1) Sea A un array bidimensional de tamaño $n \times n$ parcialmente ordenado (se supone n potencia de 2). El criterio de ordenación es el siguiente: Los elementos en cada fila y columna se encuentran en orden ascendente, esto es,

- $A[i, j] \leq A[i, j+1]$ con $i = 1, \ldots, n$ y $j = 1, \ldots, n$.
- $A[i, j] \leq A[i+1, j]$ con $i = 1, \ldots, n$ y $j = 1, \ldots, n$.

Se pretende determinar si un elemento está en la matriz. Responder a las siguientes cuestiones:

a) Construir un algoritmo básico (ad-hoc).

b) Resolver de manera eficiente con Divide y Vencerás.

c) Comprobar si los órdenes mejoran.

Matriz de ejemplo,

| | | | |
|---|---|---|---|
| 20 | 25 | 26 | 29 |
| 22 | 27 | 33 | 40 |
| 29 | 39 | 41 | 45 |
| 31 | 43 | 50 | 55 |

→ comienzo del Alg. Básico.

a) Aprovechando la información que tenemos de la matriz podemos ver (sea x el elemento a buscar):

- Si $x = A[i, j] \rightarrow$ encontrado.
- Si $x < A[i, j] \rightarrow$ puedo mirar en $A[i, j-1]$
- Si $x > A[i, j] \rightarrow$ puedo pasar a $A[i+1, j]$

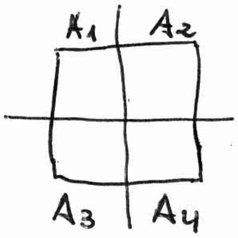
Empezando en la posición $i = 0$ y $j = n-1$ puedo ir descartando filas y columnas hasta encontrar el elemento o llegar a la posición $i = n$ ó $j = -1$.

El algoritmo detallado sería,

```
int busqueda (int n, int x, int **A) {
    int f, c;
    f=0;
    c=n-1;
    while ((f<=n-1) && (c>=0))
        if (x < A[f][c]) c=c-1;
        else
            if (x > A[f][c]) f=f+1;
            else return 1;
    return 0;
}
```

En el peor caso (cuando no se encuentra el elemento), la eficiencia es $2 \cdot n \rightarrow O(n)$.

b) Para aplicar la técnica DV podemos encontrar el elemento medio de la matriz (como se hacía en busqueda binaria) y ver en que parte o partes de la matriz tengo que seguir buscando. De esta forma,




si $x = A[n/2, n/2] \rightarrow$ encontrado.

si $x < A[n/2, n/2] \rightarrow$ hay que buscar en $A_1$, $A_2$ y $A_3$.

si $x > A[n/2, n/2] \rightarrow$ hay que buscar en $A_2$, $A_3$ y $A_4$.

El caso base sería cuando sólo quede un elemento, momento en el cual podré devolver verdadero o falso hacia arriba. Si alguno de los subcasos da verdadero, se devuelve verdadero, si no, falso.

El algoritmo detallado sería,

```
int busqueda (int fini, int cini, int ffin, int cfin, int x, int **A){
    int fmed, cmed;
    if (ffin-fini == 0)
        if (x== A[fini][cini]) return 1;
        else return 0;

    fmed = fini + (ffin-fini+1)/2-1;
    cmed = cini + (cfin-cini+1)/2-1;
    if (x== A[fmed][cmed]) return 1;
    if (busqueda (fmed+1, cini, ffin, cmed, x, A)) return 1;   -> A3
    if (busqueda (fini, cmed+1, fmed, cfin, x, A)) return 1;   -> A2
    if (x < A[fmed][cmed]) {
        if (busqueda (fini, cini, fmed, cmed, x, A)) return 1;   -> A1
    }
    else
        if (busqueda (fmed+1, cmed+1, ffin, cfin, x, A)) return 1;   -> A4
    return Ø;
}
```

