Capítulo 10

Introducción al Análisis de Algoritmos

Índice General

"Todos los caminos conducen a Roma, si bien unos son más largos que otros".

Respuesta de examen. Curso 92/93.

10.1. Introducción.

Se plantea el siguiente juego, denominado "*Adivina_mi_número*": un ordenador genera de forma aleatoria un número entre 1 y 100, y el jugador debe acertarlo. Una primera posibilidad es ir conjeturando números al azar o bien, para no repetir ninguno, ir seleccionándolos por orden: 1, 2, 3, 4...

A continuación, se propone una ligera modificación en el juego: el ordenador genera un número de forma aleatoria, pero a la hora de jugar se produce el siguiente diálogo:

- "¡Hola y bienvenido! Debes adivinar un número entre 1 y 100. ¿Qué número es?"
- "20"
- "El número es mayor que 20. ¿Qué número es?"
- "83"
- "El número es menor que 83. ¿Qué número es?"
- .....
- " 39"
- " El número es menor que 39. ¿Qué número es?"
- "37"
- "¡Enhorabuena! Lo has acertado en ... intentos"

En este caso, también se podría ir diciendo números al azar...pero puestos a intentar sacar el máximo provecho de la información de la que se va a disponer, parece que se encontrará antes el número si se selecciona el 50 y, en función de que el número a adivinar sea mayor o menor, entonces se reduzca el intervalo de búsqueda a $[51, 100]$ o a $[1, 49]$ y así, sucesivamente, hasta encontrar el número.

Es decir, de forma intuitiva parece que el segundo método -seleccionar un intervalo cada vez más pequeño de búsqueda[1]- es mejor que el primero -ir diciendo números al azar-. Pero, ¿cuánto mejor es?. Se debería establecer una técnica que permita comparar ambos métodos.

Este ejemplo se puede trasladar al campo de la Informática, ya que *es posible encontrar distintos algoritmos que resuelvan un mismo problema*. Las características de cada uno de ellos puede hacer más o menos atractivo un determinado algoritmo para su implementación como programa, dependiendo de los propósitos perseguidos para dicha implementación. Es necesario, pues, una técnica que permita realizar comparaciones entre algoritmos.

Esta técnica es lo que se conoce como *estudio de la complejidad computacional*, o estudio de los recursos computacionales necesarios para la implementación de un algoritmo y el objetivo del presente tema es presentar sus conceptos elementales.

Básicamente, para estudiar la complejidad computacional de un algoritmo se consideran dos aspectos

la complejidad espacial, que es el número de objetos que maneja el algoritmo; puesto que cada objeto necesita un tamaño de memoria para su representación, esta medida permite conocer la cantidad de memoria que utilizará el programa, y

la complejidad temporal, que es la cantidad de operaciones que hay que realizar; puesto que cada operación se realiza en un tiempo determinado, esta medida permite conocer el tiempo que necesitará el programa para finalizar su ejecución.

Un algoritmo será más eficiente que otro si su complejidad computacional es menor. Pero, ¿cómo se puede determinar la complejidad computacional de un algoritmo? Es decir, ¿cómo se determina que un algoritmo es más eficiente que otro?. Existen dos aproximaciones,

[1] Nota: esta forma de jugar es posible puesto que se conoce la relación de orden que existe entre los números.

Aproximación Empírica (o "a posteriori") consiste en codificar los distintos algoritmos en un determinado lenguaje de programación, y observar su comportamiento.

Aproximación Teórica (o "a priori") consiste en determinar matemáticamente la complejidad computacional de un algoritmo, antes de su implementación.

De ambas aproximaciones, la que reporta más ventajas es la aproximación teórica. No depende ni del ordenador, ni del lenguaje de programación. Permite evitar el esfuerzo de programar inútilmente un algoritmo ineficaz (y de realizar las pruebas correspondientes). Permite, además, conocer la eficacia de los algoritmos estudiados, sea cual sea el número de datos manejado. Esto es de especial importancia; en una aproximación empírica, se deben realizar numerosas pruebas con diversos datos si se quiere realizar un buen análisis; a medida que aumenta el tamaño de los datos, también crece el tiempo de ejecución. Esto puede suponer o bien un gran esfuerzo computacional o bien que esas pruebas no se realicen, con la consiguiente falta de rigor de los resultados obtenidos: la mayor parte de las veces, la importancia de la eficiencia de un algoritmo se pone de relieve, precisamente, cuando se trabaja con instancias de gran tamaño.

En resumen, la aproximación teórica da una medida imparcial sobre las características del algoritmo estudiado, además de ahorrar trabajo.

10.1.1. Determinación de la Complejidad Computacional: Tamaño del Problema y Operación Elemental.

Para determinar la complejidad computacional de un algoritmo, se debe determinar su complejidad temporal y su complejidad espacial.

Ambas medidas están relacionadas con el **tamaño del problema**, que es el *dato (o conjunto de datos) del problema que da una medida real de la magnitud de una instancia concreta frente a otras posibles*.

Cuanto mayor sea el tamaño del problema, se necesitará más espacio de memoria y se tendrán que realizar más operaciones.

El ejemplo expuesto en la introducción puede ayudar a comprender dicha relación: independientemente de la modalidad de juego escogida, parece que debe ser más rápido adivinar un número entre 1 y 100 que adivinar un número entre 1 y 1000, por ejemplo. Y esto es debido a que en el primer caso hay que considerar 100 posibles valores, mientras que en el segundo hay que considerar 1000 posibles valores. En este ejemplo, la talla del problema será, precisamente, el número de posibles valores a considerar. Así se podría enunciar, de forma general, que en el problema de *adivinar un número entre 1 y N* el tamaño del problema es *N*.

Nótese que, del ejemplo anterior, también se deduce otra relación importante como es que *una vez fijado el tamaño del problema*, es decir, una vez que se ha dado un valor concreto a N, el segundo método parece seguir siendo mejor que el primero. Por lo tanto, *cuando se pretenda realizar la comparación entre dos algoritmos, se ha de realizar para un mismo tamaño del problema*.

La importancia de determinar adecuadamente el tamaño del problema reside en que para determinar la complejidad es necesario determinar su expresión en función del tamaño.

Por lo que respecta a la complejidad espacial, hay que establecer una relación entre el número de objetos manejados y el tamaño del problema. Como unidad se pueden tomar los tipos elementales: el tamaño de un `ENTERO` o de un `REAL` o de un `BOOLE`, es 1. El tamaño de un vector de n componentes

reales, será n; el tamaño de una tupla con tres campos reales será 3 y el tamaño de un vector de n componentes que sean registros con tres campos, será $3n$. Ello supone, por ejemplo, que al considerar el tamaño de los objetos de tipo `CADENA` se considere como unidad cada uno de los caracteres básicos que lo forman; por lo tanto, su tamaño coincidirá con su longitud (si es una cadena de longitud k, su tamaño será k).

Para expresar la complejidad temporal, hay que establecer una relación entre el número de operaciones realizadas y el tamaño del problema. Pero, ¿qué unidad se toma para establecer esa relación?. Es decir, ¿qué se entiende por operación elemental?

Esta discusión no es obvia, ya que dependiendo del objetivo del análisis se podrán definir distintas referencias. Es decir, dependiendo de unos intereses concretos, se debe elegir una operación del algoritmo (o algoritmos) a analizar como *Operación Elemental*. Eso sí, sea cual sea el criterio seguido, una vez fijada la operación elemental debe ser mantenida de forma invariable a lo largo de todo el análisis.

Podrán ser candidatas a operación elemental cualquiera de las operaciones expresadas mediante los operadores elementales (+, -, /, *, los de la división entera (`%` y `/`), `and`, `or`, `not` ...). También se podría considerar cualquier operación específica de una estructura de datos (por ej: acceder a un elemento de un vector o realizar la operación *push* (apilar) en una pila), e incluso operaciones tan sencillas como pueden ser las asignaciones y comparaciones son las más idóneas, por ejemplo, en los algoritmos de ordenación.

En resumen, antes de analizar un determinado algoritmo se determinará cuál es la operación elemental (bien porque sea la más frecuente, o porque sea la que nos permita realizar comparaciones con otros algoritmos, o bien porque sea una operación especialmente costosa, etc.) y una vez elegida no se podrá cambiar hasta que no haya finalizado la determinación de la complejidad temporal.

Como medida de la complejidad se obtendrá, normalmente, una función $f(N)$, a la que se denominará *función de coste*, en la que N es el tamaño del problema. Funciones típicas son, por ejemplo,

$$f(N) = aN^3 + bN^2,\ o\ f(N) = a2^N,\ o\ f(N) = a\log_2 N.$$

La gran ventaja de este análisis es que no sólo se expresa así el número de objetos manejados o el número de operaciones ejecutadas en función del tamaño del problema, sino que, además, permite obtener la medida de cómo afecta un cambio en el valor de N a la complejidad.

Nótese que tanto al hablar de medidas de complejidad espacial como al hablar de medidas de complejidad temporal, se han establecido unas medidas absolutas que no dependen de ninguna implementación concreta. Pero es que esta medida absoluta es mucho más relevante, ya que indica las características de un algoritmo por si mismo. Por lo tanto, permite obtener una buena medida para comparar algoritmos, no implementaciones.

Actualmente se suele conceder mucha más importancia a rebajar el tiempo de ejecución de un programa que a rebajar la cantidad de memoria que ocupan sus variables, ya que la tecnología actual ha abaratado considerablemente el coste de la memoria. Por lo tanto, es habitual que cuando se habla de que un algoritmo es más eficiente que otro, sólo se esté haciendo referencia a que su complejidad temporal es menor. Por ello, el resto del tema se centrará en el análisis de la complejidad temporal, si bien todos los conceptos que se desarrollarán pueden extenderse al análisis de la complejidad espacial.

10.1.2. Algunos Ejemplos Típicos del Cálculo de la Función de Coste.

Una vez introducido el concepto de función de coste, es hora de plantear cómo se obtiene. Como ya se ha comentado, el estudio se centra en el cálculo de la complejidad temporal. El cálculo de la función de coste consiste, básicamente, en llevar cuenta de cuántas veces se realiza la operación elemental.

- Por ejemplo, si en el siguiente algoritmo se toma como operación elemental la multiplicación,

```
void Auxiliar(int a1, int b1, int *a2, int *b2) {
    int aux1, aux2;

    aux1=(a1*(*a2))+(b1*(*b2));
    aux2=(a1*(*b2))+(b1*(*a2))+(b1*(*b2));
    *a2=aux1;
    *b2=aux2;
}
```

  siempre se hacen 5 multiplicaciones; por lo tanto, su función de coste es $f(N) = 5$. Sin embargo, en el algoritmo

```
int fact(int N){
   int f,i;

   f=1;
   for (i=1; i<=N, i++){
      f=f*i;
   }
   return f;
}
```

  se hacen tantas multiplicaciones como valga N: su función de coste es $f(N) = N$.

- Hay funciones de orden muy típicas: las polinómicas suelen derivarse de algoritmos donde se utilizan bucles del tipo

```
     .........
   for(i=0;i<N;i++){
      for(j=0;j<N;j++){
          /*k Operaciones elementales*/
      }
   }
     .........
```

  En este ejemplo hay k operaciones que se repetirán `N` veces ya que se realizan tantas veces como se ejecute el bucle gobernado por el contador `j`; además, este bucle se ejecutará `N` veces, tantas como indica el bucle gobernado por el contador `i`. En total, se obtiene $f(N) = kN^2$.

- Otro ejemplo es el siguiente algoritmo que, a partir de un valor entero, devuelve otro formado al dar la vuelta al número original (si N fuera 357, el valor devuelto sería 753):

```
int Invertir(int N) {
    int aux, r;

    aux = N;
```

```
    r = 0;
    while (aux > 0){
         r = r * 10 + (aux % 10);
         aux = aux / 10;
    }
    return r;
}
```

Se toma la división como operación elemental. El tamaño del problema es `N`, ya que de de su valor depende el número de divisiones a realizar. ¿Cuántas son? El algoritmo finaliza cuando el valor de la variable auxiliar `aux` es igual a `0`. Su valor inicial coincide con el de `N`. En cada iteración el valor de `aux` se divide entre `10`. Por lo tanto, el número de iteraciones del bucle coincide con el número de veces que se puede dividir `N` entre `10` antes de obtener `0` como resultado. Este valor se puede calcular como

$$((\ldots(((NDIV10)DIV10)DIV10)\overset{k)}{\ldots})DIV10) = NDIV10^k = 0 \Leftrightarrow$$
$$k = (\log_{10} N) + 1.$$

El número de iteraciones del bucle coincide con el logaritmo en base `10` de `N` más `1`; en cada iteración se realizan 2 divisiones. Por lo tanto, se obtiene que $f(N) = 2((\log_{10} N) + 1) = 2\log_{10} N + 2$.

