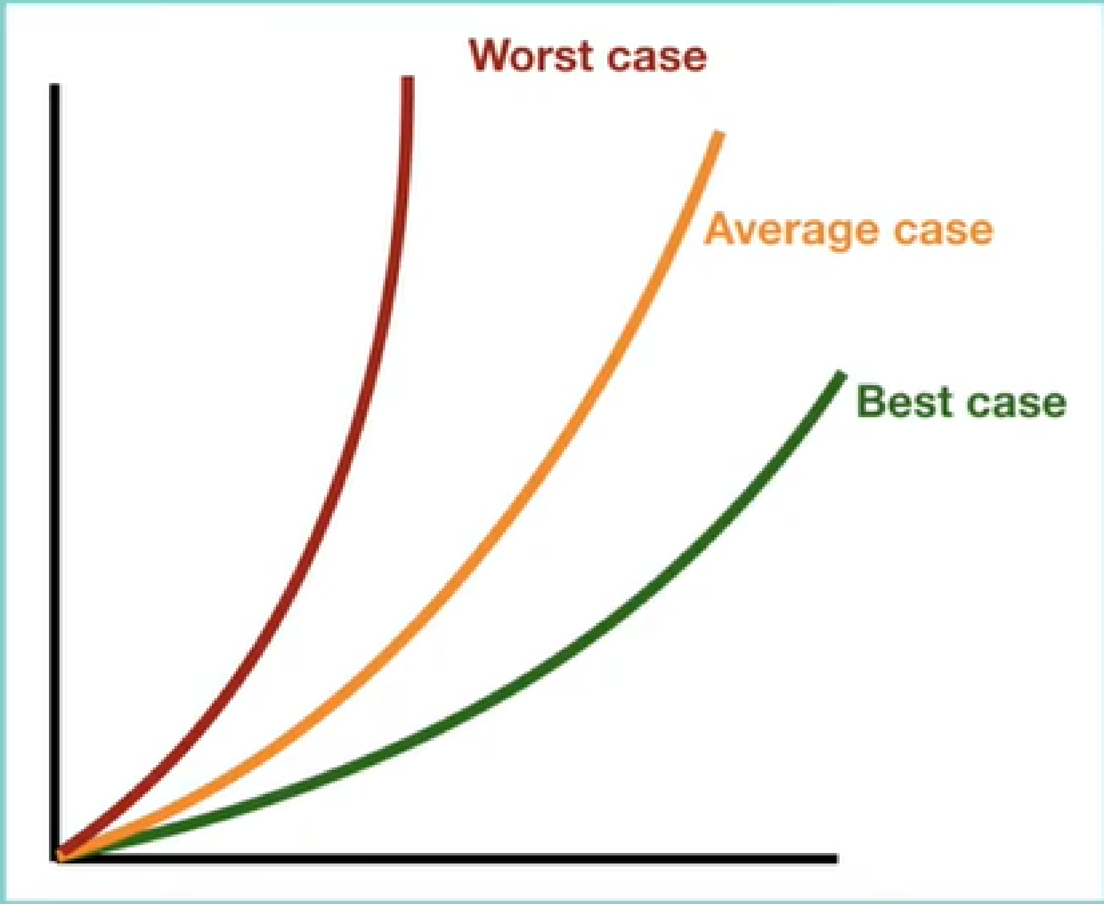
Big O Notations

Algorithm run time notations

- City traffic - 20 liters
- Highway - 10 liters
- Mixed condition - 15 liters

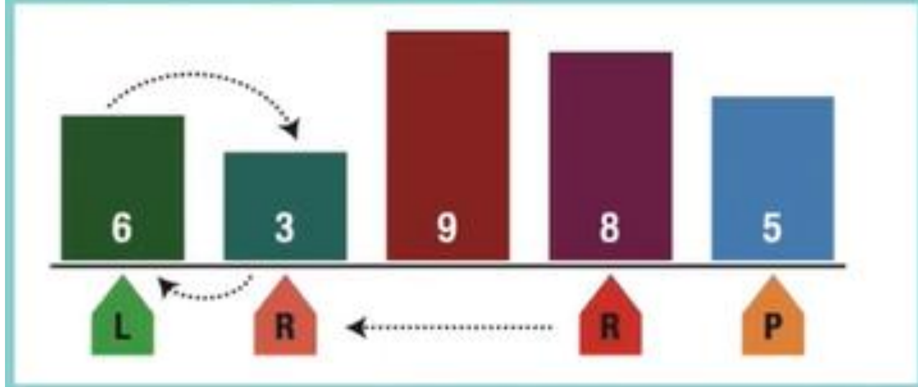
Algorithm run time notations

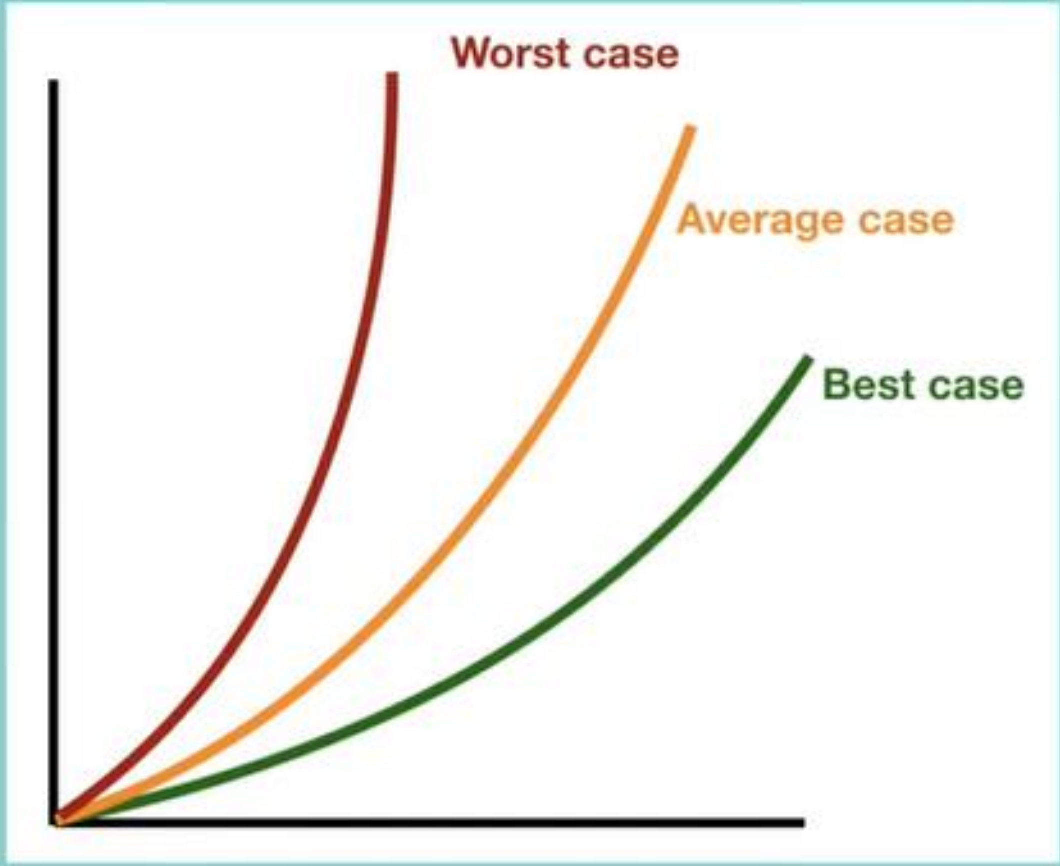
- Best case
- Average case
- Worst case



Algorithm run time notations

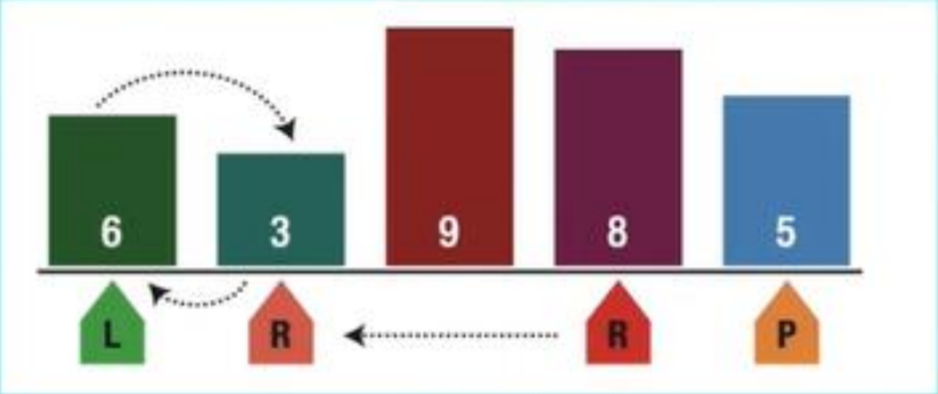
Quick sort algorithm

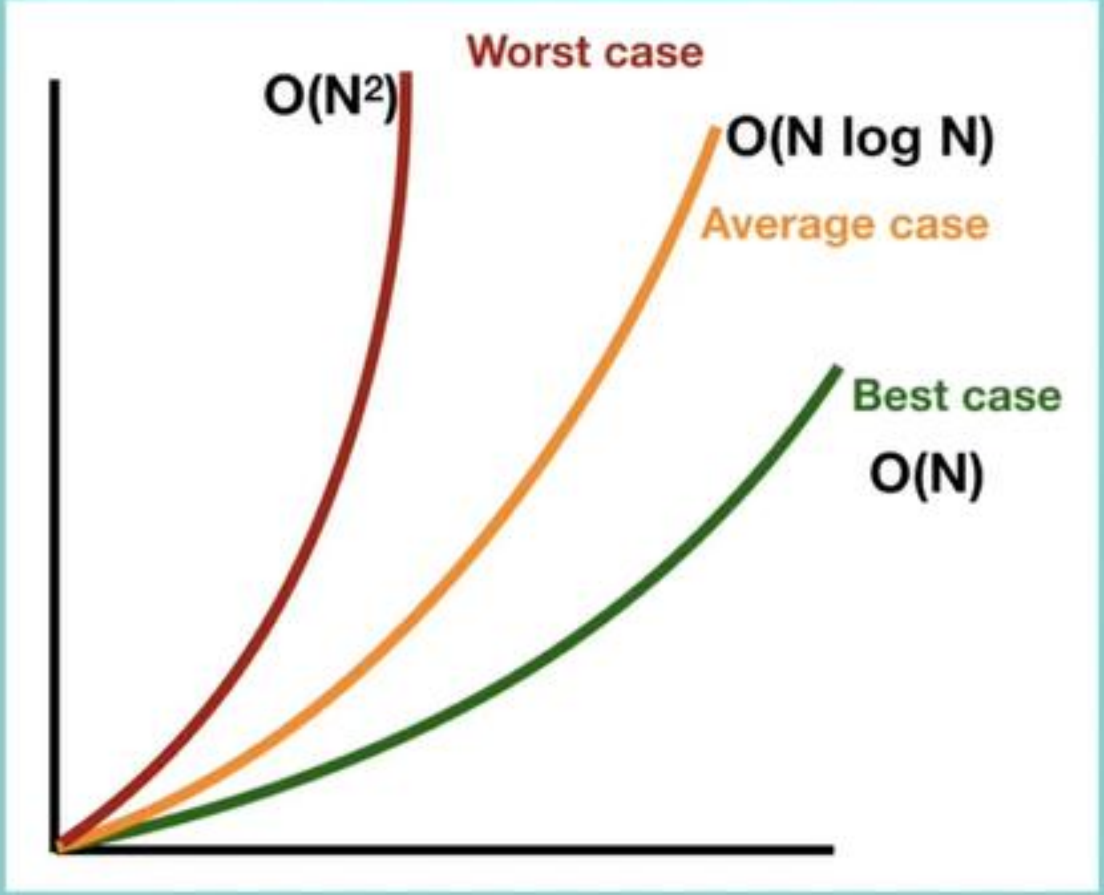




Algorithm run time notations

Quick sort algorithm

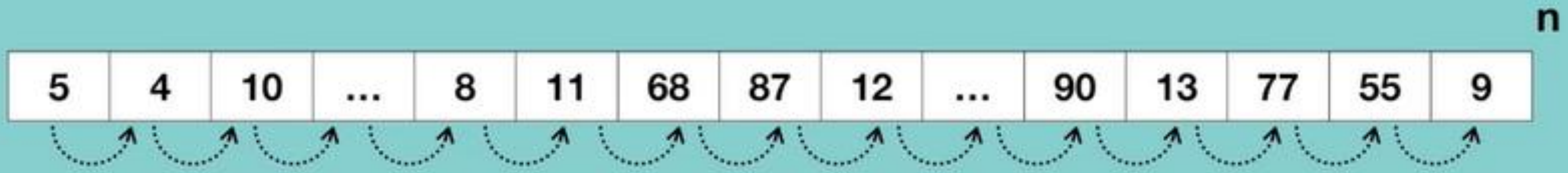


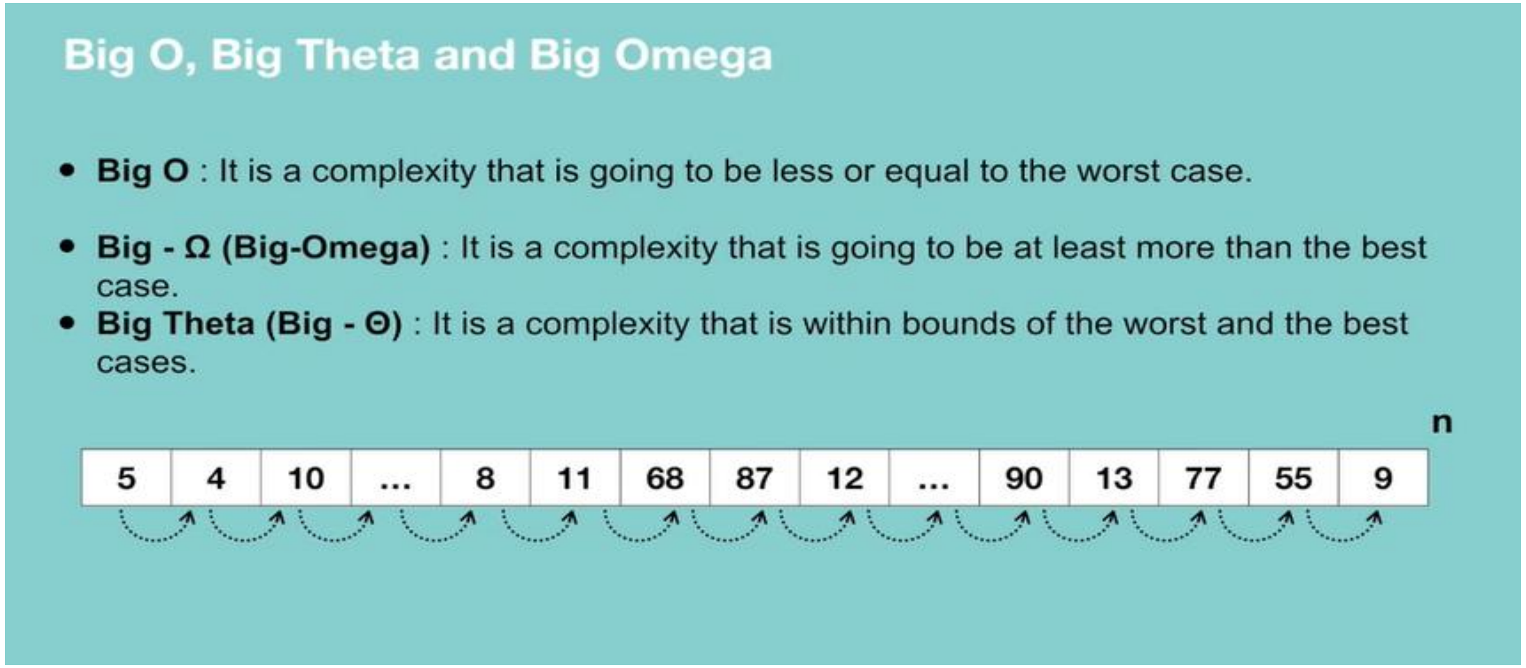


Big O, Big Theta and Big Omega

- **Big O** : It is a complexity that is going to be less or equal to the worst case.