En el peor de los casos la función de tiempo está definida por,

$$T(n) = 3\,T(n/2) + 1 \Rightarrow l = 3 > b^k = 2^0 = 1,$$

$$O(n^{\lg_b l}) = O(n^{\lg 3})$$

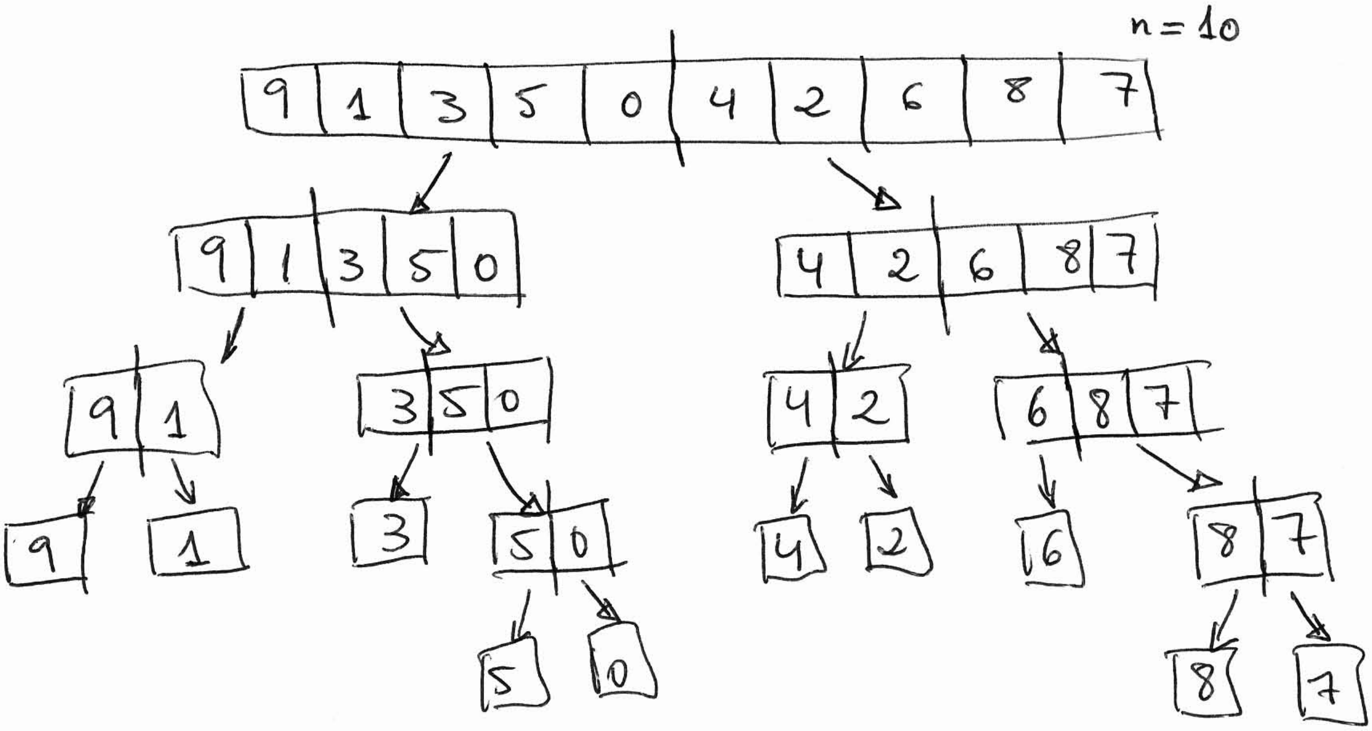
$$\Downarrow$$

El orden no se mejora al aplicar DyV.