Las funciones de coste más típicas suelen tomar forma de *polinomio*, $f(N) = aN^k$, si bien también son muy importantes las funciones *logaritmo*, especialmente en base 2, y *exponencial*, normalmente $f(N) = 2^N$. Más adelante, se introducirán otras funciones típicas, así como el análisis del comportamiento de un algoritmo afectado por una determinada función de coste.

10.2. Análisis en el Peor y en el Mejor Caso.

En este punto y teniendo en cuenta los conceptos introducidos, se debería estar en condiciones de poder establecer formalmente cuál de los dos métodos descritos en la introducción para jugar al juego "Adivina_mi_número" es el mejor.

Para ello, primero se debe elegir la operación elemental; dado el ejemplo, se tomará como operación elemental un intento de adivinar. ¿Cuántos intentos habrá que realizar si se utiliza el primer método -conjeturar números al azar- y cuántos cuando se utiliza el segundo -acotar el intervalo de búsqueda- para adivinar un número entre 1 y 100?

Pues no es posible asegurarlo, a pesar de que ya se ha definido la operación elemental, porque en el juego interviene el azar. Es decir, a la hora de jugar se puede ser muy afortunado y acertar el número en el primer intento. Pero eso no parece ser una medida válida del número de intentos: no siempre se tendrá tanta suerte.

El mismo análisis se podría intentar con un enfoque totalmente pesimista: ¿cuántos intentos se necesitan en ambos casos si se tiene tan mala suerte que el número sólo se descubre al final, cuando ya sólo queda una posible opción?

Al decir números al azar, en un caso de realmente mala suerte se necesitan 100 intentos: se dicen los 99 números que no son, antes del que se tiene que adivinar. Pero al jugar por el segundo método, cada vez se reduce el tamaño del problema a la mitad. La primera vez las posibilidades son 1 entre 100; pero la segunda son 1 entre 50, la tercera 1 entre 25, la cuarta 1 entre 12, la quinta 1 entre 6,

la sexta 1 entre 3... al séptimo intento se adivinará el número. Y ahora sí que se podría establecer una comparación fiable: no siempre se tendrá tan mala suerte, pero con el primer método se puede llegar a necesitar realizar 100 intentos[2], mientras que en el segundo, como mucho, se necesitarán 7 intentos[3]. Por lo tanto, el segundo es mejor método que el primero.

De nuevo se traslada este ejemplo al mundo de la Informática. En los algoritmos no interviene el azar. Pero en los algoritmos puede haber estructuras de control condicionales (`if`, `if/else`) y bucles condicionales (`while`). Es decir, habrá casos en los que la ejecución de una instrucción, o el número de veces que se ejecute, dependa de la evaluación de una condición. Y, en un análisis a priori, no se puede asegurar cuál será el resultado de evaluar las condiciones de un algoritmo ya que dependerá siempre de los valores concretos de los datos y de las variables del entorno.

No se trata de un caso de buena o mala suerte, pero para evaluar estos algoritmos en los que aparecen condicionales y en los que el número de operaciones a ejecutar puede variar entre dos casos particulares de valores diferentes de los objetos tratados hay que realizar, como mínimo, un análisis en el mejor y en el peor de los casos.

El **análisis en el mejor caso** consiste en realizar el *análisis del algoritmo asumiendo que la combinación de los valores que intervienen en la evaluación de las condiciones es lo más favorable posible* (se realizará el menor número posible de operaciones elementales). Para el **análisis en el peor caso** se debe asumir *la peor combinación posible* (se realizará el mayor número posible de operaciones elementales).

Ejemplo 1: En la siguiente secuencia, que determina el máximo entre las variables `a`, `b` y `c`,

```
.........
max = a;
if (b > max) {
   max = b;
}
if (c > max) {
   max = c;
}
.........
```

según que los valores de a, b y c sean

$$\texttt{a} = 100,\ \texttt{b} = 10,\ \texttt{c} = 5 \tag{10.1}$$

o que sean

$$\texttt{a} = 1,\ \texttt{b} = 10,\ \texttt{c} = 500 \tag{10.2}$$

el número de operaciones realizadas es en (10.1) 1 asignación y 2 comparaciones y en (10.2) 3 asignaciones y 2 comparaciones.

[2]En el caso general, adivinar un número entre 1 y N, serían N intentos.
[3]En el caso general, adivinar un número entre 1 y N, serían $\log_2 N$ intentos.

Ejemplo 2: el siguiente fragmento de algoritmo determina el elemento máximo de un vector `x` *de* `n` *componentes,*

```
.........
max = x[0];
for (i=1; i <n; i++) {
   if (x[i] > max) {
      max = x[i];
   }
}
.........
```

Si, por ejemplo,

$$\texttt{x} = \begin{pmatrix} 100 & 25 & 30 & 1 & 2 & 4 & 8 & 10 & 20 & 31 & 32 & 33 & 40 \end{pmatrix}$$

se realizan 1 asignación y 12 comparaciones. Pero si

$$\texttt{x} = \begin{pmatrix} 1 & 2 & 4 & 8 & 10 & 20 & 25 & 30 & 31 & 32 & 33 & 40 & 100 \end{pmatrix}$$

se realizan 13 asignaciones y 12 comparaciones.

En el primer ejemplo, el caso (10.1) representa el mejor caso y el caso (10.2) representa el peor caso, porque se parte de unos valores de las variables que dan lugar a que se realicen todas las operaciones. El segundo ejemplo supone la generalización del problema: el mejor caso (cuando se realizan menos operaciones) será cuando el primer elemento del vector contenga el valor máximo. Si lo que ocurre es que el valor máximo se encuentra en `x[n-1]`, y además el vector está ordenado por orden creciente (`x[0]` $<$ `x[1]` $<$ `x[2]` $<\ldots<$ `x[n-1]`), en cada iteración del bucle será cierto el predicado del condicional y, por lo tanto, se ejecutan todas las asignaciones sobre `max`.

Ejemplo 3: Dado un real `x`*, escribir un algoritmo que determine si existe una copia del valor de* `x` *entre las* `n` *componentes del vector real* `v`*.*

```
boole buscar (float x, float v[], int n) {
   int i;
   boole enc;

   i = 0;
   enc = FALSO;
   while (!(enc) && (i < n)){
      if (v[i] == x)
         enc = CIERTO;
      else
         i = i + 1;
   }
   return enc;
}
```

En este ejemplo, el mejor caso ocurre cuando el valor del primer elemento del vector coincide con el valor de la variable `x`*. Se realizan las dos primeras asignaciones y el bucle sólo se ejecuta una vez (lo que supone realizar 3 comparaciones, 2 operaciones* `not` *y* `and` *y 1 asignación).*

El peor caso, será cuando no hay ningún elemento de `x` *con el mismo valor de* `x`*, ya que hay que recorrer todo el vector. En total, se realizan 2 asignaciones (las dos primeras),* `n+1` *comparaciones y* `n+1` *operaciones* `not` *y* `and` *(en la evaluación del predicado del bucle* `while`*) y* `n` *comparaciones,* `n` *sumas y* `n` *asignaciones dentro del cuerpo del bucle.*

Es importante darse cuenta de que

> *el mejor y el peor caso no dependen del tamaño del problema*, sino de que varíe el número de instrucciones que se ejecute en cada caso, sin que varíe el tamaño del problema.

Para un mismo tamaño se puede presentar una buena o una mala combinación de los valores de entrada. De hecho, esto es lo que ocurría en los ejemplos anteriores, en los que el tamaño no varía del mejor al peor caso; lo que varía son las combinaciones de los valores concretos con los que se trabaja.

El análisis en el peor de los casos da una buena medida del comportamiento del algoritmo, sobre todo cuando los requerimientos en cuanto a velocidad de respuesta son críticos. Hoy en día, en un gran número de aplicaciones informáticas, obtener un tiempo de respuesta pequeño, es decir, que un programa proporcione los resultados en el menor tiempo posible, es muy importante. Baste pensar en un programa que gobierne la seguridad en una central nuclear (la respuesta ante cualquier fallo detectado debe ser inmediata), o en cualquiera de los programas que gobiernan los automatismos de un ingenio espacial. Ante un algoritmo como los anteriores, si se asegura cual será su comportamiento en el peor de los casos, se asegura que, en cualquier otra circunstancia, su respuesta aún será más rápida. Con el análisis en el peor de los casos, se da una cota superior a su tiempo de respuesta de cara a una implementación.

Ello no quita para que el análisis completo de un algoritmo deba contemplar también el **análisis en el caso medio**, que da una medida más real del comportamiento habitual del algoritmo. Para realizar este análisis, se consideran todos los casos posibles, el número de operaciones de cada caso y la probabilidad de que ocurra (esto puede o no ser factible, dependiendo del algoritmo). Por ejemplo, para determinar el coste en asignaciones en el problema de encontrar el valor máximo de un vector de `n` elementos, existen n casos posibles: que el máximo esté en el primer elemento, en el segundo, ..., o en el n-ésimo. Para realizar el análisis en el caso medio de este algoritmo, se debe considerar el coste en operaciones de cada uno de estos casos y ponderar este coste por la probabilidad de que ocurra. Es decir,

$$\sum_{i=0}^{n-1} núm.\_op.\ caso_i * prob.\ caso_i = \sum_{i=0}^{n-1}(i+1) * \frac{1}{n} = \frac{1}{n} * \frac{n(n+1)}{2} = \frac{n+1}{2}.$$

Este tipo de análisis suele ser mucho más complicado que los anteriores. Además de considerar todos los casos posibles, hay que tener en cuenta que un algoritmo puede formar parte de un entorno global, que conlleve una distribución determinada del valor de los datos. Siguiendo con el ejemplo del máximo de un vector, es posible que este algoritmo tome como datos los resultados de otro algoritmo que provoque una probabilidad del 65 % de que el máximo esté en un determinado elemento del vector, y el 35 % restante se divida por igual entre los demás elementos. Esto tipo de hecho debe tenerse en cuenta al realizar el análisis, ya que ahora cada caso no es equiprobable.

10.3. Órdenes de Complejidad.

Para introducir el concepto de Orden de Complejidad, se propone como ejemplo el cálculo de la complejidad temporal, o coste temporal, de la siguiente secuencia algorítmica, tomando la asignación `a[i][j] = 0`, como operación elemental,

```
.............
for (i=0; i<N; i++)
    for (j=0; j<=i; j++)
        a[i][j] = 0;
.............
```

¿Cuántas veces se ejecuta la operación? Cuando `i` vale 0, se ejecuta una vez (`j`=0), cuando `i` vale 1, se ejecuta dos veces (`j`=0,1), cuando `i` vale 2, se ejecuta tres veces (`j`=0,1,2)... así hasta que `i` valga `N-1`, que se ejecutará `N` veces. Por lo tanto, se puede calcular el número total de ejecuciones como[4],

$$\sum_{cont=1}^{N} cont = \frac{N(N+1)}{2}.$$

Es decir, la función de coste es $f(N) = \frac{1}{2}N^2 + \frac{1}{2}N$.

Conociendo esta relación, se puede predecir cuál será el número exacto de operaciones, dependiendo del valor concreto de N,

N	$\frac{1}{2}N$	$\frac{1}{2}N^2$	a[i][j] = 0
10	5	50	55
100	50	5.000	5.050
1.000	500	500.000	500.500
10.000	5.000	50.000.000	50.005.000

Además, en la tabla se puede observar que, a medida que el valor de `N` crece, la contribución del término $\frac{1}{2}N$ decrece, en comparación con $\frac{1}{2}N^2$. Aún más: la expresión total del número de veces que se ejecuta la operación de asignación, $a[i][j] = 0$, a medida que `N` crece, se parece mucho más a $\frac{1}{2}N^2$ que a $\frac{1}{2}N$.

Es decir, si se pretende estimar el comportamiento de la secuencia anterior para valores de `N` muy grandes, se podría simplificar la expresión obtenida y decir que la operación se realiza, aproximadamente, $\frac{1}{2}N^2$ veces.

Esto es así porque, matemáticamente, se tiene que la función N^2 domina a la función N, para valores de `N` suficientemente grandes (se suele expresar como $N^2 \gg N$), ya que

$$\lim_{N\to\infty} \frac{N}{N^2} = 0.$$

[4] Al margen de este razonamiento, podría haberse resuelto el número de ejecuciones por el procedimiento habitual, resolviendo la expresión derivada de los sumatorios correspondientes a los bucles: $\sum_{i=0}^{N-1} \sum_{j=0}^{i} 1$. Se puede comprobar que se obtiene el mismo resultado.

