Examen final – 25 de junio de 2018

Apellidos, nombre _____ NIF: _____

Pregunta 1 (1 p.)

Responde a las siguientes preguntas (Nota: Resuelve las operaciones intermedias respondiendo con el valor final y sin utilizar calculadora):

a) (0,5 p.) Considere un algoritmo de complejidad igual al famoso algoritmo de Floyd, realizado mediante programación dinámica. Si para n=30 se puede resolver un problema en un tiempo t=5 segundos, ¿cuál sería el tiempo necesario para resolver un problema de tamaño n=90?

$t_1 = 5$ -- $n_1 = 30$

$t_2 = ?$ -- $n_2 = 90$

Floyd tiene una complejidad $O(n^3)$

$$t_2 = \frac{f(n_2)}{f(n_1)} * t_1$$

$$t_2 = \frac{n_2^3}{n_1^3} * t_1$$

$$n_2 = k * n_1 => k = n_2/n_1$$

$$t_2 = k^3 * t_1 = 3^3 * 5 = 27 * 5 = \ 135\ segundos$$

b) (0,5 p.) Considere un algoritmo de complejidad igual que el de la búsqueda binaria (o dicotómica) Si para t=16 milisegundos, puede resolver un problema de tamaño n=16 ¿cuál sería el tamaño del problema resuelto si dispusiéramos de un tiempo de 40 milisegundos?

$t_1 = 16$ -- $n_1 = 16$

$t_2 = 40$ -- $n_2 = ?$

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

¡Cuidado! Hay que recordar que la fórmula anterior no se puede simplificar a

$$f^{-1}\left(\frac{t_2}{t_1} \cdot f(n_1)\right) \neq f^{-1}\left(\frac{t_2}{t_1}\right) \cdot n_1$$

$\log(n_2) = (t_2/t_1) * \log(n_1)$

$\log(n_2) = 2{,}5 * \log(n_1)$

Da igual que tipo de logaritmos aplicamos: logaritmo decimal, logaritmo neperiano o logaritmo en base dos, para todos sale el mismo resultado si utilizamos el suficiente número de decimales. Evidentemente si aplicamos logaritmo base 2 se puede hacer sin calculadora.

$n_2 = 2^{2,5 * \log(16)}$

$n_2 = 2^{2,5 * 4} = 2^{10} =$ **$1024 = n_2$**

Pregunta 2 (1 p.)

Dados los siguientes métodos, indica su complejidad y explica cómo lo has calculado claramente (asumimos que las instrucciones básicas en Java tienen una complejidad O(1)) (contesta directamente en esta hoja junto a cada apartado):

a) (0.5 p.)

La complejidad buscada es la de `maxSum3()`.

```
public int maxSum3(int[]v) {
	return maxSumByDivision(0, v.length-1, v);
}

private int maxSumByDivision(int left, int right, int[]v) {
	if (left==right)
		return v[left];
	else {
		int center = (left+right)/2;
		int maxLeft = maxSumByDivision(left,center, v);
		int maxRight = maxSumByDivision(center+1,right, v);

		int sum1=0;
		int maxSum1=0;
		for(int i=center; i>=left; i--) {
			sum1 = sum1+v[i];
			if (sum1>maxSum1)
				maxSum1=sum1;
		}

		int sum2=0;
		int maxSum2=0;
		for(int i=center+1; i<=right; i++) {
			sum2=sum2+v[i];
			if (sum2>maxSum2)
				maxSum2=sum2;
		}
		return biggest(maxLeft, maxRight, maxSum1+maxSum2);
	}//else
}

private int biggest(int a, int b, int c) {
	if (a>=b && a>=c) return a;
	else if (b>=a && b>=c) return b;
	else return c;
}
```

Solución: Estamos ante un algoritmo recursivo por divide y vencerás por división. Admite como entrada un vector y ofrece como salida la máxima suma de subsecuencias.

