Análisis de la eficiencia de los algoritmos

Isabel Pita.

Facultad de Informática - UCM

23 de septiembre de 2017

Índice

Biliografía Recomendada

- Algoritmos correctos y eficientes: Diseño razonado ilustrado con ejercicios. *Matí-Oliet, N.; Segura Diaz, C. M., Verdejo Lopez, A.*. Ibergarceta Publicaciones, 2012.
- 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

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 del tiempo de ejecución de un algoritmo, que nos permita:
   - *Comparar la eficiencia de distintos algoritmos para un mismo problema.*
   - *Identificar algoritmos/problemas con un tiempo de ejecución inaceptable para un tamaño de entrada.*
2. Obtener una medida de la memoria requerida por un algoritmo.

Eficiencia en tiempo

Buscamos una medida

- independiente del ordenador concreto en que estemos ejecutando,

- que nos permita obtener una relación entre los datos de entrada y el tiempo que tarda en ejecutarse el programa.

  Por ejemplo: sea un algoritmo que ordena los valores de un vector. Si el algoritmo tarda un segundo en ordenar un vector de tamaño 100.000, debemos poder responder a la pregunta ¿Cuánto tiempo tarda en ordenar un vector de tamaño doble?.

Eficiencia en tiempo

Utilizamos la siguiente aproximación para medir el tiempo de ejecución: **contar** cuántas instrucciones de cada tipo se ejecutan, multiplicar este número por el tiempo que emplea la instrucción en ejecutarse, y realizar la suma para los diferentes tipos.

