Programación 2

APUNTES TEÓRICOS

Unidad 1: Introducción

Unidad 1: Introducción

Nuestra materia se centra en el “buen” diseño y elección de estructuras de datos y algoritmos. Dicho de manera simple, una ***estructura de datos*** es una forma sistemática de organizar y dar acceso a los datos, y un ***algoritmo*** es un procedimiento detallado para hacer alguna tarea en una cantidad finita de tiempo. Pero para calificar como “buenas” algunas estructuras de datos y algoritmos debemos contar con alguna forma precisa de analizarlas.

1- Complejidad Algorítmica

1.1 Algoritmia

Dimos ya una definición de algoritmo. Otras pueden ser:

- Un *algoritmo* es una secuencia finita de instrucciones que especifican cómo se resuelve un problema.
- Un *algoritmo* es cualquier procedimiento computacional bien definido, junto con un conjunto especificado de datos, que produce un valor o conjunto de valores como salida.

Esta última definición resalta que un algoritmo debe producir una salida válida para todas las entradas permitidas.

Pero, ¿qué es la algoritmia?

Muchas veces contamos con muchos algoritmos diferentes que resuelven un mismo problema, entonces surge la pregunta de ¿cuál usar? Para responder a esta pregunta está la algoritmia.

La ***algoritmia*** es la ciencia que estudia técnicas para diseñar algoritmos eficientes y evaluar la eficiencia de un algoritmo que ya existe.

Un algoritmo es mucho mejor cuantos menos recursos consuma, eso es evidente, pero se deben tener en cuenta otros factores antes de establecer la conveniencia o no de este. Por ejemplo, la facilidad para programarlo, la facilidad de entenderlo, la robustez, etc., y por supuesto, el criterio de eficiencia buscado: relación entre recursos consumidos y productos conseguidos.

De forma general podemos decir que los recursos que consume un algoritmo dependen de factores internos y externos.

- Factores internos
    - Tamaño de los datos de entrada
    - Naturaleza de los datos de entrada (peor caso, caso promedio o mejor caso)
- Factores externos
    - El ordenador donde lo ejecutamos: procesador, memoria, etc..
    - El lenguaje de programación y el compilador usado.
    - La implementación que haga el programador del algoritmo, en particular de las estructuras de datos usadas.

De estos dos factores los externos no aportan información sobre el algoritmo, porque no pueden ser controlados por el usuario. Mientras que los factores internos, el tamaño y contenido de los datos de entrada, si pueden ser controlados por el usuario, por lo que aportan la información necesaria para poder realizar el análisis del algoritmo.

La ***algoritmia*** es la ciencia que permite evaluar el efecto de los diferentes factores internos sobre los algoritmos, de tal modo que sea posible seleccionar el que más se ajuste a nuestras circunstancias, además de indicar la forma de diseñar un nuevo algoritmo para una tarea concreta.

1.1.1 Fases en el desarrollo de un algoritmo:

Cualquier consideración sobre el desarrollo de un algoritmo con cierta complejidad conceptual debe comenzar aislando cada una de sus fases componentes. Se pueden identificar las siguientes etapas de una manera más o menos general.

1. *Análisis del problema:* Se refiere a la etapa en la cual el programador toma conocimiento del problema antes de proceder a desarrollar una solución. Un análisis inadecuado puede conducir a una mala interpretación del enunciado del problema. Los errores en esta etapa son, con frecuencia, difíciles de detectar y consumen mucho tiempo al arrastrarse hacia fases posteriores.
2. *Desarrollo de la solución:* Una vez definido el problema y teniendo cierta idea de cómo resolverlo, se puede utilizar alguna de las técnicas conocidas de diseño de algoritmos. Muchas veces, debido a la complejidad interna de los problemas a resolver, se puede ir describiendo la solución como una secuencia de pasos bastante generales (esto puede hacerse en lenguaje natural), que cada vez se van detallando o refinando más hasta obtener una solución. En esta etapa también se empiezan a tomar decisiones sobre las estructuras de datos que se utilizarán para representar los datos del problema.
3. *Codificación de la solución:* Considerando que la solución algorítmica ha sido bien definida, este proceso resulta casi completamente mecánico. Utilizando las reglas sintácticas y semánticas de un lenguaje de programación, el algoritmo se escribe teniendo en cuenta también ciertos criterios de estilo o estructura.
4. *Verificación y análisis de la solución:* Aquí debemos tener en cuenta tres aspectos claves del algoritmo: su corrección o eficiencia, su claridad o legibilidad y su eficacia o uso óptimo de los recursos.