¿Qué ocurriría al comparar $\frac{1}{2}N^2$ y N^2 ?. O ¿qué ocurriría al comparar N^2 con $100N$? ¿Influye significativamente un valor constante a la hora de ver si una función domina a otra?. Para determinar esto, se construye otra tabla:

N	$\frac{1}{2}N^2$	N^2	$100N$
10	50	100	1.000
50	1.250	2.500	5.000
100	*5.000*	**10.000**	**10.000**
200	**20.000**	*40.000*	**20.000**
500	125.000	250.000	50.000
1.000	500.000	1.000.000	100.000
10.000	50.000.000	100.000.000	1.000.000
100.000	5.000.000.000	10.000.000.000	10.000.000

En la tabla se puede observar:

- la diferencia entre $\frac{1}{2}N^2$ y N^2 siempre se mantiene *constante* ($\frac{1}{2}$),
- para valores pequeños, parece que $100N$ es mayor que las otras dos funciones, pero para valores mayores que 100 siempre será más pequeña que la función N^2 y para valores mayores que 200 siempre será más pequeña que $\frac{1}{2}N^2$.

De todo esto se puede concluir que *la diferencia entre dos algoritmos de costes cN^2 y dN^2, siendo c y d constantes positivas, siempre será proporcional para cualquier valor de N*. Mientras que si se comparan algoritmos cuyas funciones de coste son tales que una domina a la otra, la diferencia varía de forma muy significativa, estén o no las funciones afectadas por constantes más o menos grandes.

Si se tiene esto en cuenta, aún es posible simplificar más la expresión del coste de la secuencia que se estaba analizando: se puede decir que el número de las operaciones que realiza es proporcional a N^2, lo que se expresará con la notación $O(N^2)$, que quiere decir que la función de coste del algoritmo es del orden de la función N^2.

Matemáticamente, el concepto de *orden de $f(N)$*, representa el conjunto de todas las funciones cuyo valor está limitado superiormente por un múltiplo de $f(N)$. En el ámbito de la Informática, este concepto se define como

Definición 10.1 *Un algoritmo tiene un coste de ejecución $O(f(N))$, para alguna función $f(N)$, si, para un valor de N suficientemente grande, el número de operaciones de ese algoritmo nunca es mayor que $cf(N)$, para alguna constante positiva c.*

En la práctica algunos órdenes de complejidad tienden a aparecer de forma muy frecuente. Los más comunes se presentan en la siguiente tabla:

Complejidad	Cuando N vale el doble, el tiempo de ejecución...	**Nombre**
$O(1)$	... no cambia.	Constante
$O(\log N)$	... se incrementa en una constante.	Logarítmico
$O(N)$	... vale el doble.	Lineal
$O(N \log N)$	... vale un poco más del doble.	NlogN
$O(N^2)$	... se incrementa por un factor de 4.	Cuadrático
$O(N^3)$	... se incrementa por un factor de 8.	Cúbico
$O(N^k)$	... se incrementa por un factor de 2^k.	Polinómico
$O(\alpha^N), \alpha > 1$	... se eleva al cuadrado	Exponencial
$O(N!)$	... crece muy rápidamente: es (2N)!	Factorial

En la tabla los órdenes aparecen en orden creciente: cuanto mayor sea el orden de complejidad de un algoritmo, mayor es el número de operaciones que realiza. Por lo tanto, si se dispone de dos algoritmos para resolver el mismo problema, siempre se debe elegir aquel cuya función de coste sea de menor orden.

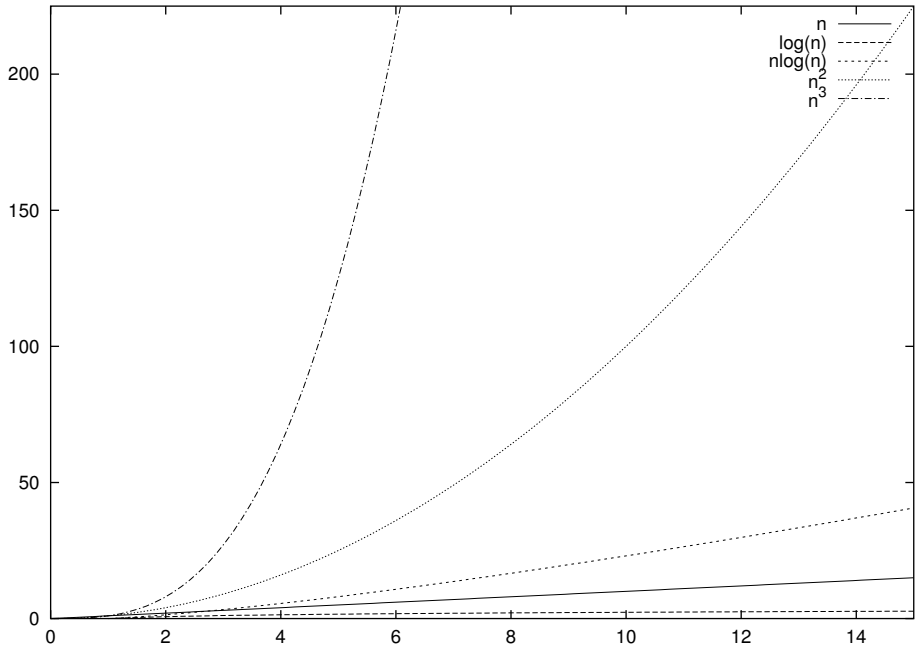


Figura 10.1: Funciones *logarítmica*, *lineal*, $n \log n$, n^2 y n^3.

En las figuras 10.1 y 10.2 se ha comparado el crecimiento de estas funciones.

En la figura 10.1, se compara el crecimiento de las funciones *logarítmica*, *lineal*, $N \log N$, N^2 y N^3. Se puede observar una gran diferencia en el crecimiento de N^3 con respecto a las demás funciones: es mucho más rápido.

La figura 10.2 compara el comportamiento de N^3 con respecto a las funciones *exponencial* y *factorial*. En esta figura se ha cambiado (y mucho) la escala con respecto a la anterior para poder representar completamente la función *exponencial*... y, aún así, no se puede representar la función *factorial* más allá de N=7. En comparación con estas funciones, el crecimiento de N^3, que tan rápido parecía en la figura anterior, es muy lento.

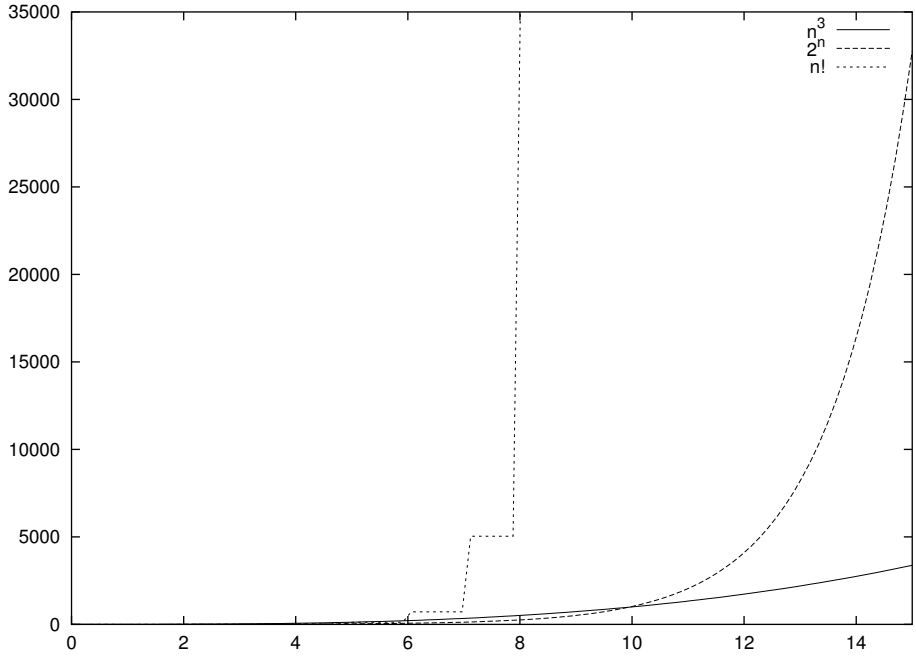


Figura 10.2: Funciones n^3, *exponencial* y *factorial*.

Ni que decir tiene que si al calcular el coste temporal de un algoritmo se obtiene una complejidad de orden *exponencial* o *factorial*, se tendrá que asumir que el algoritmo se convertirá en un programa con tiempo de ejecución prohibitivo.

Para remarcar aún más lo reflejado en las figuras anteriores, se presenta la siguiente tabla:

N	$O(1)$	$O(\log N)$	$O(N)$	$O(N \log N)$	$O(N^2)$	$O(N^3)$	$O(2^N)$	$O(N!)$
1	1	0	1	0	1	1	2	1
2	1	1	2	2	4	8	4	2
3	1	1,5849	3	4,7548	9	27	8	6
4	1	2	4	8	16	64	16	24
5	1	2,3219	5	11,609	25	125	32	120
6	1	2,5849	6	15,509	36	216	64	720
7	1	2,8073	7	19,651	49	343	128	5.040
8	1	3	8	24	64	512	256	40.320
9	1	3,1699	9	28,529	81	729	512	362.880
10	1	3,3219	10	33,219	100	1000	1.024	3.628.800
11	1	3,4594	11	38,053	121	1331	2.048	39.916.800
12	1	3,5849	12	43,019	144	1728	4.096	479.001.600
13	1	3,7004	13	48,105	169	2197	8.192	6,227 e+9
14	1	3,8073	14	53,302	196	2744	16.384	8,7178 e+10
15	1	3,9068	15	58,603	225	3375	32.768	1,3077 e+12

10.4. ¿Por Qué se Buscan Algoritmos Eficaces?

Una ojeada rápida a la evolución de la Informática y, en particular, a la evolución de la tecnología que la soporta, muestra rápidamente que los avances más notables y espectaculares se refieren sobre todo a la mejora de la velocidad de proceso (en concreto, esta mejora ha supuesto un orden superior a 10^9). Por ello, es posible que surja la siguiente cuestión: si los ordenadores son cada vez más rápidos ¿merece la pena buscar algoritmos más eficaces que los disponibles? ¿no bastaría con confiar en los avances técnicos para conseguir mejorar la velocidad de ejecución?.

Para responder a esta cuestión, se plantea el siguiente problema: desarrollar un algoritmo para calcular el valor de x^{2^N}. La solución más directa da lugar al siguiente algoritmo:

```
float intento1 (float x, int N) {
    int i, aux;
    float res;

    res = 1;
    aux = 2^N; /*licencia poética*/
    for (i = 0; i < aux; i++)
       res = res * x;

    return res;
}
```

Se toma la multiplicación como operación elemental. Hay una multiplicación dentro de un bucle cuya ejecución se realiza 2^N veces; es decir, su coste es 2^N multiplicaciones, $O(2^N)$. Se implementa el algoritmo en un computador en el que cada multiplicación se puede realizar en 10^{-4} segundos. Por lo tanto, el tiempo de ejecución será de unos $10^{-4} \times 2^N$ segundos. Según esto, si

N = 10	el tiempo de ejecución es	0.1 segundos
N = 20	el tiempo de ejecución es	2 minutos
N = 30	el tiempo de ejecución es	1 día
N = 38	el tiempo de ejecución es	1 año

Las cantidades expuestas en la tabla reflejan tiempo *ininterrumpido* de cálculo. Si se continuara haciendo números, se podría ver que para resolver un problema con N = 50, el tiempo de ejecución resultante sería de 3570 años.

Como parece excesivo, se implementa el mismo algoritmo en otro computador que es 100 veces más rápido; es decir, el problema se puede resolver en $10^{-6} \times 2^N$ segundos. Un cálculo sencillo muestra que la mejora obtenida supone un año de trabajo ininterrumpido para procesar un problema de tamaño 45 (y 35.70 años de trabajo para el problema de tamaño 50).

La mejora obtenida no se corresponde con las expectativas derivadas del incremento en la velocidad de cálculo. Por lo tanto, se cambia la línea de trabajo, intentando mejorar el algoritmo que resuelve el problema. Al estudiarlo de nuevo, se obtiene el algoritmo:

```
float intento2 (float x, int N) {
    int i;
    float res;
```

```
    res = x;
    for (i = 0; i < N; i++)
       res = res * res;

    return res;
}
```

Este nuevo algoritmo también realiza una multiplicación en cada iteración del bucle. Pero como el bucle se repite N veces, eso supone un coste total de N multiplicaciones, $O(N)$. Por lo tanto, el tiempo de ejecución sobre el primer computador es de $10^{-4} \times N$ segundos. Según esto, si

N = 10	el tiempo de ejecución es	0.001 segundos
N = 20	el tiempo de ejecución es	0.002 segundos
N = 30	el tiempo de ejecución es	0.003 segundos
N = 50	el tiempo de ejecución es	0.005 segundos

En un día de cálculo ininterrumpido, se podría resolver un problema en el que N = 864.000.000; un año de trabajo, permitiría resolver problemas de tamaño 315.360.000.000.

