9 INTRODUCCIÓN AL ESTUDIO DE ALGORITMOS Y SU COMPLEJIDAD

9.1. Definición de complejidad y su medida

9.1.1. Introducción

El objetivo de este tema es sentar las bases que permitan determinar qué algoritmo es mejor (más eficiente) dentro de una familia de algoritmos que resuelven el mismo problema.

La eficiencia de un algoritmo se relaciona con la cantidad de recursos que requiere para obtener una solución al problema (menor cantidad de recursos, mayor eficiencia).

Se asume que todos los algoritmos tratados va a ser eficaces, es decir, resuelven adecuadamente el problema para el que se han diseñado.

> Definición 1:
>
> Se define el coste o complejidad temporal de un algoritmo como el tiempo empleado por éste para ejecutarse y proporcionar un resultado a partir de los datos de entrada.
>
> Definición 2:
>
> Se define el coste o complejidad espacial de un algoritmo como cantidad de memoria requerida(suma total del espacio que ocupan las variables del algoritmo) antes, durante y después de su ejecución.

A partir de la definiciones anteriores se puede intuir que aparecerán problemas a la hora de evaluar la eficiencia de un algoritmo de una manera objetiva. Puesto que pueden aparecer dependencias en la medida de los costes temporales y espaciales con elementos ajenos al propio algoritmo, como por ejemplo: el lenguaje de programación empleado, la máquina en donde se ejecute, el compilador utilizado, la experiencia del programador, etc. Además de depender de elementos como los datos de entrada (número de datos, valor de las variables iniciales, ...), la forma de realizar llamadas a otras bibliotecas de funciones, variables auxiliares (del propio lenguaje), etc ...

Para evitar estos problemas, tiene sentido estimar el coste de los algoritmos con independencia de los programas que los implementen. De la misma manera que tiene sentido analizar la adecuación del diseño de un edificio o de un puente con independencia de su construcción y, por ello, en todos los procesos de ingeniería civil se realizan estudios sobre los diseños propuestos antes de su construcción real.

Lo que se pretende al analizar un algoritmo no es medir el coste temporal exacto (segundos) y la cantidad de memoria (bytes) que necesita para su ejecución, puesto que esto depende del proceso de programación (construcción) y de ejecución de un programa y, por tanto, de la maquina o del compilador empleado. Lo que se pretende es obtener un valor estimado de estos valores mediante el cálculo del número de operaciones que será preciso realizar y del número de variables que será necesario emplear.

Así y todo este coste será difícil de evaluar debido a que diferentes operaciones cuestan un tiempo diferente (la operación 3.24*7.5 no cuesta la mismo que la operación 3+4), existen llamadas a funciones de librería de las que a priori no conocemos el tiempo de ejecución, diferencias de tiempo para la misma operación con diferentes tipos de datos, acceso a periféricos, etc...

Por todo ello, y para simplificar el cálculo, lo que se hará será una estimación de los costes de los algoritmos agrupando las operaciones realizadas en tres clases diferentes y asumiendo un coste temporal único para cada uno de ellos:

- Operaciones aritméticas: t_o
- Asignaciones: t_a
- Comparaciones: t_c

Ejemplo 1: Realizar un algoritmo que calcule

$$y=\sum_{i=1}^{100} x$$

Opción 1. Si se realiza el algoritmo basándose exclusivamente en el enunciado del problema se tendría:

```
y ← 0                              /*1*/
i ← 1                              /*2*/
mientras (i ≤ 100) Hacer           /*3*/
     y ← y + x                     /*4*/
     i ← i + 1                     /*5*/
fin_mientras                       /*6*/
```

Si se analiza el número de asignaciones, operaciones y comparaciones que se realizan en cada una de las líneas del algoritmo se obtiene:

/*1*/ t_a
/*2*/ t_a
/*3*/ $100*t_c$ (comparaciones de repetición del bucle)
/*4*/ $100*(t_o+t_a)$
/*5*/ $100*(t_o+t_a)$
/*6*/ t_c (comparación de salida del bucle)

La estimación de tiempo en función de las operaciones realizadas será entonces:

$Tiempo_1 = 2t_a + 100*(t_c+2t_o+2t_a) + t_c = 202\ t_a + 202\ t_o + 101\ t_c$

Opción 2. Si se estudia con cierto detalle el problema se puede llegar fácilmente a la conclusión que sumar cien veces la variable `x` es análogo a multiplicar `x` por 100, de manera que:

$$y=\sum_{i=1}^{100} x=100\cdot x$$

El algoritmo de resolución quedaría como sigue:

```
y ← 100 * x
```

Y el coste temporal sería:

$Tiempo_2 = t_o + t_a$

Como se puede observar fácilmente, con independencia de los valores exactos de t_o, t_a y t_o, se puede concluir que el tiempo del segundo algoritmo siempre será inferior al tiempo del primer algoritmo.

A partir de los valores obtenidos en el ejemplo, podemos determinar que el segundo algoritmo es *mejor* que el primero, independientemente de la máquina que utilicemos o del compilador que tengamos.

