

- Diremos que un algoritmo es **determinista** si, para cada "entrada concreta" (**instancia**), proporciona una salida correcta. En este caso, diremos que el algoritmo **resuelve** el problema computacional.

Un algoritmo no determinista puede no resolver el problema para algunas instancias. Contrario a lo que se puede pensar, los algoritmos no deterministas pueden ser útiles algunas veces, si el margen de *error* está controlado. Por ejemplo: los algoritmos **probabilísticos** (ej. *tests de primalidad*), y los algoritmos **heurísticos** (ej. *Travelling Salesman Problem*), etc.

- Un algoritmo se puede "detallar" mediante la utilización de un lenguaje de programación de alto nivel. Pero hay una torre de Babel de lenguajes de programación, por eso muchos libros prefieren utilizar:

PSEUDOCÓDIGO

Conjunto de reglas informales para la representación de algoritmos, utilizando una combinación de instrucciones escritas en un lenguaje natural y de órdenes de control estructuradas.

Probablemente el algoritmo más antiguo y uno de los más elegantes de la historia es recursivo. Fue propuesto por Euclides para computar el máximo común divisor (**mcd**) de dos números n y m :

Calcula el resto de dividir el mayor de los dos números por el menor de ellos. Si el resto es cero, entonces el mcd es el menor de ambos números. Si el resto es distinto de cero, el mcd de n y m es el mcd de otro par de números, a saber, el formado por el menor de n y m , y por dicho resto.

programa.py

```
def mcd(n,m):  
    mayor = max(n,m)  
    menor = min(n,m)  
    resto = mayor % menor  
    if resto == 0:  
        return menor  
    else  
        return mcd(menor,resto)
```

programa.py

```
def mcd(n,m):  
    if n == 0:  
        return m  
    return mcd(m%n,n)
```




Resolver el problema de la torres de Hanoi con n discos que hemos de transferir de la aguja inicial a la aguja final requiere:

- resolver el problema de las torres de Hanoi con $n - 1$ discos, pasándolos de la aguja inicial a la aguja libre;
- mover el disco n de la aguja inicial a la aguja final; y
- resolver el problema de las torres de Hanoi con $n - 1$ discos de la aguja libre a la aguja final.

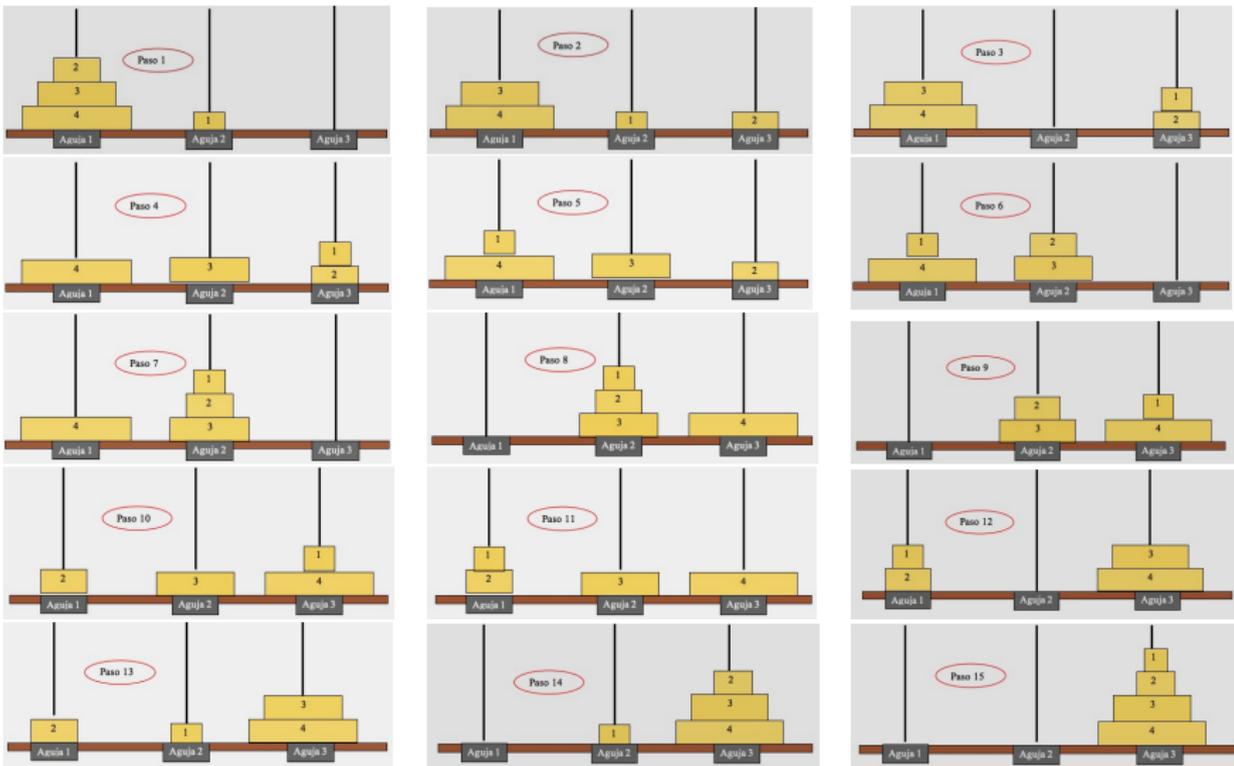
Para construir el programa, necesitamos los argumentos del procedimiento o función, a saber: el número de discos n , la aguja inicial, la libre y la final; a las que identificaremos con un número.

