Análisis de la eficiencia de los algoritmos

Isabel Pita.

Facultad de Informática - UCM

25 de septiembre de 2023

Indice

Biliografía Recomendada

- Algoritmos correctos y eficientes: Diseño razonado ilustrado con ejercicios. *Matí-Oliet, N.; Segura Diaz, C. M., Verdejo Lopez, A.*. Ibergarceta Publicaciones, 2012.
  Versión electrónica en la biblioteca UCM.
- Especificación, Derivación y Análisis de Algoritmos: ejercicios resueltos. *Narciso Martí Oliet, Clara María Segura Díaz y Jose Alberto Verdejo López*. Colección Prentice Práctica, Pearson Prentice-Hall, 2006.
  Edición anterior del mismo libro.

Para ampliar el temario, capítulo 3 de ambos libros.

- Ejercicios resueltos 3.2, 3.3, 3.5, 3.6, 3.7, 3.8, 3.9, 3.10, 3.11.
- Ejercicios propuestos 3.1, 3.2, 3.3, 3.4.

Objetivos.

1. Obtener una medida que nos permita:
   - *Comparar la eficiencia en tiempo y espacio de distintos algoritmos para un mismo problema.*
   - *Identificar algoritmos/problemas con un tiempo de ejecución o uso de memoria inaceptable para un tamaño de entrada.*
2. Conocer las reglas que nos permiten obtener el orden de complejidad en tiempo de un algoritmo.
3. Conocer las reglas que nos permiten obtener el orden de complejidad en uso de memoria de un algoritmo.

Eficiencia de un algoritmo.

No nos preocupa el tiempo/memoria real (segundos, milisegundos../bytes) de un algoritmo implementado en un lenguaje concreto y ejecutado en una máquina concreta.

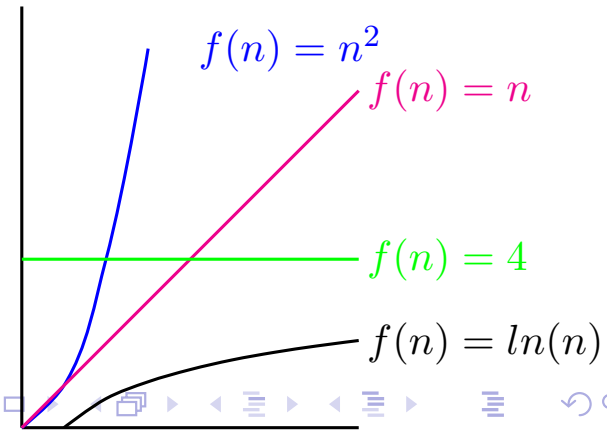
Buscamos una medida:

- independiente del ordenador concreto en que estemos ejecutando y del lenguaje de programación que utilicemos,
- que nos permita obtener una relación entre los datos de entrada y el tiempo que tarda en ejecutarse el programa o la memoria que utiliza.
- Ejemplos:
  - Sea un algoritmo que ordena los valores de un vector. Si el algoritmo tarda un segundo en ordenar un vector de tamaño 100.000, ¿Cuánto tiempo tarda aproximadamente en ordenar un vector de tamaño doble?.
  - Sea un algoritmo que calcula todas las permutaciones de una colección de elementos. ¿Podemos ejecutarlo con colecciones de más de 1000 elementos en un tiempo razonable?

Medidas asintóticas de la eficiencia

1. Estudiamos la relación entre el tiempo de ejecución o memoria utilizada por el algoritmo y el tamaño/valor de los datos de entrada.
2. El estudio se aplica tanto al tiempo de ejecución como al uso de memoria. Por defecto nos referimos al tiempo de ejecución.
3. La relación se expresa mediante funciones de coste
$$f : \mathbb{N} \longrightarrow \mathbb{R}^+ \cup \{0\},$$
los argumentos representan el tamaño/valor de los datos de entrada, y el resultado representa el coste del algoritmo.
4. De estas funciones no nos interesan los valores concretos sino su forma de crecimiento.

