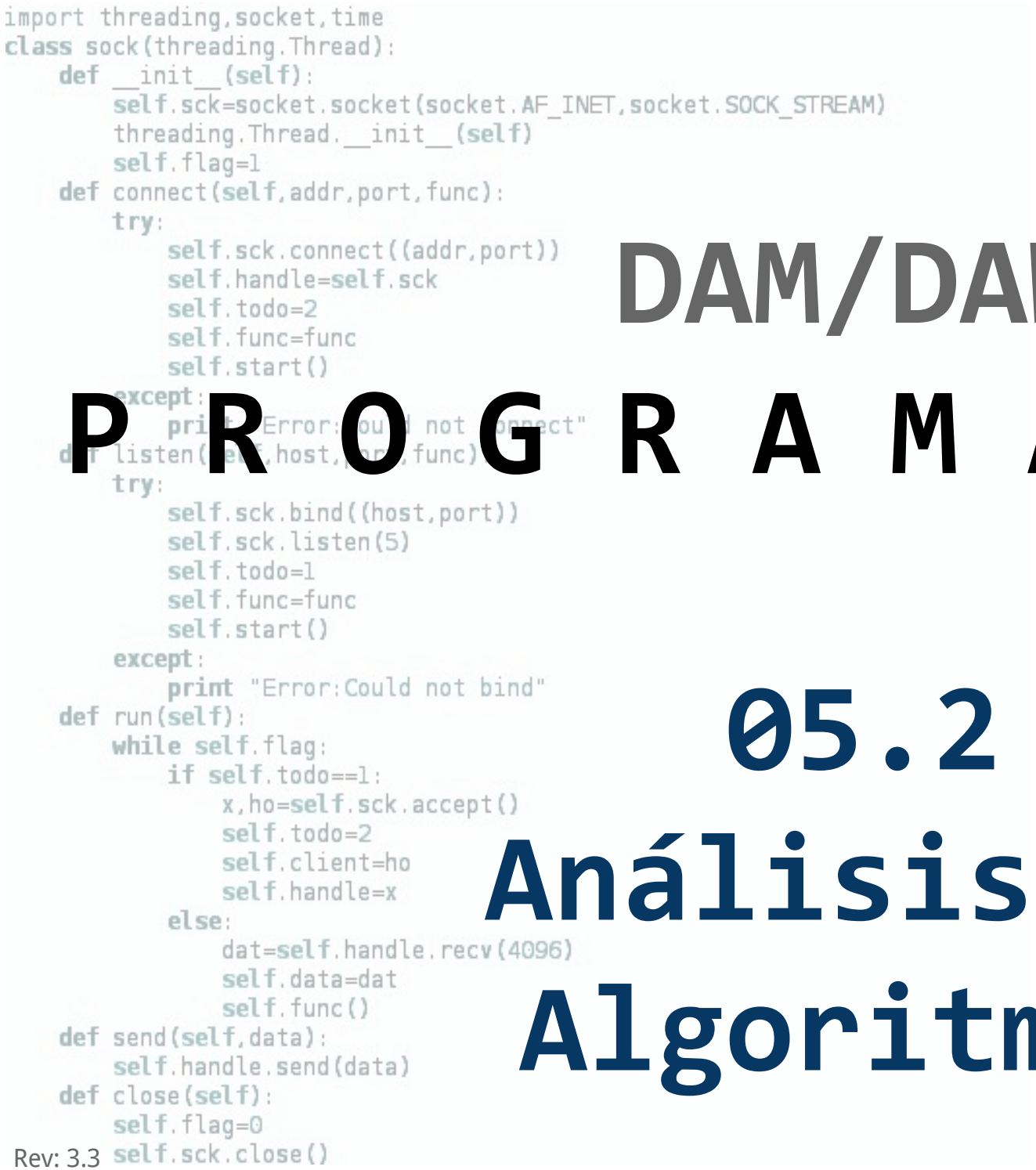


DAM/DAW

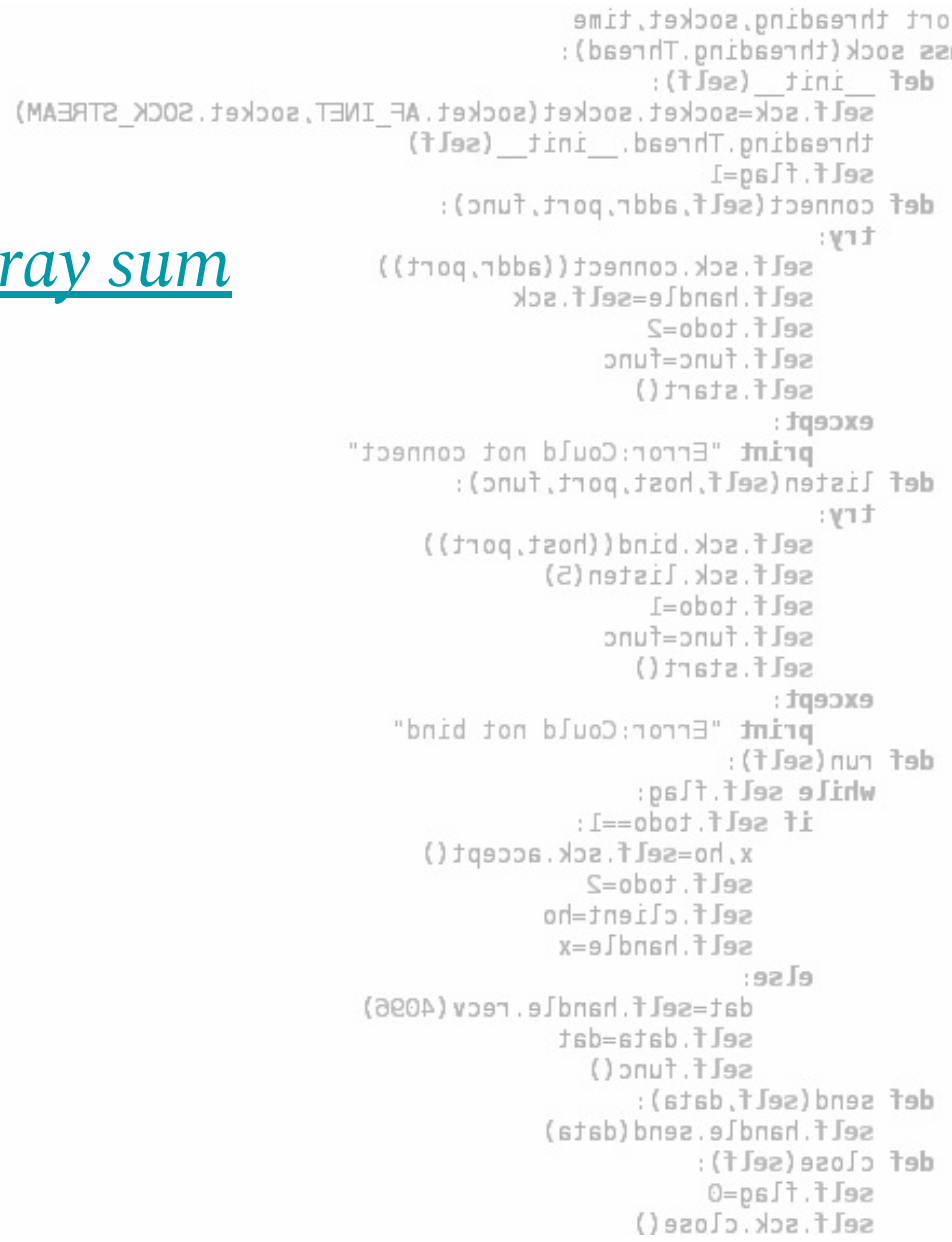
PROGRAMACIÓN

05.2 Análisis de Algoritmos

Rev: 3.3

fran@iessanclemente.net

Indice



Introducción (I)

"En casi todo cómputo, son posibles gran variedad de configuraciones para las sucesiones de procesos, y varias consideraciones deben influir en la selección entre ellas según los propósitos de un motor de cálculo.