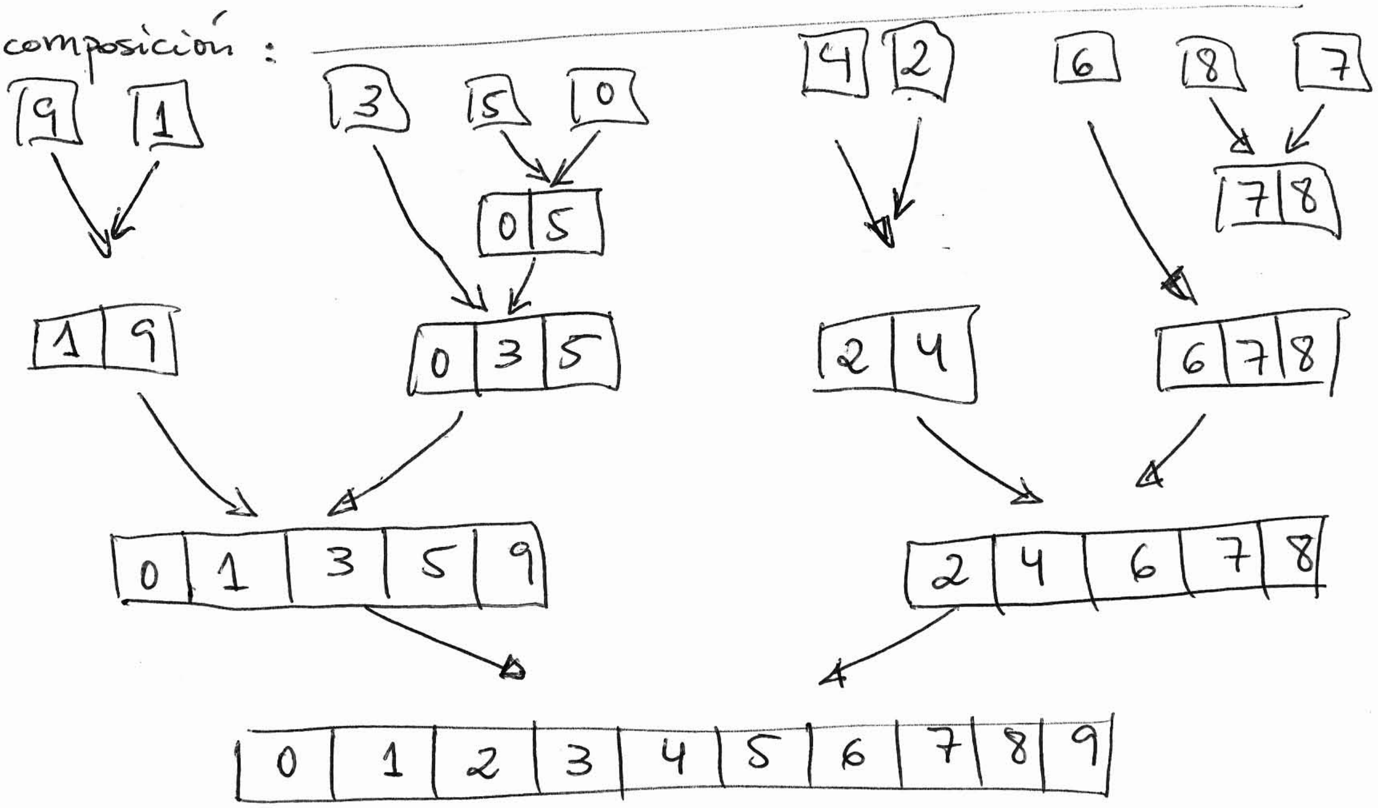
2) Ordenar los siguientes elementos utilizando Mergesort y Quicksort:

9 1 3 5 0 4 2 6 8 7

Merge:




Recomposición:



Quicksort:

pivote

| 9 | 1 | 3 | 5 | 0 | 4 | 2 | 6 | 8 | 7 |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

posiciones del vector

(elijo como pivote el elemento medio del vector)
(utilizo pivoteo lineal con dos índices e incluyendo el pivote como elemento de la parte de la derecha).

balanceo

| 9 | 1 | 3 | 5 | 0 | 4 | 2 | 6 | 8 | 7 |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

izqda. vacía

vuelvo a tener el mismo vector y se repite la situación.

Necesito cambiar algo (la elección del pivote junto con el balanceo utilizado no asegura que se reduzca el vector en todos los casos). Soluciones:

- Excluir el pivote del balanceo, sacándolo al principio y luego llevándolo a su posición.
- Elegir un pivote que asegure que haya al menos un elemento en la parte de la izquierda (el mayor de los dos primeros).