9.1.2. Concepto de talla de un problema

A parte de la problemática debida al hecho de medir el tiempo de ejecución del algoritmo en función de sus operaciones básicas, también hay que tener en cuenta que el coste del algoritmo puede depender de los datos de entrada del algoritmo. Diferentes datos de entrada pueden llevar a tiempos de ejecución distintos.

Ejemplo 2: Realizar un algoritmo que calcule

$$y=\sum_{i=1}^{n} i$$

En este caso el algoritmo tendrá como entrada el valor de `n` y como salida el valor de `y`.

`/*1*/ y ← 0`	t_a
`/*2*/ i ← 1`	t_a
`/*3*/ Mientras (i ≤ n) Hacer`	$n*t_c$
`/*4*/ y ← y + i`	$n*(t_o+t_a)$
`/*5*/ i ← i + 1`	$n*(t_o+t_a)$
`/*6*/ Fin_mientras`	t_c

Si al igual que antes se suman los tiempos de las operaciones elementales, se obtiene:

$Tiempo = 2t_a + n * (t_c + 2t_o + 2t_a) + t_c$

Como era de esperar, el tiempo total que consumirá el algoritmo para obtener el resultado depende del valor `n`, que es una entrada del algoritmo. De manera que, el coste temporal del algoritmo será distinto para diferentes valores de `n`.

Definición 3:

Se denomina ***talla*** de un problema al valor o conjunto de valores asociados a la entrada del problema y que representa una medida del tamaño (coste) del problema respecto de otras entradas posibles.

En el resto del tema se analizarán dos problemas algorítmicos particulares, búsqueda y ordenación, que servirán como ejemplo para el análisis de algoritmos. Los dos son problemas típicos que pueden ser resueltos desde diferentes aproximaciones, lo que permite por un lado, presentar una colección de algoritmos básicos para un informático y, por otro, realizar un análisis comparativo de diferentes algoritmos sobre un mismo problema.

9.2. Recuperación de información

Uno de los principales problemas a los que hay que enfrentarse cuando se gestiona gran cantidad de información es la búsqueda de un elemento concreto dentro de un conjunto (grande) de datos.

El problema de la búsqueda es un ejemplo típico en algoritmia y constituye un problema que aparece con gran frecuencia.

El métodos a emplear para abordar la búsqueda depende básicamente de la organización interna en el conjunto de datos que se esté empleando. Si asumimos que los datos están almacenados en un array, es posible que la información esté organizada de manera ordenada o que, por el contrario, no siga ningún criterio de orden en su almacenamiento. Veamos inicialmente esta última situación.

9.2.1. Búsqueda secuencial

La búsqueda secuencial se aplica cuando no existe ningún conocimiento previo sobre la ordenación de los elementos del conjunto en donde se va a realizar la búsqueda. Como no hay organización en los datos no es posible aplicar un método diferente a examinar todos los elementos del conjunto en busca del elemento deseado.

La idea general del algoritmo se puede reflejar en la siguiente función (C++):

```
bool BusquedaSecuencial (Vector v, int n, int x)
//v es el vector que contiene los datos
//n es el número de elementos en el vector
//x es el dato buscado
{
   int i;                      /*1*/
   bool enc = false;           /*2*/
   i = 0;                      /*3*/
   while (i < n)               /*4*/
   {
      if (v[i] == x)           /*5*/
         enc = true;           /*6*/
      i = i + 1;               /*7*/
    }
   return enc;                 /*8*/
}
```

La función anterior asume que los datos son de tipo entero y que, como resultado de la búsqueda, sólo interesa saber si elemento `x` está o no en el vector.

Si se analiza el algoritmo (sólo la parte esencial, es decir la parte que realiza la búsqueda), se obtiene que el número de operaciones realizadas en cada una de las líneas es:

```
/*2*/   t_a
/*3*/   t_a
/*4*/   n*t_c +  t_c (repeticiones + salida)
/*5*/   n*t_c
/*6*/   ¿?
/*7*/   n*(t_c + t_a)
```

El problema para concluir el análisis reside en que no sabemos cuantas veces se ejecutará la asignación de la línea 6, que depende del número de veces en que la condición de la línea 5 se evalúe como cierta. Este valor no depende del número de elementos del vector, sino que depende del valor de x (dato buscado) y de los elementos almacenados en el vector. Esto nos lleva a concluir que el coste del algoritmo no depende sólo de la talla del problema sino también de la configuración de los datos de entrada.

En este punto es donde aparece el concepto de mejor caso, peor caso y caso medio. Se dice que un algoritmo tiene un mejor caso cuando los parámetros del problema lleven a realizar el menor número de pasos posibles, y el tiempo que tarda el algoritmo en resolver el problema se representará por T^m. Se dice que un algoritmo tiene un peor caso cuando los parámetros del problema lleven a realizar el máximo número de pasos posibles, y el tiempo que tarda el algoritmo en resolver el problema se representará por T^p. Por último, el llamado coste en el caso medio (o coste esperado) de un algoritmo vendrá expresado por la media de pasos realizados en función de todos los casos posibles (básicamente se deben analizar todos los casos posibles teniendo en cuenta la probabilidad de que ocurra cada uno de esos casos). El tiempo medio se representará por T^μ.

