

Resumen de programación 3

Tema 7. Divide y vencerás.

Índice:

7.1. Introducción: multiplicación de enteros muy grande	3
7.2. El caso general	8
7.3. Búsqueda binaria	9
7.4. Ordenación	12
7.4.1. Ordenación por fusión (mergesort)	12
7.4.2. Ordenación rápida (quicksort)	16
7.5. Búsqueda de la mediana	21
7.6. Multiplicación de matrices	23
7.7. Exponenciación	24

Bibliografía:

Se han tomado apuntes de los libros:

- *Fundamentos de algoritmia*. G. Brassard y P. Bratley
- *Estructuras de Datos y Algoritmos*. R. Hernández

Previo a ver el capítulo explicaré una serie de características que tiene este resumen:

1. la primera de ellas es que la multiplicación, a diferencia de otros documentos pondremos la multiplicación o bien con $*$ o sin ningún signo intermedio. Es simplemente un apunte, sin ninguna importancia conceptual.
2. La segunda de ellas es que hemos omitido el último punto del temario, el 7.8, que trata de la encriptación, ya que lo veremos en la sección de ejercicios, por lo que omitiremos dar más detalles.
3. La tercera de ellas es que debido al evidente cansancio del autor al hacer los documentos y al intento por resumirlos se ha intentado explicar de modo adecuado el punto 7.7, el de la exponenciación, aunque no sé si habrá quedado claro.
4. Por último, decir que la parte de la búsqueda binaria, así como las distintas ordenaciones son **muy importantes**, por haber entrado en los últimos exámenes, de los años 2007-08, sin desatender las otras partes del tema.

Una vez comentado estos puntos pasamos a ver el resumen de este tema. Divide y vencerás es una técnica para diseñar algoritmos que consiste en:

- **Descomponer** el caso que haya que resolver en un cierto número de subcasos más pequeños del mismo problema.
- **Resolver** sucesiva e independientemente todos estos subcasos.
- **Combinar** después las soluciones obtenidas de esta manera para obtener la solución del caso original.

7.1. Introducción: multiplicación de enteros muy grandes

Recordaremos el **método clásico** de multiplicación visto en el tema 1, la cual era simplemente la multiplicación lo más sencilla posible. Un ejemplo podrá ser:

	a	b	c	d
	e	f	g	h
	$h * a$	$h * b$	$h * c$	$h * d$
				$g * d$
+				
+				

Como vemos, hemos hecho la multiplicación de h con el resto de los números de la primera fila. Iremos desplazando una posición a la izquierda al seguir multiplicando los demás números.

Por tanto, hacemos $n * n$ multiplicaciones y cerca de $2 * n$ sumas, considerándolas **operaciones elementales**, que es aquella cuyo tiempo de ejecución puede ser acotada superiormente por una constante que sólo dependerá de la implementación particular usada: de la máquina, del lenguaje de programación, etc. (recordatorio del resumen del tema 2).

Por ello, el coste del método clásico es $\theta(n^2)$.

Otro algoritmo para la multiplicación de enteros que discutimos en el tema 1 es que llamábamos técnica de “**divide y vencerás**”, que consistía en reducir la multiplicación de dos números de n cifras a 4 multiplicaciones de números de $n/2$ cifras. Un ejemplo de esto podrá ser la multiplicación de 0981×1234 , que veremos paso a paso:

1. Descompondremos ambas cifras en números de longitud mitad, como sigue:

$$\begin{array}{ll} w = 09 & y = 12 \\ x = 81 & z = 34 \end{array}$$

2. Una vez descompuestos los dos valores (nos fijaremos que los rellenaremos con ceros a la izquierda en caso de ser de cifras impares), tenemos lo siguiente:

$$0981 \times 1234 = (9 * 10^2 + 81) \times (12 * 10^2 + 34)$$

3. Sustituiremos la descomposición anterior las variables antes deducidas (w, y, x y z), por lo que tenemos:

$$\begin{aligned} 0981 \times 1234 &= (9 * 10^2 + 81) \times (12 * 10^2 + 34) = \\ &= (w * 10^2 + x) \times (y * 10^2 + z) \end{aligned}$$

4. Por último, desarrollaremos los paréntesis de esta manera:

$$\begin{aligned} (w * 10^2 + x) \times (y * 10^2 + z) &= \\ &= w * y * 10^4 + (w * z + x * y) * 10^2 + x * z. \end{aligned}$$

Queda por decir que los números elevados a 10 indican desplazamientos de las cifras, siendo 10^4 desplazamiento de 4 cifras (a la izquierda) y 10^2 de 2.

Tendremos, por tanto, la siguiente **ecuación de recurrencia** que resuelve este problema:

$$t(n) = 4 * t(n/2) + O(n)$$

Está ecuación de recurrencia indica que hay 4 subproblemas mitad (las correspondientes a x, y, w y z) y además, el coste de las sumas y los desplazamientos es lineal($O(n)$). Por ello, las variables (del tema 4) para resolver la recurrencia, siendo de reducción por división serán:

a: Número de llamadas recursivas = 4

b: Reducción del problema en cada llamada = 2

c * n^k: Coste de las operaciones extras a las llamadas recursivas. Será, en este caso, $c * n^k = n \Rightarrow k = 1$

Nuestra resolución de la reducción por división tiene 3 casos distintos, que son:

$$T(n) = \begin{cases} \theta(n^k) & \text{si } a < b^k \\ \theta(n^k * \log(n)) & \text{si } a = b^k \\ \theta(n^{\log_b a}) & \text{si } a > b^k \end{cases}$$

Sustituimos las variables en la igualdad $a = b^k$, teniendo $4 > 2^1$, que equivaldría al tercer caso. Por ello, el coste del algoritmo es $T(n) \in \theta(n^{\log_b a}) = \theta(n^{\log_2 4}) = \theta(n^2)$.

Para **mejorar** el algoritmo clásico debemos encontrar una forma de reducir la multiplicación original no a 4 multiplicaciones si no a 3 de números de tamaño mitad ($n/2$). Para ello, tenemos:

$$r = (w + x) * (y + z)$$

$$p = w * y$$

$$q = x * z$$

Desarrollamos r, como sigue:

$$\begin{aligned} r &= (w + x) * (y + z) = w * y + w * z + x * y + x * z = \\ &= \underbrace{w * y}_p + \underbrace{(w * z + x * y)} + \underbrace{x * z}_q \end{aligned}$$

Despejamos $(w * z + x * y)$ y queda:

$$(w * z + x * y) = r - \underbrace{w * y}_p - \underbrace{x * z}_q$$

Sustituimos las variables anteriores y tendremos lo siguiente:

$$(w * z + x * y) = r - p - q$$

Para nuestro caso particular, lo veremos mediante un ejemplo:

$$p = w * y = 09 * 12 = 108$$

$$q = x * z = 81 * 34 = 2754$$

$$r = (w + x) * (y + z) = 90 * 46 = 4140$$

Y, finalmente, reescribiendo la formula anterior, la multiplicación quedaría:

$$0981 * 1234 = 10^4 * p + 10^2 * (r - p - q) + q$$

Para verificar que efectivamente es correcta la multiplicación sería:

$$\begin{aligned} 0981 * 1234 &= 10^4 * 108 + 10^2 * (4140 - 108 - 2754) + 2754 = \\ &= 1080000 + 127800 + 2754 = 1210554 \end{aligned}$$

Concluyendo en la misma solución que vimos con la multiplicación empleando el modo tradicional (el ya visto en las paginas anteriores).

Cuando los operandos son muy grandes, el tiempo requerido para las sumas es despreciable frente al tiempo que requiere una sola multiplicación. En este caso simple (el de nuestro ejemplo), nos dará igual quitar una multiplicación y hacer una suma.

El tiempo para multiplicar 2 números de n cifras usando el algoritmo clásico empleando esta última técnica (de 3 multiplicaciones) será:

$$3 * h(n/2) + g(n) = 3 * c * (n/2)^2 + g(n) = \frac{3}{4} * c * n^2 + g(n) = \frac{3}{4} * h(n) + g(n).$$

siendo:

$h(n)$: Tiempo de la implementación dada del algoritmo clásico $= c * n^2$

$g(n)$: Tiempo necesario para las sumas, desplazamientos y operaciones adicionales $g(n) \in \theta(n)$.

$g(n)$ resultará despreciable frente a $h(n)$, cuando n sea suficientemente grande, lo que significa que hemos ganado aproximadamente un 25% de velocidad en comparación con el algoritmo clásico (en el que hacíamos 4 multiplicaciones). Aun así, el nuevo algoritmo tiene **coste cuadrático** de nuevo.

