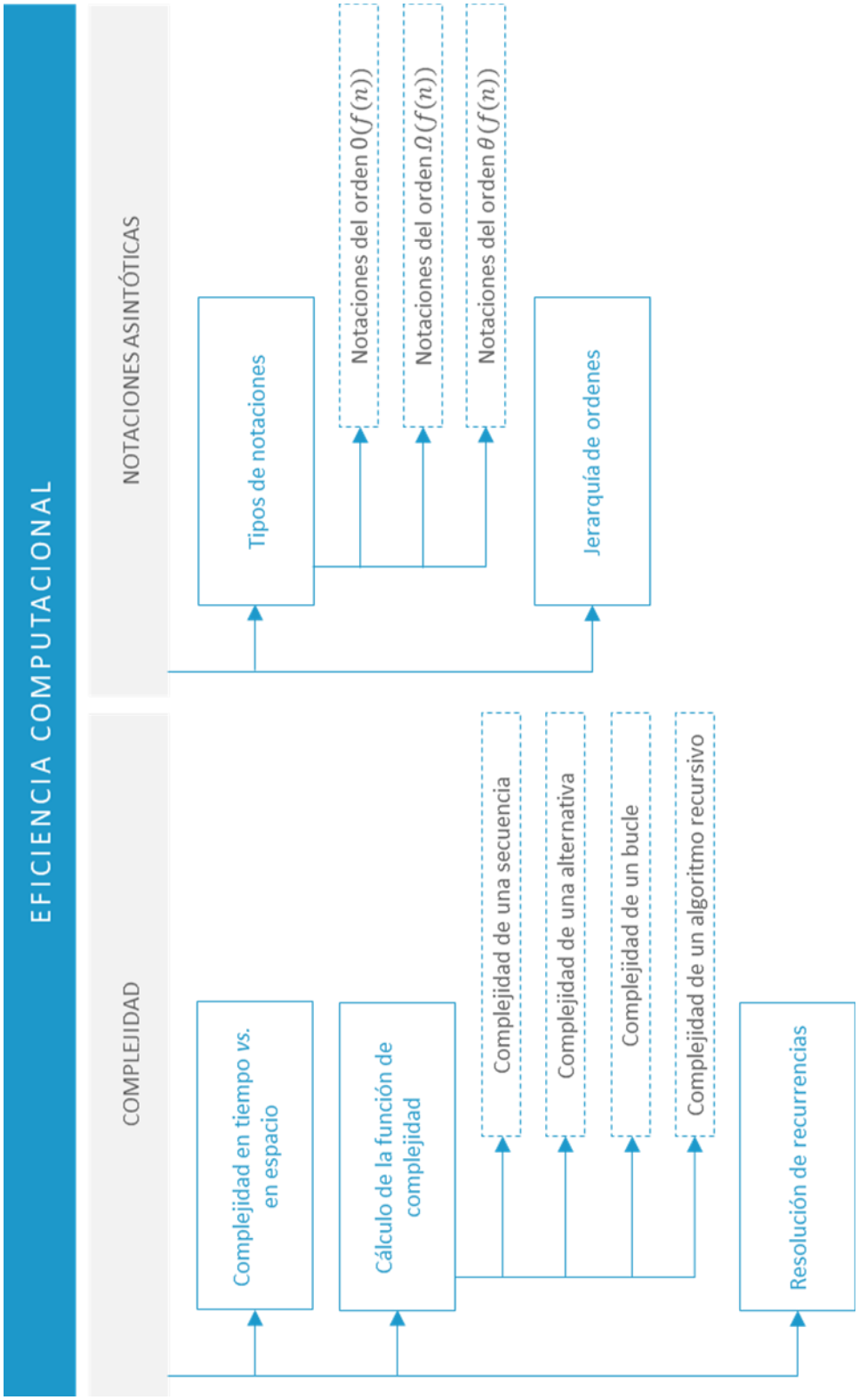
Programación Científica y HPC

Eficiencia computacional

Índice

Esquema



Ideas clave

3.1. Introducción y objetivos

El planteamiento inicial de la evaluación o análisis de un algoritmo es el estudio de su corrección, es decir, **el algoritmo hace aquello para lo que ha sido diseñado.**

Una vez satisfecha esta premisa, habría que estudiar la cantidad de recursos requeridos por el algoritmo, de esta forma, se considerará que un algoritmo es eficiente, si emplea el mínimo número de recursos para alcanzar su objetivo.

