Coste temporal

En muchas ocasiones hay que decidir entre varios algoritmos disponibles para realizar una tarea. Un posible criterio es el tiempo de ejecución. Interesa conocer el tiempo que tardaría en ejecutarse el algoritmo en función del tamaño de los casos del problema a resolver.

Eficiencia de un algoritmo.

Cuando se intenta resolver un problema es posible obtener varios algoritmos que lo consiguen, y hay que elegir uno de ellos. En principio se podrían seguir estos dos criterios:

- Algoritmos fáciles de entender, escribir y depurar -si fuese necesario-.
- Algoritmos eficientes en cuanto tiempo de ejecución, y en general, en cuanto al uso de los recursos del computador.

Se debe elegir el primer tipo de algoritmo si éste se va a utilizar pocas veces, ya que es posible que en estos casos el tiempo de escritura del algoritmo sobrepase muy por encima el tiempo de ejecución del mismo. Ahora bien, si se trata de un algoritmo que va a ser empleado en varias ocasiones, y cada una de estas ejecuciones va a requerir de un tiempo "*grande"* para finalizar, entonces se debe elegir un algoritmo de los del segundo tipo.

- si un programa se va a ejecutar muy pocas veces, los costes de codificación y depuración son los que más importan, relegando la eficiencia a un papel secundario.
- si a un programa se le prevé larga vida, hay que pensar que le tocará mantenerlo a otra persona y, por tanto, conviene tener en cuenta su legibilidad, incluso a costa de la complejidad de los algoritmos empleados.
- si se puede garantizar que un programa sólo va a trabajar sobre datos pequeños (valores bajos de N), el orden de complejidad del algoritmo que se use suele ser irrelevante.

La eficiencia de un algoritmo tiene que ver con la cantidad de recursos que el algoritmo necesita. Estos recursos son básicamente:

- tiempo de ejecución
- espacio de almacenamiento

Hay dos métodos para determinar la eficiencia temporal de un algoritmo: enfoque empírico y enfoque teórico.

Enfoque empírico

Consiste en programar los algoritmos y probarlos en un ordenador, midiendo el tiempo que consumen y el espacio que ocupan. Por ejemplo, la siguiente tabla muestra la eficiencia empírica de algunos algoritmos de ordenación de vectores. Los encabezados de las columnas se refieren al número de elementos a ordenar y el dato de cada celda indica el tiempo de ejecución en segundos para cada método.

	10.000	50.000	100.000	500.000
Inserción	0,34	9,38	37,82	918
Selección	0,70	18,19	73,65	1740
Burbuja	0,88	22,24	98,27	2068
Shell Sort	0,01	0,04	0,06	0,32
Quick Sort	0,01	0,02	0,04	0,23
Heap Sort	0,01	0,02	0,05	0,27

Enfoque teórico

Trata de calcular matemáticamente la cantidad de recursos que consume el algoritmo en función del tamaño del problema. El tamaño de un problema es cualquier parámetro en función del cual se puede expresar la complejidad del problema. La siguiente tabla muestra algunos ejemplos de problemas y de lo que significa el tamaño del problema para cada uno de ellos.

Problema	**Tamaño del problema**
Ordenación de un array	nº elementos del array
Búsqueda de un elemento en un array bidimensional	nº elementos del array
Cálculo del factorial de un número	Valor del número

Generalmente el tamaño del problema guarda relación con el volumen de los datos de entrada a tratar.

Ventajas del enfoque teórico sobre el empírico

El enfoque empírico presenta varias ventajas sobre el empírico:

- No depende del ordenador utilizado ni del lenguaje de programación en el que se programe el algoritmo.
- Como la eficiencia se obtiene en función del tamaño del problema, se puede determinar para cualquier tamaño. Si se estudia de forma empírica sólo puede hacerse para determinados tamaños de problema y hay algoritmos que se comportan de manera más eficiente que otros a partir de ciertos tamaños.

Medida del tiempo de ejecución de un algoritmo.