Usando el algoritmo de forma recursiva, la *ecuación de recurrencia* (o el tiempo) será:

$$t(n) = 3 * t(n/2) + g(n)$$

Tendremos los siguientes datos:

a: Número de llamadas recursivas $= 3$

b: Reducción del problema en cada llamada $= 2$

$c * n^k$: Coste de las operaciones extras a las llamadas recursivas. Como hemos visto en el método clásico anterior, tendremos que $g(n) \in \theta(n) \Rightarrow k = 1$

Al igual que antes y de nuevo insistiremos (hay que dar el latazo con estas formulas, así sí que se aprenden bien), la ecuación es de **reducción por división** y la resolución de dicha recursividad es:

$$T(n) = \begin{cases} \theta(n^k) & \text{si } a < b^k \\ \theta(n^k * \log(n)) & \text{si } a = b^k \\ \theta(n^{\log_b a}) & \text{si } a > b^k \end{cases}$$

Como vimos en el caso de 4 multiplicaciones, tendremos que resolver la ecuación y luego ver que inequación es la adecuada $a = b^k \Rightarrow 3 > 2^1$

Por ello, de nuevo será el tercer caso y el coste es $T(n) \in \theta(n^{\log_b a}) = \theta(n^{\log_2 3})$.

NOTA DEL AUTOR: No sé muy bien como separar dichos algoritmos clásicos, ya que nos guiamos por el libro de Brassard y no lo separan con el orden “lógico” que se quisiera. Por tanto, tendremos dos algoritmos clásicos, uno, el básico (el más burdo), de 4 multiplicaciones y otro, una mejora, de 3 multiplicaciones.

Esta mejora merecerá la pena, ya que este algoritmo puede multiplicar dos enteros muy grandes más deprisa que el algoritmo clásico de multiplicación (insistimos, el primer algoritmo clásico, el de las 4 multiplicaciones). El algoritmo de divide y vencerás (el último que hemos visto) puede resultar más *lento* que el clásico cuando los enteros son demasiados pequeños. Por tanto, un algoritmo de divide y vencerás debe de evitar seguir avanzando recursivamente cuando el tamaño de los casos ya no lo justifique.

Cuando los números tienen **longitud impar** se pueden multiplicar fácilmente partiéndolos del modo más equitativo posible: un número de n cifras se parte en $\lfloor n/2 \rfloor$ cifras (redondeo por abajo) y otro de $\lceil n/2 \rceil$ cifras.

Si fuera el tamaño de las sumas ($w + x$ y $y + z$) impar, éstas no podrán sobrepasar del tamaño $1 + \lceil n/2 \rceil$. Por ejemplo, si fuera la multiplicación de 5678 y 6789, el valor de $r = (w + x) * (y + z) = 134 * 156$.

Para multiplicar números de distintos tamaños podremos emplear estos algoritmos posibles:

- Si **m y n no difieren en más de un factor 2** (es decir, como en nuestro caso de 981 y 1234), lo mejor es rellenar el operando más pequeño con ceros (sería el operando 981) para hacerlo de longitud igual a la del otro operando. El algoritmo de divide y vencerás utilizado con relleno está en $\theta(n^{\log_2 3})$ y el algoritmo clásico requiere un tiempo $\theta(mn)$ para calcular la multiplicación de 2 enteros. La constante oculta del primero tiene más posibilidades de ser más grande que la del segundo. Vemos, por tanto, que el algoritmo de divide y vencerás con relleno es más lento que el clásico cuando $m < n^{\log(3/2)}$.
- Para obtener un algoritmo realmente mejor, la idea es **segmentar el operando más largo**, v , en bloques de tamaño m y utilizar entonces el algoritmo de divide y vencerás para multiplicar u por cada bloque v , de tal manera que se multiplicarán parejas de operandos de igual tamaño. El tiempo total de ejecución necesario para multiplicar un número de n cifras por un número de m cifras está en $\theta(nm^{\log(3/2)})$.

7.2. El caso general

Tendremos el siguiente esquema, que está sacado del libro de problemas, ya que creo que es más claro como lo ponen allí que en el libro de teoría (Brassard):

```
fun divide-y-vencerás (problema)
  si suficientemente-simple (problema) entonces
    dev solucion-simple (problema)
  si no { No es solución suficientemente simple }
    { $p_1 \dots p_k$ } ← decomposicion (problema)
    para cada  $p_i$  hacer
       $s_i$  ← divide-y-vencerás ( $p_i$ )
    fpara
      dev combinacion ( $s_1 \dots s_k$ )
  fsi
ffun
```

Las funciones que han de particularizarse son:

- **suficientemente-simple**: Decide si un problema está por debajo del tamaño umbral o no.
- **solucion-simple**: Algoritmo para resolver los casos más sencillos, por debajo del tamaño umbral.
- **descomposicion**: Descompone el problema en subproblemas en tamaño menor.
- **combinacion**: Algoritmo que combina las soluciones a los subproblemas en solución al problema del que provienen.

Algunos algoritmos de divide y vencerás no siguen exactamente este esquema, puesto que hay casos en los que no tiene sentido reducir la solución de un caso muy grande a la de uno más pequeño. Entonces, divide y vencerás recibe el nombre de **reducción (simplificación)**.

Para que el enfoque de divide y vencerás merezca la pena es necesario que se cumplan estas **tres condiciones**:

1. La decisión de utilizar el subalgoritmo básico (suficientemente-simple) en lugar de hacer llamadas recursivas debe tomarse *cuidadosamente*.
2. Tiene que ser posible descomponer el ejemplar y en subejemplares y recomponer las soluciones parciales de forma bastante eficiente.
3. Los subejemplares deben ser en la medida de lo posible aproximadamente del mismo tamaño.

Tendremos que decidir la forma en la que dividir el caso y hacer llamadas recursivas o si el caso es tan sencillo que resulte mejor invocar directamente el subconjunto básico. Esta decisión se basa en un sencillo umbral, llamado n_0 . El subalgoritmo básico se emplea para resolver todos aquellos casos cuyo tamaño no supere n_0 .

La selección del mejor umbral se ve **complicada** por el hecho de que el valor óptimo depende en general no sólo del algoritmo en cuestión, sino también de la implementación particular.

7.3. Búsqueda binaria

Probablemente se trate de la aplicación más sencilla de divide y vencerás, tan sencilla que hablando con propiedad se trata de una aplicación de *reducción* (*simplificación*) más que de divide y vencerás.

Se nos da un vector $T[1..n]$ ordenado por orden creciente, esto es, $T[i] \leq T[j]$ siempre que $1 \leq i \leq j \leq n$. Sea x un elemento. El **problema** consiste en buscar x en la matriz T , si es que está. Formalmente, deseamos encontrar el índice i tal que $1 \leq i \leq n + 1$ y $T[i - 1] < x < T[i]$, con la convención lógica consistente en que $T[0] = -\infty$ y $T[n + 1] = +\infty$.

La primera aproximación consiste en examinar secuencialmente los elementos de T , hasta que o bien lleguemos al final de la matriz o bien encontramos un elemento que no sea menor que x :

```
fun secuencial (T[1..n], x)
  { Búsqueda secuencial de x en una matriz }
  para i ← 1 hasta n hacer
    si T[i] ≥ x entonces devolver i
  devolver n + 1;
```

Este algoritmo requiere un tiempo que está en $\theta(r)$, donde r es el índice que se devuelve. En el caso peor y en el promedio, la búsqueda secuencial requiere un tiempo que está en $\theta(n)$, porque el número medio de pasadas por el bucle es $\frac{n+1}{2}$.

Para acelerar la búsqueda deberíamos buscar x en la primera mitad de la matriz o en la segunda. Para averiguar cuál de estas búsquedas es la correcta, comparamos x con un elemento de la matriz. Sea $k = \lceil n/2 \rceil$. Si $x \leq T[k]$, entonces se puede restringir la búsqueda de x a $T[1..k]$ (mitad izquierda); en caso contrario, basta con buscar en $T[k + 1..n]$ (mitad derecha).

Para evitar comparaciones repetitivas en cada llamada recursiva, es mejor verificar desde un comienzo si la respuesta es $n + 1$, esto es, si x está a la derecha de T .

NOTA DEL AUTOR (IMPORTANTE): En exámenes se verá que pidan que se detecte si el elemento x está fuera de la matriz, esto quiere decir la anterior expresión “estar a la derecha de T ”. Por eso, el siguiente algoritmo es muy importante el controlarlo (entró en los últimos exámenes de 2007-2008).