Se habla de **eficiencia computacional** para hacer referencia a **la cantidad de recursos requeridos para la ejecución.** Dentro de estos recursos se diferencian la memoria y el tiempo.

El objetivo de este tema será **fijar los fundamentos teóricos que permitan llevar a cabo el análisis de la eficiencia de los algoritmos.**

3.2. Complejidad

Complejidad en tiempo *vs.* espacio

Complejidad en espacio

En la evaluación de la memoria debería tenerse en cuenta todo el espacio requerido para ejecutar un algoritmo. Dentro de este espacio se deben considerar, entre otros, la zona de almacenamiento del código objeto, bibliotecas en tiempo de ejecución, la zona de almacenamiento de datos y la zona para los registros de activación.

Normalmente, este estudio suele limitarse a los dos últimos elementos. El espacio ocupado por las variables es fijo, a no ser que se estén utilizando estructuras dinámicas.

En cuanto a los registros de activación, el espacio estará limitado por una constante, a no ser que se emplee recursividad. En estas dos circunstancias el espacio será función del tamaño del ejemplar que se va a ejecutar, empleándose las mismas técnicas que en el estudio del tiempo.

Complejidad en tiempo

El **tiempo de ejecución de un algoritmo depende de muchos factores,** se podrían enumerar: la velocidad del procesador, los tiempos de transferencia a memoria, los cambios de contexto y el tiempo efectivo para el cómputo. Es más, este tiempo **está condicionado por la implementación misma del algoritmo,** por detalles que dependen de la habilidad del programador y el ejemplar en concreto sobre el que se ejecute.

Para simplificar el estudio se lleva a cabo un análisis teórico, independiente de la implementación, que determina la función de complejidad del algoritmo $t(n)$, como el número de instrucciones elementales que debe ejecutar una máquina ideal para resolver un ejemplar de tamaño n.

En muchos algoritmos, **el número de instrucciones varía de unos ejemplares a otros.** Por ejemplo, si se supone que se busca un número dentro de una lista de n enteros. El número de instrucciones no será el mismo, si se encuentra en la primera posición o en la última. Sería posible, entonces, realizar un estudio en el peor caso, en el mejor caso o como media.

Las **técnicas de análisis estudian la peor situación que se puede esperar,** es decir, aquella que requiere el mayor número de instrucciones. Aun así, hay casos en los que resulta interesante analizar el caso promedio.

Es importante señalar que se considera **instrucción elemental** a **aquella cuyo tiempo de ejecución,** para una determinada implementación, **está acotado por una constante.**

Cálculo de la función de complejidad

Se van a mostrar distintos casos en los que se explicará cómo se calcula la función de complejidad.

Complejidad de una secuencia

Se calcula como la suma de complejidades de las instrucciones de la secuencia.

```
función intercambio (a,b)
inicio
                aux←a
    a←b
                b←aux
fin
```

Figura 1. Secuencia de instrucciones. Fuente: elaboración propia.

La función de complejidad es $\boldsymbol{t(n) = 3}$, porque son tres sentencias elementales que se ejecutan en todos los casos.

Complejidad de una alternativa

Se calcula sumando a la complejidad de la condición, el máximo de la complejidad de la rama si y la rama si no.

```
función max(a,b)
inicio
            si a>b entonces devolver a
                          sino devolver b
fin si
fin
```

Figura 2. Función con alternativa. Fuente: elaboración propia.

La función de complejidad es:

$$t(n) = t_{cond}\ (n) + max\ (t_{si}\ (n), t_{sino}\ (n)) = 1 + 1 = 2$$

Complejidad de un bucle

Se calcula como la suma de las complejidades de las instrucciones que se ejecutan tras todas las iteraciones del bucle. En el caso de ser un bucle del tipo `repetir-mientras` hay que ponerse en el peor caso, aquel en el que se ejecute el mayor número de iteraciones.

```
funcion buscar(lista[1..n], e)
inicio
            i ← 1
            mientras (i<n) and (lista[i]!= e)
                        incrementa(i)
            fin mientras
            devolver  lista[i]=e
fin
```

Figura 3. Función con bucle `while`. Fuente: elaboración propia.

