

Ejercicios resueltos de programación 3

Tema 7. Divide y vencerás.

De nuevo, haremos la distinción entre estas partes en el documento, por lo que el *índice* es:

1. Introducción teórica	3
2. Cuestiones de exámenes	4
3. Problemas de exámenes solucionados	31
4. Problemas de exámenes sin solución o planteados	58

Introducción teórica:

Vamos a ver una breve introducción teórica sobre este tema, como sigue:

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 }
    { $p_1 \dots p_k$ }  $\leftarrow$  descomposicion (problema)
    para cada  $p_i$  hacer
       $s_i \leftarrow$  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.

1ª parte. Cuestiones de exámenes:

Trataremos de distinguir dos tipos de ejercicios que veremos en esta parte:

- Ejercicios directamente deducidos de la teoría, es decir, las típicas ordenaciones por fusión, ordenaciones rápidas, búsquedas binarias, etc. Se solicitarán ejemplos de este tipo o bien su código.
- Ejercicios **no** directamente deducidos de la teoría, es decir, aquellos en los que piden por ejemplo ordenar votos con coste lineal. Esto implicara usar los esquemas vistos en la teoría, pero de modo más o menos indirecto o no especificado en el enunciado de modo claro.

Hemos tratado de definir los dos tipos, para que así se pueda estudiar los ejercicios, asociándolo a cada uno de ellos.

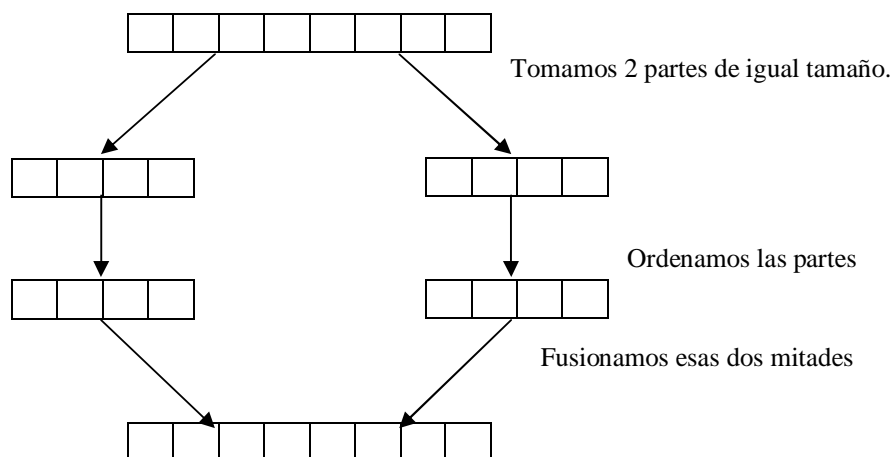
Febrero 2000-1ª (ejercicio 1)

Enunciado: Comparar la eficiencia de los algoritmos de ordenación quicksort (ordenación rápida) y mergesort (ordenación por fusión).

Respuesta: El algoritmo *quicksort* emplea un tiempo promedio de $n * \log(n)$, en el peor caso de n^2 . El algoritmo de ordenación por fusión (*mergesort*) utiliza un tiempo de $n * \log(n)$ (siempre observando la precaución de equilibrar los subcasos a tratar). Pese a esto, en la práctica quicksort es más rápido en un tiempo c constante. Además, el algoritmo *mergesort* requiere de un **espacio extra** para tratar los distintos casos si perder eficiencia (hacer la ordenación *in situ* asociado un incremento de la constante oculta bastante alto).

Como añadido a la solución del problema veremos estos ejemplos, el primero de ellos el del mergesort u ordenación por fusión y el segundo quicksort u ordenación rápida. Pasamos a verlos:

El algoritmo de ordenación por fusión (mergesort) realiza lo siguiente:



El algoritmo de ordenación rápida (quicksort) hace:



Recordemos que según la colocación del pivote tendremos estos casos, que veremos brevemente:

Mejor caso: Pivote exactamente en el centro tras ordenarlo. Coste $\theta(n * \log(n))$.

Peor caso: Pivote en la primera posición tras ordenarlo. Coste $\theta(n^2)$.

Febrero 2000-2ª (ejercicio 1)

Enunciado: Poner un ejemplo en el que un enfoque de Divide y Vencerás nos lleve a un coste exponencial de forma innecesaria.

Respuesta: Un ejemplo claro de esto es utilizar la definición recursiva de la función de Fibonacci sin más, de este modo el siguiente algoritmo conlleva un **coste exponencial**, existiendo formas de obtener la solución en un coste inferior utilizando un enfoque *distinto* al de divide y vencerás:

```
fun fib (n)
  si (n < 2) entonces devolver n
  si no
    devolver fib (n - 1) + fib (n - 2)
  fsi
ffun
```

Septiembre 2000 (ejercicio 2)

Enunciado: La sucesión de Fibonacci se define como $fib(0) = fib(1) = 1$; $fib(n) = fib(n-1) + fib(n-2)$ si $n \geq 2$. ¿Qué ocurre si aplicamos una estrategia divide y vencerás para calcular $fib(n)$ usando directamente la definición? ¿Qué otras opciones existen?

Respuesta: Si utilizamos directamente la definición obtendremos un algoritmo con un **coste exponencial** (coste muy malo, es igual al ejercicio anterior). Dicho algoritmo es el siguiente:

```
fun fib (n)
  si (n < 2) entonces devolver n
  si no
    devolver fib (n - 1) + fib (n - 2)
  fsi
ffun
```

El principal problema es que calcularía varias veces los mismos valores, para evitar esto existe otra opción dada por el siguiente algoritmo iterativo, que requiere un *tiempo lineal*. Es el siguiente:

```
fun fib-iter (n)
  i ← 1; j ← 0
  para k ← 1 hasta n hacer
    j ← i + j
    i ← j - i
  devolver j
```

Septiembre 2000 (ejercicio 3) (parecido a ejercicio 1 de Septiembre 2002-reserva)

Enunciado: ¿Cuál es la relación entre la ordenación por selección y la ordenación por montículo? ¿Cómo se refleja en la eficiencia de los dos algoritmos?

Respuesta: La solución dada por el ejercicio es completamente del autor, por lo que como siempre no sabemos si estará bien o por el contrario hay algo incorrecto.

Nos piden en el enunciado, según entendemos, que nos piden las diferencias entre ambas, así como sus parecidos. Pasamos a definirlos y luego a responder a las preguntas. La **ordenación por selección** es aquella en la que se toma el menor elemento de la parte desordenada y lo intercambia con el mejor que encuentra de tal manera que quede ya ordenado en su posición, mientras que la **ordenación por montículo** es aquella en la que se ordena el vector mediante las propiedades del montículo siguientes (relación de padre e hijo en montículo de máximos):

$$T[i] \geq T[2 * i]$$
$$T[i] \geq T[2 * i + 1]$$

Se parecen en que usan el mismo vector para realizar la ordenación, mientras que se diferencian en que mientras uno emplea las propiedades del montículo teniendo coste $\log(n)$ de flotar, el otro realiza el intercambio de un elemento a otro recorriendo el vector hasta encontrar el menor elemento, lo que significa coste lineal ($O(n)$).

El **coste asintótico**, en el caso peor, del primer algoritmo de ordenación sería cuadrático ($\theta(n^2)$), mientras que la de la ordenación por montículo sería $\theta(n * \log(n))$, por tanto, la segunda ordenación evidentemente es mucho más eficiente. Aun así, merecería la pena remontarse a los apuntes de estructuras de datos y recordarlos. Trataremos de verlos a continuación.

Además de poner esto, vamos a ampliar más conocimientos tomando apuntes del libro de *estructuras de datos y algoritmos* de R. Hernández. Empezaríamos a ver la ordenación por selección y luego la ordenación por montículo, como sigue:

La **ordenación por selección** directa es aquella cuya idea es seleccionar el menor elemento de una parte desordenada y colocarlo en la posición del primer elemento no ordenado. En un primer paso, se recorre el arreglo hasta encontrar el elemento menor. Para ello, se coloca el primer elemento en una variable temporal y se va comparando con los demás elementos del arreglo tal que si se encuentra uno menor se asigna a la variable temporal. Recorrido todo el arreglo, el elemento de la variable temporal (que será el menor) se intercambia con el de la primera posición. Seguidamente, se considera únicamente la parte del arreglo no ordenado y se repite el proceso de búsqueda el menor, y así sucesivamente. En conclusión, el algoritmo puede describirse de la siguiente forma:

- Seleccionar el elemento menor de la parte del arreglo no ordenada.
- Colocarlo en la primera posición de la parte no ordenada del arreglo.

El pseudocódigo del algoritmo (del libro de Brassard, para ello estudiamos de nuevo el tema 4 de nuestro resumen) es el siguiente:

```

procedimiento seleccionar( $T(1..n)$ )
  para  $i \leftarrow 1$  hasta  $n - 1$  hacer
     $minj \leftarrow 1$ ;  $minx \leftarrow T[i]$ ;
    para  $j \leftarrow i + 1$  hasta  $n$  hacer
      si  $T[j] < minx$  entonces
         $minj \leftarrow j$ ;
         $minx \leftarrow T[j]$ ;
      fsi
    fpara
     $T[minj] \leftarrow T[i]$ ;  $T[i] \leftarrow minx$ ;
  fpara
fprocedimiento
  
```

Veremos un **ejemplo** que corresponde con este algoritmo para así verlo más claro, aunque en el ejercicio de Septiembre de 2002 sí que nos piden que pongamos el ejemplo. Aun así, lo pondremos:

El vector a ordenar


8	14	5	9	3	23	17
---	----	---	---	---	----	----

Seleccionamos el menor de la parte desordenada,
que sería el elemento 3

<u>8</u>	14	5	9	<u>3</u>	23	17
----------	----	---	---	----------	----	----

Se intercambian los dos elementos,
y ya queda ordenado el elemento con valor 3, resaltado en negrita

3	14	5	9	8	23	17
----------	----	---	---	---	----	----




Se seleccionamos el siguiente menor de la parte desordenada,
que sería el elemento 5

3	<u>14</u>	<u>5</u>	9	8	23	17
----------	-----------	----------	---	---	----	----

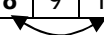
Se intercambian los dos elementos de nuevo,
y ya queda ordenado el elemento con valor 5, resaltado en negrita

3	5	14	9	8	23	17
----------	----------	----	---	---	----	----




Continuamos con el procedimiento, intercambiando este

3	5	8	9	14	23	17
----------	----------	----------	---	----	----	----




Continuamos con el procedimiento, al no haber elemento menor,
se intercambiaría con él mismo

3	5	8	9	14	23	17
----------	----------	----------	----------	----	----	----




Continuamos con el procedimiento, al no haber elemento menor,
se intercambiaría con él mismo

3	5	8	9	14	23	17
---	---	---	---	----	----	----



Por último, haremos este intercambio y ya queda ordenado el vector,
estando el último ya ordenado

3	5	8	9	14	17	23
---	---	---	---	----	----	----



En cuanto a la **ordenación por montículo** decir que una vez organizado el arreglo como un montón, la clasificación (ordenación) del mismo se obtiene teniendo en cuenta que la cima, el elemento h_1 , es el menor de todos los elementos del montón. Así para clasificar u ordenar un arreglo de n elementos se retira de la cima y se construye un montón con los demás $n - 1$ elementos. Su cima será de nuevo el menor de los que quedan, se retira y vuelve a construir un montón con los $n - 2$ restantes, y así sucesivamente. Una cuestión es donde almacenar las sucesivas cimas, para ello la solución adecuada consiste en intercambiar la cima, primer elemento del arreglo y construir un montón con los $n - 1$ elementos. Esta ampliación está sacada del libro de *estructuras de datos*, aunque no es literal del todo.

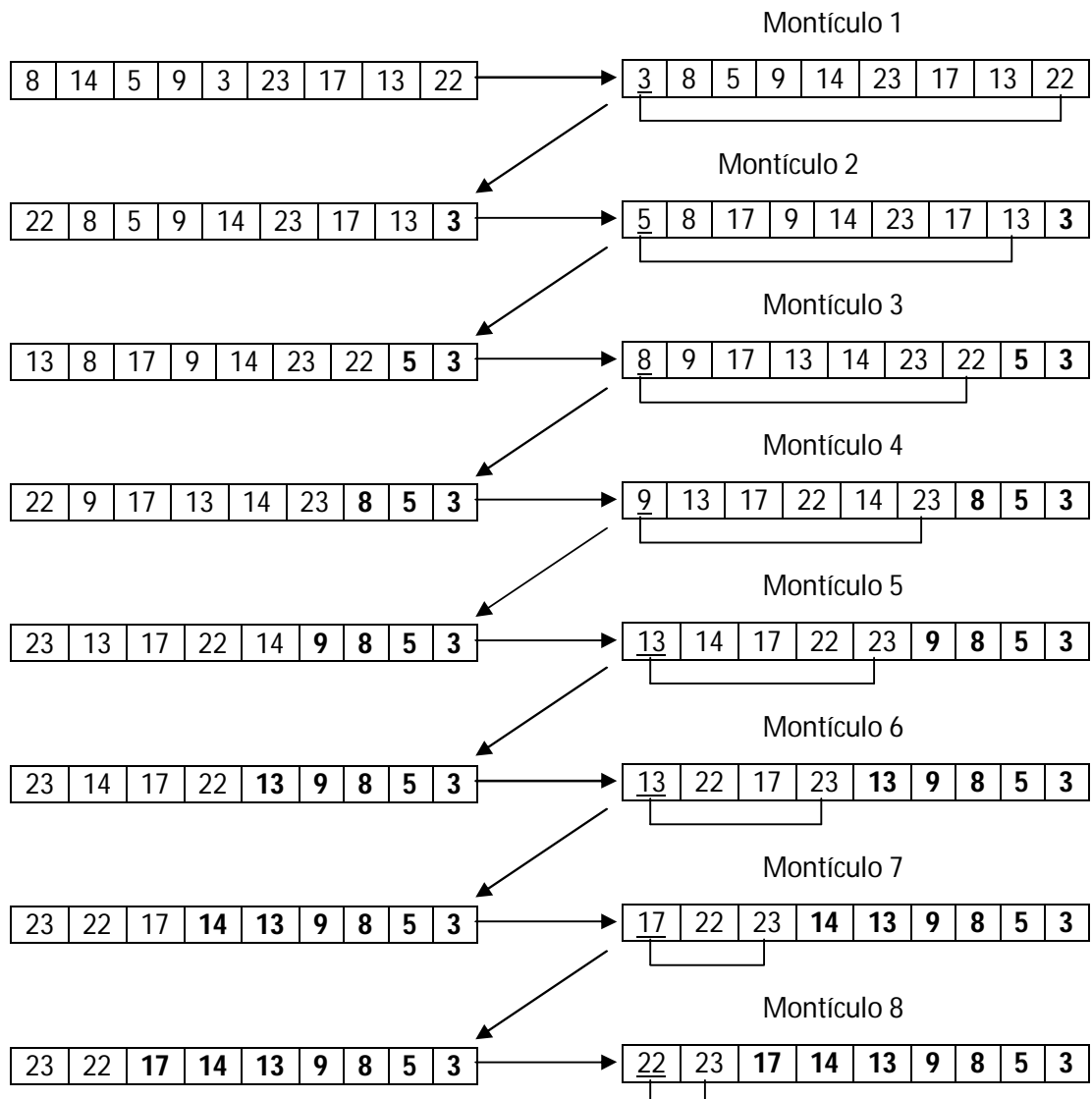
Para ello, veremos el algoritmo en pseudocódigo (visto ya en el tema 5 de nuestro resumen, de estructura de datos):

```

procedimiento ordenación por montículo ( $T[1..n]$ )
{ T es la matriz que hay que ordenar }
crear-montículo (T);
para  $i \leftarrow n$  bajando hasta 2 hacer
    intercambiar  $T[1]$  y  $T[i]$ 
    hundir ( $T[1..i - 1], 1$ )

```


Igualmente, veremos un **ejemplo** sacado del libro de *estructuras de datos y algoritmos*, como sigue:



Por tanto, tras el último montículo ya quedaría el vector ordenado, como sigue :

23	22	17	14	13	9	8	5	3
----	----	----	----	----	---	---	---	---

Por último, se podría hacer una ordenación *in situ* del vector. Pero eso ya no lo haríamos, ya que nuestro código correspondería con el anterior ejemplo. En el libro donde hemos sacado estos ejemplos estará esta última ordenación, que dejaríamos sin verlo (aunque se puede ver en el tema 5 de nuestro resumen ejemplos similares).

