Examen final – 22 de junio de 2017

Apellidos, nombre _____ NIF: _____

Pregunta 1 (1,5 p.)

Responde a las siguientes preguntas (Nota: Resuelve las operaciones intermedias):

a) (1 punto) Se ha medido tiempos empíricamente para un algoritmo y el resultado ha sido la siguiente tabla:

n	Tiempo(msegundos)
1024	1326
2048	5274
4096	21077

No disponemos del código fuente del algoritmo, pero sabemos la complejidad analítica es o bien $O(n^2)$ o bien O(nlogn). ¿A cuál de estas complejidades se ajustan mejor los tiempos medidos? Proporcione una estimación para n=2048 y para n= 4096. Explique el criterio de elección.

Solución:

K= n_2/n_1

f(n)= $O(n^2)$ → $t_2 = K^2 \cdot t_1$

f(n)= O(n·log n) → $t_2 = K \cdot \frac{\log K + \log n_1}{\log n_1} \cdot t_1$

Como n se multiplica por 2 en cada cambio K= 2 entre dos n consecutivos.

n	Medido	Estimación $O(n^2)$ Respecto a la anterior medición	Desviación en porcentaje	Estimación O(nlogn) respecto a la anterior medición	Desviación en porcentaje	Estimación O(nlogn) respecto a primera medición
1024	1326					
2048	5274	5304	1%	2917	45%	
4096	21077	21096	0%	11507	45%	6365

La desviación entre la medición y estimación es sólo del 1% para $O(n^2)$ y del 45% para O(nlogn). Por tanto, parece claro que la que mejor ajusta es $O(n^2)$.

b) (0,5 puntos) Considere un algoritmo de complejidad O(logn) Si para t=4 segundos puede resolver un problema de tamaño n=32 ¿cuál sería el tamaño del problema resuelto si dispusiéramos de un tiempo de 12 segundos?

$t_1 = 4$ -- $n_1 = 32$

$t_2 = 12$ -- $n_2 = ?$

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

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

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

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

$\log(n_2) = 3 * \log(n_1)$

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.

$n_2 = 2^{3 * \log(32)}$

$n_2 = 2^{3 * 5} = 2^{15} =$ **$32768 = n_2$**

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.)

```
public void complexity1(int n) {
    int x = n;
    while (x > 0) {
        int y = x;
        while (y > 0) {
            y = y / 2;
        }
        x = x - 1;
    }
}
```

Solución: 2 bucles anidados. El primero es lineal y el segundo es logarítmico. Entonces, **O(nlogn)**

b) (0.5 p.)

```
public void complexity2(int n) {
    int x = n;
    while (x < Integer.MAX_VALUE) {
        int y = Integer.MAX_VALUE;
        while (y > 0) {
            y = y - 2;
        }
        x = x + 2;
    }
    complexity2(n-2);
    complexity2(n-2);
    complexity2(n-2);
}
```

Solución: Divide y vencerás por substracción

a = 3; b = 2; k = 2 => a > 1 => $O(a^{n/b})$ => **$O(3^{n/2})$**

Pregunta 3 (1,5 p.)

Convertir la implementación del algoritmo ordenación por mezcla o mergesort DV recursiva a una implementación que utilice hilos para ejecutar cada una de las llamadas de forma paralela.

```
public class MezclaParalelo
{
	static int []v;

	public static void mezcla(int[]v)
	{
		mezclarec (v,0,v.length-1);
	}

	private static void mezclarec(int[] v, int iz,int de)
	{
		if (de>iz)
		{
			int m=(iz+de)/2;
			mezclarec(v,iz,m);
			mezclarec(v,m+1,de);
			combina(v,iz,m,m+1,de);
		}
	}

	public static void main (String arg [] )
	{
		int n=10;
		v = generarVector(n);
		mezcla(v);
		imprimirVector(v);
	}
...
}
```

Solución:

```
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveAction;

public class MezclaParalelo extends RecursiveAction
{
	int []v;     /** eliminamos static */
	int iz, de; /** variables de instancia para los índices */

	@Override
	protected void compute() {
		if (de>iz)
		{
			int m=(iz+de)/2;

			/** creamos instancias con las partes */
			MezclaParalelo mezIz= new MezclaParalelo(v,iz,m);
			MezclaParalelo mezDe= new MezclaParalelo(v,m+1,de);
			/** invocación en paralelo */
			invokeAll(mezIz,mezDe);
```

```
                combina(v,iz,m,m+1,de);
            }
        }

        /** constructor al que pasamos vector e índices */
        public MezclaParalelo(int[] vIni, int izIni, int deIni)
        {
            v= vIni;
            iz= izIni;
            de= deIni;
        }

        public static void main (String arg [] )
        {
            int n=10;
            /** declaramos y creamos v */
            int v[] = generarVector(n);
            imprimirVector(v);

            /** Creamos instancia inicial, pool e invocamos */
            MezclaParalelo mez= new MezclaParalelo(v,0,v.length-1);
            ForkJoinPool pool= new ForkJoinPool();
            pool.invoke(mez);

            imprimirVector(v);
        }
...
}
```

Pregunta 4 (2 p.)

