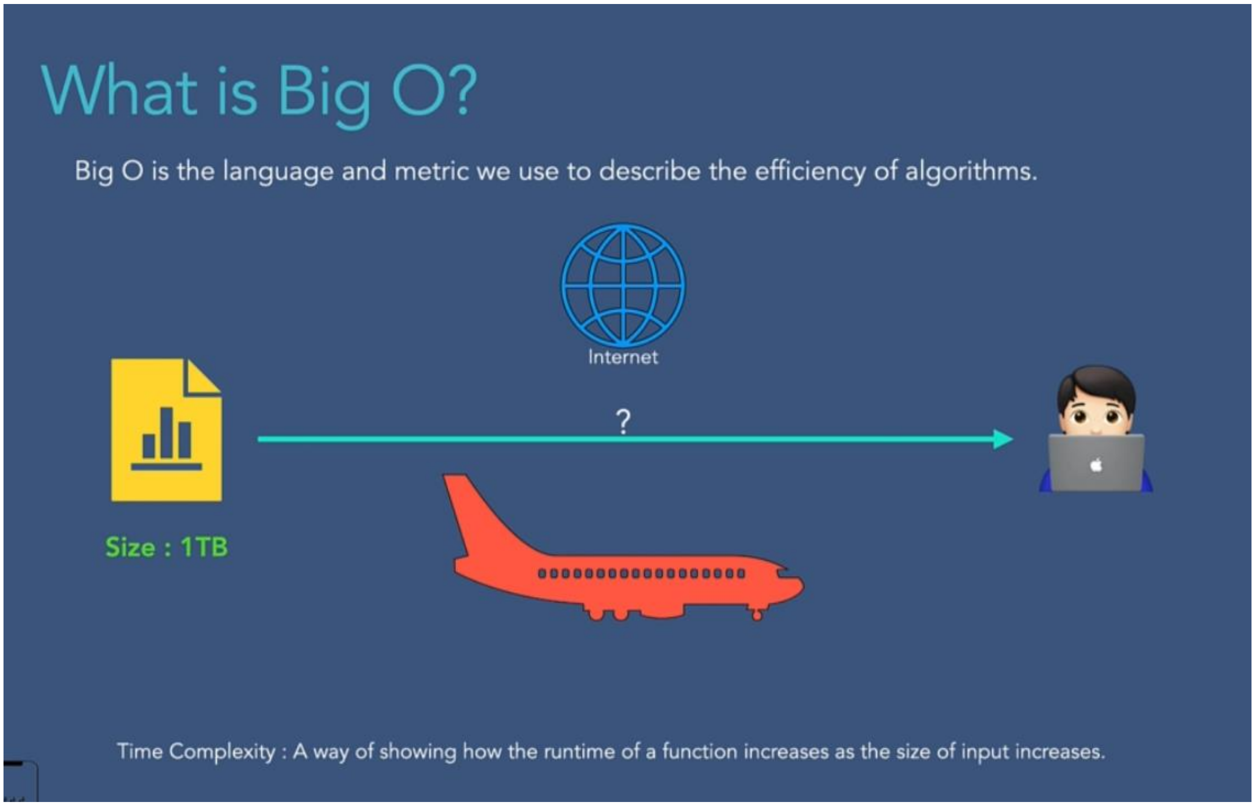