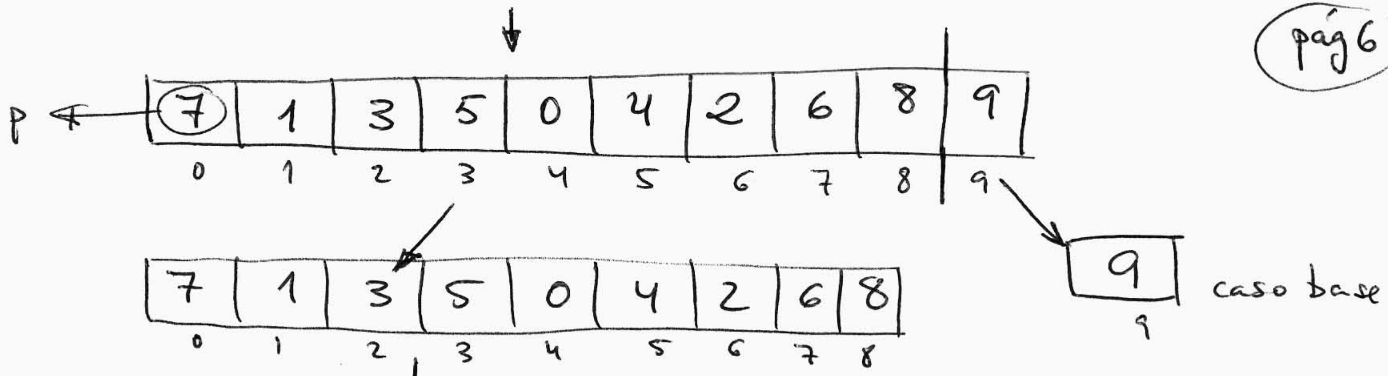
Considerando la segunda opción tendríamos:

p

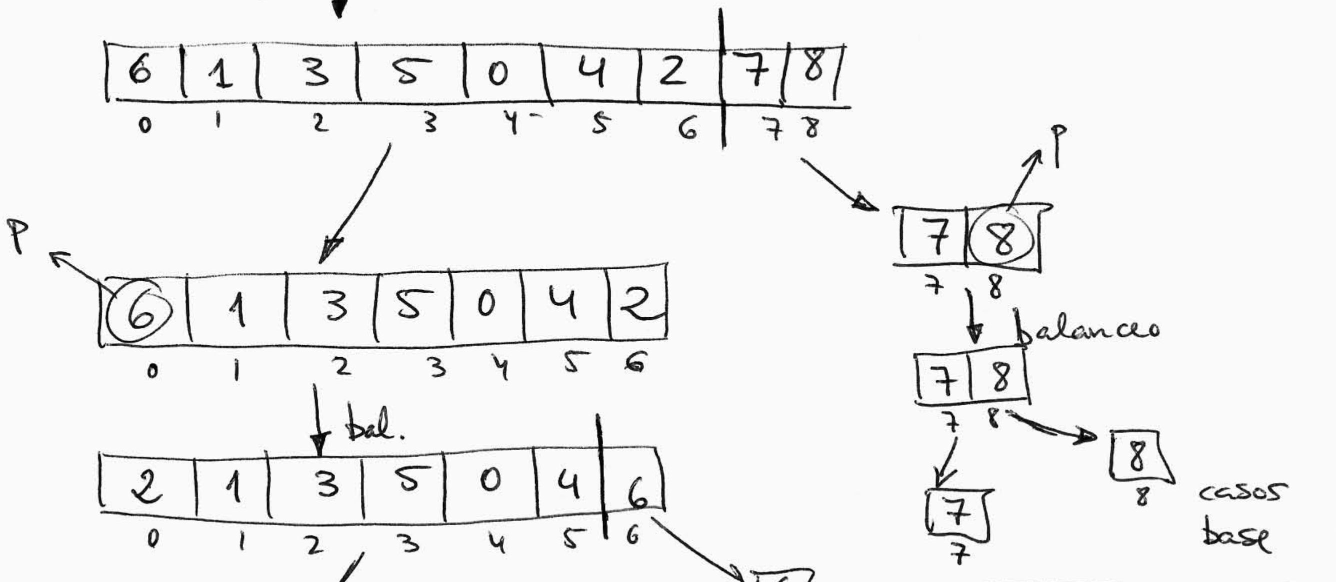
| 9 | 1 | 3 | 5 | 0 | 4 | 2 | 6 | 8 | 7 |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