Para estimar el tiempo que puede emplear un algoritmo en finalizar su tarea se puede hacer uso de una serie de técnicas creadas para ello. Este tipo de técnicas se basan en que el tiempo de ejecución de un programa depende de:

1. El tamaño de los datos con los que trabaja, es decir, si debe procesar más o menos información.

2. Lo mejor o peor que sea el código generado por el compilador o intérprete del lenguaje.
3. Lo más o menos rápido que sea el computador utilizado.
4. Lo más o menos *complicado* que sea el algoritmo.

De estos cuatro puntos, el 2º y el 3º dependen de factores externos al algoritmo, por tanto parece lógico que centrarse sólo en el 1º y en el 4º, además, ambos dan cierta idea de que no es apropiado expresar el tiempo de ejecución de un algoritmo en unidades estándar de tiempo, tales como segundos, horas, etc..., ya que esta ejecución depende del compilador y del computador empleados.

Del primer punto se puede extraer la conclusión de que al depender del *tamaño* de la entrada, el tiempo de ejecución deberá definirse en función de la entrada, en concreto, de su tamaño; por ejemplo, no tardará lo mismo un algoritmo cuando realice la ordenación alfabética de todos los alumnos de un aula, que cuando realice la ordenación alfabética de todos los alumnos de la Universidad.

Teniendo esto en cuenta, se suele utilizar la notación $T(n)$ para representar el tiempo de ejecución de un algoritmo para una entrada de tamaño n. Por ejemplo, algunos algoritmos pueden tener un tiempo de ejecución $T(n)=cn^2$, donde c es una constante. Las unidades de $T(n)$, como se verá a continuación, son el número de instrucciones ejecutadas en un computador. La medida del tiempo de ejecución de un algoritmo así calculada se dice que se ha hecho a partir de un *análisis a priori*.

Distintos algoritmos con $T(n)$ iguales a 1, n, y n^2 se dice que tienen diferentes *órdenes de magnitud*. Relacionado con los algoritmos, el orden de magnitud de una sentencia es el número de veces que ésta se ejecuta -su frecuencia-, mientras que el orden de magnitud de un algoritmo se obtiene sumando las frecuencias de todas las sentencias que lo componen.

Caso peor, caso mejor y caso medio

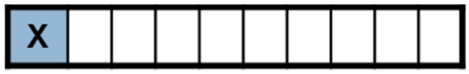
En ocasiones, el tiempo de ejecución de un algoritmo dependerá de la entrada específica en un momento dado, además del tamaño de la misma. Por ejemplo, volviendo al caso de la ordenación alfabética, es lógico que se tarde más tiempo en ordenar una misma cantidad de nombres si éstos se encuentran justo en el orden inverso al que deberían estar; así como también es lógico que se tarda menos tiempo si estos nombres ya están ordenados en el sentido correcto.

Es por esto que en ocasiones se estudia el tiempo de ejecución de un algoritmo en el *peor caso*, *mejor caso*, y en el *caso promedio*.

- Peor caso: se produce cuando es necesario que se efectúe el mayor número de operaciones elementales.
- Mejor caso: se produce cuando es necesario que se efectúe el menor número de operaciones elementales
- Caso promedio: determina el número de operaciones elementales esperado que deben efectuarse (Requiere conocer la distribución de probabilidad de los datos de entrada)

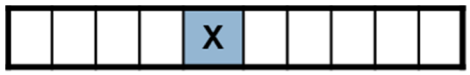
Para ilustrar estos conceptos se puede considerar el problema de buscar secuencialmente un elemento determinado en un array.

- Mejor caso**:** el elemento está en la primera posición (una comparación)



- Peor caso**:** el elemento está en la última posición o no está (n comparaciones)

- Caso medio**:** el elemento está en una posición intermedia (n/2 comparaciones)



En general, en el análisis teórico de algoritmos interesa conocer el caso peor ya que la función obtenida será una cota superior.

Notación asintótica

Describe el comportamiento de las funciones cuando el parámetro n (tamaño del problema) tiende a infinito. El objetivo es tener una idea se cómo se comporta el algoritmo para valores de n grandes, aunque a veces, si el tamaño del problema es pequeño puede suceder que el comportamiento de los algoritmos difiera del predicho por la notación asintótica.