Dicho esto, tenemos el siguiente algoritmo:

```
funcion busquedabin (T[1..n], x)
  si n = 0 ó x > T[n] entonces devolver n + 1
  si no { Elemento dentro del vector }
    devolver binrec (T[1..n], x)
```

Observamos e insistimos por su importancia que la llamada inicial (busquedabin) verifica que **el elemento esté dentro del vector**.

Por tanto, la función propiamente de búsqueda binaria será:

```

funcion binrec (T[i..j], x)
    { Búsqueda binaria de x en la submatriz T[i..j] con la seguridad de que
      T[i - 1] < x ≤ T[j] }
    si i = j entonces devolver i
    k ← (i + j) ÷ 2
    si x ≤ T[k] entonces devolver binrec (T[i..k], x)    { Mitad izquierda}
    si no devolver binrec (T[k + 1..j], x)              { Mitad derecha }

```

Como añadido del autor, tendremos otra manera para averiguar si el elemento está en el vector T y es sustituyendo en el primer “si” del algoritmo anterior.

```

si i = j entonces
    si T[i] = x entonces    { Elemento encontrado }
        dev i
    si no                    { Elemento fuera del vector T }
        dev -1

```

Ambos algoritmos son iguales, es decir, el del *busquedabin* y esta sustitución del bucle “si”. Bajo mi punto de vista lo que mejor encuentro es hacerlo tal y como viene en el libro, es decir, con ambos procedimientos distintos, así es seguro que esté correcto, aunque ver varias maneras nunca viene de más.

Sea $t(m)$ el tiempo requerido por una llamada a binrec ($T[i..j]$, x), en donde $m = j - i + 1$ (siendo m el tamaño del problema) es el número de elementos que restan a efectos de búsqueda. El tiempo requerido por una llamada a *busquedabin* ($T[i..j]$, x) es claramente $t(n)$ salvo por una pequeña constante aditiva.

Analizaremos el coste de este algoritmo, teniendo en cuenta que es una **reducción por división**. Por tanto, tendremos la siguiente ecuación de recurrencia:

$$t(n) = 1 * t(n/2) + O(1)$$

Las distintas variables son:

- a:** Número de llamadas recursivas = 1, siendo este valor por haber una llamada a un subproblema por cada bucle “si”.
- b:** Reducción del problema en cada llamada = 2
- c * n^k:** Coste de las operaciones extras a las llamadas recursivas. Tendremos que el valor de $k = 0$, al ser constante ($O(1)$).

De nuevo, la resolución de la **recurrencia por división** es:

$$T(n) = \begin{cases} \theta(n^k) & \text{si } a < b^k \\ \theta(n^k * \log(n)) & \text{si } a = b^k \\ \theta(n^{\log_b a}) & \text{si } a > b^k \end{cases}$$

Resolvemos la ecuación $a = b^k$, en este caso, tenemos que $1 = 2^0$, por lo que el caso es el segundo. Sustituyendo, el coste del algoritmo es $\theta(n^k * \log(n)) = \theta(\log(n))$.

NOTA DEL AUTOR: Es muy **IMPORTANTE** tener MUY claro este coste, porque suele ser un problema bastante fácil y el coste muchas veces se confunde como puede ser, por ejemplo, con la ordenación por montículo..., que es $\theta(n * \log(n))$.

Esta sería la versión recursiva del algoritmo de búsqueda binaria:

```
funcion binrec (T[i..j], x)
{ Búsqueda binaria iterativa en x en la matriz T }
si  $x > T[n]$  entonces devolver  $n + 1$ 
 $i \leftarrow 1; j \leftarrow n$ 
mientras  $i < j$  hacer
    {  $T[i - 1] < x \leq T[j]$  }
     $k \leftarrow (i + j) \div 2$ 
    si  $x \leq T[k]$  entonces  $j \leftarrow k$ 
    si no  $i \leftarrow k + 1$ 
devolver i
```

Este algoritmo gráficamente haría algo así:



En este caso, no es tan directa la búsqueda del elemento en ambas mitades. Según parece y es algo que he deducido con el código el puntero i se desplazaría a la derecha (sumando posiciones), mientras que el j al revés, de tal manera que hasta que i sea mayor que j seguirá haciendo estos movimientos de puntero (acabaría el algoritmo cuando i supere a j). Toda la explicación es una estimación del autor.

Por último, decir que el coste del algoritmo es **exactamente igual** al que vimos con la versión recursiva, es decir, $\theta(\log(n))$.

De nuevo, es igualmente **básico, fundamental** el quedarnos con este coste. Se insistirá numerosas veces.

7.4. Ordenación

Sea $T[1..n]$ una matriz de n elementos. Nuestro problema es ordenar estos elementos por orden *ascendente*. Previamente, hemos visto que se puede resolver mediante ordenación por selección y ordenación por inserción o mediante ordenación por montículo. Este último algoritmo de ordenación tiene coste en el caso peor y promedio $\theta(n * \log(n))$, mientras que los dos anteriores tienen coste cuadrático $\theta(n^2)$. Hay varios algoritmos de ordenación que siguen el esquema de divide y vencerás. Veremos con más detenimiento dos de ellos:

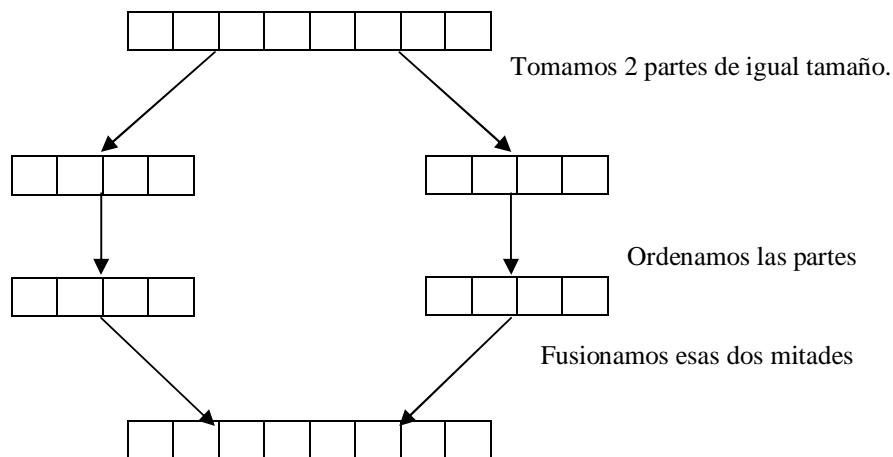
- Ordenación por fusión (mergesort).
- Ordenación rápida (quicksort).

7.4.1. Ordenación por fusión (mergesort)

El problema consiste en:

- **Descomponer** la matriz T en dos partes cuyos tamaños sean tan parecidos como sea posible.
- **Ordenar** estas partes mediante llamadas recursivas.
- **Fusionar** las soluciones en cada parte, teniendo buen cuidado de mantener el orden.

Un ejemplo de este algoritmo podría ser el siguiente:



Observamos, aunque sólo es un ejemplo a grosso modo, que el vector tras fusionar ya está ordenado.

El problema es como fusionar esas mitades y hacerlo eficiente. Emplearemos como centinela (una especie de posición auxiliar, para evitar realizar cálculos extras) la última posición en las matrices U y V.

```

procedimiento fusionar ( $U[1..m + 1]$ ,  $V[1..n + 1]$ ,  $T[1..m + n]$ )
    { Fusiona las matrices ordenadas  $U[1..m]$  y  $V[1..n]$  almacenándolas
    en  $T[1..m + n]$ ,  $U[m + 1]$  y  $V[n + 1]$  se utilizan como centinelas }
     $i, j \leftarrow 1$ ;
     $U[m + 1], V[n + 1] \leftarrow \infty$ 
    para  $k \leftarrow 1$  hasta  $m + n$  hacer
        si  $U[i] < V[j]$  entonces
             $T[k] \leftarrow U[i]; i \leftarrow i + 1$ 
        si no
             $T[k] \leftarrow V[j]; j \leftarrow j + 1$ 

```

El algoritmo de **ordenación por fusión** es como sigue, en donde utilizamos la ordenación por inserción (insertar) como subalgoritmo básico, que también añadiremos a continuación (tomándolo del tema 2).

```

procedimiento ordenarporfusion ( $T[1..n]$ )
    si  $n$  es suficientemente pequeño entonces
        insertar ( $T$ )
    si no
        matriz  $U[1..1 + \lfloor n/2 \rfloor]$ ,  $V[1..1 + \lfloor n/2 \rfloor]$ 
         $U[1.. \lfloor n/2 \rfloor] \leftarrow T[1.. \lfloor n/2 \rfloor]$ 
         $V[1.. \lfloor n/2 \rfloor] \leftarrow T[1 + \lfloor n/2 \rfloor.. n]$ 
        ordenarporfusion ( $U[1.. \lfloor n/2 \rfloor]$ )
        ordenarporfusion ( $V[1.. \lfloor n/2 \rfloor]$ )
        fusionar ( $U, V, T$ )