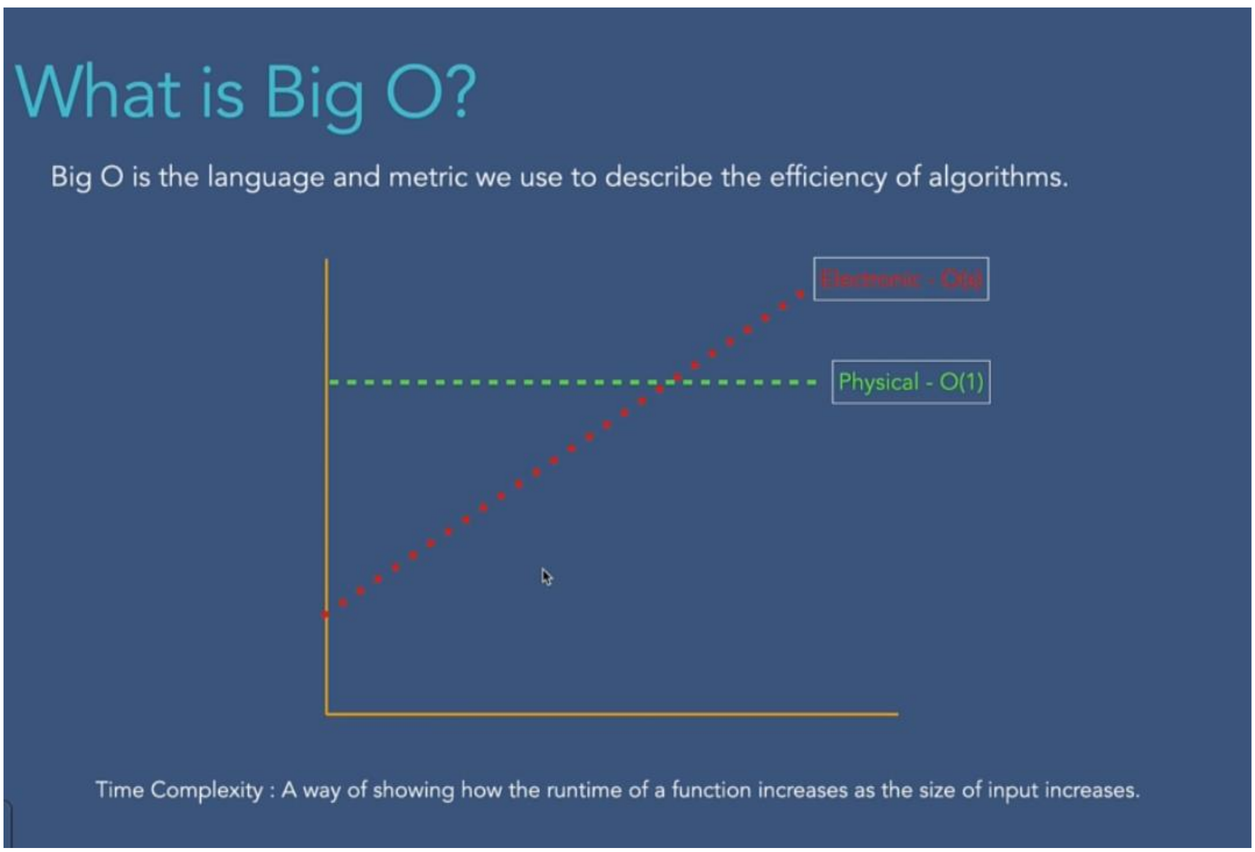