La función de complejidad es:

$$t(n) = 1 + t_{bucle}(n) + 1 = n + 2$$

Complejidad de un algoritmo recursivo

$t(n)$es el número de instrucciones que se van a ejecutar para resolver un ejemplar de tamaño n. **Los algoritmos recursivos resuelven el problema como composición de las soluciones de sus problemas más pequeños,** por tanto, aparecerá una recursión en la función de complejidad. Esta reiteración deberá resolverse con métodos matemáticos de resolución de recurrencias.

```
función factorial(n)
inicio
            si n=0  entonces devolver 1
                            sino devolver n*factorial(n-1)
    fin si
fin
```

Figura 4. Función recursiva. Fuente: elaboración propia.

La función de complejidad es:

$$t(n) = \begin{cases} 1 & si\ n = 0 \\ 1 + t(n-1) & si\ n > 0 \end{cases}$$

Si $t(n)$ es el número de instrucciones necesarias para calcular el factorial de n, el cálculo del factorial de $n-1$ requerirá $t(n-1)$ instrucciones.

La recurrencia que se presenta en la función **debe ser resuelta.** Existen varias formas de resolución, pero solo se vera la resolución de recurrencias por expansión.

Resolución de recurrencias por expansión

El método consiste en sustituir, en la función de complejidad, hasta un término $t(k)$ conocido. Por ejemplo, para la resolución de la función recurrente de complejidad del algoritmo de factorial, se debe expandir la recurrencia, en este caso, hasta $t(0)$, que es el término conocido, entonces:

$$t(n) = 1 + t(n-1) = 2 + t(n-2) = \cdots .. = i + t(n-i) = n + t(0) = n + 1$$

El término conocido es $t(0) = 1, n - i = 0 \Longrightarrow i = n$.

Antes de seguir mira el vídeo de **Ordenes de complejidad y ejemplos** del aula virtual.

Accede al vídeo

3.3. Notaciones asintóticas

Las funciones de complejidad calculan la función $t(n)$, que es el coste de ejecución cuando la entrada crece. Sin embargo, el uso de funciones de complejidad para comparar algoritmos y elegir el mejor resulta poco práctico. Es por eso, por lo que en su lugar, se trabaja con notaciones asintóticas.

Las notaciones asintóticas **sirven para caracterizar el comportamiento de la función de complejidad cuando n se hace muy grande,** es decir, infiere el orden de crecimiento de la operación elemental al crecer la entrada.

De esta forma, una vez analizado un algoritmo, se caracteriza su complejidad con la notación asintótica y, es esa última, la que se maneja a efectos de realizar comparaciones entre algoritmos y tomar decisiones.

Las notaciones asintóticas que existen son:

- Notación para el orden de $O(f(n))$: cota superior del tiempo de ejecución.
- Notación $\Omega(n)$: cota inferior del tiempo de ejecución.
- Notación $\theta(n)$: cota inferior y superior del tiempo de ejecución.

A continuación, se explicarán brevemente cada una de las notaciones enumeradas.

Notación del orden de $O(f(n))$

$O(f(n))$ es el conjunto de funciones acotadas superiormente por un múltiplo real positivo de $f(n)$.

$$O(f(n)) = \{t(n): \exists c \in \mathbb{R}^+ \ \forall n \in \mathbb{N}, t(n) \leq c * f(n).$$

En la práctica, para ver si una función $f(n)$ pertenece al orden, es decir, está en el orden de $g(n)$, se comprueba que:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \leq cte$$

Regla del límite

Para demostrar si unas funciones pertenecen al orden de otras, se usa la regla del límite, para la que dadas dos funciones cualesquiera $f\ y\ g: \mathbb{N} \to \mathbb{R}^{\geq 0}$.

- Si $\lim_{n\to\infty} \frac{f(n)}{g(n)} \in \mathbb{R}^+$, entonces $f(n) \in O(g(n))\ y\ g(n) \in O(f(n))$.
- Si $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$, entonces f(n)∈O(g(n)) pero g(n)∉O(f(n)).
- Si $\lim_{n\to\infty} \frac{f(n)}{g(n)} = +\infty$, entonces $f(n) \notin O(g(n))\ pero\ g(n) \in O(f(n))$.