1.1.2 Verificación y Análisis de Algoritmos

Es habitual considerar como objetivos de la programación los de CORRECCIÓN, CLARIDAD y EFICIENCIA. De acuerdo con ello, para evaluar un algoritmo o programa podemos tener en cuenta estos tres aspectos:

- ✓ CORRECCIÓN o EFICACIA: Si el algoritmo cumple con las especificaciones dadas y funciona para todas las entradas permitidas.
- ✓ CLARIDAD: Si el algoritmo es fácil de codificar y depurar o entender, incluso por personas distintas de su diseñador.
- ✓ EFICIENCIA: Si el algoritmo resuelve el problema usando una cantidad razonable de recursos.

Los objetivos anteriores resultan a veces contradictorios. Por supuesto, el objetivo de corrección es fundamental, pero los otros dos han de satisfacerse a veces estableciendo un compromiso entre ellos. En muchos casos podemos tener un algoritmo complejo pero muy eficiente, frente a un algoritmo sencillo pero que use muchos recursos.

El análisis de la corrección de un algoritmo se denomina *verificación*. La claridad de la implementación es difícil de medir de manera objetiva, de manera que no hay técnicas formales para ello. El análisis de eficiencia conduce a las medidas de complejidad algorítmica.

1.2 Análisis de la Eficiencia de Algoritmos

Analizar la *eficiencia* de un algoritmo supone determinar la cantidad de recursos informáticos consumidos por el algoritmo. Los recursos que habitualmente se contabilizan incluyen la cantidad de memoria y la cantidad de tiempo de cómputo requeridos por el algoritmo.

1.2.1 Tiempo de ejecución y uso de memoria

Nuestro análisis de los algoritmos y estructuras de datos se centrará principalmente en consideraciones de eficiencia. Este análisis se realiza considerando la cantidad de recursos que un algoritmo consume, como una función del tamaño de la entrada del algoritmo.

La *memoria* consumida por un algoritmo es la suma de dos componentes: una *parte fija* que es independiente de sus datos y resultados, y una *parte variable* que depende del tamaño del problema considerado.

La *memoria fija* consiste en el espacio reservado para el código del algoritmo, variables simples, constantes, estructuras de datos de tamaño fijo, etc.

La *memoria variable o dinámica* es el espacio necesitado por las variables cuyo tamaño dependa de la instancia del problema que está siendo resuelta, del espacio requerido (si es preciso) para las diferentes activaciones de los subprogramas recursivos, etc..

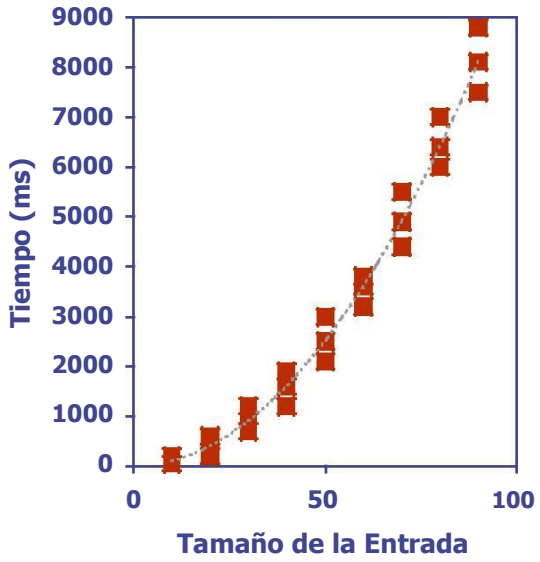
El *tiempo de ejecución* para un programa depende de factores como los datos de entrada, la calidad del código generado por el compilador, la naturaleza y velocidad de las instrucciones de máquina empleadas en la ejecución y la complejidad en el tiempo del algoritmo utilizado en el programa.

En muchos casos, el espacio y el tiempo requeridos por un algoritmo están inversamente relacionados. Esto es, podemos ser capaces de reducir los requisitos de espacio en memoria con un incremento del tiempo de ejecución, o a la inversa, reducir los requisitos de tiempo incrementando el espacio en memoria.