- **Big - Ω (Big-Omega)** : It is a complexity that is going to be at least more than the best case.
- **Big Theta (Big - Θ)** : It is a complexity that is within bounds of the worst and the best cases.





Big O -> for example, for an algo to execute suppose it take max 10 secs to execute it completely then it can’t cross 10 sec, but it might take 8/9 secs. Here 10 secs is the worst case.

Big Omega -> for example, for an algo to execute suppose it take min 2 secs to execute it completely in best case scenario then it can’t take less 2 sec in any case to execute for any condition (best, avg, and worst). Ex: an algo to sort 100 numbers takes 2 sec(which is already sorted list) takes 2 sec, then it’s a best case scenario, it can’t take less than that for any of the cases.

Big O, Big Theta and Big Omega

- **Big O** : It is a complexity that is going to be less or equal to the worst case.
- **Big - Ω (Big-Omega)** : It is a complexity that is going to be at least more than the best case.
- **Big Theta (Big - Θ)** : It is a complexity that is within bounds of the worst and the best cases.

n

5	4	10	...	8	11	68	87	12	...	90	13	77	55	9

Big O - O(N)

Big Ω - Ω(1)

Big Θ - Θ(n/2)

Algorithm run time complexities

Complexity	Name	Sample
O(1)	Constant	Accessing a specific element in array
O(N)	Linear	Loop through array elements
O(LogN)	Logarithmic	Find an element in sorted array
$O(N^2)$	Quadratic	Looking ar a every index in the array twice