a = 2, b = 2; k = 1 => a = b^k => O(n^k * logn) => O(nlogn)

b) (0.5 p.)

```
public static int method1(int n) {
    int sum= 0;
    for (int i= 2; i<n/4; i++)
        for (int j= 10; j<=n*n; j*= 2)
            for (int k= n; k>=n-10; k-=1)
                sum+= k;
```

```
    return sum;
}
```

Solución: 2 bucles anidados, ya que el tercero k no está en función de n (sólo hace 10 iteraciones) y por tanto tiene complejidad constante, así como la operación que realiza. El primero es lineal y el segundo es logarítmico. Entonces, **O(nlogn)**

Pregunta 3 (2 p.)

Supongamos que se desea resolver el siguiente problema: a partir de la tabla de alimentos, que se proporciona a continuación, elaborar un menú en el que tomemos la mayor cantidad de calorías posible, sin que supere un precio dado. Cada alimento escogido puede aparecer en el menú en una cantidad máxima de 100 gr.; pero podría aparecer en una cantidad menor.

Alimento	Kilocalorías (100gr.)	Precio céntimos de € (100gr.)
Sopa	336	14
Lentejas	324	12
Arroz	364	14
Aceite de oliva	901	34
Filete ternera	90	90
Manzana	45	6
Leche	65	5
Pan	270	12
Huevos	289	20

a) (0,5 p.) Plantear un heurístico para resolver este problema mediante un algoritmo voraz.

Se pide el heurístico que permita obtener el menú con la mayor cantidad de calorías posible, esto significa que tenemos que conseguir el máximo, una solución óptima.

Recordar el problema de la mochila visto en clase, en la que se fragmentan los objetos.

El heurístico consisten en Calcular la relación Calorías / precio e ir seleccionando los los alimentos de mayor a menor respecto a la relación calculada anteriormente.

Ir cogiendo la máxima cantidad de los alimentos en el orden obtenido y cuando el dinero restante no permita obtener los 100gr. del alimento correspondiente fraccionarlo; de tal forma que se ajuste al dinero que restante: dinero restante / precio del alimento.

b) (1 p.) Dar una solución al problema, con la tabla dada, aplicando el heurístico, suponiendo que como mucho podemos gastar 1 €.

Solución:

Alimento	Kilocalorías (100gr.)	Precio céntimos de €(100gr.)	Calorias/cent	Orden	Proporción	Precio	Calorías
Sopa	336	14	24,0	4	1	14	336

Lentejas	324	12	27,0	1	1	12	324
Arroz	364	14	26,0	3	1	14	364
Aceite de oliva	901	34	26,5	2	1	34	901
Filete ternera	90	90	1,0	9		0	0
Manzana	45	6	7,5	8		0	0
Leche	65	5	13,0	7		0	0
Pan	270	12	22,5	5	1	12	270
Huevos	289	20	14,5	6	0,7	14	202,3
Total	2684	207				100	2397,3

c) (0,5 p.) Cuál es la mejor complejidad que nos puede ofrecer un algoritmo de tipo voraz para este problema y cómo conseguirla. Razona la respuesta.

Solución:

Si no ordenamos los alimentos en función del heurístico la complejidad será $O(n^2)$, sin embargo, si los ordenamos el algoritmo voraz tiene una complejidad lineal O(n) teniendo en cuenta lógicamente el tiempo invertido en la ordenación O(nlogn). La mejor complejidad del algortimo voraz es O(nlogn).

Pregunta 4 (2 p.)

Convertir la implementación del algoritmo DV recursiva a una implementación que utilice hilos para ejecutar cada una de las llamadas de forma paralela. Completa los dos archivos de código lo máximo posible utilizando el Framework Fork/Join.

Ten en cuenta que la clase `List` en Java tiene dos métodos que podrían serte útiles:

- `size()`. Devuelve el tamaño de una lista. Signatura:
  - `int size()`
- `subList()`. Crea una sublista a partir de una lista. Signatura:
  - `List<E> subList(int fromIndex, int toIndex)`
  - El elemento apuntado por `fromIndex` formará parte de la respuesta.
  - El elemento apuntado por `toIndex` no formará parte de la respuesta.