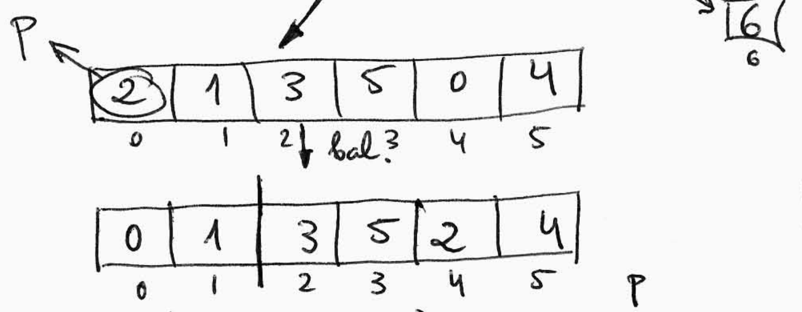
balanceo

- El puntero drecha. para en el 7.
- El puntero izqda. para en el 9.
- Intercambio el 7 con el 9.
- drcha. para en 8.
- Izqda. se cruza con drcha.

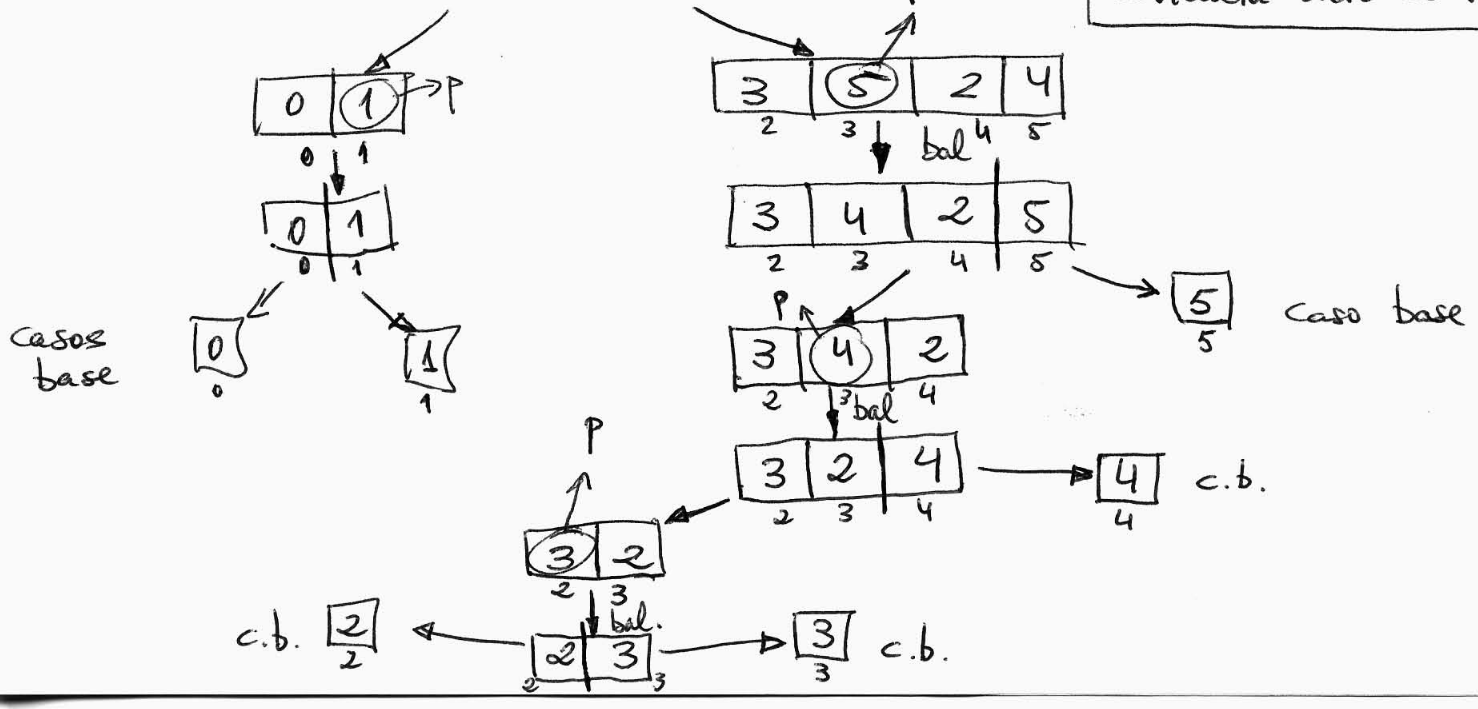



- drcha. para en 6.
- Izqda. en 7.
- Intercambio 7 y 6.
- Drcha. para en 2
- Izqda. se cruza con drcha.