```

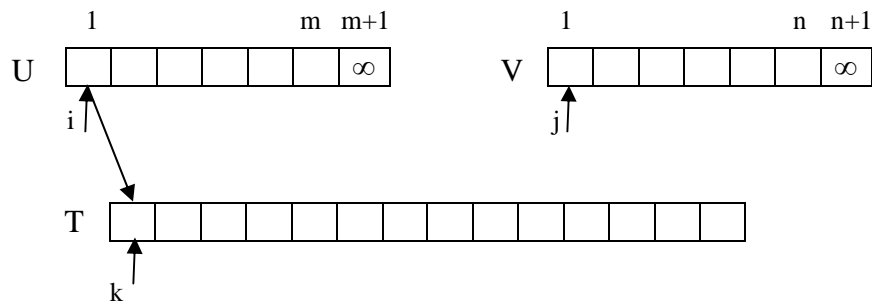
Usaremos la función de *fusionar* anterior. Vamos a ver la función de *insertar* tal y como hemos comentado previamente:

```

procedimiento insertar  $T([1..n])$ 
    para  $i \leftarrow 2$  hasta  $n$  hacer
         $x \leftarrow T[i]; j \leftarrow i - 1$ ;
        mientras  $j > 0$  y  $x < T[j]$  hacer
             $T[j + 1] \leftarrow T[j]$ ;
             $j \leftarrow j - 1$ ;
         $T[j + 1] \leftarrow x$ 

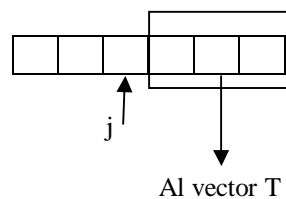
```

Gráficamente, empleando centinelas, como hemos explicado antes, tendremos el funcionamiento de la ordenación por fusión como sigue:



Tenemos dos punteros i y j . Comparamos los elementos a los que apuntan estos dos punteros y vemos cuál es el menor, para luego copiarlo a T . Al copiarlo, incrementaríamos i y k , para desplazar una posición en ambos vectores (U y T), en este caso. Dependerá de la comparación, ver cuál es el puntero menor (si i o j).

Para verificar que hemos llegado al final, al ir incrementándose ambos punteros (i o j), tendremos un momento en que uno de los dos o ambos lleguen al valor del centinela. Se nos daría un caso en que el puntero i llegara a dicho centinela y que el j no lo hiciera, eso querrá decir que los elementos menores de j ya están en T , por lo que los elementos hasta llegar a n (sin contar $n + 1$) de V se copiarían directamente a T (creo recordar que era copia de cabos en estructura de datos). Sería algo así:



Veremos un **ejemplo** práctico de una matriz a ordenar:

Matriz que hay que ordenar

3	1	4	1	5	9	2	6	5	3	5	8	9
---	---	---	---	---	---	---	---	---	---	---	---	---

La matriz se parte en dos mitades

3	1	4	1	5	9
---	---	---	---	---	---

2	6	5	3	5	8	9
---	---	---	---	---	---	---

Una llamada recursiva a *ordenar por fusión* para cada mitad

1	1	3	4	5	9
---	---	---	---	---	---

2	3	5	3	6	8	9
---	---	---	---	---	---	---

Una llamada a *fusionar*

1	1	2	3	3	4	5	5	5	6	8	9	9
---	---	---	---	---	---	---	---	---	---	---	---	---

El **procedimiento** de la ordenación por fusión es:

- Cuando el número de elementos que hay que ordenar es pequeño, se utiliza un algoritmo relativamente sencillo.
- Por otra parte, cuando esté justificado por el número de elementos, *ordenarporfusion* separa el ejemplar en dos subejemplares de tamaño mitad, resuelve las dos recursivamente y entonces combina las dos medias matrices ya ordenadas para obtener la solución del ejemplar original.

Analizaremos el coste de la **ordenación por fusión** separándolo por partes distintas, es decir, siguiendo el funcionamiento anteriormente dicho:

- La separación de T en U y V requiere un **tiempo lineal**.
- *fusionar* (U, V, T) también requiere un **tiempo lineal**.
- Tendremos la ecuación de recurrencia siguiente $t(n) = t(\lfloor n/2 \rfloor) + t(\lceil n/2 \rceil) + g(n)$, donde $g(n) \in \theta(n)$. Por tanto, esta recurrencia pasa a ser $t(n) = 2 * t(n/2) + g(n)$ cuando n es par.

Resolvemos la recurrencia por reducción por división con los siguientes valores:

a: Número de llamadas recursivas = 2

b: Reducción del problema en cada llamada = 2

c * n^k : Coste de las operaciones extras a las llamadas recursivas. Tendremos que el valor de $k = 1$, al ser el tiempo extra lineal.

De nuevo, el valor de $a = b^k \Rightarrow 2 = 2^1$, siendo el caso el segundo y resolviéndose así: $t(n) \in \theta(n^k * \log(n)) = \theta(n * \log(n))$.

Por tanto, la eficiencia de *ordenar por fusión* (*mergesort*) es similar a la de *ordenación por montículo* (*heapsort*). La ordenación por fusión puede ser ligeramente más rápida en la práctica, pero requiere una cantidad significativamente mayor de **espacio** para las matrices intermedias U y V (recordemos que la ordenación por montículo puede hacerse *in situ*).

Cuando se crean subejemplares de distinto tamaño (los ejemplares están mal distribuidos) nos queda la siguiente variante (muy mala, por cierto) del algoritmo de *ordenación por fusión*, en la que tendremos una parte en la que se tienen todos los elementos menos uno y en la otra parte tendremos un único elemento. Por ello, será en este caso una **reducción por sustracción**.

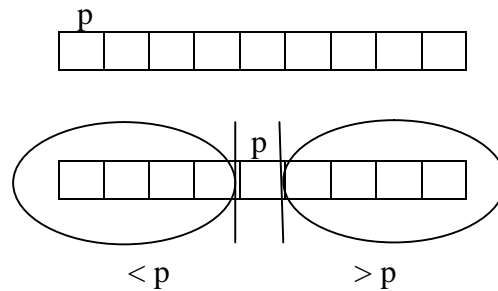
```
procedimiento ordenarporfusionmala (T[1..n])
si n es suficientemente pequeño entonces
    insertar (T)
si no
    matriz U[1..1 + ⌊n/2⌋], V[1..1 + ⌊n/2⌋]
    U[1..n - 1] ← T[1..n - 1]
    V[1] ← T[n]
    ordenarporfusion (U[1..n - 1])
    ordenarporfusion (V[1..1])
    fusionar (U, V, T)
```

Por lo anteriormente escrito, el coste es de $\theta(n^2)$, considerándose evidentemente coste **ineficiente** con respecto al otro algoritmo.

7.4.2. Ordenación rápida (quicksort)

El **funcionamiento** del algoritmo es el siguiente:

- El algoritmo selecciona como **pivote** uno de los elementos de la matriz que haya que ordenar.
- La matriz se parte a ambos lados del pivote: se desplazan los elementos de tal manera que los que sean mayores que el pivote (p) queden a la derecha, mientras que los demás queden a su izquierda. Quedaría algo así, gráficamente:



- El pivote quedará ahora en su posición definitiva. Tendremos un pequeño problema al colocar los elementos iguales al pivote (p), aunque en este caso nos dará igual donde los coloquemos, si a la derecha o a la izquierda.
- Si ahora las partes de la matriz que quedan a ambos lados del pivote se ordenan independientemente mediante llamadas recursivas al algoritmo, el resultado final es una matriz completamente ordenada.

Para equilibrar los tamaños de los dos subcasos que hay que ordenar, lo idóneo es utilizar el elemento **mediana** como pivote. Expondremos el concepto de mediana brevemente, para así comprenderlo mejor, aunque más adelante lo veremos con más detenimiento.

Se define a la *mediana* de T como su $\lceil n/2 \rceil$ -ésimo elemento. Por tanto, la mediana es aquel elemento de T , tal que en T hay tantos elementos más pequeños que él como elementos mayores que él.

