

Un algoritmo es una herramienta para resolver un problema computacional.

¿ Cómo encontrar el número mayor de una sucesión finita de números ?

Se trata de diseñar un algoritmo para resolver el siguiente problema:

Entrada: Una sucesión de n números $L = [a_1, a_2, \dots, a_n]$.

Salida: El número m de la sucesión L tal que todos los elementos a_i no son más grandes que m , es decir, $a_i \leq m$.

Dada una entrada como $L = [2, -5, 22, 6, 0, -10, 2, 22, 3]$, un algoritmo para resolverlo debería devolver el número 22.

El número más grande

numero_mas_grande(lista)

Hacer $n = longitud(lista)$, $i = 2$ y $m = lista(1)$

Mientras $i < n + 1$, Repetir

Si $lista(i) > m$, Hacer $m = lista(i)$

Hacer $i = i + 1$

Fin

Una implementación del algoritmo en **Python**:

programa.py

```
def numero_mas_grande(lista):
```

```
n=len(lista)
```

```
m=lista[0]
```

 $i=1$

```
while i<n:
```

```
if lista[i]>m:
```

```
m=lista[i]
```

$$i=i+1$$

```
return m
```

El algoritmo anterior propuesto es **iterativo** pues hemos utilizado **ciclos** o **bucles** para indicar la repetición de una serie de instrucciones.

Un algoritmo es **recursivo** cuando expresa una parte de la solución de un problema en términos de una llamada a sí mismo.

El **factorial** de un número entero $n > 0$ se define iterativamente como $n! = 1 \times 2 \times \cdots \times n$. También, lo proporciona la siguiente fórmula recursiva:

$$n! = \begin{cases} 1, & \text{si } n = 1; \\ n \times (n-1)!, & \text{si } n > 1. \end{cases}$$

programa.py

```
def fact_recursivo(n):
    if n==1:
        fact =1
    elif n>1:
        fact=n*fact_recursivo(n-1)
    return fact
```

programa.py

```
def fact_iterativo(n):
    fact=1
    for i in range(1,n+1):
        fact=fact*i
    return fact
```

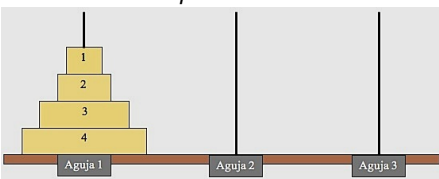
Calcula el resto de dividir el mayor de los dos números por el menor de ellos. Si el resto es cero, entonces el *mcd* es el menor de ambos números. Si el resto es distinto de cero, el *mcd* de n y m es el *mcd* de otro par de números, a saber, el formado por el menor de n y m , y por dicho resto.

```
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)
```

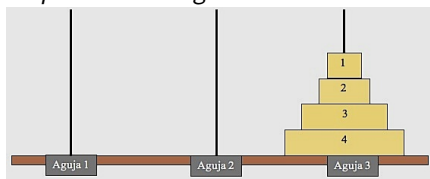
Alrededor del año 1880 apareció en Francia un antiguo rompecabezas indio, rescatado por el matemático N. Claus. Su compatriota, el escritor Henri de Parville amplió y adornó la leyenda de este juego poco tiempo después:

En el gran templo de Benarés, debajo de la cúpula que marca el centro del mundo, yace una base de bronce, en donde se encuentran acomodadas tres agujas de diamante, cada una del grueso del cuerpo de una abeja y de una altura de 50 cm aproximadamente. En una de estas agujas, Dios, en el momento de la Creación, colocó sesenta y cuatro discos de oro -el mayor sobre la base de bronce, y el resto de menor tamaño conforme se va ascendiendo-. Día y noche, incesantemente, los sacerdotes del templo se turnan en el trabajo de mover los discos de una aguja a otra de acuerdo con las leyes impuestas e inmutables de Brahma, que requieren que siempre haya algún sacerdote trabajando, que no muevan más de un disco a la vez y que deben colocar cada disco en alguna de las agujas de modo que no cubra a un disco de radio menor. Cuando los sesenta y cuatro discos hayan sido transferidos de la aguja en la que Dios los colocó, en el momento de la Creación, a otra aguja, el templo y los brahmanes se convertirán en polvo y, junto con ellos, el mundo desaparecerá.