Ejemplo 1. Notación del orden de ejemplos

$¿n^2 + 2n \in O(n^3)?$

$f(n) = n^2 + 2n \;\; g(n) = n^3$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \leq cte => f(n) \in O(n^3) \;\; pero \; g(n) \notin O(f(n))$$

$¿n^2 + 2n \in O(n^2)?$

$f(n) = n^2 + 2n \;\; g(n) = n^2$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 1 \leq cte => f(n) \in O(n^2) \;\; y \; g(n) \in O(f(n))$$

$¿n^2 + 2n \in O(n)?$

$f(n) = n^2 + 2n \;\; g(n) = n$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \nleq cte => f(n) \notin O(n) \; y \;\; g(n) \in O(f(n))$$

Figura 5. Notación del orden con ejemplos. Fuente: elaboración propia.

Propiedades de la notación del orden

- $\forall f: N \to R+, f \in O(f) (reflexividad)$.
- $Si\ f \in O(g) y\ g \in O(h)$, entonces $f \in O(h)$ $(transitividad)$.
- $\forall c > 0, O(f) \in O(cf)$.

Reglas

Regla de las sumas: $O(f) + O(g) = O(f + g) = O(max\{f, g\})$.

Regla del producto: $O(f)O(g) = O(fg)$.

Notación del orden de $\Omega(f(n))$

$\Omega(f(n))$ es el conjunto de funciones acotadas inferiormente por un múltiplo real positivo de $f(n)$.

$$\Omega(f(n)) = \{t(n): \exists d \in \mathbb{R}^+ \; \forall n \in \mathbb{N}, t(n) \geq d * f(n).$$

En este caso, se puede ver la dualidad, de forma que:

$$t(n) \in \Omega\big(f(n)\big) \Leftrightarrow f(n) \in O(t(n))$$

Notación del orden exacto $\theta(f(n))$

$\theta\big(f(n)\big)$ es el conjunto de funciones acotadas superior e inferiormente por sendos múltiplos reales positivos de $f(n)$.

$$\theta\big(f(n)\big) = \{t(n): \exists c_1, c_2 \in \mathbb{R}^+ \ \forall n \in \mathbb{N}, c_1 * f(n) \leq \ t(n) \leq c_2 * f(n)$$

En la práctica, para ver si una función $f(n)$ pertenece al orden exacto, está en el orden de $g(n)$, se comprueba que:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = cte > 0$$

Regla del límite

Para demostrar si unas funciones pertenecen al orden de otras, se usa también la regla del límite, para la que dadas dos funciones cualesquiera $f \ y \ g: \mathbb{N} \to \mathbb{R}^{\geq 0}$.

- Si $\lim_{n\to\infty} \frac{f(n)}{g(n)} \in \mathbb{R}^+$, entonces $f(n) \in \theta\big(g(n)\big)$.
- Si $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$, entonces $f(n) \in O\big(g(n)\big) \ pero \ f(n) \notin \theta(g(n))$.
- Si $\lim_{n\to\infty} \frac{f(n)}{g(n)} = +\infty$, entonces $f(n) \notin \theta\big(g(n)\big)$.

Ejemplo 2. Notación exacta

¿$n^2 + 2n \in \theta(n^3)$?

$f(n) = n^2 + 2n \;\; g(n) = n^3$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 => f(n) \notin \theta(n^3)$$

¿$n^2 + 2n \in \theta(n^2)$?

$f(n) = n^2 + 2n \;\; g(n) = n^2$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 1 > 0 => f(n) \in \theta(n^2)$$

¿$n^2 + 2n \in \theta(n)$?

$t(n) = n^2 + 2n \;\; f(n) = n$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \nleq cte => f(n) \notin \theta(n)$$

Figura 6. Notación exacta. Fuente: elaboración propia.

Propiedades de la notación asintótica

- $\forall f: N \to R+, f \in \theta(f) (reflexividad)$.
- $Si\ f \in \theta(g)\ y\ g \in \theta(h)\ entonces\ f \in \theta(h)\ (transitividad)$.
- $f \in \theta(g) \Leftrightarrow g \in \theta(f)$.