Por ejemplo, en este vector:

3	1	4	1	5	9	2	6	5
---	---	---	---	---	---	---	---	---

la mediana es el número 4, porque ordenado es:

1	1	2	3	4	5	5	6	9
---	---	---	---	---	---	---	---	---

Aunque veremos esta definición en este apartado, lo veremos con más detenimiento en el 7.5, empleando para ello una definición más formal. Aun así, el ejemplo que pondremos será similar.

Desafortunadamente, encontrar la mediana requiere un tiempo excesivo. Por esta razón, nos limitaremos a utilizar como pivote un elemento arbitrario de la matriz y esperemos tener suerte.

Es necesario que la *constante multiplicativa sea pequeña*, para que quicksort sea competitivo con otras técnicas de ordenación como la ordenación por montículo. Podremos tener estos pasos:

- Supongamos que es preciso descomponer la submatriz $T[i..j]$ empleando como pivote $p = T[i]$. Una buena forma de hacer la descomposición consiste en explorar la submatriz una sola vez, pero empezando en los dos extremos. Los punteros k y l se inicializan i y $j + 1$, respectivamente.
- A continuación, se incrementa el puntero k hasta que $T[k] > p$ y se decrementa el puntero l hasta que $T[l] \leq p$. Ahora se intercambian $T[k]$ y $T[l]$. Este proceso continúa mientras sea $k < l$.
- Finalmente, se intercambian $T[i]$ y $T[l]$ para poner el pivote en la posición correcta.

Los procedimientos serán estos, empezando por el del pivote:

```

procedimiento pivote ( $T[i..j]$ , var  $l$ )
{ Permuta los elementos de la matriz  $T[i..j]$  y proporciona un valor  $l$ ,
tal que, al final,  $1 \leq l \leq j$ ;  $T[k] \leq p$  para todo  $i \leq k < l$ ,  $T[l] = p$ , y
 $T[k] > p$  para todo  $1 < k \leq j$ , en donde  $p$  es el valor inicial de  $T[i]$  }
 $p \leftarrow T[i]$ ;
 $k \leftarrow i$ ;  $l \leftarrow j + 1$ 
repetir  $k \leftarrow k + 1$  hasta que  $T[k] > p$  o  $k \geq j$ 
repetir  $l \leftarrow l - 1$  hasta que  $T[l] \leq p$ 
mientras  $k < l$  hacer
    intercambiar  $T[k]$  y  $T[l]$ 
    repetir  $k \leftarrow k + 1$  hasta que  $T[k] > p$ 
    repetir  $l \leftarrow l - 1$  hasta que  $T[l] \leq p$ 
intercambiar  $T[i]$  y  $T[l]$ 

```

El algoritmo siguiente es el propio de la ordenación rápida (quicksort):

```

procedimiento quicksort ( $T[i..j]$ )
{ Ordena la submatriz  $T[i..j]$  por orden no decreciente }
si  $j - i$  es suficientemente pequeño entonces
    insertar ( $T[i..j]$ )
si no
    pivote ( $T[i..j]$ ,  $l$ )
    quicksort ( $T[i..l - 1]$ )
    quicksort ( $T[l + 1..j]$ )

```

Nos fijamos que usa la función *insertar*, visto ya en el apartado anterior.

Un **ejemplo** de este mismo algoritmo será el siguiente y así fijar los conceptos vistos previamente. Tendremos que ordenar este vector, idéntico al de la ordenación por fusión:

Matriz que hay que ordenar

3	1	4	1	5	9	2	6	5	3	5	8	9
---	---	---	---	---	---	---	---	---	---	---	---	---

La matriz se particiona tomando como pivote su primer elemento, $p = 3$

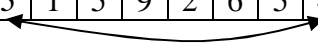
3	1	4	1	5	9	2	6	5	3	5	8	9
----------	---	---	---	---	---	---	---	---	---	---	---	---

Se busca el primer elemento mayor que el pivote (subrayado, puntero k) y el último elemento no mayor que el pivote (superrayado, puntero l)

3	1	<u>4</u>	1	5	9	2	6	5	³	5	8	9
----------	---	----------	---	---	---	---	---	---	--------------	---	---	---

Se intercambian esos elementos

3	1	3	1	5	9	2	6	5	4	5	8	9
----------	---	---	---	---	---	---	---	---	---	---	---	---



Se vuelve a explorar en ambas direcciones

3	1	3	1	<u>5</u>	9	²	6	5	4	5	8	9
----------	---	---	---	----------	---	--------------	---	---	---	---	---	---

Se intercambian

3	1	3	1	2	9	5	6	5	4	5	8	9
----------	---	---	---	---	---	---	---	---	---	---	---	---

Se explora

3	1	3	1	²	<u>9</u>	5	6	5	4	5	8	9
----------	---	---	---	--------------	----------	---	---	---	---	---	---	---

Los punteros se han cruzado (el elemento superrayado está a la izquierda del subrayado, $k > l$): se intercambia el pivote con el elemento superrayado

2	1	3	1	3	9	5	6	5	4	5	8	9
---	---	---	---	----------	---	---	---	---	---	---	---	---

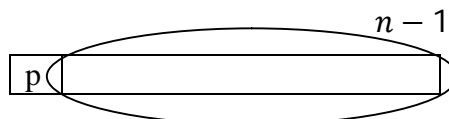
La partición ya está ordenada

Se ordenan recursivamente las submatrices a cada lado del pivote

1	1	2	3	3	4	5	5	5	6	8	9	9
---	---	---	---	----------	---	---	---	---	---	---	---	---

Vemos estos casos de colocación del pivote:

- **Cuando el pivote p queda en un extremo** (al inicio o al final, da igual): Tendremos una versión no equilibrada de ordenación rápida, en la que el tamaño del problema se reduce en una mitad. La situación tras colocar los elementos menores y mayores es:



La ecuación de recurrencia, por tanto, sería:

$$t(n) = t(n - 1) + O(n)$$

Recolocación
del pivote

Las distintas variables al igual que hemos visto previamente es:

a: Número de llamadas recursivas = 1

b: Reducción del problema en cada llamada = 1

$c * n^k$: Coste de las operaciones extras a las llamadas recursivas.
Tendremos que el valor de $k = 1$, al ser el tiempo extra lineal.

Recordemos que la resolución para la reducción de la **recurrencia por sustracción** es la siguiente:

$$T(n) = \begin{cases} \theta(n^k) & \text{si } a < 1 \\ \theta(n^{k+1}) & \text{si } a = 1 \\ \theta(a^{n \div b}) & \text{si } a > 1 \end{cases}$$

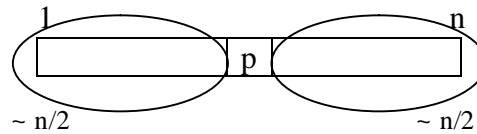
Por tanto, vemos que $a = 1$, por lo que estaremos en el segundo caso. Pasamos a resolverlo, siendo el tiempo $t(n) \in \theta(n^{k+1}) = \theta(n^2)$.

El algoritmo se comporta muy mal en este caso, siendo éste el **caso peor**. Veremos un ejemplo de este caso peor, si T ya está ordenado antes de la llamada a quicksort obtenemos $l = i$ en todas las ocasiones, lo cual implica una llamada recursiva a un caso de tamaño 1 y otra a un caso cuyo tamaño se reduce en una unidad.

Este caso podremos compararlo con el peor de la ordenación por fusión (donde estaban descompensadas las particiones), en la que recordemos tenía coste cuadrático.

- Por otra parte, **si los elementos de la matriz que hay que ordenar se encuentran inicialmente en orden aleatorio**, tendremos que los subejemplares para ordenar estarán suficientemente bien equilibrados.

En el caso peor, tendremos:



El pivote está ya ordenado.

Tendremos esta **recurrencia por división**:

$$t(n) = 2 * t(n/2) + O(n)$$

Recolocación
del pivote

De nuevo, los valores de las variables son:

a: Número de llamadas recursivas = 2

b: Reducción del problema en cada llamada = 2

c * n^k: Coste de las operaciones extras a las llamadas recursivas.

Tendremos que el valor de $k = 1$, al ser el tiempo extra lineal.

Resolvemos la ecuación $a = b^k$, siendo éste $2 = 2^1$, que es el segundo caso:

$$(n) \in \theta(n^k * \log(n)) = \theta(n * \log(n)).$$

Este caso correspondería con el **mejor caso** de esta ordenación.

- Para calcular el **tiempo promedio** haremos una suposición acerca de la distribución de probabilidad de los casos de los n elementos. La suposición más natural es que los elementos de T sean diferentes y que todas las $n!$ permutaciones iniciales posibles de los elementos son igualmente equiprobables.