La situación de partida es esta:

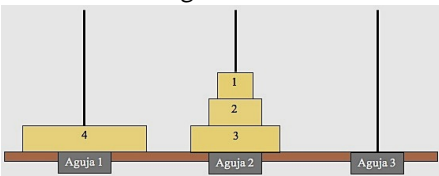


Y queremos a llegar a esta:

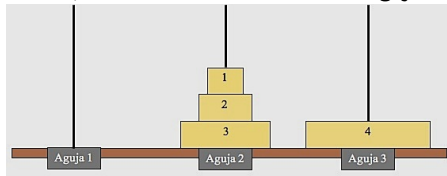


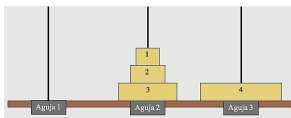
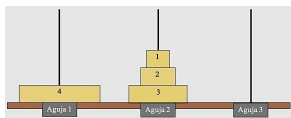
¿ Como podemos mover el disco 4 de la **Aguja 1** a la **Aguja 3** ?

Hemos de conseguir esta:



Ahora, movemos el disco 4 a la Aguja 3

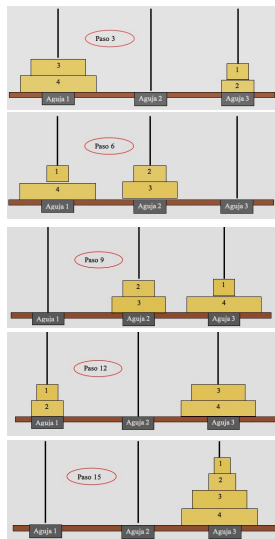




Puesto que el disco 4 no se va a mover de la aguja final. Resolver el problema de las torres de Hanoi con cuatro discos requiere:

- ① resolver el problema de las torres de Hanoi con tres discos, aunque pasándoles de la aguja inicial a la aguja libre;
- ② mover el cuarto disco de la aguja inicial a la aguja final; y
- ③ resolver el problema de las torres de Hanoi con los tres discos de la aguja libre a la aguja final.

- La solución que proponemos requiere una doble recursión. Aplicamos el mismo método a los pasos (1) y (3). Y así sucesivamente.
- Hay una situación trivial ó *de parada*: el problema de las torres de Hanoi para un sólo disco (basta con mover el disco de la aguja en la que esté insertado a la aguja final).



...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 (veremos esto más adelante).
- 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)` ?

Una gran parte de la informática consiste en estudiar la forma de buscar, ordenar y recuperar datos.

Ordenar el contenido de una lista es un problema fundamental porque se plantea en numerosos campos de la programación, la propia palabra **ordenador** lo pone de manifiesto.

DONALD E. KNUTH dice, en el tercer volumen **Sorting and searching** de *The art of computer programming*, un texto clásico de programación: los fabricantes de ordenadores de los años 60/70 estimaron que más del 25% del tiempo de ejecución en sus ordenadores se dedicaba a ordenar cuando consideraban al conjunto de sus clientes. De hecho, había muchas construcciones en las que la tarea de ordenar era responsable de más de la mitad del tiempo de computación.

Algoritmos para ordenar y buscar, también son una buena manera de explicar cómo comparar la eficacia o rendimiento de diferentes algoritmos y, de definir el concepto de tiempo de ejecución.

Muchos problemas de localizar elementos (números, nombres, caracteres, etc.) en una estructura de datos (listas, conjuntos, arrays, etc.) se pueden plantear como sigue:

► El problema de la búsqueda

Entrada: Un número a y una sucesión de n números $L = [a_1, a_2, \dots, a_n]$.

Salida: El primer índice i tal que $a_i = a$, ó -1 si a no es un elemento de L .

- Dada una entrada como $(22, [2, -5, 22, 6, 0, -10, 2, 22, 3])$, un algoritmo de búsqueda debería retornar 3 .
- Dada una entrada como $(7, [2, -5, 22, 6, 0, -10, 2, 22, 3])$, un algoritmo de búsqueda debería retornar -1 .

Un posible algoritmo consiste en buscar el elemento comparándolo secuencialmente con cada elemento de la sucesión hasta encontrarlo, o hasta que se llegue al final. La existencia se puede asegurar cuando el elemento es localizado, pero no podemos asegurar la no existencia hasta no haber comparado todos los elementos.

¿ Hay una forma más *rápida* que la búsqueda lineal ? La respuesta es sí. Lo podemos hacer mucho mejor si la lista está ordenada.

