Data Structure and Algorithms

Session-14

Dr. Subhra Rani Patra
SCOPE, VIT Chennai

What & Why of 'Algorithm Run Time Analysis'

- ***What is 'Algo Run Time Analysis' ?***
  - *'It is a study of a given algorithm's running time, by identifying its behavior as the input size for the algorithm increases. In a layman's language we can say, 'how much time will the given algorithm will take to run'.*
- ***Why should we learn this?***
  - *To measure 'efficiency' of a given algorithm.*

- **Time complexity is a description of the asymptotic behavior of running time as input size tends to infinity.**

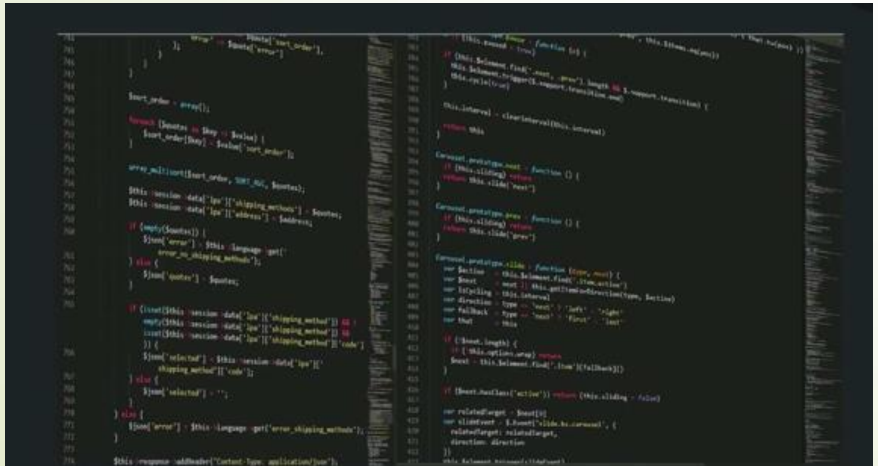
Notations for ‘Algo Run Time Analysis’

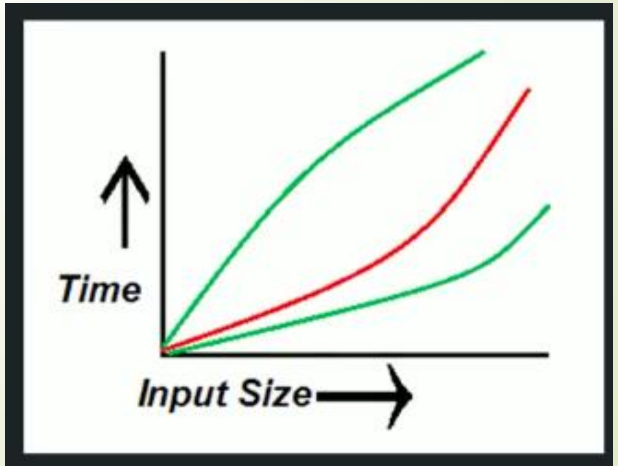
How much does this car runs on 1 litre of petrol ?

- ✓ In City traffic ?
- ✓ On highway ?
- ✓ Mixed environment ?

Notations for 'Algo Run Time Analysis'

- There are 3 notations for 'Run Time Analysis':
  - **Omega(Ω):**
    - This Notation gives the tighter lower bound of a given algorithm.
    - In a layman's language we can say that for any given input, running time of a given algorithm will not be '<u>less than</u>' given time.
- **Big-o(O):**
  - This Notation gives the tighter upper bound of a given algorithm.
  - In a layman's language we can say that for any given input, running time of a given algorithm will not be '<u>more than</u>' given time.
- **Theta(θ):**
  - This Notation decides whether upper bound and lower bound of a given algorithm are same or not.
  - In a layman's language we can say that for any given input, running time of a given algorithm will '<u>on an average</u>' be equal to given time.