Un objetivo esencial es escoger la configuración que tienda a minimizar el tiempo necesario para completar el cálculo." (*)

(*) Reflexiones de Ada Lovelace en 1843 recogidas en su trabajo sobre la máquina analítica mecánica de Charles Babbage

Introducción (y II)

- En computación, como en muchas otras disciplinas, para un mismo problema, podemos encontrar diferentes soluciones (*algoritmos*)
- Necesitamos un mecanismo que nos permita *comparar* la *eficiencia* de las diferentes soluciones propuestas
- Entendemos la eficiencia de un algoritmo como una *medida* de los recursos que "consume" en su ejecución (tiempo, espacio, energía,...)
- Las métricas de eficiencia más empleadas para la comparación de algoritmos son:
  - *Complejidad temporal*: cuánto tarda en completarse su ejecución
  - *Complejidad espacial*: cuánta memoria (general[te]. *RAM*) consume
- Para lograr una *eficiencia máxima* deberemos *minimizar* el uso de recursos

Un ejemplo: *maximum subarray sum* (I)

- Vamos a analizar un pequeño problema: el cálculo del valor máximo de la suma de cualquier subconjunto de elementos de un *array*.
- Expresado de otra manera, consistiría en buscar el *subarray* tal que la suma de sus elementos fuera la máxima con respecto a cualquier otro *subarray* que extraigamos. Por ejemplo:
- Por ejemplo:

```
array = [ -2, 1, -3, 4, -1, 2, 1, -5, 4 ]

max_subarray_sum(array) = (6, (4, -1, 2, 1))
```

- La solución de este tipo de problemas es de gran interés en campos como:
  - Genética: por ejemplo, identificación de secuencias de proteínas
  - Visión por ordenador: p.ej., detección área más brillante de un *bitmap*

Un ejemplo: *maximum subarray sum* (I)

- Vamos a plantear tres soluciones diferentes para nuestro problema, para luego poder comparar su eficiencia
- *Algoritmo 1:* La solución más inmediata a este problema consistiría en iterar por todos los *subarrays* posibles, calculando la suma de cada uno de ellos, y guardando el máximo valor encontrado
- *Algoritmo 2:* Una mejora del algoritmo anterior consiste en calcular la suma del *subarray* al mismo tiempo que desplazamos el límite por la derecha, elminando así la necesidad del bucle interno
- *Algoritmo 3*: (*Kadane's algorithm*) Es una solución que permite resolver el problema recorriendo una única vez los elementos del *array*. La idea es calcular, para cada posición del *array*, la suma máxima de un subarray que terminara en esa posición (máximo 0 si los elementos son negativos)

Un ejemplo: *maximum subarray sum* (I)

- *Pseudocódigo Algoritmo 1*

```
function max_subarr_sum_1(ar[])
    best ← 0  /* maximum sum */
    n ← length(ar[])
    for i ← 0 to (n-1) do
        for j ← i to (n-1) do
            sum ← 0
            for k ← i to j do
                sum ← sum + ar[k]
            end for
            best ← max(best, sum)
        end for
    end for
    return best
end function
```

Un ejemplo: *maximum subarray sum* (II)

- *Pseudocódigo Algoritmo 2*

```
function max_subarr_sum_2(ar[])
    best ← 0  /* maximum sum */
    n ← length(ar[])
    for i ← 0 to (n-1) do
        sum ← 0
        for j ← i to (n-1) do
            sum ← sum + ar[j]
            best ← max(best, sum)
        end for
    end for
    return best
end function
```

Un ejemplo: *maximum subarray sum* (III)

- *Pseudocódigo Algoritmo 3*

```
function max_subarr_sum_3(ar[])
    best ← 0  /* maximum sum */
    sum ← 0
    n ← length(ar[])
    for i ← 0 to (n-1) do
        sum ← max(0, sum + ar[i])
        best ← max(best, sum)
    end for
    return best
end function
```

Un ejemplo: *maximum subarray sum* (y IV)

- Es interesante estudiar la eficiencia de los algoritmos en la práctica
- Para ello, podemos calcular el tiempo transcurrido entre el inicio y finalización de la ejecución de cada algoritmo (*System.currentTimeMillis( )*) para un mismo conjunto de datos de entrada.
- La siguiente tabla muestra tiempos de ejecución de los tres algoritmos con diferentes tamaños de datos empleando un computador actual.

tamaño array (n)	Algoritmo 1	Algoritmo 2	Algoritmo 3
10^2	0.00 s	0.00 s	0.00 s
10^3	0.12 s	0.00 s	0.00 s
10^4	> 10 s	0.03 s	0.00 s
10^5	> 10 s	5.16 s	0.00 s
10^6	> 10 s	> 10 s	0.00 s
10^7	> 10 s	> 10 s	0.01 s