Es decir, el nuevo algoritmo permite una aceleración muchísimo más espectacular de la velocidad de cálculo que la obtenida al multiplicar por 100 la velocidad del procesador. Ni que decir tiene que en el ordenador rápido, el nuevo algoritmo aún será mucho mejor.

10.4.1. Clases de Problemas.

En el Tema 1 ya se comentó que puede haber problemas *no computables*, que no se puedan resolver mediante un algoritmo a ejecutar en un computador. Pero, además, al estudiar los problemas que tienen solución expresable por medio de algoritmos, es posible encontrar (tal y como se ha visto para el problema de cálculo de x^{2^N}) que determinados órdenes de coste se traducen en tiempos de ejecución demasiado altos. Por ello, al hablar de los problemas *computables* se pueden identificar dos clases:

Problemas tratables, aquellos cuya mejor solución algorítmica conocida tiene un coste a lo sumo polinómico.

Problemas intratables, aquellos cuya mejor solución algorítmica conocida es de coste mayor que el polinómico (por ejemplo, exponencial o factorial).

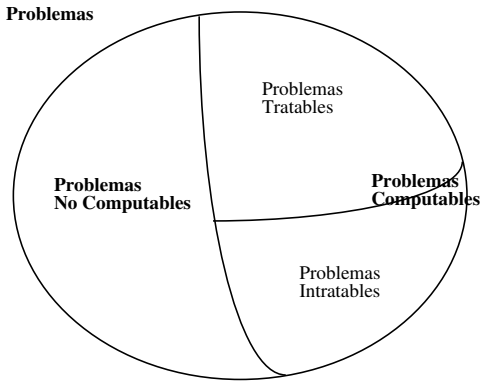
En la clasificación anterior aparece la expresión **mejor solución algorítmica conocida**: se refiere a la solución expresable por medio de un algoritmo que se conozca y que presente el menor coste de entre los algoritmos que permitan resolver dicho problema.

Esto es muy importante, ya que supone que *cualquier problema considerado intratable, dejará de pertenecer a esa clase en el momento en que se pueda demostrar que existe una solución algorítmica con coste polinómico*.

De hecho, esta clasificación plantea una de las líneas de investigación más importante (quizás la más importante) de la Informática Teórica hoy en día. La cuestión a resolver es la siguiente:

¿existen problemas intratables o lo que ocurre es que la comunidad científica no ha sido capaz de encontrar el método correcto para que dejen de serlo?. Hay muchos científicos que trabajan en la cuestión; incluso hay quien postula que esta cuestión nunca se podrá resolver porque es un problema irresoluble.

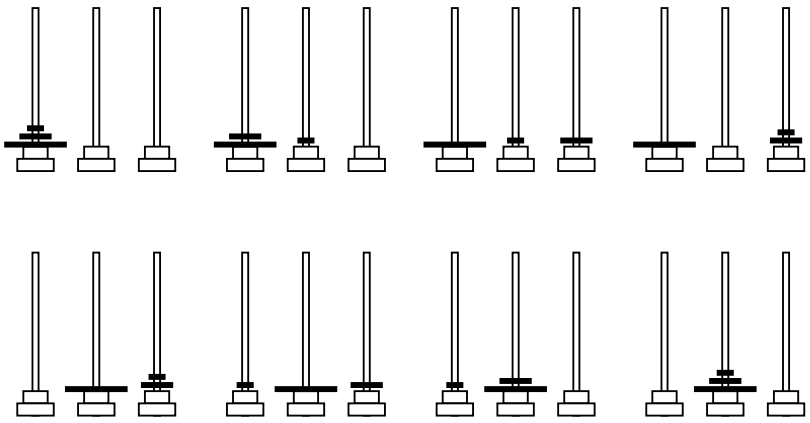
En cualquier caso, el estudio de los problemas, de los algoritmos que los resuelvan y de su complejidad computacional, permite realizar la siguiente descripción gráfica del universo de los problemas:



Dos ejemplos clásicos de problemas intratables son los siguientes:

1. Las Torres de Hanoi: Dadas tres columnas y un número `N` de discos, con diámetros diferentes, a partir de una situación inicial en la que los `N` discos están colocados en la primera columna, se debe pasar a una situación final en la que los `N` discos estén en la segunda sabiendo que nunca puede colocarse un disco encima de otro con diámetro menor. Puede utilizarse la tercera columna como auxiliar.

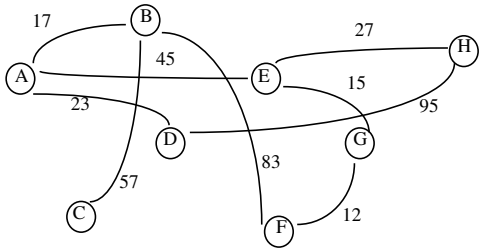
   La figura muestra un ejemplo concreto cuando N=3,



   Si se toma como operación elemental el traslado de un disco de una columna a otra, en el ejemplo son necesarios 7 movimientos y, en general, el coste es $2^N - 1$, $O(2^N)$. Por lo tanto, es un problema que se resuelve con un algoritmo de orden exponencial.

2. El Problema del Viajante de Comercio: Dada una lista de `N` ciudades y otra lista de distancias entre ellas, establecer cuál será el recorrido de longitud mínima para recorrer todas las ciudades, pasando una sola vez por cada ciudad y volviendo a la ciudad de partida.

   Normalmente, el problema se representa con un grafo como el de la figura, en el que los nodos representan las ciudades y los arcos la existencia de un camino entre ciudades y su longitud:



   Para resolver el problema, hay que evaluar todos los posibles caminos y sumar la longitud de cada recorrido para determinar cuál es el mínimo. Tomando como operación elemental la determinación de la longitud de un camino, cuando el grafo está totalmente conectado (hay un camino entre cualquier par de ciudades) el coste de resolver este problema es $O(N!)$, aunque existen algoritmos que lo reducen a $O(2^N)$.

10.5. Algoritmos de Ordenación.

"Un día de estos, me organizo el despacho...".

Gloria Martínez. Octubre 1991.

Para acabar el tema, se propone el estudio y comparación de cuatro algoritmos que resuelven el mismo problema. El problema propuesto es la ordenación *creciente* de un vector de tipo base entero. En lo que sigue, se asumirá que se ha realizado la definición de tipo siguiente:

```
typedef int vector[N];
```

Los tres primeros algoritmos pertenecen a los llamados algoritmos *cuadráticos*, es decir, su coste temporal es proporcional a N^2. Son algoritmos simples, y cuyo uso tiene un tiempo aceptable para vectores de un tamaño inferior a 500 elementos.

El cuarto algoritmo es un clásico y supuso una revolución cuando fue propuesto por Hoare en 1961: se trata del algoritmo *quicksort*, que fue el primer algoritmo de ordenación cuyo tiempo medio de ejecución consiguió rebajar su coste del orden cuadrático al orden $N \log N$.

Para comparar el comportamiento de los algoritmos, se seleccionan las operaciones que son comunes a los cuatro y que resultan imprescindibles para realizar una ordenación: la *comparación*, puesto que sin comparar no se podría establecer las posiciones relativas de dos elementos de un vector, y el *intercambio*, imprescindible para colocar un determinado elemento del vector en su sitio.

10.5.1. Ordenación por Selección (Selection Sort).

La idea básica de este algoritmo es muy simple: en un vector ordenado el primer elemento debe ser el de valor mínimo. Por lo tanto, se busca el mínimo del vector y se coloca en la primera posición. A continuación, se busca el mínimo de entre los demás elementos para colocarlo en la segunda posición, y se procede de forma similar para ir colocando el tercero, el cuarto...

En general, su comportamiento se puede describir de la siguiente forma: *"Después de haber colocado los i-1 elementos más pequeños en las posiciones 0, 1, 2, ..., i-1, en la etapa i se debe buscar el elemento de valor mínimo de los que quedan entre las posiciones i y N; cuando se encuentra se intercambia con el que ocupa su lugar, v[i]."*

El algoritmo corresponde al siguiente código:

selectionSort()

```
void selectionSort(vector v, int N){
     int i,j,min;
     int aux;

     for(i=0;i<N-1;i++){
     /* Etapa i: se busca el mínimo entre i y N */
     /* El objetivo es determinar su posición */

         min=i;
         for(j=i+1;j<N;j++){
             if(v[j]<v[min])
                 min=j;
         }
         /* Una vez conocida la posición del mínimo */
         /* se intercambia con v[i] para dejarlo en su sitio */
         aux=v[i];
         v[i]=v[min];
         v[min]=aux;
     }
}
```

A continuación se presenta una pequeña traza del funcionamiento del algoritmo. La traza pone de relieve uno de los principales defectos del método: es lo que denomina un algoritmo de "fuerza bruta", no es sensible a ningún tipo de orden previo en el vector. Esto se pone de relieve al observar en la traza como el algoritmo "obliga" a que se intercambien consigo mismos elementos que ya estaban colocados: por ejemplo, obsérvese lo que ocurre con el elemento v[2] en la tercera iteración, el v[4] en la quinta o el v[6] en la séptima.

	v[0]	v[1]	v[2]	v[3]	v[4]	v[5]	v[6]	v[7]	v[8]
Valores Iniciales:	7	15	3	1	4	23	1	42	9
i=0 → j=1..8, min=3:	7	15	3	**1**	4	23	1	42	9
intercambio:	*1*	15	3	*7*	4	23	1	42	9
i=1 → j=2..8, min=6:	1	15	3	7	4	23	**1**	42	9
intercambio:	1	*1*	3	7	4	23	*15*	42	9
i=2 → j=3..8, min=2:	1	1	**3**	7	4	23	15	42	9
intercambio:	1	1	*3*	7	4	23	15	42	9
i=3 → j=4..8, min=4:	1	1	3	7	**4**	23	15	42	9
intercambio:	1	1	3	*4*	*7*	23	15	42	9
i=4 → j=5..8, min=4:	1	1	3	4	**7**	23	15	42	9
intercambio:	1	1	3	4	*7*	23	15	42	9
i=5 → j=6..8, min=8:	1	1	3	4	7	23	15	42	**9**
intercambio:	1	1	3	4	7	*9*	15	42	*23*
i=6 → j=7..8, min=6:	1	1	3	4	7	9	**15**	42	23
intercambio:	1	1	3	4	7	9	*15*	42	23
i=7 → j=8..8, min=8:	1	1	3	4	7	9	15	42	**23**
intercambio:	1	1	3	4	7	9	15	*23*	*42*
Valores finales:	1	1	3	4	7	9	15	23	42

Pero presenta una propiedad que se pone de relieve al analizar cuántas operaciones realiza:

- Número de comparaciones: La presencia de condicionales no se traduce en un mejor y peor caso, ya que la comparación presente en el condicional `if(v[j]<v[min])`, siempre se realiza. Puesto que se ejecuta en el interior de dos bucles, el número total de comparaciones es:

$$\sum_{i=0}^{N-2}\sum_{j=i+1}^{N-1} 1 = \sum_{i=0}^{N-2}(N-1-(i+1)+1) = \sum_{i=0}^{N-2}(N-i-1) =$$

$$\sum_{i=0}^{N-2} N - \sum_{i=0}^{N-2} i - \sum_{i=0}^{N-2} 1 = N(N-1) - \frac{(N-2)(N-1)}{2} - (N-1) =$$

$$N^2 - N - \frac{N^2}{2} + \frac{3N}{2} - 1 - N + 1 = \frac{N^2}{2} - \frac{N}{2}.$$

  Por lo tanto, el número de comparaciones es de orden $O(\frac{N^2}{2})$, *cuadrático*.

- Número de intercambios: Los intercambios se realizan dentro del bucle gobernado por el índice `i`, de ahí que el total responda a la expresión,

$$\sum_{i=0}^{N-2} 1 = N - 2 + 1 = N - 1.$$

  Por lo tanto, el número de intercambios es de orden $O(N)$, *lineal*, independientemente del orden inicial existente en el vector. Esta es una propiedad interesante de este método y lo hace muy adecuado cuando el tipo base del vector es un tipo no básico (una tupla, otro vector) y los intercambios pueden suponer bastante trabajo.