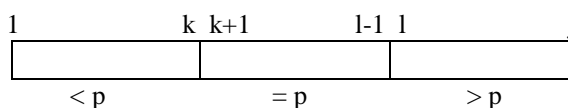
El tiempo medio requerido es:

$$t(n) = \frac{1}{n} \sum_{i=1}^n (g(n) + t(l-1) + t(n-l))$$

Realizando diversos cálculos llegamos a la conclusión que el coste requiere un tiempo en $O(n * \log(n))$ para ordenar n elementos en el caso medio.

Comparación en cuanto a costes de las distintas ordenaciones: podremos decir que la constante oculta es más pequeña que las asociadas en la *ordenación por montículo* o en *ordenar por fusión* cuando funciona bien (el caso mejor). Aun cuando seleccionemos la mediana de $T[i..j]$ como pivote, que se puede hacer en tiempo lineal, la *ordenación rápida (quicksort)* sigue requiriendo un **tiempo cuadrático** en el caso peor, lo cual sucede si son iguales todos los elementos que hay que ordenar (ojito con esta parte, puede entrar perfectamente en exámenes).

Para intentar mejorar el tiempo en el caso peor tendremos que descomponer T en 3 secciones, empleando p como pivote: después de la partición, los elementos de $T[i..k]$ son más pequeños que p , los de $T[k+1..l-1]$ son iguales a p y la de $T[l..j]$ son más grandes que p . Tendremos lo siguiente:



Tendremos este nuevo procedimiento:

procedimiento pivotebis ($T[i..j]$, p ; var k, l).

Después de hacer la partición con una llamada a *pivotebis* ($T[i..j]$, $T[i]$, k, l) hay que llamar a quicksort recursivamente con $T[i..k]$ y $T[l..j]$.

Quicksort requiere ahora un tiempo en el caso peor $O(n * \log(n))$.

7.5. Búsqueda de la mediana

Recordemos otra vez la **definición** de la mediana dada anteriormente, aunque de modo breve (en la ordenación rápida o quicksort):

Sea $T[1..n]$ una matriz de enteros y sea s un entero entre 1 y n . Se define el s -ésimo elemento de T como aquél elemento que se encontraría en la s -ésima posición si ordenara T en orden no decreciente. Dados T y s , el problema de encontrar el s -ésimo elemento de T se conoce como el problema de selección. En particular, se define la mediana de T como su $[n/2]$ -ésimo elemento.

Cuando n es impar y los elementos de T son diferentes, la mediana es simplemente aquel elemento de T , tal que en T hay tantos elementos más pequeños que él como elementos mayores que él.

Un **ejemplo** podrá ser este, en el que nos piden que calculemos la mediana de:

3	1	4	1	5	9	2	6	5
---	---	---	---	---	---	---	---	---

es 4, puesto que 3, 1, 1 y 2 son más pequeños que 4, mientras que 5, 9, 6 y 5 son mayores. Llegaremos a esta conclusión tras ordenar el vector, cosa que habremos hecho previamente.

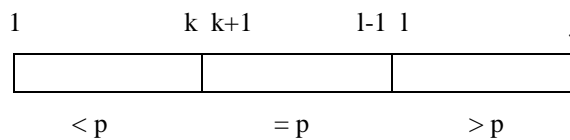
Un **algoritmo sencillo** para determinar la mediana de $T[1..n]$ consiste en ordenar la matriz y extraer entonces el $[n/2]$ -ésimo elemento. Si utilizamos *ordenación por montículo* u *ordenación por fusión*, esto requiere un tiempo que está en $O(n * \log(n))$ (de nuevo hay que saberse de memoria este coste, es básico).

Es evidente que todo algoritmo para el problema de selección se puede utilizar para hallar la mediana: basta con seleccionar el $[n/2]$ -ésimo elemento más pequeño.

NOTA DEL AUTOR: Este último párrafo supongo que querrá decir que con cualquier algoritmo de ordenación ordenaremos el vector y basta con seleccionar dicho elemento más pequeño. Está copiado literalmente del libro.

Para calcular el s-ésimo elemento más pequeño tendremos este algoritmo, resolviéndolo de forma parecida a la última mejora de quicksort, para ello usamos el procedimiento *pivotebis*. Recordemos que la llamada a *pivotebis* ($T[i..j]$, p ; var k, l) particiona a $T[i..j]$ en 3 secciones. T se organiza de tal manera que los elementos de $T[i..k]$ sean más pequeños que p , los de $T[k + 1..l - 1]$ sean iguales a p y los de $T[l..j]$ sean mayores que p . Se nos darán estos **casos** posibles:

- Tras una llamada a *pivotebis* (T, P, k, l) hemos terminado si $k < s < l$, puesto que entonces el s-ésimo elemento más pequeño de T es igual a p .
- Si $s \leq k$, entonces el s-ésimo elemento más pequeño de T es ahora el s-ésimo elemento más pequeño de $T[1..k]$.
- Por último, si $s \geq l$, entonces el s-ésimo elemento más pequeño de T es ahora el $(s - l + 1)$ -ésimo elemento más pequeño de $T[1..n]$.



Tendremos este algoritmo para el cálculo del s-ésimo elemento más pequeño:

```

funcion selección ( $T[1..n]$ ,  $s$ )
{ Busca el s-ésimo elemento más pequeño de  $T$ ,  $1 \leq s \leq n$  }
 $i \leftarrow 1$ ;  $j \leftarrow n$ 
repetir
{ La respuesta se encuentra en  $T[i..j]$  }
 $p \leftarrow \text{mediana}(T[i..j])$ 
 $\text{pivotebis}(T[i..j], p, k, l)$ 
si  $s \leq k$  entonces  $j \leftarrow k$ 
si no
    si  $s \geq l$  entonces  $i \leftarrow l$ 
    si no
        devolver  $p$ 

```

Para hacer **más eficiente** el algoritmo tendremos que seleccionar el *pivote* como $p \leftarrow T[i]$. Esto da lugar a que el algoritmo invierta un tiempo cuadrático en el caso peor (como pasaba con quicksort). Sin embargo, en el caso promedio el algoritmo modificado funciona en un tiempo lineal.

Para **mejorar el caso peor** de orden cuadrático tendremos que hacer que el número de pasadas por el bucle siga siendo logarítmico, siempre y cuando seleccionemos un pivote cercano a la mediana, lo que se denomina pseudomediana. El tiempo en el caso peor para hallar el s-ésimo elemento más pequeño usando el método de la pseudomediana es lineal.

La última parte, la de la pseudomediana, debido a su complejidad para resumirla, prácticamente no la hemos tocado, por tanto, se dejaría como lectura obligada y comprensiva.

7.6. Multiplicación de matrices

El algoritmo clásico de multiplicación de matrices sigue su definición:

$$C_{ij} = \sum_{k=1}^n A_{ik} * B_{kj}$$

El tiempo está en $\theta(n^2)$ para calcularse n^2 entradas.

Una posible mejora es multiplicar una matriz 2 x 2 en vez de en 8 multiplicaciones en 7 (parecido a la multiplicación de enteros muy grandes). La reducción de una multiplicación con respecto a más sumas es insignificante cuando el tamaño del problema es pequeño, pero es importante el ahorro cuando las matrices son grandes.

La ecuación de recurrencia es:

$$t(n) = 7 * t(n/2) + g(n)$$

Deduciendo de esta ecuación, tendremos las siguientes variables para la **reducción por división**:

a: Número de llamadas recursivas = 7

b: Reducción del problema en cada llamada = 2

c * n^k: Coste de las operaciones extras a las llamadas recursivas. El significado de $g(n)$ es el coste de sumar y restar matrices, que sería $\theta(n^2)$.

Por tanto, el valor de k es $k = 2$.

Resolviendo la recurrencia tendremos lo siguiente:

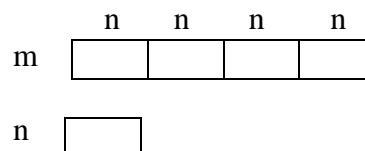
$$T(n) = \begin{cases} \theta(n^k) & \text{si } a < b^k \\ \theta(n^k * \log(n)) & \text{si } a = b^k \\ \theta(n^{\log_b a}) & \text{si } a > b^k \end{cases}$$

Como en ocasiones anteriores sustituyendo en la ecuación $a = b^k$, tendremos la inecuación $7 > 2^2$, por lo que estaremos en el caso tercero, con coste $t(n) \in \theta(n^{\log_b a}) = \theta(n^{\log_2 7})$. Nos fijamos que mejoramos el coste visto en el algoritmo que emplea 8 multiplicaciones.

