Examen final – 12 de mayo de 2016

Apellidos, nombre _____ NIF: _____

Pregunta 1 (1 p.)

Responde a las siguientes preguntas (Nota: no utilices calculadora y responde con un valor exacto):

a) (0,5 puntos) Si la complejidad de un algoritmo es $O(n^2 \log_2 n)$, y dicho algoritmo toma 3 segundos para n=2, calcula el tiempo que tardará para n=4.

$n_1 = 2 \; – t_1 = 3$ segundos

$n_2 = 4 \; – t_2 = ?$

$n_1 * k = n_2 \Rightarrow k = n_2/n_1 = 4/2 = 2$

$t_2 = (n_2^2 \log_2 n_2 / n_1^2 \log_2 n_1) * t_1$

$t_2 = 4^2/2^2 * (\log_2 4 / \log_2 2) * 3$

$\mathbf{t_2} = 4 * 2 / 1 * 3 =$ **24 segundos**

b) (0,5 puntos) Considere un algoritmo con complejidad $O(2^n)$. Si para t = 4 segundos el método pudiera resolver un problema con un tamaño de n = 1000, ¿cuál podría ser el tamaño del problema si dispusiéramos de un tiempo de 8 minutos y 32 segundos?

t1= 4 n1= 1000

t2= 8*60+32=512 n2 = ?

$t_2 = Kt_1 \Rightarrow K = \frac{t_2}{t_1} = \frac{512}{4} = 128$

$f(n_2) = \frac{t_2}{t_1} \cdot f(n_1) \Rightarrow n_2 = f^{-1}(\frac{t_2}{t_1} \cdot f(n_1))$

$\log_2(\frac{512}{4} \cdot 2^{1000}) = \log_2(128 \cdot 2^{1000}) = \log_2(2^7 \cdot 2^{1000}) = \log_2 2^{1007} \rightarrow$ **n2 = 1007**

Pregunta 2 (1 p.)

Teniendo en cuenta los siguientes métodos, indica sus complejidades y explica cómo las has calculado:

a) (0,5 puntos) Complejidad temporal

```
public static String buy(int price, int...b){
    int sum=0; String stat="IMPOSSIBLE";
    for (int i=0;i<b.length;i++)
        sum=sum+b[i];
    if(sum==price)
        return "POSSIBLE";
    if(b.length>1){
        stat=buy(price,Arrays.copyOfRange(b,0,b.length-1));
        stat=buy(price,Arrays.copyOfRange(b,1,b.length));
    }
    return stat;
}
```

Divide y vencerás por sustracción con **a = 2; b = 1; k = 1**; Como a > 1 la complejidad sigue la forma **$O(a^{n/b})$ => $O(2^n)$**

b) (0,5 puntos) Complejidad temporal

```
public void metodo2(int n) {
      if (n <= 5) a= n/2;
      else {
            for (int i = 3; i <= n; i++)
                  for (int j = n-2; j >= 0; j--) {
                        b= i+j;
                  }
            for (int k = 0; k < 9; k++)
                  metodo2(n/3);
      }
}
```

Divide & Vencerás por división con **a = 9, b =3 y k = 2,** ¡ojo! el bucle k hace que se realicen 9 llamadas recursivas. Como a = b^k entonces la complejidad es **$O(n^k \log n)$ = $O(n^2 \log n)$**

Pregunta 3 (2 p.)

Necesitamos procesar un gran número de ficheros en nuestro ordenador para realizar diferentes tareas. Para realizar esto, tenemos una implementación secuencial de un algoritmo usando Java:

```
class ProcesarFicheros {
      List<File> fichJava = null;

      public ProcesarFicheros(List<File> fichJava) {
            this.fichJava = fichJava;
      }

      protected void processFiles() {
            for (File fich : fichJava){
               //*** realización de las tareas ***
            }
      }
}
```

Para probar el código, hemos realizado el siguiente caso de prueba:

```
@Test
public void executeTask() {
    List<File> fichJava = new ArrayList<File>();
    File sourceDir = new File("C://myFolder");
    for (File fich : sourceDir.listFiles())
      fichJava.add(fich);

    FileProcessing problems = new FileProcessing(fichJava);
    problema.processFiles();
}
```

La prueba funciona, pero como todos los ficheros son independientes, podríamos hacer la misma operación usando un algoritmo paralelo y así mejoraríamos el tiempo de ejecución. Por tanto, hemos decidido usar el *framework Fork/Join* disponible desde la versión Java 1.7.

a) (1.5 puntos) Reescribir la clase anterior "ProcesarFicheros" para que se beneficie del *framework Fork/Join* para procesar los ficheros. Llamar a la nueva clase "ProcesarFicherosParalelo".