10.5.2. Método de la Burbuja (Bubble Sort).

En este método también se pretende que, en cada iteración, un elemento acabe ocupando su posición correcta dentro del vector. Pero, además, se pretende que cada iteración contribuya a introducir más orden entre los elementos aún no colocados. Debe su nombre a la forma en que lo consigue: cada iteración consiste en intercambiar dos elementos consecutivos cuando estén descolocados entre sí. Si, como es el caso, se pretende realizar una ordenación creciente, entonces un elemento v[j] debe ser menor que el siguiente, v[j+1]; si no fuera así, se intercambian. Con eso se asegura que el máximo parcial en cada iteración, la *burbuja*, acabe en su sitio.

Su descripción general, por lo tanto, es: *"Después de haber colocado los elementos mayores en las posiciones N-1, N-2, ..., i+1, en la etapa i se debe colocar el mayor de los elementos restantes en la posición i del vector, intercambiando, además, todos los pares de elementos desordenados entre sí."*

bubbleSort()

```
void bubbleSort(vector v, int N){
    int i,j;
    int aux;

    for(i=N-1;i>0;i--){
    /* Etapa i: el mayor entre  v[0] y v[i], acaba en v[i] */

       for(j=0;j<i;j++){
       /* Se recorre ese trozo de vector, intercambiando */
       /* pares de elementos no ordenados entre sí */

           if (v[j]>v[j+1]){
                aux=v[j];
                v[j]=v[j+1];
                v[j+1]=aux;
           }
       }
    }
}
```

También de este algoritmo se presenta una traza. En este caso, la traza permite observar que, si bien el algoritmo de la burbuja es más sensible al orden del vector que el de selección, su implementación a base de bucles `for` se convierte en un defecto, puesto que aunque se consiga el orden antes de que finalice la ejecución completa (en el ejemplo, en la iteración del bucle `i=4`), la ejecución sigue. De ahí, que sus resultados prácticos no sean muy buenos.

	v[0]	v[1]	v[2]	v[3]	v[4]	v[5]	v[6]	v[7]	v[8]
Valores Iniciales:	7	15	3	1	4	23	1	42	9
i=8 → j=0, 1:	7	**15**	**3**	1	4	23	1	42	9
intercambio:	7	*3*	*15*	1	4	23	1	42	9
i=8 → j=2:	7	3	**15**	**1**	4	23	1	42	9
intercambio:	7	3	*1*	*15*	4	23	1	42	9
i=8 → j=3:	7	3	1	**15**	**4**	23	1	42	9
intercambio:	7	3	1	*4*	**15**	23	1	42	9
i=8 → j=4,5:	7	3	1	4	15	**23**	**1**	42	9
intercambio:	7	3	1	4	15	*1*	*23*	42	9
i=8 → j=6,7:	7	3	1	4	15	1	23	**42**	**9**
intercambio:	7	3	1	4	15	1	23	*9*	*42*
i=7 → j=0:	**7**	**3**	1	4	15	1	23	9	42
intercambio:	*3*	*7*	1	4	15	1	23	9	42
i=7 → j=1:	3	**7**	**1**	4	15	1	23	9	42
intercambio:	3	*1*	*7*	4	15	1	23	9	42
i=7 → j=2:	3	1	**7**	**4**	15	1	23	9	42
intercambio:	3	1	*4*	*7*	15	1	23	9	42
i=7 → j=3,4:	3	1	4	7	**15**	**1**	23	9	42
intercambio:	3	1	4	7	*1*	*15*	23	9	42
i=7 → j=5,6:	3	1	4	7	1	15	**23**	**9**	42
intercambio:	3	1	4	7	1	15	*9*	*23*	42
i=6 → j=0:	**3**	**1**	4	7	1	15	9	23	42
intercambio:	*1*	*3*	4	7	1	15	9	23	42
i=6 → j=1,2,3:	1	3	4	**7**	**1**	15	9	23	42
intercambio:	1	3	4	*1*	*7*	15	9	23	42
i=6 → j=4,5:	1	3	4	1	7	**15**	**9**	23	42
intercambio:	1	3	4	1	7	*9*	*15*	23	42
i=5 → j=0,1,2:	1	3	**4**	**1**	7	9	15	23	42
intercambio:	1	3	*1*	*4*	7	9	15	23	42
i=5 → j=3,4:	1	3	1	4	7	9	15	23	42
i=4 → j=0,1:	1	**3**	**1**	4	7	9	15	23	42
intercambio:	1	*1*	*3*	4	7	9	15	23	42
i=4 → j=2,3:	1	1	3	4	7	9	15	23	42
i=3 → j=0,1,2:	1	1	3	4	7	9	15	23	42
i=2 → j=0,1:	1	1	3	4	7	9	15	23	42
i=1 → j=0:	1	1	3	4	7	9	15	23	42
Valores finales:	1	1	3	4	7	9	15	23	42

En el análisis de operaciones realizadas, se obtiene el siguiente resultado:

- Número de comparaciones: La comparación del condicional `if (v[j]>v[j+1])`, se realiza siempre, independientemente de que el vector esté ordenado o no; puesto que aparece en el interior de dos bucles, el número total de comparaciones es:

$$\sum_{i=1}^{N-1}\sum_{j=0}^{i-1} 1 = \sum_{i=1}^{N-1}(i-1+1) = \sum_{i=1}^{N-1} i = \frac{(N-1)N}{2} = \frac{N^2}{2} - \frac{N}{2}.$$

  Por lo tanto, el número de comparaciones es también de orden $O(\frac{N^2}{2})$, *cuadrático*.

- Número de intercambios: En el caso de los intercambio sí que existe un mejor caso (si el vector está ordenado, no se realiza ninguno) y un peor caso (que se realicen todos, cada vez que se realiza la comparación del `if`, lo que ocurrirá cuando el vector esté ordenado por el criterio inverso al deseado). Es decir, el número de intercambios variará entre 0 y $\frac{N^2}{2} - \frac{N}{2}$. En el peor caso, el orden de intercambios es *cuadrático*.

10.5.3. Ordenación por Inserción (Insertion Sort).

Este método consigue ordenar el vector de forma muy rápida, ya que es el que más partido saca del orden existente previamente en el vector. Ello se debe a que el método no establece como objetivo básico en cada iteración que un determinado elemento acabe situado en su posición final, sino que acabe ordenado con respecto a los elementos previamente tratados. La forma en que se consigue este objetivo preserva cualquier tipo de orden ya presente en el vector, ya que al insertar un elemento, los que deban desplazarse lo hacen "en bloque", moviéndose todos una posición.

La descripción general del método es: *"Después de haber colocado los elementos mayores en las posiciones N-1, N-2, ..., i+1, en la etapa i todos los elementos colocados en posiciones posteriores a la i cuyo valor sea inferior al de v[i], se deben desplazar a la posición anterior creando así el hueco donde colocar el valor de v[i]."*

Esta idea da lugar, en una primera aproximación, al siguiente código en el que se suponen realizadas las declaraciones pertinentes:

```
for(i=N-2;i>-1;i--){
   aux=v[i];
   j=i;
   while(v[j+1]<aux){
      v[j]=v[j+1];
      j++;
   }
   v[j]=aux;
}
```

Según esto, si se parte de los valores iniciales:

	v[0]	v[1]	v[2]	v[3]	v[4]	v[5]	v[6]	v[7]	v[8]
Valores Iniciales:	7	15	3	1	4	23	1	42	9

y se analiza lo que ocurre con `i=7`, por ejemplo, se puede detectar una inconsistencia lógica:

```
i=7
    aux=v[7]   (aux=42)
    j=7
    v[8]<aux?  (9<42?) SI
          v[7]=v[8]  (v[7]=9)
          j=8
    v[9]<aux?  INCONSISTENCIA!!! (Quién es v[9]?)
```

Como se ve en el ejemplo, cuando un elemento es mayor que todos los que ocupan posiciones más altas que la suya, se hace referencia a un elemento NO definido en el vector. Para remediar esto se puede utilizar un *centinela*: se crea la posición `v[N]` y, además, se asegura que en caso de llegar a tener que realizar la comparación con su valor (cuando aparezca un máximo parcial) su evaluación asegure el fin del bucle. Para ello, basta con asignar a este elemento el mismo valor que se le asigna a `aux`, es decir, el de `v[i]`. Con esta consideración, el código del algoritmo de inserción es:

insertionSort()

```
void insertionSort(int v[], int N){
   int i,j;
   int aux;

   for(i=N-2;i>-1;i--){
   /* Etapa i: se inserta v[i] en su posición relativa */
   /* respecto a los valores ya ordenados */

      v[N]=v[i];
      /* se copia su valor en el centinela */

      aux=v[i];
      j=i;
      /* valores iniciales del bucle de inserción */

      while(v[j+1]<aux){/* Parada:un valor mayor o centinela */
         v[j]=v[j+1];
         /* cada elemento menor se mueve una posición */
         j++;
      }
      v[j]=aux;
      /* se inserta v[i] en su posición,dada por el valor de j */
   }
}
```

Por supuesto, el uso de un centinela obliga a redefinir el vector, de forma que sea de tamaño `N+1` y no de tamaño `N`, aun cuando sólo las posiciones de la `0` a la `N-1` contengan información significativa.

La siguiente tabla presenta un ejemplo de traza de esta algoritmo. Para cada valor del índice `i` se indica qué valores iría tomando el índice `j` (las filas en las que aparece `(NO)` denotan que no llega a realizarse ninguna iteración del bucle `while`, puesto que la condición es falsa inicialmente). En las filas posteriores, se indica qué valores se desplazarían como consecuencia de la ejecución del `while` y en la última fila asociada a un determinado valor de `i`, se indica la posición en la que se inserta el valor inicial de `v[i]`.

	v[0]	v[1]	v[2]	v[3]	v[4]	v[5]	v[6]	v[7]	v[8]	v[9]
Valores Iniciales:	7	15	3	1	4	23	1	42	9	_
i=7 → j=7,8:	7	15	3	1	4	23	1	**42**	**9**	42
desplazamiento:	7	15	3	1	4	23	1	*9*	*9*	42
inserción:	7	15	3	1	4	23	1	*9*	*42*	42
i=6 → j=6 (NO):	7	15	3	1	4	23	1	9	42	1
i=5 → j=5,6:	7	15	3	1	4	**23**	**1**	**9**	42	23
desplamiento:	7	15	3	1	4	*1*	*9*	*9*	42	23
inserción:	7	15	3	1	4	*1*	*9*	*23*	42	23
i=4 → j=4:	7	15	3	1	**4**	**1**	9	23	42	4
desplamiento:	7	15	3	1	*1*	*1*	9	23	42	4
inserción:	7	15	3	1	*1*	*4*	9	23	42	4
i=3 → j=3 (NO):	7	15	3	1	1	4	9	23	42	1
i=2 → j=2,3:	7	15	**3**	**1**	**1**	4	9	23	42	3
desplazamiento:	7	15	*1*	*1*	*1*	4	9	23	42	3
inserción:	7	15	*1*	*1*	*3*	4	9	23	42	3
i=1 → j=1,2,3,4,5,6:	7	**15**	**1**	**1**	**3**	**4**	**9**	23	42	15
desplazamiento:	7	*1*	*1*	*3*	*4*	*9*	*9*	23	42	15
inserción:	7	*1*	*1*	*3*	*4*	*9*	*15*	23	42	15
i=0 → j=0,1,2,3,4:	**7**	**1**	**1**	**3**	**4**	9	15	23	42	7
desplazamiento:	*1*	*1*	*3*	*4*	*4*	9	15	23	42	7
inserción:	*1*	*1*	*3*	*4*	*7*	9	15	23	42	7
Valores finales:	1	1	3	4	7	9	15	23	42	_

Del análisis del número de operaciones realizadas, se desprende el siguiente resultado:

- Número de comparaciones: El número de comparaciones realizadas está asociado al número de iteraciones que se realizan del bucle `while(v[j+1]<aux)`. La ejecución de este bucle tiene un mejor caso, que el vector esté inicialmente ordenado, ya que entonces sólo se realiza una comparación (la inicial del bucle, que siempre será falsa ya que seguro que no hay elementos fuera de su sitio) dentro del bucle `for` gobernado por el índice `i`. Se obtiene, entonces, en el mejor caso:

$$\sum_{i=0}^{N-2} 1 = N - 1.$$

  Sin embargo, cuando el vector está ordenado por el criterio contrario, en este caso de mayor a menor, seguro que cada elemento considerado es un máximo parcial en el vector y el bucle `while` debe ejecutarse siempre hasta realizar la comparación con el centinela. En cuanto al número de ejecuciones realizadas este bucle sería equivalente a un `for` cuyo índice variara entre los valores `i` (inicial) y `N` (final, con este valor se realiza la última comparación, la del centinela). Por lo tanto, en el peor caso, el número total de comparaciones es:

$$\sum_{i=0}^{N-2}\sum_{j=i}^{N} 1 = \sum_{i=0}^{N-2}(N-i+1) = \sum_{i=0}^{N-2} N - \sum_{i=0}^{N-2} i + \sum_{i=0}^{N-2} 1 =$$

$$N(N-1) - \frac{(N-1)(N-2)}{2} + (N-1) = N^2 - \frac{N^2}{2} + \frac{3N}{2} - 2 = \frac{N^2}{2} + \frac{3N}{2} - 2.$$

  Por lo tanto, el número de comparaciones varía entre $O(N)$, *lineal*, en el mejor caso y entre $O(\frac{N^2}{2})$, *cuadrático*, en el peor caso.