Con el análisis asintótico se pueden comparar dos algoritmos según el orden de magnitud de sus complejidades y saber sin necesidad de probarlos cuál de los dos es más eficiente, más rápido.

Un algoritmo requiere un **tiempo de orden de *T(n)*,** si existe una constante positiva ***c*** y una implementación del algoritmo capaz de resolver todos los casos de tamaño ***n*** en un tiempo no superior a ***cT(n)***. **T(n)** depende del algoritmo, ***c*** de la implementación.

Si se considera un algoritmo cuyo tiempo de ejecución ha resultado ser

$$T(n) = 32n^2 + 78n + 54$$

Puesto que $(n \leq n^2)$ y $(1 \leq n^2)$ para todo $n \geq 1$, se cumple que

$$T(n) = 32n^2 + 78n + 54 \leq 32n^2 + 78n^2 + 54n^2 = 164n^2$$

Este algoritmo tiene un tiempo de ejecución cuadrático, es decir, en el orden de n^2

¿Qué importancia tiene la constante *c* que antes se ha empleado en $T(n)=cn^2$?, la verdad es que poca, por no decir ninguna, ya que como se va a ver, para valores *grandes* de *n* -que es con los que se trabaja habitualmente- se puede despreciar. El siguiente ejemplo aclara esta idea:

¿Qué algoritmo es más eficiente para resolver un mismo problema, uno que tarda $T(n)=100n^2$, o uno que tarda $T(n)=5n^3$?

La siguiente tabla muestra el T(n) de ambos algoritmos para distintos valores de n:

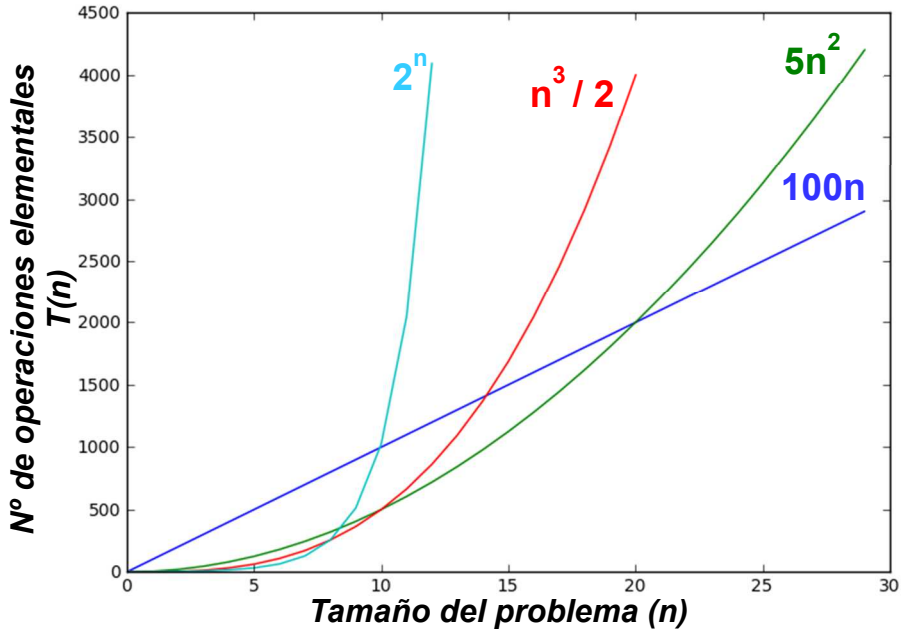
n	$100n^2$	$5n^3$
1	100	5
…	…	…
10	10000	5000
…	…	…
20	40000	40000
…	…	…
30	90000	135000
…	…	…
40	160000	320000
…	…	…

Resumiendo, para valores de *n* tales que $n<20$, el que invierte un tiempo $T(n)=5n^3$ será mejor. Conforme crece el tamaño de la entrada, el algoritmo con $T(n)=n^3$ requiere un tiempo mayor que el que invierte $T(n)=n^2$, independientemente de la constante en uno y otro caso. Si cambiase la constante, bastaría con encontrar un nuevo valor para *n* a partir del cual se volvería a cumplir lo que hemos enunciado.