Otro ejemplo: *Fibonacci* (I)

- La **sucesión de Fibonacci** es la sucesión infinita de números naturales:

  [0,] 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144,...

  donde cada número de la serie es resultado de sumar los dos precedentes

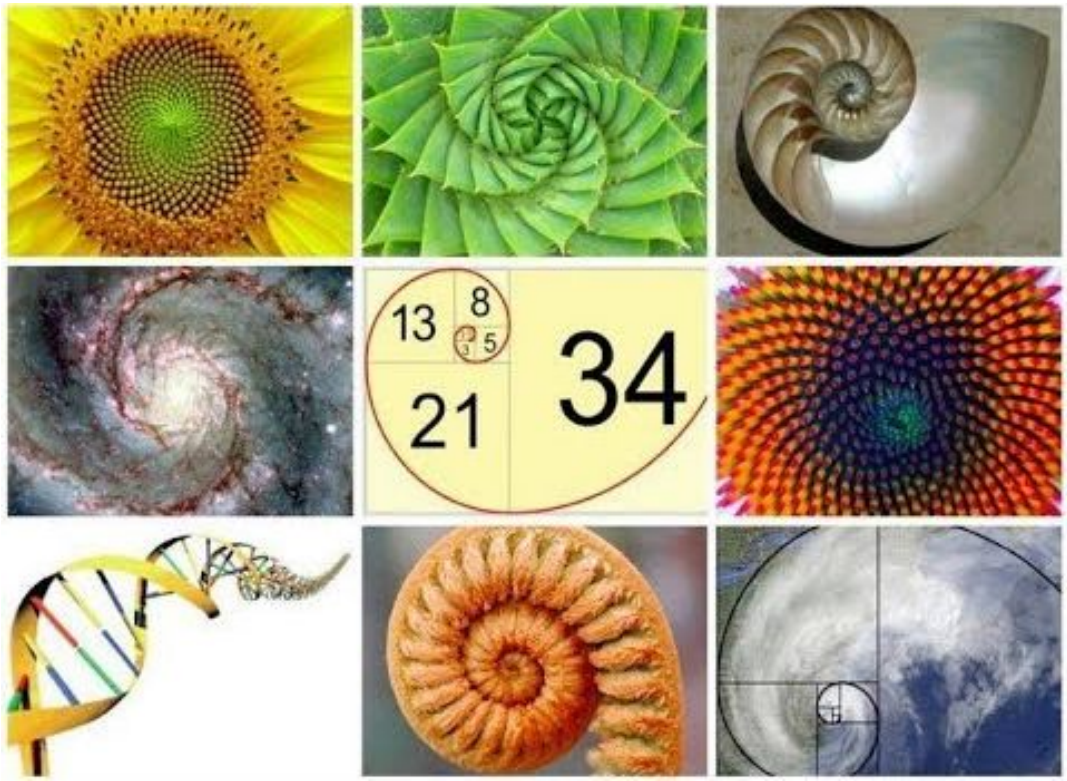
- Fue descrita en Europa por *Leonardo de Pisa*, conocido como *Fibonacci*, matemático italiano del siglo XIII, en su obra (*Liber Abaci*, 1202), aunque su origen se remonta a estudios del 200 A.C. de un matemático indio, *Pingala*, sobre patrones de poesía en sánscrito.

- Tiene numerosas aplicaciones en computación, matemáticas y teoría de juegos. También aparece en configuraciones biológicas, por ejemplo: en las ramificaciones de los árboles, en la configuración de las piñas de las coníferas, en la reproducción de los conejos y en cómo el ADN codifica el crecimiento de formas orgánicas complejas. También se encuentra en la estructura espiral del caparazón de algunos moluscos, como el nautilus.

