Algorithm Complexity

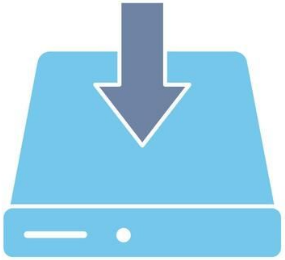
By: Guillermo Andres De Mendoza Corrales

Theory

In simple terms, **algorithm complexity** is a way to measure how much **"effort"** a computer has to put in to run a piece of code.

Rather than measuring time in seconds (which changes depending on whether you have a supercomputer or a potato), we measure how the requirements **grow** as the amount of **data** you give it increases.

Time Complexity



Space Complexity

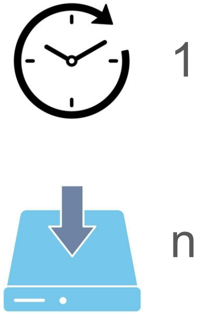
"Big O" Notation

We use **Big O notation** to describe the "worst-case scenario." It's like a label that tells you

Notation	Name	Intuition	Typical Example
$O(1)$	Constant	Time remains the same regardless of input size.	Accessing a specific element in an array.
$O(\log n)$	Logarithmic	Time grows slowly as input grows; usually involves halving the problem.	Binary search in a sorted array.
$O(n)$	Linear	Time grows proportionally to the input size.	Traversing a list or array once.
$O(n \log n)$	Linearithmic	Faster than quadratic, slower than linear. Common in efficient sorting.	Merge Sort, Quick Sort.
$O(n^2)$	Quadratic	Time grows dramatically; usually a nested loop.	Bubble Sort, checking for duplicates with nested loops.
$O(2^n)$	Exponential	Time doubles with each added element. Very slow.	Recursive solutions for the Fibonacci sequence.
$O(n!)$	Factorial	The slowest complexity; grows astronomically fast.	Brute force solutions to the Traveling Salesperson Problem.

O(1)

Takes the same time regardless of size.




```
numbers = [ 3, 5, 12, 43, 45, 53, 77, 87, 143 ]
print( numbers[7] )
```

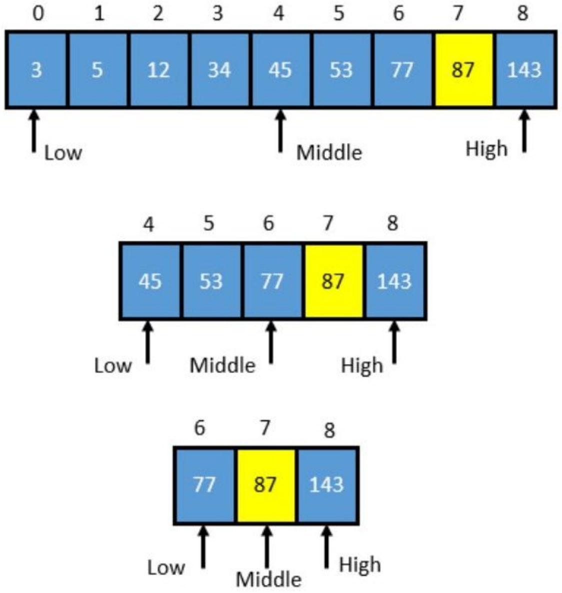
```
87
```

0	1	2	3	4	5	6	7	8
3	5	12	34	45	53	77	87	143

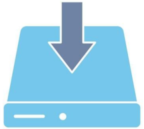
O(log(n))

The "divide and conquer" approach.

```python
def busqueda_binaria(lista, objetivo):
    inicio = 0
    fin = len(lista) - 1

    while inicio <= fin:
        medio = (inicio + fin) // 2
        valor_medio = lista[medio]

        if valor_medio == objetivo:
            return medio
        elif valor_medio > objetivo:
            fin = medio - 1
        else:
            inicio = medio + 1

    return -1

mi_lista = [10, 22, 35, 47, 50, 63, 75, 88, 99]
resultado = busqueda_binaria(mi_lista, 75)

print(f"El número está en el índice: {resultado}")
```




log(n)


n

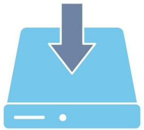
O(n)

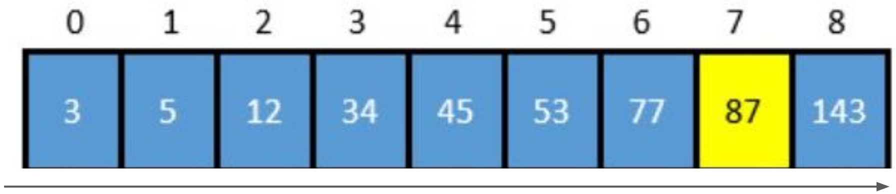
You have to look at every single item once.

```python
def busqueda(lista, objetivo):
   for index in range(len(lista)):
    if lista[index] == objetivo:
      return index

numbers = [ 3, 5, 12, 43, 45, 53, 77, 87, 143 ]
i = busqueda(numbers, 87)
print(f"El número está en el índice: {i}")
```

```
El número está en el índice: 7
```