Para abordar esta cuestión hay que combinar dos cosas:

1. La conversión del algoritmo dado, que es secuencial, en un algoritmo que permita tratar en partes disjuntas el conjunto de ficheros indicado, para ello una buena opción es convertirlo a un DV y vencerás, como haríamos con el recorrido secuencial de un vector.
2. La implementación propiamente dicha del framework Fork/Join

Hay que tener en cuenta que la llamada inkokeAll que se encarga de generar los hilos y esperar a que acaben todos necesita recibir como parámetros las instancias concretas de la clase que queremos paralelizar, lo normal es que sean dos. No se puede pasar un vector de instancias. Si sólo pasamos una no se paralelizará el algoritmo.

```
class FileProcessingTask extends RecursiveAction {
	private static final int THRESHOLD = 5;
	private List<File> javaFiles = null;

	public FileProcessingTask(List<File> javaFiles) {
		this.javaFiles = javaFiles;
	}

	@Override
	protected void compute() {
	  if (javaFiles.size() <= THRESHOLD) {
		processFiles();
	  }
	  else {
		int center = javaFiles.size() / 2;
		List<File> part1 = javaFiles.subList(0, center);
		List<File> part2 = javaFiles.subList(center, javaFiles.size());
		invokeAll(new FileProcessingTask(part1),
			new FileProcessingTask(part2));
	  }
	}

	protected void processFiles() {
		for (File file : javaFiles){
			//***lots of things done here***		}
	}
}
```

b) (0.5 puntos) Reescribir el caso de prueba previo para poder probar el algoritmo Fork/Join que se ha creado.

Tenemos que crear una instancia para el conjunto de ficheros y otra instancia para ForkJoinPool (no hace falta indicar el número de hilos, lo calcula automáticamente).

```
@Test
public void executeTask() {
    List<File> javaFiles = new ArrayList<File>();
    File sourceDir = new File("C://windows");
    for (File file : sourceDir.listFiles())
       javaFiles.add(file);

    FileProcessingTask problem = new FileProcessingTask(javaFiles);
    ForkJoinPool pool = new ForkJoinPool();
    pool.invoke(problem);
}
```

Pregunta 4 (2 p.)

Supongamos que estamos desarrollando un videojuego, en el cual los jugadores pueden crear su propia ciudad situando construcciones en diferentes puntos del mapa. El videojuego tiene un motor de inteligencia artificial para saber cómo de "bueno" es un jugador. Basándose en esa información, el videojuego puede realizar diferentes acciones como proporcionar más o menos ayuda, generar desastres naturales, etc.

Para gestionar esta funcionalidad, el juego utiliza internamente una tabla basada en los recursos manejados por el jugador. En la tabla, se incluye es coste de crear una nueva construcción en una localización específica y los puntos que puede proporcionar al jugador. Debajo se muestra una vista de la tabla manejada por el sistema durante el juego.

	Hospital en situación A	Hospital en situación B	Escuela en situación A	Escuela en situación B	Área residencial en situación C
Coste	3	4	5	7	5
Puntuación	6	12	13	26	5

Usando esta información, cuál podría ser la mejor puntuación posible para el jugador en un momento dado, si cuando está jugando tiene un total de 10 recursos para gastar en la ciudad. Resuelve el problema con la técnica de programación dinámica.

Esto es exactamente el problema de la mochila (sin fragmentación). Donde los objetos que tenemos son las construcciones y el límite de la mochila es el dinero disponible para invertir. La máxima puntuación posible en esta situación es de 32. Para calcularlo desarrollamos la siguiente tabla:

Max coste / Construc.	0	1	2	3	4	5	6	7	8	9	10
1. Hosp. A	0	0	0	6	6	6	6	6	6	6	6
2. Hosp. B	0	0	0	6	12	12	12	18	18	18	18
3. Esc. A	0	0	0	6	12	13	13	18	19	25	25
4. Esc. B	0	0	0	6	12	13	13	26	26	26	32
5. Resid. C	0	0	0	6	12	13	13	26	26	26	**32**

Pregunta 5 (2 p.)