- Número de intercambios: A la hora de contar los intercambios, se debe tener en cuenta que en este algoritmo no se realizan intercambios "clásicos", sino que se produce un desplazamiento de elementos. Para simplificar el análisis, se van a identificar desplazamientos con intercambios, si bien no se debe olvidar que en realidad se está realizando menos trabajo (un desplazamiento supone una asignación, mientras que el intercambio supone tres asignaciones).

  El mejor caso para esta operación es, de nuevo, el caso en el que el vector ya está ordenado: el bucle `while` no llega a ejecutarse. Se asume que se hace un intercambio, al asimilar como tal las asignaciones

```
aux=v[i];
....
v[j]=aux;
```

  Por lo tanto, se obtiene

$$\sum_{i=0}^{N-2} 1 = N - 1.$$

  Como peor caso, también se ha de considerar aquel en el que el vector esté ordenado por criterio inverso. Según lo que ya se ha dicho al estimar el número de comparaciones, en este caso el bucle `while` se comporta como un `for` gobernado por el índice `j`, desde el valor inicial `1` hasta alcanzar el valor `N-1` [5]. Si se asume que cada uno de los desplazamientos realizados es un intercambio y que fuera del bucle se producen las asignaciones que ya se estimaron en el mejor caso, la expresión a resolver para obtener la función de coste es la siguiente:

$$\sum_{i=0}^{N-2}(1+\sum_{j=i}^{N-1}1) = \sum_{i=0}^{N-2}(1+(N-i)) = \sum_{i=0}^{N-2}N - \sum_{i=0}^{N-2}i + \sum_{i=0}^{N-2}1 =$$

$$N^2 - \frac{N^2}{2} + \frac{3N}{2} - 2 = \frac{N^2}{2} + \frac{3N}{2} - 2.$$

  Es decir, también el número de "intercambios", varía entre $O(N)$, *lineal*, en el mejor caso y entre $O(\frac{N^2}{2})$, *cuadrático*, en el peor caso.

10.5.4. Ordenación Rápida (QuickSort).

Este método fue propuesto por C.A.R. Hoare en 1961, y, salvo en casos especiales, es un algoritmo de ordenación muy rápido (de ahí el nombre).

La idea básica del método consiste en dividir el vector en dos partes y, entonces, ordenar cada parte de forma independiente aplicando otra vez el mismo método. Es, por lo tanto, un método recursivo (más concretamente, es un método definido recursivamente y utilizando la técnica "divide y vencerás"). Para ordenar un vector `v`, desde el elemento que ocupa la posición `s` hasta el elemento que ocupa la posición `r`, se divide el vector en dos partes sobre las que se vuelve aplicar el método. En el momento en que se puede resolver directamente el problema de la ordenación se van combinando los resultados obtenidos hasta obtener el vector ordenado. La clave en la eficiencia de este método de ordenación consiste en localizar la posición óptima a partir de la cual dividir el vector.

[5] Cuidado: a la hora de contar comparaciones hay que incluir también la que provoca la salida del `while`, pero al contar las operaciones que se realizan dentro del bucle sólo hay que contar hasta el último valor que ha provocado una ejecución del cuerpo del bucle.

Esa posición, `i`, tal que $s \leq i \leq r$, debe cumplir con las siguientes condiciones:

1. El elemento `v[i]` está, inicialmente, en la posición final del vector.
2. Todos los elementos `v[s]`, ... , `v[i-1]` son menores o iguales al valor `v[i]`.
3. Todos los elementos `v[i+1]`, ..., `v[r]` son mayores o iguales al valor `v[i]`.

Para seleccionar esa posición, se sigue la siguiente estrategia: según indica la primera de las condiciones anteriores, se toma como referencia el último valor del vector; por lo tanto, para ordenar un vector de la posición `s` a la `r`, se recorre el vector con un índice, `i`, que se va incrementando de forma que se recorren las elementos `v[s]`, `v[s+1]`, `v[s+2]`, ... parando en el momento en el que se encuentra un elemento con valor mayor o igual que el de `v[r]` (se le denominará `v[k]`); también se recorre el vector con un índice, `j`, que se va decrementando de forma que se recorren las posiciones `v[r-1]`, `v[r-2]`, ... parando en el momento en el que se encuentra un elemento con valor menor o igual que el de `v[r]` (se le denominará `v[t]`). Estos dos elementos están desordenados con respecto a `v[r]`; y para que se cumplan las condiciones 2) y 3) hay que intercambiarlos. Tras el intercambio, se repite el proceso de recorrer el vector de forma ascendente y después descendente haciendo todos los intercambios necesarios.

Si el índice `i` se incrementa y el `j` se decrementa, llegará un momento en el que se crucen. Cuando esto se cumpla, el valor `v[r]` se deja en la posición `i` y el valor de `v[i]` en la posición `r`. Con esto, seguro que se cumplen las condiciones 2) y 3) para todos los elementos con respecto a `v[i]`. Por lo tanto, `i` es el valor correcto para realizar la partición y para ordenar los elementos del vector hay que ordenar de forma independiente los subvectores de la posición `s` a la posición `i-1` y de la posición `i+1` a la posición `r`. Por ejemplo, la llamada al algoritmo *quickSort(v,0,8)* con el vector que se ha utilizado en los ejemplos previos supondría:

	v[0]	v[1]	v[2]	v[3]	v[4]	v[5]	v[6]	v[7]	v[8]
Valores Iniciales:	7	15	3	1	4	23	1	42	9
s=0, r= 8 → i=0:	**7**	15	3	1	4	23	1	42	**9**
$7 < 9 \rightarrow$ i=1:	7	**15**	3	1	4	23	1	42	**9**
$15 > 9 \Rightarrow$ k=i=1; comienza a decrementarse j									
s=0, r= 8 → j=7:	7	15	3	1	4	23	1	**42**	**9**
$42 > 9 \rightarrow$ j=6:	7	15	3	1	4	23	**1**	42	**9**
$1 < 9 \rightarrow$ t=j=6 y k < t: intercambio, tras el que se vuelve a incrementar i									
intercambio v[i] con v[j]:	7	*1*	3	1	4	23	*15*	42	**9**
i=2 :	7	1	**3**	1	4	23	15	42	**9**
$3 < 9 \rightarrow$ i=3:	7	1	3	**1**	4	23	15	42	**9**
$1 < 9 \rightarrow$ i=4:	7	1	3	1	**4**	23	15	42	**9**
$4 < 9 \rightarrow$ i=5:	7	1	3	1	4	**23**	15	42	**9**
$23 > 9 \Rightarrow$ k=i=5; comienza a decrementarse j									
j=5:	7	1	3	1	4	**23**	15	42	**9**
$23 > 9 \rightarrow$ j=4:	7	1	3	1	**4**	23	15	42	**9**
$4 < 9 \rightarrow$ t=j=4, pero k > t: se cruzan los índices, se intercambian v[i] y v[r]									
intercambio v[i] con v[r]:	7	1	3	1	4	*9*	15	42	*23*
Partición obtenida:	**7**	**1**	**3**	**1**	**4**	9	**15**	**42**	**23**

Esta llamada inicial, provocará a su vez dos llamadas recursivas: *quickSort(v, 0, 4)* y *quickSort(v, 6, 8)*, que se resolverían de modo similar y que provocarían, a su vez, nuevas llamadas recursivas, llamadas que dejarán de producirse cuando se alcance el caso trivial, que es la ordenación de un vector de tamaño 1. El algoritmo sería el siguiente:

quickSort()

```
void quickSort(vector v, int s, int r){
   int i,j;
   int aux, piv;

   if(r>s){ /* si el tamaño del vector es mayor que 1 */
      piv=v[r];
      i=s-1;
      j=r;

      /* se incr. i hasta encontrar un elemento mayor que piv */
      do{
         i++;
      } while(v[i]<piv);

      /* se decr. j hasta encontrar un elemento menor que piv */
      /* o que se crucen los índices */
      do{
         j--;
      }while(v[j]>piv && j>(i-1));

      while(i<j){ /* mientras no se crucen los índices */
      /* se intercambian v[i] y v[j] */
         aux=v[j];
         v[j]=v[i];
         v[i]=aux;

      /* se sigue incrementando i */
         do{
            i++;
         } while(v[i]<piv);

      /* se sigue decrementando j */
         do{
            j--;
         } while(v[j]>piv);
      }
      /* ya se ha encontrado el valor óptimo para la partición */
      aux=v[r];
      v[r]=v[i];
      v[i]=aux;

      /* llamadas recursivas para ordenar los */
      /* dos subvectores resultantes */
      quickSort(v,s,i-1);
      quickSort(v,i+1,r);
   }
}
```

La tabla siguiente muestra una traza completa sobre el vector ejemplo.

	v[0]	v[1]	v[2]	v[3]	v[4]	v[5]	v[6]	v[7]	v[8]
Valores Iniciales:	7	15	3	1	4	23	1	42	9
quickSort(v,0,8) → i=0:	**7**	15	3	1	4	23	1	42	**9**
$7 < 9 \rightarrow$ i=1:	7	**15**	3	1	4	23	1	42	**9**
$15 > 9 \rightarrow$ j=7:	7	15	3	1	4	23	1	**42**	**9**
$42 > 9 \rightarrow$ j=6:	7	15	3	1	4	23	**1**	42	**9**
intercambio v[i] con v[j]:	7	*1*	3	1	4	23	*15*	42	**9**
i=2 :	7	1	**3**	1	4	23	15	42	**9**
$3 < 9 \rightarrow$ i=3:	7	1	3	**1**	4	23	15	42	**9**
$1 < 9 \rightarrow$ i=4:	7	1	3	1	**4**	23	15	42	**9**
$4 < 9 \rightarrow$ i=5:	7	1	3	1	4	**23**	15	42	**9**
$23 > 9 \rightarrow$ j=5:	7	1	3	1	4	**23**	15	42	**9**
$23 > 9 \rightarrow$ j=4:	7	1	3	1	**4**	23	15	42	**9**
intercambio v[i] con v[r]:	7	1	3	1	4	*9*	15	42	*23*
Partición obtenida:	**7**	**1**	**3**	**1**	**4**	9	**15**	**42**	**23**
quickSort(v,0,4) → i=0:	**7**	1	3	1	**4**	_	_	_	_
$7 > 4 \rightarrow$ j=3:	7	1	3	**1**	**4**	_	_	_	_
intercambio v[i] con v[j]:	*1*	1	3	*7*	**4**	_	_	_	_
i= 1:	1	**1**	3	7	**4**	_	_	_	_
$1 < 4 \rightarrow$ i=2:	1	1	**3**	7	**4**	_	_	_	_
$3 < 4 \rightarrow$ i=3:	1	1	3	**7**	**4**	_	_	_	_
$7 > 4 \rightarrow$ j=2:	1	1	**3**	7	**4**	_	_	_	_
intercambio v[i] con v[r]:	1	1	3	*4*	*7*	_	_	_	_
Partición obtenida:	**1**	**1**	**3**	4	**7**	_	_	_	_
quickSort(v,0,2) → i=0:	**1**	1	**3**	_	_	_	_	_	_
$1 < 3 \rightarrow$ i=1:	1	**1**	**3**	_	_	_	_	_	_
$1 < 3 \rightarrow$ i=2:	1	1	**3**	_	_	_	_	_	_
$3 = 3 \rightarrow$ j=1:	1	1	**3**	_	_	_	_	_	_
intercambio v[i] con v[r]:	1	1	*3*	_	_	_	_	_	_
Partición obtenida:	**1**	**1**	3	_	_	_	_	_	_
quickSort(v,0,1) → i=0:	1	**1**	_	_	_	_	_	_	_
$1 = 1 \rightarrow$ j=0:	1	**1**	_	_	_	_	_	_	_
intercambio v[i] con v[r]:	*1*	*1*	_	_	_	_	_	_	_
Partición obtenida:	**1**	_	_	_	_	_	_	_	_
quickSort(v,0,0) :	1	_	_	_	_	_	_	_	_
quickSort(v,2,1) :	_	_	_	_	_	_	_	_	_
fin **quickSort(v,0,1)**:	1	1	_	_	_	_	_	_	_
quickSort(v,3,2) :	_	_	_	_	_	_	_	_	_
fin **quickSort(v,0,2)**:	1	1	3	_	_	_	_	_	_
quickSort(v,4,4) :	_	_	_	_	7	_	_	_	_
fin **quickSort(v,0,4)**:	1	1	3	4	7	_	_	_	_
quickSort(v,6,8) → i=6:	_	_	_	_	_	_	**15**	42	**23**
$15 < 23 \rightarrow$ i=7:	_	_	_	_	_	_	15	**42**	**23**
$42 > 23 \rightarrow$ j=7:	_	_	_	_	_	_	15	**42**	**23**
$42 > 23 \rightarrow$ j=6:	_	_	_	_	_	_	15	42	**23**
intercambio v[i] con v[r]:	_	_	_	_	_	_	15	*23*	*42*
Partición obtenida:	_	_	_	_	_	_	**15**	23	**42**
quickSort(v,6,6):	_	_	_	_	_	_	15	_	_
quickSort(v,8,8):	_	_	_	_	_	_	_	_	42
fin **quickSort(v,6,8)**:	_	_	_	_	_	_	15	23	42
fin **quickSort(v,0,8)**:	1	1	3	4	7	9	15	23	42

El análisis del coste del algoritmo QuickSort es complejo. Por ello, sólo se expondrán los resultados de este análisis, para poder compararlos con los otros tres métodos también presentados. La siguiente tabla resume los resultados obtenidos para Selección, Burbuja e Inserción; además, se presentan los resultados para el caso medio, que se pueden consultar en el libro "Algorithms in C" de R. Sedgwick.