Tenemos todos los ingredientes para construir un programa en **Python**: la doble recursión, la condición trivial o de parada y los argumentos:

programa.py

```
def hanoi(n,inicial,final,libre):  
    if n == 1:  
        print ('mover disco superior de aguja', inicial,'a', final)  
    else:  
        hanoi(n-1,inicial,libre,final)  
        print ('mover disco superior de aguja', inicial,'a', final)  
        hanoi(n-1,libre,final,inicial)
```

```
>>> hanoi(2,1,3,2)  
mover disco superior de aguja 1 a 2  
mover disco superior de aguja 1 a 3  
mover disco superior de aguja 2 a 3
```

Un simulador del rompecabezas utilizando el modulo **tkinter** de **Python**:
<http://svn.python.org/projects/python/trunk/Demo/tkinter/guido/hanoi.py>

...y el mundo desaparecerá. ¿Pero cuándo?

Los sacerdotes del templo de Brahma deben pasar los 64 discos de la **Aguja 1** a la **Aguja 3**. ¿Podemos ayudarles con el programa de **Python** ?

- Es muy fácil observar (y demostrar) que el número de movimientos totales de discos es $2^n - 1$. De hecho, el disco i lo movemos exactamente 2^{n-i} veces.
- Si los sacerdotes cambiaran un disco de sitio cada segundo, aproximadamente unos 2^{25} movimientos por año, necesitarían unos 2^{39} años, es decir, unos $2^5 = 32$ veces la edad del universo 2^{34} .
- Los algoritmos recursivos son mucho más elegantes que los iterativos, pero suelen ser menos **eficientes**, pues cada llamada a función requiere tiempo de cálculo, espacio de memoria y gestión de pila.
- Todo algoritmo **recursivo** admite uno **iterativo**, otra cuestión es explicitarlo. En el caso de las Torres de Hanoi existen un par de métodos iterativos muy simples, pero la solución recursiva propuesta es óptima, es decir, no hay un algoritmo que necesite menos movimientos.
- ¿ Qué puede pasar si ejecutamos en una sesión de **Python** el programa `hanoi(64,1,3,2)` ?

programa.py

```
from time import process_time
from random import randint

def tiempo_lista(M,N):
    t_0 = process_time()
    L = []
    i = 0
    while i < 10**M:
        L.append(randint(0,10**N))
        i += 1
    t = process_time()
    return t - t_0

def tiempo_busqueda_lineal(N,M):
    t_0 = process_time()
    busqueda_lineal(randint(0,10**N),range(10**M))
    t = process_time()
    return t - t_0
```