Septiembre 2000-reserva (ejercicio 1)

Enunciado: En el algoritmo quicksort (ordenación rápida), ¿qué implicaciones tiene para la eficiencia el escoger como pivote la mediana exacta del vector?

Respuesta:

El **algoritmo quicksort** tiene un tiempo promedio de $O(n * \log(n))$. En la práctica es más rápido que la ordenación por montículo (heapsort) y la ordenación por fusión (mergesort) en un tiempo constante.

Al ser un algoritmo recursivo sería ideal poder dividir el caso en subcasos de tamaños similares de manera que cada elemento sea de profundidad $\log(n)$, es decir, poder equilibrar los subcasos. Sin embargo, aunque seleccionemos la mediana del vector en el peor caso (todos los elementos a ordenar son iguales) el orden será cuadrático.

Para obtener alguna ventaja utilizando como pivote la mediana tendremos que hacer algunas modificaciones al algoritmo original. Utilizaremos una nueva función pivote (recordemos que se llamaba *pivotebis*) que divida en 3 secciones el vector, de modo que dado un pivote p una parte conste de elementos menores que él, otra de elementos iguales y otra con los más grandes que p . Tras hacer esto se harían las llamadas recursivas correspondientes al algoritmo, una para el subvector de elementos menores que p , y otra para el de los mayores que p . Esta modificación consigue que, utilizando la **mediana** como pivote el orden del algoritmo sea $O(n * \log(n))$ incluso en el peor caso. Pero el coste que implica esta modificación hace que el tiempo constante crezca haciendo mejor el algoritmo de ordenación por montículo que el quicksort en todos los casos, lo que hace que la modificación no sea factible.

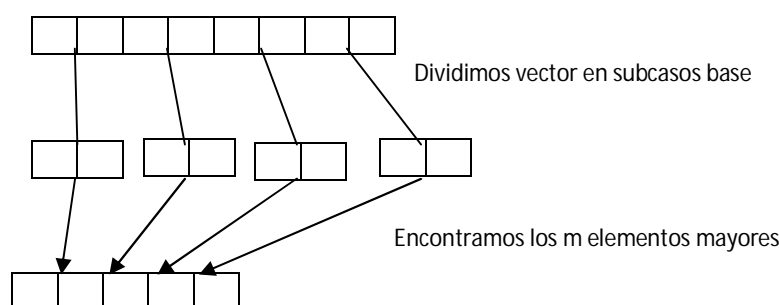
Febrero 2001-1ª (ejercicio 3)

Enunciado: Un vector T contiene n elementos. Se quieren encontrar los m mayores elementos del vector y se supone $n \gg m$ (n mucho mayor que m). Describe una forma eficiente de hacer esto sin ordenar el vector y calcula qué coste tiene.

Respuesta: Podemos hacer un enfoque similar a como hacemos en la **ordenación por fusión**, pero en este caso iremos generando un subvector con aquellos elementos mayores que m . Dividimos, por tanto, el vector original de forma equilibrada y progresivamente, hasta llegar a un tamaño base adecuado (lo suficientemente pequeño) para aplicar en él la búsqueda de todo elemento mayor que m . Cada vez que encontremos un elemento mayor que m lo almacenaremos en un vector resultado, que posteriormente se fusionará con los encontrados en las distintas llamadas. Igualmente podemos utilizar otro algoritmo de Ordenación pero sin ordenar el vector, si no almacenar en otro vector los índices en el orden correspondiente, por ejemplo.

NOTA DEL AUTOR: Se ha hecho una modificación al resultado de este ejercicio debido a que nos solicitan que encontremos los m elementos mayores. Por ello, al encontrar el elemento mayor lo iremos almacenando en el vector solución. En la solución dada del ejercicio (no sabemos si es oficial o no) está justamente al revés, por lo que hemos modificado esta parte en nuestra solución. Además, este ejercicio es un ejemplo del segundo tipo dado al inicio de los ejercicios.

Como añadido al ejercicio, **gráficamente** sería algo así:



Febrero 2002-2ª (ejercicio 1)

Enunciado: Poner un ejemplo de un vector de 10 elementos que suponga un ejemplo de caso peor para el algoritmo de ordenación Quicksort. Ordenar el vector mediante este algoritmo detallando cada uno de los pasos.

Respuesta: Recordemos que el caso peor de este algoritmo es aquél en el que tras poner el **pivot** en su sitio correcto es hay que ordenarlo salvo un elemento, que sería el ya ordenado. El caso peor es aquel en el que el vector está ordenado de modo creciente, como sigue:

1	2	3	4	5	6	7	8	9	10
---	---	---	---	---	---	---	---	---	----

Recordemos los procedimientos de este algoritmo:

procedimiento $\text{pivot}(T[i..j], \text{var } l)$

{ 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 propiamente dicho de ordenación rápida (*quicksort*) es:

procedimiento $\text{quicksort}(T[i..j])$

{ Ordena la submatriz $T[i..j]$ por orden no decreciente }

si $j - i$ es suficientemente pequeño entonces

insertar ($T[i..j]$)

si no

$\text{pivot}(T[i..j], l)$

$\text{quicksort}(T[i..l - 1])$

$\text{quicksort}(T[l + 1..j])$

Seguiremos los pasos como sigue:

1^{er} paso: Hará lo siguiente:

Tenemos que el vector inicial es $T[1..10] = \{1,2,3,4,5,6,7,8,9,10\}$, por lo que la llamada inicial a pivote es pivote ($T[1..10]$, L).

Tendremos, por tanto, que:

$$i = 1, j = 10, p = 1, k = 1, L = 11$$

Con estos valores seguiremos este procedimiento:

- El puntero k , se mueve hasta el valor de 2, debido a que $T[2] > p$.
- L llega hasta 1, por ser $T[1] \leq p$.
- Al superar el puntero k a L no tendremos que hacer ningún intercambio, excepto en la última instrucción intercambiar $T[i]$ por $T[L]$, como L apunta a 1, e i es 1, $T[L]$ queda igual.

2^o paso: A continuación haríamos una llamada a quicksort ($T[1..0]$), siendo la llamada a un caso de tamaño cero. Además, haríamos otra llamada a quicksort ($T[2..10]$), siendo éste el resto del vector.

Ignoramos la primera llamada al ser de un tamaño 0, mientras que la otra tendría un efecto similar al del paso anterior, siguiendo este procedimiento:

- En este caso en el procedimiento pivote L descendería hasta llegar a $T[2]$ y k quedaría en el tercer elemento.
- Tendremos exactamente lo mismo que antes con los punteros, por lo que no se provoca cambio ninguno.

3^{er} paso: De nuevo volvemos a hacer una tercera llamada a quicksort, una de ellas para un caso de tamaño 0, que sería quicksort ($T[3..2]$) y otra para el resto del vector ($T[3..10]$).

El proceso se repetirá hasta llegar al último elemento, produciéndose tantas llamadas como elementos posea el vector.

NOTA DEL AUTOR: Nos fijamos que están mal distribuidos los vectores que hay que ordenar, por ello sería el algoritmo peor, como hemos visto en la teoría de este capítulo. Además, esta solución está sacada de la respuesta dada en el examen, aunque desconocemos si es oficial (u "oficiosa").

Febrero 2002-2ª (ejercicio 1)

Enunciado: Escribe la secuencia completa de pasos para ordenar por fusión el vector [3,2,7,5,9,3,4,8].

Respuesta: Esta solución está hecha por el autor, aunque es simplemente seguir el procedimiento dado en la teoría y hacer el dibujo respectivo a cada paso. Por tanto, veremos los pasos y los explicaremos:

Tenemos el vector:

3	2	7	5	9	3	4	8
---	---	---	---	---	---	---	---

1º paso: Dividimos en dos partes el vector, que nos damos cuenta que tiene longitud par. Por tanto, tendremos:

3	2	7	5
9	3	4	8

De momento, no hemos llegado al caso suficientemente simple, por lo que seguiremos dividiendo hasta encontrarlo.

2º paso: Dividiremos la parte izquierda en dos partes, de nuevo:

3	2
7	5

3º paso: Al ser suficientemente simples (tienen dos elementos) ambas partes podremos ordenarlas de modo creciente, quedando como sigue:

2	3
5	7

4º paso: Fusionaremos estas dos partes y tendremos:

2	3	5	7
---	---	---	---

En este caso, ya está ordenada la parte izquierda, por lo que a continuación veríamos la parte derecha del vector.

5º paso: Seguiríamos con el mismo procedimiento que antes y, por tanto, dividimos la parte izquierda:

9	3
4	8

6º paso: Los ordenamos quedándonos lo siguiente:

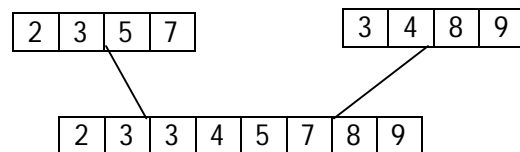
3	9
4	8

7º paso: Fusionaremos ambas partes:

3	4	8	9
---	---	---	---

A diferencia de la otra fusión, en este caso tendremos que hacer una comparación de ambos punteros, que recordemos de la teoría eran la i y j para luego fusionarlos en el otro vector.

8º paso: Por último, fusionaremos ambas mitades ordenadas:



Estando ya ordenado el vector correctamente.

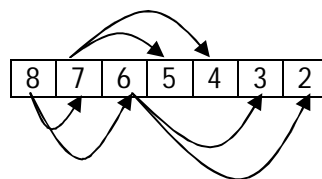
Hemos omitido algunos pasos, en los que se compararían ambos punteros y, por tanto, se vería como se incrementa el puntero en ambas mitades, así como en el vector solución, aunque eso ya lo hemos estudiado en la teoría de la asignatura y en ejercicios anteriores. Con estos pasos, creo que vale para solucionar este ejercicio.

Septiembre 2002 (ejercicio 1)

Enunciado: En el algoritmo de ordenación por montículo (heapsort). ¿Cuáles son las mejores y peores disposiciones iniciales de los elementos que hay que ordenar en cuanto al tiempo de ejecución? Razonar la respuesta y poner ejemplos.

Respuesta: Esta solución está dada igualmente que la anterior por el autor.

El caso mejor del algoritmo de ordenación por montículo (de máximos) es aquél en el que todos los elementos del vector cumple las **propiedades del montículo**, que recordemos eran $T[i] \geq T[2 * i]$ y $T[i] \geq T[2 * i + 1]$. Es decir, ordenado de modo *no* natural (por orden decreciente). Para verlo más claro, pondremos un ejemplo:



En este caso, por tanto, no haremos ningún intercambio, ya que estarían ordenados siguiendo las propiedades del montículo. Por tanto, en el **caso mejor** el coste es $\theta(n)$, en el que sólo recorre los elementos del montículo y lo compara sin intercambiarlo.

El peor caso será justamente lo contrario, es decir, aquél en el que los elementos estén ordenados de modo creciente, como puede ser:

2	3	4	5	6	7	8
---	---	---	---	---	---	---

Un ejemplo de ordenación por montículo lo hemos visto previamente, al igual que el pseudocódigo que emplearemos para ordenarlo. Por ello, no lo ordenaremos, dejándolo como ejercicio extra (aunque fácil, una vez visto el ejercicio anterior).

En este caso, en el caso peor el coste es $\theta(n * \log(n))$.

Septiembre 2002 (ejercicio 3)

Enunciado: De los algoritmos de ordenación que has estudiado. ¿Cuál es el más eficiente en términos de coste asintótico temporal en el caso peor? Razonar la respuesta.

Respuesta: Esta solución es mía personal. Hemos estudiado por el momento estos algoritmos de ordenación (aunque hay más), que es el algoritmo de ordenación por fusión (*mergesort*), ordenación rápida (*quicksort*), ordenación por montículo (*heapsort*), ordenación por inserción. Estos dos primeros algoritmos tienen coste en el caso peor cuadrático (recordemos que era por estar mal distribuida las partes), aunque el tercer algoritmo tiene coste $O(n * \log(n))$ en el peor caso. El último de ellos, tendrá coste cuadrático. Diríamos, por tanto, aunque no es seguro que esté correcta la solución particular dada.

Septiembre 2002-reserva (ejercicio 2)

Enunciado: Cuando un cartero reparte cartas en los buzones de un edificio, lo hace en un tiempo lineal, suponiendo que cada carta se asigna a un buzón en tiempo constante. Explicar, a partir del comportamiento del cartero, cómo pueden ordenarse n ejemplares de números entre 1 y m , siendo m una cantidad conocida y fija, en un tiempo lineal. ¿Por qué decimos entonces que el algoritmo quicksort es muy eficiente, si tiene un coste promedio de $O(n * \log(n))$?