Este ejemplo debe servir para aprender una lección muy importante: hay que procurar construir algoritmos cuyos tiempos de ejecución sean lo más pequeños posible, ya que a medida que los computadores son más y más rápidos, los usuarios de los mismos quieren resolver problemas más grandes, pero a menos que un algoritmo tenga un tiempo de ejecución del tipo $T(n)=n$ -lineal con el tamaño de la entrada- o $T(n)=n \log n$ un incremento pequeño en la rapidez de un computador no va a repercutir de manera visible en el tamaño del problema más grande que es posible resolver en una cantidad fija de tiempo. Ilustremos esta afirmación con un ejemplo:

Disponemos de un tiempo de uso de 1000 segundos de un computador con una determinada velocidad de proceso, el cual se puede hacer que funcione 10 veces más rápido. En él vamos a ejecutar cuatro algoritmos, cuyos tiempos de ejecución son: $T(n)=100n$, $T(n)=5n^2$, $T(n)=n^3/2$, y $T(n)=2^n$. ¿Cuál es el tamaño máximo del problema que se puede resolver al principio y después, cuando es 10 veces más rápido?

Para visualizar cómo crece el tiempo invertido por cada uno de los cuatro algoritmos cuando aumenta el tamaño de la entrada, se puede analizar la siguiente figura:



En la siguiente tabla, la primera columna representa el tipo de algoritmo -$T(n)$-, la segunda columna el tamaño máximo del problema para el computador inicial, la tercera columna el tamaño máximo del problema para el computador 10 veces más rápido, y la cuarta y última columna refleja el incremento del tamaño máximo del problema:

T(n)	Tamaño máximo del problema en el computador inicial	Tamaño máximo del problema en el nuevo computador	Incremento del tamaño máximo del problema
$100n$	10	100	10.0
$5n^2$	14	45	3.2
$n^3/2$	12	27	2.3
2^n	10	13	1.3

Si se analiza en detalle los resultados de la tabla anterior, se observa que:

- En el computador original, los cuatro algoritmos pueden resolver problemas de tamaño similar.
- Los algoritmos con $T(n)=n$ permiten un incremento del tamaño del problema del 1000% cuando la velocidad del computador se incrementa también en un 1000%.
- Los algoritmos con $T(n)=n^3$ y $T(n)=n^2$ permiten un incremento del tamaño del problema del 230% y el 320%, respectivamente, cuando la velocidad del computador se incrementa en un 1000%.
- Los algoritmos con $T(n)=2^n$ permiten un incremento del tamaño del problema de sólo el 130% cuando la velocidad del computador se incrementa en un 1000%.

De este ejemplo se pueden extraer unas conclusiones muy interesantes:

- A partir de una *determinada expresión* del tiempo de ejecución de un algoritmo, aumentos de la rapidez de un computador en, por ejemplo, un 1000% dan lugar a incrementos pequeños de los tamaños de los problemas que pueden resolver.
- Saber calcular el orden de magnitud de un algoritmo es algo muy importante, ya que si para un mismo problema se logra obtener un segundo algoritmo que es un orden de magnitud menor, éste será mucho más rápido que el primero.

Ordenes de complejidad según su eficiencia

Los tiempos de ejecución más comunes ordenados de mayor a menor eficiencia son:

Orden	*T(n)*
logarítmico	**log(n)**
lineal	**n**
cuasi-lineal	**n log(n)**
cuadrático	**n^2**
polinomial (a > 2)	**n^a**
exponencial (a ≥ 2)	**a^n**
factorial	**n!**

Esto quiere decir que si un algoritmo para resolver un determinado problema tarda un tiempo $T(n)=n$, éste será mejor que otro que tarde $T(n)=$ n log y, por supuesto, mucho mejor que otro que tarde $T(n)=n^3$.

