

ANÁLISIS DE ALGORITMOS

Concepto de Eficiencia

Un algoritmo **es eficiente** cuando logra llegar a sus objetivos planteados utilizando la menor cantidad de recursos posibles, es decir, minimizando el uso memoria, de tiempo y de esfuerzo humano.

Un algoritmo **es eficaz** cuando alcanza el objetivo primordial, el análisis de resolución del problema lo realiza prioritariamente.

Puede darse el caso de que exista un algoritmo eficaz pero no eficiente, en lo posible debemos de manejar estos dos conceptos conjuntamente.

La eficiencia de un programa tiene dos ingredientes fundamentales: **memoria y tiempo**.

Análisis de Algoritmos

Consiste en la evaluación de la eficiencia del algoritmo.

Los recursos que se evalúan son la memoria y **el tiempo**.

Eficiencia en memoria y en tiempo son conceptos inversos a la cantidad de memoria y tiempo que consume.

Evaluación de Memoria

La determinación de la cantidad de memoria que precisa un algoritmo suele ser sencilla en la mayoría de los casos, pero cuando el algoritmo utiliza memoria o variables dinámicas o consume gran cantidad de memoria estática se hace necesario el análisis de memoria. Para ello se utilizan los mismos métodos que en la evaluación del tiempo.

Evaluación de Tiempo

El tiempo que utiliza un algoritmo para resolver un determinado caso de un problema depende de tres factores:

- 1.- Tamaño del ejemplar (datos de entrada)
- 2.- Implantación del algoritmo (ordenador + programa compilado).
- 3.- Características del ejemplar sobre el que actúa.

Estrategias de Análisis

Siempre que se trata de resolver un problema, puede interesar considerar distintos algoritmos, con el fin de utilizar el más eficiente. Pero, **¿cómo determinar cuál es "el mejor"?**.

- ♦ **La estrategia empírica (a posteriori)** consiste en programar los algoritmos y ejecutarlos en un computador sobre algunos ejemplares de prueba.
- ♦ **La estrategia teórica (a priori)** consiste en determinar matemáticamente la cantidad de recursos (tiempo, espacio, etc.) que necesitará el algoritmo en función del tamaño del ejemplar considerado. Se llaman funciones de complejidad respecto al algoritmo.

Concepto de Tamaño

Formalmente es el número de bits necesario para representar un ejemplar del problema de forma razonablemente compacta. En la práctica se sustituye por algún parámetro que sea más o menos proporcional al tamaño en bits.

- ♦ En ordenación : número de elementos.
- ♦ En problemas aritméticos : se usa la longitud de los operandos.
- ♦ Etc.