What is Big O?
Big O is the language and metric we use to describe the efficiency of algorithms.
Internet
?
Size : 1TB
Time Complexity : A way of showing how the runtime of a function increases as the size of input increases.


What is Big O?
Big O is the language and metric we use to describe the efficiency of algorithms.
Electronic - O(n)
Physical - O(1)
Time Complexity : A way of showing how the runtime of a function increases as the size of input increases.

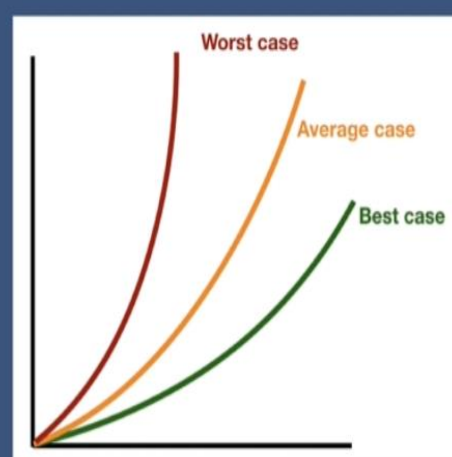
What is Big O?

Types of Runtimes:

$O(N)$, $O(N^2)$, $O(2^N)$

Time complexity : $O(wh)$

Big O Notations

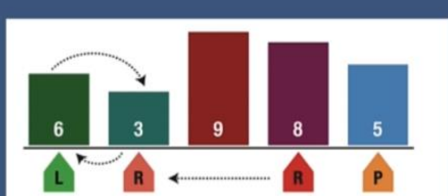


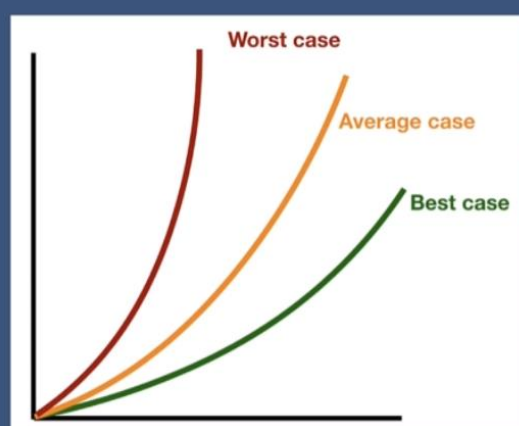
Big O Notations

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

Big O Notations

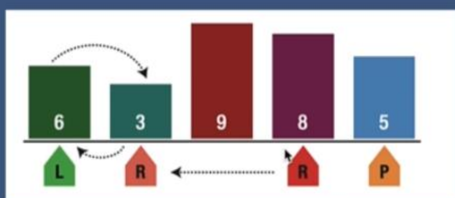
Quick sort algorithm

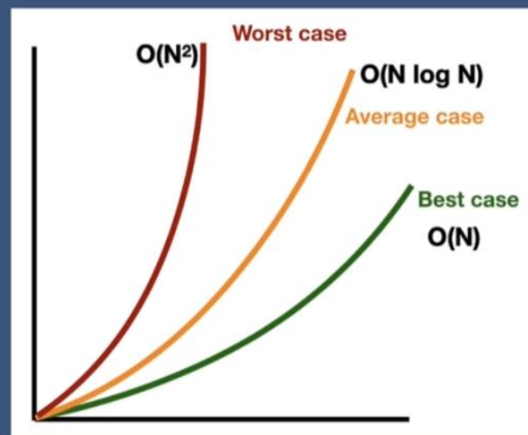




Big O Notations

Quick sort algorithm





Big O Notations

- **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 - O(N)

Big Ω - Ω(1)

Big Θ - Θ(n/2)

Runtime 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

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

Runtime 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

random card

Runtime 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

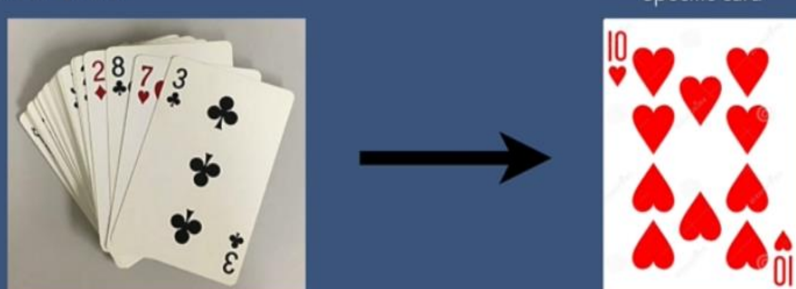
```
int[] custArray = {1, 2, 3, 4, 5}
for (int i = 0; i < custArray.length; i++) {
  System.out.println(custArray[i]);
}
//linear time since it is visiting every element of array
```


Runtime 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

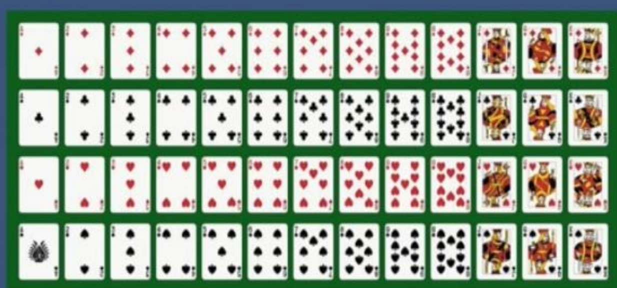
Specific card



Runtime 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

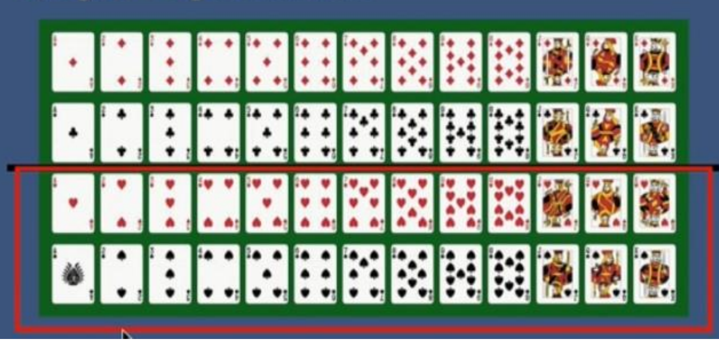
O(Log N) - Logarithmic time



Runtime 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(Log N) - Logarithmic time