Dadas dos cadenas de texto **str1** y **str2** y las operaciones (indicadas debajo), que pueden ser realizadas sobre **str1**, encuentra el mínimo número de ediciones (operaciones) necesarias para convertir **str1** en **str2**.

1. Insertar
2. Eliminar
3. Reemplazar

Todas las operaciones tienen el mismo coste. Algunos ejemplos:

```
Entrada:   str1 = "geek", str2 = "gesek"
Salida:  1
Podemos convertir str1 en str2 insertando una 's'.

Entrada:   str1 = "cat", str2 = "cut"
Salida:  1
Podemos convertir str1 en str2 re'a' with 'u'.

Entrada:   str1 = "sunday", str2 = "saturday"
```

```
Salida:  3

Los últimos tres caracteres y el primero son los mismos. Básicamente, necesitamos
covertir "un" a "atur".  Podemos remplazar 'n' por 'r', insertar 't', insertar 'a'
```

a) (1 p.) Crea un método llamado **EditDistance** (utilizando pseudocódigo) para indicar cómo implementarías una solución a este problema utilizando programación dinámica. El método recibirá como parámetros las dos cadenas de texto (una por cada palabra). Ten en cuenta la tabla proporcionada en el siguiente apartado.

Solución:

```
int EditDistance(char s[1..m], char t[1..n])
   // For all i and j, d[i,j] will hold the Levenshtein distance between
   // the first i characters of s and the first j characters of t.
   // Note that d has (m+1) x (n+1) values.
   let d be a 2-d array of int with dimensions [0..m, 0..n]

   for i in [0..m]
     d[i, 0] ← i // the distance of any first string to an empty second string
                 // (transforming the string of the first i characters of s into
                 // the empty string requires i deletions)
   for j in [0..n]
     d[0, j] ← j // the distance of any second string to an empty first string

   for j in [1..n]
     for i in [1..m]
       if s[i] = t[j] then
         d[i, j] ← d[i-1, j-1]        // no operation required
       else
         d[i, j] ← minimum of
                    (
                      d[i-1, j] + 1,  // a deletion
                      d[i, j-1] + 1,  // an insertion
                      d[i-1, j-1] + 1 // a substitution
                    )

   return d[m,n]
```

b) (1 p.). Completa la tabla correspondiente, teniendo en cuenta los valores proporcionados, que resuelve el problema para las palabras (**Sunday** y **Saturday**), indicando claramente el resultado final.

		S	**a**	**t**	**u**	**r**	**d**	**a**	**y**
	0	1	2	3	4	5	6	7	8
S	1								
u	2								
n	3								
d	4								

a	5								
y	6								

Solución:

		S	**a**	**t**	**u**	**r**	**d**	**a**	**y**
	0	1	2	3	4	5	6	7	8
S	1	0	1	2	3	4	5	6	7
u	2	1	1	2	2	3	4	5	6
n	3	2	2	2	3	3	4	5	6
d	4	3	3	3	3	4	3	4	5
a	5	4	3	4	4	4	4	3	4
y	6	5	4	4	5	5	5	4	3

Pregunta 5 (2 p.)

Necesitamos que un autobús haga un recorrido desde la estación de autobuses (a través de **n** ciudades diferentes) minimizando la cantidad de kilómetros que realiza hasta que vuelva a la estación de autobuses. Asumiendo que hay una carretera entre cada par de ciudades y que, como somos expertos en Algoritmia, sabemos que la solución tiene que ser un ciclo simple, completa el siguiente código para encontrar todas las posibles soluciones al problema:

```
import java.util.Random;
public class Problem {
	static int n; //number of cities
	static String []nodes; //cities
	static int[][]weights; //distances between cities (-1 if not possible)
	static int source; //the initial city
	static boolean []mark; //to mark the already visited cities
	static int[] path;  //one of the possible paths
	static int cost; //cost the one of one of the possible paths
	static int length; //size (level or length) of one of the paths
	static int nsol; //the total number of solutions

	public static void main (String arg[]) {
		//we can consider that all the variables are correctly
		Initialized…
		backtracking(              );
		if (nsol==0)
			System.out.println("THERE ARE NO CYCLES");
		else System.out.println("NUMBER OF CYCLES = " + nsol);
```

```
	}

	static void backtracking(                ) {
		if (current==source && length>0) {  //it is a solution state
			if (               ) {

				for (                      )
					System.out.print(                   ); //the
					name of the cities in order
				System.out.println("ITS COST IS = "+cost);
			}
		}
		else
		     for (                      )
		         if(                                      ) {




					backtracking(                   );




				}
	} //backtracking
}
```

