

CORTE 3

# FECHAS DE IMPORTANTES

- 13, 15 y 20 DE MAYO CLASES (Eficiencia algorítmica y grafos)
- Viernes 15 quiz (Python 1 ejercicio 1 hora)
  - Bitácora de Python curso 1 terminado para poder presentar quiz
- Miercoles 20 Asserts java (panaderia)
- Viernes 22 de mayo EXAMEN FINAL
  - Todo lo de la materia
- MIERCOLES 27 DE MAYO (TRABAJO FINAL DE LA MATERIA)

# ASSERTS EN JAVA

- <http://junit.sourceforge.net/javadoc/org/junit/Assert.html>

|  |  |
|--|--|
| <b>Overview</b> <b>Package</b> <b>Class</b> <b>Tree</b> <b>Deprecated</b> <b>Index</b> <b>Help</b> |  |
| <a href="#">PREV CLASS</a> <a href="#">NEXT CLASS</a>  |  |
| SUMMARY: NESTED   FIELD   <a href="#">CONSTR</a>   <a href="#">METHOD</a>                          |  |
| <a href="#">FRAMES</a> <a href="#">NO FRAMES</a> <a href="#">All Classes</a>                       |  |
| DETAIL: FIELD   <a href="#">CONSTR</a>   <a href="#">METHOD</a>                                    |  |

org.junit

## Class Assert

java.lang.Object  
└─ org.junit.Assert

```
public class Assert
extends java.lang.Object
```

A set of assertion methods useful for writing tests. Only failed assertions are recorded. These methods can be used directly: `Assert.assertEquals(...)`, however, they read better if they are referenced through static import:

```
import static org.junit.Assert.*;
...
assertEquals(...);
```

**See Also:**  
[AssertionError](#)

|                            |   |
|----------------------------|---|
| <b>Constructor Summary</b> |   |
| protected                  | <a href="#">Assert()</a><br>Protect constructor since it is a static only class |

|                       |   |
|-----------------------|---|
| <b>Method Summary</b> |   |
| static void           | <a href="#">assertArrayEquals</a> (byte[] expecteds, byte[] actuals)<br>Asserts that two byte arrays are equal. |
| static void           | <a href="#">assertArrayEquals</a> (char[] expecteds, char[] actuals)<br>Asserts that two char arrays are equal. |
| static void           | <a href="#">assertArrayEquals</a> (int[] expecteds, int[] actuals)<br>Asserts that two int arrays are equal.    |

# ASSERTS - ASERCIONES

- Aserción: es una **condición lógica** insertada en código en java
  - Lógica de proposiciones.
  - Generalmente para validar el concepto de verdad o falsedad
  - Reporte de pruebas
  - Contratos de software (especifico un programa menciono como se va a comportar)
    - Casos de uso menciono (@afert, @before)
      - **Precondiciones** = inicio del código, antes de que se ejecute una rutina
      - **Postcondiciones** = Después del código, después de que se ejecute una rutina

# ASSERTS - ASERCIONES

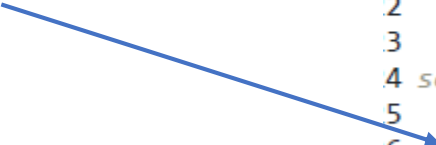
- `assertcondiciónBooleana: [expresión]`

```
public class AssertionExample{  
    public static void main(String[] args) {  
        int x = -15;  
        DataAccess da = new DataAccess();  
        assert x > 0 : "El valor debe ser positivo";  
        System.out.println("Valor positivo x: " + x);  
    }  
}
```

# ASSERTS - ASERCIONES - POSTCONDICION

- Un pequeño ejemplo de los *asserts* podría ser el siguiente en la que en el método *nextNumber* hay una postcondición según la cual el método debe devolver un número entero entre 0 y 9 (incluidos):

```
5 public class Main {
6
7     private Random random;
8
9     public Main() {
10         random = new Random();
11     }
12
13     /**
14      * Devuelve un número entero entre 0 y 9.
15      */
16     public int nextNumber() {
17         int i = random.nextInt(40);
18
19         // Si el cálculo del número fuese más complejo incluyendo un
20         // assert
21         // podemos asegurar en tiempo de desarrollo en esta postcondición
22         // el valor generado por este método.
23         // La línea de código anterior según el contrato del método debería
24         // ser:
25         // int i = random.nextInt(10);
26         assert i >= 0 && i < 10: String.format("El número devuelto no cumple
27 la postcondición (%d)", i);
28
29         return i;
30     }
31
32     public static void main(String[] args) {
33         Main main = new Main();
34         System.out.println(main.nextNumber());
35     }
36 }
```



# ASSERTS - ASERCIONES - PRECONDICIONES

```
/**
 * Constructor de la clase Empleado.
 *
 * @param nombre Nombre del empleado.
 * @param edad Edad del empleado en años.
 * @param nif NIF del empleado.
 */
public Empleado(String nombre, int edad, String nif){
    → assert(!nombre.equals(""));
      this.nombre = nombre;

    → assert(edad > 0);
      this.edad = edad;

    → assert( (!nif.equals("")) && (nif.length() == 9) );
      this.nif = nif;
}
```

## 2.1. Annotations

| Annotation         | Description  |
|--------------------|--|
| @Test              | Denotes a method that is a test. It is not declared in the JUnit 4 API and is dedicated to JUnit 5.    |
| @ParameterizedTest | Denotes a test method that is parameterized. It is inherited from JUnit 4's @Parameterized.            |
| @RepeatedTest      | Denotes a test method that is repeated. It is inherited from JUnit 4's @Repeat.                        |
| @TestFactory       | Denotes a method that returns a stream of test instances. It is inherited from JUnit 4's @TestFactory. |
| @TestTemplate      | Denotes a test method that is a template. It is inherited from JUnit 4's @TestTemplate.                |
| @TestMethodOrder   | Used to specify the order of test methods. It is inherited from JUnit 4's @FixMethodOrder.             |
| @TestInstance      | Used to specify the test instance. It is inherited from JUnit 4's @TestInstance.                       |

|             |  |
|-------------|--|
| @BeforeEach | Denotes that the annotated method should be executed <i>before each</i> @Test , @RepeatedTest , @ParameterizedTest , or @TestFactory method in the current class; analogous to JUnit 4's @Before . Such methods are <i>inherited</i> unless they are <i>overridden</i> .   |
| @AfterEach  | Denotes that the annotated method should be executed <i>after each</i> @Test , @RepeatedTest , @ParameterizedTest , or @TestFactory method in the current class; analogous to JUnit 4's @After . Such methods are <i>inherited</i> unless they are <i>overridden</i> .   |
| @BeforeAll  | Denotes that the annotated method should be executed <i>before all</i> @Test , @RepeatedTest , @ParameterizedTest , and @TestFactory methods in the current class; analogous to JUnit 4's @BeforeClass . Such methods are <i>inherited</i> (unless they are <i>hidden</i> or <i>overridden</i> ) and must be <i>static</i> (unless the "per-class" <a href="#">test instance lifecycle</a> is used). |
| @AfterAll   | Denotes that the annotated method should be executed <i>after all</i> @Test , @RepeatedTest , @ParameterizedTest , and @TestFactory methods in the current class; analogous to JUnit 4's @AfterClass . Such methods are <i>inherited</i> (unless they are <i>hidden</i> or <i>overridden</i> ) and must be <i>static</i> (unless the "per-class" <a href="#">test instance lifecycle</a> is used).   |

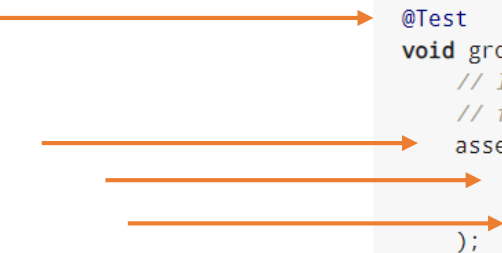
<https://junit.org/junit5/docs/current/user-guide/>



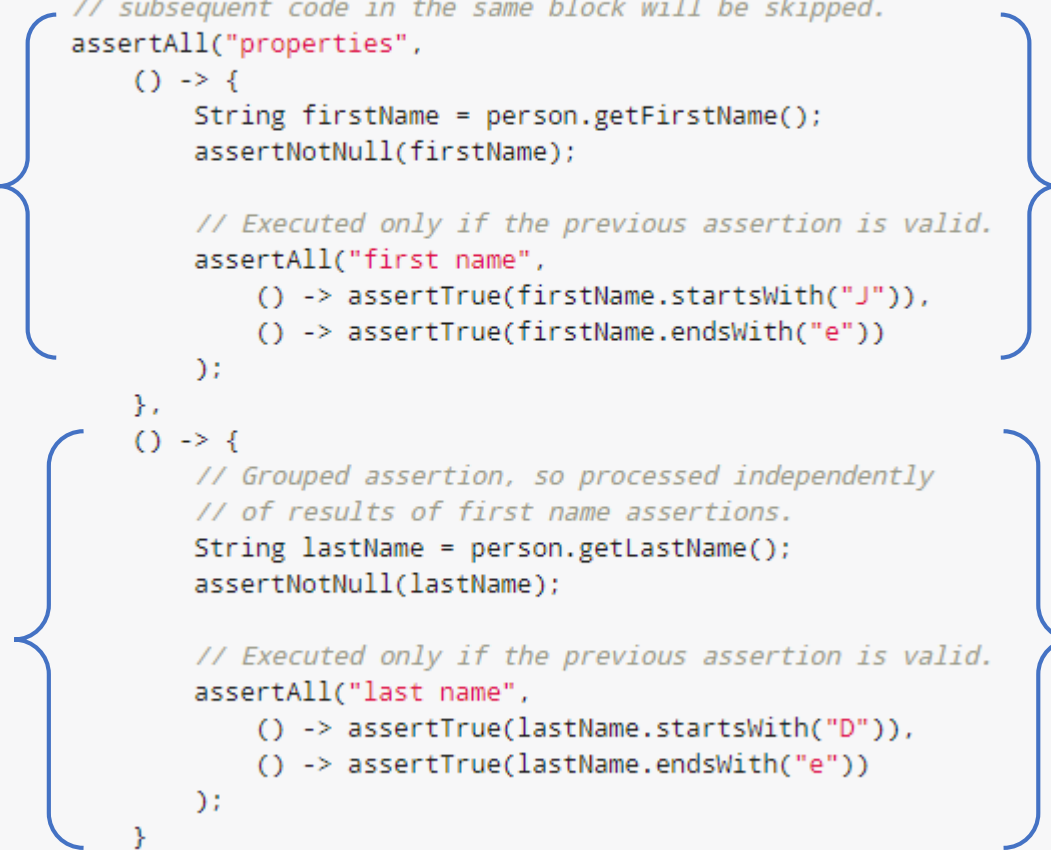
# Assertions

```
class AssertionsDemo {  
  
    private final Calculator calculator = new Calculator();  
  
    private final Person person = new Person("Jane", "Doe");  
  
    → @Test  
    void standardAssertions() {  
        → assertEquals(2, calculator.add(1, 1));  
        → assertEquals(4, calculator.multiply(2, 2),  
            "The optional failure message is now the last parameter");  
        → assertTrue('a' < 'b', () -> "Assertion messages can be lazily evaluated -- "  
            + "to avoid constructing complex messages unnecessarily.");  
    }  
}
```

