Especificación: (2 puntos)
Especificar un algoritmo que tiene como entrada un array de enteros con 2*N componentes y como salida un valor booleano que es TRUE si en el array aparece cada uno de los números 1, 2,…, N, dos veces; y además para cada valor i= 1, 2,.., N, entre las dos apariciones de i hay i componentes.
Ejemplos:
Para N=2 es imposible [2, 1, 1, 2];
Para N=3 un array válido es [3, 1, 2, 1, 3, 2]

Solución:

Primera Parte: explicación informal
El array de 2*N componentes, debe cumplir dos condiciones:

- la primera es que cada número desde 1 hasta N aparezca dos veces en el array y
- la segunda es que entre cada dos apariciones de cada número i, desde 1 hasta N, haya i huecos.

Vamos a hablar de la primera condición de forma extendida:

El número de veces que aparece el 1 en el array es 2
El número de veces que aparece el 2 en el array es 2
El número de veces que aparece el 3 en el array es 2
….
El número de veces que aparece el N en el array es 2

Y esto se expresa de manera más resumida diciendo que para todo i desde 1 hasta N, el número de veces que aparece el i en el array es 2. Asi pues habrá que utilizar el símbolo $\forall$ y también la función *N,* que expresa el número de veces que se cumple una propiedad (Nota: esta condición se ha visto en un ejercicio en clase).

Vamos a hablar de la segunda condición de forma extendida:

Si aparece el 1 en la posición p y aparece el 1 en una posición q, y p está después de q, entonces entre p y q hay un hueco, ósea p = q+2 (en la posición q está el 1, en la posición q+1 no me importa lo que hay, en la posición q+2 hay otro 1).

Si aparece el 2 en la posición p y aparece el 2 en una posición q, y p está después de q, entonces entre p y q hay 2 huecos, ósea p = q+3 (en la posición q está el 2, en la posición q+1 no me importa lo que hay, en la posición q+2 no me importa lo que hay, en la posición q+3 hay otro 2).

….
Si aparece el N en la posición p y aparece el N en una posición q, y p está después de q, entonces entre p y q hay N huecos, ósea p = q+N+1 (en la posición.

Esto se puede expresar de forma abreviada diciendo que para todo j desde 1 hasta N, si hay una posición p, entre 1 y 2N, que tiene el valor j, y hay una posición q desde 1 hasta 2N, que tiene el mismo valor j, y p está después de q, entonces entre p y q hay j huecos.
(Nota: está condición sola, se vió en un ejercicio en clase).

Segunda Parte: expresión formal

Pre ≡ {int v[1...2*N] }

Post≡ { b= { ($\forall$i$\in${1..N} ($N_{k=1}^{2*N}$(v[k]=i) =2):)) $\wedge$
($\forall$j$\in${1..N}: $\exists$p$\in${1..2*N}, $\exists$q$\in${1..2*N}: v[p]=j $\wedge$ v[q]=j $\wedge$ p>q $\Rightarrow$ p-q= j+1) }

Análisis de complejidad (2 puntos)

Resolver la siguiente ecuación de recurrencia

$$\boldsymbol{T(n)} = \begin{cases} \mathbf{1} & \boldsymbol{si\ n = 4} \\ \boldsymbol{T\left(\frac{n}{4}\right) + \sqrt{n} + 1} & \boldsymbol{si\ n > 4} \end{cases}$$

a) por el Teorema maestro y
b) por el método del polinomio característico,

Solución:

a) Por el Teorema Maestro: a=1, b=4, d=1/2, a<b^d , entonces T(n)$\in\theta(n^{1/2})$
b) Por el Polinomio característico:
   Hacemos el cambio n= 4^k
   Resulta: T(4^k) = T(4^{k-1}) + 2^k + 1

   Llamando F(k) = T(4^k) , tenemos que F(k-1) = T(4^{k-1})

   Y queda:

   F(k) = F(k-1) + 2^k + 1 = F(k-1) + 1* 2^k + 1 * 1^k

   Entonces el polinomio es (x-1)(x-2)(x-1)

   Es decir: raíz 1 con multiplicidad 2, raíz 2 con multiplicidad 1.