Definición 4

Una instancia de un problema corresponde a todas las configuraciones diferentes de la entrada, de una talla determinada, que dan lugar al mismo comportamiento del algoritmo.

En el caso que nos ocupa, el mejor de los casos se dará cuando la condición de la línea 5 siempre se evalúe como falso y no se realice nunca la operación de la línea 6, es decir, el elemento x no se encuentra en el vector. En ese caso, sumando el coste de cada una de las líneas, excepto la correspondiente a la asignación (línea 6), se tendrá que:

$$T^m = 2t_a + n * (2t_c + t_o + t_a) + t_c$$

Agrupando términos en función de la dependencia que tienen en n se obtendría:

$$T^m = (2t_c + t_o + t_a) * n + (t_c + 2t_a) = A * n + B$$

Siendo A y B constantes independientes de la talla.

Como se ve, en el peor caso, el coste del algoritmo depende linealmente con n. Por lo tanto, si la talla crece, el coste seguirá su mismo ritmo de creciemiento.

El peor de los casos para el algoritmo de búsqueda secuencial se daría cuando los elementos del vector fuese tales que hiciesen que la condición de la línea 5 siempre fuera cierta y, por tanto, también se ejecutaría siempre la asignación de la línea 6. En este caso, el vector estaría compuesto por el valor x buscado repetido n veces. El coste temporal en este caso sería:

$$T^p = 2t_a + n * (2t_c + t_o + 2t_a) + t_c$$

Al igual que antes, agrupando términos en función de su dependencia con n, se obtiene:

$$T^p = (2t_c + t_o + 2t_a) * n + (t_c + 2t_a) = A' * n + B$$

La diferencia entre un caso y otro estriba exclusivamente en las constantes A y A' ($A' > A$) y el coste temporal estimado es una función lineal de la talla del problema en ambos casos.

Búsqueda secuencial con parada

El algoritmo anterior es susceptible de ser mejorado de diferentes maneras. Una mejora obvia consiste en finalizar la búsqueda en el momento en que se encuentre el elemento buscado, ya que entonces no tiene sentido seguir con la búsqueda en el resto del vector. La parte fundamental del algoritmo se podría escribir de la siguiente manera:

```
i = 0;
while ( (i < n) && (v[i] != x) )
   i = i + 1;

if (i == n)
   enc = false;
else
   enc = true;
```

Con esta modificación, el bucle debe controlar dos condiciones, que representan los dos motivos por los que se puede finalizar la búsqueda: porque se ha encontrado el elemento o porque se ha alcanzado el final del vector sin encontrarlo.

Si se analizan los dos casos extremos del algoritmo (mejor y peor), se obtendría que el mejor caso se da cuando el elemento x se encuentra en la primera posición del vector. En ese caso, la primera vez que se evalúe la doble condición de la línea 2 ésta fallará, puesto que se cumple que $v[0]$ es igual a x. Por tanto, el coste será:

$$T^m = 2t_a + 3t_c$$

que se puede ver que no depende de la talla del problema y es lo que se conoce como un coste constante (número fijo de operaciones).

El peor caso se dará cuando el elemento x no se encuentra en el vector y, por tanto, es preciso comprobar hasta el final todos sus elementos. El coste de esta situación sería:

$$T^p = (2t_c + 2t_o + 2t_a) * n + (2t_c + 2t_a)$$

Este coste es ligeramente peor que el del primer algoritmo, ya que se duplican el número de comparaciones. Para reducir el número de operaciones para encontrar un elemento se ha empeorado la situación cuando el elemento no se encuentra.

A continuación se analiza una alternativa típica a esta versión de la búsqueda que intenta reducir el número de comparaciones a realizar en el bucle.

Búsqueda secuencial con centinela

En el anterior algoritmo se ha visto que una de las condiciones que hay que comprobar es que el índice permanezca en el rango adecuado (es decir, entre 0 y $n-1$) y la búsqueda no continúe fuera de este rango. Esta comprobación no sería necesaria si se pudiera asegurar que se va a encontrar el elemento buscado.

Se puede estar seguro de encontrar el elemento, si en un momento determinado se ubica "manualmente" en una posición del vector, por ejemplo, en la posición n.

Con ello, la condición de que el índice permanezca en el vector no es necesaria, y el algoritmo tendría una comparación menos cada vez que se pasa por el bucle. Este algoritmo,

conocido por el nombre búsqueda con centinela (por la utilización de un elemento que juega ese papel) queda como sigue:

```
v[n] = x;
i = 0;
while (v[i] != x)
   i = i + 1;

if (i == n)
   enc = false;
else
   enc = true;
```

Se puede comprobar fácilmente que los casos mejor y peor del algoritmo coinciden con los de la versión sin centinela y que el coste en estos casos es:

$T^m = 3t_a + 2t_c$

$T^p = (t_c + t_o + t_a) * n + (3t_c + 4t_a + t_o)$

9.2.2. Búsqueda binaria (dicotómica)

Si se sabe que el vector está ordenado se puede aprovechar esta situación para mejorar sustancialmente el algoritmo de búsqueda. Si en lugar de iniciar la búsqueda en el primer elemento del vector se inicia en una posición central del mismo, es posible determinar, si no se ha encontrado ya el elemento, en que mitad del vector se puede encontrar, descartando la búsqueda en la otra mitad.