# Assertions



```
@Test
void groupedAssertions() {
    // In a grouped assertion all assertions are executed, and all
    // failures will be reported together.
    assertAll("person",
        () -> assertEquals("Jane", person.getFirstName()),
        () -> assertEquals("Doe", person.getLastName())
    );
}
```

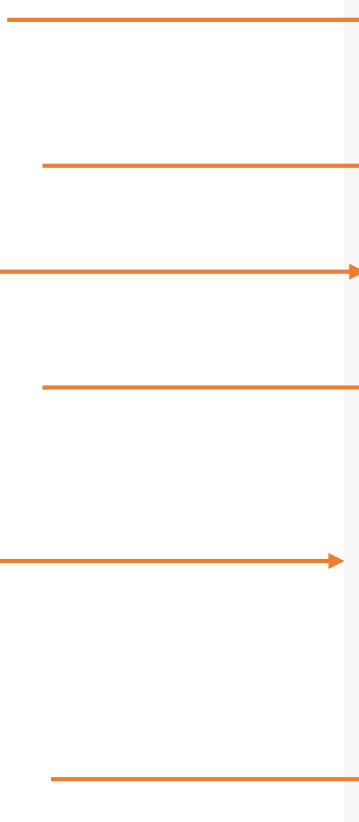


```
@Test
void dependentAssertions() {
    // Within a code block, if an assertion fails the
    // subsequent code in the same block will be skipped.
    assertAll("properties",
        () -> {
            String firstName = person.getFirstName();
            assertNotNull(firstName);

            // Executed only if the previous assertion is valid.
            assertAll("first name",
                () -> assertTrue(firstName.startsWith("J")),
                () -> assertTrue(firstName.endsWith("e"))
            );
        },
        () -> {
            // Grouped assertion, so processed independently
            // of results of first name assertions.
            String lastName = person.getLastName();
            assertNotNull(lastName);

            // Executed only if the previous assertion is valid.
            assertAll("last name",
                () -> assertTrue(lastName.startsWith("D")),
                () -> assertTrue(lastName.endsWith("e"))
            );
        }
    );
}
```

# Assertions

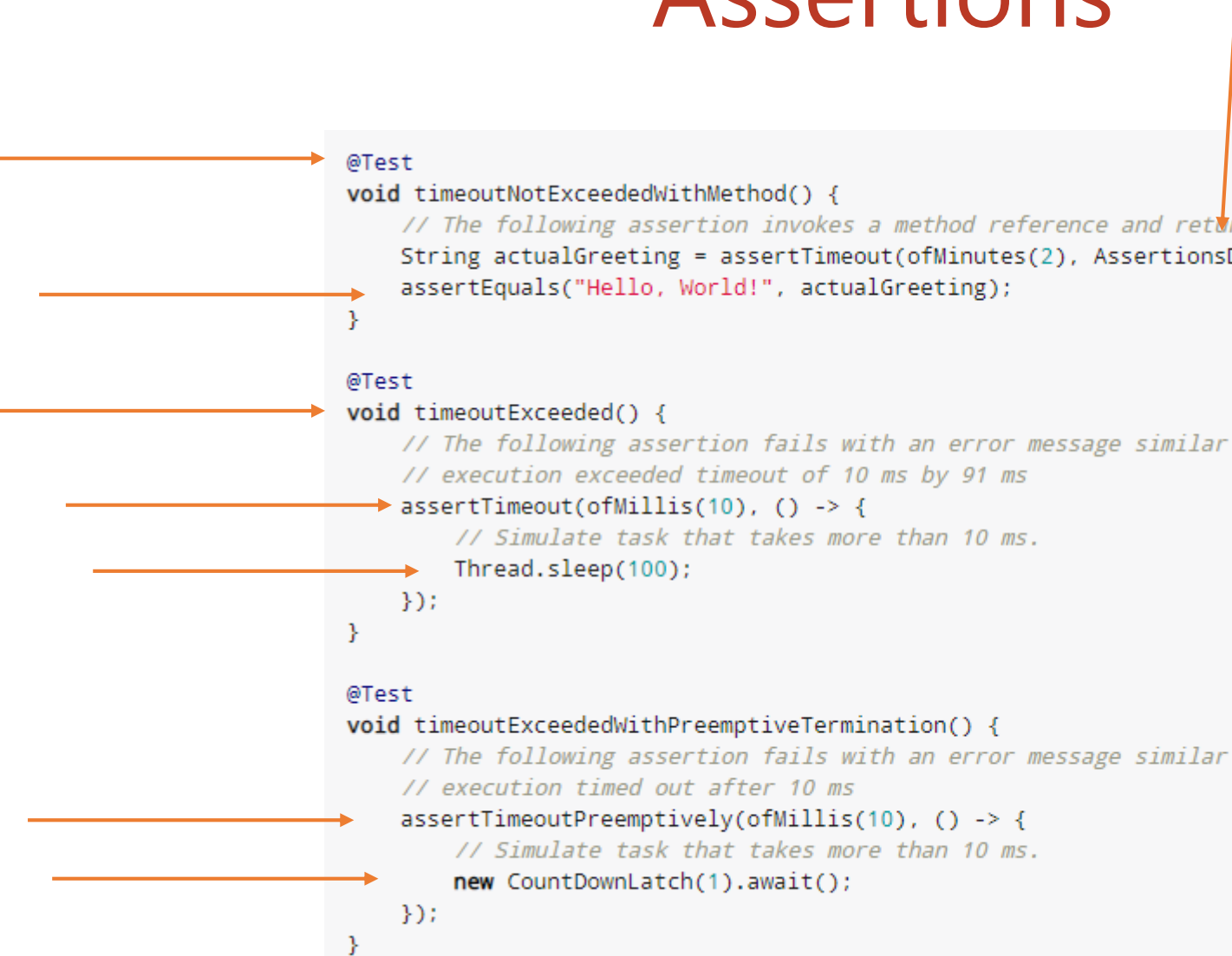


```
@Test
void exceptionTesting() {
    Exception exception = assertThrows(ArithmeticException.class, () ->
        calculator.divide(1, 0));
    assertEquals("/ by zero", exception.getMessage());
}

@Test
void timeoutNotExceeded() {
    // The following assertion succeeds.
    assertTimeout(ofMinutes(2), () -> {
        // Perform task that takes less than 2 minutes.
    });
}

@Test
void timeoutNotExceededWithResult() {
    // The following assertion succeeds, and returns the supplied object.
    String actualResult = assertTimeout(ofMinutes(2), () -> {
        return "a result";
    });
    assertEquals("a result", actualResult);
}
```

# Assertions



```
@Test
void timeoutNotExceededWithMethod() {
    // The following assertion invokes a method reference and returns an object.
    String actualGreeting = assertTimeout(ofMinutes(2), AssertionsDemo::greeting);
    assertEquals("Hello, World!", actualGreeting);
}

@Test
void timeoutExceeded() {
    // The following assertion fails with an error message similar to:
    // execution exceeded timeout of 10 ms by 91 ms
    assertTimeout(ofMillis(10), () -> {
        // Simulate task that takes more than 10 ms.
        Thread.sleep(100);
    });
}

@Test
void timeoutExceededWithPreemptiveTermination() {
    // The following assertion fails with an error message similar to:
    // execution timed out after 10 ms
    assertTimeoutPreemptively(ofMillis(10), () -> {
        // Simulate task that takes more than 10 ms.
        new CountDownLatch(1).await();
    });
}
```

# ENLACES DE REFERENCIAS

- <https://phpunit.readthedocs.io/es/latest/assertions.html#assertisint>
- <https://junit.org/junit5/docs/current/user-guide/#writing-tests-assertions-third-party>

# TAREA Asserts java

- Miércoles 20 de mayo
- Taller de la panadería (arreglar tarea 1)
- Asserts (pruebas, ejecución, análisis)( 20 asserts (diferentes))
- Código fuente a github
- Video expliquen eso

# EFICIENCIA ALGORITMICA

- Ciencias de la computación
- Propiedad que tiene un algoritmo para conocer la cantidad de recursos computacionales que usa.
- Recursos:
  - Tiempo (No es tiempo humano, no se mide en s, m,h etc)
  - Espacio (RAM)
- Complejidad

- En programación el rendimiento o complejidad o eficiencia de un algoritmo se suele medir con algo que se llama
- **BIG O, notación asintótica o notacion Landau** (Edmund landau)