7.7. Exponenciación

Como recordatorio veremos esta comparación de costes entre los métodos hasta el momento vistos, en la que las multiplicaciones son *operaciones elementales*, es decir, que no tiene coste extra hacerlas:

- **Método clásico:** Tenemos que $T(m, n) \in \theta(m^2 n^2)$, demostrando que ambas cotas (inferior y superior) tienen el mismo coste.
- **Divide y vencerás:** Empleando una multiplicación de dos números de n cifras tiene coste $O(n^{\log_2 3})$. Recordemos, que lo vimos previamente y que era el coste hacer 3 multiplicaciones en vez de 4, por lo que sería el algoritmo ya mejorado. Tendremos distintos casos usando el algoritmo de divide y vencerás:
 1. Si **multiplicamos dos números grandes**, como hemos hecho en este apartado, empleando para ello un algoritmo de multiplicación mediante el esquema de divide y vencerás, tendremos que $T(m, n) \in \theta(n^2 m^{\log_2 3})$.
 2. Si **multiplicamos dos números de n y m cifras respectivamente**, siendo m mucho mayor que n ($n \ll m$), tendríamos lo siguiente:



Haremos $\frac{m}{n}$ multiplicaciones de dos números de n cifras, por lo que el **coste** será:

$$O\left(\frac{m}{n} n^{\log_2 3}\right) = O(m n^{\log_2 3 - 1}) = O(m n^{\log_2 3 - \log_2 2}) = O(m n^{\log_2(3/2)})$$

Sean a y n dos enteros. Deseamos calcular la exponenciación $x = a^n$. Por sencillez, supondremos en toda esta sección que $n > 0$. Si n es pequeño, el **algoritmo evidente** resulta adecuado:

```
funcion exposec (a, n)
  r ← a
  para i ← 1 hasta n − 1 hacer r ← a * r
  devolver r
```

que equivale a hacer:

$$x = a^n = \underbrace{a * a * \dots * a}_{n \text{ veces}} = a * a^{n-1}$$

Este algoritmo requiere un tiempo que está en $\theta(n)$, puesto que la instrucción $r \leftarrow a * r$ se ejecuta exactamente $n - 1$ veces, *siempre y cuando las multiplicaciones cuenten como operaciones elementales*. El inconveniente para analizar el coste es que el número que se multiplica puede ser grande.

A continuación y en lo que acaba el apartado, veremos la multiplicación cuando los **operandos son muy grandes**, para ello es preciso tener en cuenta el tiempo necesario para cada multiplicación. Sea $M(q, s)$ el tiempo necesario para multiplicar dos enteros de tamaños q y s . Supongamos que $q_1 \leq q_2$ y $s_1 \leq s_2$ implican que $M(q_1, s_1) \leq M(q_2, s_2)$. Estimemos cuánto tiempo invierte nuestro algoritmo multiplicando enteros cuando se invoca *exposec* (a, n):

Como añadido del autor veremos cuánto cambia el tamaño de r en cada pasada (recordemos que teníamos que calcular a^r). Aplicaremos esta idea al problema nuestro.

Haremos un producto de i y j cifras respectivamente, teniendo dos casos:

- $i + j$ en el *peor* caso.
- $i + j - 1$ en el *mejor* caso.

Veremos el exponente, el valor y el tamaño de r :

Exponente	1	2	3	i
Valor	a	a^2	a^3	a^i
Tamaño	m	$\begin{cases} 2m \\ 2m - 1 \end{cases}$	$\begin{cases} 3m \\ 3m - 2 \end{cases}$	$\begin{cases} im \\ im - i + 1 \end{cases}$

El tamaño indica el número de cifras de la exponenciación.

En el nivel i de la exponenciación los casos que se nos dan son los siguientes:

- im** : Es el caso peor y el coste de las multiplicaciones es $M(m, im)$.
- $im - i + 1$** : Es el caso mejor y el coste es $M(m, im - i + 1)$.

Sea m el tamaño del problema a . En primer lugar, observemos que el producto de dos enteros de tamaños i y j tiene un tamaño que es al menos $i + j - 1$ y como máximo $i + j$. Sean r_i y m_i el valor y el tamaño de r al principio de la i -ésima pasada por el bucle. Claramente, $r_1 = a$ y, por tanto, $m_1 = m$. Dado que $r_{i+1} = ar_i$, el tamaño de r_{i+1} es al menos $m + m_i - 1$ y como máximo $m + m_i$. La demostración por inducción matemática de que $im - i + 1 \leq m_i \leq im$ para todo i se sigue inmediatamente. Por tanto, la multiplicación efectuada en la i -ésima pasada por el bucle afecta a un entero de tamaño m y a un entero cuyo tamaño se encuentre entre $im - i + 1$ e im , lo cual requiere un tiempo que está entre $M(m, im - i + 1)$ y $M(m, im)$. El tiempo total $T(m, n)$ que se invierte en multiplicaciones cuando se calcula a^n con *exposec* es:

$$\underbrace{\sum_{i=1}^n M(m, im - i + 1)}_{\text{Caso mejor } (\Omega)} \leq T(m, n) \leq \underbrace{\sum_{i=1}^n M(m, im)}_{\text{Caso peor } (O)}$$

donde m es el tamaño de a .

NOTA DEL AUTOR: Hay una pequeña errata en la ecuación anterior en el libro, en el que la cota superior es $\sum_{i=1}^n M(i, im)$ cuando anteriormente estaba puesto $M(m, im)$. Al escribirlo en nuestro resumen la hemos subsanado y escrito correctamente.

Distinguiremos dos casos de los algoritmos clásicos, en la que las multiplicaciones ya *no* son operaciones elementales, como dijimos previamente:

- **Algoritmo sin mejorar** (realiza 4 multiplicaciones): $T(m, n)$ es una buena estimación del tiempo total requerido por *exposec*, puesto que la mayor parte del trabajo se invierte en realizar estas multiplicaciones. Si utilizamos el algoritmo clásico de multiplicación, entonces $M(q, s) \in \theta(qs)$. Sea c tal que $M(q, s) \leq cqs$:

$$T(m, n) \leq \sum_{i=1}^{n-1} M(i, im) \leq \sum_{i=1}^{n-1} c * m * im = cm^2 \sum_{i=1}^{n-1} i < cm^2 n^2$$

Hemos calculado la cota superior. El cálculo para la cota inferior será igual, concluyendo que para el algoritmo sin mejorar es:

$$T(m, n) \in \theta(n^2 m^2)$$

Esta demostración está sacada del libro. Observamos que se ha hecho con el método clásico *sin* mejorar.

- **Algoritmo mejorado** (realiza 3 multiplicaciones): Haremos el mismo cálculo para calcular la cota superior mediante este método mejorado:

$$T(m, n) \leq \sum_{i=1}^{n-1} M(i, im) = \sum_{i=1}^{n-1} \frac{im}{m} m^{\log_2 3} = m^{\log_2 3} \sum_{i=1}^{n-1} i \leq n^2 m^{\log_2 3}$$

Tendremos mediante esta demostración que:

$$T(m, n) \in O(n^2 m^{\log_2 3})$$

Calcularíamos la cota inferior, pero es igual, por tanto, concluimos que:

$$T(m, n) \in \theta(n^2 m^{\log_2 3})$$

Para mejorar *exposec* haremos que $a^n = (a^{(n/2)})^2$, cuando n es par. Se puede calcular $a^{(n/2)}$ aproximadamente cuatro veces más deprisa que a^n con *exposec* y basta una única elevación al cuadrado para obtener el resultado deseado a partir de $a^{(n/2)}$. Tendremos lo siguiente:

$$a^n = (a^{(n/2)}) * (a^{(n/2)}) \quad \text{cuando } n \text{ es } \mathbf{par}.$$

$$a^n = a * a^{n-1} \quad \text{cuando } n \text{ es } \mathbf{impar}. \text{ En la siguiente llamada recursiva la resolveríamos con el algoritmo de arriba, ya que } n \text{ sería ya par.}$$

La recurrencia que produce es:

$$a^n = \begin{cases} a & \text{si } n = 1 \\ (a^{(n/2)})^2 & \text{si } n \text{ es par} \\ a * a^{n-1} & \text{en caso contrario} \end{cases}$$

Un **ejemplo** que tendremos es:

$$a^{29} = a * a^{28} = a * (a^{14})^2 = a * ((a^7)^2)^2 = \dots = a * ((a * (a * a^2)^2)^2)^2$$

Sólo utilizaría 3 multiplicaciones y 4 elevaciones al cuadrado en lugar de las 28 multiplicaciones que se necesitan con *exposec*.