1.2.2 Análisis de un Algoritmo

Si se ha implementado el algoritmo se puede estudiar su tiempo de ejecución, corriéndolo con diversas entradas de prueba y anotando el tiempo real que se uso en cada ejecución. Esas medidas se pueden tomar en forma exacta, usando llamadas de sistema que se incorporan al lenguaje o al sistema operativo para el que se escribió el algoritmo (por ejemplo usando el método System.currentTimeMillis()). En general, lo que interesa es determinar la dependencia entre el tiempo de ejecución y el tamaño de la entrada.

Para determinarlo, se pueden hacer algunos experimentos con muchas entradas de prueba sistintas, de varios tamaños; y luego visualizar los resultados de esos experimentos, graficando el funcionamiento de cada corrida del algoritmo como un punto con abscisa x igual al tamaño de la entrada, n, y cuya ordenada sea el tiempo de ejecución, t.



Para que esto tenga sentido, se requiere que para este análisis se escojan entradas de muestra buenas, y que se prueben las suficientes como para establecer afirmaciones estadísticas acerca del algoritmo.

El tiempo de ejecución de un programa depende, en la mayoría de los casos, de los datos particulares con los que opera. Esto quiere decir que la eficiencia de un programa debe establecerse no como una magnitud fija para cada programa, sino como una función que nos dé el tiempo de ejecución para cada tamaño o cantidad de los datos que deba procesar.

Esta idea nos lleva, por su parte, a la necesidad de establecer previamente una medida del tamaño de los datos o *tamaño del problema*, para, en función de ella, establecer la medida de la eficiencia del programa que los procesa.

El tamaño del problema se puede expresar bien por la cantidad de datos a tratar, o bien por los valores particulares de los datos. Por ejemplo, para un programa que obtenga el valor medio o la suma de una serie de datos, la cantidad de números a sumar o promediar es una buena medida del tamaño del problema. En cambio, para un programa que calcule una potencia de un número, el tamaño significativo puede ser el valor del exponente más que el del número original.

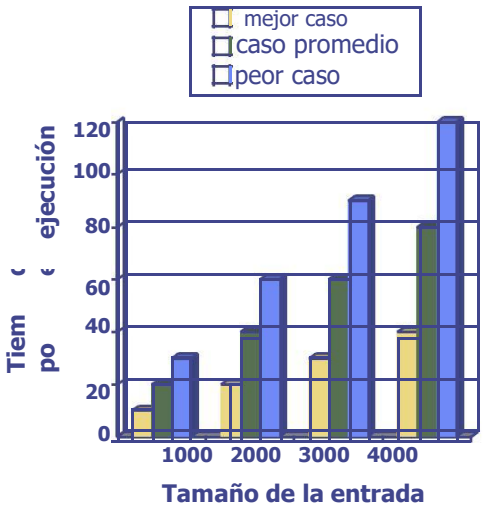
La complejidad algorítmica en tiempo se mide como la función que da el tiempo de ejecución según el tamaño del problema. Denotamos a esa función como *T(n).*

$$T(n)= cf(n)$$

Cuando decimos que la complejidad de un algoritmo es $T(n) = cf(n)$ no estamos haciendo referencia a una determinada unidad de tiempo, sino a que el tiempo de ejecución del algoritmo es proporcional a cierta función *f(n)*, donde la constante de proporcionalidad *c* depende del computador.

En general se tienen tres criterios para calcular *T(n):*

- ✓ Considerar *T(n)* como el tiempo para el *peor caso* de los posibles.
- ✓ Considerar *T(n)* como el tiempo para el *mejor caso* de los posibles.
- ✓ Hacer *T(n)* igual al *tiempo medio* de todos los casos posibles.



A primera vista, es mejor el caso promedio pero hay que tener en cuenta que no todos los casos posibles son equiparables. Si no se tiene en cuenta los efectos probabilísticos en el cálculo de *T(n)*, esta medida puede resultar errónea. La primera medida también puede no resultar totalmente correcta pero es más segura y fácil de evaluar, por ello nos centrarnos en el tiempo de ejecución en el peor caso. Además, es crucial en aplicaciones como juegos, finanzas y robótica.

Si bien los estudios experimentales de los tiempos de ejecución tienen utilidad, tienen también tres limitaciones importantes:

- ➢ Es necesario implementar y ejecutar un algoritmo, para estudiar en forma experimental su tiempo de ejecución.
- ➢ Es difícil comparar la eficiencia de dos algoritmos, a menos que se hayan hecho experimentos para conocer sus tiempos de ejecución en los mismos ambientes de hardware, de software y con las mismas muestras de entradas.
- ➢ Solo se pueden hacer experimentos con un conjunto limitado de entradas de prueba, y los experimentos pueden no ser indicativos del tiempo de ejecución con otras entradas que no se hayan incluido en el experimento.

