Day 2

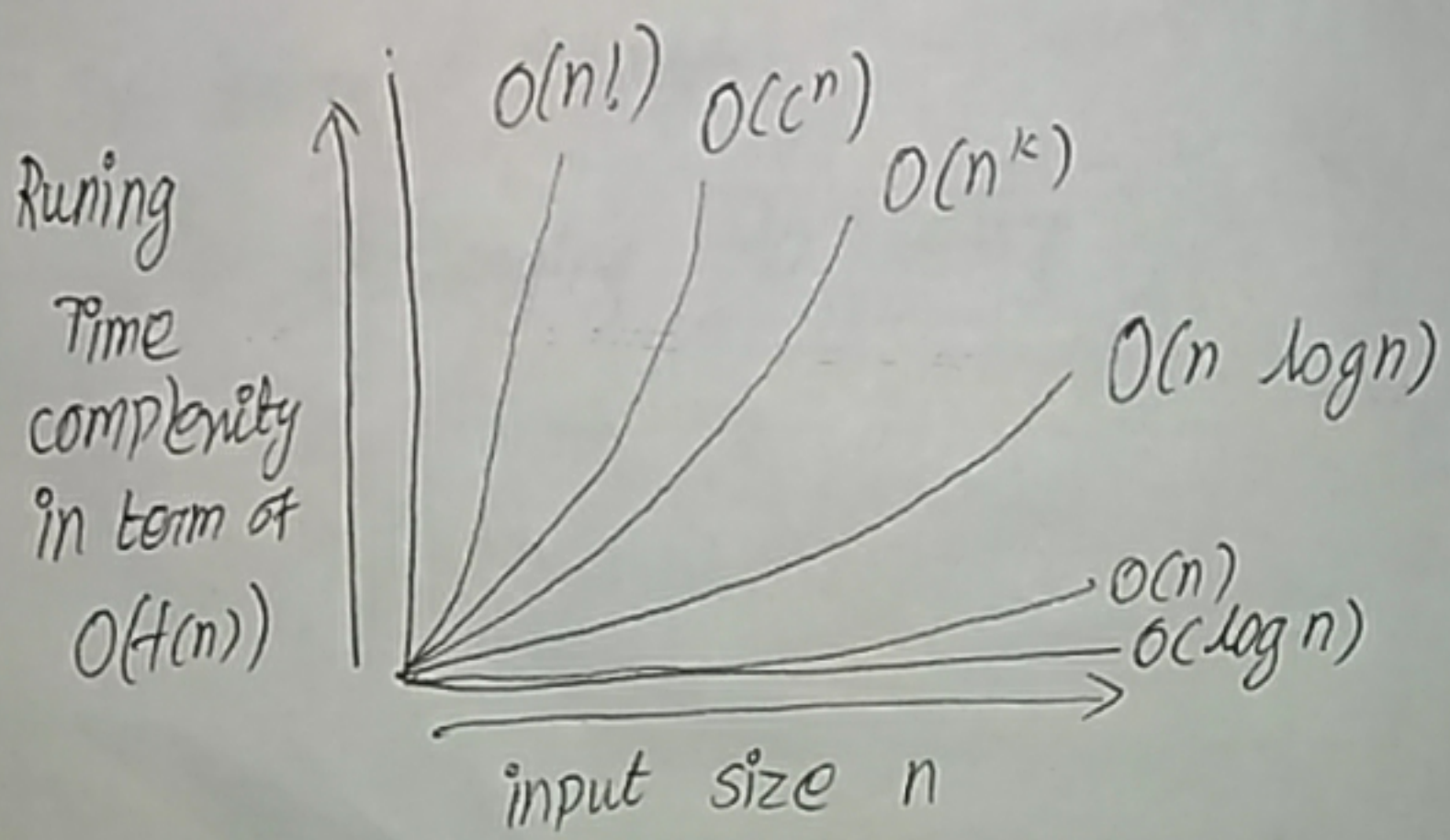
Big O notation and time Complexity

(time to takes to run your function)

Types of Complexity:-

* O(1) - Constant runing time - Best
* O(log n) - logarithmic runing time - Good (Binary search) (Linear Search)
* O(n) - linear runing time - Fair
* O(n log n) - log - linear runing time (merge sort)
* O(n^k) - polynomial runing time (Bubble sort)
* O(2^n) - Enponential runing time (tower of Honai)
* O(n!) - Factorial runing time (Brute force search algorithm)