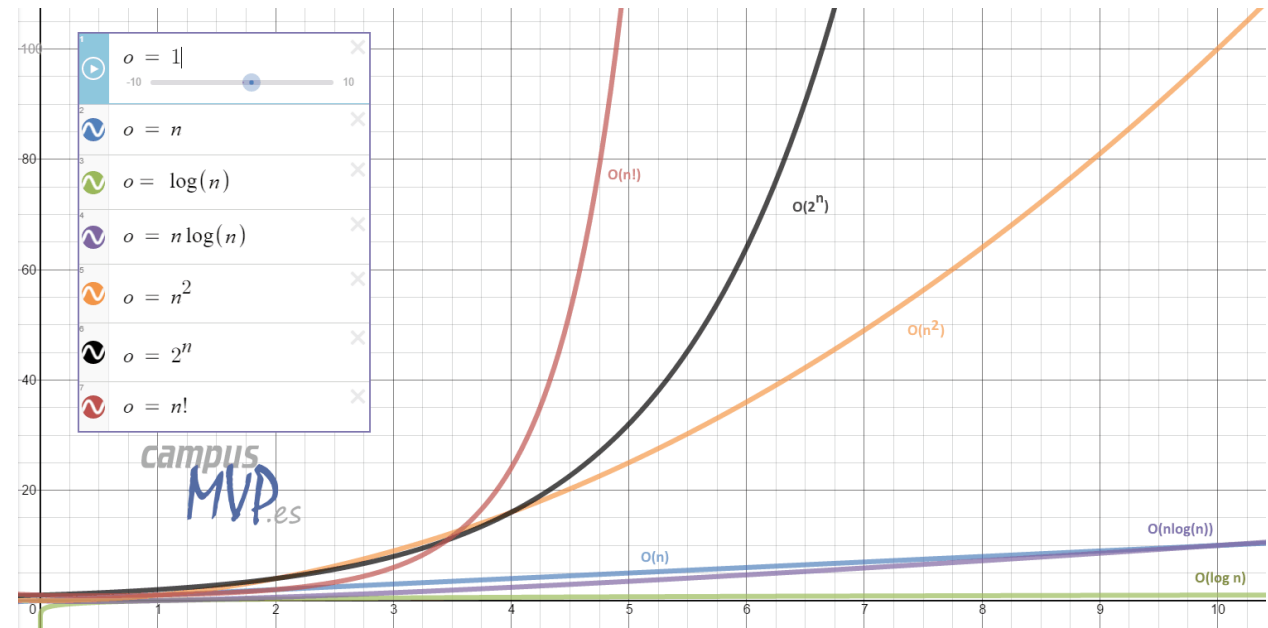


# ENTENDER NOTACION BIG O

- Big – O
- Es la manera de saber como se va a comportar un algoritmo en función de los argumentos que le pasemos y la escala de los mismos.
- **Caso**: localizar un elemento dentro de una lista (elementos previamente guardadas)
- Complejidad  $O(1)$  = el esfuerzo de computo necesario para encontrar un elemento

# Big O las notaciones mas comunes en los algoritmos son:

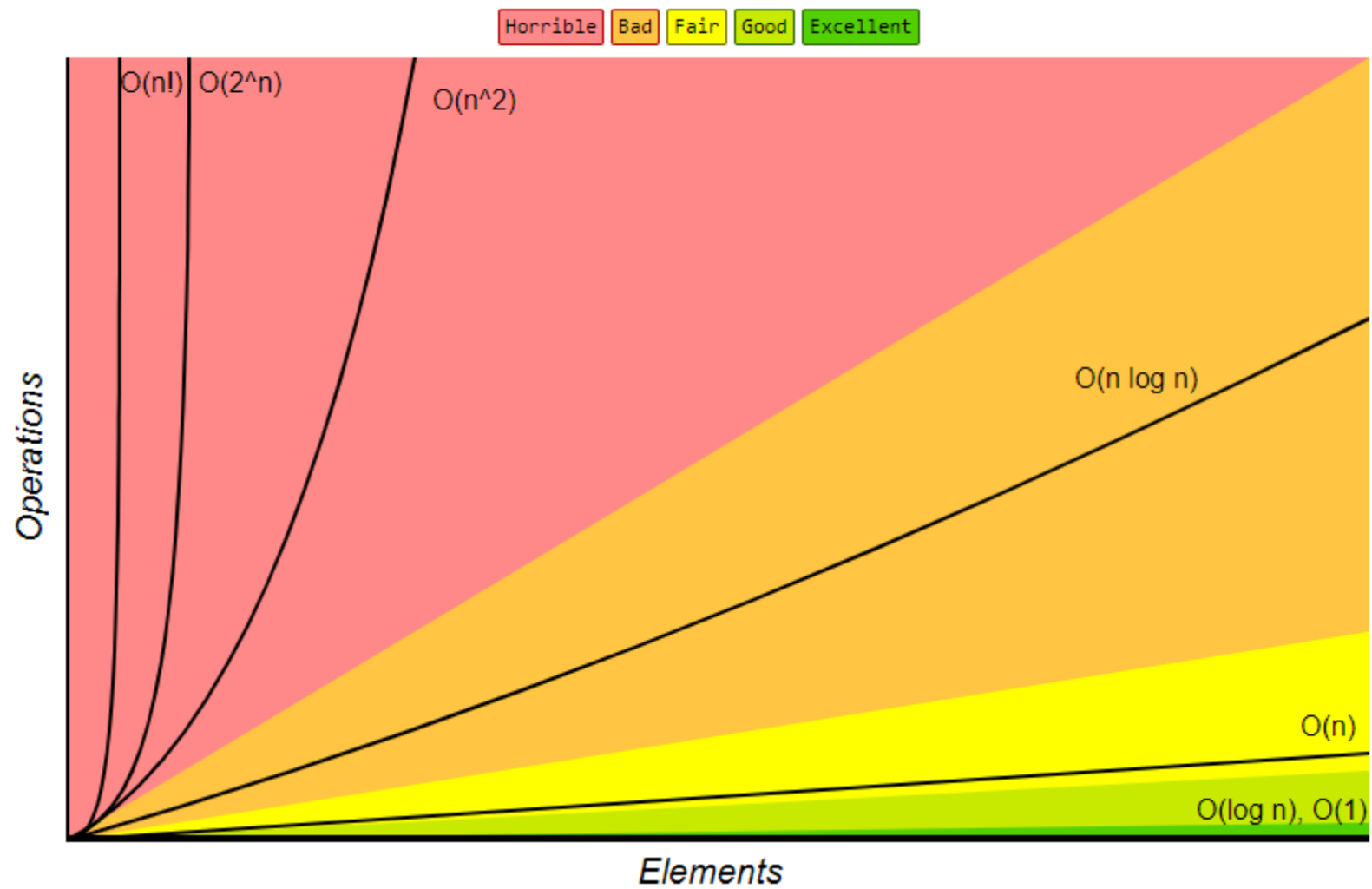
- **$O(1)$**  = constante
- **$O(n)$**  = lineal
- **$O(n \log n)$**  = logarítmico
- **$O(n^2)$**  = Cuadrático
- **$O(2^n)$** : Exponencial
- **$O(n!)$** ; explosión combinatoria



- X= numero elementos
- Y= tiempo

| Notación        | Nombre                  | Ejemplos  |
|-----------------|-------------------------|---|
| $O(1)$          | constante               | Determinar si un número es par o impar. Usar una <a href="#">tabla de consulta</a> que asocia constante/tamaño. Usar una <a href="#">función hash</a> para obtener un elemento.   |
| $O(\log n)$     | logarítmico             | Buscar un elemento específico en un array utilizando un <a href="#">árbol binario de búsqueda</a> o un <a href="#">árbol</a> de búsqueda balanceado, así como todas las operaciones en un <a href="#">Heap binomial</a> .               |
| $O(n)$          | lineal                  | Buscar un elemento específico en una lista desordenada o en un árbol degenerado (peor caso).  |
| $O(n \log n)$   | loglinear o quasilinear | Ejecutar una <a href="#">transformada rápida de Fourier</a> ; <a href="#">heapsort</a> , <a href="#">quicksort</a> (caso mejor y promedio), o <a href="#">merge sort</a>  |
| $O(n^2)$        | cuadrático              | Multiplicar dos números de $n$ dígitos por un algoritmo simple. <a href="#">bubble sort</a> (caso peor o implementación sencilla), <a href="#">Shell sort</a> , <a href="#">quicksort</a> (caso peor).                                  |
| $O(c^n), c > 1$ | exponencial             | Encontrar la solución exacta al <a href="#">problema del viajante</a> utilizando <a href="#">programación dinámica</a> . Determinar si dos sentencias lógicas son equivalentes utilizando una <a href="#">búsqueda por fuerza bruta</a> |