Por esto adoptamos una metodología general para analizar el tiempo de ejecución de algoritmos que:

- ➢ Considera todas las entradas posibles.
- ➢ Permite evaluar la eficiencia relativa de dos algoritmos cualesquiera en una forma que es independiente del ambiente del hardware y del software.
- ➢ Puede ejecutarse estudiando una descripción de alto nivel del algoritmo, sin implementarlo o experimentar con él en realidad.

Por tanto, para la evaluación de la complejidad en tiempo se hace referencia a una máquina virtual hipotética con tiempo de ejecución estándar, donde consideramos que cada operación elemental del lenguaje de programación: suma, resta, división, multiplicación, escritura, asignación, decisión según condición, llamada a un método (función o procedimiento), regreso de un método, etc., dura una unidad de tiempo.

Con esta simplificación, el análisis de la eficiencia de un programa se centra en establecer cuántas instrucciones se ejecutan en total, dependiendo del tamaño o cantidad de los datos a procesar.

Usando el criterio del *peor caso* en análisis de algoritmos, debemos tener en cuenta las siguientes reglas:

Regla 1) La complejidad de un esquema secuencial es la suma de las complejidades de sus acciones componentes.

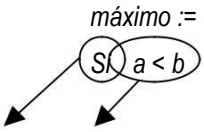
Regla 2) La complejidad de un esquema de selección equivale a la de la alternativa más compleja, es decir, de ejecución más larga, más la complejidad de la evaluación de la condición de selección.

Regla 3) La complejidad de un esquema de iteración se obtiene sumando la serie correspondiente al número de instrucciones en las repeticiones sucesivas.

Veamos cómo es este análisis en algunos casos concretos. Tomemos como ejemplo el siguiente fragmento de programa que obtiene el máximo de dos números:

```
máximo := a;
Si   a < b
        entonces    máximo: = b;
```

El esquema global es una secuencia de dos acciones: una asignación, seguida de un esquema condicional (Si). Anotaremos el programa con el número correspondiente a cada sentencia, para lo cual contaremos el número de operadores (+,-,*,<,:=,etc.) y decisiones (Si, Mientras, etc.). Aplicando las reglas para los esquemas tendremos:




```
máximo := a;                                  1
                                              2
                                              3        (regla 2)
1  +  1      entonces   máximo := b;          1
                                  Total = 4    (regla 1)
```

T(n) = 4

La complejidad en este caso es fija y no depende de una medida de tamaño del problema.

A continuación, analizaremos un bucle que obtiene en *f* el factorial de un número *k.* Anotaremos el programa con el número de instrucciones de cada sentencia.

```
k:= 1;                          1
f:= 1;                          1
Mientras  k<n  hacer            2
        k:= k + 1;              2      6(n-1)     (Regla3)
        f:= f * 1;              2
fin-Mientras;
                          Total =6n – 4     (Regla1)
```

T(n) = 6n - 4

Para calcular el número de instrucciones del bucle se ha multiplicado el número de instrucciones en cada repetición por el número de repeticiones.

La complejidad aparece expresada en función del valor de *n,* que en este caso resulta una medida natural del tamaño del problema

Considérense los siguientes fragmentos de algoritmos.

1) *x : = x + 1;* T(n)= 2

2) *Para i de 1 a n hacer* T(n)= 2n
   x : = x + 1;
   fin-Para;

3) *Para i de 1 a n hacer* $T(n)= 2n^2$
Para j de 1 a n hacer
x : = x + 1 ;
fin-Para;
fin-Para;

En el punto número (1), la sentencia de asignación se ejecuta sólo una vez; en la (2), la misma sentencia de ejecuta *n* veces; en la (3), se ejecuta n^2 veces (por existir dos bucles anidados en *n* repeticiones cada uno). Por ejemplo, para n = 10 corresponderían a 1, 10 y 100 repeticiones, respectivamente. Estos valores permiten diferenciar los distintos órdenes de complejidad de cada uno de los algoritmos.