La recursión anterior da lugar a este algoritmo:

```
funcion expoDV (a, n)
  si n = 1 entonces devolver a
  si n es par entonces devolver [expoDV(a, n/2)]^2
  devolver a * expoDV(a, n - 1)
```

El número de multiplicaciones sólo es una función del exponente n , por lo que denotamos $N(n)$. Veremos los casos distintos:

- Cuando $n = 1$, no se efectúa operación, así que $N(1) = 0$.
- Cuando n es **par**, se efectúa una multiplicación (la elevación del cuadrado de $(a, n/2)$, además de las $N(n/2)$ multiplicaciones implicadas en la llamada recursiva a *expoDV* ($a, n/2$).
- Cuando n es **impar**, se efectúa una multiplicación (a por a^{n-1}) además de las $N(n-1)$ multiplicaciones requeridas por la llamada recursiva a *expoDV* ($a, n-1$).

Por tanto, la recurrencia de nuevo en función de $N(n)$ es:

$$N(n) = \begin{cases} 0 & \text{si } n = 1 \\ N(n/2) + 1 & \text{si } n \text{ es par} \\ N(n-1) + 1 & \text{en caso contrario} \end{cases}$$

Acotamos inferior y superiormente la función mediante funciones no decrecientes. De nuevo, tendremos estos casos:

- Cuando $n > 1$ es **impar**:

$$N(n) = N(n-1) + 1 = N\left(\frac{(n-1)}{2}\right) + 2 = N(\lfloor n/2 \rfloor) + 2.$$
- Cuando $n > 1$ es **par**:

$$N(n) = N(\lfloor n/2 \rfloor) + 1.$$

Por tanto, tendremos la siguiente recurrencia a partir de los casos anteriores:

$$\underbrace{N(\lfloor n/2 \rfloor) + 1}_{n \text{ par}} \leq N(n) \leq \underbrace{N(\lfloor n/2 \rfloor) + 2}_{n \text{ impar}}$$

Tendremos las siguientes variables empleando para ello la ecuación anterior:

- a:** Número de llamadas recursivas = 1, que sería 1 si es impar o par.
- b:** Reducción del problema en cada llamada = 2
- c * n^k:** Coste de las operaciones extras a las llamadas recursivas. Al ser una constante la operación extra, tendremos que $k = 0$.

Ahora estaremos en el caso segundo siendo $t(n) \in \theta(n^k * \log(n)) = \theta(\log(n))$.

Las funciones anteriores (empleando **operaciones elementales**) están en orden de $\theta(\log(n))$, por lo que igualmente lo está $N(n)$. En este mismo caso, *exposec* usaba un número de multiplicaciones de tiempo $\theta(n)$ para efectuar la misma exponenciación, por tanto, podemos concluir que *expoDV* es más eficiente que *exposec* usando este criterio. Añadiremos más adelante una tabla comparativa incluyendo este coste también, a modo de recordatorio.

Sea $M(q, s)$ el tiempo necesario para multiplicar 2 enteros de tamaños q y s y $T(m, n)$ el tiempo que invierta en multiplicar una llamada a *expoDV* (a, n), en donde m es el tamaño de a . Tendremos, por tanto, esta recurrencia:

$$T(m, n) \leq \begin{cases} 0 & \text{si } n = 1 \\ T(m, n/2) + M(mn/2, mn/2) & \text{si } n \text{ es } \mathbf{par} \\ T(m, n-1) + M(m, m(n-1)) & \text{si } n \text{ es } \mathbf{impar} \end{cases}$$

Se ha hecho otra pequeña modificación en el caso en el que n es impar, por lo que se varía un parámetro de la multiplicación M , poniendo del modo correcto bajo mi punto de vista.

Las siguientes multiplicaciones M son:

$$M(mn/2, mn/2) = (m * n/2) * (m * n/2).$$

$$M(m, m(n-1)) = m * m^{n-1}$$

Calculamos $T(m, n)$ en el caso peor, teniendo esta ecuación para calcular el tiempo:

$$T(m, n) \leq \underbrace{T(m, \lfloor n/2 \rfloor) + M(m\lfloor n/2 \rfloor, m\lfloor n/2 \rfloor)}_{\text{cuando } n \text{ es par}} + \underbrace{M(m, m(n-1))}_{\text{cuando } n \text{ es impar}}$$

Recordemos que la expresión $\lfloor n/2 \rfloor$ significa tomar la parte inferior de la mitad de n .

Dicho esto, veremos la resolución de esta ecuación puesta arriba empleando los distintos métodos ya conocidos:

1. Utilizando el *método clásico*, que recordemos que realizaba 4 multiplicaciones. Tendremos lo siguiente:

$$M(m\lfloor n/2 \rfloor, m\lfloor n/2 \rfloor) \sim m^2 n^2$$

$$M(m, m(n-1)) \sim m^2 n$$

Sustituyendo en la ecuación anterior nos quedará:

$$T(m, n) \leq T(m, \lfloor n/2 \rfloor) + \mathbf{m^2 n^2}$$

Podremos decir que la parte de la multiplicación de las dos mitades es la que impera en el tiempo total del algoritmo, la cual hemos resaltado en negrita.

Pasamos a resolver la recurrencia de **reducción por división** empleando las siguientes variables:

a: Número de llamadas recursivas = 1

b: Reducción del problema en cada llamada = 2

c * n^k: Coste de las operaciones extras a las llamadas recursivas. $k = 2$, recordemos que había que hacer 4 multiplicaciones, siendo $k = \log_2 4 = 2$. Previamente, hemos visto cual era ese coste.

Como es habitual la resolución de la recurrencia es:

$$T(n) = \begin{cases} \theta(n^k) & \text{si } a < b^k \\ \theta(n^k * \log(n)) & \text{si } a = b^k \\ \theta(n^{\log_b a}) & \text{si } a > b^k \end{cases}$$

Sustituyendo en $a = b^k$, tendremos que $a < b^k$, por lo que el coste es $t(n) \in \theta(n^2)$. Por tanto, $t(m, n) \in \theta(m^2 n^2)$, y además vimos que imperaba este coste en el algoritmo.

Posteriormente veremos una tabla comparativa de costes, aunque podemos decir que como conclusión que el coste del método clásico empleando *exposec* y *expoDV* es el mismo, siendo éste el deducido previamente.

2. Utilizando el *método mejorado*, que recordemos que consistía en hacer 3 multiplicaciones. Igualmente, tendremos:

$$M(m\lfloor n/2 \rfloor, m\lfloor n/2 \rfloor) \sim m^{\log_2 3} n^{\log_2 3}$$

$$M(m, m(n-1)) \sim m^{\log_2 3} n$$

La ecuación sería:

$$T(m, n) \leq T(m, \lfloor n/2 \rfloor) + \mathbf{m^{\log_2 3} n^{\log_2 3}}$$

Recordemos que $\log_2 3 > 1$, por lo que imperará $m^{\log_2 3} n^{\log_2 3}$.

Resolveremos, de nuevo, la recurrencia de **reducción por división** empleando las siguientes variables:

a: Número de llamadas recursivas = 1

b: Reducción del problema en cada llamada = 2

c * n^k: Coste de las operaciones extras a las llamadas recursivas, siendo $k = \log_2 3$, como hemos visto previamente.

Resolviendo $a = b^k$, tendremos de nuevo que $a < b^k$, por lo que el coste es $t(n) \in \theta(n^{\log_2 3})$. Por tanto, $t(m, n) \in \theta(m^{\log_2 3} n^{\log_2 3})$.

Recordemos que el coste de realizar 3 multiplicaciones empleando *exposec* era $\theta(n^2 m^{\log_2 3})$, mientras que empleando el algoritmo *expoDV* es $\theta(m^{\log_2 3} n^{\log_2 3})$, el cual acabamos de hallar. Veremos una tabla comparativa a modo de último recordatorio.

Compararemos los distintos métodos vistos hasta el momento:

	Multiplicación		
	Operaciones elementales	Clásica	D y V
exposec	$\theta(n)$	$\theta(m^2 n^2)$	$\theta(m^{\log_2 3} n^2)$
expoDV	$\theta(\log(n))$	$\theta(m^2 n^2)$	$\theta(m^{\log_2 3} n^{\log_2 3})$

Por último, tendremos una versión iterativa del algoritmo de exponenciación:

```

funcion expoiter (a, n)
  i ← n; r ← 1; x ← a
  mientras i > 0 hacer
    si i es impar entonces r ← rx
    x ← x2
    i ← i ÷ 2
  devolver r

```