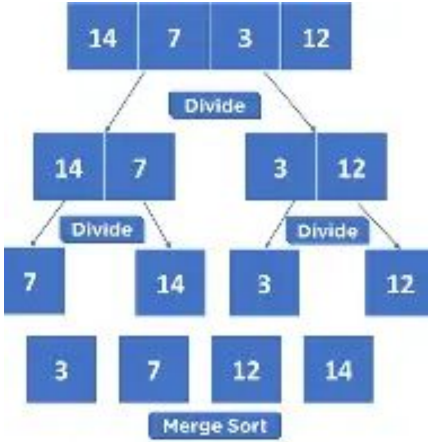

n


n



O(n log(n))

Efficient sorting.




```python
def merge_sort(lista):
    # Caso base: si la lista tiene 1 o 0 elementos, ya está ordenada
    if len(lista) <= 1:
        return lista

    # 1. DIVIDIR: Encontrar el punto medio
    medio = len(lista) // 2
    izquierda = lista[:medio]
    derecha = lista[medio:]

    # Llamadas recursivas para dividir más
    izquierda = merge_sort(izquierda)
    derecha = merge_sort(derecha)

    # 2. MEZCLAR: Combinar las dos mitades ordenadas
    return mezclar(izquierda, derecha)

def mezclar(izquierda, derecha):
    resultado = []
    i = j = 0

    # Comparamos elementos de ambas listas y los añadimos ordenados
    while i < len(izquierda) and j < len(derecha):
        if izquierda[i] < derecha[j]:
            resultado.append(izquierda[i])
            i += 1
        else:
            resultado.append(derecha[j])
            j += 1

    # Añadimos los elementos restantes (si quedan)
    resultado.extend(izquierda[i:])
    resultado.extend(derecha[j:])
    return resultado

# Ejemplo de uso:
desordenada = [38, 27, 43, 3, 9, 82, 10]
ordenada = merge_sort(desordenada)
print(f"Lista ordenada: {ordenada}")
```

n log(n)

n

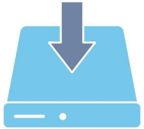
O(n^2)

A loop inside a loop.

```python
def sumar_matrices(matriz_a, matriz_b):
    n = len(matriz_a)
    # Creamos una matriz resultado del mismo tamaño, inicializada en 0
    matriz_resultado = [[0 for _ in range(n)] for _ in range(n)]

    # Bucle externo: recorre las filas
    for i in range(n):
        # Bucle interno: recorre las columnas
        for j in range(n):
            matriz_resultado[i][j] = matriz_a[i][j] + matriz_b[i][j]

    return matriz_resultado

# Ejemplo de uso:
A = [[1, 2],
     [3, 4]]

B = [[5, 6],
     [7, 8]]

resultado = sumar_matrices(A, B)
print("Resultado de la suma:")
for fila in resultado:
    print(fila)
```

```
Resultado de la suma:
[6, 8]
[10, 12]
```

n^2



n^2

O(c^n)

Growth doubles with every new item.

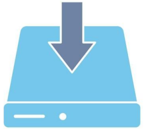
if c = 62 (uppercase + lowercase + numbers) and the size of the password is n = 8:

Total combinations = 62^8 = 218,340,105,584,896.

```python
import itertools
import string

def fuerza_bruta(contrasena_objetivo):
    caracteres = string.ascii_lowercase + string.digits
    longitud_maxima = len(contrasena_objetivo)

    for longitud in range(1, longitud_maxima + 1):
        for combinacion in itertools.product(caracteres, repeat=longitud):
            intento = ''.join(combinacion)

            if intento == contrasena_objetivo:
                return f"Contraseña encontrada: {intento}"

    return "Contraseña no encontrada"

objetivo = "abc"
print(fuerza_bruta(objetivo))
```

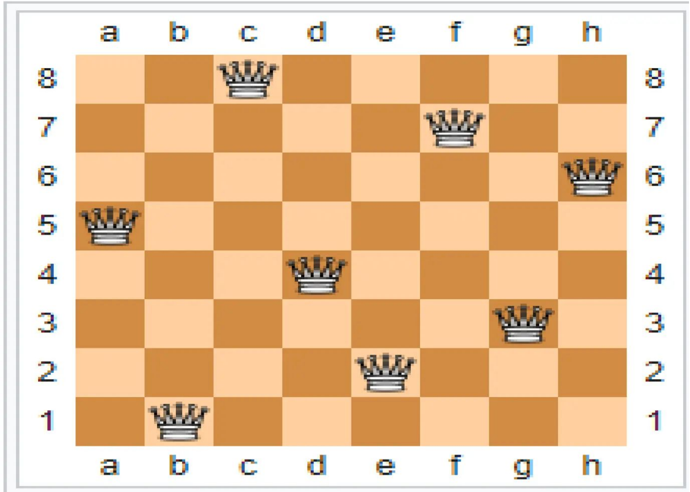
```
Contraseña encontrada: abc
```

n^n

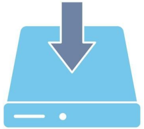


n

O(n!) - HARD NP



n!



n

Big-O Complexity Chart