Introducción a la algoritmia

Tema 1

Los contenidos de esta presentación han sido adaptados a partir del material creado por los profesores Javier Junquera Sánchez y Francisco Manuel Sáez de Adana Herrero.

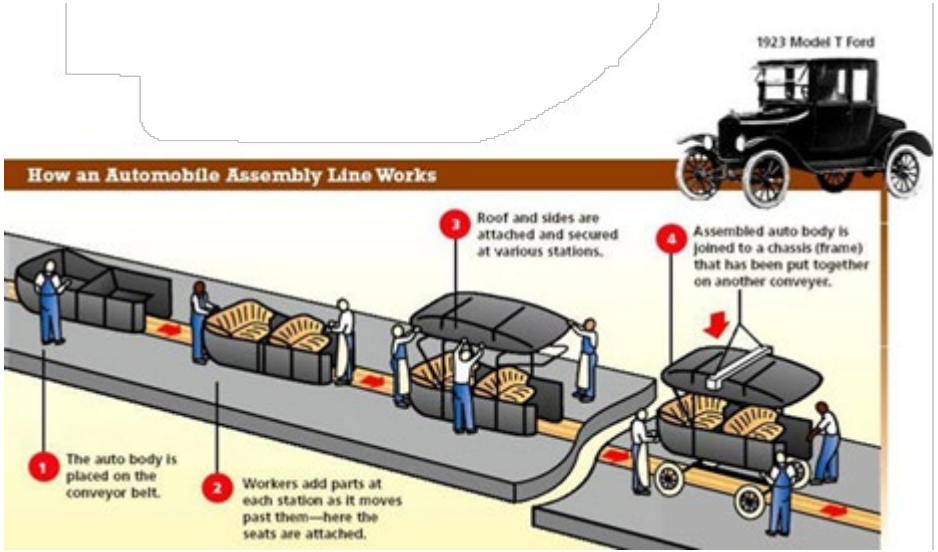
Definición

Universidad de Alcalá

Un algoritmo es una descripción abstracta y ordenada de todas las acciones que se deben realizar, así como la descripción de los datos que deben ser manipulados por dichas acciones para llegar a la solución de un problema.

Objetivo

El objetivo de un algoritmo es resolver un problema, independientemente de cómo se presente. Puede haber varios algoritmos que resuelvan el mismo problema de forma distinta, pero todos tienen que poder resolver el problema.



Principios

Universidad de Alcalá

Un algoritmo debe:

- Ser **independiente** tanto del lenguaje de programación en que se exprese, como del ordenador en que se vaya a ejecutar.
- Ser **claro, sencillo y ordenado**.
- Tener un número **finito** de pasos (así como un principio y un fin).
- Ser **flexible** (para facilitar su mantenimiento).
- Ser **determinista**, de manera qué si se ejecuta con los mismos datos de entrada, debe obtener el mismo resultado.

Características

Universidad de Alcalá

- Los algoritmos resuelven problemas.
- Un problema suele tener muchos ejemplares.
- Los algoritmos deben funcionar correctamente en todos los casos del problema que afirman resolver.
- Un solo caso erróneo, hace que el algoritmo sea incorrecto.
- Normalmente encontramos más de una forma de resolver un problema, es decir, hay varios algoritmos que resuelven el mismo problema (eficiencia o mantenibilidad).

Multiplicación

Universidad de Alcalá

🇪🇸 España:

		2	3
	x	1	3
		6	9
2		3	
2		**9**	**9**

🇷🇺 Rusia:

Doble	Mitad
23	**13**
46	6
92	**3**
184	**1**

Mitades Impares

23 x 13 = 23 + 92 + 184 = **299**

Pseudocódigo

Universidad de Alcalá

Forma de expresar algoritmos de manera estructurada sin depender de un lenguaje específico.

Reglas generales:

- **Estructura clara:** Usa palabras clave como INICIO, FIN, SI, MIENTRAS, etc.
- **Uso de indentación:** Para representar estructuras anidadas.
- Variables declaradas antes de su uso.
- Evitar instrucciones ambiguas.
- Ejemplo:

```
INICIO
    Definir A, B, SUMA como Enteros
    Escribir "Introduce dos números"
    Leer A, B
    SUMA ← A + B
    Escribir "La suma es: ", SUMA
FIN
```

Pseudocódigo

Universidad de Alcalá

Ejemplo de la multiplicación Rusa

```
INICIO
    LEER m, n
    RESULTADO = 0

    REPETIR
        SI m es impar ENTONCES
            RESULTADO = RESULTADO + n
        FIN SI
        m = m / 2
        n = n + n
    HASTA m = 1

    IMPRIMIR RESULTADO
FIN
```

```
INICIO
    LEER m, n
    RESULTADO = 0

    REPETIR
        SI m es impar ENTONCES
            RESULTADO = RESULTADO + n
        m = m / 2
        n = n + n
    HASTA m = 1

    IMPRIMIR RESULTADO
FIN
```

OJO: Se puede usar o no FIN SI, pero es recomendable usarlo cuando existe mas de una condición para evitar confusión.