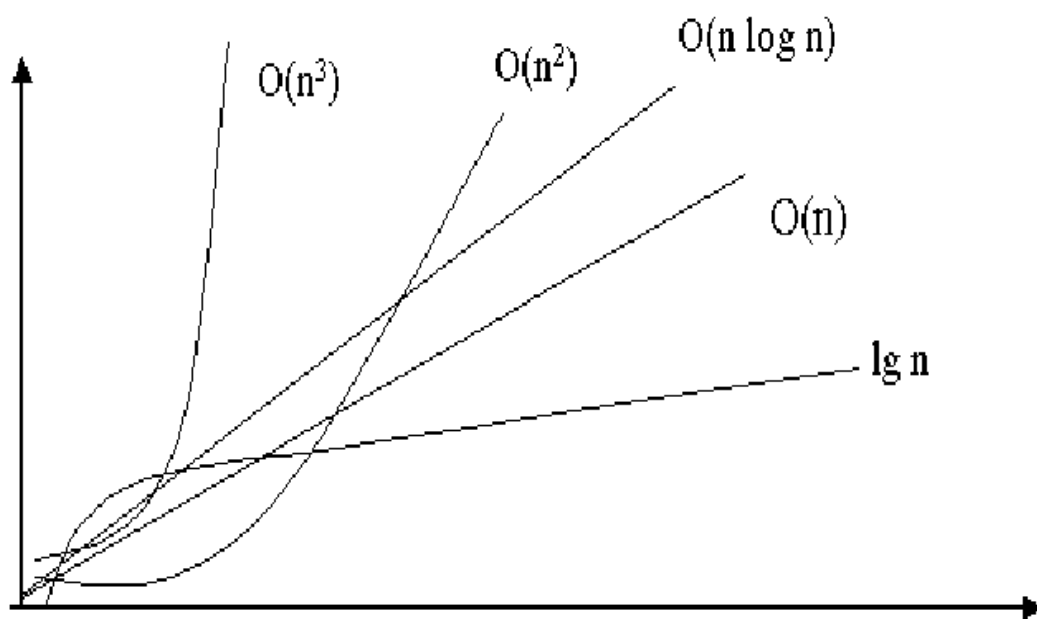
Órdenes de Complejidad

Se dice que $O(f(n))$ define un "**orden de complejidad**". Escogeremos como representante de este orden a la función $f(n)$ más sencilla del mismo. Así tendremos

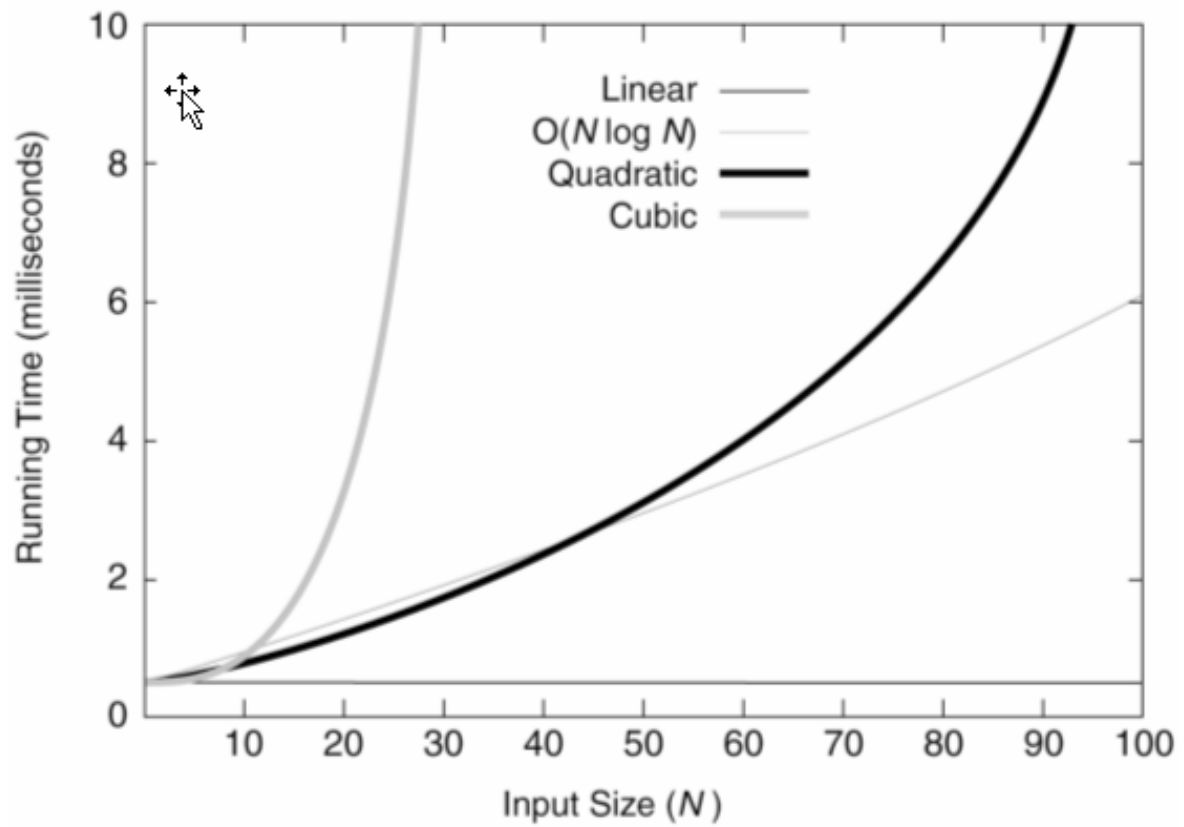
$O(1)$	orden constante
$O(\log n)$	orden logarítmico
$O(n)$	orden lineal
$O(n \log n)$	orden nlogarítmico
$O(n^2)$	orden cuadrático
$O(n^a)$	orden polinomial ($a > 2$)
$O(a^n)$	orden exponencial ($a > 2$)
$O(n!)$	orden factorial

