|  |
| --- |
| ESCUELA SUPERIOR DE INFORMÁTICA, UCLM  15/03/2021 |

|  |
| --- |
| HITO 1 |
| PRACTICA 1 RAÍZ CUADRADA |
| **Estudio empírico de la complejidad de Algoritmos** |

**ALUMNOS:**

Esther Camacho Caro

Diego Dorado Galán



Contenido

[Tareas a realizar: 2](#_Toc66729150)

[COMPLEJIDADES TEÓRICAS DE LOS ALGORITMOS IMPLEMENTADOS 3](#_Toc66729151)

[Algoritmo clase Math 3](#_Toc66729152)

[Método Babilonio 3](#_Toc66729153)

[Búsqueda Binaria 3](#_Toc66729154)

[Algoritmo recursivo 4](#_Toc66729155)

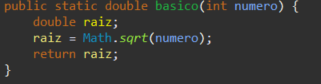
# 

# Tareas a realizar:

1. Implementar un programa Java que utilice cuadro formas de cálculo o algoritmos diferentes para calcular la raíz cuadrada perfecta de la siguiente lista de números enteros positivos: [1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256, 289, 324, 361, 400]. Estos algoritmos deben cumplir los siguientes requisitos:
   1. Una forma de cálculo debe usar la clase **Math** para obtener la raíz cuadrada.
   2. Se deben implementar dos algoritmos iterativos para calcular la raíz cuadrada, siendo uno de ellos el método **Babilonio**, y otro un método basado en la **búsqueda binaria** (Binary Search).
   3. Un cuarto método debe utilizar un **algoritmo recursivo** para calcular la raíz cuadrada. Nota: Se puede buscar y utilizar la implementación de los algoritmos anteriores, siempre y cuando se entienda su método de cálculo.
2. Determinar la complejidad teórica de cada algoritmo implementado.
3. Determinar empíricamente el tiempo de ejecución de los algoritmos y mostrar los resultados por pantalla.

# Complejidades teóricas de los algoritmos implementados

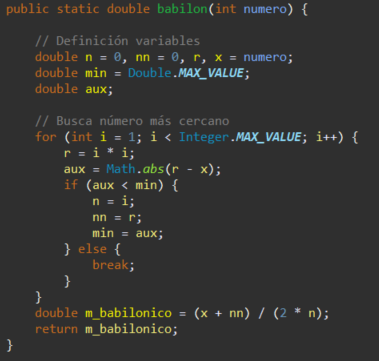
## Algoritmo clase Math

Las instrucciones se consideran de complejidad constante

T(n) = 1+1+1+1 → T(n)=4 → O **(1)**

**Complejidad constante**

## Método Babilonio



El bucle *for* tiene una complejidad O(n). El resto de las operaciones se consideran de complejidad constante → T(n) = O (1)

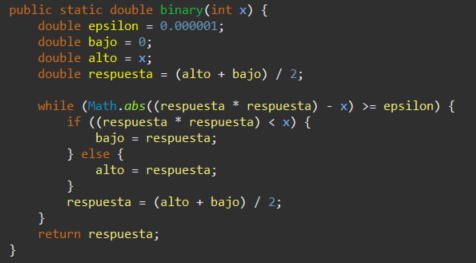
T(n) = 1+4+1+1+n\*(2+1)\*(1+1)+2+1 →

→ T(n) = 7+6n+3 → T(n)=6n+10 → **O(n)**

**Complejidad lineal**

## 

## Búsqueda Binaria

 En cada ciclo del bucle *while* se divide por dos el valor de n → T(n/2)

El resto de las operaciones se consideran de complejidad constante → T(n) = O (1)

T(n) = T(n/2) + O (1) = **O(log(n))** **Complejidad logarítmica**

## Algoritmo recursivo



Las llamadas recursivas tienen un valor de T(n) = T(n-1). El resto de las operaciones se consideran de complejidad constante → T(n) = O (1)

T(n) = T(n-1) + T(n+1) + 1

T(n) = 2T(n-1) +1

tn = 2 tn-1 + 1

tn - 2 tn-1 = 1

bn \* p(n)d

1n \* n0

xn – 2xn-1 = 1

xn-1 (x – 2) = 1

(x-2) \*(x-1)

tn = C1 (2)n + C2 (1)n  **O(2n) Complejidad exponencial**

**CONCLUSIÓN**

Obviando los datos de las primeras averiguaciones donde se tarda más en ejecutar los nuevos procesos encargados de procesar nuestro código (por lo que el tiempo calculado es mayor), podemos llegar a unas breves conclusiones.

Podemos deducir de los resultados comprobados empíricamente que el método más eficaz es el integrado en las librerías de nuestro IDE (Math.sqrt()). Los métodos de cálculo recursivo y de búsqueda binaria guardan una relación directamente proporcional con el tamaño del número a calcular. A medida que incrementa el valor del número del cual queremos calcular las raíces, mayor tiempo tardará en calcular dicha solución.

El método babilónico, en cambio, es prácticamente constante en los tiempos de ejecución. Hay escasa relación entre el tamaño del número y el tiempo de cálculo.