conquer approch by repetedly dividing the search interval in half

4. SORTING ALGORITHM :

Sorting Algorithms are used to rearrange elements in an array or a given data structure either in an ascending or descending order.

ANALYSIS OF COMMON LOOP :

While() Loop :

- Loops are essential constructs in programming that allows us to repeat a block of code multiple times.
- Python offers several types of loops and one of the most versatile is the while() loop.
- The while loop continues to execute a block of code as long as a given condition remains true.

- ADDITION USING While() LOOP :

```
def add(a,b):
    result=a
    while b>0:
        result +=1
        print(result)
        b=b-1
a=10
b=7
add(a,b).
```

OUTPUT:

```
11
12
13
14
15
16
17
```

- In the above code, we define a function called add(), that takes two numbers a & b as parameters.
- we initialize the variable result with value of a
- The while loop executes 'b' times, incrementing the result by 1, on each iteration while decrement 'b' by 1.
- TIME COMPLEXITY ANALYSIS:
  - Time complexity : O(b)
  - similar to multiplication, the while loop executes 'b' times, where b is the value being added. Thus, the time complexity is linear and proportional to b.

- Time complexity analysis of substraction using a while loop : O(a/b).
  The time complexity is linear & depends on magnitude of a and b.
- ThTime complexity analysis of Multiplication using a while loop: O(log n).
- Exponentiation : O(log(log n)).
- Nested while loops:

```
n=int(input())
i=0
while i<=5:
    j=1
    while j<=n:
        j+=1
    i+=1
```

time complexity : O(nlogn)

- CONCLUSION :

The while loop is a powerful construct in python that allows us to repeat a block of code as long as a specific condition remains true. In this article, we explored the use of while loops for substraction, multiplication, exponentiation and nested loops.

By leverging the flexibility of the while loop, you can implement various algorithms and handle different scenarios efficiently. Keep in mind that while using while loops, it's essential to ensure the condition eventually becomes false to avoid infinite loops.

ANALYSIS OF RECURSION :

Many algorithms are recursive. When we analyze them, we get a recurrence relation for time complexity. We get running time on an input of size 'n' as a function of n and the running time on inputs of smaller size.

- For example, In Merge Sort, to sort a given array, we divide it into two halves. Finally, we merge the results. Time complexity of Merge Sort can be written as $T(n) = 2T(\frac{n}{2}) + cn$.

1. SUBSTITUTION METHOD :

We make a guess for the solution and then we use mathematical induction to prove the guess is correct or incorrect.

For example consider the recurrence $T(n) = 2T(\frac{n}{2}) + n$
we guess the solution as $T(n) = O(n\log n)$.
Now we use induction to prove our guess:

We guess the need to prove that $T(n) \le c^{*}n\log n$
We can assume that it is true for values smaller than n.

$$T(n) = 2T(\tfrac{n}{2}) + n$$

$$T(n) \le 2c(\tfrac{n}{2}\log(\tfrac{n}{2})) + n$$

$$= cn\log n - cn\log 2 + n$$

$$= cn\log n - cn + n$$

$$\le cn\log n.$$

2. RECURRENCE TREE METHOD :-

In this method, we draw a recurrence tree & calculate the time taken by every level of the tree. Finally, we sum the work done at all levels. To draw the recurrence tree, we start from the given recurrence and keep drawing till we find a pattern among levels. The pattern is typically arithmetic or geometric series.

For example, consider the recurrence relation
$T(n) = T(\frac{n}{4}) + T(\frac{n}{2}) + cn^2$.

$$
\begin{array}{ccc}
 & cn^2 & \\
 / & & \backslash \\
T(\frac{n}{4}) & & T(\frac{n}{2})
\end{array}
$$

If we further break down the expression $T(\frac{n}{4})$ & $T(\frac{n}{2})$, we get following recursion tree.

$$
\begin{array}{ccccccc}
 & & & cn^2 & & & \\
 & & / & & \backslash & & \\
 & c(n^2)/16 & & & & c(n^2)/4 & \\
 / & & \backslash & & / & & \backslash \\
T(\frac{n}{16}) & & T(\frac{n}{8}) & & T(\frac{n}{8}) & & T(\frac{n}{4})
\end{array}
$$

Breaking down further gives us following

$$
\begin{array}{ccccccc}
 & & & cn^2 & & & \\
 & & / & & \backslash & & \\
 & c(n^2)/16 & & & & c(n^2)/4 & \\
 / & & \backslash & & / & & \backslash \\
c(n^2)/256 & & c(n^2)/16 & & c(n^2)/64 & & c(n^2)/64 \\
/\ \backslash & & /\ \backslash & & /\ \backslash & & /\ \backslash
\end{array}
$$

To know the value of $T(n)$, we need to calculate the sum of tree nodes level by level. If we sum the above tree level by level, we get the following series,

$T(n) = c(n^2 + 5(n^2)/16 + 25(n^2)/256) + \text{------}$

The above series is a geometrical progression with a ratio of 5/16.