$O(2^N)$	Exponential	Double recursion in Fibonacci

Algorithm run time complexities

Complexity	Name	Sample
O(1)	Constant	Accessing a specific element in array
O(N)	Linear	Loop through array elements
O(LogN)	Logarithmic	Find an element in sorted array
$O(N^2)$	Quadratic	Looking ar a every index in the array twice
$O(2^N)$	Exponential	Double recursion in Fibonacci

O(1) - Constant time

```
array = [1, 2, 3, 4, 5]
array[0] // It takes constant time to access first element
```

Algorithm run time complexities

Complexity	Name	Sample
O(1)	Constant	Accessing a specific element in array
O(N)	Linear	Loop through array elements
O(LogN)	Logarithmic	Find an element in sorted array
$O(N^2)$	Quadratic	Looking ar a every index in the array twice
$O(2^N)$	Exponential	Double recursion in Fibonacci

O(N) - Linear time

```
array = [1, 2, 3, 4, 5]
for element in array:
     print(element)
//linear time since it is visiting every element of array
```

Algorithm run time complexities

Complexity	Name	Sample
O(1)	Constant	Accessing a specific element in array
O(N)	Linear	Loop through array elements
O(LogN)	Logarithmic	Find an element in sorted array
$O(N^2)$	Quadratic	Looking ar a every index in the array twice
$O(2^N)$	Exponential	Double recursion in Fibonacci

O(LogN) - Logarithmic time

```
array = [1, 2, 3, 4, 5]
for index in range(0,len(array),3):
     print(array[index])
//logarithmic time since it is visiting only some elements
```