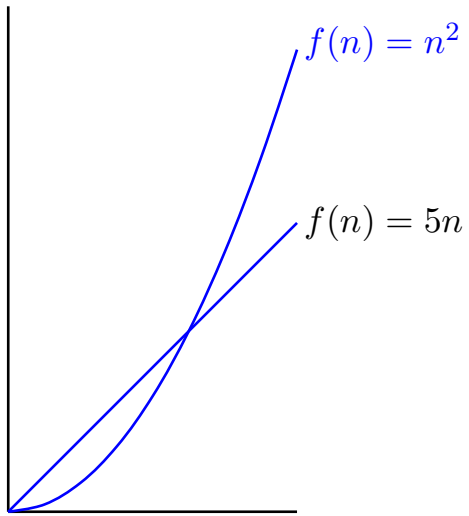
Si nuestro algoritmo tiene como función de coste $f(n) = n^2$ sabemos que al crecer el tamaño/valor de los datos de entrada el tiempo de ejecución aumenta muy rápido.



Medidas asintóticas de la eficiencia. Cuestiones

1. Dado un algoritmo cuya función de coste es $f(n) = 4$. ¿Cómo aumenta el tiempo de ejecución al aumentar el tamaño/valor de los datos de entrada?

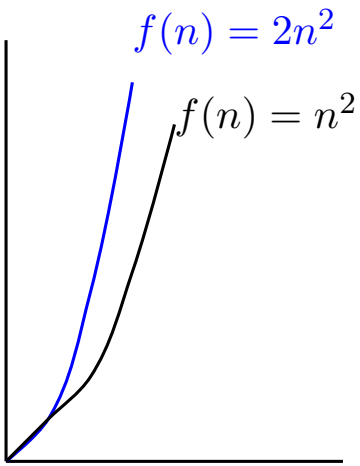
2. Dados dos algoritmos, cuyas funciones de coste son: $f(n) = 5n$ y $f(n) = n^2$ ¿Cuál de los algoritmos es más rápido para tamaños de entrada *grandes* ($n \in (5 \ldots \infty)$)? ¿Y cuál para tamaños de entrada *pequeños* ($n \in [0 \ldots 5)$)?

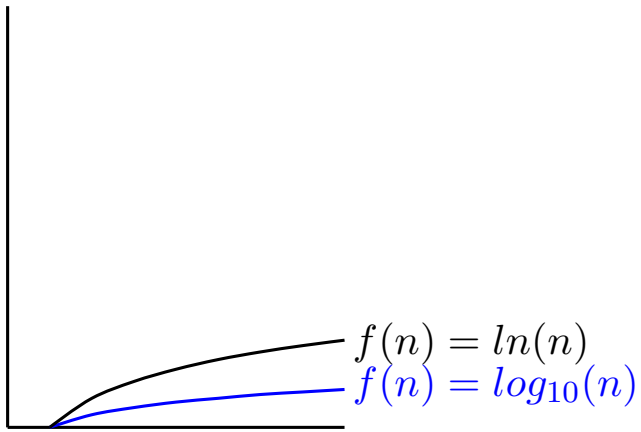


Medidas asintóticas de la eficiencia. Cuestiones

1. Dados dos algoritmos, cuyas funciones de coste son: $f(n) = 2n^2$ y $f(n) = n^2$. ¿Hay *mucha diferencia* en su comportamiento para *tamaños grandes* ($n \in [10^6 \dots 10^8]$)?

2. Dados dos algoritmos, cuyas funciones de coste son: $f(n) = log_{10}(n)$ y $f(n) = ln(n)$ ¿Hay *mucha diferencia* en su comportamiento para *tamaños grandes*?





Medidas asintóticas de la eficiencia

Dada una función de coste f, consideramos equivalentes a ella todas las funciones que están acotadas superiormente por un múltiplo de f.
La clase de equivalencia del *orden de f*, $\mathcal{O}(f)$ está formada por todas las funciones equivalentes a f.

Definición 4.1

Sea una función de coste $f : \mathbb{N} \longrightarrow \mathbb{R}^+ \cup \{0\}$. *El conjunto de las funciones* **del orden de** $f(n)$, *denotado* $\mathcal{O}(f(n))$, *se define como:*

$$\mathcal{O}(f(n)) = \{g : \mathbb{N} \longrightarrow \mathbb{R}^+ \cup \{0\} \mid \exists c \in \mathbb{R}^+, n_0 \in \mathbb{N} \,. \; \forall n \geq n_0 \,.\, g(n) \leq cf(n)\}$$

Asímismo, diremos que una función g **es del orden de** $f(n)$ *cuando* $g \in \mathcal{O}(f(n))$. *También diremos que* g **está** *en* $\mathcal{O}(f(n))$.