Utilizaremos la funcion *sample* del modulo RANDOM

```
>>> from random import sample
>>> # Lista de 8 enteros " aleatorios" y menores que 100.
>>> L = sample(range(100),8)
>>> L
[74,79, 64, 55, 89, 35, 34, 73]
```

▶ PROGRAMA TIEMPO

programa.py

```
def tiempo_ordenar_burbuja(N,M):
    from time import process_time
    L = sample(range(10**N),10**M)
    t_0 = process_time()
    ordenar_burbuja(L)
    t = process_time()
    return t - t_0
```

programa.py

```
def tiempo_ordenar_insertando(N,M):
    from time import process_time
    L = sample(range(10**N),10**M)
    t_0 = process_time()
    ordenar_insertando(L)
    t = process_time()
    return t - t_0
```

```
>>> tiempo_ordenar_insertando(3,3)
0.09154700000000056
>>> tiempo_ordenar_burbuja(3,3)
0.18744800000000034
>>> tiempo_ordenar_insertando(4,4)
8.3046499999999995
>>> tiempo_ordenar_burbuja(4,4)
18.323788999999999
>>> tiempo_ordenar_insertando(5,5)
909.73051
>>> tiempo_ordenar_burbuja(5,5)
```

La última instancia de la sesión de Python fue abortada después de dos horas. De hecho, algunos autores sugieren no explicar el algoritmo de la burbuja !!

Mezclando dos pilas

El punto clave del método es, justamente, mezclar las dos sucesiones ordenadas en la última etapa.

Supongamos que tenemos dos pilas de cartas con la cara por arriba. Cada pila está ordenada tal que la carta más pequeña es la superior. Queremos mezclar las dos pilas para obtener una única pila ordenada.

- Elegimos la más pequeña entre de las dos cartas superiores de cada pila;
- la quitamos de esa pila y la ponemos en una nueva pila con la cara por abajo;
- repetimos este procedimiento hasta que no haya cartas en alguna de las pilas;
- entonces sólo tenemos que coger el resto de las cartas de la otra pila y ponerlas boca abajo en la nueva pila.

Programa *mezclando*

programa.py

```
def mezclando(izquierda, derecha):  
    ordenada = []  
    i, j = 0, 0  
    while i < len(izquierda) and j < len(derecha):  
        if izquierda[i] <= derecha[j]:  
            ordenada.append(izquierda[i])  
            i = i + 1  
        else:  
            ordenada.append(derecha[j])  
            j = j + 1  
    ordenada += izquierda[i:]  
    ordenada += derecha[j:]  
    return ordenada
```


En el prestigioso **The Big-White-Book** de los algoritmos: *Introduction to Algorithms*. MIT press. T. CORMEN, C. LEISERSON, R. RIVEST, C. STEIN, dicen:

A good algorithm is like a sharp knife -it does exactly what it is supposed to do with a minimum amount of applied effort. Using the wrong algorithm to solve a problem is like trying to cut a steak with a screw-driver: you may eventually get a digestible result, but you will expend considerably more effort than necessary, and the result is unlikely to be aesthetically pleasing.

Algorithms devised to solve the same problem often differ dramatically in their efficiency. These differences can be much more significant than difference between a personal computer and a supercomputer.

Algorithms, like computer hardware, are a **technology** . Total system performance depends on choosing efficient algorithms as much as on choosing fast hardware. Just as rapid advances are being made in other computer technologies, they are being made in algorithms as well.

El **tiempo de ejecución** de un algoritmo con una particular entrada es el número de operaciones primitivas o **etapas ejecutadas**.
Deberíamos definir el concepto de **etapa** independientemente de la máquina. De momento adoptamos lo siguiente:

Una cantidad constante es requerida para ejecutar cada línea del programa. Una línea puede tomar una cantidad constante distinta de otra línea, pero asumiremos que cada ejecución de la línea i -ésima necesita un tiempo c_i , donde c_i es constante (modelo **RAM**).