Principio de Invarianza

Dos implementaciones diferentes de un mismo algoritmo no diferirán en eficiencia más que, a lo sumo, en una constante multiplicativa.

$$T_1(n) = cT_2(n)$$

Diremos que un algoritmo consume un tiempo de orden $T(n)$, para una función T, si existe una constante positiva c y una implementación del algoritmo, capaz de resolver cualquier caso del problema en un tiempo acotado superiormente por $cT(n)$ unidades de tiempo. Donde n es el tamaño del caso considerado.

El principio de invarianza es válido, independientemente del ordenador usado o del lenguaje de programación.

Tiempo de ejecución

Universidad de Alcalá

Como hemos dicho, el número de entradas (n), es un factor determinante para calcular el tiempo de ejecución.

```
t = T(n)
```

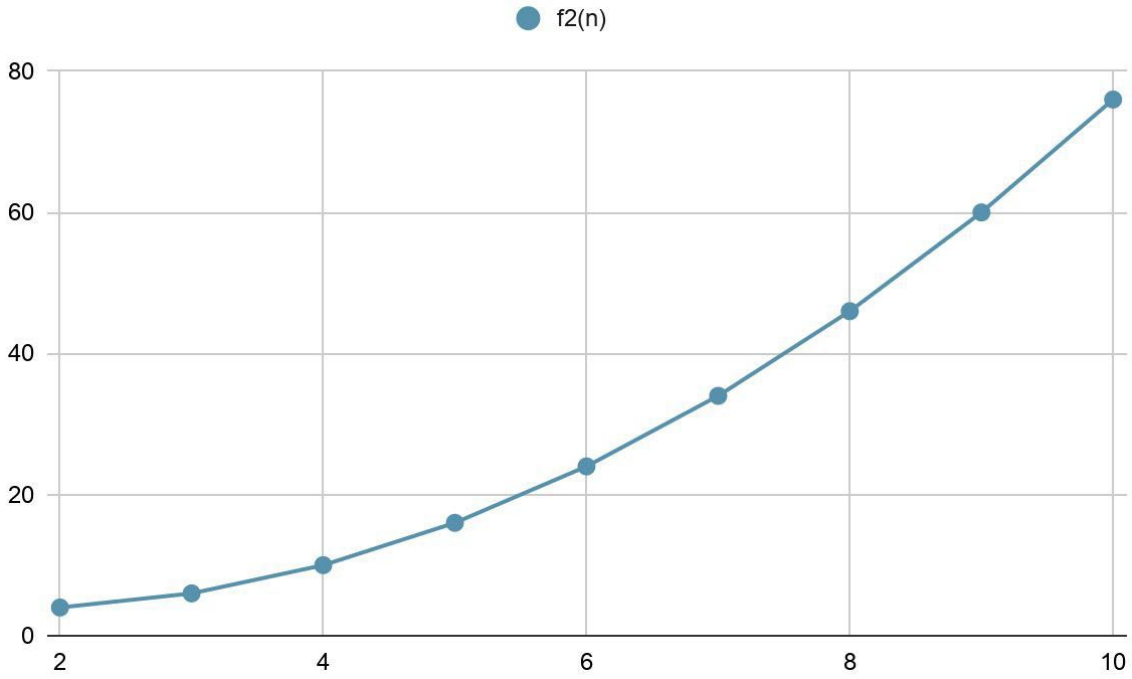
Pero el tiempo final dependerá de “que forma” tienen estas entradas.

```
Caso Mejor < Caso Medio < Caso Peor
```

Tiempo de ejecución

Universidad de Alcalá

- Coste de cualquier operación = 1



$$f_2(n) = n^2 - 3n + 6$$

Notación asintótica

Universidad de Alcalá

Un algoritmo con un tiempo de ejecución $T(n)$ se dice que **es de orden** $O(f(n))$ si existe una constante positiva c y un número entero n_0 tales que para todo $n>=n_0$ entonces $T(n)<=cf(n)$.