Medidas asintóticas de la eficiencia. Cuestiones

- Indica tres funciones de coste que pertenezcan a la clase $\mathcal{O}(n)$
- Indica tres funciones de coste que pertenezcan a la clase $\mathcal{O}(n^2)$
- ¿Pertenece la función de coste $f(n) = 5n + 3$ a la clase $\mathcal{O}(n^2)$?
- ¿Pertenece la función de coste $f(n) = \frac{1}{2}n^2$ a la clase $\mathcal{O}(n)$?

Medidas asintóticas de la eficiencia

Las clases $\mathcal{O}(f(n))$ para diferentes funciones $f(n)$ se denominan **clases de complejidad**, u **órdenes de complejidad**.

Se elije como representante del orden $\mathcal{O}(f(n))$ la función $f(n)$ **más sencilla** posible dentro del mismo.

Esta función da nombre al orden:

1. $\mathcal{O}(1)$: **constantes**
2. $\mathcal{O}(\log n)$: **logarítmico**
3. $\mathcal{O}(n)$: **lineal**
4. $\mathcal{O}(n^2)$: **cuadrático**
5. $\mathcal{O}(n^k)$: **polinomial**
6. $\mathcal{O}(2^n)$: **exponencial**
7. $\mathcal{O}(n!)$: **factorial**

Para demostrar que una función pertenece a un orden se aplica directamente la definición.

- Demostrar que $(n+1)^2 \in O(n^2)$.
- Un modo de hacerlo es por inducción sobre n.
- Elegimos $n_0 = 1$ y $c = 4$, es decir demostraremos $\forall n \geq 1 \,.\, (n+1)^2 \leq 4n^2$:

Caso base: $n = 1$, $(1+1)^2 \leq 4 \cdot 1^2$
Paso inductivo: h.i. $(n+1)^2 \leq 4n^2$. Demostrémoslo para $n+1$:

$$\begin{aligned} (n+1+1)^2 &\leq 4(n+1)^2 \\ (n+1)^2 + 1 + 2(n+1) &\leq 4n^2 + 4 + 8n \\ (n+1)^2 &\leq 4n^2 + \underbrace{6n+1}_{\geq 0} \end{aligned}$$

Medidas asintóticas de la eficiencia

Para demostrar que una función no pertenece a un orden se hace por reducción al absurdo.

Probar que $3^n \notin O(2^n)$.

- Si perteneciera, existiría $c \in \mathbb{R}^+$, $n_0 \in \mathbb{N}$ tales que $3^n \leq c \cdot 2^n$ para todo $n \geq n_0$.
- Esto implicaría que $(\frac{3}{2})^n \leq c$ para todo $n \geq n_0$.
- Pero esto es falso porque dado un c cualquiera, bastaría tomar $n > \log_{1,5} c$ para que $(\frac{3}{2})^n > c$, es decir $(\frac{3}{2})^n$ no se puede acotar superiormente.

Medidas asintóticas de la eficiencia

- La notación $\mathcal{O}(f(n))$ nos da una cota superior al tiempo de ejecución $t(n)$ de un algoritmo.

Cuestión:
Dada la función de coste $f(n) = 5n$. Demuestra que $f(n) \in \mathcal{O}(n^2)$

Medidas asintóticas de la eficiencia

- Normalmente estaremos interesados en la **menor** función $f(n)$, tal que $t(n) \in \mathcal{O}(f(n))$.

Jerarquía de órdenes de complejidad

$$\underbrace{\underbrace{O(1) \subset O(\log n) \subset O(n) \subset O(n \log n) \subset O(n^2)}_{\text{razonables en la práctica}} \subset \ldots \subset O(n^k) \subset}_{\text{tratables}}$$

$$\underbrace{\ldots \subset O(2^n) \subset O(n!)}_{\text{intratables}}$$

La clase *omega de f* $(\Omega(f))$ está formada por todas las funciones de coste que están acotadas inferiormente por un múltiplo de f.