O(LogN) - Logarithmic time

Binary search

```
search 9 within [1,5,8,9,11,13,15,19,21]
compare 9 to 11 → smaller
search 9 within [1,5,8,9]
compare 9 to 8 → bigger
search 9 within [9]
compare 9 to 9
return
```

```
N = 16
N = 8 /* divide by 2 */
N = 4 /* divide by 2 */
N = 2 /* divide by 2 */
N = 1 /* divide by 2 */
```

$2^k = N \rightarrow \log_2 N = k$

Algorithm run time complexities

Complexity	Name	Sample
$O(1)$	Constant	Accessing a specific element in array
$O(N)$	Linear	Loop through array elements
$O(LogN)$	Logarithmic	Find an element in sorted array
$O(N^2)$	Quadratic	Looking ar a every index in the array twice
$O(2^N)$	Exponential	Double recursion in Fibonacci

$O(N^2)$ - Quadratic time

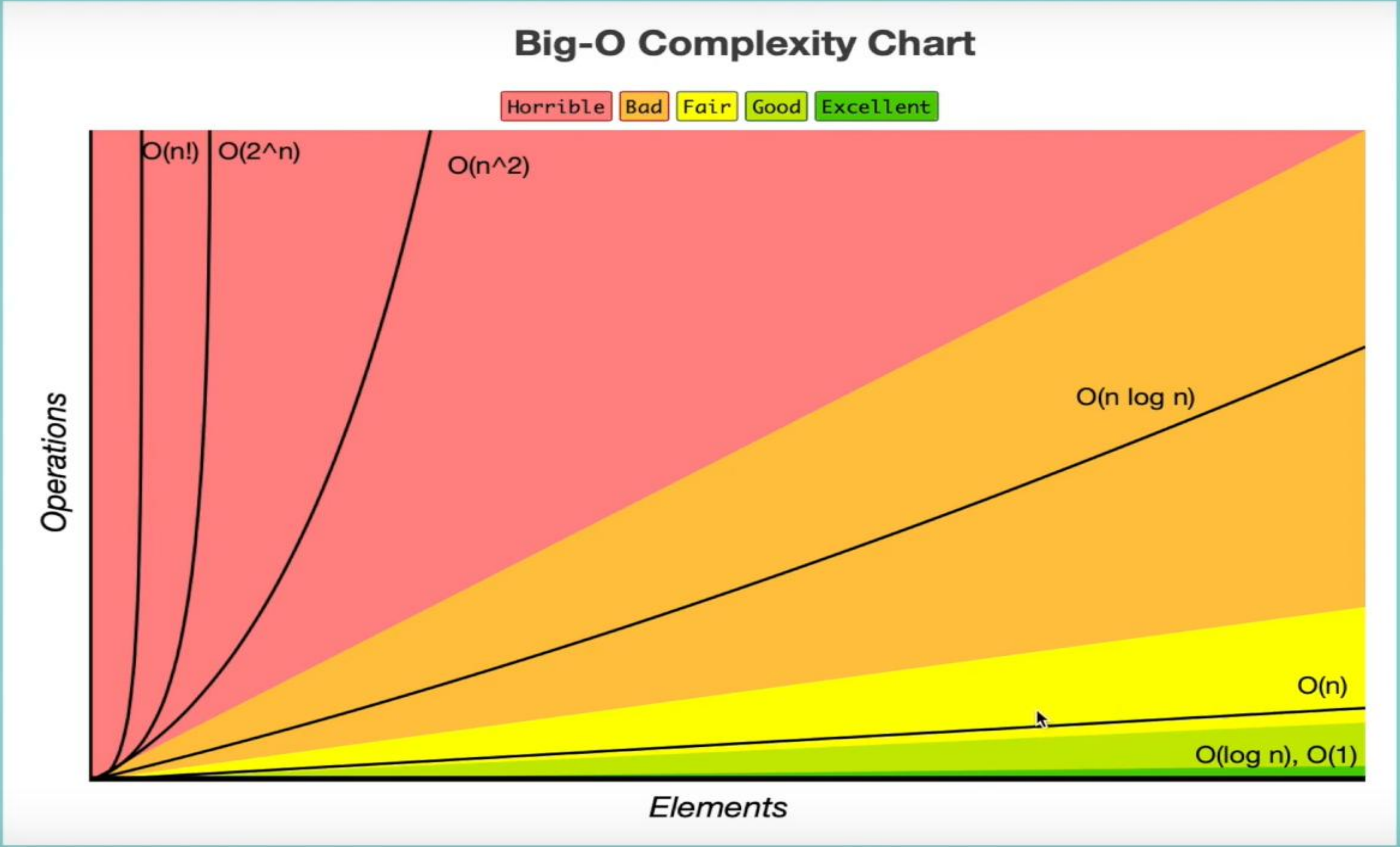
```
array = [1, 2, 3, 4, 5]

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

Algorithm run time complexities

Complexity	Name	Sample
O(1)	Constant	Accessing a specific element in array
O(N)	Linear	Loop through array elements
O(LogN)	Logarithmic	Find an element in sorted array
$O(N^2)$	Quadratic	Looking ar a every index in the array twice
$O(2^N)$	Exponential	Double recursion in Fibonacci

$O(2^N)$ - Exponential time

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


Big-O Complexity Chart
Horrible
Bad
Fair
Good
Excellent
O(n!)
O(2^n)
O(n^2)
O(n log n)
O(n)
O(log n), O(1)
Operations
Elements

Space Complexity

Space complexity

an array of size **n**

$$a = \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{bmatrix}$$

O(n)

an array of size **n*n**

$$a = \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \end{bmatrix}$$

$O(n^2)$

Space complexity - example