Respuesta: Este ejercicio se parece mucho al 1 de Febrero de 2003-2ª semana, que más adelante veremos, sólo que pondremos este enunciado por seguir el orden cronológico de ejercicios. En él, se nos pide que ordenemos un vector de unos elementos, al igual que pasa con este ejercicio. Para ello, tendremos que seguir estos pasos:

1. El vector está inicializado a 0 ($O(n)$).
2. Recorremos la lista de valores a ordenar ($O(n)$).
 - a. Por cada valor i extraído de la lista, incrementamos el valor de la posición y del vector.
3. Recorremos el vector generado mostrando los valores generados en el paso anterior (2) y omitiendo aquellas posiciones del vector que contengan un 0.

Además de lo dicho en este ejercicio, la importancia de escribirlo por separado es que la última pregunta es interesante, debido a que nos preguntan sobre el algoritmo quicksort. Esta pregunta realmente no la acabo de comprender, ya que el algoritmo quicksort (ordenación rápida) es igual de eficiente que puede ser el mergesort (ordenación por fusión), siendo ambos costes iguales en el caso mejor. Por ello, no entiendo la pregunta a que se refiere, e incluso según la teoría de Brassard nos comentan que a nivel de espacio requiere más que la ordenación por fusión, aunque ello requiere rebajar la constante multiplicativa.

Diciembre 2002 (ejercicio 3)

Enunciado: Ordenar ascendentemente mediante el algoritmo de quicksort el vector $V \rightarrow [2,3,8,1,9,4,2,2,6,5,4,3,7,4]$ detallando todos los pasos.

Respuesta: Tenemos el siguiente vector:

2	3	8	1	9	4	2	2	6	5	4	3	7	4
---	---	---	---	---	---	---	---	---	---	---	---	---	---

Tomamos el pivote como primer elemento, para luego buscar a la derecha el primer elemento mayor (subrayado) y a la izquierda el menor o igual que el pivote (superrayado) e

intercambiamos. Seguimos haciendo esto hasta que los punteros se crucen. Tendremos, por tanto:

2	3	8	1	9	4	2	2	6	5	4	3	7	4
---	---	---	---	---	---	---	---	---	---	---	---	---	---

Pondremos los pasos que requieran de intercambio, ya que así haremos el ejercicio algo más corto, por no hacer pasos innecesarios. Por tanto, tendremos:

2	<u>3</u>	8	1	9	4	2	<u>2</u>	6	5	4	3	7	4
---	----------	---	---	---	---	---	----------	---	---	---	---	---	---

2	2	<u>8</u>	1	9	4	<u>2</u>	3	6	5	4	3	7	4
---	---	----------	---	---	---	----------	---	---	---	---	---	---	---

2	2	2	<u>1</u>	9	4	8	3	6	5	4	3	7	4
---	---	---	----------	---	---	---	---	---	---	---	---	---	---

Tras este último intercambio decir que ya ambos punteros se han cruzado, por lo que se puede intercambiar con el pivote previamente puesto y así ordenar las subpartes que lo separa el pivote. Por ello, tendremos tras este primer paso lo siguiente:

1	2	2	<u>2</u>	9	4	8	3	6	5	4	3	7	4
---	---	---	----------	---	---	---	---	---	---	---	---	---	---

Como anotación de este ejercicio, decir que misteriosamente y curiosamente el realizado con solución al final tras esos intercambios no se queda ordenado el vector. Por ello, lo haremos de modo correcto en nuestra solución.

Partimos, por tanto, por la parte izquierda, aunque si escogiéramos el pivote como el elemento número 1, al momento veríamos que esta parte está ya ordenada. Por tanto, seguiremos por la parte derecha, quedando:

9	4	8	3	6	5	4	3	7	<u>4</u>
----------	---	---	---	---	---	---	---	---	----------

En este caso el pivote sería el elemento mayor, por lo que lo podríamos poner a la derecha, intercambiándolo con el 4 superrayado, estando ya este último elemento ordenado. Tendremos:

4	4	8	3	6	5	4	3	7	9
---	---	---	---	---	---	---	---	---	----------

De nuevo, ordenaremos la parte izquierda, salvo el valor 9:

4	<u>4</u>	8	3	6	5	4	<u>3</u>	7
----------	----------	---	---	---	---	---	----------	---

Como ya controlamos el procedimiento, realizaremos los intercambios sucesivos como sigue:

4	3	<u>8</u>	<u>3</u>	6	5	4	4	7
----------	---	----------	----------	---	---	---	---	---

4	3	<u>3</u>	8	6	5	4	4	7
----------	---	----------	---	---	---	---	---	---

3	3	4	8	6	5	4	4	7
---	---	----------	---	---	---	---	---	---

De nuevo, la parte izquierda ya está ordenada, así que continuamos por la derecha:

8	6	5	4	4	7
---	---	---	---	---	---

No podremos intercambiar ningún elemento, así que nos quedará:

7	6	5	4	4	8
---	---	---	---	---	---

De nuevo, la parte izquierda es:

7	6	5	4	4
---	---	---	---	---

No podremos intercambiar con ningún elemento mayor, como antes, así que tendremos:

4	6	5	4	7
---	---	---	---	---

Otra vez, la parte izquierda será:

4	<u>6</u>	5	4
---	----------	---	---

4	4	5	6
---	---	---	---

Observamos, sin extendernos más, que ya lo que queda está ordenado, por lo que el vector completo ordenado es:

1	2	2	2	3	3	4	4	4	5	6	7	8	9
---	---	---	---	---	---	---	---	---	---	---	---	---	---

NOTA DEL AUTOR: Siento si este ejercicio queda algo liado, pero no sé hacerlo de otra manera, en la que quede más claro. Se me ocurre que puede ser la mejor.

Febrero 2003-1ª (ejercicio 3)

Enunciado: Se desea implementar una función para *desencriptar* un mensaje numérico. La función *desencriptar* recibe tres enteros: el mensaje cifrado c , la clave privada s y la clave publica z ; y devuelve el mensaje original a . El mensaje original se recompone con la fórmula:

$$a := c^s \bmod z$$

Sabiendo que no se dispone del operador de potencia, implementar la función utilizando el esquema de divide y vencerás.

Respuesta:

Como hemos dicho en la teoría, este apartado no lo vimos, por lo que es el momento adecuado para ello. Tendremos dos modos de hacerlo:

- a) El primero de ellos es en el que nos dan en el enunciado la función de *desencriptar*, que toma 3 parámetros, lo que sería:

```
fun desencriptar (c, s, z: entero)
    devolver expoDV (c, s) mod z
ffun
```

Para calcularlo usando el esquema de divide y vencerás usaremos el algoritmo *expoDV*, visto anteriormente:

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

- b) Otra manera de hacerlo es empleando para ello *expomod*, que es una modificación del dado anterior *expoiter*:

```
funcion expomod (c, s, z: entero)
    { Calcula  $c^s \bmod z$  }
     $i \leftarrow s; r \leftarrow 1; x \leftarrow c$ 
    mientras  $i > 0$  hacer
        si i es impar entonces  $r \leftarrow c \bmod z$ 
         $x \leftarrow x^2 \bmod z$ 
         $i \leftarrow i \div 2$ 
    devolver r
```

NOTA DEL AUTOR: Entiendo que en la solución aportada de este ejercicio se nos dan dos partes, pero no me acaba de quedar claro realmente cual es la diferencia entre dos. Por ello, asumo que son dos maneras distintas de hacerlo, una de modo recursivo (*expoDV*) y otra de modo iterativo (*expomod*).

Febrero 2003-2ª (ejercicio 1) (igual a ejercicio 2 de Septiembre 2006-reserva y a ejercicio 2 de Diciembre 06)

Enunciado: Explica cómo pueden ordenarse n valores enteros positivos en tiempo lineal, sabiendo que el rango de valores es limitado. Explica las ventajas e inconvenientes de este método.

Respuesta: Suponiendo que el rango de los valores a ordenar es **limitado**, es decir, sabemos que, por ejemplo, el rango de números a ordenar va de 0 a 2000, podemos generar un vector de 2000 elementos del tipo entero que almacenara el número de ocurrencias de cada valor de la lista a ordenar en la posición del vector correspondiente. Seguiremos estos pasos:

1. El vector está inicializado a 0 ($O(n)$).
2. Recorremos la lista de valores a ordenar ($O(n)$).
 - a. Por cada valor i extraído de la lista, incrementamos el valor de la posición y del vector.
3. Recorremos el vector generado mostrando los valores generados en el paso anterior (2) y omitiendo aquellas posiciones del vector que contengan un 0.

Septiembre 2003 (ejercicio 2)

Enunciado: Una matriz T contiene n elementos. Se pide encontrar los m elementos más pequeños de T (con $m \ll n$). Explicar cómo hacer esto de la manera más eficiente.

Respuesta:

Este ejercicio se ha sacado del libro *Estructuras de datos y métodos algorítmicos. Ejercicios resueltos* de Narciso Martí, Y. Ortega, J.A. Verdejo, es el número 11.11, además de parecerse bastante al ejercicio 3 de Febrero 2001-1ª semana. Pasamos a verlo:

Un procedimiento parcial de ordenación por selección nos da los m elementos más pequeños con coste $O(m * n)$. Una ordenación eficiente, sin embargo, lo haría en $O(n * \log(n))$. Si $m \approx n$, el coste vendría a ser cuadrático, lo cual nos haría desechar el procedimiento de selección, sin embargo, el orden $O(m * n)$ se puede considerar lineal y, en este caso, el algoritmo de selección puede ser más eficiente.

Tendríamos estos métodos:

- El **primer método** es: Ordenar $V[1..n]$ y coger los m primeros elementos. Tiene coste $\theta(n^2)$.
- El **segundo método** es: Utilizar un algoritmo de coste lineal para la selección. Tiene coste $\theta(m * n)$.
- El **tercer método** es:
 1. Mediante una llamada a selección2 ($V, 1, n, m, v$) devuelve el m -ésimo elemento menor de V .
 2. Llamamos a partición ($V, 1, n, v, i, j$), en la que se particionan los m -ésimos menores elementos para guardarlos en V .
 3. Devolvemos $V[1..m]$.

En este método, el coste es $\theta(n)$.

Evidentemente, tendremos que el tercer método, como vimos en el ejercicio anterior es el adecuado y el más eficiente para nuestro problema.

NOTA DEL AUTOR: Creo que no ha quedado claro este ejercicio al ser una copia literal del ejercicio, aunque se ha intentado. Sobre todo lo interesante es ver los métodos y estudiarlos.

Septiembre 2003-reserva (ejercicio 1)

Enunciado: Ordenar completamente el vector en orden no decreciente por el método quicksort indicando claramente cuál es el contenido del vector en cada paso.

3	4	5	1	3	7	6	2
---	---	---	---	---	---	---	---

Respuesta:

Este ejercicio está resuelto por el autor completamente. Ya hemos visto un ejemplo similar a éste, por lo que lo que haremos será ordenar el vector indicando que se intercambia. Lo curioso de este ejercicio es que piden orden no decreciente, es decir, orden creciente (es rizar el rizo). Por tanto, veremos los pasos:

3	4	5	1	3	7	6	2
---	---	---	---	---	---	---	---

3	<u>4</u>	5	1	3	7	6	<u>2</u>
---	----------	---	---	---	---	---	----------

3	2	<u>5</u>	1	<u>3</u>	7	6	4
---	---	----------	---	----------	---	---	---

3	2	3	<u>1</u>	5	7	6	4
---	---	---	----------	---	---	---	---

1	2	3	3	5	7	6	4
---	---	---	---	---	---	---	---

1	2	3	3	5	<u>7</u>	6	<u>4</u>
---	---	---	---	---	----------	---	----------

1	2	3	3	5	4	6	7
---	---	---	---	---	---	---	---

1	2	3	3	4	5	6	7
---	---	---	---	---	---	---	---

El último intercambio ya sería el que ordenara completamente el vector, por lo que quedaría así:

1	2	3	3	4	5	6	7
---	---	---	---	---	---	---	---

Septiembre 2003-reserva (ejercicio 3)

Enunciado: ¿Se puede aplicar el procedimiento de búsqueda binaria sobre árboles con estructura de montículo? Razonar la respuesta.

Respuesta: Esta solución es del autor personalmente. La respuesta sería que **no**, porque el vector al tener estructura de montículo no está ordenado de modo creciente (aunque el montículo sea de mínimos), siendo una de las condiciones para realizar la búsqueda binaria. Al ser montículo puede haber hijos que sean iguales que los padres.

Un **ejemplo** será:

[3,4,3, ...]

↑ No ordenado

Diciembre 2003 (ejercicio 3)

Enunciado: ¿Cuáles son los casos mejor y peor para el algoritmo de ordenación rápida (quicksort)? ¿Cuál es el orden de complejidad en cada uno de ellos? Razona tu respuesta.

Respuesta: Este ejercicio lo hemos visto previamente en la teoría, por lo que lo recordaremos a continuación:

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:

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

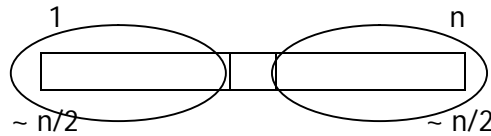
Por tanto, vemos que $a = 1$, por lo que estaremos en el segundo caso. Pasamos a resolverlo, siendo el tiempo $t(n) \in \theta(n^{k+1}) = \theta(n^2)$.

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.

Febrero 2004-1ª (ejercicio 1)

Enunciado: Programar una función "potencia (n, m)" que halle n^m -se supone que no existe la operación potencia y que el coste de una operación de multiplicación es $O(1)$ - mediante Divide y Vencerás con coste $\theta(\log(n))$.

Respuesta: La solución dada está hecho por un alumno, que es el autor. La función que nos piden programar no es ni más ni menos que la *expoDV* que la tenemos en el libro de Brassard, página 276 y en el resumen de la asignatura. Insistimos en la importancia de comprender bien estos códigos, ya que este ejercicio se parece mucho a la cuestión 3 de Febrero de 2008, que veremos más adelante con detenimiento. El algoritmo será, por tanto:

funcion potencia (n, m) dev entero

si $m = 1$ entonces devolver n

si m es par entonces devolver $[potencia(n, m/2)]^2$

devolver $a * potencia(n, m - 1)$

Una curiosidad es que en la teoría, que es de donde hemos sacado este algoritmo no ponemos que devuelva ningún valor, no obstante, sí que lo devuelve, como se puede ver. Esto es otra pequeña característica que habría que tener en cuenta en los códigos, pero que a veces se salta.