## Big-O Complexity Chart



## Common Data Structure Operations

| Data Structure            | Time Complexity   |                   |                   |                   |                   |                   |                   |                   | Space Complexity    |
|---------------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|---------------------|
|                           | Average           |                   |                   |                   | Worst             |                   |                   |                   | Worst               |
|                           | Access            | Search            | Insertion         | Deletion          | Access            | Search            | Insertion         | Deletion          |                     |
| <u>Array</u>              | $\Theta(1)$       | $\Theta(n)$       | $\Theta(n)$       | $\Theta(n)$       | $\Theta(1)$       | $\Theta(n)$       | $\Theta(n)$       | $\Theta(n)$       | $\Theta(n)$         |
| <u>Stack</u>              | $\Theta(n)$       | $\Theta(n)$       | $\Theta(1)$       | $\Theta(1)$       | $\Theta(n)$       | $\Theta(n)$       | $\Theta(1)$       | $\Theta(1)$       | $\Theta(n)$         |
| <u>Queue</u>              | $\Theta(n)$       | $\Theta(n)$       | $\Theta(1)$       | $\Theta(1)$       | $\Theta(n)$       | $\Theta(n)$       | $\Theta(1)$       | $\Theta(1)$       | $\Theta(n)$         |
| <u>Singly-Linked List</u> | $\Theta(n)$       | $\Theta(n)$       | $\Theta(1)$       | $\Theta(1)$       | $\Theta(n)$       | $\Theta(n)$       | $\Theta(1)$       | $\Theta(1)$       | $\Theta(n)$         |
| <u>Doubly-Linked List</u> | $\Theta(n)$       | $\Theta(n)$       | $\Theta(1)$       | $\Theta(1)$       | $\Theta(n)$       | $\Theta(n)$       | $\Theta(1)$       | $\Theta(1)$       | $\Theta(n)$         |
| <u>Skip List</u>          | $\Theta(\log(n))$ | $\Theta(\log(n))$ | $\Theta(\log(n))$ | $\Theta(\log(n))$ | $\Theta(n)$       | $\Theta(n)$       | $\Theta(n)$       | $\Theta(n)$       | $\Theta(n \log(n))$ |
| <u>Hash Table</u>         | N/A               | $\Theta(1)$       | $\Theta(1)$       | $\Theta(1)$       | N/A               | $\Theta(n)$       | $\Theta(n)$       | $\Theta(n)$       | $\Theta(n)$         |
| <u>Binary Search Tree</u> | $\Theta(\log(n))$ | $\Theta(\log(n))$ | $\Theta(\log(n))$ | $\Theta(\log(n))$ | $\Theta(n)$       | $\Theta(n)$       | $\Theta(n)$       | $\Theta(n)$       | $\Theta(n)$         |
| <u>Cartesian Tree</u>     | N/A               | $\Theta(\log(n))$ | $\Theta(\log(n))$ | $\Theta(\log(n))$ | N/A               | $\Theta(n)$       | $\Theta(n)$       | $\Theta(n)$       | $\Theta(n)$         |
| <u>B-Tree</u>             | $\Theta(\log(n))$ | $\Theta(\log(n))$ | $\Theta(\log(n))$ | $\Theta(\log(n))$ | $\Theta(\log(n))$ | $\Theta(\log(n))$ | $\Theta(\log(n))$ | $\Theta(\log(n))$ | $\Theta(n)$         |
| <u>Red-Black Tree</u>     | $\Theta(\log(n))$ | $\Theta(\log(n))$ | $\Theta(\log(n))$ | $\Theta(\log(n))$ | $\Theta(\log(n))$ | $\Theta(\log(n))$ | $\Theta(\log(n))$ | $\Theta(\log(n))$ | $\Theta(n)$         |
| <u>Splay Tree</u>         | N/A               | $\Theta(\log(n))$ | $\Theta(\log(n))$ | $\Theta(\log(n))$ | N/A               | $\Theta(\log(n))$ | $\Theta(\log(n))$ | $\Theta(\log(n))$ | $\Theta(n)$         |
| <u>AVL Tree</u>           | $\Theta(\log(n))$ | $\Theta(\log(n))$ | $\Theta(\log(n))$ | $\Theta(\log(n))$ | $\Theta(\log(n))$ | $\Theta(\log(n))$ | $\Theta(\log(n))$ | $\Theta(\log(n))$ | $\Theta(n)$         |
| <u>KD Tree</u>            | $\Theta(\log(n))$ | $\Theta(\log(n))$ | $\Theta(\log(n))$ | $\Theta(\log(n))$ | $\Theta(n)$       | $\Theta(n)$       | $\Theta(n)$       | $\Theta(n)$       | $\Theta(n)$         |

## Array Sorting Algorithms

| Algorithm             | Time Complexity     |                        |                        | Space Complexity  |
|-----------------------|---------------------|------------------------|------------------------|-------------------|
|                       | Best                | Average                | Worst                  | Worst             |
| <u>Quicksort</u>      | $\Omega(n \log(n))$ | $\Theta(n \log(n))$    | $\Theta(n^2)$          | $\Theta(\log(n))$ |
| <u>Mergesort</u>      | $\Omega(n \log(n))$ | $\Theta(n \log(n))$    | $\Theta(n \log(n))$    | $\Theta(n)$       |
| <u>Timsort</u>        | $\Omega(n)$         | $\Theta(n \log(n))$    | $\Theta(n \log(n))$    | $\Theta(n)$       |
| <u>Heapsort</u>       | $\Omega(n \log(n))$ | $\Theta(n \log(n))$    | $\Theta(n \log(n))$    | $\Theta(1)$       |
| <u>Bubble Sort</u>    | $\Omega(n)$         | $\Theta(n^2)$          | $\Theta(n^2)$          | $\Theta(1)$       |
| <u>Insertion Sort</u> | $\Omega(n)$         | $\Theta(n^2)$          | $\Theta(n^2)$          | $\Theta(1)$       |
| <u>Selection Sort</u> | $\Omega(n^2)$       | $\Theta(n^2)$          | $\Theta(n^2)$          | $\Theta(1)$       |
| <u>Tree Sort</u>      | $\Omega(n \log(n))$ | $\Theta(n \log(n))$    | $\Theta(n^2)$          | $\Theta(n)$       |
| <u>Shell Sort</u>     | $\Omega(n \log(n))$ | $\Theta(n(\log(n))^2)$ | $\Theta(n(\log(n))^2)$ | $\Theta(1)$       |
| <u>Bucket Sort</u>    | $\Omega(n+k)$       | $\Theta(n+k)$          | $\Theta(n^2)$          | $\Theta(n)$       |
| <u>Radix Sort</u>     | $\Omega(nk)$        | $\Theta(nk)$           | $\Theta(nk)$           | $\Theta(n+k)$     |
| <u>Counting Sort</u>  | $\Omega(n+k)$       | $\Theta(n+k)$          | $\Theta(n+k)$          | $\Theta(k)$       |
| <u>Cubesort</u>       | $\Omega(n)$         | $\Theta(n \log(n))$    | $\Theta(n \log(n))$    | $\Theta(n)$       |

# LEGEND

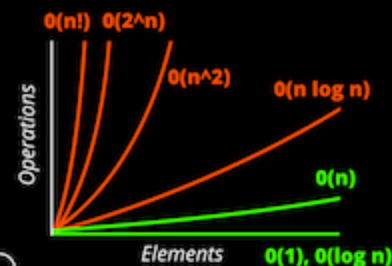
TIME Complexity vs. SPACE Complexity

Good Fair Bad

Good Fair Bad

## <BIG-O-CHEATSHEET>

www.bigocheatsheet.com



### DATA STRUCTURE Operations

### ARRAY SORTING Algorithms

#### DATA Structure

#### TIME Complexity

#### SPACE Complexity

Average

Worst

Worst

Access Search Insertion Deletion

Access Search Insertion Deletion

| DATA Structure     | Access            | Search            | Insertion         | Deletion          | TIME Complexity (Average) | TIME Complexity (Worst) | SPACE Complexity (Worst) |
|--------------------|-------------------|-------------------|-------------------|-------------------|---------------------------|-------------------------|--------------------------|
| Array              | $\Theta(1)$       | $\Theta(n)$       | $\Theta(n)$       | $\Theta(n)$       | $\Theta(1)$               | $\Theta(n)$             | $\Theta(n)$              |
| Stack              | $\Theta(n)$       | $\Theta(n)$       | $\Theta(1)$       | $\Theta(1)$       | $\Theta(n)$               | $\Theta(n)$             | $\Theta(1)$              |
| Queue              | $\Theta(n)$       | $\Theta(n)$       | $\Theta(1)$       | $\Theta(1)$       | $\Theta(n)$               | $\Theta(n)$             | $\Theta(1)$              |
| Singly-Linked List | $\Theta(n)$       | $\Theta(n)$       | $\Theta(1)$       | $\Theta(1)$       | $\Theta(n)$               | $\Theta(n)$             | $\Theta(1)$              |
| Doubly-Linked List | $\Theta(n)$       | $\Theta(n)$       | $\Theta(1)$       | $\Theta(1)$       | $\Theta(n)$               | $\Theta(n)$             | $\Theta(1)$              |
| Skip List          | $\Theta(\log(n))$ | $\Theta(\log(n))$ | $\Theta(\log(n))$ | $\Theta(\log(n))$ | $\Theta(n)$               | $\Theta(n)$             | $\Theta(n \log(n))$      |
| Hash Table         | N/A               | $\Theta(1)$       | $\Theta(1)$       | $\Theta(1)$       | N/A                       | $\Theta(n)$             | $\Theta(n)$              |
| Binary Search Tree | $\Theta(\log(n))$ | $\Theta(\log(n))$ | $\Theta(\log(n))$ | $\Theta(\log(n))$ | $\Theta(n)$               | $\Theta(n)$             | $\Theta(n)$              |
| Cartesian Tree     | N/A               | $\Theta(\log(n))$ | $\Theta(\log(n))$ | $\Theta(\log(n))$ | N/A                       | $\Theta(n)$             | $\Theta(n)$              |
| B-Tree             | $\Theta(\log(n))$ | $\Theta(\log(n))$ | $\Theta(\log(n))$ | $\Theta(\log(n))$ | $\Theta(\log(n))$         | $\Theta(\log(n))$       | $\Theta(n)$              |
| Red-Black Tree     | $\Theta(\log(n))$ | $\Theta(\log(n))$ | $\Theta(\log(n))$ | $\Theta(\log(n))$ | $\Theta(\log(n))$         | $\Theta(\log(n))$       | $\Theta(n)$              |
| Splay Tree         | N/A               | $\Theta(\log(n))$ | $\Theta(\log(n))$ | $\Theta(\log(n))$ | N/A                       | $\Theta(\log(n))$       | $\Theta(n)$              |
| AVL Tree           | $\Theta(\log(n))$ | $\Theta(\log(n))$ | $\Theta(\log(n))$ | $\Theta(\log(n))$ | $\Theta(\log(n))$         | $\Theta(\log(n))$       | $\Theta(n)$              |
| KD Tree            | $\Theta(\log(n))$ | $\Theta(\log(n))$ | $\Theta(\log(n))$ | $\Theta(\log(n))$ | $\Theta(n)$               | $\Theta(n)$             | $\Theta(n)$              |