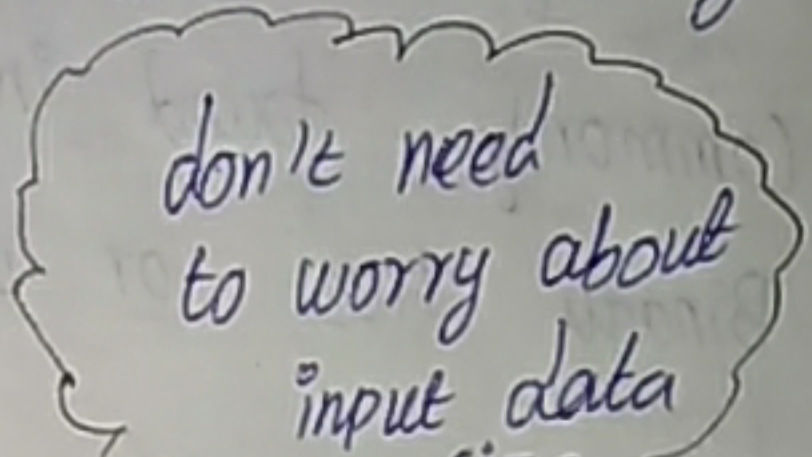
Bad ← Worst case



* Time Complexity : A way of showing how the runtime of a function increases as the size of input increases.

Constant Runing Time - O(1)

An algorithm is said to have a constant time when it is not dependent on the input data (n). no matter the size of input data, the runing time will always be the same.



Example :-

```
if a>b:
    return True
else :
    return False
```

logarithmic Runing Time - O(log n)

A logarithmic time Complexity when it reduce the size of the input data in each steb.

Example :-

```
for index in range (0, len(data), 3):
    print (data[index])
```

"don't need to look at all values of the input data"

* logarithmic time complexity are commonly found in operations on Binary tree or when using binary Search.

Linear Runing Time - O(n)

A linear Time Complexity when the runing time increases at most linearly with the size of the input data. This is the best possible time complexity when the algorithm must examine all values in the input data.

"need to ~~find~~ look at all values in the list. to find the value we are looking for"

Example:-

```
for value in data:
    print (value)
```

* linear time complexity are commonly found in linear search operations.

log - linear Runing Time - O(n log n)

* A quasilinear (or) log linear Runing time Complexity when each operations in the input data have ~~been~~ a logarithm time complexity.

* It is commonly seen in sorting algorithms. (merge sort, timsort, heapsort)

Example:-

for each value in the data1 (O(n))
use the binary search (O(logn)) to search the same value in data2.

```
for value in data1:
    result.append(binary search(data2, value))
```

Polynomial (or) Quadratic Runing time - $O(n^k)$

* Quadratic Runing time Complexity when it needs to perform a linear time operation for each value in the input data.

* "Bubble sort" is a great example of quartic time Complexity since for each value it need to Compare to all other values in the list.

Example :-

```
for x in data:
    for y in data:
        print(x, y)
```

"It ~~nested~~ operation have k "for" loop Then time Complexity is $O(n^k)$"

k - "for" loop count

* If the input size doubles the runtime quadruples.

Exponential Runing Time - $O(2^n)$

* An exponential time Complexity when the growth doubles with each addition to the input data set. This kind of time Complexity is usually seen in brute-force algorithm.

* Using a exponential algorithm to do this, it becomes incredibly resource-expesive to brute-force crack a long password versus a short one. This is one reason that a long password is consider more secure than the shorter one.

Example :-

```
def fibonacci(n):
    if n<=1
        return n
    return fibonacci(n-1) + fibonacci(n-2)
```

~~* If the input size doubles the runtime~~

* If the input size increases by one, The runtime doubles

Factorial Runing Time Complexity - O(n!)

* A factorial Runing time Complexity when it grows in a factorial way based on the size of the input data.

* Heap's algorithm is best example of factorial Runing Complexity. which is used for generating all possible permulations of n object.

```
def Heap_permulation (data, n):
    if n==1:
        print (data)
        return

    for i in range(n):
        Heap_permulation (data, n-1)
        if n%2 ==0
            data[i], data[n-1] = data[n-1], data
        else:
            data[0], data[n-1] = data[n-1], data
```

"it will grow in a factorial way, based on size of the input data"

How to analyzing the time Complexity of an algorithm with several operation?

* we need to dicribe the algorithm based on the largest complexity among all operations.

example:-

```
def func(data):
    first_ele = data[0]                    ⟶ O(1)
    for value in data:
        print (value)                      ⟶ O(n)

    for x in data:
        for y in data:
            print (x,y)                    ⟶ O(n²)
```

* we can see that it has multiple time Complexities: $O(1) + O(n) + O(n^2)$.

Based on this, we can describe the time complexity of this algorithm as worst case ($O(n^2)$).

* If operation has multiple time Complexites, then the time complexity is worst case of the algorithm.