Reglas

Regla de las sumas: $\Theta(f) + \Theta(g) = \Theta(f + g) = \Theta(max\{f, g\})$.

Regla del producto: $\Theta(f)\Theta(g) = \Theta(fg)$.

Análisis de algoritmos recurrentes divide y vence

El algoritmo divide y vence muestran una función recurrente de complejidad con la siguiente forma:

$t(n) = \begin{cases} f(n) & si\ 0 \leq n \leq n_0 \\ at\left(\frac{n}{b}\right) + g(n) & si\ n > n_0 \end{cases}$ donde $g(n) \in \theta(n^k).\ g(n)$ es el coste de dividir y combinar los subproblemas

Si $\alpha = \log_b a$, entonces:

$$t(n) = \begin{cases} \theta(n^k) & si\ \alpha < k \\ \theta(n^k \log n) & si\ \alpha = k \\ \theta(n^\alpha) & si\ \alpha > k \end{cases}$$

Jerarquía de ordenes

Nótese que, entre los distintos ordenes, se puede definir una jerarquía de menor a mayor complejidad. Su representación se muestra en la Figura 7.

La jerarquía de complejidades es:

$$O(1) \subset O(log\ n) \subset O(n) \subset O(n\ log\ n) \subset O(n^2) \subset O(n^3) \subset O(2^n) \subset O(n!) \subset O(n^n)$$

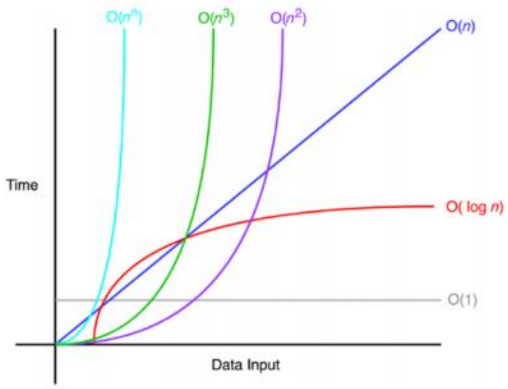


Figura 7. Representación ordenes de complejidad. Fuente: https://ichi.pro/es/entendiendo-la-notacion-big-o-214560921424604

3.4. Referencias bibliográficas

Ichi.Pro (s. f.). *Representación ordenes de complejidad* [Imagen]. Entendiendo la notación Big-O (ichi.pro)

3.5. Cuaderno de ejercicios

Ejercicio1

Se debe demostrar que $t(n) = 5 * 2^n + n^2 \in \theta(2^n)$, es decir, $t(n)$ pertenece al orden exacto de 2^n.

Para ello, se aplicará la regla del límite:

$$\lim_{n\to\infty} \frac{5 * 2^n + n^2}{2^n} = \lim_{n\to\infty} \frac{5 * 2^n * \log(2) + 2 * n}{2^n * \log(2)} = \lim_{n\to\infty} (5 + \frac{2 * n}{2^n * \log(2)})$$

$$= \lim_{n\to\infty} (5 + \frac{2 * n}{\log 2^{2^n}})$$

$$= \lim_{n\to\infty} (5 + \frac{2 * n}{\mathrm{n} * \log 2^2}) = \lim_{n\to\infty} (5 + \frac{2}{\log 2^2}) = \lim_{n\to\infty} 5 + \frac{2 * n}{\log 2^{2^n}} = 6$$

Como es una constante se puede afirmar que $5 * 2^n + n^2 \in \theta(2^n)$.

Ejercicio2

Se debe demostrar que $t(n) = n^3 + 9n^2 + \log(n) \in \theta(n^3)$, es decir, $t(n)$ pertenece al orden exacto de n^3.

Para ello, se aplicará la regla del límite:

$$\lim_{n\to\infty} \frac{n^3 + 9n^2 + \log(n)}{n^3} = (se\ \ aplica\ L'Hopital) = \lim_{n\to\infty} \frac{3n^2 + 18n + \frac{1}{\mathrm{n}}}{3n^2} = \lim_{n\to\infty} (1 + \frac{6}{n} + 1/3n^3) = 1$$

Como es una constante, se puede afirmar que $n^3 + 9n^2 + \log(n) \in \theta(n^3)$.