Otro ejemplo: *Fibonacci* (II)

- La razón de crecimiento de la *espiral de Fibonacci*, formada a partir de cuadrados cuyos lados siguen la serie, aproxima la *razón aúrea*

$$\varphi = \frac{1+\sqrt{5}}{2} = 1.618\,033\,988\,749\,894\ldots$$



Otro ejemplo: *Fibonacci* (III)

- Existen diferentes soluciones del algoritmo que nos permiten determinar el valor del término *n-ésimo* de la serie. Vamos a ver dos:

- *Algoritmo 1 (recursivo):*

```
function fibonacci_1(n)
    if n<2 then
        return n
    else
        return fibonacci_1(n-1) + fibonacci_1(n-2)
end function
```

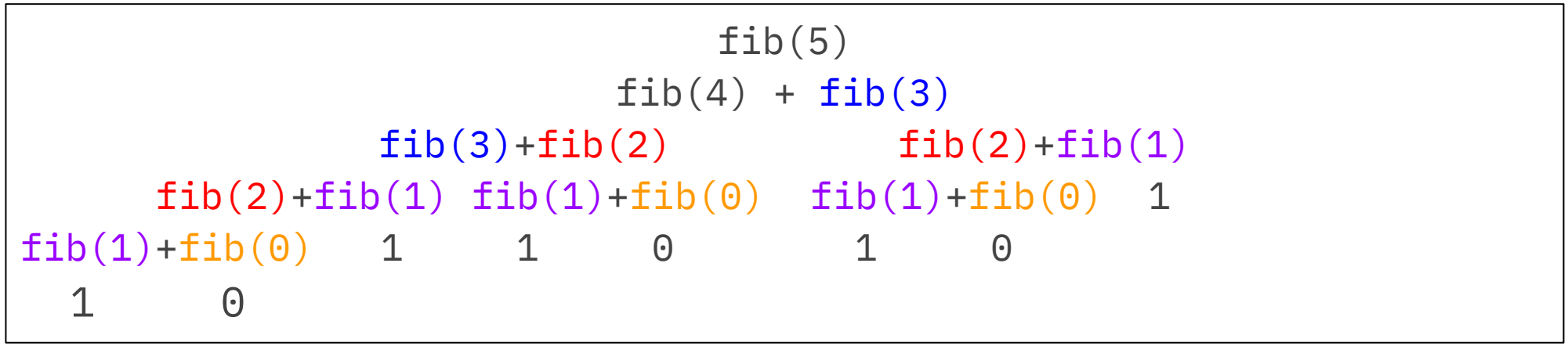
Otro ejemplo: *Fibonacci* (IV)

- *Algoritmo 2 (iterativo):*

```
function fibonacci_2(n)
    if n<2 then
        return n
    a ← 0
    b ← 1
    fib ← 0
    for i ← 1 to (n-1) do
        fib ← a + b
        a ← b
        b ← fib
    end for
    return fib
end function
```

Otro ejemplo: *Fibonacci* (y V)

- Si comparamos los tiempos de ejecución de ambos algoritmos, podemos observar como, aunque el primero de ellos *(recursivo)* tiene una formulación más simple (y más "elegante"), su rendimiento empieza a decaer considerablemente a partir del término 45. Sin embargo, este nivel de degradación no se aprecia al ejecutar el segundo algoritmo.
- La razón de ello, es fácil de "descubrir" si trazamos el árbol de ejecución del algoritmo recursivo. Podemos ver como continuamente estamos repitiendo cálculos ya realizados, creciendo de forma exponencial.



Herramientas de Análisis (I)

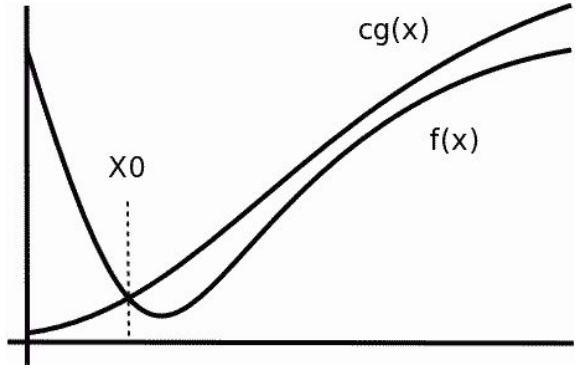
- Aunque analizar tiempos de ejecución es útil, presenta limitaciones:
  - Sólo lo podemos realizar sobre un conjunto reducido de entradas
  - Debemos realizar los experimentos sobre el mismo *hardware/software*
  - Tenemos que implementar el algoritmo
- Idealmente, necesitamos una herramienta de análisis que nos permita analizar algoritmos sin necesidad de experimentación de forma que:
  - Tenga en cuenta todas las posibles entradas
  - De forma independiente al entorno hardware/software
  - Se pueda realizar sobre una descripción de alto nivel del algoritmo sin necesidad de implementarlo