El problema del salto del caballo consiste en recorrer un tablero de ajedrez completo (8x8) a base de movimientos como los que realiza el caballo de este juego. Se quiere buscar al menos una forma de recorrer el tablero que dé solución al problema utilizando la técnica de *backtracking* (vuelta atrás).

a) (1,5 puntos) Completar el código Java para dar solución a este problema utilizando la técnica de *Backtracking* (escribirlo en los espacios preparados a tal efecto).

```
public class AjedrezCaballo
{
        static int n;                   // tamaño tablero
        static int[][] tab;             // tablero de ajedrez
        static int[] hor= {1, 2, 2, 1, -1, -2, -2, -1};// desp. horizontal
        static int[] ver= {2, 1, -1, -2, -2, -1, 1, 2};// desp. vertical
        static boolean seEncontro= false;       // solución encontradas

        static void backtracking (int salto,int x,int y)
        {
                if (salto==n*n+1)
                {
                        seEncontro=true;
                        imprimirEstado(salto);
                }
                else
                        for (int k=0; k<=7; k++)
                        {
                                int u= x+hor[k];
                                int v= y+ver[k];

                                if (!seEncontro &&
                                u>=0 && u<=n-1 && v>=0 && v<=n-1 && tab[u][v]==0)
                                {
                                        tab[u][v]=salto;
                                        backtracking (salto+1,u,v);
                                ➔       if (!seEncontro)
                                                tab[u][v]=0;
                                }
                        }
        }

        public static void main (String arg[])
        {
                n= Integer.parseInt(arg[0]);
                int salidax= Integer.parseInt(arg[1]);
                int saliday= Integer.parseInt(arg[2]);

                tab= new int [n][n];        // Crea la matriz del tablero
                tab[salidax][saliday]= 1;  // posición inicial del caballo

                backtracking(2,salidax,saliday);

                if (!seEncontro)
                        System.out.println ("NO HAY SOLUCION");
        }
```

b) (0,5 puntos) Qué código se debe añadir y dónde (señalar con un flecha), para poder imprimir el estado solución encontrado en el método *main()* después de salir del método *backtracking()*.

Si no se modifica el código respecto al que hay la solución se imprime dentro del método recursivo de backtracking; pero al volver de las llamada recursivas se va borrando el array del estado con la solución que se había conseguido y por tanto cuando en el main() retornamos de la llamada recursiva el array del estado está otra vez en su estado inicial.

Para solucionar esto hay dos planteamiento principales:

1. Copiar el array solución cuando alcancemos esta solución y luego poder imprimirlo. Ojo, esto tiene que ser una copia no una asignación por referencia porque sino estaríamos en las mismas.
2. Evitar el borrar el estado a la vuelta de las llamadas recursivas. Para esto simplemente hay que añadir una condición después de la llamada recursiva para que cuando hayamos alcanzado una solución no ejecute el código de borrar estado. Se ha introducido este código en el problema y señalado con una flecha.

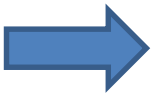
Pregunta 6 (2 p.)

El problema del puzzle se desarrolla sobre un tablero de 16 posiciones, donde hay colocadas 15 fichas, quedando una posición vacía. Las fichas no se pueden levantar del tablero, por lo que sólo es posible su movimiento por medio de desplazamientos sobre el mismo.

El objetivo del juego es, a partir de un estado inicial dado, alcanzar un estado final con los números ordenados y el vacío al final (ver gráfico abajo). Los únicos movimientos válidos son aquellos en que una ficha adyacente a la posición libre, se mueve hacia ésta. Este movimiento no se puede realizar en diagonal.

Queremos resolver este problema mediante la técnica de ramificación y poda. Y para ello utilizaremos el siguiente heurístico de ramificación: *Calcular el número de piezas que están en una posición distinta de la que les corresponde en la disposición final*.