```
def sum(n):
    if n <= 0:
        return 0
    else:
        return n + sum(n-1)
```

```
1   sum(3)
2     → sum(2)
3         → sum(1)
4             → sum(0)
```

Space complexity : O(n)

Space complexity - example

```
def pairSumSequence(n):
    sum = 0
    for i in range(0,n+1):
        sum = sum + pairSum(i, i+1)
    return sum

def pairSum(a,b):
    return a + b
```

Space complexity : O(1)

Add vs Multiply

```
for a in arrayA:
    print(a)

for b in arrayB:
    print(b)
```

Add the Runtimes: *O(A + B)*

```
for a in arrayA:
    for b in arrayB:
        print(a, b)
```

Multiply the Runtimes: *O(A * B)*

How to measure the codes using Big O?

No	Description	Complexity
Rule 1	Any assignment statements and if statements that are executed once regardless of the size of the problem	O(1)
Rule 2	A simple “for” loop from 0 to n (with no internal loops)	O(n)
Rule 3	A nested loop of the same type takes quadratic time complexity	O(n^2)
Rule 4	A loop, in which the controlling parameter is divided by two at each step	O(log n)
Rule 5	When dealing with multiple statements, just add them up	

No	Description	Complexity
Rule 1	Any assignment statements and if statements that are executed once regardless of the size of the problem	$O(1)$
Rule 2	A simple "for" loop from 0 to n (with no internal loops)	$O(n)$
Rule 3	A nested loop of the same type takes quadratic time complexity	$O(n^2)$
Rule 4	A loop, in which the controlling parameter is divided by two at each step	$O(\log n)$
Rule 5	When dealing with multiple statements, just add them up	

sampleArray

5	4	10	...	8	11	68	87	...

```
def findBiggestNumber(sampleArray):
    biggestNumber = sampleArray[0]
    for index in range(1,len(sampleArray)):
        if sampleArray[index] > biggestNumber:
            biggestNumber = sampleArray[index]
    print(biggestNumber)
```

No	Description	Complexity
Rule 1	Any assignment statements and if statements that are executed once regardless of the size of the problem	$O(1)$
Rule 2	A simple "for" loop from 0 to n (with no internal loops)	$O(n)$
Rule 3	A nested loop of the same type takes quadratic time complexity	$O(n^2)$
Rule 4	A loop, in which the controlling parameter is divided by two at each step	$O(\log n)$
Rule 5	When dealing with multiple statements, just add them up	

sampleArray

5	4	10	...	8	11	68	87	...