Si hay llamadas a métodos (funciones o procedimientos), éstas deben ser analizadas primero y sumadas luego. Por ejemplo:

cont:= 0;	1
x : = 1;	1
Mientras x < n hacer	2
x:= x +1;	2
cont:= cont + Proceso(x, n);	2 + 1 + T(Proceso(x, n)) } n veces
Fin_mientras	suponemos T()= 2n

$T(n)= 2 + 7n + 2n^2$

Si hay procedimientos recursivos, hay varias opciones. Si la recursión es un ciclo *for* ligeramente disfrazado, el análisis suele ser trivial. Por ejemplo, la siguiente función es un ciclo sencillo:

```
Función factorial (n: entero): entero;
  Si n=0 o n=1
        entonces
        sino
Fin_función          factorial:= 1;
                     factorial:= n * factorial (n-1);
```

Este ejemplo es realmente un uso ineficiente de la recursión. Cuando la recursión se usa adecuadamente, es difícil convertirla en una estructura iterativa sencilla. En este caso, el análisis implicará una relación de recurrencia que hay que resolver, y el tiempo de ejecución es logarítmico.

1.2.3 Comportamiento asintótico

En los análisis de eficiencia o complejidad se considera muy importante la manera en que la función de complejidad va aumentando con el tamaño del problema. *Lo que interesa es la forma de crecimiento del tiempo de ejecución, y no tanto el tiempo particular empleado*.

Como ejemplo podemos comparar dos programas, uno que tarda un tiempo 100n en resolver un problema de tamaño n, y otro que tarda un tiempo n^2. La comparación puede hacerse escribiendo una tabla con los tiempos de cada uno para diferentes tamaños del problema.

Tamaño	100n	n^2
1	100	1
2	200	4
3	300	9
10	1.000	100
100	10.000	10.000
1000	100.000	1.000.000

Al principio de la tabla el primer programa parece menos eficiente que el segundo, ya que tarda mucho más tiempo, pero a medida que aumenta el tamaño del problema ambos programas llegan a tardar lo mismo (para tamaño 100) y a partir de ahí el segundo programa demuestra ser mucho menos eficiente que el primero.

La menor eficiencia del segundo programa para tamaños grandes del problema no cambia por el hecho de que se modifique el coeficiente multiplicador; es decir, si el primer programa tardase 10

veces más (1000n en lugar de 100n), acabaría igualmente por resultar mejor que el segundo a partir de un cierto tamaño del problema.

Lo que importa es la forma de la función, que en el primer caso es *lineal* y en el segundo es *cuadrática*. La forma en que crece la función para tamaños grandes se dice que es de **crecimiento asintótico**, y se representa mediante la notación o grande:

$$O(g(n))$$

Siendo *n* el tamaño del problema, *g* la forma o función de crecimiento asintótico, y O el **orden de crecimiento**.

Sea una función *f(n)*= tiempo de ejecución del algoritmo, decimos que dicha función es *O(g(n))* si se cumple que existe una constante $c > 0$ y un número n_0 (que no depende de n) tales que:

$$f(n) / g(n) \leq c \qquad \text{para todo} \quad n_0 \leq n$$

Ejemplos:

- Si $T(n)= 4n^3 + 2n^2$ tenemos que T(n) es O(n) ya que tomando $c = 5$ y $n_0=2$ obtenemos:

  $$(4n^3 + 2n^2) / n^3 \leq 5 \text{ para todo } n_0 \leq n$$

  entonces $O(n^3)$

1.2.4 Funciones de complejidad en tiempo más usuales

Las funciones de complejidad algorítmica más usuales, ordenadas de mayor a menor eficiencia son:

O (1)	*complejidad constante*	Es la situación más deseada pero difícil de encontrar en la mayoría de los problemas.
O(*log n*)	*complejidad logarítmica*	Esta complejidad suele aparecer en determinados algoritmos recursivos, en aquellos en los que el uso de la recursión es aconsejable. Es una complejidad óptima.
O(n)	*complejidad lineal*	Es en general, una complejidad buena y bastante usual. Suele aparecer en la evaluación de un bucle simple cuando la complejidad de las operaciones interiores es constante o en algoritmos con recursión estructural (simples de pasar a iterativos).
O(n log n)	*complejidad logarítmica*	También aparecen en algoritmos con recursión no estructural (por ejemplo, ordenación por el método rápido) y se considera una complejidad buena.
O(n^2)	*complejidad cuadrática*	Aparece en bucles dobles o similares.
O(n^3)	*complejidad cúbica*	Aparece en bucles triples. Para un valor grande de *n* empieza a crecer en exceso.
O(n^k)	*complejidad polinómica (k > 3)*	Si *k* crece, la complejidad del programa es bastante mala.
O(2^n)	*complejidad exponencial*	Debe evitarse en la medida de lo posible. Puede aparecer en un subprograma recursivo que contenga dos o tres llamadas internas. En programas donde aparece esta complejidad suele hablarse de *explosión combinatoria.*

Téngase en cuenta que aunque un algoritmo posea una eficiencia que en términos absolutos pueda calificarse como “buena”, esto no significa que lo sea en términos relativos, pues puede ocurrir que la naturaleza del problema permita una solución más eficiente (esto es, de menor complejidad).