Vimos en la teoría que la ecuación de recurrencia sería:

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

Las distintas variables serían:

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 $k = 0$.

La resolución de la **recursividad por división** es:

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

Por tanto, deducimos que el coste al sustituir los valores en $a = b^k$ es $1 = 2^0$, siendo el caso el segundo. Por tanto, de nuevo el coste $t(n) \in \theta(n^0 * \log(n)) = \theta(\log(n))$, tal y como nos piden en el enunciado. Nos fijamos, además, que las operaciones son elementales, por ello, todas las multiplicaciones tienen coste constante, si no sería otro coste totalmente distinto, para eso habría que remontarse a la teoría.

Septiembre 2005 (ejercicio 1)

Enunciado: Suponga que N personas numeradas de 1 a N deben elegir por votación a una entre ellas. Sea V un vector en el que la componente $V[i]$ contiene el número de candidato que ha elegido el votante i. ¿Qué algoritmo utilizarías para determinar si una persona ha obtenido más de la mitad de los votos?

Respuesta: Podría utilizarse una estrategia similar a la que emplea para la ordenación el algoritmo de la *casilla* (página 80 del libro de Brassard), pero trasladado al conteo de elementos. La siguiente función almacena en el vector *votos* el número de votos que va sumando cada candidato votado. Devuelve un valor TRUE si hay alguno con mayoría indicando de qué candidato se trata:

```
funcion contar_votos (V[1..n]) dev (boolean, elegido)
  votos[1..n] = 0;
  para i = 1 hasta N hacer
    votos[V[i]] = votos[V[i]] + 1;
    si votos[V[i]] > N/2 entonces devolver (TRUE, V[i]) fsi
  fpara
  devolver (FALSE, 0);
```

Observamos que tendremos dos vectores, uno donde almacenan los votos (V) y otro donde los cuentan (votos). Este ejercicio es parecido a aquellos de los "bombones pesados" y similar, por lo que lo veremos en la sección de problemas resueltos más ampliados. Aun así, como es habitual este algoritmo sirve para estos ejercicios.

Septiembre 2006 (ejercicio 3)

Enunciado: Dibuja como evolucionaría el siguiente vector al ordenarlo mediante el algoritmo de ordenación rápida (quicksort). Indica únicamente cada una de las modificaciones que sufriría el vector.

6	5	1	2	3	4	7	8	9
---	---	---	---	---	---	---	---	---

Respuesta:

Hemos visto numerosos ejercicios de este tipo, por tanto, omitiremos dar más detalles y literalmente copiaremos la solución dada. Nuestro criterio es tomar como **pivote** el *primer elemento*.

6	5	1	2	3	4	7	8	9
----------	---	---	---	---	---	---	---	---

4	5	1	2	3	6	7	8	9
---	---	---	---	---	---	---	---	---

4	3	1	2	5	6	7	8	9
---	---	---	---	---	---	---	---	---

2	3	1	4	5	6	7	8	9
---	---	---	---	---	---	---	---	---

2	1	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---	---

1	2	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---	---

Ahora nuestro criterio es tomar como **pivote** el elemento que ocupa la *posición central*, que en este caso el primer elemento es el elemento que ocupa la posición quinta, que es el 3:

6	5	1	2	3	4	7	8	9
---	---	---	---	----------	---	---	---	---

2	5	1	6	3	4	7	8	9
---	---	---	---	---	---	---	---	---

2	1	5	6	3	4	7	8	9
---	---	---	---	---	---	---	---	---

2	1	3	6	5	4	7	8	9
---	---	---	---	---	---	---	---	---

1	2	3	6	5	4	7	8	9
---	---	---	---	---	---	---	---	---

1	2	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---	---

Observamos que hace el mismo número de pasos con ambos criterios.

Febrero 2007-1ª (ejercicio 2) (parecido a ejercicio 3 de Septiembre 2008)

Enunciado: Escribir en pseudocódigo una función recursiva de búsqueda binaria en un vector, suponiendo que en algunos casos el elemento no está en el vector. Demuestra el coste que tiene la función implementada.

Respuesta: Los códigos que se piden en la primera pregunta son los siguientes, recordando que el primero indicaba si el elemento estaba dentro del vector y el segundo es el propio de la búsqueda binaria:

```
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 ( $T[1..n]$ ,  $x$ )

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

En cuanto a la segunda pregunta, no haría falta más que sustituir los valores en la ecuación de recurrencia y ver que de nuevo (es bastante importante) el coste del algoritmo sería $\theta(\log(n))$.

Este ejercicio se parece mucho al 3 de Septiembre de 2008, en el que piden que al no existir el número devuelva -1 , habría que sustituir en el algoritmo busquedabin $n + 1$ por -1 , que es simplemente un formalismo. Por otro lado, se resuelve igual a como viene en el libro que es de donde se han sacado estos algoritmos anteriores.

Febrero 2007-1ª (ejercicio 3)

Enunciado: Dado un vector $C[1..n]$ de números enteros distintos y un número entero S , se pide plantear un programa de complejidad $\theta(n * \log(n))$ que determine si existen o no dos elementos de C tales que su suma sea exactamente S . En caso de utilizar algoritmos conocidos no es necesario codificarlos.

Respuesta: Hemos resuelto este problema tomando como base el ejercicio 11.16 del libro de Martí. Por tanto, podremos resolver el problema siguiendo la siguiente idea:

1. Ordenar el vector C .
2. Para cada elemento x de C buscar, utilizando búsqueda binaria, el entero $S - x$. Si alguna de las búsquedas tiene éxito, habremos encontrado dos elementos de C que sumen S .

El coste del primer paso está en $\theta(n * \log(n))$ y el del segundo igualmente en $\theta(n * \log(n))$, ya que en el caso peor, se hacen n búsquedas binarias de coste $\theta(n * \log(n))$ para cada una de ellas. Por tanto, el **coste total** del algoritmo está en $\theta(n * \log(n))$.

Septiembre 2007 (ejercicio 2)

Enunciado: Supongamos que un polinomio se representa por un vector $v[0..n]$ de longitud $n + 1$ donde el valor $v[i]$ es el coeficiente de grado i . describir claramente los cálculos y la descomposición algorítmica necesaria para plantear un algoritmo, de orden mejor que cuadrático, que multiplique polinomios $P(x)$ y $Q(x)$ mediante divide y vencerás. NOTA: La solución típica trivial de orden cuadrático puntúa 0 puntos.

Respuesta: Es relativamente simple dar con la solución de orden cuadrático. En este caso, se descompone en mitades $P(x) = Ax^{n/2} + B$ y $Q(x) = Cx^{n/2} + D$ con A, B, C, D polinomios de grado $n/2$ sacando factor común $x^{n/2}$ como se detalla en las expresiones. De esta forma se ve claramente que $P * Q$ es $(Ax^{n/2} + B) * (Cx^{n/2} + D) = ACx^n + (AD + BC)x^{n/2} + BD$ lo que la solución conlleva 4 multiplicaciones de grado $n/2$. El coste sería en este caso cuadrático.

Sin embargo, hay una manera de organizar las operaciones mediante la cual, no es necesario calcular $AD + BC$ mediante 2 productos, sino solo con uno, aprovechando que ya tenemos realizados los productos BD y AC . En este último caso basta con observar que $(A + B) * (C + D) = AC + BC + AD + BD$ y que $BC + AD = (A + B) * (C + D) - AC - BD$ con lo que es posible realizar el cálculo con 3 productos en lugar de 4, ya que el coste de las sumas, si consideramos las multiplicaciones como lineales, sería constante. De manera que $(A + B) * (C + D)$ sería uno de los productos, y AC y BD los otros dos.

Este problema es similar al problema de la multiplicación de enteros muy grandes del texto base (ver resumen de la asignatura o libro de Brassard).

Septiembre 2007 (ejercicio 3)

Enunciado: Con respecto al algoritmo de ordenación por fusión (mergesort).

- a) Escribe el algoritmo.
- b) Dibuja la secuencia de llamadas del algoritmo y la evolución del siguiente vector al ordenarlo. Para ello ten en cuenta que hay que considerar que el problema es suficientemente pequeño cuando el array es de tamaño 2.

(2,0,2,1,9,6,2,3,5,8)

Respuesta:

- a) El algoritmo lo hemos visto en la teoría, aunque lo veremos de nuevo poniendo el pseudocódigo siguiente:

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
       $T[j + 1] \leftarrow T[j];$ 
       $j \leftarrow j - 1;$ 
     $T[j + 1] \leftarrow x$ 

```

- b) Para variar, veremos una solución alternativa (o no visto previamente) de esta ordenación, como sigue:

```

OrdFusion (2, 0, 2, 1, 9, 6, 2, 3, 5, 8)
  OrdFusion (2, 0, 2, 1, 9)
    OrdFusion (2, 0)
      Insertar (2, 0)  $\rightarrow$  (0, 2)
    OrdFusion (2, 1, 9)
      OrdFusion (2)
        Insertar (2)  $\rightarrow$  (2)
      OrdFusion (1, 9)
        Insertar (1, 9)  $\rightarrow$  (1, 9)
      Fusionar ((2), (1, 9))  $\rightarrow$  (1, 2, 9)
      Fusionar ((0, 2), (1, 2, 9))  $\rightarrow$  (0, 1, 2, 2, 9)
  OrdFusion (6, 2, 3, 5, 8)
    OrdFusion (6, 2)
      Insertar (6, 2)  $\rightarrow$  (2, 6)
    OrdFusion (3, 5, 8)
      OrdFusion (3)
        Insertar (3)  $\rightarrow$  (3)
      OrdFusion (5, 8)
        Insertar (5, 8)  $\rightarrow$  (5, 8)
      Fusionar ((3), (5, 8))  $\rightarrow$  (3, 5, 8)
      Fusionar ((2, 6), (3, 5, 8))  $\rightarrow$  (2, 3, 5, 6, 8)
  Fusionar ((0, 1, 2, 2, 9), (2, 3, 5, 6, 8))  $\rightarrow$  (0, 1, 2, 2, 2, 3, 5, 6, 8, 9)

```

Septiembre 2007-reserva (ejercicio 2)

Enunciado: Dos amigos juegan a un sencillo juego de adivinación: uno de ellos piensa un número natural positivo y el otro debe adivinarlo solamente preguntado si es menor o igual que otros números. ¿Qué esquema utilizaría para adivinar el número en tiempo logarítmico? Diseñe el algoritmo y escríbalo en pseudocódigo.

Respuesta: No hay solución oficial a este ejercicio, aunque nos basaremos en el 11.1 del libro de Martí. Por tanto, pasamos a verlo, ya que es interesante este tipo de ejercicio.

Como el número a adivinar puede ser arbitrariamente grande, empezar preguntando por el 1 y seguir después en secuencia (búsqueda lineal) hasta alcanzar el número propuesto no es método práctico. En su lugar, necesitamos hacer una **búsqueda binaria**, que nos permita reducir de forma más rápida el conjunto de candidatos. Sin embargo, el algoritmo de búsqueda binaria funciona con vectores de tamaño fijo y conocido, por lo que el primero será adivinar una **cota superior** del número a adivinar (en principio una cota inferior es 1). Para encontrar dicha cota habrá que seguir un método que genere números cada vez más grandes y de modo que el incremento también aumente de forma rápida. A tal efecto, podremos utilizar las potencias de 2, por ejemplo.

Como anotación mía propia decir que en vez del código del citado libro lo sustituiremos y pondremos el que vemos en la asignatura, recordemos que el primero indicaba si el elemento estaba dentro del vector y el segundo es el propio de la búsqueda binaria:

```
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 ( $T[1..n]$ ,  $x$ )

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

Decir que estos pseudocódigos los he copiado igualmente del compañero Antonio Rivera Cuesta, así que le agradezco personalmente la innegable aportación que siempre da.

Febrero 2008-2ª (ejercicio 3)

Enunciado: Dado n potencia de 2, escriba un algoritmo recursivo que calcule en tiempo logarítmico el valor de a^n suponiendo que sólo se puedan realizar multiplicaciones y que éstas tienen coste unitario. Demostrar el coste mediante la ecuación de recurrencia. No justifique el esquema usado, aplíquelo.

Respuesta: Este ejercicio ya lo pusimos anteriormente, así que nos evitaremos explicarlo, por lo que pondremos el pseudocódigo como sigue:

```
fun exp (a: entero; n: natural) dev entero
  si  $n = 1$  entonces dev a
  si no
    si  $n = 0$  entonces dev 1
    si no
       $t \leftarrow \text{exp}(a, n \text{ DIV } 2)$ 
      dev  $t * t$ 
    fsi
  fsi
ffun
```

Veríamos la ecuación de recurrencia, que es:

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

De nuevo, podremos sacar las distintas variables de la **reducción por división** de la recursión como sigue:

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

Siendo dichas variables:

a: Número de llamadas recursivas = 1

b: Reducción del problema en cada llamada = 2

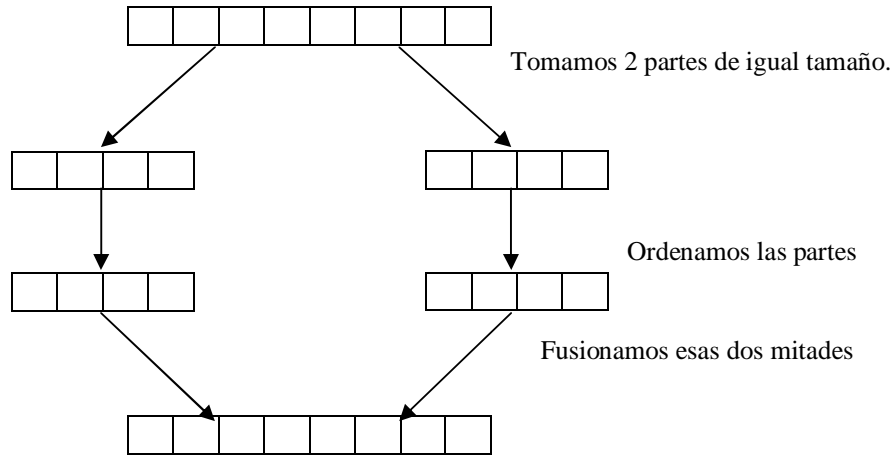
$c * n^k$: Coste de las operaciones extras a las llamadas recursivas. Tendremos que el valor de $k = 0$, al ser el tiempo extra constante.

De nuevo, sustituimos en la ecuación $a = b^k$, siendo, por tanto, $1 = 2^0$, el coste sería entonces el del segundo caso, por ello, $t(n) \in \theta(n^k * \log(n)) = \theta(\log(n))$.

Septiembre 2007-reserva (ejercicio 2)

Enunciado: El algoritmo 'mergesort' posee una complejidad $T(n) \in \theta(n * \log(n))$, describa y demuestre un caso en el que 'mergesort' tiene una complejidad $T(n) \in \theta(n^2)$.

Respuesta: Recordemos que el algoritmo de ordenación por fusión (mergesort) realizaba lo siguiente:



En este caso, veíamos que estaban bien distribuidas las mitades, es decir, se dividían exactamente por dos. El peor caso, el caso en el que es cuadrático el coste corresponde cuando no lo están y queda un elemento solo para ordenar y $n - 1$ en la otra partición. Además, esto lo vimos en la teoría de la asignatura, que conviene estudiarla previo a hacer los ejercicios:

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

Como se observa y ya se demostró en ese apartado al reducirse la recursividad en 1, tendríamos coste cuadrático. Sólo hay que sustituir los valores en la ecuación de la recurrencia y luego resolverlo con la fórmula adecuada.

2ª parte. Problemas de exámenes solucionados:

Tendremos en cuenta los pasos a seguir para resolver problemas:

- Elección del esquema (voraz, vuelta atrás, divide y vencerás).
- Identificación del problema con el esquema.
- Estructura de datos.
- Algoritmo completo (con pseudocódigo).
- Estudio del coste.

Antes de pasar a ver los problemas definiremos brevemente el preorden bien fundado, que es un preorden \preceq (es un signo de precedencia) en D, se dice que es bien fundado si no existen en D sucesiones infinitas estrictamente decreciente, es decir:

$$\{x_i\} \text{ tales que } \forall i \in \mathbb{N}, x_{i+1} < x_i$$

Un ejemplo de preorden bien fundado es la relación \leq de los naturales o también el orden lexicográfico (es decir, el abecedario).

Ponemos esta definición en la segunda parte debido a que veremos en algunos problemas de exámenes que soliciten el preorden bien fundado, no haciéndolo en la primera parte o en general, en las cuestiones. Así mismo, dicha definición se ve mucho más ampliada en la asignatura de programación 2, por lo que para su ampliación se debería ver los temas de la citada asignatura.

Febrero 1996-1ª (problema 1) (igual a 3.3 libro de problemas resueltos)

Enunciado: El tiempo de ejecución de un algoritmo viene dado por $T(n) = 2^{n^2}$. Encontrar una forma eficiente de calcular $T(n)$, suponiendo que el coste de multiplicar dos enteros es proporcional a su tamaño en representación binaria.

Respuesta:

1. Elección razonada del esquema algorítmico

Una forma eficiente de calcular exponenciaciones es aplicando la técnica de Divide y Vencerás y usando la relación

$$a^n = (a^{n \text{ div } 2})^2 * a^{n \text{ mod } 2}$$

(es decir, $a^n = (a^{n \text{ div } 2})^2$ si n es par y $a^n = (a^{(n-1) \text{ div } 2})^2$ si n es impar)

Tenemos que tener en cuenta que *div* significa el resultado de la división y *mod* el resto.

2. Descripción del esquema usado e identificación con el problema

El **esquema general** es:

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

Particularizaremos las siguientes funciones para este problema de la siguiente manera:

- **Tamaño umbral y solución-simple:** El preorden bien fundado para los problemas se deriva directamente del orden total entre los exponentes (n) como números naturales. Podemos tomar como tamaño umbral $n = 1$, caso en que la exponenciación es trivial y se devuelve como resultado el mismo dato de entrada.
- **Descomposición:** 2^{n^2} se descompone en un único subproblema, $2^{n^2/2}$ de la mitad de tamaño. Se trata, pues, de un problema de reducción más que descomposición. El convertir un problema de tamaño n en otro de tamaño $n/2$ es lo que nos proporcionará un algoritmo eficiente.
- **Combinación:** La función de combinación viene dada por la fórmula anterior para $a = 2$:

$$2^n = (2^{n \text{ div } 2})^2 * 2^{n \text{ mod } 2}.$$

3. Estructuras de datos

La única estructura de datos necesaria en el problema son los **enteros**. Según las condiciones del problema, supondremos que se dispone de un algoritmo de multiplicación de enteros de coste proporcional al tamaño de éstos en representación binaria (0 ó 1). En particular, 2^n se representa en binario como un 1 seguido de n ceros, de forma que el coste de la operación $(2^n)^2$ es proporcional a n (por ejemplo, $2^5 = 10000$).

4. Algoritmo completo a partir del refinamiento del esquema general

Tendremos las siguientes funciones, siendo la *principal* la siguiente:

```
fun T (n: entero) dev entero
   $m \leftarrow n * n$ 
  dev exp (2, m)
ffun
```


La función *exp*, que será la que realmente realice la exponenciación (nuestro problema de divide y vencerás) es:

```

fun exp (a, n: entero) dev entero
  si  $n \leq 1$  entonces dev solución-simple (a, n)
  si no
     $p \leftarrow n \text{ div } 2$            // Cociente
     $r \leftarrow n \text{ mod } 2$        // Resto
     $t \leftarrow \text{exp}(a, p)$ 
    dev combinación (t, r, a)
  fsi
ffun

```

En los argumentos de *combinación* sólo el primero es un subproblema. Los otros dos son parámetros auxiliares que utiliza la función de *combinación*, la cual veremos a continuación:

```

fun combinación (t, r, a: entero) dev entero
  dev  $t * t * a^r$ 
ffun

```

Por último, veremos la función solución-simple:

```

fun solución-simple (a, n: entero) dev entero
  si  $n = 1$  entonces dev a
  si no dev 1
  fsi
ffun

```

5. Estudio del coste

El coste de calcular una exponenciación según este algoritmo cumple la siguiente relación:

$$T(n) = T(n/2) + cte * n$$

Es decir, calcular a^n se reduce a calcular $a^{n \text{ div } 2}$ y multiplicar dos o tres enteros cuyo tamaño es del orden de n . Las operaciones $n \text{ div } 2$ y $n \text{ mod } 2$ tienen un coste constante sobre números en representación binaria.

La ecuación de recurrencia será:

$$T(n) = \begin{cases} c * n^k & \text{si } 1 \leq n < b \\ a * T(n/b) + c * n^k & \text{si } n \geq b \end{cases}$$

La resolución de la recurrencia es:

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

Las variables de la ecuación de recurrencia correspondientes con nuestro problema serán:

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

Siguiendo el procedimiento habitual, sustituiremos en la ecuación $a = b^k$, siendo éste el caso primero, por lo que el coste será $\theta(n)$. Como la cantidad que necesitamos calcular es $T(n) = 2^{n^2}$, esa operación tendrá un coste $\theta(n^2)$.

Si no empleáramos el algoritmo de divide y vencerás tendrá un coste $\theta(n^3)$.

Febrero 1996-2ª (problema 2) (igual a 3.1 libro de problemas resueltos)

Enunciado: Dada la sucesión definida como $f_n = af_{n-3} + bf_{n-2} + cf_{n-1}$ se pide diseñar un algoritmo que calcule un tiempo logarítmico en termino f_n .

Sugerencia: Utilizad la siguiente relación:

$$\begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ a & b & c \end{pmatrix} \cdot \begin{pmatrix} f_{n-3} \\ f_{n-2} \\ f_{n-1} \end{pmatrix} = \begin{pmatrix} f_{n-2} \\ f_{n-1} \\ f_n \end{pmatrix}$$

Respuesta:

1. Elección razonada del esquema algorítmico

Cada término de la sucesión se calcula en función de las 3 anteriores. Por tanto, la forma trivial de calcular f_n consiste en calcular los n términos anteriores, lo que supone un coste lineal $O(n)$. Sin embargo, la ecuación que se nos proporciona nos da una forma más eficiente de resolver el problema, recurriendo a la técnica de **Divide y Vencerás**. Efectivamente, si llamamos:

$$F = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ a & b & c \end{pmatrix}$$

Tendremos

$$\begin{pmatrix} f_{n-2} \\ f_{n-1} \\ f_n \end{pmatrix} = F \cdot \begin{pmatrix} f_{n-3} \\ f_{n-2} \\ f_{n-1} \end{pmatrix} = F^2 \cdot \begin{pmatrix} f_{n-4} \\ f_{n-3} \\ f_{n-2} \end{pmatrix} = \dots = F^{n-2} \cdot \begin{pmatrix} f_0 \\ f_1 \\ f_2 \end{pmatrix}.$$

donde f_0 , f_1 y f_2 no pueden calcularse en término de los elementos anteriores y han de ser, por tanto, datos del problema.

Para calcular f_n es suficiente, por tanto, con evaluar F^n y hacer una multiplicación de una matriz de 3 x 3 por un vector de 3 elementos (que tiene coste constante). Sabemos que la forma más eficiente de calcular exponenciaciones es mediante la técnica de divide y vencerás, aplicada mediante la igualdad:

$$F^n = (F^{n \text{ div } 2})^2 * F^{n \text{ mod } 2}$$

(es decir, $F^n = (F^{n \text{ div } 2})^2$ si n es par y $F^n = (F^{(n-1) \text{ div } 2})^2$ si n es impar)

Nos fijamos que es igual a la exponenciación del ejercicio anterior, casi sin diferencia.

2. Descripción del esquema e identificación con el problema

El esquema de Divide y Vencerás es una **técnica recursiva** que consiste en dividir un problema en varios subproblemas del mismo tipo. Las soluciones a estos subproblemas se combinan a continuación para dar la solución al problema original. Cuando los subproblemas son más pequeños que un umbral prefijado, se resuelven mediante un algoritmo específico. Si su tamaño es mayor, se vuelven a componer.

El **esquema general** es el siguiente:

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

Los elementos de este esquema se particularizan así a nuestro caso:

- **Tamaño umbral y solución-simple:** El preorden bien fundado para los problemas se deriva directamente del orden total entre los exponentes (n) como números naturales. Podemos tomar como tamaño umbral $n = 1$, caso en que la exponenciación es trivial y se devuelve como resultado la matriz de entrada.
- **Descomposición:** F^n se descompone en un único subproblema, $F^{n/2}$ de la mitad de tamaño. Se trata, pues, de un problema de *reducción* más que descomposición. El convertir un problema de tamaño n en otro de tamaño $n/2$ es lo que nos proporcionará un algoritmo eficiente.
- **Combinación:** La función de combinación es la ecuación mencionada más arriba.

Nos fijamos, de nuevo, que incluso las definiciones son muy parecidas a las vistas previamente en el problema anterior.

3. Estructuras de datos

En el problema intervienen solamente enteros y matrices de 3 x 3. Los datos de entrada y salida del algoritmo son enteros; las matrices se utilizan como resultados intermedios. Daremos por conocida la multiplicación de matrices.

4. Algoritmo completo

Suponemos conocidos a, b, c, f_0, f_1, f_2 . Empezaremos viendo la función principal, como sigue:

```

fun f (n: entero) dev entero
    
$$F \leftarrow \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ a & b & c \end{pmatrix}$$

    caso  $n = 0$ : dev  $f_0$ ;
    []  $n = 1$ : dev  $f_1$ ;
    []  $n = 2$ : dev  $f_2$ ;
    [] verdadero: hacer
         $S \leftarrow \text{exp\_mat}(F, n/2)$ 
        
$$s \leftarrow S \begin{pmatrix} f_0 \\ f_1 \\ f_2 \end{pmatrix}$$

        dev  $s[3]$ 
    fcaso
ffun
    
```

La siguiente función es la exponencial:

```

fun exp_mat (M: vector[3,3] de enteros, n: entero) dev vector[3,3] de enteros
    si  $n \leq 1$  entonces
        devolver solución_simple (M, m)
    si no
         $p \leftarrow n \text{ div } 2$ 
         $r \leftarrow n \text{ mod } 2$ 
         $T \leftarrow \text{exp\_mat}(M, p)$ 
        dev combinación (T, r, M)
    fsi
ffun
    
```

Se modifica algo la anterior función, ya que el vector es de enteros, tanto el del argumento como el que devuelve.

En los argumentos de combinación sólo el primero es un subproblema. Los otros dos son parámetros auxiliares que utiliza la función de combinación (recordemos que lo vimos en el ejercicio anterior):

```

fun combinación (T: vector[3,3] de enteros, r: entero, M: vector[3,3] de enteros)
    dev entero
        dev  $T * T * M^r$ 
ffun
    
```

donde $M^1 = M$; $M^0 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$

Por último, veremos la función solución-simple:

```

fun solución-simple (M: vector[3,3] de enteros, n: entero) dev entero
  si n = 1 entonces dev M
  si no
    dev  $\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$ 
  fsi
ffun

```

5. Estudio del coste

El coste del algoritmo recae sobre la exponenciación de la matriz F. Como se reduce un problema de tamaño n a otro de tamaño $n/2$, el coste cumple la **ecuación de recurrencia** $T(n) = T(n/2) + cte$, donde *cte* hace referencia al tiempo necesario para hacer una o dos multiplicaciones de matrices 3×3 . Seguiremos con nuestro planteamiento del ejercicio anterior, siendo ambos casi idénticos.

La ecuación de recurrencia de reducción por división será:

$$T(n) = \begin{cases} c * n^k & \text{si } 1 \leq n < b \\ a * T(n/b) + c * n^k & \text{si } n \geq b \end{cases}$$

La resolución de la recurrencia es:

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

Las variables de la ecuación de recurrencia correspondientes con nuestro problema serán:

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. Tendremos que el valor de $k = 0$, al ser el tiempo extra constante, lo que vimos previamente.

Siguiendo el procedimiento habitual, sustituiremos en la ecuación $a = b^k$, siendo éste el caso segundo, por lo que el coste será $\theta(\log(n))$. Hemos visto ejercicios anteriores éste sería el coste de este algoritmo, como puede ser el de búsqueda binaria.

NOTA DEL AUTOR: Este problema se parece bastante al de Febrero de 2003-2ª semana, la única diferencia es que la sucesión definida es $f_n = af_{n-1} + bf_{n-2}$, por lo que hay que hacer 2 sumas en vez de 3 que nos solicitan en este ejercicio. Tenía idea ponerlo aparte, pero me he dado cuenta que es repetir el mismo ejercicio de manera absurda.

Septiembre 1996-reserva (problema 1) (igual a 3.2 libro de problemas resueltos)

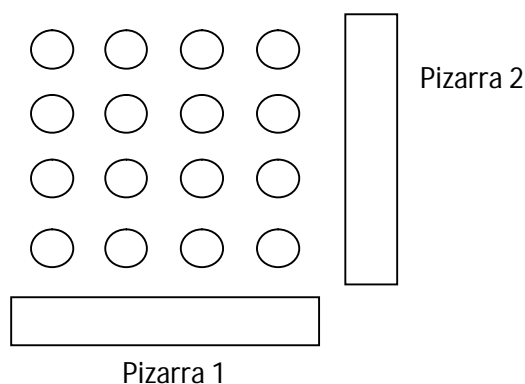
Enunciado: En una clase hay f filas y c columnas de pupitres. Delante de la primera fila se encuentra la pizarra:

- Diseñar un algoritmo que reparta los $f * c$ alumnos de forma que, al mirar hacia la pizarra, ninguno se vea estorbado por otro alumno más alto que él.
- A mitad del curso se coloca una pizarra adicional en una de las paredes adyacentes con la primera pizarra. Diseñar un algoritmo que coloque a los alumnos de forma que puedan mirar también a esa segunda pizarra sin estorbarse. Este algoritmo debe sacar partido de la colocación anterior.

Respuesta:

1. Elección razonada del esquema algorítmico

Para que nadie tenga un alumno más alto que él al mirar la pizarra, es necesario que, dentro de cada columna, los alumnos están ordenados según su altura de mayor a menor:



Para resolver el apartado a) de forma eficiente basta con **dividir** los $f * c$ alumnos en c subconjuntos de f elementos escogidos al azar y, a continuación, debe ordenarse cada uno de esos subconjuntos. Cada uno de ellos será una columna en la clase. Como algoritmo de ordenación puede escogerse cualquiera de los estudiados; nosotros utilizaremos el algoritmo de Divide y Vencerás de *fusión*, por ser más eficiente asintóticamente en el caso peor.

Al colocar una segunda pizarra adyacente a la primera, los alumnos de cada fila deben estar, a su vez, ordenados entre sí. Para que estén ordenadas las columnas y las filas, es necesario ordenar a todos los alumnos de menor a mayor, y colocarlos en la clase de forma que el más bajito ocupe el pupitre que está en la intersección de las dos pizarras, y el más alto en el vértice opuesto de la clase. Por lo tanto, para obtener la disposición final de los alumnos en el apartado b) debe hacerse una ordenación de $f * c$ elementos. Pero si aprovechamos la disposición anterior no es necesario, esta vez, aplicar ningún algoritmo de ordenación: basta con realizar una fusión de c subconjuntos ya ordenados (equivaldría al último paso de un algoritmo de ordenación por fusión en el que el factor de división fuera c). Así, la ordenación final puede obtenerse en **tiempo lineal**.

2. Descripción del esquema usado e identificación con el problema

Dado el esquema de divide y vencerás:

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

Se particulariza para nuestro problema de la siguiente manera:

- **Tamaño umbral y solución-simple:** El preorden bien fundado para los problemas se deriva directamente del orden total entre el tamaño de los subconjuntos problema. Podemos tomar como tamaño umbral $n = 1$, caso en que la exponenciación es trivial y consiste simplemente en devolver el elemento.
- **Descomposición:** Dividiremos el conjunto problema en dos subconjuntos formados por los $n \div 2$ primeros elementos por un lado y el resto por otro.
- **Combinación:** Es necesario fundir los dos subconjuntos ordenados, mediante un bucle que toma cada vez el menor elemento de entre los primeros de uno y otro subconjunto todavía sin seleccionar.

3. Estructuras de datos

La única estructura de datos que necesitamos es una matriz de enteros de tamaño $f * c$ que almacene las alturas de los alumnos. También podremos utilizar un *vector* de tamaño $f * c$ sabiendo que cada f elementos representa una columna, aunque es menos natural.

4. Algoritmo completo a partir del refinamiento del esquema general

Llamaremos a a la función que obtiene la ordenación requerida en el apartado a) y b a la que soluciona en el apartado b), es decir, obtiene una ordenación total a partir de la que se tiene en a).

La función a es la siguiente:

```
fun a (clase: vector[1..f, 1..c] de reales) dev vector[1..f, 1..c] de enteros
  para  $i \leftarrow 1$  hasta  $c$  hacer
    clase[1..f,  $i$ ]  $\leftarrow \text{ordenacion}(\text{clase}[1..f, i])$ 
  fpara
  dev clase
ffun
```

La función ordenación equivale a la función general divide y vencerás, que modificaremos ligeramente para incluir como argumento vectores que sean siempre del mismo tamaño:

```

fun ordenación (v: vector[1..n]; i, j: entero)
  si  $i \neq j$  entonces
     $k \leftarrow (i + j) \text{ div } 2$ 
    ordenación (v, i, k)
    ordenación (v, k + 1, j)
    fusión (v, i, j, 2)
  fsi
ffun

```

La función de fusión tiene **cuatro parámetros**: el vector, el principio y final del tramo que contiene a los dos subvectores que hay que fusionar, y el factor de división empleado. Para el apartado a) podríamos ahorrarnos este último argumento (es 2), pero esa generalización nos permitirá usar la misma función que el apartado b).

```

proc fusión (v: vector [1..n]; inicial, final, factor: entero)
  { Inicializa el vector solución }
   $v' \leftarrow \text{vector } [1..n]$ 
  { Inicializa punteros al comienzo de los vectores por fusionar }
  para  $k \leftarrow 1$  hasta factor hacer
     $i_k \leftarrow \text{inicio} + \text{factor} * (k - 1)$ 
  fpara
   $I \leftarrow \{i_1 \dots i_{\text{factor}}\}$ 
  { Selecciona el menor elemento de entre los principios de vector para
    incluirlo en la solución, y a continuación lo borra para que no vuelva a ser
    considerado }
  para  $h \leftarrow \text{inicio}$  hasta final hacer
     $i_x \leftarrow \text{elemento que maximiza } v[i_x]$ 
     $v'[i_x] \leftarrow v[i_x]$  // Guarda en vector solución
    si  $i_x < \text{inicio} + \text{factor} * x - 1$  entonces
       $i_x \leftarrow i_x + 1$  // Incrementa valor del puntero
    si no
       $I \leftarrow I \setminus \{i_x\}$ 
    fsi
  fpara
   $v \leftarrow v'$ ;
fproc

```

La función *b* debe conseguir una ordenación completa del conjunto de alumnos, pero debe tener en cuenta que ya exista una ordenación parcial entre ellos: los alumnos de cada columna están ordenados entre sí. Si representamos el conjunto de los alumnos como un vector $f * c$ elementos, en el que los f primeros elementos corresponden a la primera columna, los f siguientes a la segunda, etc. El problema queda solucionado llamando a la función de *fusión* definida anteriormente, pero utilizando un factor de división c en lugar de 2:

```

fun b (v: vector[1..c * f]; c: entero) dev vector[1..c * f]
  fusión (v, 1, c * f, c)
ffun

```


5. Estudio del coste

Apartado a

La ordenación por fusión tiene un coste que cumple:

$$T(n) = 2 * T(n/2) + cte * n$$

De nuevo estamos ante un caso de reducción de la **recursividad por división**. El algoritmo de fusión tiene un coste $O(n)$ (consta de dos bucles consecutivos). De esa igualdad se obtiene un coste $O(n * \log(n))$. Como se realizan c combinaciones de f elementos cada una, el coste total es $O(c(f * \log(f)))$ (estimo que es una ordenación parcial, de ahí ese coste).

Mediante una ordenación total habríamos resuelto también el problema, pero con un coste $O(n * \log(n)) \rightarrow O(cf * \log(cf))$.

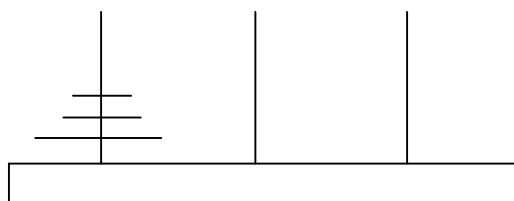
Apartado b

Se resuelve mediante una llamada al algoritmo de fusión, que tiene un coste lineal; como el tamaño del problema es $c * f$, el coste es $O(c * f)$. El coste es mucho menor que en el caso que no aprovechemos la ordenación parcial que se obtiene en el apartado a.

NOTA DEL AUTOR: Se observa que los vectores, salvo algunos añadidos míos no se saben de qué tipo de son, poniendo sólo `vector[1..n]`, por no estar seguro del mismo. Se dejará como ejercicio el hacerlos, ya que desconozco de que tipos pueden ser, si reales o enteros.

Problema 3.4 del libro de problemas resueltos (sin correspondencia con ningún ejercicio de examen)

Enunciado: Se tienen 3 varillas y n discos agujereados por el centro. Los discos son todos de diferente tamaño y en la posición inicial se tienen los discos insertados en el primer palo y ordenados en tamaños decrecientes desde la base hasta la punta de la varilla. El problema consiste en pasar los discos de la primera a la tercera varilla observando las siguientes reglas: se mueven los discos de uno en uno y nunca un disco puede colocarse encima de otro menor que éste.



Respuesta:

1. Elección razonada del algoritmo

Aparentemente el problema parece solucionarse mediante una búsqueda en un árbol de búsqueda como vimos con los juegos en el tema 9 (de vuelta atrás), sin embargo, hay una forma que nos asegura que utilizamos el mínimo número de movimientos mediante Divide y Vencerás. La justificación de la elección se basa, por tanto, en la **eficiencia** más que en la imposibilidad de utilizar el esquema de Vuelta Atrás.

Por otra parte, hay que indicar que un esquema voraz es imposible de aplicar, ya que la solución no consta de unos elementos elegidos de un conjunto de candidatos, sino de los movimientos efectuados sobre ellos. Para que un conjunto de candidatos de tal naturaleza

fuera posible, habría que añadir a cada movimiento información acerca del número de discos que descansa sobre cada varilla, lo cual es tanto como enumerar todas las posibles combinaciones del juego.

2. Descripción del esquema usado e identificación con el problema

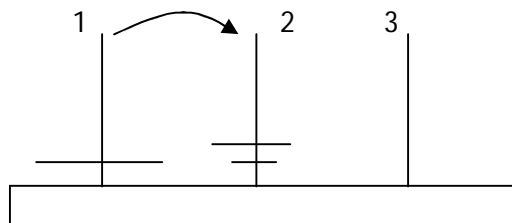
El **esquema general** es:

```

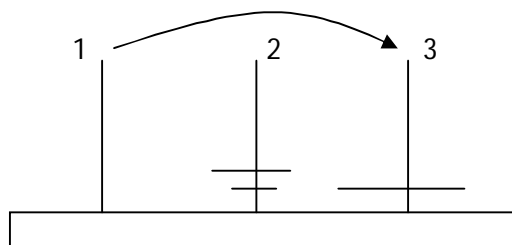
fun divide-y-vencerás (problema)
  si suficientemente-simple (problema) entonces
    dev solucion-simple (problema)
  si no
    { No es solución suficientemente simple }
     $\{p_1 \dots p_k\} \leftarrow \text{decomposicion (problema)}$ 
    para cada  $p_i$  hacer
       $s_i \leftarrow \text{divide-y-vencerás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

y se particulariza de la siguiente forma:

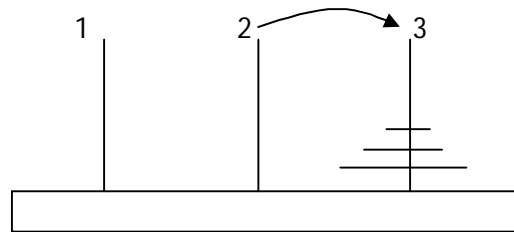
- **Tamaño umbral y solución-simple:** El preorden bien fundado se establece sobre los tamaños de los sucesivos problemas en la cadena de llamadas recursivas. El único problema verdaderamente simple es el de solucionar el pasatiempo para un único disco, por lo que el tamaño umbral es $n = 1$.
- **Descomposición:** Cualquier solución para pasar n discos del poste 1 al 3 pasa por:
 1. Conseguir pasar $n - 1$ discos del poste 1 al 2. Siguiendo las condiciones del enunciado sería algo así:



2. Pasar el disco que queda (que además será el de mayor radio de los que habrá en el poste 1) del 1 al 3:



3. Pasar, por último, los $n - 1$ discos que habíamos puesto en el poste auxiliar 2, de dicho poste al 3:



En general, si tenemos k discos en i y queremos pasarlo a j , siendo $i, j \in \{1, 2, 3\}$ habrá que calcular $6 - i - j$ para hallar el número del poste restante que se utilizará como *auxiliar* y se siguen los pasos anteriores.

NOTA DEL AUTOR: He hecho los dibujos para que se vea gráficamente este ejercicio paso a paso. Creo que así queda más claro.

- **Combinación:** No hay función de combinación. Las soluciones parciales son validas en sí mismas. En el caso que nos ocupa, éstas (las soluciones) son movimientos expresados mediante origen y destino.

3. Estructuras de datos

No hay estructuras complejas de datos. Se utilizarán únicamente tipos de datos **estándar**.

4. Algoritmo completo a partir del refinamiento del esquema general

Los parámetros del algoritmo serán la varilla origen, la varilla destino y el número de discos que deben ser movidos del origen al destino. La llamada inicial tendrá como origen la varilla 1, como destino la 3 y con un número n de discos.

El **algoritmo refinado** se expone a continuación:

```

fun hanoi (origen, destino, n: entero)
  si  $n = 1$  entonces
    escribe "Mover disco de poste origen a poste destino"
  si no
    hanoi (origen,  $6 - destino - origen$ ,  $n - 1$ )
    hanoi (origen, destino, 1)
    hanoi ( $6 - origen - destino$ , destino,  $n - 1$ )
  fsi
ffun
  
```

La función de combinación **no** es necesaria, ya que cada descomposición del problema en un subproblema produce un movimiento y dos llamadas a la función.

5. Estudio del coste

Debemos plantear la ecuación de recurrencia. En el caso que nos ocupa, al entrar a la función tenemos una instrucción *si ... entonces* con coste 1 seguida de 3 llamadas a la función, de manera que $T(n) = 2 * T(n - 1) + T(1) + 1$. Simplificando $T(1)$ tenemos finalmente que $T(n) = 2 * T(n - 1) + 2$ y resolviendo con las formulas antes puestas tiene coste del orden de $O(2^n)$.

Problema 3.5 del libro de problemas resueltos (sin correspondencia con ningún ejercicio de examen)

Enunciado: Se tiene un vector de enteros no repetidos y ordenados de menor a mayor. Diseñar un algoritmo que compruebe en tiempo logarítmico si existe algún elemento del vector que coincida con su índice.