Definición 4.2

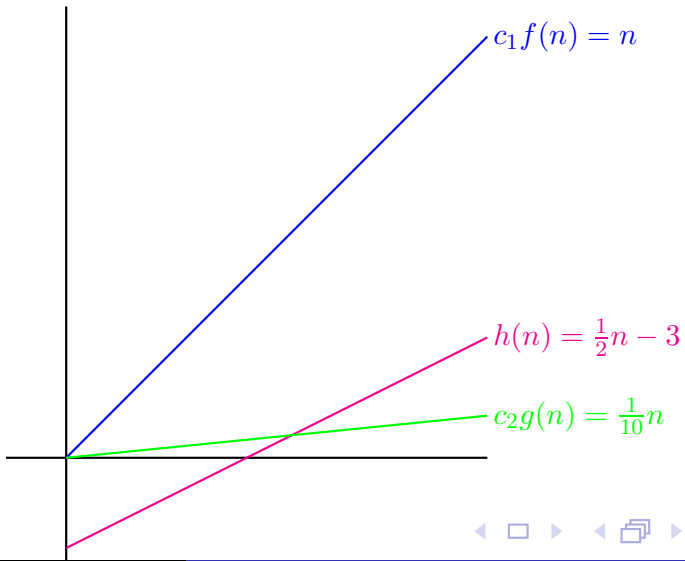
Sea la función de coste $f : \mathbb{N} \longrightarrow \mathbb{R}^+ \cup \{0\}$. *El conjunto* $\Omega(f(n))$, *leído* **omega de** $f(n)$, *se define como:*

$$\Omega(f(n)) = \{g : \mathbb{N} \longrightarrow \mathbb{R}^+ \cup \{0\} \mid \exists c \in \mathbb{R}^+, n_0 \in \mathbb{N} \,. \; \forall n \geq n_0 \,.\, g(n) \geq cf(n)\}$$

Medidas asintóticas de la eficiencia

- Si el tiempo $t(n)$ de un algoritmo en el caso peor está en $\mathcal{O}(f(n))$ y en $\Omega(g(n))$, lo que estamos diciendo es que $t(n)$ no puede valer más que $c_1 f(n)$, ni menos que $c_2 g(n)$, para dos constantes apropiadas c_1 y c_2 y valores de n suficientemente grandes.

Por ejemplo, la función de coste $h(n) = \frac{1}{2}n - 3$ está en $\mathcal{O}(n)$ y en $\Omega(n)$. Tomando $c_1 = 1$ y $c_2 = \frac{1}{10}$ se comprueba que para todo $n > 8$ se cumple que $\frac{1}{2}n - 3 \leq n$ $(h(n) \subset \mathcal{O}(n))$ y $\frac{1}{2}n - 3 \geq \frac{1}{10}n$ $(h(n) \subset \Omega(n))$



Medidas asintóticas de la eficiencia

Para tratar los casos en que la cota superior e inferior del tiempo de un algoritmo es la misma función se define la siguiente medida.

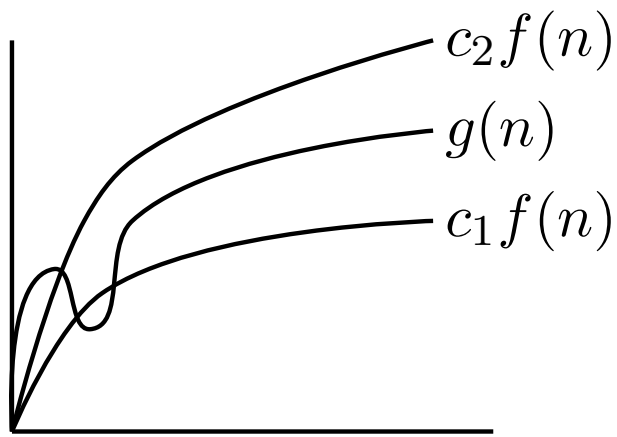
Definición 4.3

El conjunto de funciones de coste $\Theta(f(n))$, *leído* **del orden exacto de** $f(n)$, *se define como:*

$$\Theta(f(n)) = \mathcal{O}(f(n)) \cap \Omega(f(n))$$

Medidas asintóticas de la eficiencia

- También se puede definir como:
$$\Theta(f(n)) = \{g : \mathbb{N} \longrightarrow \mathbb{R}^+ \cup \{0\} \mid \exists c_1, c_2 \in \mathbb{R}^+, n_0 \in \mathbb{N} \,.\, \forall n \geq n_0 \,.\, c_1 f(n) \leq g(n) \leq c_2 f(n)\}$$



- Siempre que sea posible, daremos el orden exacto del coste de un algoritmo, por ser más informativo que dar solo una cota superior.