Supongamos que tenemos que buscar el número de teléfono de una persona con la ayuda de una guía telefónica. Abrimos la guía por la mitad y mirando el primer nombre de la página, inmediatamente sabemos en qué mitad de la guía está el número. Podemos hacer el mismo procedimiento con esa parte y, así sucesivamente. Con solamente comprobar dos nombres, hemos eliminado $\frac{3}{4}$ de los números de la guía.

Esta técnica se denomina búsqueda binaria, porque en cada etapa divide la lista (=la guía de teléfonos) en dos partes iguales: los valores que están antes y los que están después del valor que hemos comprobado. Para hacernos una idea de la rapidez del método:

- Una lista de $256 = 2^8$ requiere a lo sumo 8 comprobaciones.
- Y una lista de 500000 números (la población de Cantabria), requiere $\log_2(500000) \equiv 19$ comprobaciones.
- Y una lista de un millón de números 2×500000 , requiere 20 comprobaciones, puesto que $2 \times 2^{19} = 2^{20}$.
- Claramente es un algoritmo recursivo, aunque aquí presentamos un programa iterativo en **Python**.

programa.py

```
def busqueda_binaria(a, L):
    menor=0
    grande=len(L)-1
    while menor <= grande:
        mitad=(menor + grande) / 2
        if L[mitad] > a:
            grande=mitad + 1
        elif L[mitad] < a:
            menor=mitad +1
        else:
            return mid +1
    return -1
```

Comparamos los tiempos que necesita **Python** para ejecutar los programas *busqueda_lineal* y *busqueda_binaria*, utilizando la primitiva *time* de Python y, el siguiente programa para un procedimiento *general* con entrada *argumentos*:

```
def tiempo_general(argumentos):
    t1=time.time()
    general(argumentos)
    t2=time.time()
    return(t2-t1)
```

```
>>> tiempo_busqueda_lineal(30000, range(10**6))
0.000890
>>> tiempo_busqueda_binaria(30000, range(10**6))
0.000845
>>> tiempo_busqueda_lineal(500000, range(10**6))
0.013966
>>> tiempo_busqueda_binaria(500000, range(10**6))
0.000257
>>> tiempo_busqueda_lineal(10^6, range(10**6))
0.023949
>>> tiempo_busqueda_binaria(10^6, range(10**6))
0.000322
```

Hemos observado que existen algoritmos más rápidos para el problema de la búsqueda si la secuencia está ordenada. Planteamos el problema de la ordenación, como sigue:

► El problema de la ordenación

Entrada: Una secuencia de n números $L = [a_1, a_2, \dots, a_n]$.

Salida: Una permutación (reordenamiento) $[a'_1, a'_2, \dots, a'_n]$ de L tal que $[a'_1 \leq a'_2 \leq \dots \leq a'_n]$.

- Dada una entrada como $L = [2, -5, 22, 6, 0, -10, 2, 22, 3]$, un algoritmo para resolverlo debería retornar $[-10, -5, 0, 2, 2, 3, 6, 22, 22]$.
- Ordenar es, quizás, el problema más estudiado en algorítmica y/o informática. Existen una gran variedad de soluciones diferentes, cada una con sus ventajas e inconvenientes o especialmente adaptada para tratar casos particulares.

Aquí, nosotros sólo veremos tres de ellas.

Tal vez la manera más obvia para ordenar es comparar a_1 y a_2 , intercambiar si no están ordenados, a continuación, hacer lo mismo con a_2 y a_3 , con a_3 y a_4 , etc. Durante este procedimiento, las números grandes tienden a moverse hacia la derecha, y de hecho en la primera pasada, el mayor ha quedado fijo. El método se denomina "clasificación por el método de la burbuja" porque los elementos más grandes van hacia arriba a su posición correcta, como la "burbuja".

programa.py

```
def ordenar_burbuja(L):
    for i in range(1,len(L)):
        for j in range(0, len(L)-i):
            if L[j] > L[j+1]:
                temp=L[j]
                L[j]=L[j+1]
                L[j+1]=temp
    return L
```