Por ejemplo, si $T(0)=1$, $T(1)=4$ y $T(n)=(n+1)^2$. Entonces $T(n)$ es $O(n^2)$, puesto que si tomamos $n_0=1$ y $c=4$, se verifica que para todo $n>=1$ entonces $(n+1)^2<=4n^2$.

$$T(n) \in O(f(n))$$

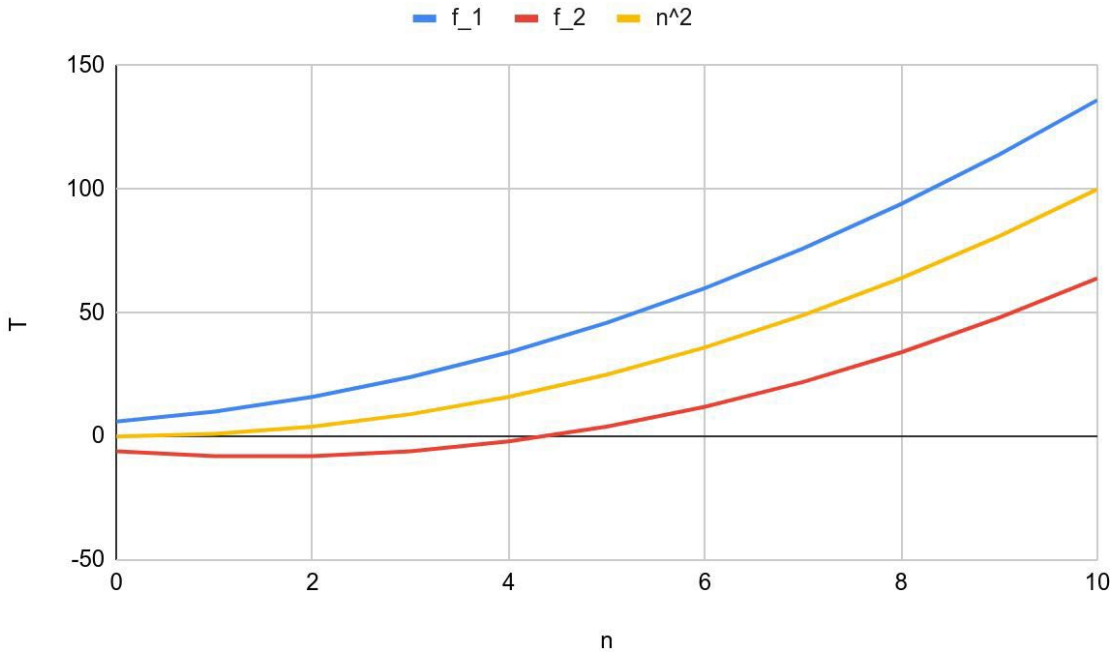
Cuando $T(n)$ es $O(f(n))$, estamos dando una **cota superior para el tiempo de ejecución**, que siempre referiremos al peor caso.