#### ARRAY Algorithms

#### TIME Complexity

#### SPACE Complexity

Best

Average

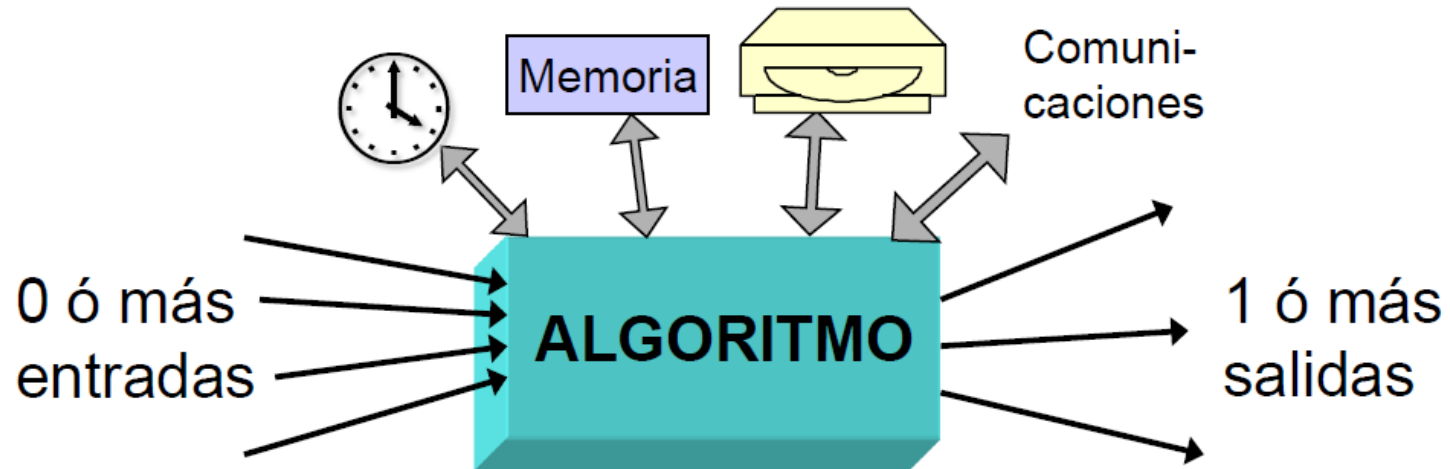
Worst

Worst

| ARRAY Algorithms | TIME Complexity (Best) | TIME Complexity (Average) | TIME Complexity (Worst) | SPACE Complexity (Worst) |
|------------------|------------------------|---------------------------|-------------------------|--------------------------|
| Quicksort        | $\Omega(n \log(n))$    | $\Theta(n \log(n))$       | $\Theta(n^2)$           | $\Theta(\log(n))$        |
| Mergesort        | $\Omega(n \log(n))$    | $\Theta(n \log(n))$       | $\Theta(n \log(n))$     | $\Theta(n)$              |
| Timsort          | $\Omega(n)$            | $\Theta(n \log(n))$       | $\Theta(n \log(n))$     | $\Theta(1)$              |
| Heapsort         | $\Omega(n \log(n))$    | $\Theta(n \log(n))$       | $\Theta(n \log(n))$     | $\Theta(1)$              |
| Bubble Sort      | $\Omega(n)$            | $\Theta(n^2)$             | $\Theta(n^2)$           | $\Theta(1)$              |
| Insertion Sort   | $\Omega(n)$            | $\Theta(n^2)$             | $\Theta(n^2)$           | $\Theta(1)$              |
| Selection Sort   | $\Omega(n^2)$          | $\Theta(n^2)$             | $\Theta(n^2)$           | $\Theta(1)$              |
| Tree Sort        | $\Omega(n \log(n))$    | $\Theta(n \log(n))$       | $\Theta(n^2)$           | $\Theta(n)$              |
| Shell Sort       | $\Omega(n \log(n))$    | $\Theta(n(\log(n))^2)$    | $\Theta(n(\log(n))^2)$  | $\Theta(1)$              |
| Bucket Sort      | $\Omega(n+k)$          | $\Theta(n+k)$             | $\Theta(n^2)$           | $\Theta(n)$              |
| Radix Sort       | $\Omega(nk)$           | $\Theta(nk)$              | $\Theta(nk)$            | $\Theta(n+k)$            |
| Counting Sort    | $\Omega(n+k)$          | $\Theta(n+k)$             | $\Theta(n+k)$           | $\Theta(k)$              |
| Cubesort         | $\Omega(n)$            | $\Theta(n \log(n))$       | $\Theta(n \log(n))$     | $\Theta(n)$              |

# ANALISIS DE ALGORITMOS

**Algoritmo:** conjunto de reglas para resolver un problema. Su ejecución requiere unos recursos.



Un algoritmo es mejor cuantos menos recursos consume. Pero....

**Otros criterios:** facilidad de programarlo, corto, fácil de entender, robusto, reutilizable...

**Criterio empresarial:** maximizar la eficiencia.

**Eficiencia:** relación entre los recursos consumidos y los productos conseguidos.

**Recursos consumidos:**

- Tiempo de ejecución.
- Memoria principal.
- Entradas/salidas a disco.
- Comunicaciones, procesadores,...

**Lo que se consigue:**

- Resolver un problema de forma exacta.
- Resolverlo de forma aproximada.
- Resolver algunos casos...



# ALGORITMO

- 1. ENTRADA Y LA SALIDA
- 2. PRECISA (RECETA PIZZA)
- 3. deterministas (fin)
- 4. Finito (Determinado numero de pasos computacion) (código semilla)
- 5, Efectivo (tener el resultado esperado) OUTPUT - INPUT

# TIEMPO

El tiempo de un algoritmo depende de los datos de entrada, de la implementación del programa, del procesador y la complejidad del algoritmo.

Dimate tiene algoritmo y quiere calcular tiempo

El tiempo que requier un algoritmo en resolver un problema , esta dado en función del tamaño  $N$  del conjunto de datos de entrada que voy a procesar  $T(n)$

Def function(int valor);

Valor= 1

Valor = 9999999999

# TIEMPO

- ESTUDIO A PRIORI ( $t(N)$ ) medida teórico N= datos de entrada ()  
Matematicamente
- ESTUDI A POSTERI (DESPUES DE ) , MEDIDA REAL ( $3s$ )

El tiempo de ejecución  $T(n)$  de un algoritmo para una entrada de tamaño  $n$  no debe medirse en segundos, milisegundos, etc.,

$T(n)$  debe ser independiente del tipo de computadora en el que corre

# LA EFICIENCIA DE UN ALGORITMO

## *Principio de invarianza*

Dado un algoritmo y dos implementaciones suyas  $I1$  e  $I2$ , que tardan  $T1(n)$  y  $T2(n)$  respectivamente, el *Principio de invarianza* afirma que existe una constante real  $c > 0$  y un número natural  $n0$  tales que para todo  $n \geq n0$  se verifica que:

$$T_1(n) \leq cT_2(n).$$

• *Principio de invarianza*

El tiempo de ejecución de dos implementaciones distintas de un algoritmo dado no va a diferir más que en una constante multiplicativa [Guerequeta y Vallecillo, 2000].

Esto significa que el tiempo para resolver un problema mediante un algoritmo depende de la naturaleza del algoritmo y no de la implementación del algoritmo.:

# El tiempo $T(N)$

- OE (OPERACIONES ELEMENTALES)
  - OPERACIÓN ARITMETICA
  - ASIGNACION A UNA VARIABLE
  - LLAMADA A UNA FUNCION
  - RETORNO DE UNA FUNCION
  - COMPARACION LOGICA (CON SALTO)
  - ACCESO A UNA ESTRUCTURA DE DATOS (STACK)
  - OPERACIONES, CRUD, COLLECCTION
- No
  - DOCUMENTACION

# EJEMPLO 1

Análisis de operaciones elementales en un algoritmo que busca un número dado dentro de un arreglo ordenado

- Veamos cuantas operaciones elementales contiene el algoritmo del ejemplo



# Calcular $t(N)$ Datos de entrada (mismo tipo)

- Caso promedio  $T(n) = 16 \text{ OE}$  , cuando  $n=9$  /  $T(9) = 16 \text{ OE}$
- Mejor caso  $T(n) = T(n) = n = 9$  /  $T(9) = 11$
- Peor caso

```
#include <cstdlib>
#include <iostream>
```

¿Cuántas operaciones elementales se realizan en este algoritmo?

```
using namespace std;
```

```
// L
int main(int argc, char *argv[])
```

```
{
```

```
int ArregloOrdenado[9]={1,3,7,15,19,24,31,38,40};
```

1 OE (asignación)

```
int buscado, j=0;
```

1 OE (asignación)

```
cout<<"¿Que numero entero quieres buscar en el arreglo?";
```

1 OE (salida)

```
cin>>buscado;
```

1 OE (entrada)

```
while (ArregloOrdenado[j] < buscado && j < 9)
```

4 OE (1 acceso + 2 comparaciones + 1 AND)

```
j = j+1;
```

2 OE (incremento + asignación)

```
if ( ArregloOrdenado[j] == buscado)
```

2 OE (acceso + comparación)

```
cout<<"tu entero si esta en el arreglo";
```

1 OE (salida)

```
else
```

```
cout<<"lastima! no esta";
```

1 OE (salida)

```
system("PAUSE");
```

1 OE (pausa)

```
return EXIT_SUCCESS;
```

1 OE (regreso)

```
}
```

# Mejor de los casos

- El elemento a buscar este en la primera posición del arreglo

```
#include <cstdlib>
#include <iostream>
```

EL MEJOR CASO:  $T(9) = 4 + 2 + 2 + 3 = 11$

```
using namespace std;
```

```
int main(int argc, char *argv[])
{
```

```
    int ArregloOrdenado[9]={1,3,7,15,19,24,31,38,40};
```

1 OE (asignación)

```
    int buscado, j=0;
```

1 OE (asignación)

```
    cout<<"¿Que numero entero quieres buscar en el arreglo?";
```

1 OE (salida)

```
    cin>>buscado;
```

1 OE (entrada)

(buscado = 1)

```
    while(ArregloOrdenado[j] < buscado && j<9 )
```

2 OE (1 acceso + 1 comparación)

```
        j = j+1;
```

```
    if ( ArregloOrdenado[j] == buscado)
```

2 OE (acceso + comparación)

```
        cout<<"tu entero si esta en el arreglo";
```

1 OE (salida)

```
    else
```

```
        cout<<" lastima! no esta";
```

Total: 11 OE

```
    system("PAUSE");
```

1 OE (pausa)

```
    return EXIT_SUCCESS;
```

1 OE (regreso)

```
}
```

# PEOR CASO

Análisis de operaciones elementales en un algoritmo que busca un número dado dentro de un arreglo ordenado (peor caso)

- analizamos el algoritmo para el peor caso analizaremos, es decir, cuando se recorre todo el arreglo y no se encuentra al elemento buscado

```

#include <cstdlib>
#include <iostream>

using namespace std;

int main(int argc, char *argv[])
{
    int ArregloOrdenado[9]={1,3,7,15,19,24,31,38,40};
    int buscado, j=0;

    cout<<"¿Que numero entero quieres buscar en el arreglo?";
    cin>>buscado;

    while(ArregloOrdenado[j] < buscado && j<9 )
        j = j+1;

    if ( ArregloOrdenado[j] == buscado)
        cout<<"tu entero si esta en el arreglo";

    else
        cout<<" lastima! no esta";
    system("PAUSE");

    return EXIT_SUCCESS;
}

```

¿Cuántas operaciones elementales se realizan en este algoritmo?