Herramientas de Análisis (II)

- Para analizar algoritmos utilizaremos una metodología que asociará a cada algoritmo una función *f(n)* que caracterizará su tiempo de ejecución en base al tamaño de la entrada (*n*)
- Partiendo del *pseudo-código* de nuestro algoritmo, realizaremos un conteo de las *operaciones primitivas* (asignaciones, comparaciones, llamadas a funciones, operaciones aritméticas,...) que se realizan en función del tamaño de la entrada. Asumiremos que todas esas operaciones primitivas tienen el mismo coste temporal.
- Se evalúa el tiempo de ejecución del algoritmo para el peor escenario posible. Por ejemplo, si estamos evaluando un algoritmo de búsqueda que recorre una lista de elementos, supondremos que tiene que recorrer toda la lista, obviando cualquier otro caso
- Sólo se calculan órdenes de magnitud, no valores exactos

Notación *Big-Oh* (I)

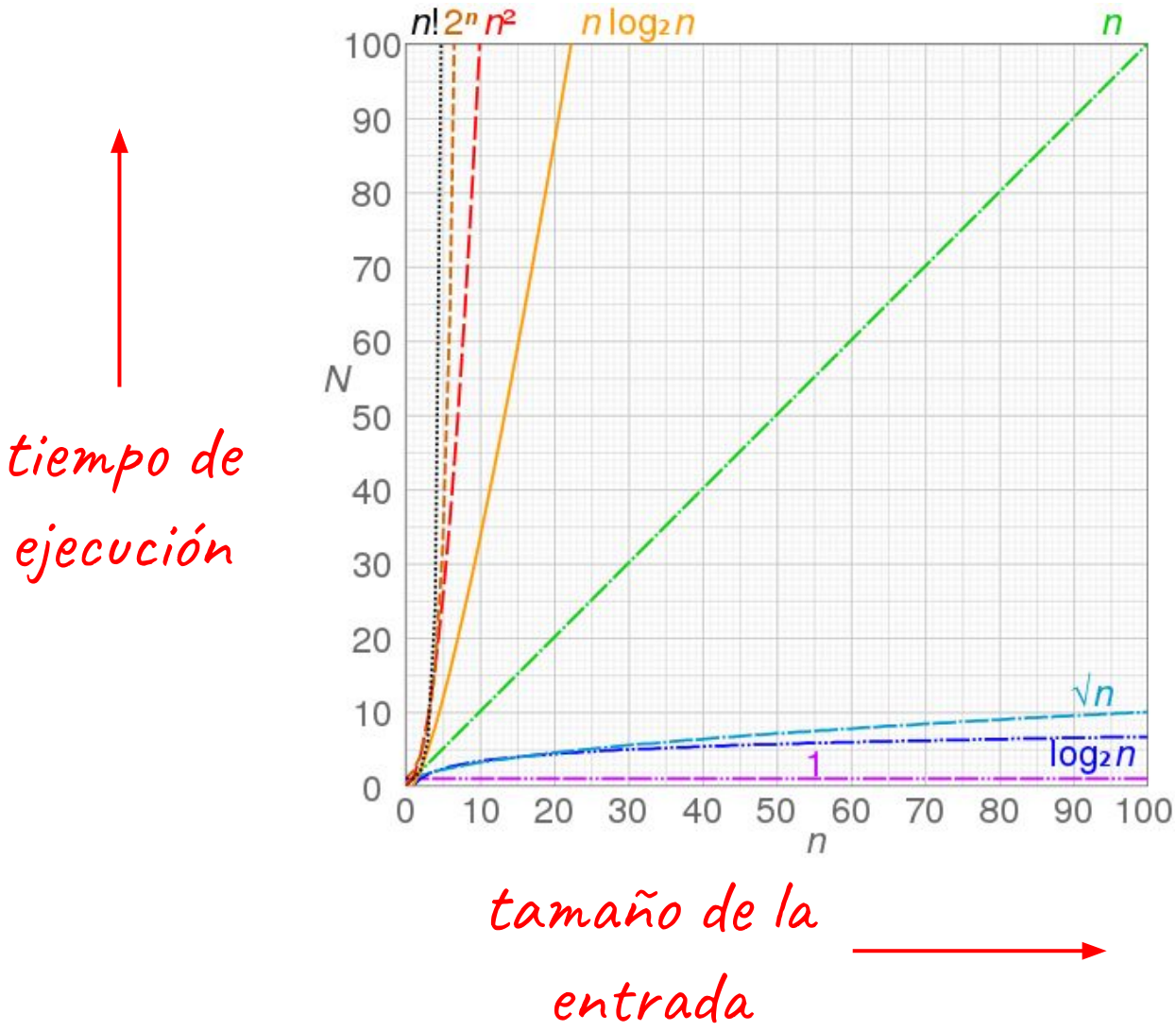
- En análisis de algoritmos, la *cota superior asintótica* es una función ($c \cdot g(x)$) que sirve de cota superior de otra función ($f(x)$) cuando el argumento (x) tiende a infinito