Analizamos el tiempo de ejecución del programa ▶ ordenar_insertando

def *ordenar_insertando*(L):

1. for *j* in range(1,len(L)):

2. *i* = *j* - 1

3. *temp* = L[*j*]

4. while *i* > -1 and L[*i*] > *temp*:

5. L[*i* + 1] = L[*i*]

6. *i* = *i* - 1

7. L[*i* + 1] = *temp*

8. return L

Coste **Veces**

c_1 $n - 1$

c_2 $n - 1$

c_3 $n - 1$

c_4 $\sum_{j=2}^n t_j$

c_5 $\sum_{j=2}^n (t_j - 1)$

c_6 $\sum_{j=2}^n (t_j - 1)$

c_7 $n - 1$

0 - - - - -

Notas

- n es la longitud de la lista.
- t_j es el número de veces que se testa la línea 4, en el bucle **while** para el valor j .
- $T(n)$ es el tiempo total de ejecución del algoritmo.

$T(n)$ es la suma del tiempo de ejecución de cada instrucción ejecutada; una instrucción que requiere un tiempo c_i y es ejecutada m veces contribuye con un total de $c_i m$ al tiempo de ejecución:

$$T(n) = (c_1 + c_2 + c_3 + c_7)(n - 1) + c_4 \sum_{j=2}^n t_j + (c_5 + c_6) \sum_{j=2}^n (t_j - 1).$$

En el algoritmo anterior hemos analizado el tiempo de ejecución del mejor y el peor caso. En la mayor parte de estas notas nos centraremos en determinar el tiempo de ejecución del caso peor de un algoritmo, es decir, el tiempo más grande para cualquier entrada de tamaño n . Aquí van algunas razones para esta decisión:

- 1 El tiempo de ejecución del **caso peor** es una cota superior. Así garantizamos que el algoritmo nunca costará más.
- 2 Para algunos algoritmos el caso peor es muy frecuente. Por ejemplo en problemas de búsqueda, a menudo ocurre que el dato a buscar no está en la lista.
- 3 El **caso promedio** es a menudo tal malo como el **caso peor**. Por ejemplo, supongamos que elegimos n números aleatorios. ¿Cuánto tiempo necesitaremos para determinar donde insertar en la sublista $L[1, \dots, j-1]$ el elemento $L[j]$? En media, la mitad de los elementos son menores y la otra mitad mayores que $L[j]$. Luego $t_j = \frac{j}{2}$, pero el tiempo de ejecución será, también, una función cuadrática en n , como el caso peor.

Por supuesto, en algunos algoritmos es interesante analizar el caso promedio o esperado. Pero tendremos que precisar que constituye el término promedio para el algoritmo en cuestión.

Supongamos que hemos analizado dos algoritmos con una entrada de tamaño n :

- El algoritmo **Alg. 1** requiere $100n + 5$ etapas.
- El algoritmo **Alg. 2** requiere $n^2 + n + 5$ etapas.

La siguiente tabla muestra el tiempo de ejecución de estos algoritmos para diferentes n :

Tamaño n	Tiempo Alg. 1	Tiempo Alg. 2
10	1005	105
100	10005	10105
1000	100005	1001005
10000	1000005	100010005

Para $n = 10$, **Alg. 2** es 10 veces más rápido que el **Alg. 1**. Pero para $n = 100$ son casi iguales, y para grandes valores n el **Alg. 1** es mucho más rápido. La razón fundamental es que para valores grandes de n , cualquier función cuadrática crece más rápida que una función lineal. El punto de cruce depende de los detalles del algoritmo y del hardware, por lo que suele ser ignorado para la complejidad computacional, pero eso no significa que se deba obviar.