Notación asintótica

$f_1(n) = n^2 + 3n + 6$

$O(n^2)$

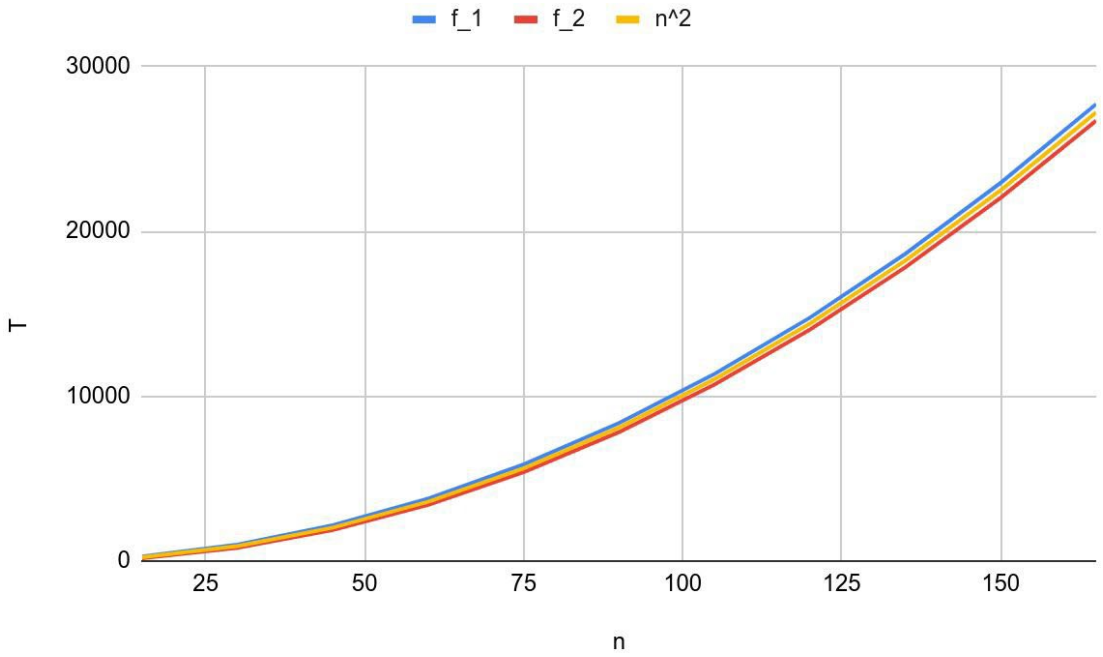
$f_2(n) = n^2 - 3n - 6$



Notación asintótica

$f_1(n) \in O(n^2)$

$f_2(n) \in O(n^2)$



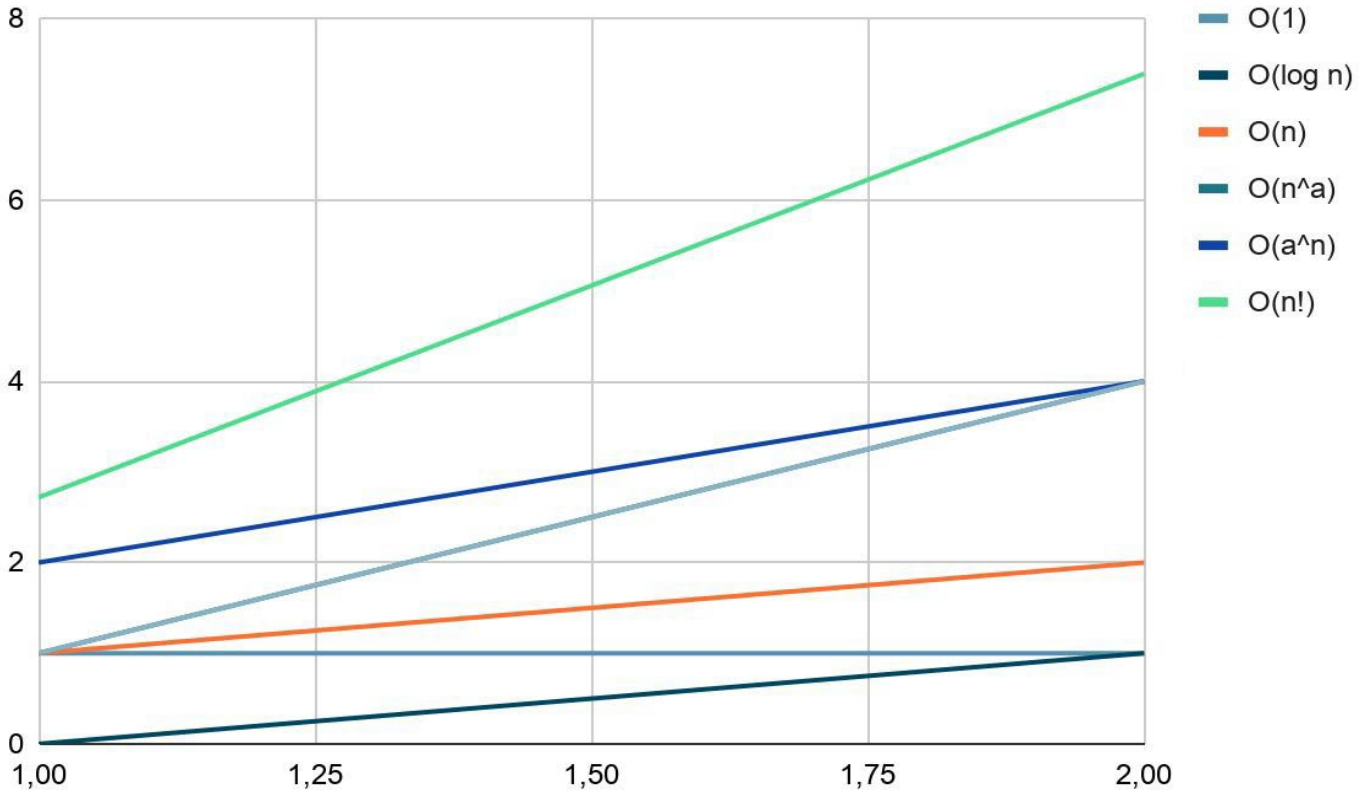
Orden de complejidad

Universidad de Alcalá

$O(1)$	Constante
$O(\log n)$	Logarítmico
$O(n)$	Lineal
$O(n^2)$	Cuadrático
$O(n^a)$	Polinómico
$O(a^n)$	Exponencial
$O(n!)$	Factorial
$O(n^n)$	Potencial