La tabla siguiente muestra el crecimiento de las diferentes funciones según el parámetro *n*.

n	log n	n log n	n2	n3	2n
1	0	0	1	1	2
5	0.7	3	25	125	32
10	1.0	10	100	1000	1024
20	1.3	26	400	8000	1048576
50	1.7	85	2500	25000	$1125900000x10^{6}$
100	2.0	200	10000	1000000	1267651000x1021
200	2.3	460	40000	8000000	-
500	2.7	1349	250000	125000000	-
1000	3.0	3000	1000000	1000000000	-

1.2.5 Eficiencia versus claridad

Las grandes velocidades de los microprocesadores actuales, junto con el aumento considerable de las memorias centrales, y los bajos precios de los mismos, hacen que los recursos típicos tiempo y almacenamiento no sean hoy en día parámetros fundamentales para la medida de la eficiencia de un programa.

Por otra parte es preciso tener en cuenta que a veces los cambios para mejorar la eficiencia de un programa pueden traducirse en menor legibilidad o menor claridad; lo cual hace que una modificación o agregado luego sea más dificultoso. En programas grandes la legibilidad suele ser más importante que el ahorro en tiempo y en almacenamiento en memoria. Como norma general, cuando la elección en un programa se debe hacer entre claridad y eficiencia, generalmente se elegirá la claridad del programa.

1.3 Elección de un Algoritmo

- ✓ Para resolver un problema debemos encontrar un algoritmo que tenga una complejidad lo más pequeña posible dentro de la tabla anterior. Si dudamos entre dos algoritmos de la misma complejidad podemos calcular la complejidad en media o hacer una prueba en el ordenador.
- ✓ Si un programa va a ser usado un número pequeño de veces se puede decir (salvo que utilice una cantidad excesiva de recursos) que el costo de escribir el programa y de depurarlo domina. Por tanto, se optará por el algoritmo más sencillo de realizar correctamente.
- ✓ Si el programa va a ser utilizado con una cantidad de datos relativamente pequeña, debemos darnos cuenta que la notación O() no es muy significativa. En este caso, si es posible, se puede acudir al tiempo medio o a una comprobación empírica del comportamiento de los algoritmos.
- ✓ Un algoritmo eficiente pero excesivamente complicado puede no ser muy útil, sobre todo si el algoritmo va a ser utilizado por otra persona que no sea el diseñador.
- ✓ Los accesos a memorias secundarias suelen ser muy lentos. Deben, por tanto, reducirse a las imprescindibles.
- ✓ En algoritmos de análisis numéricos en problemas matemáticos puede se necesario usar algoritmos más complejos para satisfacer la exigencia de precisión requerida.

1.4 Problemas tratables e intratables

El hecho de que un problema sea resoluble en teoría no quiere decir que se pueda siempre resolver en la práctica. Desde un punto de vista pragmático, sólo se puede considerar resuelto un problema si se encuentra un algoritmo suficientemente eficiente para obtener la solución.

Problema Resoluble:

En teoría --------> Existe algoritmo

En la práctica ---> Existe algoritmo y es eficiente

En la mayoría de los casos lo que puede ocurrir es que un determinado problema sólo sea resoluble en la práctica para tamaños pequeños de los datos. Esto ocurre con aquellos problemas para los que sólo se conocen algoritmos de complejidad elevada.

Los problemas resolubles en teoría, pero no en la práctica por la ineficiencia de los algoritmos conocidos, se denomina problemas intratables. Los textos de complejidad algorítmica establecen normalmente como límite de eficiencia de los problemas tratables el correspondiente a un

crecimiento asintótico de tipo potencia de exponente constante $O(n^k)$. Este crecimiento se da en los algoritmos cuya función de coste es de tipo polinómico. Los problemas que pueden resolverse con algoritmos de complejidad polinómica se consideran tratables.

Los problemas para los que sólo existen algoritmos menos eficientes que los de complejidad polinómica (por ejemplo, exponencial O(kn)) se consideran problemas intratables