La notación O mayúscula, $O(f)$, denota el conjunto de las funciones g que crecen a lo más tan rápido como f :

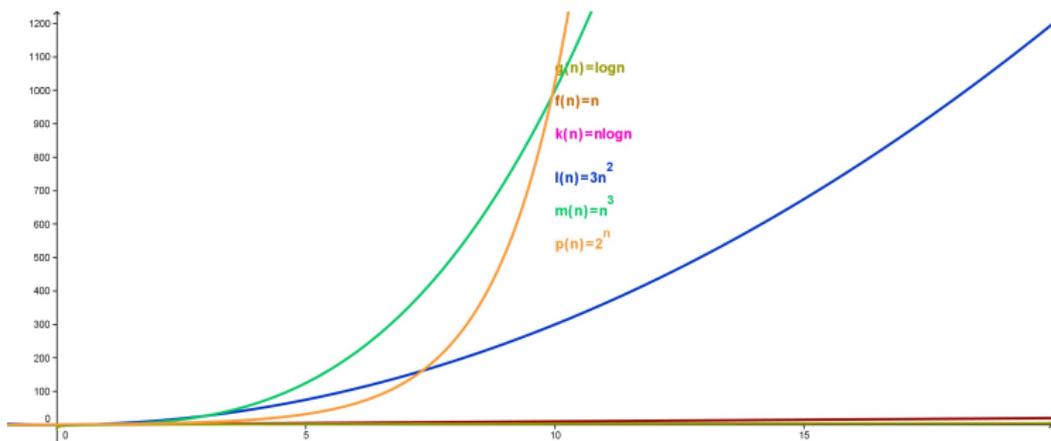
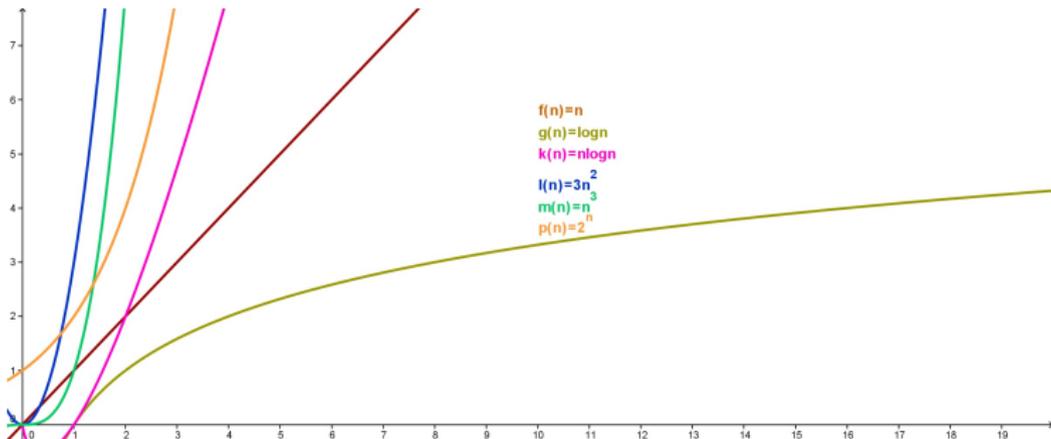
Decimos que $g(n)$ es $O(f(n))$ (escribiremos $g \in O(f)$) si existen constantes $c \in \mathbb{R}$ y $n_0 \in \mathbb{N}$ tales que $g(n) \leq cf(n)$ para todo $n \geq n_0$.

Si $f(n) = n^2$ y $g(n) = 100n$, entonces $g \in O(f)$, pero $f \notin O(g)$.

Si $f(n) = 3n^2$ y $g(n) = 50n^2 + 23n + 1$, se tiene que $g \in O(f)$ y $f \in O(g)$.

- $g \in O(f) \iff cg \in O(f) \iff g + c \in O(f)$, para $c > 0$.
- Si $h \in O(f)$ y $g \in O(f)$, entonces $h \in O(f)$.
- $g \in O(f) \iff O(g) \subset O(f)$.
- Si $g_1 \in O(f_1)$ y $g_2 \in O(f_2)$ entonces:
 - $g_1 + g_2 \in O(\max(f_1, f_2))$.
 - $g_1g_2 \in O(f_1f_2)$.
- $f = a_t n^t + \dots + a_1 n + a_0 \in O(n^t)$, para $a_t > 0$.