Propiedades de los órdenes de complejidad

- $O(a \cdot f(n)) = O(f(n))$ con $a \in \mathbb{R}^+$.

  ($\subseteq$) $g \in O(a \cdot f(n)) \Leftrightarrow \exists c \in \mathbb{R}^+, n_0 \in \mathbb{N}$ tal que $\forall n \geq n_0 \,.\, g(n) \leq c \cdot a \cdot f(n)$. Tomando $c' = c \cdot a$ se cumple que $\forall n \geq n_0 \,.\, g(n) \leq c' \cdot f(n)$, luego $g \in O(f(n))$.

  ($\supseteq$) $g \in O(f(n)) \Leftrightarrow \exists c \in \mathbb{R}^+, n_0 \in \mathbb{N}$ tal que $\forall n \geq n_0 \,.\, g(n) \leq c \cdot f(n)$.
  Entonces tomando $c' = \frac{c}{a}$ se cumple que $\forall n \geq n_0 \,.\, g(n) \leq c' \cdot a \cdot f(n)$, luego $g \in O(a \cdot f(n))$.

Propiedades de los órdenes de complejidad

- La base del logaritmo no importa: $O(\log_a n) = O(\log_b n)$, con $a, b > 1$. La demostración es inmediata sabiendo que:

$$\log_b n = \frac{\log_a n}{\log_a b}$$

- Si $f \in O(g)$ y $g \in O(h)$, entonces $f \in O(h)$.

$$f \in O(g) \Rightarrow \exists c_1 \in \mathbb{R}^+, n_1 \in \mathbb{N} \text{ tal que } \forall n \geq n_1 \,.\, f(n) \leq c_1 \cdot g(n)$$
$$g \in O(h) \Rightarrow \exists c_2 \in \mathbb{R}^+, n_2 \in \mathbb{N} \text{ tal que } \forall n \geq n_2 \,.\, g(n) \leq c_2 \cdot h(n)$$

Tomando $n_0 = \text{máx}(n_1, n_2)$ y $c = c_1 \cdot c_2$, se cumple

$$\forall n \geq n_0 \,.\, f(n) \leq c_1 \cdot g(n) \leq c_1 \cdot c_2 \cdot h(n)$$

Y por tanto $f \in O(h)$.

Propiedades de los órdenes de complejidad

- Regla de la suma: $O(f+g) = O(\text{máx}(f,g))$.

  ($\subseteq$) $h \in O(f+g) \Rightarrow \exists c \in \mathbb{R}^+, n_0 \in \mathbb{N} \,.\, \forall n \geq n_0 \,.\, h(n) \leq c \cdot (f(n)+g(n))$. Pero $f \leq \text{máx}(f,g)$ y $g \leq \text{máx}(f,g)$, luego:

  $$h(n) \leq c \cdot (\text{máx}(f(n),g(n)) + \text{máx}(f(n),g(n)))$$

  $$= 2 \cdot c \cdot \text{máx}(f(n),g(n))$$

  Tomando $c' = 2 \cdot c$ se cumple que $\forall n \geq n_0 \,.\, h(n) \leq c' \cdot \text{máx}(f(n),g(n))$ y por tanto $h \in O(\text{máx}(f,g))$.

  ($\supseteq$) $h \in O(\text{máx}(f,g)) \Rightarrow \exists c \in \mathbb{R}^+, n_0 \in \mathbb{N} \,.\, \forall n \geq n_0 \,.\, h(n) \leq c \cdot \text{máx}(f(n),g(n))$. Pero $\text{máx}(f,g) \leq f+g$, luego $h \in O(f+g)$ trivialmente.

Propiedades de los órdenes de complejidad

- Teorema del límite para la clase de funciones de coste $\mathcal{O}(f(n))$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = m \in \mathbb{R}^+ \Rightarrow f \in O(g) \text{ y } g \in O(f) \Leftrightarrow O(f) = O(g)$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \Rightarrow f \in O(g) \text{ y } g \notin O(f) \Leftrightarrow O(f) \subset O(g)$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = +\infty \Rightarrow f \notin O(g) \text{ y } g \in O(f) \Leftrightarrow O(f) \supset O(g)$$