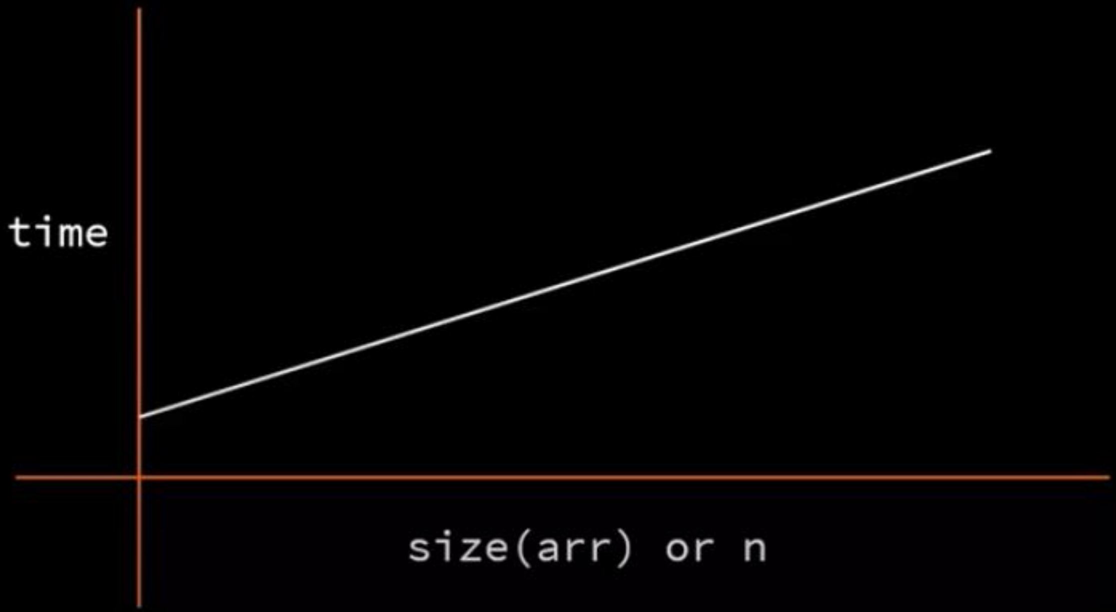


Notations for 'Algo Run Time Analysis'

5	18	3	54	26	...	55	41	...	19	1	10

- ✓ **Omega(Ω):**
  - ✓ *In a layman's language we can say that for any given input, running time of a given algorithm will not be 'less than' given time.*
  - ✓ $\Omega(1)$
- ✓ **Big-o(O):**
  - ✓ *In a layman's language we can say that for any given input, running time of a given algorithm will not be 'more than' given time.*
  - ✓ $O(n)$
- ✓ **Theta(θ):**
  - ✓ *In a layman's language we can say that for any given input, running time of a given algorithm will 'on an average' be equal to given time.*
  - ✓ $\Theta(n/2)$

```
def foo(arr):   size(arr) → 100  → 0.22 milliseconds
   ...          size(arr) → 1000 → 2.30 milliseconds
```



time = a*n + b

↓

1. Keep fastest growing term

↓

time = a*n

↓

2. Drop constants

↓

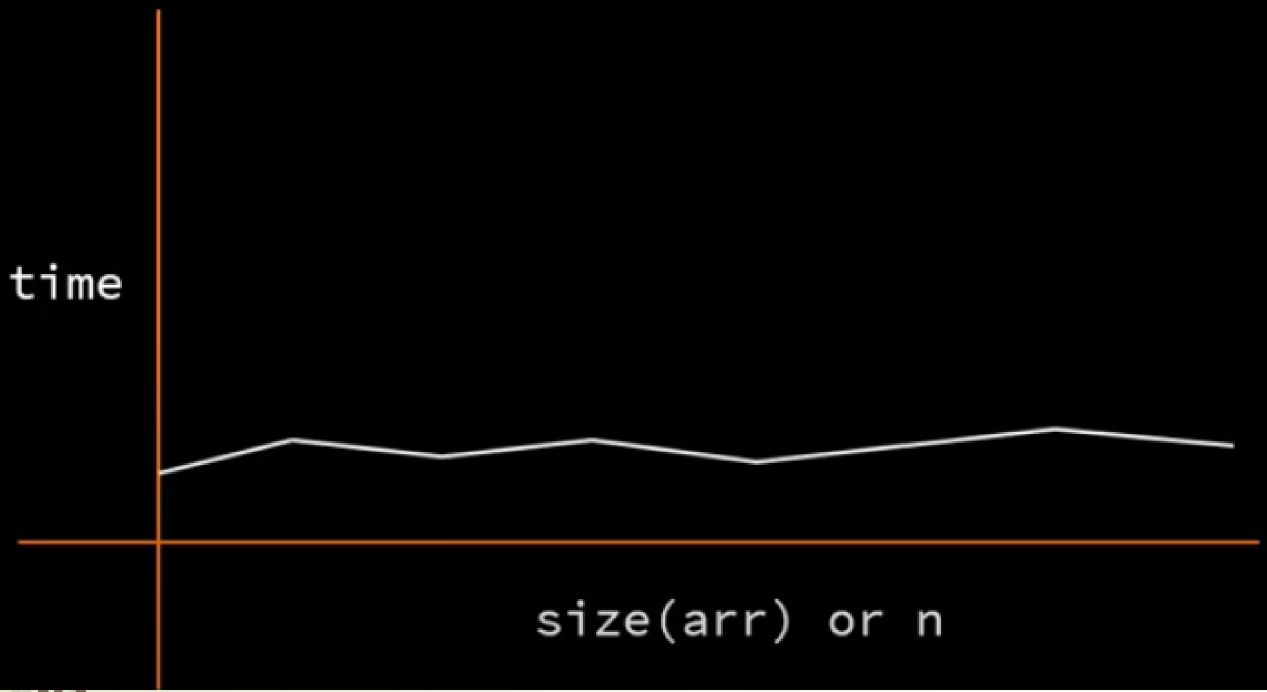
time = O(n)

```
def get_squared_numbers(numbers):
    squared_numbers = []
    for n in numbers:
        square_numbers.append(n*n)
    return squared_numbers


numbers = [2,5,8,9]
get_square_numbers(numbers)
# returns [4,25,64,81]
```