- Se utiliza la *notación de Landau*, $O(g(x))$, coloquialmente *Big-Oh*, para referirse a todas las funciones acotadas superiormente por $g(x)$.
- Al analizar la *complejidad temporal* de un algoritmo, buscaremos la cota superior que mejor aproxime su tiempo de ejecución en función del tamaño de la entrada (cómo se comporta cuando la entrada crece)
- Algunas de las funciones de cota más empleadas son, en orden creciente: $O(1)$ (*constante*), $O(\log n)$ (*logarítmica*), $O(\sqrt{n})$ (raíz), $O(n)$ (*lineal*), $O(n \cdot \log n)$, $O(n^2)$ (*cuadrática*), $O(n^3)$ (*cúbica*), $O(2^n)$ (*exponencial*), $O(n!)$ (*factorial*)

Notación *Big-Oh* (II)

- *Comparación gráfica de complejidades computacionales*



Notación *Big-Oh* (III)

❖ Reglas generales de aplicación

- A la hora de buscar la función de cota (*límite asintótico*) que mejor define el comportamiento de nuestro algoritmo, usaremos las siguientes reglas:

1. Orden de magnitud:

Un análisis de complejidad temporal no nos indica el número exacto de veces que se ejecuta nuestro código. Sólo nos importa el *orden de magnitud*

Así, la siguiente función, que intercambia dos posiciones de un *array*:

```
function swap(ar[], i, j)
    temp ← ar[i]
    ar[i] ← ar[j]
    ar[j] ← temp
end function
```

Aunque se ejecutan tres sentencias, diremos que es *O(1)*, pues no depende del tamaño de la entrada (*array*). Se ejecuta en tiempo constante

Notación *Big-Oh* (IV)

2. Bucles:

Una razón habitual por la que un algoritmo es lento es porque contiene numerosos bucles que recorren la entrada. Cuantos más bucles anidados contenga, más lento será. En general, si k es el número de bucles anidados, su complejidad temporal será $O(n^k)$

- Este bucle tiene complejidad temporal $O(n)$:

```
for i ← 1 to n do
    // code...
end for
```

- Y la complejidad de éste es $O(n^2)$:

```
for i ← 1 to n do
    for j ← 1 to n do
        // code...
    end for
end for
```

Notación *Big-Oh* (V)

- En relación con la *regla 1*, debemos fijarnos sólo en el *orden de magnitud*

- En los siguiente ejemplos, el código dentro del bucle se ejecuta $(3 \cdot n)$, $(n + 5)$ y $(n/2)$ veces, pero la complejidad siempre será $O(n)$

```
for i ← 1 to 3*n do
    // code...
end for
```

```
for i ← 1 to n+5 do
    // code...
end for
```

```
for i ← 1 to n step 2 do
    // code...
end for
```

Notación *Big-Oh* (VI)

3. Fases:

Cuando un algoritmo consiste en la ejecución de fases consecutivas, la complejidad temporal la determinará la mayor de las diferentes fases. La razón de esto es que la fase más lenta es generalmente el cuello de botella (*bottleneck*) del algoritmo.