La idea del ordenamiento por inserción es la misma a como ordenamos las cartas en una partida de *tute*. Supongamos que $1 < j < n$ y, que las cartas del palo de *bastos* a_1, \dots, a_{j-1} han sido reordenadas tales que $a'_1 \leq a'_2 \leq \dots \leq a'_{j-1}$. Se compara la nueva carta a_j con $a'_{j-1}, a'_{j-2}, \dots$ hasta que a_j debe ser insertado entre a'_i y a'_{i+1} ; entonces movemos las cartas $a'_{i+1}, \dots, a'_{j-1}$ un espacio a la derecha y ponemos la nueva carta en la posición $i + 1$.

programa.py

```
def ordenar_insertando(L):
    for j in range(1,len(L)):
        i=j-1
        temp=L[j]
        while i > -1 and L[i] > temp:
            L[i+1]=L[i]
            i=i-1
        L[i+1]=temp
    return L
```

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

25 / 62

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
```

Programa: *ordenar_mezclando*

programa.py

```
def ordenar_mezclando(L):
    if len(L) < 2:
        return L
    else:
        mitad= len(L)/2
        izquierda=ordenar_mezclando(L[:mitad])
        derecha=ordenar_mezclando(L[mitad:])
        return mezclando(izquierda,derecha)
```

Igual que hemos hecho con el resto de los programas, vamos a comparar el tiempo necesario que necesita **Python** para ejecutar los programas *ordenar_insertando* y *ordenar_mezclando*. Utilizamos la primitiva *random* de **Python** y el

► Programa tiempo

Comparando los algoritmos *insertando* y *mezclando*