O(n)

```
def foo(a):     size(arr) → 100  → 0.22 milliseconds
   ...          size(arr) → 1000 → 0.23 milliseconds
```



time = a

↓

1. Keep fastest growing term
2. Drop constants

↓

time = O(1)

```python
def find_first_pe(prices, eps, index):
    pe = prices[index]/eps[index]
    return pe
```

O(1)

```python
numbers = [3,6,2,4,3,6,8,9]

for i in range(len(numbers)):
    for j in range(i+1, len(numbers)):
        if numbers[i] == numbers[j]:
            print(numbers[i] + " is a duplicate")
            break
```

$$\text{time} = a * n^2 + b \rightarrow O(n^2)$$

```python
numbers = [3,6,2,4,3,6,8,9]
duplicate = None
for i in range(len(numbers)):
    for j in range(i+1, len(numbers)):
        if numbers[i] == numbers[j]:
            duplicate = numbers[i]
            break

for i in range(len(numbers)):
    if numbers[i] == duplicate:
        print(i)
```

n^2 iterations

n iterations

time = $a*n^2 + b*n + c$

1. Keep fastest growing term
2. Drop constants

BigO refers to very large value of n. Hence if you have a function like,

time = $5*n^2$ + 3*n + 20

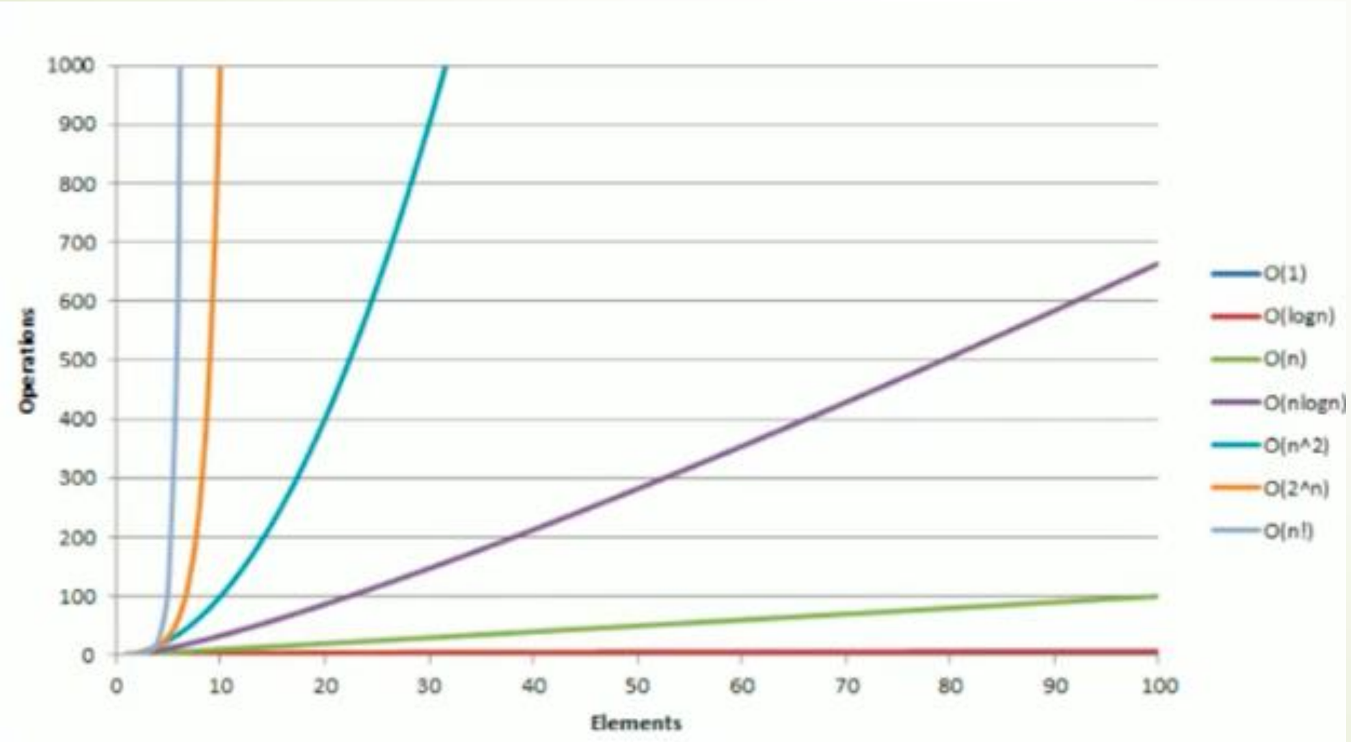
When value of n is very large **b*n + c** become irrevalant