- Los algoritmos de complejidad O(n) y O(n log n) son los que muestran un comportamiento más "natural": prácticamente a doble de tiempo, doble de datos procesables.
- Los algoritmos de complejidad logarítmica son un descubrimiento fenomenal, pues en el doble de tiempo permiten atacar problemas notablemente mayores, y para resolver un problema el doble de grande sólo hace falta un poco más de tiempo (ni mucho menos el doble).
- Los algoritmos de tipo polinómico no son una maravilla, y se enfrentan con dificultad a problemas de tamaño creciente. La práctica viene a decir que son el límite de lo "tratable".
- Cualquier algoritmo por encima de una complejidad polinómica se dice "intratable" y sólo será aplicable a problemas pequeños.

Cálculo del tiempo de ejecución de un algoritmo.

Hasta ahora se ha descrito el concepto de tiempo de ejecución de un algoritmo o $T(n)$, pero no se ha dicho nada acerca de cómo calcularlo. Eso es precisamente lo que se va a hacer en esta sección.

Para calcular el tiempo de ejecución de un algoritmo se siguen las siguientes reglas generales:

1. Las sentencias de asignación, lectura y escritura tienen un tiempo de ejecución de 1 unidad, salvo en

aquellos casos donde aparezca una llamada a un subalgoritmo, entonces la complejidad de la sentencia será la del subalgoritmo más una unidad -la de la propia sentencia-.

2. El tiempo de ejecución de una secuencia de acciones se calcula sumando el tiempo de ejecución de cada una de ellas por separado.

3. El tiempo invertido en una sentencia `if else` es la suma del tiempo invertido en evaluar la condición más el tiempo de:

(a) Las sentencias que se ejecutan a partir del `if`, si la condición es cierta.

(b) O de las sentencias que se ejecutan a partir del `else`, si la condición es falsa.

Coste(condición) + MÁXIMO[Coste(S_1), Coste(S_2)]

4. El tiempo invertido en ejecutar un bucle es la suma para todas las iteraciones del bucle, del tiempo de ejecución del cuerpo más el tiempo empleado en evaluar la condición del mismo. S representa el cuerpo del bucle.

- bucle con condición inicial: `while (condición) {S}`

[Coste(condición) + Coste(S)] * nº de iteraciones + Coste(Condición)

- bucle con condición final: `do {S} while (condición)`

[Coste(S) + Coste(condición)] * nº de iteraciones

- bucle controlado por contador: `for(inicializ; cond; incr) {S}`

Coste(inic) + [Coste(cond)+Coste(S)+Coste(incr)]* nº iteraciones + Coste(cond)

Las distintas expresiones que se emplean para calcular el tiempo de ejecución de un bucle están determinadas por el funcionamiento de cada uno de ellos. Por ejemplo, en el bucle con condición inicial, hay que sumar al final el coste de la condición porque en este tipo de bucles la condición se evalúa una vez más que el número de veces que se ejecuta el cuerpo del bucle. En el caso del bucle controlado por contador hay que tener en cuenta el coste de la inicialización del contador, del incremento del mismo, etc.

Las unidades que aquí se manejan se denominan *pasos de programa*, es decir, lo que se está contando al hablar del *tiempo* invertido por un algoritmo para resolver un problema, es el número de pasos o sentencias que debe ejecutar. Al tiempo invertido por un algoritmo en su ejecución, también se le denomina *complejidad* del algoritmo.

Calculando el número de iteraciones de un bucle

Para calcular el número de iteraciones que hace un bucle hay que tener en cuenta varios factores. Por ejemplo, si se trata de un bucle controlado por contador, dependiendo de los valores inicial y final del contador así como del incremento, el número de iteraciones será totalmente distinto. A continuación se muestran varios ejemplos de bucles controlados por contador donde el número de iteraciones es distinto:

```
for(i=1; i<=n; i++)
```

En este caso el número de iteraciones claramente es n

```
for(i=1; i<=n; i=i+2)
```

Para este ejemplo, dado que el contador se incrementa sumándole 2, es obvio que alcanzará el valor final en la mitad de iteraciones. El número de iteraciones es n/2.