```
>>> L1=random.sample(range(10^4),10**4);
>>> L12=random.sample(range(2*10**4),2*10**4);
>>> L2=random.sample(range(10**5),10**5);
>>> tiempo_ordenar_insertando(L1)
0.9200
>>> tiempo_ordenar_mezclando(L1)
1.6200
>>> tiempo_ordenar_insertando(L12)
3.5700
>>> tiempo_ordenar_mezclando(L12)
3.1900
>>> tiempo_ordenar_insertando(L2)
88.7700
>>> tiempo_ordenar_mezclando(L2)
15.5500
```

En el [capítulo siguiente](#) intentaremos explicar formalmente estos y el resto de los resultados computacionales producidos por los programas de **Python**.

El **Análisis de algoritmos** es una rama de la informática que estudia la **eficiencia** o **complejidad** o **coste computacional** o el **tiempo de ejecución** de los algoritmos.

- El objetivo práctico de medir la eficiencia de los algoritmos es guiar las decisiones de diseño del programa y mantenerlo tan bajo como sea posible el consumo de los diversos recursos.
- Es medir la cantidad de recursos necesarios para su ejecución, con objeto de compararla con la de otros algoritmos ya construidos o por inventar.
- Sería interesante que el coste de estos recursos necesarios para ejecutar el programa se formulara en una unidad monetaria. Pero, se prefiere que sea independiente de este tipo de factores, ajenos propiamente al diseño del algoritmo.
- Se eligen como medida factores que se puedan definir en términos de programación y que tienen repercusión directa en el coste real:

- tiempo de cálculo y,
- cantidad de memoria interna (en menor medida)

- Es un concepto relativo, un algoritmo es más eficiente que otro si realiza las mismas tareas con menos recursos. Aspecto que puede depender de:

Puesto que pretendemos medir el tiempo de ejecución en función del tamaño de los datos, debemos clarificar el concepto de "**tamaño de los datos**".

Lo mejor que podemos decir, de momento, es que depende del problema:

- Para muchos problemas, tales como ▶ El problema de la búsqueda y ▶ El problema de la ordenación la medida más natural es el número de elementos de sucesión.
- Para el Algoritmo de las ▶ Torres de Hanoi, el número de discos.
- Para multiplicar dos números enteros el número de bits de ambos.
- Para el algoritmo de Euclides del ▶ máximo común divisor, también el número de bits.

- 1 Sobre datos de números naturales, es preciso especificar cuál de estas dos posibilidades se estudia: el valor, n ó el número de bits, $\log n$). Pues se diferencian en una función exponencial. La factorización de un número n en factores primos es un problema computacionalmente costoso, de hecho la seguridad de muchas actuaciones con tarjetas de crédito o internet depende de esta dificultad. Sin embargo, se puede diseñar elementalmente un algoritmo lineal en n .
- 2 Sobre datos de listas de números, quizás sea interesante considerar el número total de ellos n y, el máximo de número de bits de los números, *aparentemente* es más costoso, ordenar números de 1024 bits que de 16.

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

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

Notas

c_1

$n - 1$

● *n* es la longitud de la lista.

c_2

$n - 1$

● t_j es el número de veces que se testa la línea 4, en el bucle **while** para el valor *j*.

c_3

$n - 1$

c_4

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

c_5

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

c_6

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

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

c_7

$n - 1$

0

— — — — —

$T(n)$ 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:

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

El tiempo de ejecución de un algoritmo puede depender del estado de la entrada para un tamaño dado. Por ejemplo, en el algoritmo *ordenar_insertando*

El caso mejor

Si la secuencia está ya ordenada, se tiene que $L(i) \leq temp$, si $i = j-1$. Luego $t_j = 1$, para cada $j = 2, \dots, n$. Expresamos $T(n)$ como:

$$T(n) = an + b,$$

donde $a = c_1 + c_2 + c_3 + c_4 + c_7$ y $b = -a$.

$T(n)$ es una función lineal de n .

El caso peor

Si la secuencia está ordenada de forma inversa. Entonces tenemos que comparar cada elemento $L(j)$ con cada elemento de la subsucesión ordenada $L[1, \dots, j-1]$ y, así $t_j = j$, para cada $j = 2, 3, \dots, n$. Puesto que:

$$\sum_{j=2}^n j = \frac{n(n+1)}{2} - 1; \sum_{j=2}^n (j-1) = \frac{n(n-1)}{2},$$

podemos expresar $T(n)$ como:

$$T(n) = an^2 + bn + c;$$

para ciertas constantes a , b y c que dependen del coste c_i en la instrucción i .

$T(n)$ es una función cuadrática de n .

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:

- ❶ El tiempo de ejecución del **caso peor** es una cota superior. Así garantizamos que el algoritmo nunca costará más.
- ❷ 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.
- ❸ 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 $f \in O(g)$.

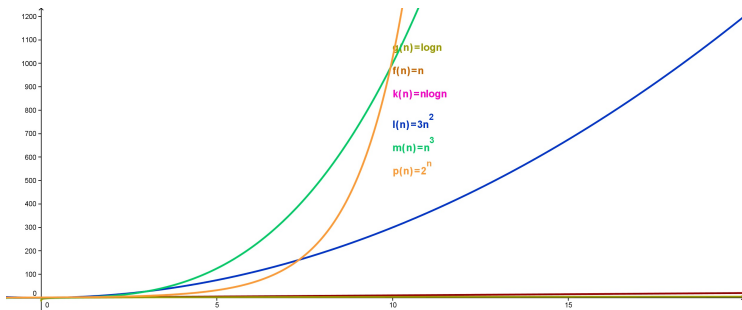
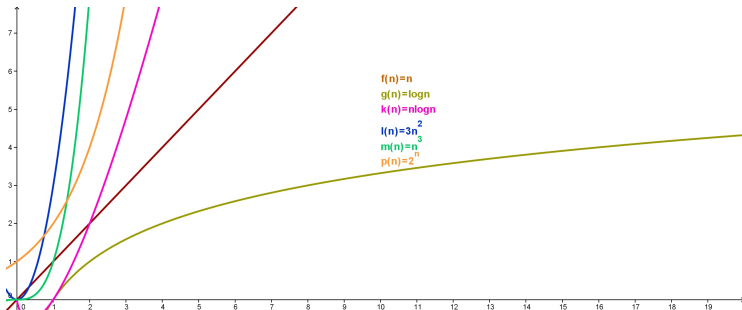
- $g \in O(f) \iff cg \in O(f) \iff g + c \in O(f)$, para $c > 0$.
- Si $h \in O(f)$ y $g \in O(f)$, entonces $h \in O(f)$.
- $g \in O(f) \iff O(g) \subset O(f)$.
- Si $g_1 \in O(f_1)$ y $g_2 \in O(f_2)$ entonces:
 - $g_1 + g_2 \in O(\max(f_1, f_2))$.