En un vector ordenado se cumple que:

`v[i]≤v[j], si j>i, 0≤i,j<n`

`v[i]≥v[k], si k<i, 0≤i,k< n`

entonces si para el elemento buscado *x* se cumple que `x≠v[i]` y `x<v[i]` resulta que, si *x* está en el vector, *x* sólo se podrá localizar en una posición `v[k]`, tal que `k<i`. Por el contrario, si se cumple que `x≠v[i]` y `x>v[i]` resulta que, si x está en el vector, *x* sólo se podrá localizar en una posición `v[j]`, tal que `j>i`. Aplicando esta idea repetidamente sobre el vector se obtiene el siguiente algoritmo:

```
bool BusquedaBinaria (Vector v, int n, int x)
//v es el vector que contiene los datos
//n es el número de elementos en el vector
//x es el dato buscado
{
   int izq, der, cen;
   bool enc = false;

   izq = 0;
   der = n - 1;
   cen = (izq + der) / 2;

   /*
    * Mientras no se encuentre el elemento
    * y existan más de dos elementos en el subvector continúa
    * la búsqueda
    */
```

```
      while ( (izq <= der) && (v[cen] != x) )
      {
         if (x < v[cen])
            der = cen - 1;
         else
            izq = cen + 1;
         cen = (izq + der) / 2;
      }
      /*
       * Se puede salir del bucle por haber encontrado el
       * elemento o por haber llegado a un subvector de un solo elemento
       */
      if (izq > der)
         enc = false;
      else
         enc = true;

      return enc;
   }
```

En este algoritmo en el mejor de los casos el elemento a buscar estaría situado en el centro del vector y el coste no dependería de la cantidad de elementos contenidos en el vector (coste constante).

En el peor de los casos, se obtendría una división sucesiva del vector en mitades, hasta llegar en el límite a un único elemento (o ninguno). Como el número de veces que es posible dividir por dos un valor n y obtener un resultado mayor que cero es $lg_2 n$ resulta el máximo número de elementos que es preciso comparar en el vector en el peor de los casos será $lg_2 n$**.** En el caso medio el coste también es logarítmico, pero la constante que afecta al termino es la mitad que en el caso anterior.

La cota obtenida para el coste temporal de este algoritmo es mucho mejor que la obtenida para las distintas versiones de la búsqueda secuencial, puesto que el coste logarítmico supone incremento lento del coste con la talla, en comparación con una dependencia lineal. Para observar mejor este efecto, en la siguiente tabla se muestra el número de pasos que realizaría un algoritmo en función de la talla del problema y de la dependencia del coste con la misma:

	Coste del algoritmo (nº de operaciones)		
Talla (n)	***n^2***	***n***	***lg n***
10	100	10	4
20	400	20	5
100	10.000	100	7
1.000	1.000.000	1.000	10
1.000.000	1.000.000.000.000	1.000.000	20

Se puede comprobar que si la talla crece desde $n_0=10$ hasta $n_4=1.000.000$ (cien mil veces mayor que n_0), el coste sólo crece cinco veces respecto al coste incial, si la dependencia con la talla es logarítmica, pero crece cien mil veces si la dependencia es lineal y hasta diez mil millones (!) de veces si la dependencia es cuadrática.

9.3. El problema de la ordenación

Para poder aplicar un método de búsqueda tan eficiente como la búsqueda binaria sobre un vector es preciso que este vector se encuentre ordenado. El problema de la ordenación de vectores es también un problema típico en el estudio de algoritmos y es el que se presenta en esta sección.

La ordenación consiste en reorganizar los elementos de un vector de manera que se cumpla que:

$$v[0] \leq v[1] \leq \ldots \leq v[i] \leq v[i+1] \leq \ldots \leq v[n-1]$$

Para resolver el problema se van a estudiar distintos métodos. En todos ellos, el proceso de ordenar va a implicar necesariamente la comparación de elementos del vector y la asignación de elementos a posiciones dentro del vector distintas de las originales. De manera que, a la hora de analizar y comparar los diversos algoritmos presentados nos centraremos en la estimación del número de comparaciones y asignaciones sobre elementos del vector que cada una realiza y no se tendrán en cuenta otro tipo de operaciones, generalmente sobre variables auxiliares (control de bucles, normalmente).

Los tres primeros métodos forman parte de la familia de los llamados algoritmos directos de ordenación y el cuarto es el conocido como método rápido de ordenación (*quicksort*). Los tres métodos directos tienen una filosofía inicial común: el vector tiene dos partes, una ordenada y otra desordenada. Inicialmente la parte ordenada ocupa todo el vector y la parte ordenada está vacía. Repetidamente, se va a hacer disminuir la parte desordenada para incrementar la parte ordenada, hasta que ésta llegue a ocupar todo el vector. Cada uno de los métodos difiere de los demás en la forma en que se hace crecer la parte ordenada.