En general, el número de iteraciones se puede calcular como **(valor final – valor inicial +1)/incremento**

En los ejemplos anteriores es sencillo calcular el número de iteraciones, pero en otros casos es más complicado. El siguiente ejemplo muestra un par de bucles anidados. El número de iteraciones del bucle interno depende de la iteración concreta en la que está el bucle externo.

```
for(i=1;i<=n;i++)
      for (j=1; j<=i; j++)
            cout << "Hola ";
```

Cuando i vale 1, la instrucción `cout << "Hola "` se ejecuta 1 vez, cuando i vale 2, se ejecuta 2 veces, así hasta que i vale n, que el bucle interno se ejecuta n veces. Es decir, la sentencia `cout << "Hola "` se ejecuta en total `1+2+3+…+n` veces. Esto es una serie aritmética cuyo resultado es:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

En el siguiente ejemplo, el bucle está controlado por la variable i, que empieza con valor n y va disminuyendo de valor hasta alcanzar el valor 1. La forma que tiene de disminuir su valor es dividiéndose entre 2:

```
i = n;
While (i>1){
      i = i/2;
}
```

La siguiente tabla muestra, para distintos valores de n, el número de iteraciones que hace el bucle:

	n=1	**n=2**	**n=3**	**n=4**	**n=8**
		2/2=1	3/2=1	4/2=2	8/2=4
				2/2=1	4/2=2
					2/2=1
TOTAL IT	0	1	1	2	3

Se observa que el número de iteraciones es la parte entera del $\log_2 n$

Ejemplo: Calcular el coste temporal del siguiente programa

```
#include <iostream>
using namespace std;
int calcula(int);
int main() {
   int a, n, c;

   cin >> n;                  1
   a = 1;                     1
   while (a <= n) {           Coste(while)=[1+1+Coste(calcula)+1]*n+1
                              =[1+1+3n+4+1]*n + 1 = 3n²+7n+1
      c=calcula(n);
      a = a+1;                1
   }
}

int calcula(int n) {          Coste(calcula)=1+Coste(for)+1=3n+4
   int cal, i;
   cal = 1;                   1
   for (i=1; i <= n; i++) {   Coste(for)=1+[1+1+1]*n +1 = 3n+2
      cal = cal * n;          1
   }
   return(cal);               1
}
```

Para calcular el coste hay que ir calculando el coste de cada una de las estructuras que forman el algoritmo.

Se empieza por el `main()`. Cada operación elemental se considera un paso de programa, es por ello que la operación de lectura y la asignación se corresponden con un paso de programa cada una.

A continuación hay un bucle `while`. Para poder calcular el coste del `while` hay que calcular el coste de evaluar la condición y el coste del cuerpo del bucle. La condición en una expresión elemental por lo que su coste es 1. Con respecto al cuerpo del bucle, está formado por dos instrucciones:

- una asignación con llamada a una función. Para saber el coste de esta instrucción hay que calcular el coste de la función. El coste será 1+ el coste de la función.
- una asignación: su coste es 1.

Para poder continuar hay que calcular el coste de la función `calcula`. Esta función contiene:

- una asignación: su coste es 1
- un bucle `for`. En este tipo de bucle hay que calcular:
    - coste de inicialización: 1. Es una asignación simple (i=1)
    - coste de la condición: es una expresión elemental (i<=n). Su coste es 1
    - coste del cuerpo: es una asignación. Su coste es 1
    - coste del incremento: es una asignación (i++). Su coste es 1
    - número de iteraciones que hace el bucle: n.
    - Por tanto el coste del bucle es `1+[1+1+1]*n +1 = 3n+2`
- una instrucción `return()`. Su coste es 1.

El coste total de la función `calcula` es `1+Coste(for)+1=3n+4`

Ya se puede calcular el coste del bucle `while`:

- coste de la condición: 1 (a<=n)
- coste del cuerpo: `1+Coste(calcula)+1`
- número de iteraciones que hace el bucle: `n`
- por tanto el coste del bucle es `=[1+1+Coste(calcula)+1]*n+1 =[1+1+3n+4+1]*n + 1 =` $3n^2+7n+1$

El coste total es:

$T(n) = Coste(main) = 2 + Coste(while) = 2+3n^2+7n+1 = 3n^2+7n+3 \in O(n^2)$