Es más, se puede identificar una jerarquía de órdenes de complejidad que coincide con el orden de la tabla anterior; jerarquía en el sentido de que cada orden de complejidad superior tiene a los inferiores como subconjuntos. Si un algoritmo A se puede demostrar de un cierto orden O, es cierto que también pertenece a todos los órdenes superiores (la relación de orden cota superior es transitiva); pero **en la práctica lo útil es encontrar la "menor cota superior", es decir el menor orden de complejidad que lo cubra.**

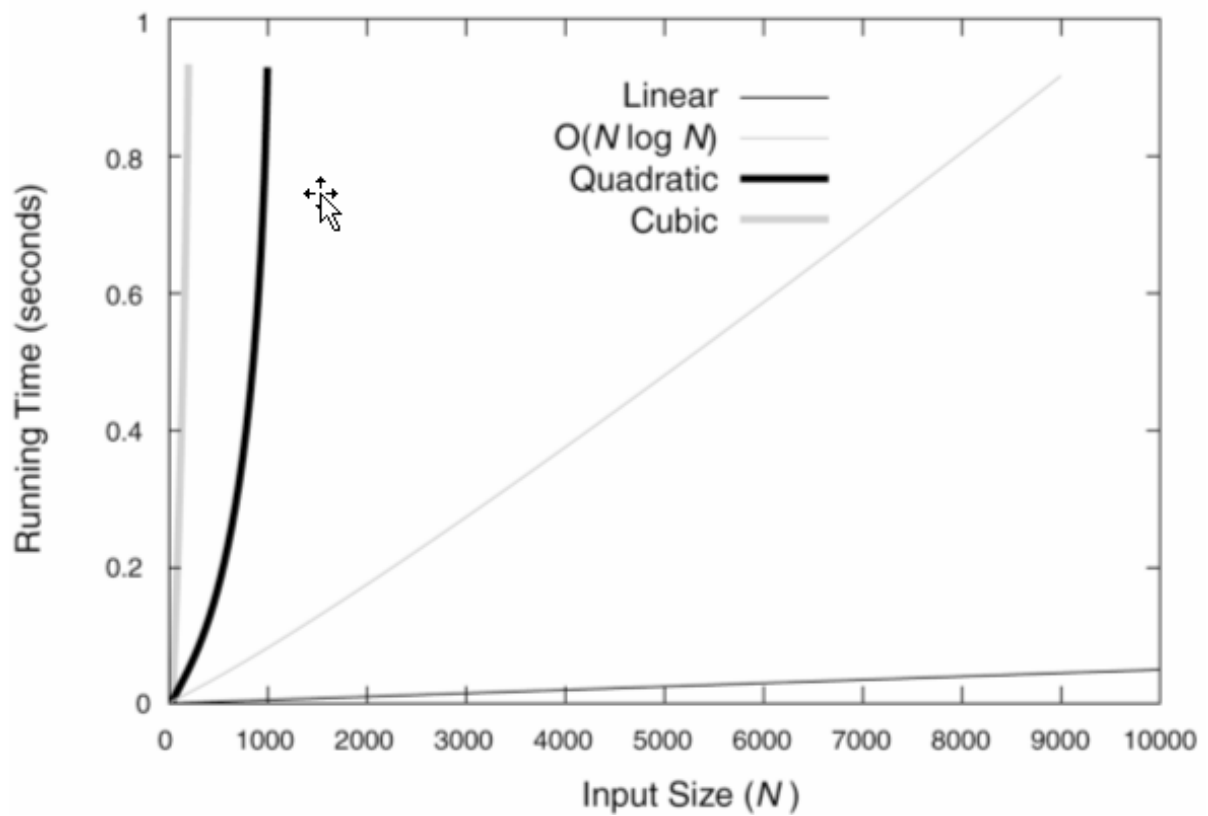
Un ejemplo de algunas de las funciones más comunes en análisis de algoritmos:



Tiempo ejecución para entradas pequeñas



Tiempo ejecución para entradas grandes



Efecto de duplicar el tamaño del problema

$T(n)$	$n=100$	$n=200$
$k_1 \log n$	1 h.	1,15 h.
$k_2 n$	1 h.	2 h.
$k_3 n \log n$	1 h.	2,30 h.
$k_4 n^2$	1 h.	4 h.
$k_5 n^3$	1 h.	8 h.
$k_6 2^n$	1 h.	$1,27 \cdot 10^{30}$ h.

Efecto de duplicar el tiempo disponible

$T(n)$	$t=1$ h.	$t=2$ h.
$k_1 \log n$	$n=100$	$n=10000$
$k_2 n$	$n=100$	$n=200$
$k_3 n \log n$	$n=100$	$n=178$
$k_4 n^2$	$n=100$	$n=141$
$k_5 n^3$	$n=100$	$n=126$
$k_6 2^n$	$n=100$	$n=101$

- Notación O (cota superior):

$$O(f(n)) = \left\{ g : \mathcal{N} \rightarrow \mathcal{R}^{\geq 0} \mid \exists c \in \mathcal{R}^+, \exists n_0 \in \mathcal{N} \cdot \forall n \geq n_0 \cdot g(n) \leq cf(n) \right\}$$

- Notación Ω (cota inferior):

$$\Omega(f(n)) = \left\{ g : \mathcal{N} \rightarrow \mathcal{R}^{\geq 0} \mid \exists c \in \mathcal{R}^+, \exists n_0 \in \mathcal{N} \cdot \forall n \geq n_0 \cdot g(n) \geq cf(n) \right\}$$

- Notación Θ (orden exacto):

$$\Theta(f(n)) = \left\{ g : \mathcal{N} \rightarrow \mathcal{R}^{\geq 0} \mid \exists c, d \in \mathcal{R}^+, \exists n_0 \in \mathcal{N} \cdot \forall n \geq n_0 \cdot \right. \\ \left. \cdot cf(n) \leq g(n) \leq df(n) \right\}$$

Notación O

$T(N)$ es $O(F(N))$ si existen las constantes positivas c y N_0 tales que para $N \geq N_0$ se verifica que $T(N) \leq c \cdot F(N)$

Notación Ω (Omega)

$T(N)$ es $\Omega(F(N))$ si existen las constantes positivas c y N_0 tales que para $N \geq N_0$ se verifica que $T(N) \geq c \cdot F(N)$

Notación Θ (Theta)

$T(N)$ es $\Theta(F(N))$ si y solo si $T(N)$ es $O(F(N))$ y $T(N)$ es $\Omega(F(N))$

Nosotros nos centraremos en la notación O grande (big O)

La notación O se utiliza para comparar funciones. Resulta particularmente útil cuando se quiere analizar la complejidad de un algoritmo, en otras palabras, la cantidad de tiempo que le toma a un ordenador ejecutar un programa.

¿Qué quiere decir todo esto?

Básicamente, que una función es siempre menor que otra función (por ejemplo, el tiempo en ejecutar tu programa es menor que x^2) si no tenemos en cuenta los factores constantes (eso es lo que significa la \mathcal{C}) y si no tenemos en cuenta los valores pequeños (eso es lo que significa la N).

¿Por qué no tenemos en cuenta los valores pequeños de N ?

Porque para entradas pequeñas, el tiempo que tarda el programa no es significativo y casi siempre el algoritmo será suficientemente rápido para lo que queremos.