FileProcessingTask.java

```
import java.io.File;
import java.util.ArrayList;
import java.util.List;

/**
 * To process several system files in parallel
*/
class FileProcessingTask {
	private List<File> javaFiles = null;
	private String dirPath;

	public FileProcessingTask(String dirPath, List<File> javaFiles){
		this.dirPath = dirPath;
```

```
            this.javaFiles = javaFiles;
        }

        protected void process() {
          if (javaFiles == null) { //First time to start processing files
                javaFiles = new ArrayList<File>();
                File sourceDir = new File(dirPath);
                if (sourceDir.isDirectory()) {
                    for (File file : sourceDir.listFiles()){
                       javaFiles.add(file);
                    }
                }
          }
          processFiles(javaFiles);  // Procesamiento secuencial
        }

        protected void processFiles(List<File> filesToProcess) {
                for (File file : filesToProcess){
                       //Can be any other complex and long task
                       System.out.println(file.getName());
                }
        }
}
```

FileProcessingTaskTests.java

```
import org.junit.Test;

/**
 * FileProcessingTask JUnit tests
*/
public class FileProcessingTaskTest {
        /**
         * Process several files in parallel. Useful to do tests…
         */
        @Test
        public void executeTask() {
          FileProcessingTask problem = new FileProcessingTask("c:\\TEMP", null);
           problem.process();
        }

}
```

Solución:

FileProcessingTask.java

```
import java.io.File;
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.RecursiveAction;

/**
 * To process several system files in parallel
*/
class FileProcessingTask extends RecursiveAction {
        private static final int THRESHOLD = 5;
        private List<File> javaFiles = null;
        private String dirPath;
```

```
	public FileProcessingTask(String dirPath, List<File> javaFiles){
		this.dirPath = dirPath;
		this.javaFiles = javaFiles;
	}

	@Override
	protected void compute() {  // desde process()
	  if (javaFiles == null) { //First time to start processing files
		javaFiles = new ArrayList<File>();
		File sourceDir = new File(dirPath);
		if (sourceDir.isDirectory()) {
		    for (File file : sourceDir.listFiles()){
		       javaFiles.add(file);
		    }
		}
	  }

	// Divide y vencerás paralelo
	if (javaFiles.size() <= THRESHOLD) {
		processFiles(javaFiles);
	  }
	  else {
	      int center = javaFiles.size() / 2;
	      List<File> part1 = javaFiles.subList(0, center);
	      List<File> part2 = javaFiles.subList(center, javaFiles.size());
	      invokeAll(new FileProcessingTask(dirPath, part1),
			new FileProcessingTask(dirPath, part2));
	  }
	}

	protected void processFiles(List<File> filesToProcess) {
		for (File file : filesToProcess){
			//Can be any other complex and long task
			System.out.println(file.getName());
		}
	}
}
```

FileProcessingTaskTests.java

```
import java.util.concurrent.ForkJoinPool;
import org.junit.Test;

/**
 * FileProcessingTask JUnit tests
*/
public class FileProcessingTaskTest {
	private static ForkJoinPool pool;

	/**
	 * Process several files in parallel. Useful to do tests…
	 */
	@Test
	public void executeTask() {
	   pool = new ForkJoinPool();
	   FileProcessingTask problem = new FileProcessingTask("c:\\TEMP", null);
	   pool.invoke(problem);
	}
}
```

Pregunta 5 (2 p.)

Tenemos un laberinto representado por una matriz cuadrada (n x n), de enteros donde 0 (camino) significa que se puede pasar y 1 (muro) que no se puede pasar.

El punto de *inicio* en el laberinto estará situado siempre en una de las filas de la primera columna y estará representado por una coordenada (Ini_x, Ini_y); y el punto de destino estará situado en la última columna en una de sus filas y estará representado por (des_x, des_y).