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 }
```

Eficiencia en tiempo

Sean

$$\begin{aligned} t_a &= \text{tiempo de una asignación entre enteros} \\ t_c &= \text{tiempo de una comparación entre enteros} \\ t_i &= \text{tiempo de incrementar un entero} \\ t_v &= \text{tiempo de acceso a un elemento de un vector} \end{aligned}$$

- Coste línea (3) : t_a.
- Coste líneas (4) y (5):
  - Si el elemento no pertenece al vector (caso peor):
    línea (4) $(n+1)t_c + n(t_v + t_c)$.
    línea (5) nt_i.
  - Si el elemento está en la primera posición del vector (caso mejor):
    línea (4) $t_c + t_v + t_c$.
    línea (5) No se ejecuta.
- Coste línea (6) : t_c.

Eficiencia en tiempo

En el caso peor el coste del algoritmo es:

$$\begin{aligned} &t_a + (n+1)t_c + n(t_v + t_c) + nt_i + t_c = \\ &\qquad 2(n+1)t_c + t_a + n(t_v + t_i) \in \mathcal{O}(n) \end{aligned}$$

siendo n el número de elementos del vector.

En el caso mejor el coste del algoritmo es:

$$t_a + t_c + t_v + t_c + t_c = 3t_c + t_a + t_v \in \mathcal{O}(1)$$

En muchos casos no estaremos interesados en un análisis tan *fino*, sin embargo esta medida tiene varias de las características que buscamos.

Eficiencia en tiempo

Con la aproximación utilizada anteriormente para medir el tiempo de ejecución, éste depende de:

1. El **tamaño** de los datos de entrada:
   - longitud del vector (algoritmos de ordenación de vectores...),
   - valor del dato de entrada (sucesión de fibonacci, cálculo de la potencia de un número...) ,
   - número de cifras del dato de entrada (cambio de base binaria, decimal ....)
   - ...
2. El **contenido** de los datos de entrada. Fijado un tamaño de los datos de entrada, dependiendo del valor de los datos el tiempo es diferente. Por ejemplo, para vectores de un mismo tamaño dependiendo del valor de las componentes del vector obtenemos un tiempo u otro (T_{min} y T_{max}).
3. El código generado por el **compilador** y el **computador** concreto utilizados, que afectan a los tiempos elementales $(t_a, t_c, t_i \ldots)$.

Eficiencia en tiempo

Realizamos las siguientes modificaciones a nuestra aproximación.

- No estamos interesados en el tiempo para un valor concreto de entrada, sino en el tiempo para cualquier valor de un cierto tamaño. Para ello se puede:
  1. Dar una cota superior midiendo el **caso peor**, es decir la ejecución que tarde más tiempo de todos los ejemplares de tamaño n.
     - Establece una cota superior fiable para **todos** los casos del mismo tamaño.
     - Es fácil de calcular
  2. Medir todos los casos de tamaño n y calcular el tiempo del **caso promedio**.
     - Es más difícil de calcular pero a veces es más informativo.
     - Exige conocer la probabilidad con la que se va a presentar cada caso. Si no se conoce se considera uniforme.
  3. Dar una cota inferior midiendo el **caso mejor**. Muy raramente resulta útil.

Eficiencia en tiempo

- Queremos un tiempo de ejecución independiente de la máquina o el compilador utilizado, por ello consideramos todos los tiempos elementales iguales

Aplicando la simplificación anterior y centrándonos en el caso peor, obtenemos una medida del tiempo de ejecución en función únicamente del **tamaño** de los datos de entrada.

- Por ejemplo, el algoritmo de búsqueda tenía un coste:
- $2(\mathrm{n}+1)\mathrm{t_c} + \mathrm{t_a} + \mathrm{n}(\mathrm{t_v} + \mathrm{t_i}) = 4n + 3$, siendo n el tamaño del vector.
- Para un vector de tamaño 100 el coste es: $4 * 100 + 3$.
- Para un vector de tamaño doble el coste será: $4 * 200 + 3$. Aproximadamente el doble.

Este nuevo análisis sigue resultando muy *fino* para una primera aproximación al tiempo de ejecución de un algoritmo.
Normalmente es suficiente con saber como crece la función con respecto al tamaño de los datos.

Eficiencia en tiempo

Tiempo que tardarían en ejecutarse algoritmos según sus funciones de complejidad y el tamaño de datos de entrada, suponiendo que el tiempo de proceso para un dato de entrada es de *1ms*.

n	$\log_{10} n$	n	n^2	n^3	2^n
10	$1\ ms$	$10\ ms$	$0{,}1\ s$	$1\ s$	$1{,}02\ s$
10^3	$3\ ms$	$1\ s$	$16{,}67\ m$	$11{,}57\ d$	$3{,}4 * 10^{291}\ sig$
10^6	$6\ ms$	$1{,}67\ h$	$31{,}71\ a$	$317\,097{,}9\ sig$	$3{,}1 * 10^{301020}\ sig$

Medidas asintóticas de la eficiencia

El criterio asintótico para medir la eficiencia de los algoritmos se basa en tres principios:

1. El coste o eficiencia es una función que solo depende del tamaño de la entrada, e.g. $f(n) = n^2$. Para cada problema habrá que definir qué se entiende por tamaño del mismo.