9.3.1. Método de Inserción

El algoritmo de inserción directa se basa en la idea de ir insertando en cada iteración un nuevo elemento de la parte desordenada en la parte ya ordenada del vector. En el ejemplo siguiente se puede visualizar el proceso:

Situación inicial del vector

53	22	41	11	-3	37	0	15	38	5

Paso 0: parte ordenada formada por 1 elemento, el último

									5

Paso 1: ¿Dónde se sitúa, dentro de la parte ordenada, el primer elemento de la parte desordenada? 38

								38	5
								5	(38)

Repetir el paso 1 para cada elemento de la parte desordenada

15

							15	5	38
							5	(15)	38

0

						0	5	15	38
						(0)	5	15	38

37

					37	0	5	15	38
					0	5	15	(37)	38

-3

				-3	0	5	15	37	38
				(-3)	0	5	15	37	38

11

			11	-3	0	5	15	37	38
			-3	0	5	(11)	15	37	38

41

		41	-3	0	5	11	15	37	38
		-3	0	5	11	15	37	38	(41)

22

	22	-3	0	5	11	15	37	38	41
	-3	0	5	11	15	22	37	38	41

53

53	-3	0	5	11	15	22	37	38	41
-3	0	5	11	15	22	37	38	41	53

De esta manera, el algoritmo que ordenaría un vector *v* de *n* elementos siguiendo el criterio mostrado queda reflejado en la siguiente función:

```
void OrdenarInsercion (Vector v, int n)
{
      int i, j;

      //la parte ordenada está al final de vector
      for (i = n-2; i>=0; i--)
      {
            v[n] = v[i]; //centinela            /*1*/
            j = i+1;
            while ( v[j] < v[n] )               /*2*/
            {
                  v[j-1] = v[j];                /*3*/
                  j++;
            }
            v[j-1] = v[n];                      /*4*/
      }

}
```

En el algoritmo se introduce un elemento centinela para, al igual que en la búsqueda secuencial, reducir el número de comparaciones a realizar.

Si el estudio de la complejidad se centra en aquellas operaciones que manipulan elementos del vector, habrá que fijarse en las líneas numeradas del algoritmo.

La asignaciones de la líneas marcadas como 1 y 4 se ejecutarán un número fijo de veces, puesto que están dentro de un bucle `for` controlado por el valor de la talla, *n*. El número de veces que itera el bucle es exactamente *n-1* y, por tanto, la líneas 1 y 4 contribuyen al coste del algoritmo (en cualquier caso) con *2(n-1)* asignaciones.

El número de veces que se ejecuta el bucle interior (marcado como línea 2) depende de la condición *v[j] < v[n]*, de la cual dependerán los posibles casos del algoritmo (mejor y peor)

En el mejor de los casos, no se entrará ninguna vez en el bucle (en todos los casos el elemento a insertar es el primero de la parte ordenada, es decir, el vector ya está ordenado), con lo que la línea 2 sólo contribuirá al coste con una comparación y la línea 3 no se llegará a ejecutar. Por tanto, el coste en el mejor caso del algoritmo sería:

$Asig^m = 2(n-1)t_a$

$Comp^m = (n-1)t_c$

En el peor de los casos, el bucle interno finaliza tras hacer todas las comparaciones y asignaciones posibles (sólo cuando el vector está inversamente ordenado). Si se analiza para cada valor de `i` el número máximo de veces que se puede entrar en el bucle de la línea 3 se obtiene que:

`i= n-2`, entra 2 veces

`i= n-3`, entra 3 veces

...

`i= 0`, entra n veces

Por tanto, el número total de veces que se entra el bucle para todos los valores de i será:

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

y entonces el coste de asignaciones y comparaciones del algoritmo será:

$$Asig^p = \left[\frac{n}{2}(n+1)-1+2(n-1)\right]t_a = \left(\frac{n}{2}(n+5)-3\right)t_a$$

$$Comp^p = \left[\frac{n}{2}(n+1)-1+(n-1)\right]t_c = \left(\frac{n}{2}(n+3)-2\right)t_c$$

En este caso, el coste del algoritmo de inserción presenta una dependencia cuadrática con `n` (ver tabla de costes de la sección anterior).

Se puede demostrar que el coste en el caso medio de este algoritmo es también cuadrático, es decir, se encuentra más cerca del peor caso que del mejor.

9.3.2. Método de Selección

El algoritmo de selección se basa en la idea de, para todas las posiciones del vector, seleccionar el elemento que debería estar en esa posición cuando el vector esté ordenado. Si se inicia el proceso en la primera posición, encontrar el elemento que deberá ocupar esa posición con el vector ordenado se reduce a buscar el menor elemento de todos los del vector. Este criterio se puede extender a todas las posiciones, teniendo en cuenta que la búsqueda del mínimo elemento se debe reducir al conjunto de elementos que todavía no están en la zona ordenada. El siguiente ejemplo muestra el proceso de ordenación de un vector:

Situación inicial del vector

53	22	41	11	-3	37	0	15	38	5

Paso 1: ¿Qué elemento debe ocupar la primera posición del vector ordenado? El mínimo