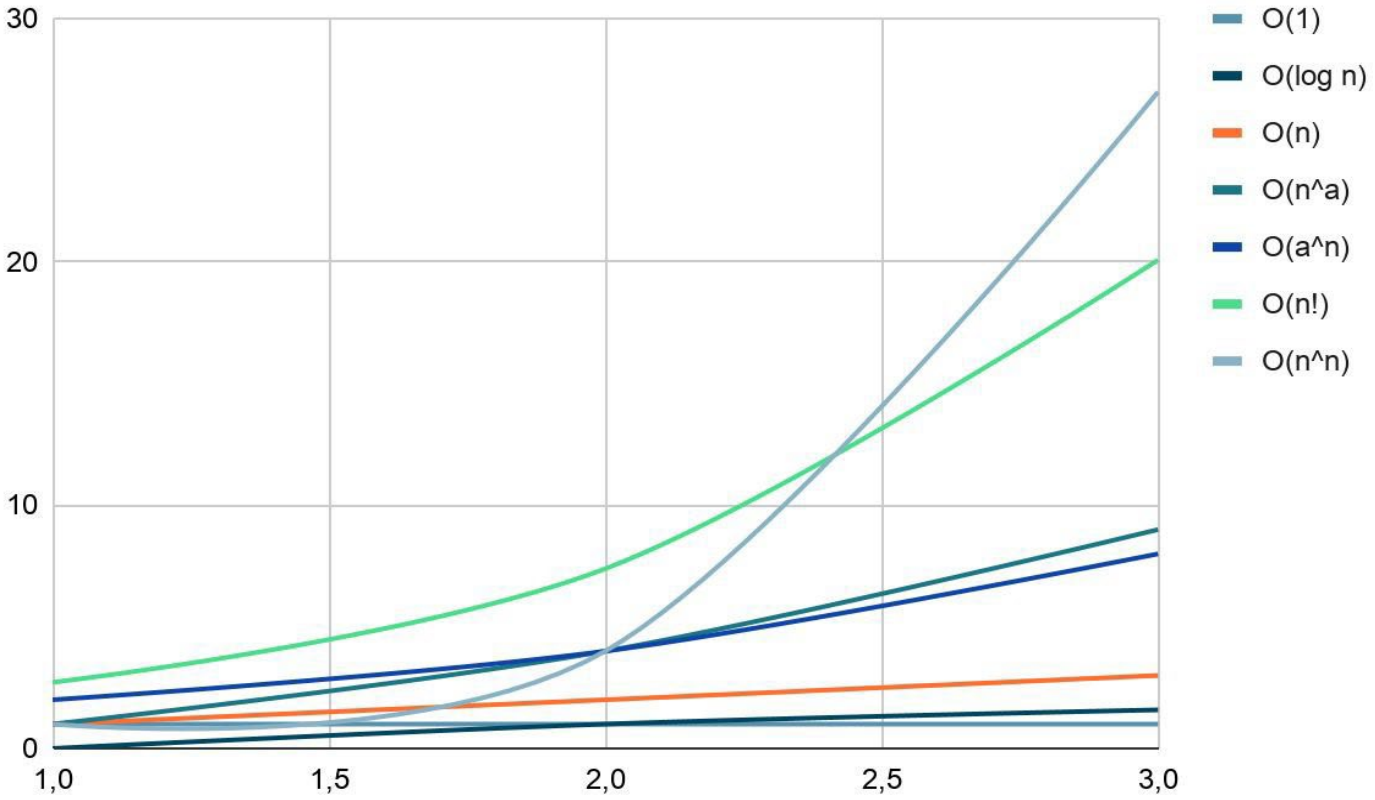
Orden de complejidad

Universidad de Alcalá

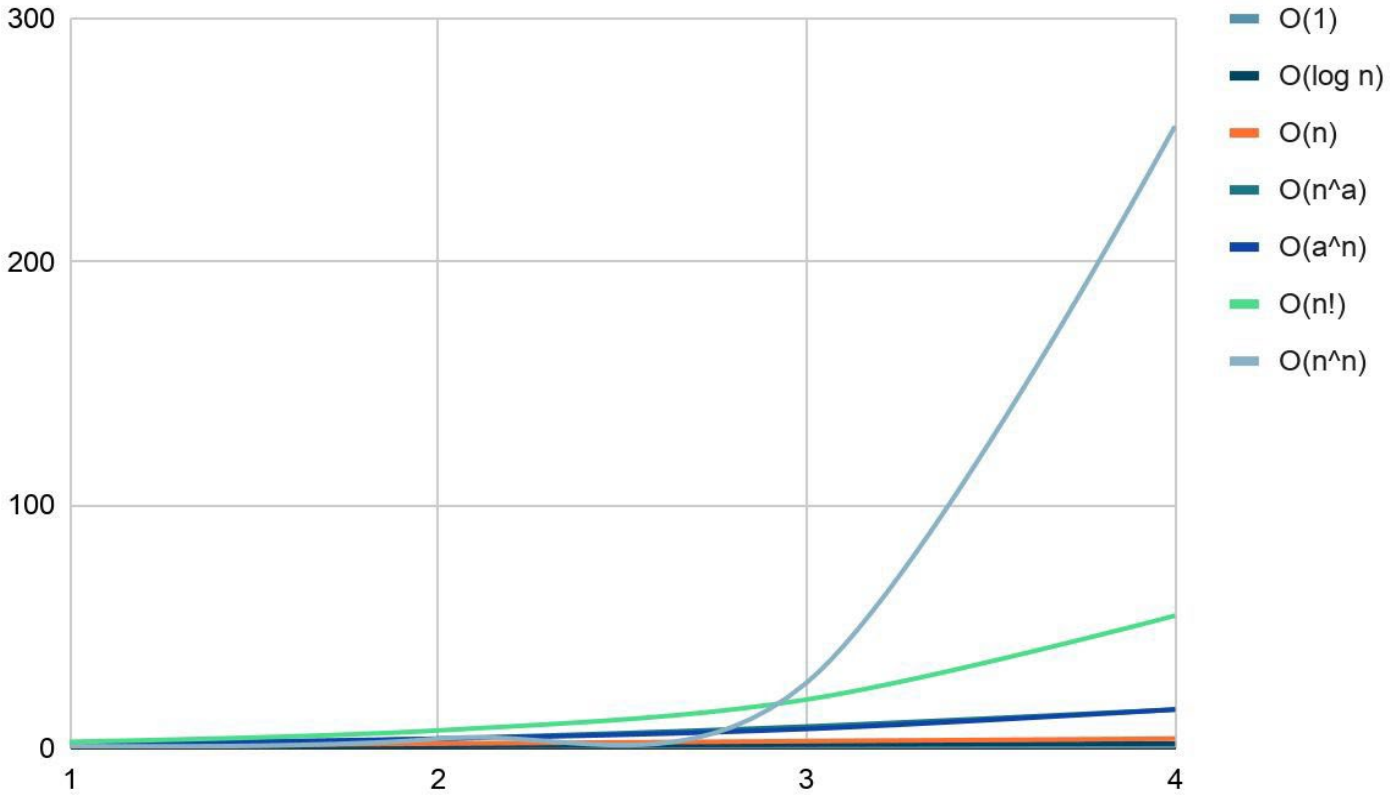


Orden de complejidad

Universidad de Alcalá



Orden de complejidad



Cálculo de la eficiencia

Universidad de Alcalá

- **Regla de la suma:** partes independientes.

Si $T_1(n) \rightarrow O(f(n))$

y $T_2(n) \rightarrow O(g(n))$

entonces $T_1(n) + T_2(n) \rightarrow O(\max\{ f(n), g(n) \})$

- **Regla del producto:** si una parte se ejecuta dentro de otra.

Si $T_1(n) \rightarrow O(f(n))$

y $T_2(n) \rightarrow O(g(n))$

entonces $T_1(n) * T_2(n) \rightarrow O(f(n) * g(n))$

Cálculo de la eficiencia

Universidad de Alcalá

- **Ejemplo:**

Tenemos un algoritmo constituido por 3 etapas, en el que cada una de ellas puede ser un fragmento arbitrario de algoritmo con bucles y ramas. Supongamos sus tiempos respectivos $O(n^2)$, $O(n^3)$ y $O(n*\log(n))$.

- El tiempo de ejecución de las dos primeras etapas ejecutadas secuencialmente es $O(\max\{n^2, n^3\})$, es decir $O(n^3)$.