Runtime 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(Log N) - 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 */
```

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

Runtime 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

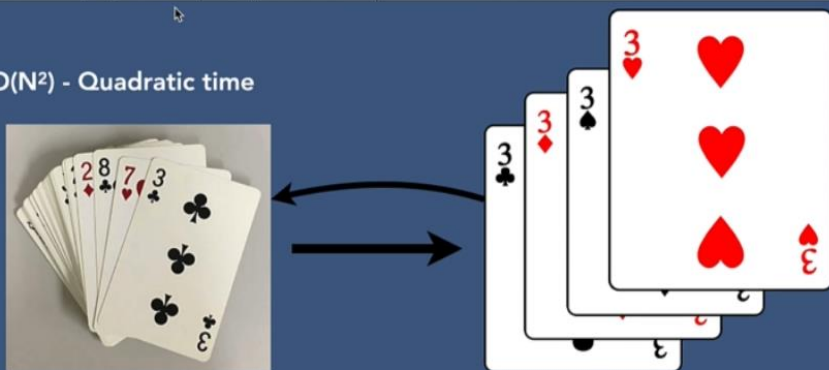
$O(N^2)$ - Quadratic time

```java
int[] custArray = {1, 2, 3, 4, 5}
for (int i = 0; i < custArray.length; i++) {
    for (int j = 0; j < custArray.length; j++) {
        System.out.println(custArray[i]);
    }
}
```

Runtime 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(N²)	Quadratic	Looking ar a every index in the array twice
O(2ᴺ)	Exponential	Double recursion in Fibonacci

$O(N^2)$ - Quadratic time



Runtime Complexities

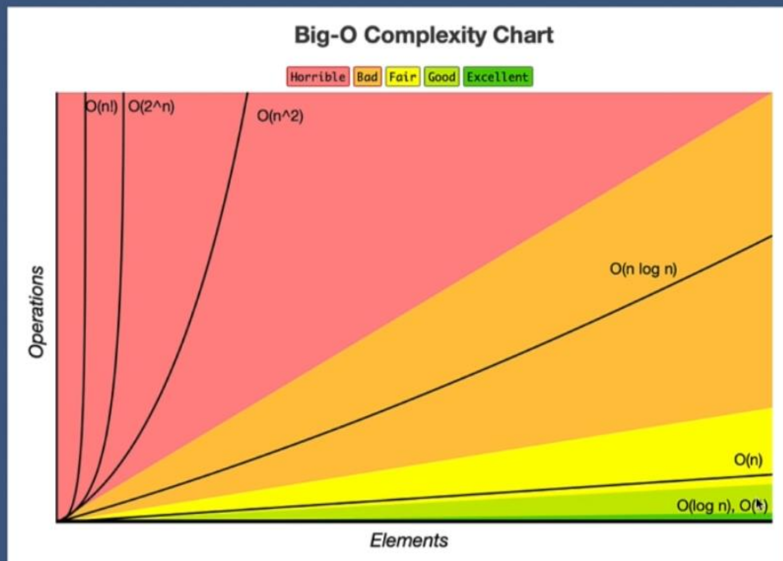
Complexity	Name	Sample
O(1)	Constant	Accessing a specific element in array
O(N)	Linear	Loop through array elements
O(LogN)	Logarithmic	Find an element in sorted array
O(N²)	Quadratic	Looking ar a every index in the array twice
O(2ᴺ)	Exponential	Double recursion in Fibonacci

$O(2^N)$ - Exponential time

```
public int fibonacci(int n) {

    if (n==0 || n==1) {
      return n;
    }
    return fibonacci(n-1) + fibonacci(n-2);
  }
```

Runtime Complexities



Space Complexity

an array of size **n**

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

O(n)

Space Complexity

```
static int sum(int n) {
    if (n <= 0) {
        return 0;
    }
    return n + sum(n-1);
}
```

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

Space complexity : O(n)

Space Complexity