Las soluciones parciales son 1^k, $k*1^k$, 2^k

Y la solución completa es $F(k) = a*1^k + b* k*1^k + c\ 2^k$

Es decir: $F(k) = a + b* k + c\ 2^k$

Ahora hace falta formar un sistema de tres ecuaciones:

$F(1) = T(4^1)=T(4) = 1$

$F(2) = F(1) + 2^2 + 1 = 6$

$F(3) = F(2) + 2^3 + 1 = 15$

Entonces:
(1) a + b + 2c = 1
(2) a + 2b + 4c = 6
(3) a + 3b + 8c = 15

A (3) le resto (2): b + 4c = 9 (4)
A (2) le resto (1): b + 2c= 5 (5)

Ahora resto (4) – (5): 2c = 4, c=2
Sustituyo en (5): b=1
Sustituyo en (1): a= -4

Entonces $F(k) = -4 + k + 2\ 2^k$

Ahora deshago el cambio, sabiendo que $n= 4^k$ resulta que

$\boldsymbol{2^k = \sqrt{n}\ ;\ k = log_4 n}$

$T(4^k) = -4 + k + 2\ 2^k$

$T(n) = -4 + \boldsymbol{log_4 n + 2\sqrt{n}} \in \theta(n^{1/2})$

Ejercicio de Teoría de Complejidad : ver las diapositivas de clase**.**

CLASES TEMPORALES

Clase P. Problemas resolubles en tiempo polinómico.

Son tratables porque pueden ser abordados en la práctica. Los problemas que no pertenecen a la clase P se denominan problemas intratables.

Clase NP. Problemas intratables pero donde existe un algoritmo polinómico que indica si una solución dada es válida o no.

Se utilizan heurísticas, y en un tiempo polinómico se calcula la validez de la solución obtenida (No-deterministas polinómicos).

Clase EXP. Problemas resolubles en tiempo exponencial.

CLASES ESPACIALES

La complejidad espacial se encarga de estudiar el coste de memoria principal que los algoritmos necesitan para su funcionamiento.

En ocasiones el coste de almacenar la entrada también es ignorado, puesto que será el mismo para todos los algoritmos que resuelvan el mismo problema.

El acceso a bases de datos o datos en memoria secundaria dependerá de la cantidad de esa información que se guarda en memoria principal.

Clase L. Problemas resolubles utilizando espacio logarítmico respecto a la entrada.

Clase PSPACE. Problemas resolubles utilizando espacio polinómico.

$L \subseteq P \subseteq NP \subseteq PSPACE \subseteq EXP$

Divide y Vencerás (5 puntos)

Dado un array A, ordenado creciente, de números naturales, todos distintos, se pide encontrar el menor elemento que no está.

Ejemplos:
,
Input: A[] = [0, 1, 2, 6, 9, 11, 15]
Output: El menor elemento omitido es 3

Input: A[] = [1, 2, 3, 4, 6, 9, 11, 15]
Output: El menor elemento omitido es 0

Input: A[] = [0, 1, 2, 3, 4, 5, 6]
Output: El menor elemento omitido es 7

Solución:

a) Desarrollar un algoritmo de fuerza bruta (1 punto)

```
IF A[0] > 0 {
minOmitido = 0;
} else {
i=1;

WHILE ( (i< A.length) && (A[i] =i) ) {
i++;
}
minOmitido = i;
}
```

b) Calcular su complejidad de forma razonada (0,5 puntos)

En el caso peor, el bucle WHILE se ejecuta tantas veces como la longitud del array T(n) = 2+ 6*n + 1.
Luego T(n) $\in \Theta(n)$

c) Desarrollar un algoritmo por la técnica de Divide y Vencerás (3 puntos)

Razonamiento:
Sea n +1 la longitud del array A con índices de componentes desde 0 hasta n.

Veamos primero los casos base:

Si A[0] > 0, todas las componentes del array son números mayores que 0. Entonces, ¿Cuál es el menor que no está? Pues el 0.
Sino, si A[n]= n, como tienen que ser todos distintos y están en orden creciente, esto obliga a que estén todos desde el 0. Entonces el primero omitido es n+1.
En el resto de los casos A[0]=0 y A[n]>n, lo que obliga a que se ha omitido uno, o muchos, en medio.