- El tiempo de ejecución de las tres juntas es $O(\max\{n^2, n^3, n*\log(n)\})$, es decir $O(n^3)$.

Cálculo de la eficiencia

Universidad de Alcalá

- **Operaciones elementales:**

Operaciones aritméticas:	`+, -, *, /, %, //`
Operaciones lógicas:	`and, or, not`
Operaciones de comparación:	`<, <=, >, >=, ==, !=`
Entrada / Salida:	`read, write`
Asignación de variables:	`=`
Devolución de valores:	`return`

Las operaciones elementales tienen un coste unitario:

$$T(n) \in O(1)$$

Cálculo de la eficiencia

Universidad de Alcalá

- **Condicional:**

```
if <cond>:
    bloqueA
else:
    bloqueB
```

```
T(cond) + Max{T(bloqueA), T(bloqueB)}
```

- T(cond): tiempo de evaluar la condición.

Instrucción	Coste
`if a < 100:`	`T(cond) = 1`
`    a *= 4`	`T(bloqueA) = 2`
`else:`	
`    a = 100`	`T(bloqueB) = 1`

$$T(n) = 3$$

Cálculo de la eficiencia

Universidad de Alcalá

- **Bucle while:**

```
while <cond>:
    bloque
```

$$T(cond) + \sum_{cond=false} \{T(bloque) + T(cond)\}$$

- T(cond): tiempo de evaluar la condición.