 - $g_1 g_2 \in O(f_1 f_2)$.
- $f = a_t n^t + \dots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

La escala como usualmente clasificamos el tiempo de ejecución o complejidad computacional $f(n)$ de un algoritmo de tamaño n , consiste en la siguiente serie de funciones :

- Constante: $O(1)$.
- Lineal : $O(n)$.
- Quasilineal : $O(n \log n)$.
- Cuadrático: $O(n^2)$.
- Cúbico: $O(n^3)$.
- Polinómico: $O(n^t)$ con t fijo.
- Exponencial: $O(t^n)$ con $t > 1$ fijo.

- Otras formas de crecimiento que pueden aparecer con cierta frecuencia son: $O(\sqrt{n})$, $O(n!)$ o $O(n^n)$.
- Algunas veces se entiende por crecimiento **quasi-lineal** el que corresponde a $O(n(\log n)^t)$.
- Suele admitirse como algoritmo eficiente el que alcanza un coste quasi-lineal. Cualquier coste superior a polinómico, e incluso polinómico de grado alto, se le considera intratable.

Formas de crecimiento más habituales



No existen normas generales y, aunque en ocasiones es simple encontrar un cota superior del tiempo de ejecución de un programa, en otras puede ser una tarea difícil. Algunas reglas que se aplican con cautela son:

- Cada asignación, lectura o escritura es $O(1)$.
- El de una secuencia de instrucciones viene determinado por la regla de las sumas. Esto es, el tiempo de ejecución de la secuencia es (sin tener en cuenta un factor constante) el de la instrucción de la secuencia que más tarda.
 - De una instrucción 'if' es el costo correspondiente a la ejecución de las instrucciones condicionales, más el tiempo para evaluar la condición.
 - Ejecutar un bucle es la suma, sobre el número de veces que se ejecuta, del tiempo empleado por el cuerpo, más el tiempo empleado en evaluar la condición, que típicamente es $O(1)$.
- Cuando existen subprogramas, se comienza la evaluación por aquellos que no llaman a otros.
- Cuando el algoritmo es recursivo, lo usual es considerar un tiempo desconocido $T(n)$ y tratar de encontrar una ecuación de recurrencia para $T(n)$.

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

c_6

n

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

c_7

n

0

—

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

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

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

- 46 / 62

Analizamos dos clásicos algoritmos recursivos:

- Sea $T(n)$ la complejidad computacional del algoritmo de las Torres `hanoi` con n discos. Se tiene que

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

Claramente, iterando la ecuación de arriba se tiene: $T(n)$ es $O(2^n)$.

- Tanto para el caso del algoritmo `fact_iterativo` como para el algoritmo `fact_recursivo` la complejidad computacional es $O(n)$.

- El algoritmo es eficiente, puesto que es lineal. Sin embargo, la complejidad computacional cuando el tamaño de la entrada es el número de bits de n , es decir, $t = \log n$, entonces resulta exponencial, como veremos en

► Complejidad binaria

- Por otro lado, tanto la versión iterativa como la recursiva del algoritmo presenta el mismo coste computacional, eso no es la situación en general como ilustra el siguiente tema

► Números de Fibonacci

Los números de Fibonacci es una secuencia muy especial:

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

programa.py

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

- 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.

Suponemos $T(n)$ el tiempo de ejecución para $\text{fibonacci}(n)$. 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:

$$\begin{pmatrix} T(n) \\ T(n-1) \\ 0 \end{pmatrix} = \begin{pmatrix} c & 1 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ T(n-1) \\ T(n-2) \end{pmatrix}$$

Iterando la ecuación de arriba, obtenemos:

$$\begin{pmatrix} T(n) \\ T(n-1) \\ 0 \end{pmatrix} = \begin{pmatrix} c & 1 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix}^{n-2} \begin{pmatrix} 1 \\ T(2) \\ T(1) \end{pmatrix}$$

Puesto que la matriz $A = \begin{pmatrix} c & 1 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix}$ es diagonalizable: $A = PDP^{-1}$, con

$$P = \begin{pmatrix} 1 & 1 & 1 \\ 0 & 1-c & 0 \\ 0 & 0 & -c \end{pmatrix}, D = \begin{pmatrix} c & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix}, P^{-1} = \begin{pmatrix} 1 & \frac{1}{c-1} & \frac{1}{c} \\ 0 & \frac{1}{1-c} & 0 \\ 0 & 0 & \frac{-1}{c} \end{pmatrix}.$$

$$\begin{pmatrix} T(n) \\ T(n-1) \\ 0 \end{pmatrix} = (PDP^{-1})^{n-2} \begin{pmatrix} 1 \\ T(2) \\ T(1) \end{pmatrix} = PD^{n-2}P^{-1} \begin{pmatrix} 1 \\ T(2) \\ T(1) \end{pmatrix}$$

$$\begin{pmatrix} T(n) \\ T(n-1) \\ 0 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 \\ 0 & 1-c & 0 \\ 0 & 0 & -c \end{pmatrix} \begin{pmatrix} c^{n-2} & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} 1 & \frac{1}{c-1} & \frac{1}{c} \\ 0 & \frac{1}{1-c} & 0 \\ 0 & 0 & \frac{-1}{c} \end{pmatrix} \begin{pmatrix} 1 \\ T(2) \\ T(1) \end{pmatrix}$$

Operando en la ecuación de arriba:

$$T(n) = c^{n-2} + (c^{n-3} + c^{n-4} + \dots + c + 1)T(2) + c^{n-3}T(1) =$$

$$c^{n-2} + (2c^{n-3} + c^{n-4} + \dots + c + 1)d \in O(c^n)$$

El cálculo de la sucesión de Fibonacci recursivo es exponencial.

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

programa.py