Respuesta:

1. Elección razonada del esquema algorítmico

Se trata del esquema de **Divide y Vencerás** por dos motivos:

- Se exige coste logarítmico y esto solo es posible si el problema se reduce (al menos) a la mitad en cada paso (mucho ojito con esto, en muchas ocasiones es determinante).
- Se puede descomponer la estructura de datos en partes de su misma naturaleza.

Un esquema de búsqueda ciega en el vector carece por otra parte, de sentido (no es vuelta atrás) y tampoco se trata de un esquema voraz, ya que no hay candidatos que haya que escoger ningún tipo de conjunto (no es voraz).

2. Descripción del esquema usado e identificación con el problema

El **esquema general** es:

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

Se trata de un problema en el que no hay que efectuar una combinación posterior de las soluciones. A este tipo de problemas se les denomina de reducción o simplificación dentro del esquema de Divide y Vencerás.

Lo más importante de este tipo de problemas de simplificación es que hay que tener en cuenta con qué mitad del vector nos quedamos para seguir buscando. Este criterio es prácticamente lo único que tenemos que decidir, ya que el algoritmo debe sólo devolver falso o dar un índice del vector.

Supongamos que es i es el elemento del vector v , o bien i, j los elementos centrales en caso de que tengan un número par de elementos. Si $v[i] < i$ sabemos que podemos descartar la parte izquierda del vector como susceptible de contener en ella algún elemento tal que $v[i] = i$. Lo mismo le ocurre a la derecha para la condición contraria. Esto ocurre por dos motivos:

- Porque se trata de números enteros.
- Porque se prohíben las repeticiones.

Se debería hacer la comprobación de que son dos condiciones imprescindibles para que el algoritmo pueda ser resuelto con los requisitos de coste del enunciado.

Lo único que resta es comprobar las condiciones para vectores con un número par e impar de elementos. Para dividir el vector se utilizará un índice sobre él que se pasa como argumento.

3. Estructuras de datos

No hay más estructura de datos que el **vector de enteros**.

4. Algoritmo completo a partir del refinamiento del esquema general

La *condición trivial* se aplica a vectores de un solo elemento:

```
fun solución-trivial (v: vector; i: natural) dev boolean
  dev (v[i] = i)
ffun
```

El *esquema refinado* será el siguiente:

```
fun indice (v: vector; i, j: natural)
  si i = j entonces solución-trivial (v, i)
  si no
     $k \leftarrow (i + 1) \text{ div } 2$ 
    si  $v[k] \geq k$  entonces índice (v, i, k)
    si no
      índice (v, i, k)
    fsi
  fsi
ffun
```

NOTA DEL AUTOR: Esta parte de problemas del libro de problemas resueltos por norma general están bastante bien hechos y claros, sólo que este ejercicio tiene una errata en la condición del "si" más interior, es decir, en la expresión $v[k] \geq k$, que hemos tratado de subsanarlo. Como es habitual, está hecho por un alumno, así que no se sabe si está realmente bien, aunque se intenta. Nos fijamos, además, que es de nuevo una *búsqueda binaria*, con las implicaciones de coste que tanto hacemos hincapié.

5. Estudio del coste

La ecuación de recurrencia del algoritmo es $T(n) = T(n/2) + 1$. Su resolución es, como esperábamos (en este caso, no lo haremos por haberlo hecho en innumerables ocasiones), $T(n) \in \log(n)$.

Febrero 2002-1ª (problema)

Enunciado: una caja con n bombones se considera "aburrida" si se repite un mismo tipo de bombón (denominado bombón "pesado") más de $n/2$ veces. Programar el pseudocódigo un algoritmo que decida si una caja es "aburrida" y devuelva (en su caso) el tipo de bombón que le confiere dicha propiedad. El coste debe ser a lo sumo $\theta(n * \log(n))$.

NOTA: Si una caja tiene un bombón "pesado", entonces necesariamente también lo es de al menos una de sus dos mitades.

Respuesta:

Este ejercicio no está resuelto por el equipo docente (si no estoy equivocada), por lo que se toma esta solución de otro sitio distinto. Notaremos que todos los ejercicios de este mismo tipo (de contar votos o similares) se parecen entre sí mucho, y que posteriormente veremos.

1. Elección razonada del esquema algorítmico

Se trata del esquema de Divide y Vencerás, ya que se puede descomponer la estructura de datos en partes de su misma naturaleza, resolviendo el problema con el coste requerido. Además, la propiedad de la nota del enunciado hace referencia a las mitades del problema.

El tamaño del problema viene dado por el número de bombones que es un natural. En los naturales no hay sucesiones decrecientes infinitas, por tanto, tendremos un **preorden bien fundado**.

2. Descripción del esquema usado e identificación con el problema

El esquema de Divide y Vencerás es una técnica recursiva que consiste en **dividir** el problema en varios subproblemas del mismo tipo. Las soluciones a estos subproblemas se **combinan** a continuación para dar la solución al problema original. Cuando los subproblemas son más pequeños que un umbral prefijado, se resuelven mediante un algoritmo específico. Si su tamaño es mayor, se vuelven a descomponer.

El **esquema general**, por tanto, es el siguiente:

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

Se particulariza estos elementos a nuestro caso:

- **Tamaño umbral y solución-simple:** El preorden se establece sobre los tamaños de los sucesivos problemas en la cadena de llamadas recursivas. El único problema verdaderamente simple es el de solucionar el problema para una caja de bombón, por lo que el tamaño umbral es $n = 1$.
- **Descomposición:** Dividiremos el conjunto problema en dos subproblemas de tamaño mitad ($n/2$).

- **Combinación:** Las dos llamadas devuelven:

$$\langle b_1, e_1 \rangle \langle b_2, e_2 \rangle$$

Siendo b_1 y b_2 dos booleanos que nos informan de la existencia de “bombón pesado” en cada una de las dos mitades respectivas, y e_1 y e_2 los “bombones pesados”, en caso de que alguno o ambos existan.

Tendremos que resolver el problema para que nos dé la solución del “bombón pesado” en todo el vector, es decir, hallar sabiendo las mitades cuál es la solución $\langle b, e \rangle$. Para ello, veremos el enunciado:

Si e es pesado en el vector $[1..n]$, lo será en la primera mitad o en la segunda. Como ejemplo, podrá ser:

A	A		A	B
---	---	--	---	---

Sin embargo, la implicación en el otro sentido no es cierta. Por ejemplo:

A	A		B	B
---	---	--	---	---

Por tanto, un “bombón pesado” en alguna de las mitades sólo será candidato a “bombón pesado” en todo el vector, requiriendo un conteo posterior del mismo.

Tendremos estos **casos** a distinguir:

1. Si $b_1 = \text{falso}$ y $b_2 = \text{falso}$, entonces significa que no hay ningún candidato y, por tanto, deducimos que $b = \text{falso}$.
2. Si $b_1 = \text{verdadero}$ y $b_2 = \text{falso}$, entonces se ha encontrado un candidato que es el e_1 . Por ello, hay que contar en todo el vector si supera la mitad de los elementos de dicho candidato. Es decir, verificamos esta condición:

$$\text{contar}(e_1, v[1..n]) > n/2$$

dándose estos casos posibles:

- SI: Existe “bombón pesado” y es el candidato e_1 , es decir, $b = \text{verdadero}$, $e = e_1$.
 - NO: No existe “bombón pesado”, o lo que es igual, $b = \text{falso}$.
3. Si $b_1 = \text{falso}$ y $b_2 = \text{verdadero}$, entonces se ha encontrado un candidato que es el e_2 . Por ello, hay que contar en todo el vector si supera la mitad de los elementos de dicho candidato. Es decir, verificamos esta condición:

$$\text{contar}(e_2, v[1..n]) > n/2$$

dándose estos casos posibles:

- SI: Existe “bombón pesado” y es el candidato e_2 , es decir, $b = \text{verdadero}$, $e = e_2$.
- NO: No existe “bombón pesado”, o lo que es igual, $b = \text{falso}$.

4. El último caso es el más especial y es aquél en el que $b_1 = \text{verdadero}$ y $b_2 = \text{verdadero}$, es decir, ha encontrado dos candidatos distintos en ambas partes, por lo que se tiene que verificar si en todo el vector sale el candidato e_1 . Al igual que antes tendríamos que comprobar esta condición:

$$\text{contar}(e_1, v[1..n]) > n/2$$

dándose estos casos posibles:

- SI: Existe “bombón pesado” y es el candidato e_1 , es decir, $b = \text{verdadero}$, $e = e_1$.
- NO: Esto significa que podría ser que el otro candidato posible fuera “bombón pesado” por lo que tendríamos de nuevo que contar, así que de nuevo verificaríamos esta condición:

$$\text{contar}(e_2, v[1..n]) > n/2$$

de nuevo, tendremos otros subcasos, que serían los siguientes:

- SI: Existe “bombón pesado” y es el candidato e_2 , es decir, $b = \text{verdadero}$, $e = e_2$.
- NO: No existe “bombón pesado”, o lo que es igual, $b = \text{falso}$.

3. Estructuras de datos

La única estructura de datos que necesitamos es un **vector de n componentes** para almacenar los bombones; para representar un bombón se puede utilizar un tipo estándar, teniendo siempre en cuenta que los bombones en principio no tienen porque admitir ninguna ordenación.

4. Algoritmo completo a partir del refinamiento del esquema general

Siguiendo los casos previos es tan fácil como transcribirlos a pseudocódigo como sigue:

```

fun pesado ( $v[1..n]$  de Tbombon,  $c, f$ : natural) dev  $\langle b$ : boolean,  $e$ : Tbombon  $\rangle$ 
  si  $c = f$  entonces
     $b \leftarrow$  cierto
     $e \leftarrow v[c]$ 
  si no
     $m \leftarrow (c + f) \text{ div } 2$ 
     $\langle b_1, e_1 \rangle \leftarrow$  pesado ( $v, c, m$ )
     $\langle b_2, e_2 \rangle \leftarrow$  pesado ( $v, m + 1, f$ )
    casos
       $b_1 = \text{falso} \wedge b_2 = \text{falso}$ :  $b \leftarrow$  cierto
       $b_1 = \text{falso} \wedge b_2 = \text{cierto}$ :  $\text{num} \leftarrow$  contar ( $e_2, v, c, f$ )
        si  $\text{num} > (f - c + 1) \text{ div } 2$  entonces
           $b \leftarrow$  cierto;  $e \leftarrow e_2$ 
        si no
           $b \leftarrow$  falso
      fsi
       $b_1 = \text{cierto} \wedge b_2 = \text{falso}$ :  $\text{num} \leftarrow$  contar ( $e_1, v, c, f$ )
        si  $\text{num} > (f - c + 1) \text{ div } 2$  entonces
           $b \leftarrow$  cierto;  $e \leftarrow e_1$ 
        si no
           $b \leftarrow$  falso
      fsi
       $b_1 = \text{cierto} \wedge b_2 = \text{cierto}$ :  $\text{num} \leftarrow$  contar ( $e_1, v, c, f$ )
        si  $\text{num} > (f - c + 1) \text{ div } 2$  entonces
           $b \leftarrow$  cierto;  $e \leftarrow e_1$ 
        si no
           $\text{num} \leftarrow$  contar ( $e_2, v, c, f$ )
          si  $\text{num} > (f - c + 1) \text{ div } 2$  entonces
             $b \leftarrow$  cierto;  $e \leftarrow e_1$ 
          si no
             $b \leftarrow$  falso
          fsi
        fsi
      fsi
    fcasos
  fsi
ffun

```

5. Estudio del coste

Veremos, como en ocasiones anteriores la **ecuación de recurrencia** será:

$$T(n) = \begin{cases} c * n & \text{si } n = 1 \\ 2 * T(n/2) + c * n & \text{si } n \geq 1 \end{cases}$$

La resolución de la recurrencia es:

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

Las variables de la ecuación de recurrencia correspondientes con nuestro problema serán:

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.

Siguiendo el procedimiento habitual, sustituiremos en la ecuación $a = b^k$, siendo éste el caso segundo, por lo que el coste será $\theta(n * \log(n))$.

NOTA (del ejercicio): El coste de contar es lineal.

Febrero 2005-2ª (problema)

Enunciado: Utiliza el esquema de divide y vencerás para implementar una función que tome un vector de enteros y le dé estructura de montículo con el menor coste posible.

Respuesta:

Lo interesante de este problema es que emplean cambios de variables para resolverlo. Nunca antes hemos visto hecho así ningún problema en este tema. Para ello, convendría repasar el tema de costes (tema 4) de Brassard y el de estructuras de datos (tema 5), para así entender mejor el problema. Pasamos a explicarlo a continuación.

1. Elección del esquema

No es necesario razonar la elección del esquema puesto que viene impuesto en el problema.

2. Esquema general

El esquema general de **divide y vencerás** consiste en:

- **Descomponer** el ejemplar en subejemplares del mismo tipo que el original.
- **Resolver** independientemente cada subejemplar (subproblemas).
- **Combinar** los resultados para construir la solución del ejemplar original.

En los problemas de los últimos años se nos da un esquema alternativo que, por curiosidad pasamos a escribirlo, aunque como es habitual tomaremos el nuestro empleado en todos los ejercicios hasta el momento:

```

fun divide-y-vencerás (X)
  si suficientemente-pequeño (X) entonces
    dev subalgoritmo_basico (X)
  si no
    decomponer (X, X1 .. Xn)
    para i = 1 hasta n hacer
      Yi := divide-y-vencerás (pi)
    fpara
    recombinar (Y1 .. Yn, Y)
  dev Y
fsi
ffun

```

El **esquema general** es el siguiente:

```

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

```

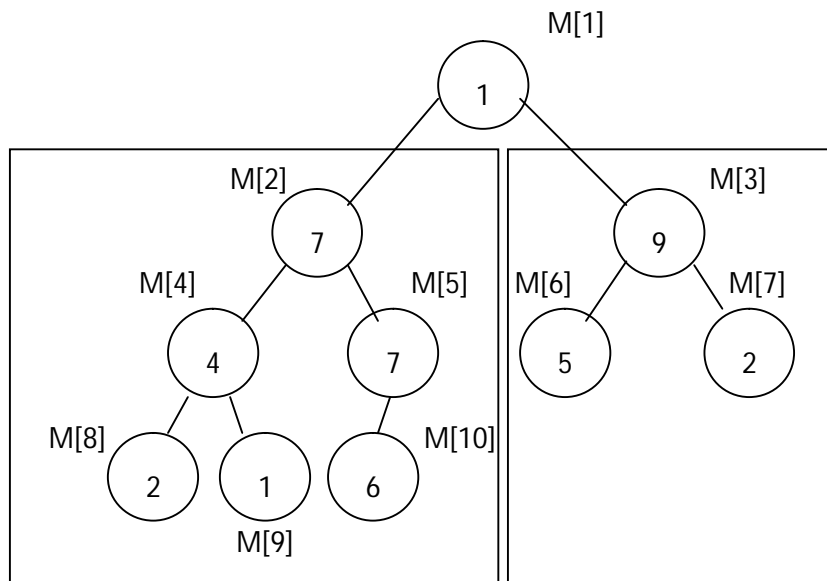
3. Estructuras de datos

No se necesita ninguna estructura adicional a partir del vector de entrada.