	Comparaciones			Intercambios		
Método	Mejor Caso	Peor Caso	Caso Medio	Mejor Caso	Peor Caso	Caso Medio
Selección	$O(\frac{N^2}{2})$	$O(\frac{N^2}{2})$	$O(\frac{N^2}{2})$	$O(N)$	$O(N)$	$O(N)$
Burbuja	$O(\frac{N^2}{2})$	$O(\frac{N^2}{2})$	$O(\frac{N^2}{2})$	0	$O(\frac{N^2}{2})$	$O(\frac{N^2}{4})$
Inserción	$O(N)$	$O(\frac{N^2}{2})$	$O(\frac{N^2}{4})$	$O(N)$	$O(\frac{N^2}{2})$	$O(\frac{N^2}{4})$

La tabla pone de relieve por qué estos métodos son conocidos como *cuadráticos*: ese es su coste medio, especialmente en el caso de las comparaciones, que es la operación de la cual depende en mayor medida. Estos algoritmos de ordenación suelen ser denominados "elementales"; ello no quiere decir que sean inadecuados, sino que son métodos básicos a partir de los cuales se han diseñado algoritmos más sofisticados, como el QuickSort, el ShellSort, el HeapSort u otros. Se debe recordar, además, que en los casos en que el número de elementos a ordenar no sea superior a un orden de 500, estos métodos pueden ser tan adecuados como los otros.

A estos otros métodos más elaborados se les suele denominar "eficientes". Esto es así porque consiguen en su comportamiento medio un orden inferior al cuadrático, $O(N \log N)$.

Tómese como ejemplo el QuickSort, del que se realizará una aproximación a su complejidad, tomando como operación básica el número de comparaciones (tanto, en la referencia anterior ("Algorithms", R. Sedgwick), como en el libro "Fundamentos de Algoritmia" de G. Brassard y P. Bratley, se puede encontrar el estudio detallado).

La eficiencia de este método depende de lo equilibrada que sea la partición; es decir, es más rápido cuanto más similares sean en tamaño los dos trozos en que se divide el vector para ordenarlos de forma independiente. Así, para este algoritmo cambia el mejor y el peor caso: el mejor caso es cuando se consigue dividir siempre el vector por la mitad y el peor, cuando se divide en un vector de tamaño 1 y otro de tamaño `n-1` (lo cual, curiosamente, ocurre cuando el vector ya está ordenado).

Para estudiar el mejor caso, es decir, cuando se puede dividir el vector en dos trozos iguales, el orden del algoritmo se plantea a partir de una expresión recurrente,

$$\begin{cases} C(1) = 0, \\ C(N) = 2C(\frac{N}{2}) + N \end{cases}$$

Es decir, el coste del caso trivial se considera nulo y el coste de ordenar un vector de tamaño N es del orden del doble del coste de ordenar un vector de tamaño N/2, más el orden de recorrer el vector de tamaño N. Al resolver esta expresión, se obtiene que $C(N) \approx N \log_2 N$. Por lo tanto, es $O(N \log_2 N)$ en el mejor caso.

Se puede demostrar que en el peor caso, el orden de coste es $O(N^2)$, mientras que en el caso medio se obtiene un orden de coste $O(N \log N)$. En esta expresión, no se fija la base en la que se calcula el logaritmo, pero puesto que $\log_B N$ se puede expresar como $\log_C B \times \log_C N$, es decir, una constante por otro logaritmo, conocer el valor exacto de B no afecta a la expresión del orden de coste.

10.6. Bibliografía.

1. Capítulo 2 de “Thinking Recursively”. Eric S. Roberts. John Wiley & Sons. 1986.
2. “Algorithms in C++, parts 1–4”. Robert Sedgewick. Ed. Addison-Wesley. 1998.

   Tengo debilidad por este libro, lo confieso. Más que pensando en este tema os lo dejo como referencia para cualquier duda que tengáis sobre “ ¿... existirá un algoritmo para resolver este problema?”. Si existe, estará en este libro.
3. “Fundamentos de Algoritmia”. Brassard & Bratley. Ed. Prentice Hall. 1997.

   Es un libro de nivel avanzado; os lo dejo como referencia futura, si alguna vez tenéis inquietudes sobre temas de complejidad computacional.
4. Si lo de leer los libros os sobrepasa, pasad directamente a esta página web (con el Java puesto... y con Explorer :-(...):

   `http://www.seas.gwu.edu/~idsv/idsv.html`

   Tiene animaciones de algoritmos. Creo recordar que pide un login; introducid cualquiera. Después id al apartado *Sorting* y disfrutad ;-)

10.7. Problemas Propuestos.

1. Calcular la función de coste de la siguiente secuencia tomando como operación elemental, primero, la comparación y, después, la asignación:

```
....
k=0;
for(i=0;i<(n-2);i++){
   for(j=i+1;j<n;j++){
      if(a[i]<b[j])
         c[k]=a[i];
      else
         c[k]=b[j];
      k=k+1;
   }
}
.....
```

2. Calcular la función de coste en multiplicaciones del siguiente fragmento de algoritmo:

```
....
/* Pre: entero n, entero k y  n, k >= 1 */
i=1;
while(i<n){
  j=i-1;
  while(j<k){
     a[i]=a[i-1]+b[j+1]*b[j+2];
     j++;
  }
  i++;
}
....
```

3. Calcular, atendiendo al número de sumas, el orden de la función de coste del algoritmo:

```
int Sumas(int n){
   int i,j,s;

   i=0;
   s=0;
   while(i<=n){
      s=s+i;
      i=i+1;
      j=i;
      while(j<(2*(i-1))){
         s=s+j;
         j=j+1;
      }
   }
   return s;
}
```

4. Dos alumnos de Informática han quedado para estudiar programación y discuten por culpa del siguiente enunciado, *"Calcular el orden de coste del siguiente algoritmo tomando como operación elemental la comparación:"*

```
int Costoso(int a, int b, int N){
   int i,j,c;

   for(i=0;i<N;i++){
      j=0;
      while(j<i){
         if(j>=i)
            j++;
         else
            j=i;
      }
   }
   c=j;
   return c;
}
```

   Uno dice que el coste en comparaciones es $O(N^2)$ en el peor caso y $O(N)$ en el mejor y el otro dice que no, que es $O(3N)$ tanto en el mejor como en el peor caso. ¿Quién tiene razón y por qué?

5. Calcular la función de coste del siguiente fragmento, en el mejor y en el peor caso:

```
/* Pre: entero n >= 1, entero m >= 2 */
for(i=0;i<n;i++){
   j=m;
   while((j>2) || (i>n)){
      /* 2 operaciones elementales */
      j--;
   }
}
```

6. Dada la siguiente secuencia de instrucciones,

```
longitud = strlen(cadena);
for (i=0; i<longitud; i=i+1)
   if (cadena[i] != ' ')
       for (j=i+1; j<longitud; j=j+1)
          if (cadena[j] == cadena[i])
             cadena[j]=' ';
```

   calcular su función de coste en el mejor y en el peor caso asumiendo que la operación básica es la comparación. Se supone que la cadena tiene N caracteres.

7. Calcular la función de coste en el mejor y en el peor de los casos en el siguiente fragmento de algoritmo, tomando como operación elemental la multiplicación:

```
for(i=1;i<n;i++){
   j=0;
   while(j<n){
      if(a[i]<b[j]){
         j=j+1;
         c[j]=a[i]*b[j];
      }
      else{
         j=j+10;
         c[j]=a[j]*b[i];
      }
   }
}
```

8. Se tiene el siguiente fragmento de código:

```
for(i=0;i<n;i++){
   if(a>0)
      x[i]=x[i]+1;
   else
      x[i]=0;
}
```

   a) ¿Cuál es su coste en comparaciones?

   b) ¿Puedes escribir un algoritmo equivalente con un coste menor?¿Cuál es su coste?

9. Dado el fragmento de algoritmo:

```
for(i=0;i<n;i++){
   if(i==0)
      aux=v[i]*w[i];
   else
      aux=aux+v[i]*w[i];
}
```

   a) ¿Cuál es su coste en comparaciones?

   b) ¿Puedes escribir un algoritmo equivalente con un coste menor?¿Cuál es su coste?

10. ¿Verdadero o Falso (justificar)?: "Si un algoritmo A tiene como función de coste $8n^2+3n+2$ y B es un algoritmo de $O(n^2)$ en el peor caso, entonces el algoritmo B es más eficiente que el algoritmo A".

11. ¿Verdadero o Falso (justificar)?: "Si un algoritmo A tiene como función de coste $8n^2+3n+2$ en el mejor caso y B es un algoritmo que tiene como función de coste $4n^2 + 17n$ en el peor caso, entonces el algoritmo B es más eficiente que el algoritmo A".

12. Dada la secuencia

```
for(i=0;i<n;i++){
   for(j=0;j<m;j++){
      if(A[i][j] > v[0])
         alg_cost_lineal(v,p,A[i][j]);
   }
}
```

y sabiendo que `alg_cost_lineal` tiene un coste de orden lineal, $O(p)$, ¿qué orden de coste tiene la secuencia anterior?

13. Dado el algoritmo `Costoso`,

```
void Costoso (int v[], int N, int *indice){
  int i;
  *indice=0;
  for (i=1; i<N; i=i+1)
    if (v[i]>v[i-1])
      *indice=i;
}
```

calcular la función de coste en el mejor y en el peor caso del siguiente bucle (en el que se supone que todas las variables han sido convenientemente definidas), en el que se realizan llamadas a `Costoso`:

```
  for (k=0; k<N; k=k+1){
    Costoso(w, k, &aux);
    w[k]=aux;
  }
```

Se considera que la operación básica es la *asignación*.

14. Calcular la función de coste en operaciones básicas del siguiente fragmento:

```
for(i=0;i<p;i++){
   j=0;
   while((m[i][j]!=0) && (j<q)){
      j++;
   }
   if(m[i][j]==0){
      Operacion_basica;
      for(k=j+1;k<q;k++){
         Operacion_basica;
      }
   }
}
```

15. Calcular la función de coste en operaciones básicas del siguiente fragmento:

```
i=0;
while((v[i]<c) && (i<n)){
   Operacion_basica;
   i++;
}
if(v[i]>=c){
   j=i;
   while(j<n)){
      Operacion_basica;
      j++;
   }
   j=0;
   while(j<=i){
      Operacion_basica;
      j++;
   }
}
```

16. Calcular el orden de coste de la siguiente secuencia, tomando la asignación como operación básica:

```
i=0;
s=0;
while(i<n){
   j=i;
   while(j<n){
      s=s+a[i][j];
      j=j+2;
   }
   j=i+1;
   while(j<n){
      s=s-a[i][j];
      j=j+2;
   }
   i=i+2;
}
```

17. Calcular la función de coste en `Operaciones_Basicas` del siguiente fragmento de código:

```
i = 0;
while ((i < N) && (v[i] != aux)) {
    /* 2 Operaciones_Basicas */
    i++;
}
j = i;
while (j < N) {
  if (v[i] == aux) {
       /* 2 Operaciones_Basicas */
  }
  j++;
}
```

18. Calcular la función de coste del siguiente algoritmo en el mejor y en el peor caso, considerando como operación elemental la comparación de 2 números reales.

```
#define N ...
#define M ...

typedef float tMatrizReal[N][M];

int contarFilasMarcadas (tMatrizReal matriz,
                         float marca) {
/* Pre: no hay */

   int i, j, cont;
   boole marcada;

   cont = 0;
   i=0;
   while (i<N) {
      marcada = falso;
      j=0;
      while (j<M && !marcada) {
         marcada = (matriz[i][j] == marca);
         j++;
      }
      if (marcada)
         cont++;
      i++;
   }
   return cont;
/* Post: devuelve el número de filas que contienen
   al menos un elemento igual al valor de marca */
}
```

19. Hay que calcular la función de coste en el mejor y en el peor caso, considerando como operación elemental la comparación entre un elemento de la tabla y el parámetro `valor` (las operaciones en negrita).