Pero igual que A[n]=n obliga a que estén todos, A[i]=i, obliga a que estén todos los anteriores a i. Luego hay que buscar la primera vez que A[i]>i, y entonces el resultado será i (piénsalo un poco).

Conclusión: me sitúo en la componente mitad si A[med]= med, entonces lo que busco está hacia arriba. Si A[med]>med, lo que busco está hacia abajo. Tengo que repetir el proceso hasta que me quede solo uno. Entonces ocurrirá que A[med]> med y A[med-1] = med-1, con lo cual el resultado es med.

En resumen: una variante de búsqueda binaria.

El siguiente código integra la versión fuerza bruta, iterativa y recursiva.

FB = Fuerza Bruta; Iter= iterativo; Rec= Recursivo

```
class Ada {
// Desarrollado por el Prof. Ricardo Conejo. Mi agradecimiento
    public static int menorOmitidoFB(int[] a) {
        int minOmitido;
        int i=0;
        if (a[0]>0) {
            minOmitido = 0;
        } else {
            i = 1;
        }
        while ( (i<a.length) && (a[i]==i) ) {
            i++;
        }
        minOmitido = i;
        return minOmitido;
    }

    public static int menorOmitidoIter(int[] a) {
        int minOmitido;
        int n = a.length;
        if (a[0] > 0) {
                    minOmitido = 0;
        } else if (a[n-1] == n-1) {
                    minOmitido = n;
        } else {
            int izda = 0,dcha = a.length-1;
            int medio = (izda + dcha)/2;
            while (dcha-izda >= 1){
```

```
                if (a[medio] == medio) {
                    // el que falta está a la derecha
                izda = medio + 1;
                }  else {
                    // el que falta está a la izquierda y puede
ser ese
                    dcha = medio;
                }
                medio = (izda + dcha)/2;
            }
            minOmitido = izda;
        }
        return minOmitido;
    }

    // Función para encontrar el menor elemento omitido
    // en un array ordenado de naturales distintos
    public static int menorOmitidoRec(int[] a, int inf, int sup)
{
        int minOmitido;
        int n = a.length;
        // caso base
        if (a[inf] > inf) {
                    minOmitido = inf;
        } else if ( a[sup] == sup ) {
                    minOmitido = sup+1;
        } else if  (sup-inf <= 1) {
                    minOmitido=inf;
        } else {
            int med = (inf + sup) / 2;
            // si med coincide con su valor, el que falta  está¡
a la derecha
            if (a[med] == med) {
                minOmitido = menorOmitidoRec(a, med+1, sup);
            } else {
                  // el que falta está¡ a la izquierda
                  minOmitido = menorOmitidoRec(a, inf, med-1);
              }
        }
        return minOmitido;
    }

    /** Main */

    public static void main(String args[]) {
        int[] a = { 0, 1, 2, 4, 5, 6 };

        System.out.println("El menor elemento omitido es "+
menorOmitidoFB(a));
        System.out.println("El menor elemento omitido es "+
menorOmitidoIter(a));
        System.out.println("El menor elemento omitido es " +
menorOmitidoRec(a, 0, a.length - 1));

    }
}
```

d) Calcular su complejidad de forma razonada (0,5 puntos)

Análisis de la versión iterativa:

Aparte de los casos base que consumen un tiempo C, en cada ejecución del bucle el intervalo se reduce a la mitad:

Ejecución 1: tamaño intervalo n
Ejecución 2: n/2
Ejecución 3: $n/ 2^2$

Ejecución k: $n/ 2^{k-1}$

…. Y acabamos cuando: $n/ 2^{k-1} = 1$; $n = 2^{k-1}$; $k-1 = \log n$; $k = 1 + \log n$

Entonces $T(n) = C + (1+\log n) D$

Donde D es el tiempo consumido dentro del bucle (los if y las asignaciones)

Por tanto: $T(n) \in \Theta(\log n)$

Análisis de la versión recursiva:

Como en cada ejecución el tamaño del problema se reduce a la mitad, mas ciertos pasos que son fijos, tenemos que $T(n) = T(n/2) + k$.

Aplicando (el estudiante debe hacerlo de forma explícita) el Teorema Maestro resulta $T(n) \in \Theta(\log n)$