Example: n = 1000

time = $5*1000^2$ + 3*1000 + 20
time = 5000000 + 3020

Examples of ‘Algorithm run time complexities’ :

Time Complexity	Name	Example
O(1)	Constant	Adding and element at front of linked list
O(log n)	Logarithmic	Finding an element in sorted array
O(n)	Linear	Findin an elemnet in unsorted array
O(n logn)	Linear Logarithmic	Merge Sort
$O(n^2)$	Quadratic	Shortest path between 2 nodes in a graph
$O(n^3)$	Cubic	Matrix Multiplication
$O(2^n)$	Exponential	Tower of Hanoi Problem



How to Calculate 'Algorithm Time Complexity' ?

Example#1: Time Complexity of 'Iterative Algo'

5	18	3	54	26	...	55	41	...	19	1	10

```
FindBiggestNumber (int arr[ ]):
    biggestNumber = arr[0]
    loop: i = 1 to length(arr)-1
        if arr[ i ] > biggestNumber
            biggestNumber = arr[ i ]
    return biggestNumber
```

Example#1: Time Complexity of 'Iterative Algo'(continued)

```
FindBiggestNumber (int arr[]):
    biggestNumber = arr[0] ------------------------------------------------ O(1)
    loop: i = 1 to length(arr)-1 ----------------------- O(n)  ]
        if arr[ i ] > biggestNumber ------- O(1) ]  -------- O(1) ] --------- O(n)
            biggestNumber = arr[ i ] ------- O(1) ]
    return biggestNumber ------------------------------------------------- O(1)
```

Time Complexity = $O(1) + O(n) + O(1)$

Example#2: Time Complexity of 'Recursive Algo'

5	18	3	54	26	...	55	41	...	19	1	10

```
FindBiggestNumber(A,n):
    static highest = Integer.Min
    if  n equals -1
      return highest
   else
       if A[n] > highest
          update highest
    return FindBiggestNumber(A,  n-1)
```

```
FindBiggestNumber(A,n): ---------------------------- T(n)
    static highest = Integer.Min---------------------- O(1)
    if  (n equals -1) --------------------------------- O(1)
      return highest ---------------------------------- O(1)
    else  ------------------------------------------------ O(1)
      if A[n] > highest -------------------------------- O(1)
        update highest -------------------------------- O(1)
    return FindBiggestNumber(A,  n-1) --------------- T(n-1)
```

Back Substitution:

$T(n) = O(1) + T(n-1)$ -------------------- Equation#1

$T(-1) = O(1)$ ------------------------------------ Base Condition

$T(n-1) = O(1) + T((n-1)-1)$ --------------- Equation#2

$T(n-2) = O(1) + T((n-2)-1)$ --------------- Equation#3

$$
\begin{aligned}
T(n) &= 1 + T(n-1) \\
&= 1 + (1 + T((n-1)-1)\ ) \\
&= 2 + T(n-2) \\
&= 2 + 1 + T((n-2)-1) \\
&= 3 + T(n-3) \\
&= k + T(n-k) \\
&= (n+1) + T(n-(n+1)) \\
&= n+1 + T(-1) \\
&= n+1+1 \\
&= O(n)
\end{aligned}
$$

Space Complexity

- **Space complexity measures the total amount of memory that an algorithm or operation needs to run according to its input size.**

1) fixed part
2) variable part

Algorithm abc(x, y, z)

return x * y * z * (x - y);

$$S(P) = \underset{\substack{\downarrow \\ \text{fixed}}}{C} + \underset{\substack{\downarrow \\ \text{variable}}}{S_p}$$

Algorithm Sum(x, n)

{

total := 0

for i – 1 to n do

total := total + x[i]

}

$S(P) = C + S_p$

$S(P) = 3 + n$


Thank you