Así, $3N^3 + 5N^2 - 9 = O(N^3)$ no significa que existe una función $O(N^3)$ que es igual a $3N^3 + 5N^2 - 9$.

Debe leerse como:

“ $3N^3 + 5N^2 - 9$ es O-Grande de N^3 ”

que significa:

“ $3N^3 + 5N^2 - 9$ está *asintóticamente* dominada por N^3 ”

Observaciones :

1. La función que figura en el paréntesis es representativa del conjunto.
Se suele colocar la función más sencilla de todo el conjunto que la acote superiormente.
2. Cuando un algoritmo tiene una función de complejidad $t(n) \in O(f(n))$, se dice que la complejidad del algoritmo es O grande de f . Conviene en ese caso que $f(n)$ sea la función más pequeña que acota superiormente al tiempo que utiliza el algoritmo.
3. $t(n) = 3n^2 + 2n + 5 \in O(n^3)$
 $t(n) = n^2 + 1 \in O(n^3)$
 $t(n) = 2n \in O(n^3)$

Propiedades :

1. $f(n) \in O(f(n))$

2. a) $O(f(n)) \subset O(g(n)) \Leftrightarrow f(n) \leq g(n)$

b) $O(f(n)) = O(g(n)) \Leftrightarrow f(n) \leq g(n) \text{ y } g(n) \leq f(n)$

3. Si $t(n) \in O(f(n))$ y $t'(n) \in O(g(n))$

a) $c * t(n) \in O(f(n))$

b) $t(n) + t'(n) \in O(f(n) + g(n)) = O[\max(f(n), g(n))]$

c) $t(n) * t'(n) \in O(f(n)g(n))$

Ej. : $t(n) = 3n^2 + 6n = O(3n^2 + 6n)$

$$= O[\max(3n^2, 6n)] = O(3n^2) = O(n^2)$$

Ej. : $t(n) = 4 \log n + 6n \quad \uparrow \quad = O(\log n + n) = O[\max(\log n, n)] = O(n)$

$4 \log n \in O(\log n)$

$6n \in O(n)$

Ej. : $(n+1)^2 = O(?)$

$$(n+1)^2 = n^2 + 2n + 1$$

$$n^2 = O(n^2) \quad 2n = O(2n) = O(n)$$

$$1 = O(1)$$

$$n^2 + 2n + 1 = O(n^2) + O(n) + O(1)$$

$$= (\max(n^2, 2n, 1)) = O(n^2)$$

Reglas Prácticas para el cálculo de la eficiencia

Los algoritmos bien estructurados combinan las sentencias de alguna de las formas siguientes

1. sentencias sencillas
2. secuencia (;)
3. decisión (if)
4. bucles
5. llamadas a procedimientos (métodos)

Sentencias sencillas

Nos referimos a las sentencias de asignación, entrada/salida, etc. siempre y cuando no trabajen sobre variables estructuradas cuyo tamaño este relacionado con el tamaño N del problema. La inmensa mayoría de las sentencias de un algoritmo requieren un tiempo constante de ejecución, **siendo su complejidad $O(1)$** .

Secuencia (;) $S1 = O(f_1(n))$ y $S2 = O(f_2(n))$

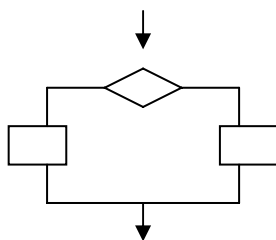
$$S1;S2 = O(f_1(n)+f_2(n)) = O(\max(f_1(n), f_2(n)))$$

La complejidad de una serie de elementos de un programa es del orden de **la suma de las complejidades individuales**, aplicándose las operaciones arriba expuestas.

Decisión (if) *Si B entonces S1 sino S2 finSi*

$$O(\max(f_b(n), f_1(n), f_2(n)))$$

La condición suele ser de $O(1)$, complejidad a sumar **con la peor posible**, bien en la rama THEN, o bien en la rama ELSE. En decisiones múltiples (ELSE IF, SWITCH CASE), se tomara la peor de las ramas.