-3	22	41	11	***53***	37	0	15	38	5

Repetir el paso 1 para todas las posiciones del vector

-3	**0**	41	11	53	37	***22***	15	38	5
-3	0	**5**	11	53	37	22	15	38	***41***
-3	0	5	**11**	53	37	22	15	38	41
-3	0	5	11	**15**	37	22	***53***	38	41
-3	0	5	11	15	**22**	***37***	53	38	41
-3	0	5	11	15	22	**37**	53	38	41
-3	0	5	11	15	22	37	**38**	***53***	41
-3	0	5	11	15	22	37	38	**41**	***53***

La función en C++ que realizaría esta ordenación sería:

```
void OrdenarSeleccion (Vector v, int n)
{
   int i, j, pos_min;
   int i_aux;

   for (i = 0; i < n - 1; i++)
   {
      pos_min = i;
      for (j = i + 1; j < n; j++)
         if (v[j] < v[pos_min])          /*1*/
            pos_min = j;

      if (pos_min != i)
      {
         i_aux = v[i];                   /*2*/
         v[i] = v[pos_min];              /*3*/
         v[pos_min] = i_aux;             /*4*/
      }
   }
}
```

Si se analiza esta función centrándose exclusivamente en las operaciones que afectan a elementos del vector, se obtiene que: la línea identificada como 1 se debe ejecutar siempre, con independencia de como esté organizado el vector, mientras que las líneas 2, 3 y 4 (que representan un intercambio) sólo serán accesibles cuando el mínimo se encuentre en una posición diferente a la tratada en cada iteración, `i`. Por lo tanto, el mejor caso se dará cuando el vector esté ordenado (no hay ningún intercambio) y el peor cuando el vector esté inversamente ordenado (un intercambio para cada valor de `i`).

El bucle externo (`i`) se ejecuta `n` veces y el bucle interno (`j`) se realizará $\sum_{k=0}^{n-1} k = \frac{n}{2}(n-1)$ veces. Entonces se cumple que:

$Asig^m = 0$

$Asig^p = 3n$

$Comp^m = Comp^p = \frac{n}{2}(n-1) = \frac{1}{2}(n^2 - n)$

Hay que destacar que el número de comparaciones no depende de la ordenación inicial del vector y depende siempre de forma cuadrática con la talla, mientras que el número de asignaciones presenta en el peor de los casos una dependencia lineal.

9.3.3. Método de Intercambio (o de la burbuja)

La idea básica de este algoritmo es ordenar repetidamente pares de elementos. Para que esta ordenación de parejas de elementos llegue a ordenar todo el vector es preciso que las parejas tengan elementos solapados, es decir, si primero se realiza la ordenación relativa de la pareja formada por los elementos `v[0]` y `v[1]`, a continuación se debe proceder a ordenar la pareja `v[1]` y `v[2]`, `v[2]` y `v[3]` y así sucesivamente. La ordenación de las parejas es muy sencilla, simplemente ha que comprobar si el elemento con menor índice de la pareja corresponde también con el que contiene el menor dato. Si no es así, hay que intercambiar la posición de ambos. Si se visualiza el vector en posición vertical, el método consiste en llevar los elementos más ligeros (los que contienen claves de ordenación más pequeñas) hacia arriba, o al contrario llevar los elementos más pesados hacia abajo. Todo depende de la dirección en que se hacen las comparaciones.

En el ejemplo se pueden ver los primeros pasos del algoritmo. No se muestra el proceso completo debido a la longitud del mismo

Situación inicial del vector

53	22	41	11	-3	37	0	15	38	5

Paso 1: Comparar primera pareja de elementos en forma ascendente

53	22	41	11	-3	37	0	15	**38**	**5**

Como no están en el orden correcto, intercambiar

53	22	41	11	-3	37	0	15	***5***	***38***

Paso 2: repetir el proceso para todas las parejas de elementos

53	22	41	11	-3	37	0	**15**	**5**	38
53	22	41	11	-3	37	0	***5***	***15***	38
53	22	41	11	-3	37	**0**	**5**	15	38

53	22	41	11	-3	37	0	**15**	**5**	38
53	22	41	11	-3	**37**	**0**	15	5	38
53	22	41	11	-3	***0***	***37***	15	5	38
53	22	41	11	**-3**	**0**	37	15	5	38
53	22	41	**11**	**-3**	0	37	15	5	38
53	22	41	***-3***	***11***	0	37	15	5	38
53	22	**41**	**-3**	11	0	37	15	5	38
53	22	***-3***	***41***	11	0	37	15	5	38
53	**22**	**-3**	41	11	0	37	15	5	38
53	***-3***	***22***	41	11	0	37	15	5	38
53	**-3**	22	41	11	0	37	15	5	38
-3	***53***	-3	41	11	0	37	15	5	38

Tras este primer recorrido del vector, el elemento más pequeño se ha colocado en su posición definitiva y no es preciso volver a considerarlo en la ordenación, por lo que ahora se procede a realizar una nueva secuencia de comparaciones e intercambios pero sobre un vector con un elemento menos. Si el proceso se repitiera *n-1* veces el vector quedaría ordenado.