```
static int pairSumSequence(int n) {
    var sum = 0;
    for (int i = 0; i <= n; i++) {
        sum = sum + pairSum(i, i+1);
    }
    return sum;
}

static int pairSum(int a, int b) {
    return a + b;
}
```

Space complexity : O(1)

Drop Constants and Non Dominant Terms

Drop Constant

$O(2N) \longrightarrow O(N)$

Drop Non Dominant Terms

$O(N^2+N) \longrightarrow O(N^2)$

$O(N+\log N) \longrightarrow O(N)$

$O(2*2^N+1000N^{100}) \longrightarrow O(2^N)$

Drop Constants and Non Dominant Terms

- It is very possible that O(N) code is faster than O(1) code for specific inputs
- Different computers with different architectures have different constant factors.

Fast computer
Fast memory access
Lower constant

Slow computer
Slow memory access
Higher constant

- Different algorithms with the same basic idea and computational complexity might have slightly different constants

  Example: a*(b-c) vs a*b - a*c

- As $n \to \infty$, constant factors are not really a big deal

Add vs Multiply

```
for (a=0; arrayA.length; a++) {
        System.out.println(arrayA[a]);
}

for (b=0; arrayB.length; b++) {
        System.out.println(arrayB[b]);
}
```

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

```
for (a=0; arrayA.length; a++) {
     for (b=0; arrayB.length; b++) {
            System.out.println(arrayB[b] + arrayA[a]);
     }
}
```

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

- If your algorithm is in the form "do this, then when you are all done, do that" then you add the runtimes.
- If your algorithm is in the form "do this for each time you do that" then you multiply the runtimes.

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	

sampleArray

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

```
Public static void findBiggestNumber([] sampleArray) {
    var biggestNumber = sampleArray[0];  ------------------> O(1)
    for (index=1; sampleArray.length; index++) {  -------> O(n) }
        if (sampleArray[index] > biggestNumber) {--> O(1) }            } ---> O(n)
            biggestNumber = sampleArray[index];   --> O(1) } --> O(1) }
        }
    }
    System.out.println(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

```
public int findMaxNumRec(int [] sampleArray, int 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
```

How to measure Recursive Algorithm?

sampleArray

5	4	10	...	8	11	68	87	10

```
public int findMaxNumRec(int [] sampleArray, int 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)
```

$M(n)=O(1)+M(n-1)$

$M(1)=O(1)$

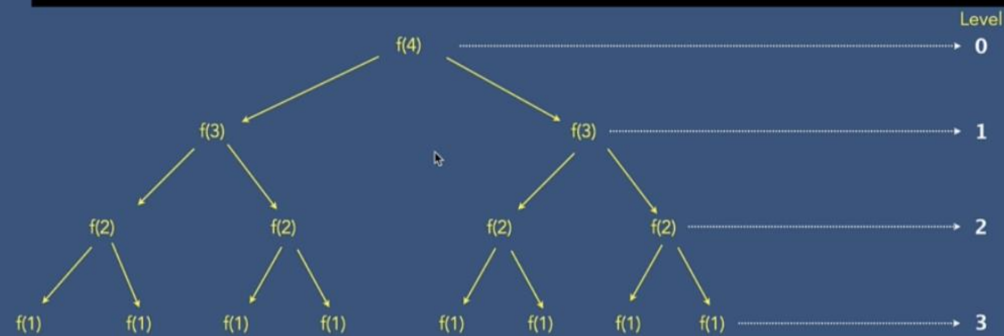
$M(n-1)=O(1)+M((n-1)-1)$

$M(n-2)=O(1)+M((n-2)-1)$

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

How to measure Recursive Algorithm with multiple calls?

```
public int f(int n):
    if (n <= 1) {
        return 1; }
    return f(n-1) + f(n-1);
```



$2^0+2^1+2^2+\ldots+2^n=2^{n+1}-1$

$2^n-1 \longrightarrow O(2^n)$

N	Level	Node#	Also can be expressed..	or..
4	0	1		2^0
3	1	2	2 * previous level = 2	2^1
2	2	4	2 * previous level = 2 $*2^1=2^2$	2^2
1	3	8	2 * previous level = 2 $*2^2=2^3$	2^3

$O(branches^{depth})$