Propiedades de los órdenes de complejidad

- Teorema del límite para la clase de funciones de coste $\Omega(f(n))$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = m \in \mathbb{R}^+ \Rightarrow g \in \Omega(f) \text{ y } f \in \Omega(g) \Leftrightarrow \Omega(f) = \Omega(g)$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \Rightarrow g \in \Omega(f) \text{ y } f \notin \Omega(g) \Leftrightarrow \Omega(f) \supset \Omega(g)$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = +\infty \Rightarrow g \notin \Omega(f) \text{ y } f \in \Omega(g) \Leftrightarrow \Omega(f) \subset \Omega(g)$$

Propiedades de los órdenes de complejidad

- Aplicando la definición de $\Theta(f)$, también tenemos:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = m \in \mathbb{R}^+ \Rightarrow g \in \Theta(f) \text{ y } f \in \Theta(g)$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \Rightarrow f \in O(g) \text{ pero } f \notin \Theta(g)$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = +\infty \Rightarrow g \in O(f) \text{ pero } g \notin \Theta(f)$$

Reglas de cálculo de la complejidad en tiempo.

- Las operaciones aritméticas, comparaciones, asignaciones a expresiones de tiempo constante tienen tiempo constante $\Theta(1)$.
- El coste de una composición secuencial $I_1; I_2$ es la suma de los costes de las instrucciones I_1 e I_2. Aplicando la regla de la suma, es el máximo de los costes de las instrucciones I_1 e I_2.
- El coste de una instrucción condicional es la suma de los costes de evaluar la condición booleana más el máximo entre el coste de cada una de las instrucciones condicionales.
- El coste de una instrucción iterativa se calcula sumando el coste de cada una de las vueltas que se dan en el bucle. Si todas las vueltas tienen el mismo coste se multiplica el coste de una vuelta por el número de vueltas que se ejecutan.

Reglas de cálculo de la complejidad en tiempo.

- En las instrucciones condicionales, e iterativas el coste depende del valor de expresiones booleanas que pueden ser ciertas o falsas dependiendo de los valores de entrada.
- Para que la medida sea independiente de los datos de entrada se consideran las siguientes estimaciones:
  - Coste en el caso peor. Coste máximo que hay que esperar para que acabe la ejecución. Es el que se estudiará este curso.
  - Coste en el caso mejor. Coste mínimo que hay que esperar para que acabe la ejecución. No suele tener interés.
  - Coste en el caso medio. Suma de los costes de todos los casos de entrada del mismo tamaño dividido entre el número de casos. Requiere un cálculo más complejo que los anteriores.
  - Coste amortizado. Se analiza el coste de ejecutar el algoritmo muchas veces dividido entre el número de veces que se ejecuta. Es útil cuando el caso peor de una operación ocurre cada cierto número de ejecuciones.

Reglas de cálculo de la complejidad en tiempo.

Cálcula la función de coste y el orden de complejidad del siguiente programa que comprueba si un valor pertenece a un vector recorriéndolo de izquierda a derecha.

```
1 template <class T>
2 bool Search (std::vector<T> const& v, T const& x) {
3     size_t i = 0;
4     while (i < v.size() && v[i] != x)
5         ++i;
6     return i < v.size();
7 }
```

Reglas de cálculo de la complejidad en tiempo.

- El programa es una composición secuencial, por lo tanto el coste es el máximo de los costes de cada instrucción.
- El coste de la instrucción de la línea 3 del código es constante porque se trata de una asignación.
- Las instrucciones 4 y 5 son una instrucción iterativa,
  - el coste de cada vuelta del bucle (comparaciones de la línea 4 y línea 5) es constante porque son comparaciones y un incremento.
  - el número de vueltas del bucle depende de los datos de entrada, en el caso peor no se encuentra el elemento y el bucle da `v.size()` vueltas. Es decir da tantas vueltas como elementos tenga el vector.
- El coste de la instrucción de retorno (línea 6) es constante porque se trata de una comparación.
- Si consideramos que cada instrucción constante se hace en tiempo 1, la función de coste es: $f(n) = 3n + 2 \in \Theta(n)$, siendo n el número de elementos del vector.

Reglas de cálculo de la complejidad en tiempo.

Programas recursivos.