1 OE (asignación)

1 OE (asignación)

1 OE (salida)

1 OE (entrada)

4 OE (1 acceso + 2 comparaciones+ 1 AND)

2 OE ( incremento + asignación)

2 OE (acceso + comparación)

1 OE (salida)

1 OE (salida)

1 OE (pausa)

1 OE (regreso)

**O (1)** = contante

**O (n)** = lineal

**O (n log n)** = logarítmico

**O(n<sup>2</sup>)** = Cuadrático

**O(2<sup>n</sup>)**: Exponencial

**O(n!)**; explosión combinatoria

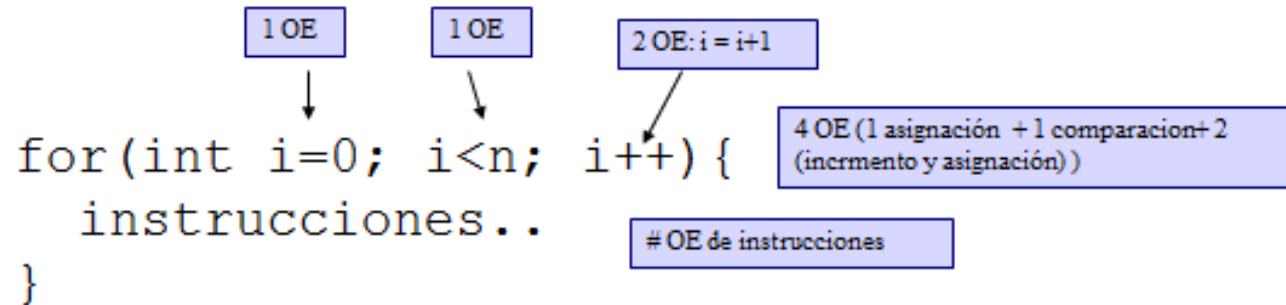
Por definición Va a ser de Orden 1

O(1)

$$T(n) = 4 + \sum_{j=0}^{n-1} (4 + 2) + 4 + 2 + 3 = 4 + 6n + 9$$

Para  $n = 9$   $T(n)$  es 67.

# Para calcular el tiempo de ejecución del ciclo for



$$1 + 1 + \sum_{i=0}^{n-1} (\text{\#OE de instrucciones} + 2 + 1)$$

Forma equivalente del for:

```
int i=0      1 OE
while( i<n ) { 1 OE
    instrucciones.. # OE de instrucciones
    i=i+1;      2 OE: i = i+1
}
```

# ANALISIS DE COMPLEJIDAD DE UN ALGORITMO

- Cotas de complejidad (medidas asintoticas)
- Clasificar los algoritmos y compararlos



# COTAS DE COMPLEJIDAD O MEDIDAS ASINTOTICAS MAS COMUNES

La notación  $O$  (o  
mayúscula),

La notación  $\Omega$   
(omega  
mayúscula)

La notación  $\Theta$   
(theta  
mayúscula)

todas se basan  
en el peor caso

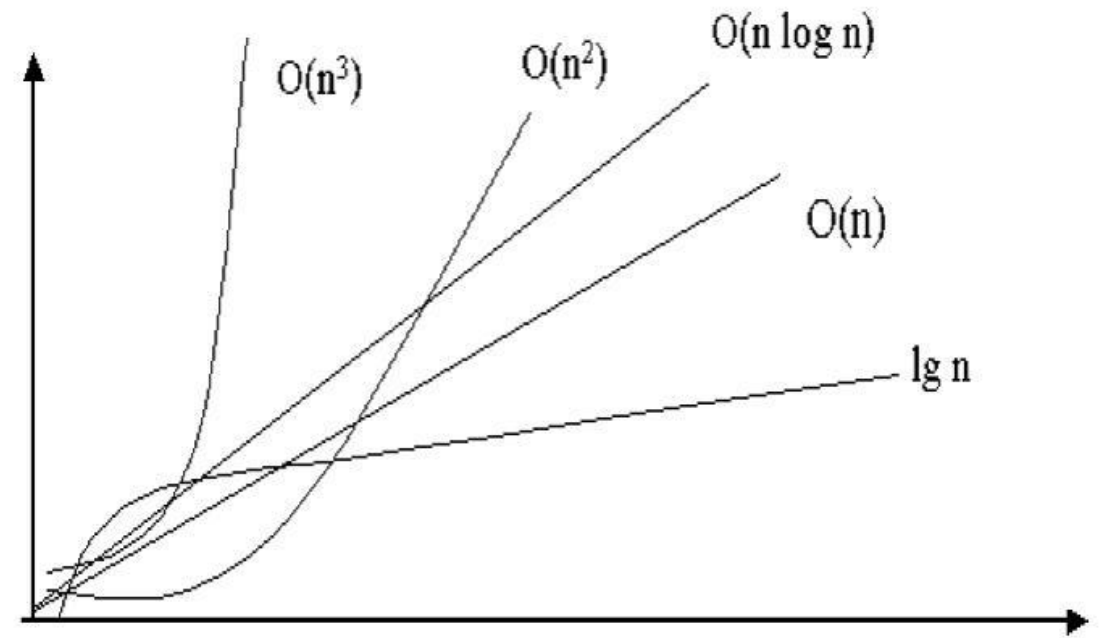
Cota superior asintótica

**Notación  $O$  (o mayúscula)**

- $O(1)$**       **Complejidad constante.-** Es la más deseada. Por ejemplo, es la complejidad que se presenta en secuencias de instrucciones sin repeticiones o ciclos (ver ejemplo 1 en la sección II.4.1).
- $O(\log n)$**       **Complejidad logarítmica.-** Puede presentarse en algoritmos iterativos y recursivos (ver ejemplo 2 en la sección II.8).
- $O(n)$**       **Complejidad lineal.-** Es buena y bastante usual. Suele aparecer en un *ciclo* principal simple cuando la complejidad de las operaciones interiores es constante (ver ejemplo 2 en la sección II.4.1).
- $O(n \log n)$**       **Complejidad  $n \cdot \log n$ .-** Se considera buena. Aparece en algunos algoritmos de ordenamiento (ver ejemplo 7 en la sección II.4.1).
- $O(n^2)$**       **Complejidad cuadrática.-** Aparece en ciclos anidados, por ejemplo *ordenación por burbuja*. También en algunas recursiones dobles (ver ejemplo 3 en la sección II.4.1).
- $O(n^3)$**       **Complejidad cúbica.-** En ciclos o en algunas recursiones triples. El algoritmo se vuelve lento para valores grandes de  $n$ .
- $O(n^k)$**       **Complejidad polinomial.-** Para ( $k \in \mathbb{N}$ ,  $k > 3$ ) mientras más crece  $k$ , el algoritmo se vuelve más lento (ver ejemplo 4 en la sección II.4.1).
- $O(C^n)$**       **Complejidad exponencial.-** Cuando  $n$  es grande, la solución de estos algoritmos es prohibitivamente costosa. Por ejemplo, problemas de explosión combinatoria.

# Funciones de complejidad más usuales

- $O(1)$ . Complejidad constante.
- $O(\log n)$ . Complejidad logarítmica.
- $O(n)$ . Complejidad lineal.
- $O(n \log n)$ .
- $O(n^2)$ . Complejidad cuadrática.
- $O(n^3)$ . Complejidad cúbica.
- $O(n^k)$ . Complejidad polinomial.
- $O(2^n)$ . Complejidad exponencial.



Algunas funciones ordenadas de menor a mayor complejidad.

|            | Caso Promedio | Peor Caso       | Metodo      | Ventaja y Desventaja   |
|------------|---------------|-----------------|-------------|--|
| Burbuja    | $O(n^2)$      | $O(n^2)$        | Intercambio | 1. Sencillo, simple y lento<br>2. Estable<br>3. Ineficiente en tablas largas   |
| Inserccion | $O(n^2)$      | $O(n^2)$        | Insercción  | 1. Eficiente para listas pequeñas y ordenadas<br>2. Simple y buen rendimiento<br>3. se guarda en la memoria                    |
| Heap Sort  | $O(n \log n)$ | $O(n \log n)$   | Selección   | 1. Mas eficiente que el de seleccion<br>2. No es necesario espacio adicional<br>3. Es mejor que Quick sort en los peores casos |
| Quick Sort | $O(n \log n)$ | $O(n^2)$        | Partición   | 1. Es rapido y eficiente<br>2. No requiere memoria adicional<br>3. consume mucho tiempo  |
| Shell Sort | $O(n \log n)$ | $O(n \log^2 n)$ | Insercción  | 1. la complejidad depende del orden de espaciado<br>2. Poco estable, ya que pierde el orden relativo<br>3. No utiliza pivote   |

## Unidades de Medidas de Almacenamiento

| Medida     | Simbologia | Equivalencia | Equivalente en Bytes                            |
|------------|------------|--------------|---|
| byte       | b          | 8 bits       | 1 byte  |
| kilobyte   | Kb         | 1024 bytes   | 1 024 bytes                                     |
| megabyte   | MB         | 1024 KB      | 1 048 576 bytes                                 |
| gigabyte   | GB         | 1024 MB      | 1 073 741 824 bytes                             |
| terabyte   | TB         | 1024 GB      | 1 099 511 627 776 bytes                         |
| Petabyte   | PB         | 1024 TB      | 1 125 899 906 842 624 bytes                     |
| Exabyte    | EB         | 1024 PB      | 1 152 921 504 606 846 976 bytes                 |
| Zetabyte   | ZB         | 1024 EB      | 1 180 591 620 717 411 303 424 bytes             |
| Yottabyte  | YB         | 1024 ZB      | 1 208 925 819 614 629 174 706 176 bytes         |
| Brontobyte | BB         | 1024 YB      | 1 237 940 039 285 380 274 899 124 224 bytes     |
| Geopbyte   | GB         | 1024 BB      | 1 267 650 600 228 229 401 496 703 205 376 bytes |