    Las constantes `NF`, `NC` y `N` (`N=NF`×`NC`) se suponen bien definidas.

```
typedef int tMiSudoku [N][N];

void asignarCasilla (tMiSudoku tabla,
          int f, int c, int valor, boole *error) {
/* Pre: tabla es una matriz de NxN enteros de 0 a N;
   0<=f<N, 0<=c<N, 1<=valor<=N */
  int i,j,ff,cc;

  tabla[f][c]=valor;
  *error=falso;
  i=0;
  while (i<N) && (!(*error)) {
    *error=(i!=f)&&(tabla[i][c]==valor);
    i++;
  }
```

```
  j=0;
  while (j<N) && (!(*error)) {
    *error=(j!=c)&&(tabla[f][j]==valor);
    j++;
  }
  ff=(f/NF)*NF;
  cc=(c/NC)*NC;
  i=ff;
  while (i<ff+NF) && (!(*error)) {
    j=cc;
    while (j<cc+NC) && (!(*error)) {
      *error=(!((i==ff)&&(j==cc)));
      *error=*error && (tabla[i][j]==valor);
      j++;
    }
    i++;
  }
}
/*Post: se asigna valor a la posicion (f,c) de la tabla;
error es cierto si hay un valor identico en la misma
fila, columna o en la submatriz de dicha posicion */
```

20. Dada la secuencia

```
paso=1;
v[0]='0';

for(i=1; i<N; i++){
   for(j=0; j<paso; j++){
      if(v[j]=='1')
         v[paso+j]='0';
      else
         v[paso+j]='1';
   }
   paso=paso*2;
}
```

calcular su coste temporal en asignaciones. El resultado debe ser función de N **exclusivamente**. Nota:

$$\sum_{i=0}^{N-1}(2^i) = 2^N - 1$$

21. Para resolver el problema de sobreescribir los elementos nulos de un vector con la media de los elementos anteriores, se proponen dos secuencias de instrucciones:

Solucion1

```
acum=v[0];
for (i=1;i<N;i=i+1) {
   if (v[i]==0)
      v[i]=acum/i;
   acum=acum+v[i];
}
```

Solucion2
```
for (i=1;i<N;i=i+1) {
   if (v[i]==0) {
       acum=0;
       for (j=0;j<i;j=j+1)
           acum=acum+v[j];
       v[i]=acum/i;
   }
}
```

En ambas se asume que todas las variables han sido convenientemente declaradas.

a) Justificar que ambas secuencias resuelven correctamente el problema.

b) Justificar **calculando las funciones de coste de ambas y comparándolas** cuál es más eficiente. Para ello se asume que la operación básica es la asignación.

22. Para resolver el problema de calcular la suma de los divisores pares de un entero `n`, se proponen dos secuencias de instrucciones:

Solucion1
```
acum=0;
for (i=1;i<=n/2;i=i+1){
   if ((n%i==0) && (i%2==0))
      acum=acum+i;
}
```

Solucion2
```
acum=0;
if (n%2==0){
   for (i=2;i<=n/2;i=i+2){
      if (n%i==0)
         acum=acum+i;
   }
}
```

En ambas se asume que todas las variables han sido convenientemente declaradas y que la suma de los divisores pares se obtiene en `acum`. Justificar brevemente que ambas secuencias calculan correctamente dicha suma.

Justificar **calculando las funciones de coste de ambas y comparándolas** cuál es más eficiente. Para ello se asume que la operación básica es la comparación.

23. Responder a las siguientes cuestiones, dados los algoritmos:

Algoritmo1
```
acum=0
for i in lista1:
   for j in lista2:
      if (i==j):
          if (lista1.index(i)<=lista2.index(j)):
              acum=acum+1
```

Algoritmo2
```
acum=0
for i in lista1:
   for j in range(lista1.index(i),len(lista2)):
```

```
      if (i==lista2[j]):
          acum=acum+1
```

a) ¿Verdadero o falso? (justificar): *"Dadas dos listas,* `lista1` *y* `lista2`, sin elementos repetidos *los dos algoritmos hacen la misma operación sobre ambas listas"*.

Tanto si es verdadero como si es falso, hay que indicar cuál es la operación que realizan ambos algoritmos.

b) Calcular la función de coste de ambos algoritmos, tomando como operación elemental la **comparación**. Se asume que en `lista1` hay N elementos y en `lista2`, M, con M, $N > 0$. Elegir el mejor algoritmo, a la vista del resultado obtenido.

24. Responder a las siguientes cuestiones, dados los algoritmos:

Algoritmo1

```
int binADecimalv1(char *binario) {
/* Pre: binario es una cadena de 0's y 1's */
   int aux=0, i=0, k=1;

   while (binario[i]!='\0') {
      aux=aux+(binario[i]-'0')*k;
      i=i+1;
      k=2*k;
   }
   return aux;
/* Post: .... */
}
```

Algoritmo2

```
int binADecimalv2(char *binario) {
/* Pre: binario es una cadena de 0's y 1's */
   int aux=0, i=0, k=1;

   while (binario[i]!='\0') {
      if (binario[i]-'0')
         aux=aux+k;
      i=i+1;
      k=2*k;
   }
   return aux;
/* Post: .... */
}
```

a) ¿Verdadero o falso? (justificar): *"Dada una cadena de caracteres que sólo contiene los caracteres* `'1' o '0'`*, ambos algoritmos devuelven el valor decimal equivalente al número binario representado en la cadena"*.

b) Calcular la función de coste de ambos algoritmos, tomando como operación elemental **todas las operaciones aritméticas**, es decir, indicando el total de sumas, restas y multiplicaciones. Elegir el mejor algoritmo, a la vista del resultado obtenido.

25. Diseñar un algoritmo para ordenar un grupo de N alumnos, para cada uno de los cuales se dispone de la siguiente información:

número de expediente(ENTERO)
nombre(CADENA)
grupo(CARACTER)

de modo que, opcionalmente, se pueda obtener una relación ordenada de la siguiente forma:

- Opción (a) por orden alfabético,
- Opción (b) por número de expediente,
- Opción (c) por grupo, y dentro de cada grupo, por orden alfabético.

26. La Secretaría de la Universidad almacena la información referente a los estudiantes de ITIS en un vector de tamaño p, y cada elemento refleja la siguiente información:

    nombre(CADENA)
    número de expediente(ENTERO)
    nota_acceso(REAL)

    El día 1 de Octubre ese vector estaba ordenado alfabéticamente, por el nombre. Entre el 1 de Octubre y el 1 de Diciembre, la información en el vector varió, ya que se produjeron "bajas" (del orden de un 6 % del total de inscritos). Puesto que había una lista de espera, las bajas permitieron la matrícula de otras personas que estaban en dicha lista de espera. Se pide:

    a) Escribir un algoritmo que permita realizar bajas de acuerdo al siguiente procedimiento: conocido el número de expediente del alumno que causa baja y toda la información relativa al primer alumno en lista de espera, localizar en el vector el elemento que contiene la información sobre el alumno que causa baja y sobreescribirla con la información del nuevo.

    b) Escribir un algoritmo que permita reordenar de nuevo el vector por orden alfabético, una vez que se hayan realizado todas las bajas. ¿Qué algoritmo de ordenación cuadrático sería el más indicado?¿Por qué?

27. Dados dos vectores A y B de tamaño N ordenados, construir un nuevo vector C de tamaño 2N de forma que se obtenga ordenado (Operación *Merge Sort*).

28. Se propone la siguiente modificación del método de ordenación por inserción:

```
void insertionV2(int v[], int N) {
   int i,j;
   int aux;

   for(i=N-2;i>-1;i--) {
      if(v[i+1]<v[i]){
         aux=v[i];
         v[N]=aux;
         j=i;
         do{
           v[j]=v[j+1];
           j++;
         } while(v[j+1]<aux);
         v[j]=aux;
      }
   }
}
```

    a) Justificar que esta nueva versión es equivalente a la vista en teoría.

b) Comparar esta nueva versión con la estudiada en teoría; para ello, hay que calcular su función de coste y determinar si es peor, igual o mejor.

29. ¿Qué ventajas o inconvenientes presentaría esta formulación alternativa del algoritmo de selección, respecto de la estudiada?

```
void selectionV2(int v[], int N){
  int i,j,min;
  int aux;

  for(i=0;i<N-1;i++){
     min=i;
     for(j=i+1;j<N;j++){
        if(v[j]<v[min])
           min=j;
     }
     if(min!=i){
        aux=v[i];
        v[i]=v[min];
        v[min]=aux;
     }
  }
}
```

30. Alguien ha pensado que el algoritmo de ordenación por selección sería más rápido si en cada iteración se buscaran simultáneamente el máximo y el mínimo de la porción de vector que falta por odenar y se colocaran cada uno en su sitio.

    Escribir dicha versión alternativa y comparar el coste del algoritmo obtenido con el del original.

31. ¿Qué hace este algoritmo?

```
boole queHace(int v[], int i, int j, int x){
  boole enc;
  int m;

  if(i>j)
     enc=falso;
  else{
     m=(i+j)/2;
     if(v[m]==x)
        enc=cierto;
     else{
        if(v[m]>x)
           enc=queHace(v,i,m-1,x);
        else
           enc=queHace(v,m+1,j,x);
     }
  }
  return enc;
}
```

32. Se define

```
typedef struct{
      /*de algun tipo adecuado*/ fecha1,fecha2;
      int maletas;
} perdidas;

typedef perdidas vector[N];
```

para resolver el siguiente problema: Siguiendo la directiva 145.672/NPI 99 de la CE, en el aeropuerto de Barajas quieren atajar el caos y van a comenzar estudiando cómo resolver el problema de la pérdida de maletas. Para ello, han almacenado la información en un vector en el que cada elemento tiene 3 campos,

fecha1 indica un día,

maletas indica el número de maletas perdidas el día fecha1,

fecha2 inicialmente no tiene valor; pero al final del proceso en este campo se debe indicar la fecha del día más reciente, anterior a fecha1, en el que la cantidad de maletas perdidas fue superior a la cantidad de maletas perdidas en fecha1. Si en ninguno de los días anteriores se hubieran perdido más maletas, se le da el valor de fecha1.

Escribir el algoritmo que permita rellenar el campo fecha2, sabiendo que el vector está ordenado cronológicamente, según el valor del campo fecha1.

33. Dados los tipos

```
typedef struct{
   char nombre[30];
   int puntos;
} ganador;

typedef ganador vector[N];
```

escribir un algoritmo que haga lo siguiente: Se sabe que `tetris` está definido como de tipo `vector` y que está ordenado por orden decreciente por el valor del campo entero `puntos`. También se sabe que no todas las posiciones están ocupadas, sólo las `p` primeras (las posiciones libres tienen el campo `puntos` con el valor `-1`). Escribir un algoritmo al que se le pase el nombre de un jugador y su correspondiente puntuación y la inserte en su lugar correspondiente, de forma que el vector siga ordenado.

34. Escribir un algoritmo que encuentre un punto de silla de una matriz A de reales de tamaño $m \times m$. Un punto de silla es un elemento que es máximo en su columna y mínimo en su fila. Se supone que en la matriz no hay elementos iguales.

    Por ejemplo,

$$\begin{pmatrix} 1{,}3 & 4{,}5 & 0{,}97 & 0{,}3 & 3{,}25 \\ 0{,}75 & 6{,}5 & 0{,}05 & 0{,}00 & 0{,}1 \\ 0{,}64 & 0{,}55 & 0{,}4 & 0{,}07 & 0{,}45 \\ 0{,}2 & 0{,}22 & 0{,}25 & 0{,}23 & 0{,}35 \\ 1{,}25 & 6{,}85 & 0{,}23 & 0{,}24 & 0{,}76 \end{pmatrix}.$$

    El elemento A[0][3], que vale 0.3, es un punto de silla: es el mínimo de la fila 0 y el máximo de la columna 3.