- Por ejemplo, el siguiente código consta de tres fases con complejidades $O(n)$, $O(n^2)$ y $O(n)$. La complejidad total será $O(n^2)$

```
for i ← 1 to n do
    // code...
end for
for i ← 1 to n do
    for j ← 1 to n do
        // code...
    end for
end for
for i ← 1 to n do
    // code...
end for
```

Notación *Big-Oh* (VII)

4. Varias variables:

En ocasiones, la complejidad temporal dependerá de varios factores. En este caso, la fórmula contendrá varias variables.

- Por ejemplo, considera un caso en el que por cada elemento de un array, tuviéramos que recorrer otro de tamaño diferente (por ejemplo, para hacer una búsqueda secuencial). La complejidad temporal en este caso sería $O(n \cdot m)$

```
for i ← 1 to n do
    for j ← 1 to m do
        // code...
    end for
end for
```

Notación *Big-Oh* (VIII)

5. Recursión:

La complejidad temporal de una función recursiva depende del número de veces que llamamos a la función y de la complejidad de cada llamada. La complejidad temporal total será el producto de ambos valores

- Por ejemplo, considera la siguiente función:

```
function f(n)
    if (n==1) return
    f(n-1)
end function
```

La invocación de *f(n)* generará *n* llamadas a la función, y la complejidad de cada llamada es *O(1)*, por lo que la complejidad temporal total es *O(n)*

Notación *Big-Oh* (y IX)

- Sin embargo, si nos fijamos en el siguiente ejemplo:

```
function f(n)
    if (n=1) return
    f(n-1)
    f(n-1)
end function
```

Cada llamada a *f(n)* generará dos nuevas llamadas (excepto para *n=1*)

Si construimos la tabla de llamadas para una simple invocación de *f(n)*:

Llamada a la función	Número de llamadas
f(n)	1
f(n-1)	2
f(n-2)	4
...	...
f(1)	2^{n-1}

La complejidad temporal será:

$1 + 2 + 4 + \ldots + 2^{n-1} =$

$2^{n-1} =$

$O(2^n)$

Estimando la eficiencia (I)

- Calculando la complejidad temporal de un algoritmo es posible estimar, antes de implementarlo, si va a ser suficientemente eficiente para resolver un problema dentro de los parámetros temporales establecidos

- A modo de ejemplo, supón que tenemos un límite temporal de 1 segundo para nuestro problema (habitual en competiciones) y que el tamaño de la entrada es de $n = 10^5$. Si la complejidad del algoritmo es $O(n^2)$, necesitaría realizar sobre $(10^5)^2 = 10^{10}$ operaciones para resolverlo. Suponiendo que un computador moderno realiza del orden de 10^8 operaciones por segundo, tardaría más de 10 segundos!!!

- Por otro lado, tal como muestra la tabla de la página siguiente, a partir del tamaño de la entrada, podemos *intuir* la complejidad temporal máxima del algoritmo que solucione el problema.

Estimando la eficiencia (y II)

- *Complejidad temporal máxima frente al tamaño de la entrada (1 seg max.)*

Tamaño de la entrada	Complejidad temporal (1s)
$n < 10$	$O(n!)$
$n < 20$	$O(2^n)$
$n < 500$	$O(n^3)$
$n < 5000$	$O(n^2)$
$n < 10^6$	$O(n \cdot \log n)$ ó $O(n)$
$n > 10^6$	$O(1)$ ó $O(\log n)$

- En todo caso, es importante recordar que estamos hablando de estimaciones de la eficiencia, y que quedan ocultos los *factores constantes*. Por ejemplo, un algoritmo $O(n)$ podría realizar $(n/2)$ ó $5 \cdot n$ operaciones

Ejercicio

- *Determina la complejidad temporal y tamaño máximo de la entrada (t<10s)*

Problema - Algoritmo	Complejidad temporal	N máx (t<10s)
Max Subarray Sum - Algoritmo 1		
Max Subarray Sum - Algoritmo 2		
Max Subarray Sum - Algoritmo 3		
Fibonacci - Algorimo 1 (recursivo)		
Fibonacci - Algorimo 2 (iterativo)		

RESPONDE A ESTA PREGUNTA:

¿Se corresponden tus resultados del análisis de complejidad temporal con los resultados experimentales observados?