To get the upper bound, we can sum the infinite series. We get the sum as $(n^2)/(1-\frac{5}{16})$ which is $O(n^2)$.

3. MASTER METHOD:

The master method is direct way to get the solution. The master method works only for the following type of recurrences or for recurrences that can be transformed into the following type

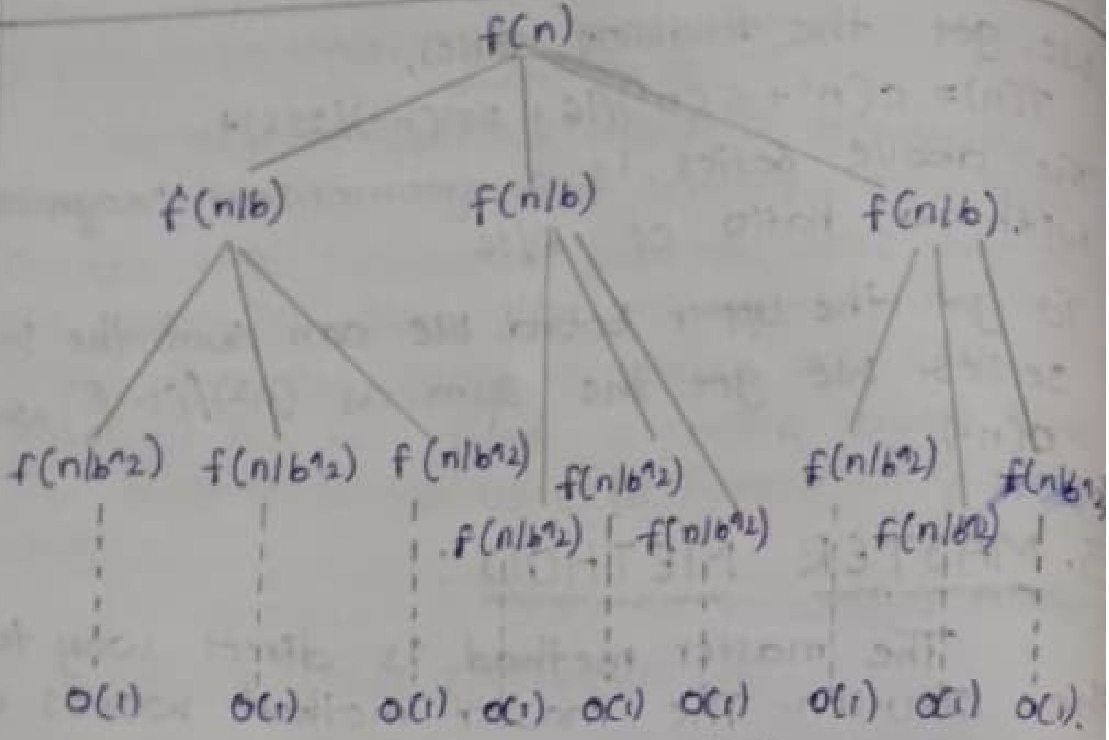
$$T(n) = aT(n/b) + f(n), \text{ where } a \geq 1 \text{ \& } b > 1.$$

There are the following three cases:

- If $f(n) = O(n^c)$, where $c < \log_b a$ then $T(n) = \Theta(n^{\log_b a})$.
- If $f(n) = \Theta(n^c)$, where $c = \log_b a$ then $T(n) = \Theta(n^c \log n)$
- If $f(n) = \Omega(n^c)$, where $c > \log_b a$ then $T(n) = \Theta(f(n))$.

→ HOW DOES THIS WORK?

The master method is mainly derived from the recurrence tree method. If we draw the recurrence tree of $T(n) = aT(\frac{n}{b}) + f(n)$, we can see that the work done at root is $f(n)$, and work done at all leaves is $\Theta(n^c)$ where c is $\log_b a$. And the height of recurrence tree is $\log_b n$.



In the recurrence tree method, we calculate the total work done. If the work done at leaves is polynomially more, then leaves are the dominant part, and our result becomes the work done at leaves (case 1). If work done at leaves and root is asymptotically the same, then our result becomes height multiplied by work done at any level (case 2). If work done at the root is asymptotically more, then our result becomes work done at the root (case 3).

- Examples of some standard algorithms whose time complexity can be evaluated using the Master method:
- MERGE SORT : $T(n) = 2T(\frac{n}{2}) + \Theta(n)$. It falls in case 2 as c is 1 and $\log_b a$ is also 1. so, the solution is $\Theta(n \log n)$.
- BINARY SEARCH : $T(n) = T(\frac{n}{2}) + \Theta(1)$. It also also falls in case 2 as c is 0 and $\log_b a$ is also 0. So the solution is $\Theta(\log n)$.