Los movimientos posibles son: arriba, abajo, izquierda y derecha. Debemos marcar el camino que se sigue desde el inicio al destino con el número 2 en cada casilla. Al final debe aparecer si se encontró solución o no, y si se encuentra mostrar el número de pasos realizados desde origen a destino.

Escribir el programa en Java, que implemente mediante backtracking la solución al problema.

```
public class LaberintoUna {
    static int n;                //tamaño del laberinto (n*n)
    static int[][] lab;          //representación de caminos y muros
    static boolean haySolucion;      //se encontró una solución
      // arriba, abajo, izquierda, derecha
    static int[] movx= {0,  0,  -1, 1};      // desplazamientos x
    static int[] movy= {-1, 1,  0,  0};      // desplazamientos y

    static int inix;             //coordenada x de la posición inicial
    static int iniy;             //coordenada y de la posición inicial

    static int desx;             //coordenada x de la posición destino
    static int desy;             //coordenada y de la posición destino

    static void backtracking(int x, int y, int pasos) {
        if ((x == desx) && (y == desy) && (!haySolucion)) {
             //encontramos una solución y terminamos
            System.out.println("SOLUCIÓN ENCONTRADA CON " + pasos + " PASOS");
            haySolucion = true; //finalizamos el proceso
        }
        else
        {
            for (int k= 0; k<4; k++)
            {
                int u= x+movx[k];
                int v= y+movy[k];

                if (!haySolucion && u>=0 && u<=n-1 && v>=0 && v<=n-1 &&
                        lab[u][v]==0)
                {
                    lab[u][v] = 2; //marcar la nueva posición
                    backtracking(u, v, pasos+1);
                    lab[u][v] = 0; //desmarcar
                }
            }
        }
    }

    public static void main(String arg[]) {
       …
        inix= 0; iniy= 0; desx= n; desy= n;//damos valores a inicio y destino
        haySolucion = false; //iniciamos la variable solución encontrada
        lab[inix][iniy] = 2; //Marcamos inicio del camino: 2 es camino
```

```
        backtracking(inix, iniy, 0);
        if (!haySolucion) System.out.println("NO HAY SOLUCIÓN");
    }
```

Pregunta 6 (2 p.)

Consideremos el problema anterior del laberinto, pero esta vez queremos aplicar la técnica de ramificación y poda para su resolución, y que disponemos de los correspondientes heurísticos de ramificación y poda. Conteste de forma razonada a las siguientes preguntas, pensando entre otras cuestiones en las ventajas de cada heurístico de esta técnica.

a) (0,66 p.) Si buscamos todas las soluciones al problema. ¿Aportará una mejora en el tiempo de ejecución sobre backtracking? Razona la respuesta ¿Bajo qué condiciones se podría dar esta mejora, si es que se da?

No obtendríamos ninguna mejora ya que obtener todas las soluciones posibles implica recorrer todos los estados del problema para ir detectando soluciones, así que da igual que tengamos heurísticos o no.

b) (0,66 p.) Si buscamos sólo una primera solución al problema. ¿Aportará una mejora en el tiempo de ejecución sobre backtracking? ¿Bajo qué condiciones se podría dar esta mejora, si es que se da?

En este caso y gracias al heurístico de ramificación podemos alcanzar la primera solución de forma más rápida ya que este heurístico ayudará a encontrar el camino más corto en el árbol.

c) (0,66 p.) Si buscamos una solución óptima al problema, por ejemplo, el camino de menor longitud para completar el laberinto. ¿Aportará una mejora en el tiempo de ejecución sobre backtracking? ¿Bajo qué condiciones se podría dar esta mejora, si es que se da?

En este caso y gracias al heurístico de poda (apoyado en el de ramificación) podemos encontrar la solución óptima de forma más rápida, ya que en vez de explorar todos los nodos como en backtracking sólo debemos explorar una parte de estos.