Formas de crecimiento más habituales



def *numero_mas_grande*(*L*):

1. $n = \text{len}(L)$

2. $m = L[0]$

3. $i = 1$

4. **while** $i < n$:

5. **if** $L[i] > m$:

6. $m = L[i]$

7. $i = i + 1$

8. **return** m

Coste

Veces

Notas

c_1

1

● n es la longitud de la lista.

c_2

1

● En la línea 4, el bucle **while** se ejecuta desde $i = 2$ hasta $n + 1$, es decir, n veces.

c_3

1

c_4

n

c_5

n

● $T(n)$ es el tiempo total de ejecución del algoritmo.

c_6

n

c_7

n

0

—

$T(n)$ es la suma del tiempo de ejecución de cada instrucción ejecutada;

$$T(n) = (c_1 + c_2 + c_3 + c_7) + (c_5 + c_6)n = O(1) + O(1)n = O(n).$$

Luego es un algoritmo lineal, y por lo tanto es eficiente.

- Hemos analizado el ▶ Tiempo de ejecución $T(n) = an^2 + bn + c$ del algoritmo ▶ *ordenar_insertando*, es decir, $O(n^2)$.
- También, es fácil comprobar que $T(n)$ del algoritmo ▶ *ordenar_burbuja* es $O(n^2)$, pues tenemos que realizar $O(n^2)$ comparaciones de los n números de la lista.

Pero hay un hecho importante que diferencia estos dos algoritmos: en ▶ *ordenar_burbuja* siempre necesitamos realizar $O(n^2)$ comparaciones, independiente de los detalles del conjunto de datos. Pero si los los datos están parcialmente ordenados (por ejemplo, en una sucesión aleatoria de números) el algoritmo ▶ *ordenar_insertando* no realiza todas las comparaciones, las constantes son más pequeñas.

Este hecho, justifica los resultados computacionales realizados en **Python** y presentados en ▶ Comparación de algoritmos: *insertando* y de la *Burbuja*

Los números de Fibonacci es una secuencia muy especial:

$$F_n = \begin{cases} 1, & \text{si } 1 \leq n \leq 2; \\ F_{n-1} + F_{n-2}, & \text{si } n > 2. \end{cases}$$

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

- Aparecen por doquier en la vida real, en informática. The Fibonacci Quarterly.
- El árbol genealógico de las abejas es : 1 zángano tiene 1 madre, 2 abuelos, 3 bisabuelos, 5 tatarabuelos, 8 tatarata-tatarabuelos, 13 tatarata-tatarata-tatarabuelos. . .
- También, la siguen muchos pétalos de flores: la flor del iris tiene 3, la de la rosa silvestre 5, la del dephinium 8, la de la cineraria 13, la de la achicoria 21... etc.

programa.py

```
def fibonacci_recursivo(n):
    if n == 1 or n == 2:
        fibo = 1
    elif n > 2:
        fibo = fibonacci_recursivo(n - 1) + fibonacci_recursivo(n - 2)
    return fibo
```

Suponemos $T(n)$ el tiempo de ejecución para $fibonacci(n)$. El tiempo de ejecución de las líneas (1) y (2) es $O(1)$ y para la línea (3) es $O(1) + T(n - 1) + T(n - 2)$. Por tanto para algunas constantes $c > 1$ y $d > 1$,

- $T(n) = c + T(n - 1) + T(n - 2)$, si $n > 2$;
- $T(n) = d$, si $n \leq 2$.

Luego $T(n) \geq c + 2T(n - 2)$ y $T(n) \in O(2^n)$. Veamos que es una cota exacta.

Hemos dicho que todo programa recursivo puede reescribirse con estructuras de control iterativas.

programa.py

```
def fibonacci_iterativo(n):  
    f1, f2 = 1,1  
    for i in range(3, n+1):  
        f1, f2 = f2, f1+ f2  
    return f2
```

Es muy fácil comprobar que el tiempo de ejecución $T(n)$ de este programa es $O(n)$

- No siempre hay una diferencia de costes tan alta entre los algoritmos iterativos y recursivos. En este caso, no obstante, podemos estar satisfechos del programa iterativo, al menos si lo comparamos con el recursivo.
- ¿Conviene usarlo siempre? No.

Hay una fórmula no recursiva de F_n que conduce a un algoritmo aún más eficiente.