Instrucción	Coste
`while a < 100:`	T(cond) = 1
`    a *= 4`	T(bloque) = 2

El tiempo depende del número de vueltas (4 vueltas):

$$T(n) = 1 + 4 * 3 = 13$$

Cálculo de la eficiencia

Universidad de Alcalá

- **Bucle for:**

```
for <var> in range(n):
    bloque
```

- T(ini): tiempo de inicializar el bucle y asignar el primer valor.
- T(var): tiempo de asignar el siguiente valor.

$$T(ini) + \sum_{var=0}^{var=n-1} \{T(bloque) + T(var)\}$$

Instrucción	Coste
`for i in range(100):`	`T(ini) = T(var) = 1`
`    a *= 4`	`T(bloque) = 2`

El tiempo depende del número de vueltas (100 vueltas):

$$T(n) = 1 + 100 * 3 = 301$$

Cálculo de la eficiencia

Universidad de Alcalá

- **Ejemplo:**

Instrucción	Coste
`if a % 2 == 0:`	`T(cond) = 2`

`    for i in range(10 * 3 + n):`	`T(bloqueA) = T(for) # 30 + n vueltas`
	`T(ini) = 3, T(var) = 1`
	`T(for) = 3 + (30 + n) * 5 = 153 + 5n`

`        a *= 2`	`T(bloque) = 4`
`        a -= 1`	
`else:`	
`    while a < 100:`	`T(bloqueB) = T(while) = 13 # Ejemplo while`
`        a *= 4`	

$$T(n) = 2 + Max\{5n + 153, 13\} = 5n + 155 \in \mathbf{O(n)}$$

Cálculo de la eficiencia

Universidad de Alcalá

- **Recursividad:**

Las matemáticas necesarias para analizar algoritmos recursivos son las relaciones de recurrencia, también llamadas ecuaciones en diferencias o simplemente **recurrencias:**

```
def factorial(n :int) -> int:
    """ Calcula el factorial de un número n """
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
```

$$T(n) = \begin{cases} 2 & \text{si } n = 0 \\ 4 + T(n-1) & \text{si } n > 0 \end{cases}$$

La resolución de recurrencias consiste en proporcionar fórmulas no recursivas equivalentes: $T(n) = 4n + 2 \in O(n)$

Veremos varias formas de resolverlas.

Cálculo de la eficiencia

Universidad de Alcalá

- **Complejidad algorítmica de algunas recurrencias simples**

Ecuaciones	Orden
$T(n) = \begin{cases} k & \text{caso base} \\ T(n-1) + k' & \text{caso recursivo} \end{cases}$	$O(n)$
$T(n) = \begin{cases} k & \text{caso base} \\ T(n-1) + n & \text{caso recursivo} \end{cases}$	$O(n^2)$
$T(n) = \begin{cases} k & \text{caso base} \\ T(n/2) + k' & \text{caso recursivo} \end{cases}$	$O(\log(n))$
$T(n) = \begin{cases} k & \text{caso base} \\ 2T(n-1) + k' & \text{caso recursivo} \end{cases}$	$O(2^n)$
$T(n) = \begin{cases} k & \text{caso base} \\ 2T(n/2) + n & \text{caso recursivo} \end{cases}$	$O(n \log(n))$

Cálculo de la eficiencia

Universidad de Alcalá

- **Teorema maestro para el cálculo del coste**

$$T_1(n) = \begin{cases} f(n) & \text{si } 0 \le n < c \\ a \cdot T_1(n-c) + b \cdot n^k & \text{si } c \le n \end{cases} \Rightarrow T_1(n) \in \begin{cases} \Theta(n^k) & \text{si } a < 1 \\ \Theta(n^{k+1}) & \text{si } a = 1 \\ \Theta(a^{n/c}) & \text{si } a > 1 \end{cases}$$

$$T_2(n) = \begin{cases} g(n) & \text{si } 0 \le n < c \\ a \cdot T_2(n/c) + b \cdot n^k & \text{si } c \le n \end{cases} \Rightarrow T_2(n) \in \begin{cases} \Theta(n^k) & \text{si } a < c^k \\ \Theta(n^k \cdot \log n) & \text{si } a = c^k \\ \Theta(n^{\log_c a}) & \text{si } a > c^k \end{cases}$$

Volviendo al ejemplo de la función factorial:

$$\mathrm{T(n)} = \begin{cases} 2 \\ \mathrm{T(n-1)} + 4 \end{cases} \quad \mathrm{a = 1, c = 1, b = 4, k = 0} \quad \in \quad \mathrm{O(n)}$$

Cálculo de la eficiencia

Universidad de Alcalá

- **Expansión de recurrencias**

Un método para calcular la solución de recurrencias sencillas consiste en expandir la recurrencia hasta llegar al caso base.

$$T(n) = \begin{cases} 2 & \text{si } n = 0 \\ 4 + T(n-1) & \text{si } n > 0 \end{cases}$$

$$\begin{aligned} T(n) &= 4 + T(n-1) \\ &= 4 + 4 + T(n-2) && = 4 * 2 + T(n-2) \\ &= 4 + 4 + 4 + T(n-3) && = 4 * 3 + T(n-3) \\ &\dots \\ &= 4i + T(n-i) \end{aligned}$$

Pensamos cuanto vale i para llegar al caso base $T(0)$: $i=n$

Sustituyendo:

$$T(n) = 4n + T(0) = 4n + 2 \qquad \in O(n)$$

Cálculo de la eficiencia

Universidad de Alcalá

- **Método general para resolver relaciones de recurrencia**

Para calcular la solución de relaciones de recurrencia de forma general, se utiliza el **método de la ecuación característica**, que consiste en encontrar las raíces de la ecuación matemática que representa el uso de recursos de un método recursivo.