- El cálculo de la función de coste de un programa recursivo requiere conocer la complejidad de la propia función para un tamaño de entrada *menor*.

```
1 long long int factorial ( int n ){
2   if ( n == 0 ) return 1;
3   else return = n * factorial(n-1);   // (n > 0)
4 }
```

- La función tiene una instrucción condicional, el coste de la función será el máximo entre el coste de la condición, y el coste de las instrucciones de las condiciones.
- El coste de calcular la condición, `n == 0`, es constante.
- El coste de calcular la primera instrucción condicional, `return 1;`, es constante.
- El coste de calcular la segunda instrucción condicional depende del coste de la propia función.

Reglas de cálculo de la complejidad en tiempo.

Programas recursivos.

```
int factorial ( int n ){
  if (n == 0) return 1;
  else return n * factorial(n-1);   // (n > 0)
}
```

- El problema es que necesitamos el coste de la función de la que estamos calculando el coste.
- El problema se resuelve utilizando ecuaciones de recurrencia.
- Para un tamaño de entrada n se plantea el coste de las instrucciones de cada condición.
- $T(n) = \begin{cases} c_0 & \text{si } n = 0 \\ c_1 + T(n-1) & \text{si } n > 0 \end{cases}$
- El coste del caso base es constante.
- El coste del caso recursivo se plantea en función del coste de la función para un tamaño de entrada menor.
- La ecuaciones de recurrencia se estudian en el tema 4.

Complejidad en tiempo vs tamaño de los datos

n	Worst AC Algorithm	Comment
$\leq [10..11]$	$O(n!), O(n^6)$	e.g. Enumerating permutations (Section 3.2)
$\leq [15..18]$	$O(2^n \times n^2)$	e.g. DP TSP (Section 3.5.2)
$\leq [18..22]$	$O(2^n \times n)$	e.g. DP with bitmask technique (Section 8.3.1)
≤ 100	$O(n^4)$	e.g. DP with 3 dimensions + $O(n)$ loop, ${}_nC_{k=4}$
≤ 400	$O(n^3)$	e.g. Floyd Warshall's (Section 4.5)
$\leq 2K$	$O(n^2 \log_2 n)$	e.g. 2-nested loops + a tree-related DS (Section 2.3)
$\leq 10K$	$O(n^2)$	e.g. Bubble/Selection/Insertion Sort (Section 2.2)
$\leq 1M$	$O(n \log_2 n)$	e.g. Merge Sort, building Segment Tree (Section 2.3)
$\leq 100M$	$O(n), O(\log_2 n), O(1)$	Most contest problem has $n \leq 1M$ (I/O bottleneck)

Table 1.4: Rule of thumb time complexities for the 'Worst AC Algorithm' for various single-test-case input sizes n, assuming that your CPU can compute $100M$ items in 3s.

[1]Competitive programming 3. S. Halim, F. Halim.

Comparación de algoritmos.

- Supongamos que tenemos varios algoritmos que resuelven el mismo problema. Cada algoritmo tiene una complejidad.
- ¿Cual de ellos es más rápido?. En el ejemplo siguiente n representa el número de elementos del vector.
- Problema: ordenar un vector de enteros:
  - Algoritmo 1: Método de ordenación por inserción. Complejidad en el caso peor $\Theta(n^2)$.
  - Algoritmo 2: Método de ordenación por mezclas. Complejidad en el caso peor $\Theta(n \log n)$.
- Por la definición de la medida asintótica se tiene que para valores de n *suficientemente grandes* la ordenación por mezcla es mejor que la ordenación por inserción, dado que $\Theta(n \log n) \subset \Theta(n^2)$

Comparación de algoritmos. Un análisis más fino

- Supongamos que tenemos varios algoritmos que resuelven el mismo problema en el mismo orden de complejidad. En este caso debemos tener en cuenta las constantes multiplicativas y aditivas para compararlos.
- ¿Cual de ellos es más rápido?. En el ejemplo siguiente n representa el número de elementos del vector.
- Problema: ordenar un vector de enteros:
  - Algoritmo 1: Método de ordenación por inserción. Complejidad en el caso peor $\Theta(n^2)$. Realizando un análisis más fino determinamos que el coste es $5n^2 + 8n - 11$.
  - Algoritmo 2: Método de ordenación de la burbuja modificado. Complejidad en el caso peor $\Theta(n^2)$. Realizando un análisis más fino determinamos que el coste es: $8n^2 - 5$
- Como para cualquier valor de n se tiene $5n^2 + 8n - 11 \leq 8n^2 - 5$ el método de ordenación por inserción es mejor que el método de la burbuja.

Comparación de algoritmos. El caso peor o el caso medio.