Realizando, como en el ejemplo, las comparaciones en forma ascendente se obtiene el algoritmo siguiente:

```
void OrdenarIntercambio (Vector v, int n)
{
   int i, j;
   int i_aux;

   for (i = 1; i < n - 1; i++)
   {
      for (j = n - 1; j > i - 1; j--)
      {
         if (v[j - 1] > v[j])            /*1*/
         {
            i_aux = v[j];                /*2*/
            v[j] = v[j - 1];             /*3*/
            v[j - 1] = i_aux;            /*4*/
         }
      }
   }
}
```

Para analizar el coste del algoritmo debemos tener en cuenta que las operaciones relevantes se encuentran dentro del bucle interno (`j`). Por lo tanto, el número de veces que se ejecuten estas operaciones dependerá del número de veces que se entre en el cuerpo del bucle, este valor es igual a: $\sum_{k=0}^{n-1} k = \frac{n}{2}(n-1)$

La comparación de la línea 1 se ejecutará siempre, por lo tanto:

$$Comp^m = Comp^p = \frac{n}{2}(n-1) = \frac{1}{2}(n^2-n)$$

Sin embargo, el intercambio realizado en las líneas 2, 3 y 4 sólo se realizará cuando la condición anterior sea cierta. En el mejor caso, la condición de la línea 1 siempre será falsa y en el peor caso, siempre será cierta. Entonces:

$$Asig^m = 0$$

$$Asig^p = 3\frac{n}{2}(n-1) = \frac{3}{2}(n^2-n)$$

En este caso, al igual que en el método de selección, el número de comparaciones es fijo y depende de la talla elevada al cuadrado, mientras que el número de asignaciones puede ser cero, en el mejor caso, y depende cuadráticamente con la talla en el peor (y en caso medio).

Este algoritmo es susceptible de realizar diversas mejoras. Entre ellas cabe destacar la posibilidad de intrducir una condición de parada cuando se detecta que en un recorrido completo del vector no se realiza ningún intercambio, lo que significa que ya está ordenado, o el alternar recorridos ascendentes y descendentes para tratar de manera homogénea los elementos grandes y los pequeños (método de la sacudida).

9.3.4. Quick-sort

En los métodos directos vistos anteriormente se establece de entrada la condición de ordenación sobre el vector, ésto implica que para cualquier posición `i` del vector se cumpla que:

```
v[i]≤v[j], si j>i, 0≤i,j<n

v[i]≥v[k], si k<i, 0≤i,k< n
```

El método rápido de ordenación, que se presenta ahora, se basa en realizar una partición del vector que cumple una condición más relajada que la vista en el párrafo anterior. Consiste básicamente en exigir lo mismo pero para un único elemento del vector, no para todos. Este elemento especial, generalmente, recibe el nombre de pivote y lo que realiza el proceso de partición del vector es asegurar que los elementos situados a la izquierda del pivote no sean mayores que él y que los elementos situados a su derecha no sean menores que el pivote. No se exige que los elementos a la izquierda o a la derecha del pivote estén ordenados. Obviamente, realizar sólo una partición de este tipo no ordena el vector, puesto que no es su objetivo, sin embargo si se aplica repetidamente el mismo proceso sobre cada una de las partes que se van obteniendo, en el límite se acabará llegando a subvectores de un elemento y si que se puede asegurar que el vector finalmente quede ordenado. En el siguiente ejemplo se puede ver el proceso:

Situación inicial del vector

53	22	41	11	-3	37	0	15	38	5

Paso 1: Seleccionar el pivote, 11

53	22	41	**11**	-3	37	0	15	38	5

Paso 2: Buscar elementos mal ubicados a la izquierda (>= pivote) y elementos mal ubicados a la derecha (<= pivote)

53	22	41	**11**	-3	37	0	15	38	**5**

e intercambiar

5	22	41	**11**	-3	37	0	15	38	***53***

Repetir el proceso hasta que las búsquedas por la izquierda y por la derecha se crucen

5	**22**	41	11	-3	37	**0**	15	38	53
5	***0***	41	11	-3	37	***22***	15	38	53
5	0	**41**	11	**-3**	37	22	15	38	53
5	0	***-3***	11	***41***	37	22	15	38	53

Paso 3: Hacer de nuevo la partición sobre cada una de las dos partes obtenidas

5	0	-3		41	37	22	15	38	53

Pivote 0

5	**0**	-3
5	0	**-3**
-3	0	***5***

Pivote 22

41	37	**22**	15	38	53
41	37	22	**15**	38	53
15	37	22	***41***	38	53
15	**37**	**22**	41	38	53
15	***22***	***37***	41	38	53

Hacer de nuevo la partición sobre cada una de las dos partes obtenidas

15		37	41	38	53

Pivote 41

37	**41**	38	53

37	**41**	**38**	53
37	***38***	***41***	53

Si se visualizan conjuntamente las distintas partes del vector que se han ido obteniendo se observa que éste ya está ordenado:

-3	0	5	15	22	37	38	41	48	53

El algoritmo que realiza las sucesivas particiones del vector es el siguiente:

```
void Particion (Vector v, int izq, int der)
//v es el vector
//izq y der delimitan la zona de v
//sobre la que se realiza la partición
{
   int i, j; //i(j) recorre desde la izquierda(derecha) el vector
   int aux, piv; //asume que v almacena int

   piv = v[(izq + der) / 2]; //elección del pivote (elemento central)
   i = izq;
   j = der;

   //La partición finaliza cuando i>j
   while (i <= j)
   {
      //buscar 1 elemento que no deba estar a la izquierda del pivote
      while (v[i] < piv)
         i++;
      //buscar 1 elemento que no deba estar a la derecha del pivote
      while (v[j] > piv)
         j--;
      //intercambiar los elementos mal ubicados
      if (i < j)
      {
         i_aux = v[i];
         v[i] = v[j];
         v[j] = i_aux;
         i++;
         j--;
      }
      else
         if (i == j)
         {
```

```
                i++;
                j--;
            }
        }

        //Realizar la partición de los subvectores izquierdo y derecho
        //sólo si tienen más de un elemento
        if (izq < j)
            Particion (v, izq, j);
        if (i < der)
            Particion (v, i, der);
    }
```

El método de ordenación consistirá sencillamente en invocar a la función de partición indicando que el proceso se debe realizar sobre todo el vector:

```
void OrdenarQuickSort (Vector v, int n)
{
    QuickSortRec (v, 0, n - 1);
}
```

El análisis de este algoritmo no se va realizar con detalle, pero si que se puede hacer una descripción intuitiva del mismo.

El coste del algoritmo depende de la posición final que ocupe el pivote tras realizar cada una de las particiones. En el peor caso, el pivote siempre queda en un extremo del vector y por lo tanto, la siguiente partición siempre se debe realizar sobre un vector con un elemento menos que el anterior. Obviamente, si este caso se da en cada una de las particiones, es preciso realizar $n-1$ particiones para alcanzar el final del algoritmo. Como en cada partición es preciso examinar cada elemento del subvector, se puede concluir que el coste en el peor caso de este algoritmo depende del cuadrado de la talla (n). Por su lado, el mejor caso del algoritmo se dará cuando en cada partición el pivote ocupe la posición central del subvector. Esto implica que en la siguiente iteración haya que realizar dos particiones sobre subvectores cuyo tamaño será la mitad del tamaño anterior. Siguiendo el mismo razonamiento realizado para el algoritmo de búsqueda binaria, como el número de veces que es posible dividir por dos un valor n y obtener un resultado mayor que cero es $lg_2 n$ resulta que el máximo número de particiones a realizar será proporcional a $lg_2 n$. Como el número de operaciones en cada partición sigue siendo proporcional a la talla n, entonces el coste total en el peor caso será del orden de $n.lg\ n$. Es posible demostrar que este orden de magnitud también es aplicable al caso medio y que, por consiguiente, este algoritmo presenta un comportamiento teórico muy superior al de los algoritmos directos analizados previamente.

9.3.5. Cuadro resumen de costes de los algoritmos de ordenación

Para ilustrar las diferencias en el coste de los diversos algoritmos de ordenación analizados, se muestra el siguiente cuadro resumen que recoge el orden de magnitud del número de operaciones realizadas por cada uno de ellos en función de la configuración del problema y de la talla.

	Asignaciones			***Comparaciones***		
Método	mejor	medio	peor	mejor	medio	peor
Inserción	2 (n-1)	$\frac{1}{4}n^2$	$\frac{n}{2}(n+5)-3$	n-1	$\frac{1}{4}n^2$	$\frac{n}{2}(n+3)-2$
Selección	0	n	3n	$\frac{1}{2}(n^2-n)$		
Intercambio	0	$\frac{3}{4}n^2$	$\frac{3}{2}(n^2-n)$	$\frac{1}{2}(n^2-n)$		
Quicksort	$lg\, n$	$0.69\, n\, lg\, n$	$\frac{3}{8}n^2$	$n\, lg\, n$	$1.38\, n\, lg\, n$	$\frac{n^2}{2}$

Ejercicios:

1. Escribir un algoritmo para sumar los elementos de un vector de tamaño `n` y analizar su coste.
2. Escribir un algoritmo para sumar los elementos de una matriz de tamaño `nxm` y analizar su coste.
3. Dados los siguientes vectores:
   a. `6 11 8 12 1 6 14 2`
   b. `14 12 13 1 4 5 10 6`

   Indicar los pasos que realizarían los algoritmos de ordenación de Inserción, Intercambio, Selección y Ordenación Rápida para ordenarlos.
4. Dados los siguientes vectores:
   a. `1 2 3 4`
   b. `4 3 2 1`

   ¿Cuántos pasos realiza cada uno de los algoritmos de ordenación para proceder a su ordenado?. ¿Cuál es el mejor algoritmo en estos casos?
5. Supongamos un vector con 100 elementos enteros que está ordenado. Si a ese vector le añadimos al final tres nuevos elementos, tal que el vector queda parcialmente desordenado, ¿cuál de los métodos de ordenación vistos en clase sería más eficiente para volver a ordenar el vector? ¿por qué?
6. Supongamos que estamos ordenando un vector de ocho enteros mediante el algoritmo *quick-sort* y se ha finalizado la primera partición del proceso dejando el vector en el siguiente estado:

   `2 5 1 7 9 12 11 10`

   ¿Qué elemento(s) podría(n) haber sido el pivote utilizado?