Pasos:

1. Calcular la relación de recurrencia.
2. Transformar la función a un polinomio.
3. Resolver la ecuación característica.

Ecuación característica

Universidad de Alcalá

- **Calcular la relación de recurrencia:**

Como hemos visto en apartados anteriores, para calcular la relación de recurrencia debemos analizar el código del algoritmo. Por ejemplo, para el algoritmo de Fibonacci, la recurrencia es:

```
def fibonacci(n :int) -> int:
    """ Calcula el n-ésimo término de la sucesión de fibonacci """
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

$$T(n) = \begin{cases} 2 & \text{si } n = 0 \\ 2 & \text{si } n = 1 \\ 3 + T(n-1) + T(n-2) & \text{si } n > 1 \end{cases}$$

Ecuación característica

Universidad de Alcalá

- **Transformar la función a un polinomio:**

Se trata de equiparar el caso recursivo con la siguiente recurrencia genérica:

$$a_0 T(n) + a_1 T(n-1) + \ldots + a_k T(n-k) = b^n p^d(n)$$

La parte de la izquierda es la llamada **ecuación homogénea** y la de la derecha es la **ecuación particular**, siendo $p(n)$ un polinomio de cierto grado $d \geq 0$. Para resolverla se hace uso de la siguiente **ecuación característica**, donde el primer paréntesis resuelve la parte homogénea y el segundo la particular:

$$(a_0 x^k + a_0 x^{k-1} + \cdots + a_k)(x-b)^{d+1} = 0$$

Ecuación característica

Universidad de Alcalá

- **Recurrencia homogénea:**

Tiene la forma:

$$a_0 T(n) + a_1 T(n-1) + \ldots + a_k T(n-k) = 0$$

Si consideramos que $T(n) = x^n$ y sustituimos en la ecuación anterior, obtenemos:

$$a_0 x^n + a_0 x^{n-1} + \cdots + a_k x^{n-k} = 0$$

Esta ecuación se satisface si:

$$p(x) = a_0 x^k + a_0 x^{k-1} + \cdots + a_k = 0$$

Ecuación característica

Universidad de Alcalá

- **Solución de la recurrencia homogénea (sin raíces múltiples):**

Por teorema fundamental del álgebra, todo polinomio $p(x)$ de grado k tiene k raíces (reales o complejas), por lo que $p(x)$ se puede factorizar como: $p(x) = \prod_{i=1}^{k}(x - r_i)$

De la factorización, se concluye que:

- $x = r_i$ es solución de la ecuación característica
- r_i^n es una solución de la recurrencia

Dado que toda combinación lineal de soluciones es también una solución, se concluye que toda solución de recurrencia $T(n)$ (sin raíces múltiples) tiene la forma:

$$T(n) = \sum_{i=1}^{k} c_i r_i^n$$

Ecuación característica

Universidad de Alcalá

- **Fibonacci:**

Volviendo al ejemplo de Fibonacci, si solo consideramos la parte homogénea, su recurrencia sería:

$$T(n) - T(n-1) - T(n-2) = 0$$

Y su polinomio: $p(x) = x^2 - x - 1 = 0$

Donde $k=2$, $a_0=1$, $a_1=-1$ y $a_2=-1$

Resolvemos las raíces del polinomio:

$$r_1 = \frac{1+\sqrt{5}}{2} \qquad r_2 = \frac{1-\sqrt{5}}{2}$$

Ecuación característica

Universidad de Alcalá

- **Fibonacci:**

Solución de la recurrencia homogénea (sin raíces múltiples):

$$T(n) = c_1 r_1^n + c_2 r_2^n$$

Sustituyendo r_1 y r_2 en la ecuación podemos conocer la complejidad asintótica del algoritmo (el resto son constantes).

Si queremos obtener el valor de las constantes c_1 y c_2, habría que evaluar el valor de la recurrencia $T(n)$ en sus casos base y resolver el sistema de ecuaciones:

$$c_1 + c_2 = 2$$
$$c_1 r_1 + c_2 r_2 = 2$$

Ecuación característica

Universidad de Alcalá

- **Fibonacci:**

Podemos concluir que:

$$T(n) = c_1 \left(\frac{1+\sqrt{5}}{2}\right)^n - c_2 \left(\frac{1-\sqrt{5}}{2}\right)^n$$

El segundo término tiende a 0 según $n \to \infty$, por tanto:

$$T(n) \in O\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right) \approx O(1{,}618^n) < O(2^n)$$