2. Las constantes multiplicativas o aditivas no se tienen en cuenta, e.g. $f(n) = n^2$ y $g(n) = 3n^2 + 27$ se consideran costes equivalentes.
3. La comparación entre funciones de coste se hará para valores de n **suficientemente grandes**, es decir los costes para tamaños pequeños se consideran irrelevantes.

Medidas asintóticas de la eficiencia

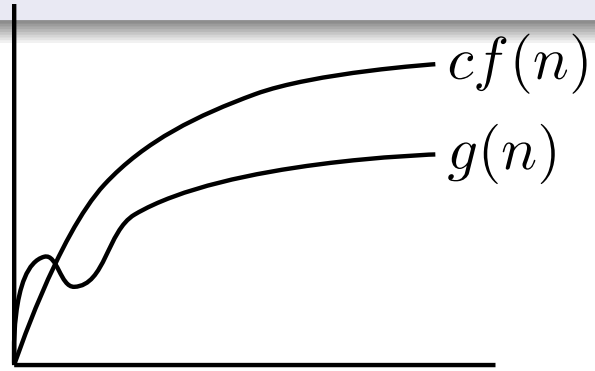
Sea $\mathbb{N}$ el conjunto de los números naturales y $\mathbb{R}^+$ el conjunto de los reales estrictamente positivos.

Definición 4.1

Sea $f : \mathbb{N} \longrightarrow \mathbb{R}^+ \cup \{0\}$. *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* $\mathcal{O}(f(n))$.



- Se admiten funciones negativas o indefinidas para un número finito de valores de n si eligiendo n_0 suficientemente grande, satisface la definición.
- Diferentes implementaciones del mismo algoritmo que difieran en el lenguaje, el compilador, o/y la máquina empleada, son del mismo orden.
- La definición se puede aplicar tanto a un análisis en el caso peor, como a un análisis en el caso promedio. Por ejemplo, hay algoritmos cuyo coste en tiempo está en $\mathcal{O}(n^2)$ en el caso peor y en $\mathcal{O}(n \log n)$ en el caso promedio (Quicksort).
- Las unidades en que se mide el coste en tiempo (horas, segundos, milisegundos, etc.), o en memoria (octetos, palabras, celdas de longitud fija, etc.) **no son relevantes** en la complejidad asintótica.

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. $\mathcal{O}(2!)$: **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.
- Normalmente estaremos interesados en la **menor** función $f(n)$, tal que $t(n) \in \mathcal{O}(f(n))$.

Jerarquía de órdenes de complejidad

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

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

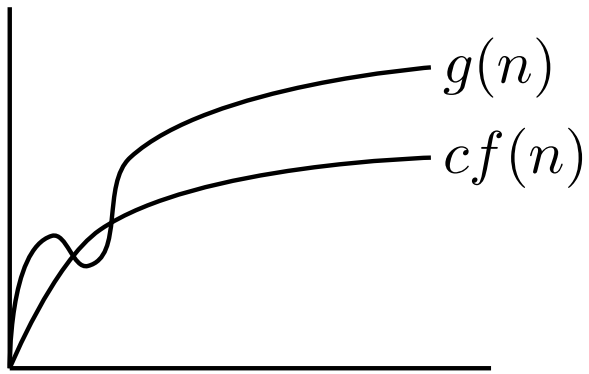
Medidas asintóticas de la eficiencia

Una forma de realizar un análisis más completo es encontrar además la **mayor** función $g(n)$ que sea una cota inferior de $t(n)$.

Definición 4.2

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

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



- NO confundir la medida $\mathcal{O}(f(n))$ como aplicable al caso peor y la medida $\Omega(f(n))$ como aplicable al caso mejor.
  Se aplican **ambas medidas a cada caso** (caso peor, caso promedio, o caso mejor).
- 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.
- Sucede con frecuencia que una misma función $f(n)$ es a la vez cota superior e inferior del tiempo $t(n)$ (peor, promedio, etc.) de un algoritmo.

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.

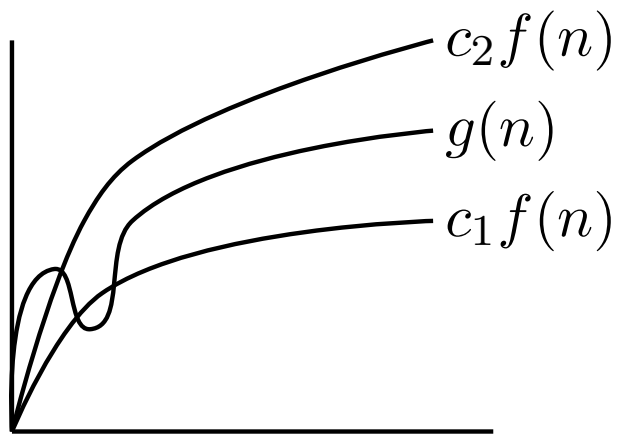
Definición 4.3

El conjunto de funciones $\Theta(f(n))$, *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 $g \in O(a \cdot f(n))$.

- 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 $f \in O(h)$.

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

- Regla de la suma: $O(f+g) = O(\max(f,g))$.

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

  $$h(n) \leq c \cdot (\max(f(n),g(n)) + \max(f(n),g(n)))$$

  $$= 2 \cdot c \cdot \max(f(n),g(n))$$

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

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

- Regla del producto: Si $g_1 \in O(f_1)$ y $g_2 \in O(f_2)$, entonces $g_1 \cdot g_2 \in O(f_1 \cdot f_2)$. La demostración es similar.
- Teorema del límite

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

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

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

- Por el principio de dualidad (ejercicio 1), también tenemos:

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

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

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

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

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

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

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

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: $3n^2 + 11n + 2$
- 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.

Comparación de algoritmos. Importancia de las constantes multiplicativas.

- Supongamos que tenemos un problema y dos algoritmos, con complejidades $\Theta(n)$, y $\Theta(n^2)$ para resolverlo.
- 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.