Solution:

```
import java.util.Random;
public class Problem {
	static int n; //number of cities
	static String []nodes; //cities
	static int[][]weights; //distances between cities (-1 if not possible)
	static int source; //the initial city
	static boolean []mark; //to mark the already visited cities
	static int[] path;  //one of the possible paths
	static int cost; //cost the one of one of the possible paths
	static int length; //size (level or length) of one of the paths
	static int nsol; //the total number of solutions

	public static void main (String arg[]) {
		//we can consider that all the variables are correctly
		Initialized…
		backtracking(source);
		if (nsol==0)
			System.out.println("THERE ARE NO CYCLES");
		else System.out.println("NUMBER OF CYCLES = " + nsol);
	}

	static void backtracking(int current) {
		if (current==source && length>0) {  //it is a solution state
			if (length==n) {
				nsol++;
				for (int l=0;l<=length;l++)
```

```
                        System.out.print(nodes[path[l]]+"*"); //the
                        cities in order
                    System.out.println("ITS COST IS = "+cost);
                }
            }
            else
                for (int j=0;j<n;j++)
                    if(!mark[j] && weights[current][j]!=-1) {
                        length++;
                        cost = cost + weights[current][j];
                        mark[j] = true;

                        path[length] = j;

                        backtracking(j);

                        length--;
                        cost = cost - weights[current][j];
                        mark[j] = false;
                    }
        } //backtracking

}
```

Pregunta 6 (2 p.)

Consideremos la técnica de ramificación y poda. Responder de forma breve pero razonada las siguientes cuestiones:

a) Escribir el código para el método principal de la técnica:

```
public void ramificaYPoda(Nodo nodoRaiz)
```

Solución:

```
public void ramificaYPoda(Nodo nodoRaiz)
{
    cola.insertar(nodoRaiz);
    cotaPoda = nodoRaiz.valorInicialPoda();
    while (!cola.vacia() && cola.estimacionMejor()<cotaPoda) {
        Nodo actual = cola.sacarMejorNodo();

        ArrayList<Nodo> hijos = actual.expandir();
        for (Nodo estadoHijo : hijos) {
            if (estadoHijo.getHeuristico() < cotaPoda)
                if (estadoHijo.solucion()) {
                    mejorSolucion = estadoHijo;
                    cotaPoda= estadoHijo.getHeuristico();
                    }
                }
                else
                    cola.insertar(estadoHijo);
        }
    } //while
}
```

b) Una de las condiciones que manejamos en el esquema es la cota de poda. Qué efecto tendría si al encontrar una solución cambiásemos esta cota de poda a 0.

La cota de poda elimina todos los nodos hijos generados con un valor mayor y además descarta los que ya estén en la cola con un valor mayor o igual. Así que finalmente tendríamos una técnica que buscaría la primera solución y pararía.

c) Si la clase cola guardase los elementos en una pila LIFO en vez de una cola de prioridad.

Básicamente se realizaría un recorrido en profundidad (parecido a backtracking), sin heurísticos que guíen la selección, con la variante que se desarrollan de cada vez todos los estados hijos de cada nodo; pero luego siempre se coge el último hijo para seguir desarrollando.
Encontrará la solución óptima; pero el tiempo de ejecución es muy alto.

d) Si el método `getHeuristico()` devolviera un valor constante para todos los estados del problema.

Si devolviese un valor constante todos los estados tendrían la misma prioridad dentro del montículo, por lo que realmente no se priorizaría ninguno y se realizaría un recorrido en anchura (si el montículo dejase los estados por orden de inserción).
Sin embargo, existe otro problema y es que en los estados solución se cambia la cota de poda (cotaPoda= estadoHijo.getHeuristico();) al valor del heurístico para ese estado, por tanto tras encontrar la primera solución, el bucle while, por la condición cola.estimacionMejor()<cotaPoda parará y no seguiría expandiendo estados y por tanto, nos quedaríamos con la primera solución encontrada sin buscar otras que pueden ser mejores.
Sólo busca la primera solución y además, no se utilizan heurísticos así que el tiempo de ejecución será mayor que un algoritmo de ramificación.