Utilizando la técnica de las transparencias anteriores se prueba fácilmente:

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

programa.py

```
def fibonacci_formula(n):
    from math import sqrt
    z = 1/sqrt(5)*(((1+ sqrt(5)) / 2)**n - ((1- sqrt(5)) / 2)**n)
    return int(z)
```

Claramente el tiempo de ejecución es constante $c = O(1)$.

Si un algoritmo es recursivo, su tiempo de ejecución puede ser descrito por una **ecuación de recurrencia**. Una recurrencia para el tiempo de ejecución $T(n)$ de un algoritmo ▶ Divide – y – vencerás está basada en tres pasos:

- Si el tamaño del problema es pequeño, digamos $n \leq c$ para alguna constante c , la solución requiere tiempo constante $O(1)$.
- Supongamos que dividimos el problema en k subproblemas, cada uno de tamaño n/b .
- Denotamos por $D(n)$ el tiempo para dividir el problema y $C(n)$ el tiempo para combinar las soluciones de los subproblemas para obtener la solución del problema original.

$$T(n) = \begin{cases} O(1), & \text{si } n \leq c; \\ kT(n/b) + D(n) + C(n), & \text{en otro caso.} \end{cases}$$

Resolver este tipo de ecuaciones recurrentes, requiere unas pocas matemáticas. Pero, para algunas ecuaciones la solución es muy fácil:

Sean a y b dos números naturales y k el máximo de bits de a y b . El tiempo de ejecución de:

- La suma y resta binaria $a + b$ y $a - b$ es $O(k)$.
- La multiplicación y división euclidea $a \times b$ y $a = b \times q + r$ es $O(k^2)$.
- El algoritmo de Euclides es $O(k^3)$, basta observar que el número de etapas en la recursión del algoritmo ▶ mcd es $O(k)$.

Ya hemos visto que el ▶ calculo factorial de un número n es $O(n)$. Si k es el número de bits de n , entonces necesitamos realizar $O(n)$ multiplicaciones de números con a lo sumo k bits, es decir,

$$O(n \times k^2) = O(2^k \times k^2).$$

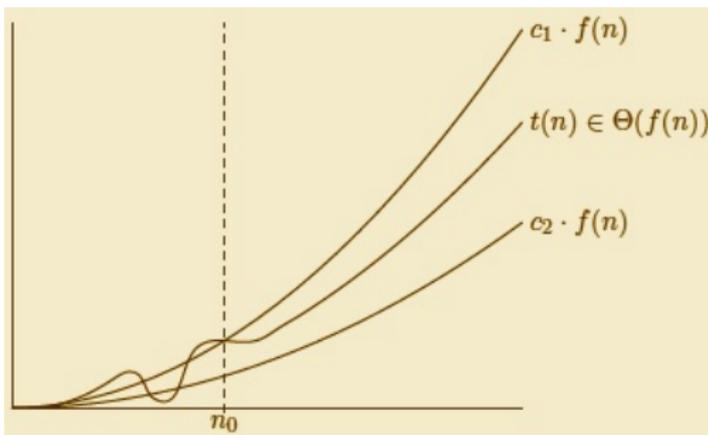
Dos de los problemas más importantes en Teoría computacional de números son:

- **Test de primalidad.** Dado un número n decidir si n es primo o no (PROBLEMA DECISIÓN).
- **Factorización de enteros.** Encontrar la factorización en primos de un número n . Por ejemplo, si $n = 12$ encontrar 2, 2 y 3, puesto que $12 = 2 \times 2 \times 3$.

La notación Θ denota el conjunto de las funciones con la misma tasa de crecimiento:

$$g \in \Theta(f) \iff (g \in O(f) \text{ y } f \in O(g)) \iff O(f) = O(g).$$

- Si $f(n) = n^2$ y $g(n) = 100n$, entonces $g \in O(f)$, pero $g \notin \Theta(f)$.
- Si $f(n) = 3n^2$ y $g(n) = 50n^2 + 23n + 1$, se tiene que $g \in \Theta(f)$.



Esta es la transparencia número:
 2^6