Ejercicio 3

Se quiere demostrar que $\forall a, b > 1 \ \log_a n \in O(\log_b n)$.

La solución se obtiene, nuevamente, por la regla del límite y propiedades de los logaritmos:

$$\lim_{n\to\infty} \frac{\log_a n}{\log_b n} = \lim_{n\to\infty} \frac{\log_a n}{\frac{\log_a n}{\log_a b}} = \log_a b => constante$$

Luego $\log_a n \in O(\log_b n)$.

Ejercicio 4. Esquema de Horner para el cálculo del valor de un polinomio en un punto

Los pasos que realiza el algoritmo son los siguientes:

- Se almacenan los coeficientes del polinomio en un array.
- Se usa el esquema de Horner según el que:

$$c_0x^3 + c_1x^2 + c_2x + c_3 = \big((c_0x + c_1)x + c_2\big)x + c_3$$

La implementación en Python es:

```
def evaluaPolinomio(p,x):
    valor=0
    for i in range(0,len(p)):
        valor=valor*x+p [i]
    return valor
polinomio=[1,2,-1,1]
evaluaPolinomio(polinomio,2)

Se obtiene  como resultado 15
```

Figura 8. Función que evalúa polinomio de Horner. Fuente: elaboración propia.

Análisis del algoritmo

Número de iteraciones=n (grado del polinomio +1, cte)

$$t(n) = cte_1(la\ inicialización) + cte_2 * n \in \theta(n) => \ \ lineal$$

Ejercicio 5. Multiplicación rusa

El método de multiplicación rusa consiste en multiplicar sucesivamente por 2 el multiplicando y dividir por 2 el multiplicador, hasta que el multiplicador tome el valor 1. Luego, se suman todos los multiplicandos correspondientes a los multiplicadores impares. Dicha suma es el producto de los dos números.

```
def  rusa(x,y):
    acumulador=0
    while x>=1:
        if x%2==1:
            acumulador=acumulador+y
        x=x//2
        y=y*2
    return acumulador
rusa(5,7)

Se obtiene  35
```

Figura 9. Multiplicación rusa. Fuente: elaboración propia.

Análisis del algoritmo

Se cuenta el número de iteraciones del bucle, según varía la x, hasta valer 1.

X	Iteración
n	1
$n/2$	2
$n/4$	3
...	...
$\frac{n}{2^{k-1}} = 1$	k

Tabla1. Valor de la variable x en cada iteración.Fuente: elaboración propia.

Entonces, $k = \log_2 n +$ t $t(n) = c_1 + c_2 \log_2 n \in \theta(\log\ n)$ logarítmica.

Ejercicio 6

¿Cuál es el orden de complejidad temporal del siguiente fragmento de código cuando la operación básica es $j\ \%\ i\ =\ 1$?

```
sum = 0
for i in range(1, 2*n):
    for j in range(1, i*i):
        for k in range(1, j):
            if j % i == 1:
                sum=sum+1
```

Figura 10. Código del ejercicio 6. Fuente: elaboración propia.

Solución: $O(n^5)$.

Ejercicio 7

Encuentre el coste computacional $C(n)$ del siguiente algoritmo:

```
def f(n):
  if n<=1:
    return 1
  else:
    return n*f(n/2)
```

Figura 11. Código del ejercicio 7. Fuente: elaboración propia.

Solución: $C(n) = \log_2 n$.

A fondo

Complejidad algorítmica ejemplos de Python

Verbel de Léon, A. (12 de febrero de 2018). *Complejidad Algorítmica-Ejemplos Python. PHP.* Mi camino Master. http://micaminomaster.com.co/grafo-algoritmo/complejidad-algoritmica-ejemplos-python-php/

Conjunto de ejemplos implementados en Python con análisis de las complejidades, esto permitirá profundizar en los conceptos asociados y la forma de calcular la complejidad de los algoritmos.

Análisis de algoritmos y complejidad

Viera Class. (29 de enero de 2018). *Análisis de Algoritmos, Complejidad de algoritmos (actualizado)* [Archivo de vídeo]. Youtube. https://www.youtube.com/watch?v=Sibd8jSTdUQ

En este vídeo se presentan ejemplos significativos para realizar el análisis de la complejidad de distintos algoritmos, que ayuda a entender cómo se concibe la complejidad.