```
def findBiggestNumber(sampleArray):
    biggestNumber = sampleArray[0] ------------------------------------> O(1)
    for index in range(1,len(sampleArray)):-----------------------> O(n)
        if sampleArray[index] > biggestNumber: ------> O(1)
            biggestNumber = sampleArray[index] ------> O(1)
    print(biggestNumber) ------------------------------------------------> O(1)
```

No	Description	Complexity
Rule 1	Any assignment statements and if statements that are executed once regardless of the size of the problem	O(1)
Rule 2	A simple “for” loop from 0 to n (with no internal loops)	O(n)
Rule 3	A nested loop of the same type takes quadratic time complexity	O(n^2)
Rule 4	A loop, in which the controlling parameter is divided by two at each step	O(log n)
Rule 5	When dealing with multiple statements, just add them up	

sampleArray	5	4	10	...	8	11	68	87	...

```
def findBiggestNumber(sampleArray):
    biggestNumber = sampleArray[0]                  O(1)
    for index in range(1,len(sampleArray)):         O(n)
        if sampleArray[index] > biggestNumber:      O(1)          O(n)
            biggestNumber = sampleArray[index]      O(1)   O(1)
    print(biggestNumber)                            O(1)
```

Time complexity : O(1) + O(n) + O(1) = O(n)

How to measure Recursive Algorithm?

sampleArray	5	4	10	...	8	11	68	87	10

```
def findMaxNumRec(sampleArray, n):
    if n == 1:
       return sampleArray[0]
    return max(sampleArray[n-1],findMaxNumRec(sampleArray, n-1))
```

sampleArray

5	4	10	...	8	11	68	87	10

```python
def findMaxNumRec(sampleArray, n):
    if n == 1:
       return sampleArray[0]
    return max(sampleArray[n-1],findMaxNumRec(sampleArray, n-1))
```

Explanation:

A =

11	4	12	7

n = 4

```
findMaxNumRec(A,4)
      ↓
findMaxNumRec(A,3)
      ↓
findMaxNumRec(A,2)
      ↓
findMaxNumRec(A,1) ⟶ A[0]=11
```

```python
def findMaxNumRec(sampleArray, n):
    if n == 1:
        return sampleArray[0]
    return max(sampleArray[n-1],findMaxNumRec(sampleArray, n-1))
```

Explanation:

A =

11	**4**	**12**	**7**

n = 4

```
findMaxNumRec(A,4) → max(A[4-1],12) → max(7,12)=12
        ↓
findMaxNumRec(A,3) → max(A[3-1],11) → max(12,11)=12
        ↓
findMaxNumRec(A,2) → max(A[2-1],11) → max(4,11)=11
        ↓
findMaxNumRec(A,1) → A[0]=11
```

sampleArray

5	4	10	...	8	11	68	87	10

```
def findMaxNumRec(sampleArray, n): --------------------> M(n)
    if n == 1: ----------------------------------------> O(1)
        return sampleArray[0] -------------------------> O(1)
    return max(sampleArray[n-1],findMaxNumRec(sampleArray, n-1))--------> M(n-1)
```

$$\left.\begin{array}{l} M(n)=O(1)+M(n-1) \\ M(1)=O(1) \\ M(n-1)=O(1)+M((n-1)-1) \\ M(n-2)=O(1)+M((n-2)-1) \end{array}\right\}$$

$$\begin{aligned} M(n) &= 1+M(n-1) \\ &= 1+(1+M((n-1)-1)) \\ &= 2+M(n-2) \\ &= 2+1+M((n-2)-1) \\ &= 3+M(n-3) \\ &\;\;\vdots \\ &= a+M(n-a) \\ &= n-1+M(n-(n-1)) \\ &= n-1+1 \\ &= n \end{aligned}$$