Bucles

mientras B hacer S finmientras

Si $B; S = O(f_{bs}(n))$ y el número de iteración es función de n $f_{iter}(n)$

El coste del bucle es $O(f_{bs}(n) * f_{iter}(n))$

En los bucles con contador explícito, podemos distinguir dos casos, que el tamaño N forme parte de los límites o que no. Si el bucle se realiza un número fijo de veces, independiente de N , entonces la repetición sólo introduce una constante multiplicativa que puede absorberse.

```
for (int i= 0; i < K; i++) { algo_de_O(1) }      =>  K*O(1) = O(1)
```

Si el tamaño N aparece como límite de iteraciones ...

```
for (int i= 0; i < N; i++) { algo_de_O(1) }      =>  N * O(1) = O(n)
```

```
for (int i= 0; i < N; i++) {  
    for (int j= 0; j < N; j++) {  
        algo_de_O(1)  
    }  
}
```

tendremos $N * N * O(1) = O(n^2)$

```
for (int i= 0; i < N; i++) {  
    for (int j= 0; j < i; j++) {  
        algo_de_O(1)  
    }  
}
```

el bucle exterior se realiza N veces, mientras que el interior se realiza 1, 2, 3, ... N veces respectivamente. En total,

$$1 + 2 + 3 + \dots + N = N*(1+N)/2 \rightarrow O(N^2)$$

A veces aparecen bucles multiplicativos, donde la evolución de la variable de control no es lineal (como en los casos anteriores)

```
c= 1;
while (c < N) {
    algo_de_O(1)
    c= 2*c;
}
```

El valor inicial de "c" es 1, siendo " 2^k " al cabo de "k" iteraciones. El número de iteraciones es tal que $2^k \geq N \Rightarrow k = \lceil \log_2(N) \rceil$ [el entero inmediato superior] y, por tanto, la complejidad del bucle es $O(\log n)$.

```
c= N;
while (c > 1) {
    algo_de_O(1)
    c= c / 2;
}
```

$N, N/2, N/4, N/8, N/16, \dots$ $N/2^k > 1 \Rightarrow N > 2^k \Rightarrow k = \lceil \log_2(N) \rceil$

Un razonamiento análogo que nos lleva a $\log_2(N)$ iteraciones y, por tanto, a un orden $O(\log n)$ de complejidad.

```
for (int i= 0; i < N; i++) {
    c= i;
    while (c > 0) {
        algo_de_O(1)
        c= c/2;
    }
}
```

tenemos un bucle interno de orden $O(\log n)$ que se ejecuta N veces, luego el conjunto es de orden $O(n \log n)$

Llamadas a procedimiento (método)

La complejidad de llamar a un procedimiento viene dada por la complejidad del contenido del procedimiento en sí. El coste de llamar no es sino una constante que podemos obviar inmediatamente dentro de nuestros análisis asintóticos. El cálculo de la complejidad asociada a un procedimiento puede complicarse notablemente si se trata de procedimientos recursivos. Es fácil que tengamos que aplicar técnicas propias de la matemática discreta.

Un ejemplo sencillo:

$$\sum_{i=1}^n i^3 = 1^3 + 2^3 + 3^3 + \dots + n^3$$

```
sumaParcial=0
desde i ← 1 hasta n hacer
    sumaParcial = sumaParcial + i*i*i;
fdesde
```

El resultado seria: $1 + 2n + 2 + 4n = 6n + 4 = O(n)$ LINEAL

Problema de la Subsecuencia Máxima

Dada la secuencia de enteros (posiblemente negativos) A_1, A_2, \dots, A_N , encontrar (e identificar la subsecuencia correspondiente) el valor máximo de $\sum_{k=i}^j A_k$. Cuando todos los enteros son negativos entenderemos que la subsecuencia de suma máxima es la vacía, siendo su suma cero.