Una forma inteligente para hacer el ejercicio más rápido hubiese sido indicar que el pivote se escoge de forma aleatoria e ir eligiendo un elemento que dividiera bien el vector.



(3) Encontrar la mediana de un vector de tamaño 
n. Se supone que los elementos no están repetidos.
En el caso de que n sea par, se considerará como mediana el elemento anterior a la mitad del vector.

El algoritmo básico podría consistir en ordenar el vector y quedarnos con el elemento medio. Tendría orden $O(n \cdot \lg n)$.

Para aplicar D y V escogeríamos un pivote para balancear los elementos a derecha e izquierda como en Quicksort. Seguiríamos buscando únicamente por la parte en la cual quedara el elemento medio. Una vez llegado al caso base, devolveríamos dicho elemento como valor de la mediana (el caso base se produce cuando sólo queda un elemento por explorar).

El algoritmo detallado sería,

```
int n_esimo (int pos, int ini, int fin, int *V) {
    int l;
    if (fin-ini == 0) return V[ini];
    l = pivot (ini, fin, V);
    if (l == pos) return V[l];
    if (pos > l)
        return n_esimo (pos, l+1, fin, V);
    else
        return n_esimo (pos, ini, l-1, V);
}
```

(porque el pivote no se va a incluir ni en la parte drcha. ni en la parte izqda.)

Se llamaría a la función de la siguiente manera:

mediana = n_esimo (n/2, 0, n-1, V);

Véase que con esta función se puede obtener el elemento que esté situado en cualquier posición.

La función pivot habría que modificarla para que devolviera la posición en la que queda el pivote. Quedaría de la siguiente manera,

```
int pivot (int ini, int fin, int *V) {
    int p, K, l, temp;
    p= V[ini];
    K = ini;
    l = fin+1;
    do {
       K++
    }while(V[K]<=p && K<fin);
    do {
       l--;
    }while(V[l]>p);
    while (K<l) {
        temp = V[K];
        V[K]= V[l];
        V[l]= temp;
        do {
           K++;
        }while(V[K]<=p);
        do {
           l--;
        }while(V[l]>p);
    }
    temp= V[ini];
    V[ini]= V[l];
    V[l] = temp;
    return l;
}
```

El orden se calcularía considerando $T(n) = T(n/2) + n$

$l = 1,\ K = 1,\ b = 2 \Rightarrow l < b^K \rightarrow 1 < 2^1 \Rightarrow$ el orden es $O(n^1)$

④ (Ej. 1 del examen de Febrero de 2007) (1'5 pt.) 

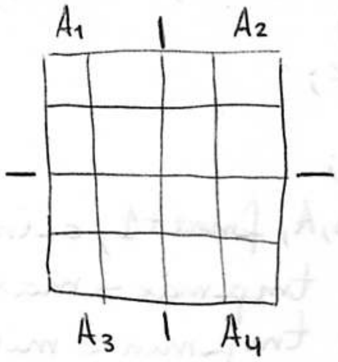
Dada una matriz cuadrada de enteros de dimensión $n \times n$, se desea encontrar el máximo y mínimo de la misma.

a) Diseñar un algoritmo basado en la técnica "Divide y Vencerás" para resolver este problema.

b) Calcular la eficiencia del algoritmo en el peor caso.

Suponer que $n$ es potencia de algún número entero conveniente.

a) Supongamos $n$ potencia de 2. La matriz se puede dividir en 4 matrices de tamaño $n/2$. Ej:

$A =$ 


El caso base se daría cuando la matriz tenga tamaño 1. En ese caso se devolvería el único valor disponible como máximo y como mínimo.

Para recomponer a partir de 4 subproblemas nos quedamos con el máximo y el mínimo de los 4 valores propuestos para cada subproblema. Desgraciadamente no se puede decrementar el número de llamadas, ya que los valores pueden estar en cualquiera de las partes.