4. Algoritmo completo

Como se trata de un árbol binario descomponemos el problema en los dos subárboles bajo el nodo i , es decir, el que tiene por raíz $2 * i$ y el que tiene por raíz $2 * i + 1$. Cada subárbol a su vez debe tener estructura de montículo, lo que resulta un problema del mismo tipo que el original y, por tanto, corresponde a sendas llamadas recursivas, una para $2 * i$ y otra para $2 * i + 1$. Por último, hay que colocar la raíz i en su lugar. Para ello hay que *hundirla*.

Ejemplo:



El algoritmo será, por tanto, el siguiente:

```

proc crear-montículo (M[1..m], i)
  si  $2 * i < m$  entonces
    crear-montículo (M,  $2 * i$ )
  fsi
  si  $(2 * i + 1) < m$  entonces
    crear-montículo (M,  $2 * i + 1$ )
  fsi
  si  $2 * i \leq m$  entonces
    hundir (M, i)
  fsi
fproc
  
```

se ha hecho una versión iterativa del crear montículo algo distinta a como lo vimos en el tema 5, pero igualmente válida aunque pondremos para refrescar la memoria la versión recursiva:

```

procedimiento crear-montículo (T[1..n])
  { Este procedimiento transforma la matriz T[1..n] en un montículo }
  para  $i \leftarrow \lfloor n/2 \rfloor$  bajando hasta 1 hundir (T, i)
  
```

Como puede observarse, dividir el vector en dos mitades $M[1..m/2]$, $M[(m/2) + 1..m]$ y tratar de resolver cada una de ellas supone un error. Una solución que recorra completamente el vector y proceda a *hundir* o *flotar* cada elemento siempre tendrá mayor coste.

Ya que estamos en un esquema de divide y vencerás, podemos plantear el procedimiento hundir como un problema de reducción. Únicamente, se tiene que decidir si el nodo debe intercambiarse por alguno de sus hijos y, en caso afirmativo, intercambiarse con el mayor de ellos y realizar la correspondiente llamada recursiva para seguir *hundiéndose*.

El procedimiento hundir es el siguiente:

```

proc hundir (T[1..n], i)
    hmayor ← i;
    { Buscar el hijo mayor del nodo i }
    si (2 * i ≤ n) y (T[2 * i] > T[hmayor]) entonces
        hmayor = 2 * i;
    fsi
    si (2 * i + 1 ≤ n) y (T[2 * i + 1] > T[hmayor]) entonces
        hmayor = 2 * i + 1;
    fsi
    { Si cualquier hijo es estrictamente mayor que el padre }
    si (hmayor > i) entonces
        intercambiar T[i] y T[hmayor]
        hundir-rec (T, hmayor)
    fsi
fproc

```

5. Estudio del coste

Esta parte es muy importante, ya que si fuera mayor el coste se puntuaría como 0 el ejercicio. El coste del procedimiento *hundir* se puede plantear mediante una **recurrencia con reducción del problema por división**, ya que *hundir* prosigue por uno de los dos subárboles y, por tanto, el problema se ha reducido a la mitad, siendo ésta la ecuación de recurrencia:

$$T(n) = \begin{cases} c * n^k & \text{si } 1 \leq n < b \\ a * T(n/b) + c * n^k & \text{si } n \geq b \end{cases}$$

La resolución de la ecuación de recurrencia es:

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

Las distintas variables son:

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. Tendremos que el valor de $k = 0$, al ser el tiempo extra constante.

Como en ocasiones anteriores y evitando el cálculo de la resolución de la recurrencia, tendremos que el coste $\theta(\log(n))$.

El coste de crear-montículo puede expresarse mediante la siguiente **ecuación de recurrencia**:

$$T(n) = 2 * T(n/2) + \log(n)$$

Sin embargo, esta recurrencia no se puede resolver con la fórmula anterior, puesto que la parte no recursiva no tiene una complejidad polinomial. Para resolverlo, vamos a utilizar un **cambio de variable**:

Sea h la altura del montículo de n nodos: $h = \log_2(n)$. En cada llamada recursiva bajamos un nivel por lo que el problema se puede expresar mediante **una recurrencia con reducción del problema por sustracción**. Es decir, si en cada paso se baja un nivel, el problema se reduce a uno. Igualmente tenemos la ecuación de recurrencia y su resolución, como vimos anteriormente es la siguiente:

$$T(n) = \begin{cases} c * n^k & \text{si } 0 \leq n < b \\ a * T(n - b) + c * n^k & \text{si } n \geq b \end{cases}$$

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

En el caso de *hundir* $T(h) = T(h - 1) + c$, con $a = 1$ y $b = 1$, luego $T(h) \in \theta(h)$ que deshaciendo el cambio de variable lleva un tiempo *en función de n*:

$$T(n) \in \theta(\log(n))$$

Como habíamos demostrado previamente.

Sin embargo, ahora ya podemos plantear la recurrencia para *crear-montículo*:

$$T(h) = 2 * T(h - 1) + h, \text{ donde } a = 2 \text{ y } b = 1, \text{ por tanto, } T(h) \in \theta(2^h).$$

Deshaciendo el cambio de variable:

$2^h = 2^{\log_2(n)} = n$ y $T(n) \in \theta(n)$, que es el menor coste posible para dar estructura de montículo a un vector.

Septiembre 2005 (problema) (parecido a problema Septiembre 2000)

Enunciado: Sea $V[1..n]$ un vector con la votación de una elecciones. La componente $V[1..n]$ contiene el nombre del candidato que ha elegido el votante i . Implementa un programa cuya función principal siga el esquema de divide y vencerás, que decida si algún candidato aparece en más de la mitad de los componentes (tiene mayoría absoluta) y que devuelva su nombre. Sirva como ayuda que para que un candidato tenga mayoría absoluta considerando todo el vector (al menos $N/2 + 1$ de los N votos), es condición necesaria pero no suficiente que tenga mayoría absoluta en alguna de las mitades del vector. La resolución del problema debe incluir, por este orden:

1. Descripción del esquema de divide y vencerás y su aplicación al problema
2. Algoritmo completo a partir del refinamiento del esquema general.
3. Estudio del coste del algoritmo desarrollado.

Respuesta:

En este ejercicio al igual que en algunos otros, veremos los 5 pasos habituales aunque de modo condensado en 3.

1. Descripción del esquema de divide y vencerás y su aplicación al problema

El esquema general de **divide y vencerás** consiste en:

- **Descomponer** el ejemplar en subejemplares del mismo tipo que el original.
- **Resolver** independientemente cada subejemplar (subproblemas).
- **Combinar** los resultados para construir la solución del ejemplar original.

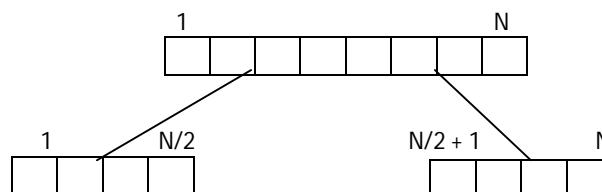
Más formalmente:

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

Vamos a hacer una ampliación del problema propuesto, por lo que añadiremos en este apartado las funciones que se particularizan habitualmente. Realmente este problema es bastante parecido al hecho de los “bombones pesados”, es casi idéntico, ya veremos en qué se diferencian más adelante:

- El problema **suficientemente simple** es 1 voto, por lo que habría mayoría absoluta, sería $V[1]$.
- Según el enunciado para tener mayoría absoluta en N es condición necesaria pero no suficiente que tenga mayoría absoluta en alguna de las mitades del vector (recordemos ese ejercicio, es igual el planteamiento).

La descomposición será, por tanto:



De nuevo, al igual que pasaba antes se pueden dar 4 casos distintos. En todo caso, necesitaremos saber si hay alguien con mayoría absoluta, qué candidato es y además con cuántos votos. Veremos estos casos posibles:

1. **No** hay mayoría absoluta en ninguna mitad, por tanto, no hay mayoría absoluta en el total (es condición del enunciado).
2. Existe un **candidato1 con mayoría absoluta** en la mitad *izquierda* y no hay nadie con mayoría en la mitad derecha (condición necesaria pero no suficiente para que sea mayoría absoluta en el vector). Para ello, hay que hacer lo siguiente:

- Contar los votos de candidato1 en la mitad derecha y sumarlos a los que tiene en la izquierda. Tendremos que averiguar si $\text{num_votos1} \geq \lfloor N/2 \rfloor + 1$.
- 3. Existe un **candidato2 con mayoría absoluta** en la mitad *derecha* y no hay nadie con mayoría en la mitad derecha (condición necesaria pero no suficiente para que sea mayoría absoluta en el vector). Para ello, hay que hacer lo siguiente:
 - Contar los votos de candidato1 en la mitad izquierda y sumarlos a los que tiene en la derecha. Tendremos que averiguar si $\text{num_votos2} \geq \lfloor N/2 \rfloor + 1$.
- 4. El **candidato1 tiene mayoría absoluta en la derecha y el candidato2 lo tiene a la izquierda**, por lo que habría que ver cuál de los dos es el que gana (o ninguno).

2. Algoritmo completo a partir del refinamiento del esquema general

El esquema siguiendo los casos antes expuestos es:

```

fun pesado ( $v[1..n]$  de Tbombon, i, j: natural)
    dev (tiene_mayoria, candidato, num_votos)
  si ( $i = j$ ) entonces
    dev (cierto,  $v[i]$ , 1)
  si no
    /* Descomponer */
    (tiene_mayoria1, candidato1, num_votos1) = contar ( $V, i, (i + j) \div 2$ )
    (tiene_mayoria2, candidato2, num_votos2) = contar ( $V, (i + j) \div 2 + 1, j$ )
    /* Combinar */
    si tiene_mayoria1 entonces // Si mayoría en parte izq.
      para k desde  $(i + j) \div 2 + 1$  hasta j hacer // Cuenta en parte der.
        si es_igual ( $v[k]$ , candidato1) entonces
          num_votos1 = num_votos1 + 1
        fsi
      fpara
      si (num_votos1 >  $(j - i + 1) \div 2$ ) entonces
        devolver (cierto, candidato1, num_votos1)
      fsi
    fsi
    si tiene_mayoria2 entonces // Si mayoría en parte der.
      para k desde 1 hasta  $(i + j) \div 2$  hacer // Cuenta en parte izq.
        si es_igual ( $v[k]$ , candidato2) entonces
          num_votos2 = num_votos2 + 1
        fsi
      fpara
      si (num_votos2 >  $(j - i + 1) \div 2$ ) entonces
        devolver (cierto, candidato2, num_votos2)
      fsi
    fsi
    devolver (falso, "", 0)
  fsi
ffun

```


La **llamada inicial** es contar (V, 1, N).

NOTA DEL AUTOR: Como hemos comentado previamente tenemos una serie de características propias a este código, aunque en el fondo sigue siendo igual al del "bombón pesado" y, por extensión, a los ejercicios de este tipo, que se harían casi iguales. En este caso, hemos añadido una nueva variable que cuenta el número de votos, que como antes vimos lo hacía la función auxiliar *contar*.

3. Estudio del coste del algoritmo desarrollado

Plantearemos la siguiente ecuación de recurrencia:

$$T(n) = 2 * T(n/2) + \frac{n}{2} + \frac{n}{2} = 2 * T(n/2) + n$$

Está ecuación se deduce de esta manera, ya que recorre ambas mitades al contar los distintos candidatos. Es por ello, que al final acaba **recorriendo todo el vector** encontrando al candidato con mayoría absoluta.

Por tanto, la reducción del problema se realiza mediante **división**, cuyos casos son los siguientes:

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

Las distintas 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 con respecto a n.

Por ello y de nuevo, es más de lo mismo, tendremos que el coste es $T(n) = \theta(n * \log(n))$.

3ª parte. Problemas de exámenes sin solución o planteados:

Febrero 1999-2ª (problema 2)

Enunciado: Una liga de n equipos e_i juega a un juego donde solo se pierde o se gana, pero no se empata. Diseñar un algoritmo que realice una *pseudo-ordenación* (e_1, e_2, \dots, e_n) a partir de la tabla de resultados de los equipos de manera que e_1 haya ganado a e_2 , e_2 a e_3 , etc. El coste debe ser a lo sumo $O(n * \log(n))$

Respuesta: Este ejercicio no sé muy bien como se haría, entiendo que al ser una pseudo-ordenación se debería plantear un esquema de divide y vencerás, pero estoy con la duda con respecto a los voraces, debido a que el coste puede ser también correspondiente a una exploración de grafos (recordemos algoritmo de Prim, Kruskal, ...). Esto último al escribir el enunciado en el editor de texto me he dado cuenta que podría llegar a ser eso. Para concluir y en resumen decir que no estoy segura personalmente ni del esquema, aunque lo pondremos en divide y vencerás (que es lo que más se asemeja).

Diciembre 2002 (problema)

Enunciado: Se tienen dos polinomios de grado n representados por dos vectores $[0..n]$ de enteros, siendo la posición i de ambos vectores la correspondiente al término x^i del polinomio. Se pide diseñar un algoritmo que multiplique ambos polinomios, valorándose especialmente que sea con un coste más eficiente que $O(n^2)$ (En concreto, hay una solución fácil de hallar con coste $O(n^{\log_2 3})$).

Respuesta: Este ejercicio se asemeja bastante al ejercicio 2 de Septiembre de 2007, en el que pedían exactamente lo mismo e igualmente lo hemos visto en la teoría, recordando, por tanto, que para tener menos coste resolvemos el problema usando 3 multiplicaciones en vez de 4. Lo único que se ha escrito este enunciado, ya que la solución más exhaustiva (la que vemos en los problemas) no la daremos, aunque sí que lo plantearemos adecuadamente.