[www.tiposdecomputadora.wordpress.com](http://www.tiposdecomputadora.wordpress.com)

# TALLER EN CLASE

- HACER LOS SIGUIENTES ALGORIMOS Y CALCULAR

- Operaciones elementales.
- Orden del algoritmo

1. Hacer un algoritmo que calcule dados 3 números, cual es primo, cual es par y cual es impar.
2. Algoritmo que transforme de grados °C a °F (números enteros  $\leq 3$  digitos)
3. Algoritmo que calcule de una frase de no mas de 10 palabras, cuantas vocales y cuantas consonantes existen.
4. Algoritmo que calcule la suma de dos matrices de  $3 \times 3$
5. Algoritmo que ordene 5 números flotantes dados de mayor a menor

# REFERENCIAS IMPORTANTES

- <http://www.cs.us.es/~jalonso/cursos/i1m/temas/tema-28.html>

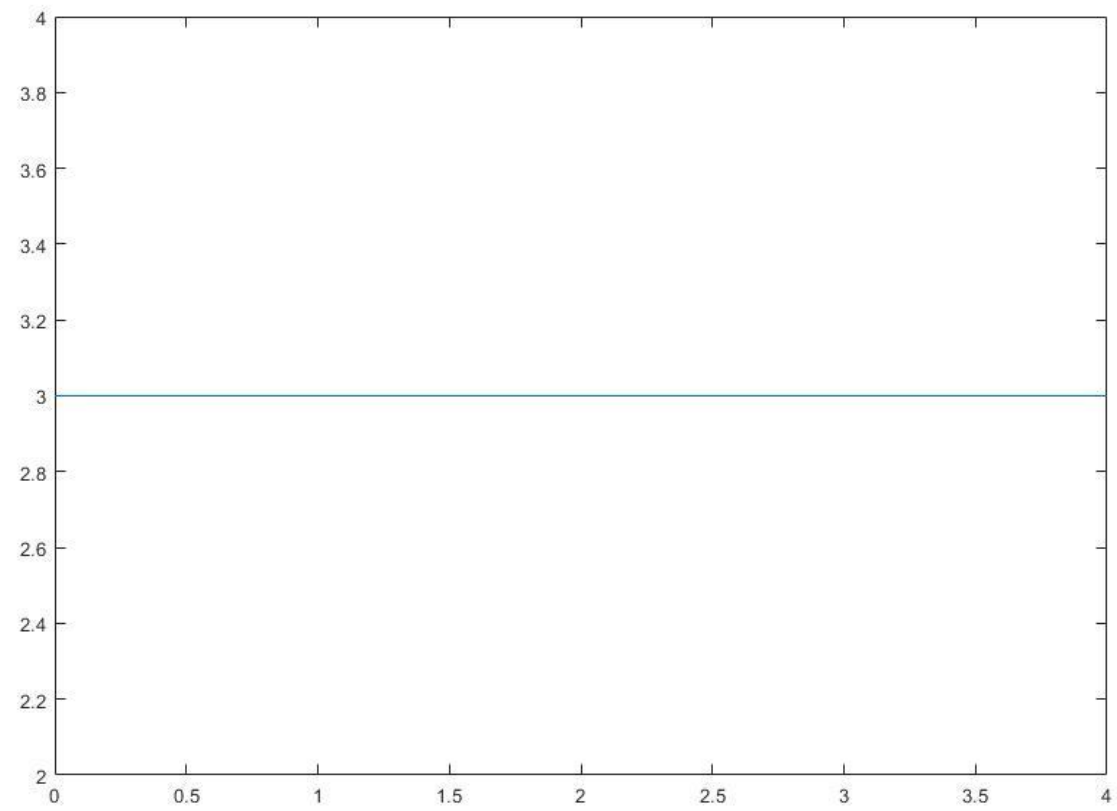


# COMPLEJIDAD ALGORITMICA - PYTHON

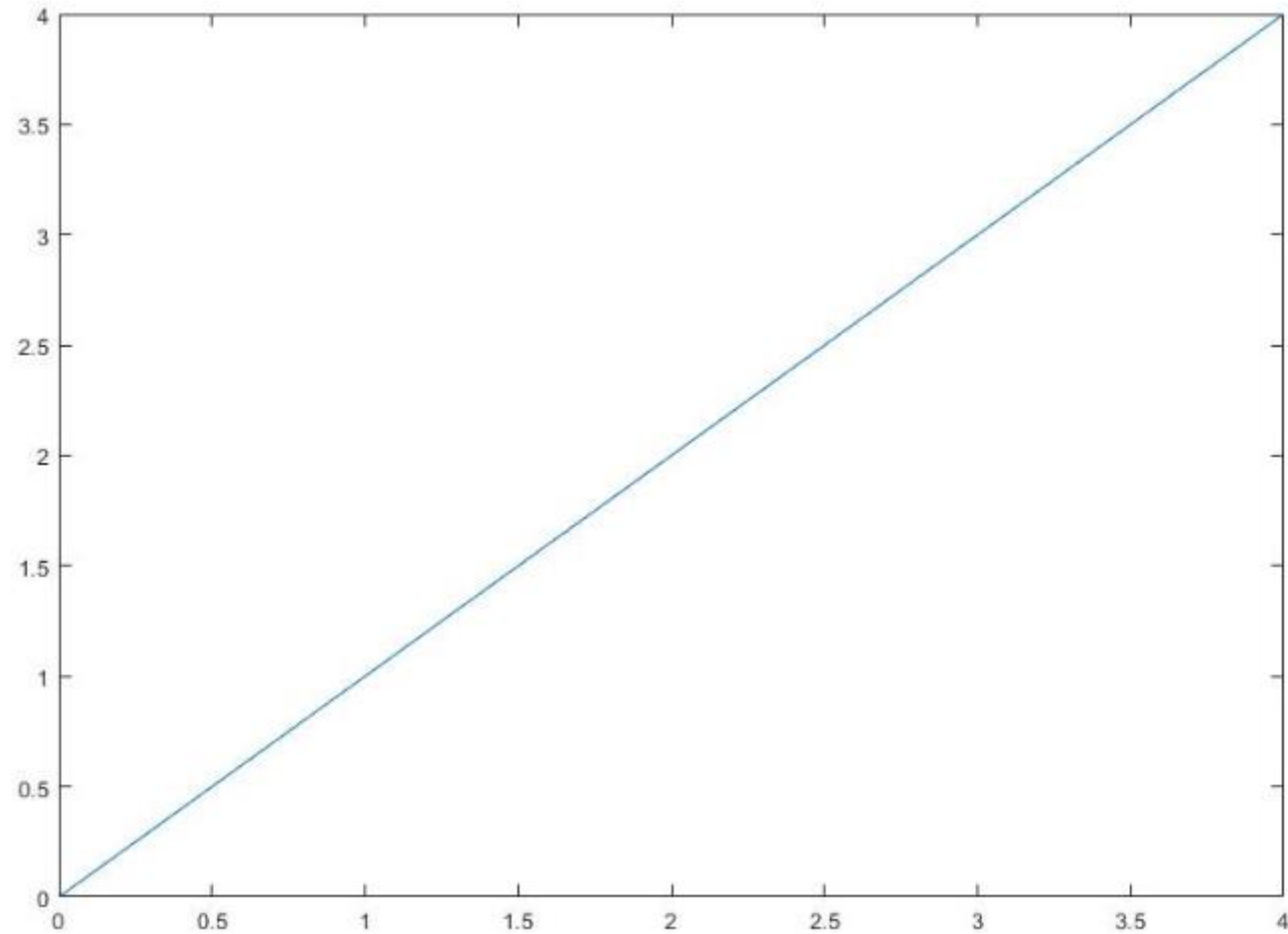
- Complejidad temporal y complejidad espacial
- Tamaño si importa (N)
  - Ordenar un arreglo 10 posiciones / 1000000 posiciones
- El resultado de la complejidad es una función
- Caso promedio
- Mejor caso
- Peor de los casos (O GRANDE, BIG O)
  - $O(n)$
  - Omega, ómicron (griega)

| $f(n)$     | Nombre      |
|------------|-------------|
| 1          | Constante   |
| $\log n$   | Logarítmica |
| $n$        | Lineal      |
| $n \log n$ | Log-lineal  |
| $n^2$      | Cuadrática  |
| $n^3$      | Cúbica      |
| $2^n$      | Exponencial |