$\{-2, \underline{11}, \underline{-4}, \underline{13}, -5, -2\}$ -----> Subsecuencia suma 20

$\{1, -3, \underline{4}, \underline{-2}, \underline{-1}, 6\}$ -----> Subsecuencia suma 7

Algoritmo cúbico

```
/**
 * Algoritmo cúbico para la subsecuencia de suma máxima.
 * secIni y secFin representan la secuencia mejor actual.
 */
public static int subsecuenciaSumaMaximaCubico(int[] a) {
    int sumaMax = 0;

    for (int i = 0; i < a.length; i++)
        for (int j = i; j < a.length; j++) {
            int sumaActual = 0;

            for (int k = i; k <= j; k++)
                sumaActual += a[k];

            if (sumaActual > sumaMax) {
                sumaMax = sumaActual;
                secIni = i;
                secFin = j;
            }
        }

    return sumaMax;
}
```

$$\sum_{k=i}^j a_k = a_j + \sum_{k=i}^{j-1} a_k$$

Algoritmo cuadrático

```

/**
 * Algoritmo cuadrático para la subsecuencia de suma máxima.
 * secIni y secFin representan la secuencia mejor actual.
 */

public static int subsecuenciaSumaMaximaCuadratico(int[] a) {
    int sumaMax = 0;

    for (int i = 0; i <= a.length; i++) {
        int estaSuma = 0;

        for (int j = i; j < a.length; j++) {
            estaSuma = estaSuma + a[j];

            if (estaSuma > sumaMax) {
                sumaMax = estaSuma;
                secIni = i;
                secFin = j;
            }
        }
    }
    return sumaMax;
}

```

Algoritmo lineal

Sea $A_{i,j}$ una subsecuencia cualquiera con $S_{i,j} < 0$. Si $q > j$ entonces $A_{i,j}$ no es subsecuencia de suma máxima.

i	j	$j+1$	q
< 0		$S_{j+1, q}$	
$< S_{j+1, q}$			


$$\begin{array}{l|l} S_{i,q} = S_{i,j} + S_{j+1,q} & \implies S_{i,q} < S_{j+1,q} \\ S_{i,j} < 0 & \end{array}$$

Para cada i , sea $A_{i,j}$ la **primera** subsecuencia que satisfaga $S_{i,j} < 0$.

Entonces para cualquier p, q , $i \leq p \leq j$ y $p \leq q$

$A_{p,q}$ o bien no es subsecuencia máxima

o bien es igual a una subsecuencia máxima ya considerada

i	j	$j+1$	q	i	q	j
 $S_{i, q}$				$S_{i, q}$		
≥ 0		$\leq S_{i, q}$		≥ 0		$\leq S_{i, q}$
$p-1$	p			$p-1$	p	

Ejemplos:

$\{1, 2, -4, \underline{3, 1}\}$

< 0

$\{\underline{1, 2, 3}, -7, 3, 1\}$

< 0

```

/**
 * Algoritmo lineal para la subsecuencia de suma máxima.
 * secIni y secFin representan la secuencia mejor actual.
 */
public static int subsecuenciaSumaMaximaLineal(int[] a) {
    int sumaMax = 0;
    int sumaActual = 0;

    for (int i = 0, j = 0; j < a.length; j++) {
        sumaActual += a[j];

        if (sumaActual > sumaMax) {
            sumaMax = sumaActual;
            secIni = i;
            secFin = j;
        } else if (sumaActual < 0) {
            i = j + 1;
            sumaActual = 0;
        }
    }

    return sumaMax;
}

```


Problemas P, NP y NP-completos

http://es.wikipedia.org/wiki/Tiempo_polin%C3%B3mico
<http://es.wikipedia.org/wiki/NP-completo>

Cuando nos enfrentamos a un problema concreto, habrá una serie de algoritmos aplicables.

El **orden de complejidad** de un problema es el del mejor algoritmo que se conozca para resolverlo.

Así se clasifican los problemas

Clase P

Los algoritmos de complejidad polinómica se dice que son tratables en el sentido de que suelen ser abordables en la práctica. Los problemas para los que se conocen algoritmos con esta complejidad se dice que forman la clase P. **Aquellos problemas para los que la mejor solución que se conoce es de complejidad superior a la polinómica, se dice que son problemas intratables.** Sería muy interesante encontrar alguna solución polinómica (o mejor) que permitiera abordarlos.