- Veamos un caso en el que el análisis del caso peor resulta poco preciso. Dado el problema de ordenar un vector:
  - Algoritmo 1: Método de ordenación por inserción. Complejidad en el caso peor $\Theta(n^2)$. Análisis fino: $5n^2 + 8n - 11$.
  - Algoritmo 2: Método de ordenación por selección. Complejidad en el caso peor $\Theta(n^2)$. Análisis fino: $3n^2 + 11n + 2$
  - Siendo n el número de elementos del vector en los dos casos
- Aparentemente el método de ordenación por selección es mejor dado que $3n^2 + 11n + 2 \leq 5n^2 + 8n - 11$ para $n \geq 4$.
- Observemos que pasa con el caso mejor:
  - Algoritmo 1: Método de ordenación por inserción. Complejidad en el caso mejor $\Theta(n)$. Análisis fino: $13n - 11$.
  - Algoritmo 2: Método de ordenación por selección. Complejidad en el caso peor $\Theta(n^2)$. Análisis fino: $\frac{5}{2}n^2 + \frac{9}{2}n + 2$
- Si el vector esta *casi ordenado* (caso mejor de inserción) el método de inserción resulta mucho mejor.
- Se recomienda analizar el coste en el caso medio.

Comparación de algoritmos. Importancia de las constantes multiplicativas.

- Supongamos que tenemos un problema y dos algoritmos para resolverlo, con complejidades $\Theta(n)$, y $\Theta(n^2)$, siendo n el tamaño de la entrada del problema.
- Realizando un análisis más fino resulta una complejidad de:
  - Algoritmo 1: $500n + 200$.
  - Algoritmo 2: $\frac{1}{2}n^2$
- Por el criterio asintótico sabemos que para valores *grandes* de n es mejor el primer algoritmo.
- Sin embargo para valores pequeños esto no es cierto.
  ¿Podemos calcular hasta que tamaño de entrada es mejor utilizar el segundo algoritmo?
- Comparamos el coste de los dos algoritmos y obtenemos el menor valor de n que cumple. $500n + 200 < \frac{1}{2}n^2$
  Resolviendo la ecuación obtenemos que para valores de $n \leq 1001$ el segundo algoritmo es mejor y para valores mayores es mejor el primero.

Identificar algoritmos con tiempo de ejecución inaceptable.

- Tenemos un algoritmo de complejidad $\Theta(n^2)$ para resolver un problema. Los requisitos del problema especifican que para unos datos de entrada de tamaño n $= 100.000$ debe resolverse en menos de un segundo. La máquina en que se ejecutará es capaz de resolver 10^8 operaciones en un segundo.
- ¿Implementamos nuestro algoritmo o necesitamos encontrar uno mejor?
- Si el usuario necesita ejecutar el algoritmo para una entrada de 10^5 elementos y la complejidad del algoritmo es de $\Theta(n^2)$, podemos suponer $(10^5)^2 = 10^{10}$ operaciones. Dado que nuestra máquina ejecuta 10^8 operaciones en un segundo, podemos suponer que el algoritmo tardará más tiempo que el que tenemos disponible. Por lo tanto:

  Intenta encontrar un algoritmo más rápido

- Se puede utilizar un ajuste más fino, pero en muchos casos una comprobación simple como esta es suficiente.

Complejidad en espacio.

- Para calcular el espacio de memoria necesario para la ejecución de un algoritmo, debemos considerar todas las variables declaradas en el programa y multiplicar cada una de ellas por el número de bytes necesarios para almacenarlo.
- Para hacer un primer análisis se emplea una medida de tipo asintótico. Se tienen en cuenta únicamente las variables estructuradas (vectores, structs..). Se considera que todos ellos ocupan el mismo espacio.
- Si se quiere hacer una análisis más fino se considerarán los tipos de las variables.
- Conviene diferenciar la memoria estática requerida por el programa de la memoria dinámica, ya que el espacio de almacenamiento es diferente.

Complejidad en espacio. Ejemplo.

Dado un algoritmo de búsqueda de un elemento en un vector:

```
1 template <class T>
2 bool Search (std::vector<T> const& v, T const& x) {
3     size_t i = 0;
4     while (i < v.size() && v[i] != x)
5         ++i;
6     return i < v.size();
7 }
```

- El espacio requerido es del orden del tamaño del vector.