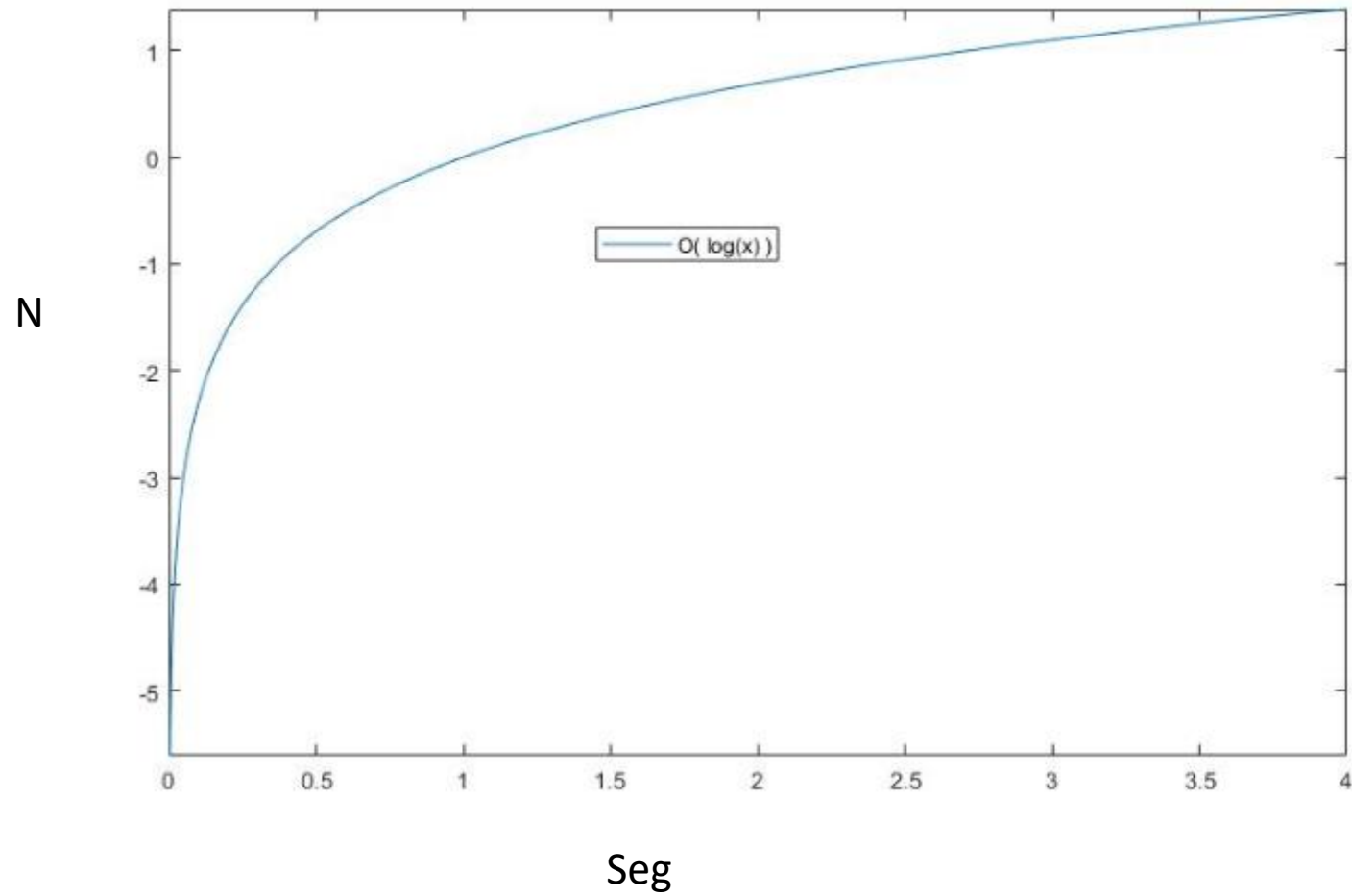
$O(1)$       constante



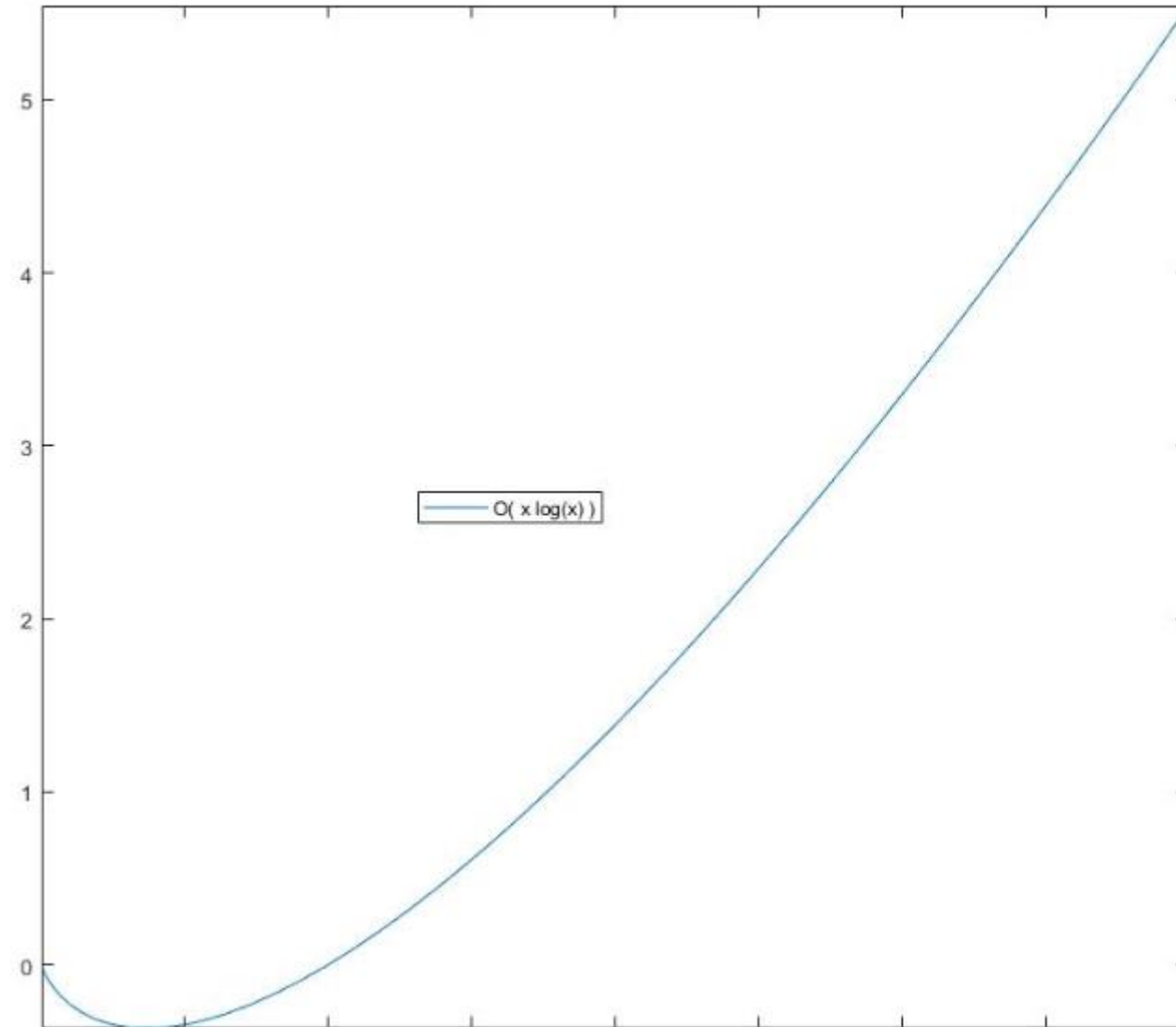
# Complejidad Lineal $O(n)$



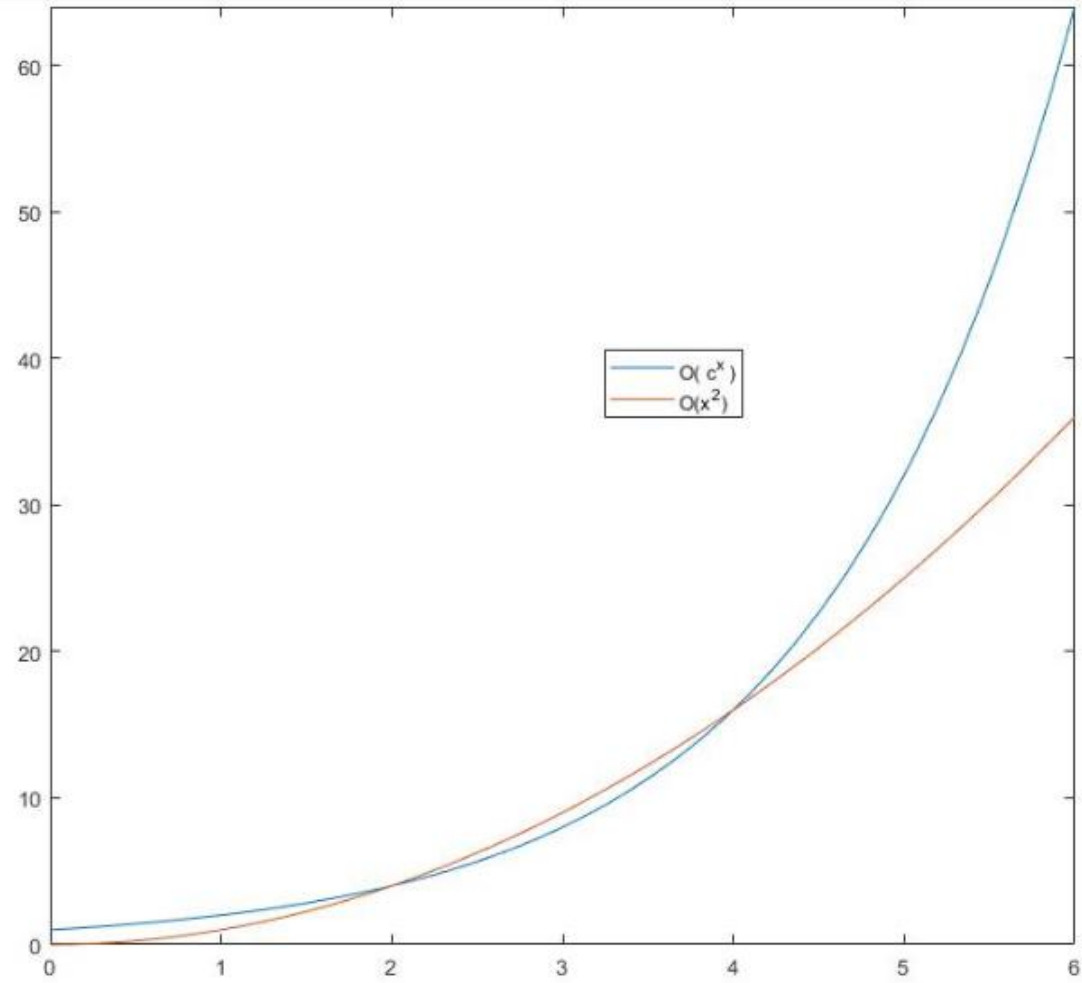
# $O(\log n)$ logarítmica



# Enelogaritmico $O(n \log n)$



# Complejidad exponencial



| $O(f(n))$  | $N=100$ | $t=2h$ | $N=200$     |
|------------|---------|--------|-------------|
| $\log n$   | 1 h     | 10000  | 1.15 h      |
| $n$        | 1 h     | 200    | 2 h         |
| $n \log n$ | 1 h     | 199    | 2.30 h      |
| $n^2$      | 1 h     | 141    | 4 h         |
| $n^3$      | 1 h     | 126    | 8 h         |
| $2^n$      | 1 h     | 101    | $10^{30}$ h |