Clase NP

Algunos de estos problemas intratables pueden caracterizarse por el curioso hecho de que puede aplicarse un algoritmo polinómico para comprobar si una posible solución es válida o no. Esta característica lleva a un método de resolución no determinista consistente en aplicar heurísticos para obtener soluciones hipotéticas que se van desestimando (o aceptando) a ritmo polinómico. Los problemas de esta clase se denominan NP (la N de no-deterministas y la P de polinómicos).

Clase NP-completos

Se conoce una amplia variedad de problemas de tipo NP, de los cuales destacan algunos de ellos de extrema complejidad. Gráficamente podemos decir que algunos problemas se hayan en la "frontera externa" de la clase NP. Son problemas NP, y son los peores problemas posibles de clase NP. Estos problemas se caracterizan por ser todos "iguales" en el sentido de que si se descubriera una solución P para alguno de ellos, esta solución sería fácilmente aplicable a todos ellos. Es más, si se descubriera una solución para los problemas NP-completos, esta sería aplicable a todos los problemas NP y, por tanto, la clase NP desaparecería del mundo científico al carecerse de problemas de ese tipo. Realmente, tras años de búsqueda exhaustiva de dicha solución, es hecho ampliamente aceptado que no debe existir, aunque nadie ha demostrado, todavía, la imposibilidad de su existencia.

El problema de la suma de subconjuntos es un problema importante en la teoría de la complejidad y en la criptografía. El problema es este: dado un conjunto de enteros, ¿existe algún subconjunto cuya suma sea exactamente cero? Por ejemplo, dado el conjunto $\{-7, -3, -2, 5, 8\}$, la respuesta es SI, porque el subconjunto $\{-3, -2, 5\}$ suma cero. Este problema es probablemente el más simple de explicar de los problemas NP-completos.

Un problema equivalente es: dado un conjunto de enteros y un entero s , ¿existe algún subconjunto cuya suma sea s ? La suma de subconjuntos también puede verse como un caso especial del problema de la mochila. Se tienen n objetos y una mochila. El objeto i tiene peso p_i y la inclusión del objeto i en la mochila produce un beneficio b_i . El objetivo es llenar la mochila, de capacidad C , de manera que se maximice el beneficio.

Algoritmo de tiempo exponencial. El algoritmo más inocente verificaría todos los posibles subconjuntos de N números y, para cada uno de ellos, compararía la suma al total buscado. El tiempo de ejecución es de orden $O(2^N N)$, dado que hay 2^N subconjuntos y, para verificar cada subconjunto, tenemos que sumar N elementos.

El **problema del viajante** (también conocido como **problema del viajante de comercio** o por sus siglas en inglés: **TSP**) es uno de los problemas más famosos (y quizás el mejor estudiado) en el campo de la optimización combinatoria computacional. A pesar de la aparente sencillez de su planteamiento, el TSP es uno de los más complejos de resolver y existen demostraciones que equiparan la complejidad de su solución a la de otros problemas aparentemente mucho más complejos que han retado a los matemáticos desde hace siglos.

Definición: Sean N ciudades de un territorio. La distancia entre cada ciudad viene dada por la matriz $D: N \times N$, donde $d[x,y]$ representa la distancia que hay entre la ciudad X y la ciudad Y . El objetivo es encontrar una ruta que, comenzando y terminando en una ciudad concreta, pase una sola vez por cada una de las ciudades y minimice la distancia recorrida por el viajante. Es decir, encontrar una permutación $P = \{c_0, c_2, \dots, c_{n-1}\}$ tal que

$$d_P = \sum_{i=0}^{N-1} d[c_i, c_{i+1 \bmod(N)}]$$

sea mínimo.

El TSP está entre los problemas denominados NP-Completo, esto es, los problemas que no se pueden resolver en tiempo polinomial en función del tamaño de la entrada (en este caso el número N de ciudades que el viajante debe recorrer).