Introduction to Big O Notation and Time Complexity

CS Dojo. (14 de mayo de 2018). *Introduction to Big O Notation and Time Complexity (Data Structures & Algorithms #7)* [Archivo de vídeo]. Youtube. https://www.youtube.com/watch?v=D6xkbGLQesk

La notación de la gran O es una de las más utilizadas y este vídeo permite profundizar sobre ella para poder representar la complejidad temporal de los algoritmos.

Practicar conceptos avanzados de *git* en el navegador

Learn Git Branching (https://learngitbranching.js.org).

Learn Git Branching te permite repasar lo básico de *git* y practicar conceptos más avanzados, como crear y fusionar ramas, revertir cambios, incorporar cambios de terceros, gestionar conflictos, etc. Todo ello en una plataforma web sin necesidad de instalar nada.

Referencia rápida de *git* en español

Git Cheat Sheets (Git Cheat Sheets - GitHub Cheatsheets).

En la página encontrarás una pequeña guía rápida con los comandos más comunes, con descripciones en español y otros idiomas. Guárdala y extiéndela con los nuevos comandos que vayas aprendiendo.

Test

1. La notación asintótica captura:

A. El comportamiento de las operaciones básicas con números pequeños.

B. El comportamiento de un algoritmo cuando n es infinito.

C. El orden de crecimiento de una operación básica en función de n.

D. Ninguna de las anteriores es correcta.

2. En la notación asintótica la función $f(n)$ es:

A. La que mide el tiempo de ejecución.

B. La que mide el número de operaciones básicas.

C. Las respuestas A y B son correctas.

D. Una función simple.

3. El conjunto de todas las funciones con menor o igual orden de crecimiento que $f(n)$ se corresponde con:

A. Cota superior asintótica O.

B. Cota inferior asintótica Ω.

C. Orden exacto θ.

D. Ninguna de ellas.

4. El conjunto de todas las funciones con mayor o igual orden de crecimiento que $f(n)$ para cualquier valor de c positivo corresponde con:

A. Cota superior asintótica O.

B. Cota inferior asintótica Ω.

C. Orden exacto θ.

D. Ninguna de ellas.

5. Dados las funciones computacionales $f(n) = 2^n \; y \; g(n) = 2^{2n}$, se cumple que:

A. $g(n) \in O(f(n))$.

B. $g(n) \notin O(f(n))$.

C. $f(n) \in \theta(g(n))$.

D. $g(n) \in \theta(f(n))$.

6. ¿Cuándo se debe resolver una recurrencia?

A. En el análisis de algoritmos iterativos.

B. En el análisis de algoritmos recursivos.

C. En ambos.

D. En ninguno.

7. ¿Cuál de estas funciones de coste de ejecución pertenecen a $O(n^3)$ y a $\theta(n^3)$?

A. $1/3n^3 + 2^{500}n^2$.

B. $0.00001 \cdot n^4$.

C. $2^{500}n^2$.

D. Ninguna de las anteriores es correcta.

8. Se considera $f(n) = \; 2^n \; y \; g(n) = \; 2^{2n}$ se puede decir que:

A. $f(n) \in O(g(n)) \; pero \; g(n) \notin O(f(n))$.

B. $f(n) \in O(g(n)) \; y g(n) \in O(f(n))$.

C. $f(n) \in O(g(n)) \; y g(n) \in \theta(f(n))$.

D. Ninguna de las anteriores es verdadera.

9. Indique cuál de las siguientes opciones es verdadera sabiendo que el tiempo de ejecución de un algoritmo es $t(n) = 4n^2 - 3n + 2$:

A. $t(n) \notin O(n^2 \, log \, (n))$.

B. $t(n) \notin O(n^3)$.

C. $t(n) \in O(nlog \, (n))$.

D. $t(n) \in O(n^2)$.

10. El orden de complejidad temporal del siguiente fragmento de código es:

```
def codigo(n):
  a = 0;
  for j in range(1, n+1):
    for k in range(1, n+1):
      a += a + ( k*j )
       return a
```

Figura 12. Código de la pregunta 10 del test. Fuente: elaboración propia.

A. $O(1)$.

B. $O(log \; n)$.

C. $O(n)$.

D. $O(n\text{^}2)$.