1	2	3	4
5	6	7	8
9	10		11
13	14	15	12



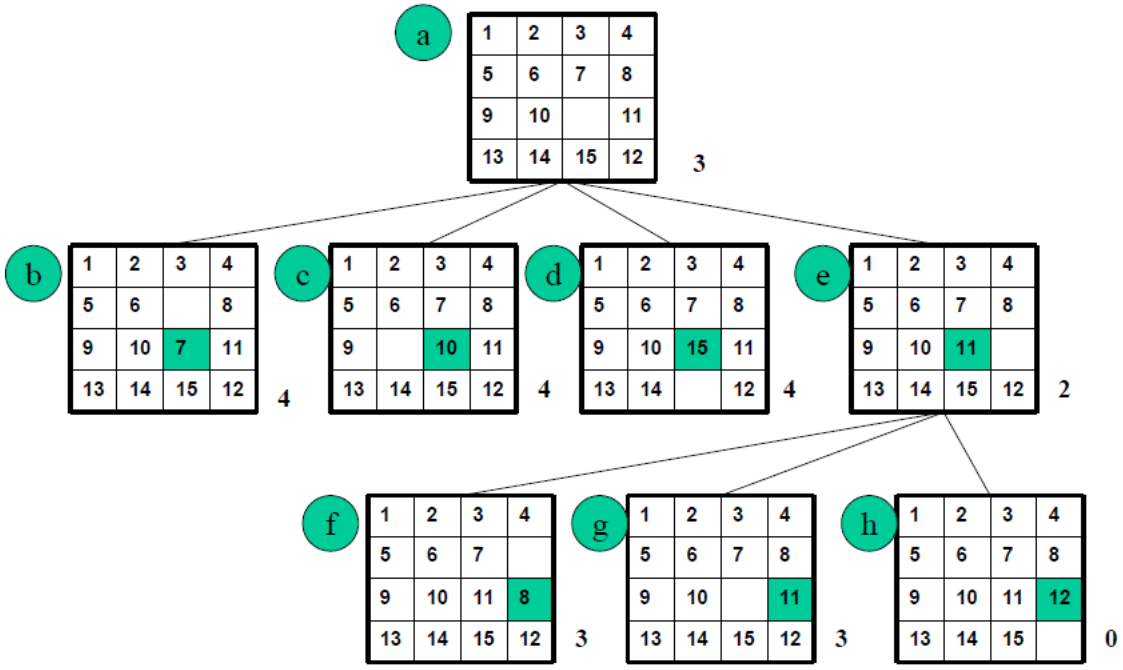
1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

a) (1 punto) Representa gráficamente el árbol de estados de la técnica ramifica y poda, para ello partiendo del estado que dado anteriormente, desarrolla dos de sus nodos.

Partimos del estado dado y desarrollamos dos nodos: el inicial y otro. Hay que tener en cuenta que el desarrollo de ramifica y poda es en anchura para cada nodo, es decir debemos desarrollar siempre todos los hijos válidos. En este caso hay cuatro hijos, salvo que tengamos

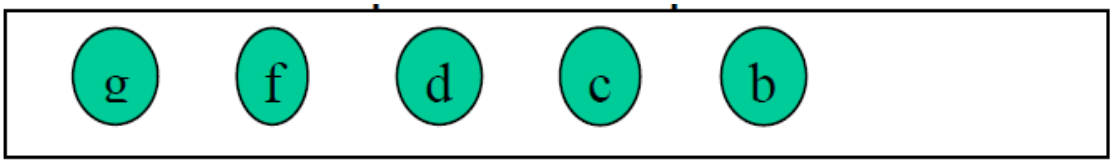
el hueco en un borde. También es correcto eliminar el estado que vuelve a ser igual que el "abuelo" el estado (g), pero habría que indicarlo.

Este problema no tiene ningún algoritmo de poda establecido, hay que considerar que no es un problema de optimización (todas las soluciones son iguales) y por tanto, tampoco podemos estimar si un estado va a permitir generar una solución mejor que otra.



b) (0,25 p.) Representa gráficamente los estados que quedan en la cola de prioridad después de realizar las operaciones del apartado anterior (puedes numerar los estados del apartado anterior para evitar volver a escribirlos).

En la cola de prioridad estarán todos los estados NO desarrollados y que no sean solución. Ordenados de menor a mayor por su heurístico de ramificación.



c) (0,25 p.) Explica las ventajas para este problema en concreto, de haber empleado esta técnica en vez de backtracking.

Este problema tiene un árbol que tiene mucha profundidad; pero es pequeño en anchura; por tanto el explorar el árbol en anchura no supone un sobre-coste grande; sin embargo la información que nos puede dar el heurístico puede significar que nos encaminemos rápidamente a la solución y sólo tengamos que explorar una pequeña parte del árbol. Además, en este problema tenemos el problema añadido de los estados repetidos que pueden llevar a ciclos sin fin en backtracking.

d) (0,5 p.) Explicar que ocurre con los estados repetidos en este problema, cómo influyen en la obtención de la solución y cómo podemos tratarlos.

En este problema aparecen de forma natural estados repetidos, ya que se forman ciclos de movimientos que devuelven al problema a un estado previamente explorado. Si no se evita esto influye negativamente en el tiempo de ejecución y en la memoria empleada y puede llevar a que explorando una y otra vez ciclos no se encuentra una solución que existe.

Debemos eliminar estados repetidos ya explorados. Para ello debemos Ir guardando en una estructura de datos (tabla hash o lista) los estados desarrollados para poder compararlos con los nuevos estados, si aparece uno repetido automáticamente se podaría.