El algoritmo detallado sería

```
void  MaxMinDV (int & max, int & min, int **A,
                int f_ini, int c_ini, int f_fin, int c_fin) {
                // se numera de 0 a n-1
    int f_med, c_med, tmp_max, tmp_min;
    if (f_fin - f_ini == 0) {  // caso base
        max = min = A[f_ini][c_ini];
        return;
    }
    f_med = (f_ini + f_fin) / 2;  // división entera
    c_med = (c_ini + c_fin) / 2;

    MaxMinDV(tmp_max, tmp_min, A, f_ini, c_ini, f_med, c_med); // A1
    MaxMinDV(max, min, A, f_ini, c_med+1, f_med, c_fin); // A2
    if (max > tmp_max)
        tmp_max = max;
    if (min < tmp_min)
        tmp_min = min;
    MaxMinDV(max, min, A, f_med+1, c_ini, f_fin, c_med); // A3
    if (max > tmp_max) tmp_max = max;
    if (min < tmp_min) tmp_min = min;
    MaxMinDV(max, min, A, f_med+1, c_med+1, f_fin, c_fin); // A4
    if (tmp_max > max) max = tmp_max;
    if (tmp_min < min) min = tmp_min;
    return;
}
```

b) La ecuación de tiempo recurrente se puede establecer de la siguiente manera:

$$T(n) = 4\,T(n/2) + 1,$$

$$l = 4,\ K = 0,\ b = 2 \Rightarrow l > b^K \rightarrow 4 > (2^0 = 1) \Rightarrow$$

$$\Rightarrow \text{el orden es } O(n^{\lg 4}) = O(n^2)$$

Este orden no mejora al del algoritmo básico (recorrer todos los elementos por filas y columnas) que es $O(n^2)$.

Para realizar la llamada se puede utilizar una función que haga de interfaz:

```
void MaxMin (int & max, int & min, int ** A, int n) {
    MaxMinDV (max, min, A, 0, 0, n-1, n-1);
    return;
}
```

(5)

(Ej. 4 del examen de Septiembre de 2007) (2 pt.).
Dado un vector T [1..n] se dice que un elemento del vector x es mayoritario si aparece en el vector un número de veces estrictamente mayor de $n/2$. Diseñar un algoritmo para decidir si dado un vector tiene un elemento mayoritario y, en su caso, indicar cuál es ese elemento.

Para resolver este problema se puede proceder de distintas formas:

1) Un algoritmo básico consistiría en recorrer el array y para cada elemento contar el número de ocurrencias de dicho elemento. Sólo habría que llegar hasta la posición $(n+1)/2$, quedándonos con el elemento que más ocurrencias tiene y devolviéndolo si el número de ocurrencias es mayor que $n/2$. Este algoritmo es $O(n^2)$.

2) Reformular el problema en dos partes: primero identificar el único elemento que podría ser mayoritario, segundo contar sus ocurrencias para ver si lo es. Se basa en la idea de que el elemento mayoritario debería estar en la posición $(n+1)/2$

si el vector estuviera ordenado. Se puede resolver de dos maneras:

a) Ordenar con Quicksort y contar el número de ocurrencias del elemento que hay en la posición $(n+1)/2$. Sería $n . \lg n + n \Rightarrow O(n . \lg n)$. Este algoritmo mejora la eficiencia del algoritmo básico.

b) Sin embargo, hay una forma mejor de proceder. Recordemos que se propuso un algoritmo DyV para calcular la mediana de un vector (ejercicio 3, pág. 7). En realidad el algoritmo que se propuso era capaz de calcular el elemento n-ésimo de cualquier vector. Bastaría con usar dicha función para calcular el elemento $(n+1)/2$-ésimo y contar el número de ocurrencias de dicho elemento. Sería $n + n \Rightarrow O(n)$.

```
int mayoritario (int & x, int * T, int n) {
    int cont = 0, i;
    x = n_esimo ((n+1)/2, 0, n-1, T);
    for (i=0; i<n; i++)
        if (T[i]==x)
            cont++;
    if (cont > n/2)
        return 1;
    return 0;
}
```

Para responder a este ejercicio habría que indicar o describir cómo se diseña el algoritmo n_esimo.