```
def fibonacci_iterativo(n):  
    if n==1 or n==2:  
        F=1  
    else  
        f1 = 1  
        f2 = 1  
        for i in range(3, n+1):  
            F=f1+f2  
            f1=f2  
            f2=F  
    return F
```

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  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:

En el algoritmo `ordenar_mezclando` la ecuación recurrente adopta esta forma:

$$T(n) = \begin{cases} O(1), & \text{si } n = 1; \\ 2T(n/2) + O(n), & \text{si } n > 1. \end{cases}$$

Basta observar que la etapa **mezclando** es lineal y, que dividir la lista en dos mitades es constante.

Iterando una vez:

$$T(n) = 2(T(n/4) + T(n/4)) + O(n/2) + O(n) = 4T(n/4) + 2O(n).$$

Iterando $k - 1$ veces:

$$T(n) = 2^k T(n/k) + (k-1)O(n).$$

El valor de k para el que $n/2^k = 1$, es $k = \log n$. Tenemos entonces,

$$T(n) = nT(1) + O(n) \log n/2 = O(1) + O(n) \log n/2 = O(n) \log n = O(n \log n).$$

La diferencia del algoritmo cuadrático `ordenar_insertando` y este quasi-lineal, queda patente en los resultados computacionales presentados en

- La suma y resta binaria $a + b$ y $a - b$ es $O(k)$.
- La multiplicación y división euclídea $a \times b$ 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 $O(k)$.

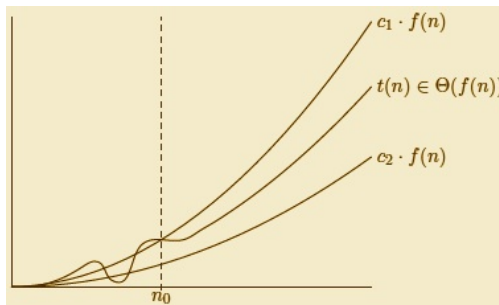
$$O(n \times k^2) = O(2^k \times k^2).$$

- **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 $g \in \Theta(f)$.



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

Es muy fácil comprobar que el tiempo de ejecución $T(n)$ de todos los algoritmos calculados en estas notas verifican que

Si $T(n) \in O(f(n))$, entonces $T(n) \in \Theta(f(n))$.

Supongamos que queremos ordenar con el ordenador un millón de números: Dos opciones:

- 1 Un superordenador que ejecuta 100 millones de instrucciones por segundo. Y que el programador más astuto del mundo necesitó $2n^2$ instrucciones para ordenar n números implementando el algoritmo `ordenar_insertando` en lenguaje máquina.
- 2 Un PC que ejecuta sólo 1 millón de instrucciones por segundo. Y que un programador mediocre (yo mismo) utilizó $50n \log n$ instrucciones para los n números al implementar el algoritmo `ordenar_mezclando` en un lenguaje de alto nivel (C++ ó Python) con un compilador ó interprete ineficiente.

Para ordenar el millón de números el superordenador necesitó:

$$\frac{2 \times (10^6)^2 \text{ instrucciones}}{10^8 \text{ instrucciones/segundo}} = 20000 \text{ segundos} \equiv 5.56 \text{ horas},$$

Y el PC:

$$\frac{50 \times 10^6 \log 10^6 \text{ instrucciones}}{10^6 \text{ instrucciones/segundo}} \equiv 1000 \text{ segundos} \equiv 16.67 \text{ minutos.}$$

