

## Examen final – 14 de mayo de 2015

Apellidos, nombre \_\_\_\_\_ NIF: \_\_\_\_\_

### Pregunta 1 (1 p.)

Responde a las siguientes preguntas:

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

$n_1 = 2$  –  $t_1 = 5$  segundos

$n_2 = 4$  –  $t_2 = ?$

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

$$t_2 = (3^4) / (3^2) * t_1 = 3^{4-2} * 5 = 9 * 5 = 45 \text{ segundos} = t_2$$

- b) (0,5 puntos) Considere ahora un algoritmo con complejidad  $O(n^3)$ . Si para  $t = 2$  segundos el método pudiera resolver un problema con un tamaño de  $n = 100$ , ¿cuál podría ser el tamaño del problema si dispusiéramos de un tiempo de 54 segundos?

$t_1 = 2$  –  $n_1 = 100$

$t_2 = 54$  –  $n_2 = ?$

$$t_2 = K t_1 \Rightarrow K = \frac{t_2}{t_1} = \frac{54}{2} = 27$$

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

$$f(n) = n^c \rightarrow n_2 = \sqrt[c]{K} \cdot n_1 = \sqrt[3]{27} \cdot 100 = 3 * 100 = 300 = n^2$$

### Pregunta 2 (1 p.)

Teniendo en cuenta los siguientes métodos, indica:

```
public void method1(int n) {
    if (n > 0) {
        for (int i = 1; i < n; i++)
            method2(n+3);
        method1(n-1);
    }
}

public void method2(int n) {
    if (n <= 0) System.out.println("Hello"); //O(1)
    else {
        method2(n/4);
        method2(n/4);
        for (int i = 3; i <= n; i++)
            for (int j = n-2; j >= 0; j++)
                System.out.println(j); //O(1)
    }
}
```

- a) (0,5 puntos) Complejidad temporal de **method2**

Divide & Vencerás por división con  $a = 2$ ,  $b = 4$  y  $k = 2$ . Como  $a < b^k$  entonces la

Complejidad es  $O(n^k) = O(n^2)$

b) (0,5 puntos) Complejidad temporal de **method1**

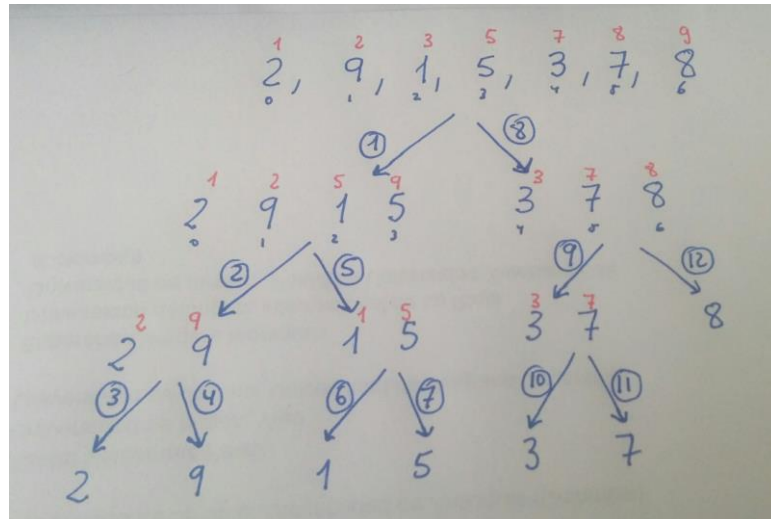
Divide & Vencerás por substracción con  $a = 1$ ,  $b = 1$  y  $k = 3$ . Como  $a = 1$  entonces la Complejidad es  $O(n^{k+1}) = O(n^4)$

### Pregunta 3 (3 p.)

Teniendo en cuenta el algoritmo de ordenación de Mezcla (Mergesort), responde a las siguientes preguntas:

a) (1 punto) Realiza la traza para ordenar la secuencia de números: 2, 9, 1, 5, 3, 7, 8. Deja MUY CLARO las divisiones, el orden en el cual se crean dichas divisiones y las ordenaciones parciales que hagas durante el proceso.

Solución:



b) (1 punto) Considerando que tenemos una clase en Java denominada **Mezcla** con la siguiente forma:

```
public class Mezcla {
    private int []elementos;

    public Mezcla(int[]elementos) {
        this.elementos = elementos;
        mezcla(0, elementos.length-1);
    }

    private void mezcla(int izquierda, int derecha) {
        //TODO
    }

    /* It is a linear process O(n) which combines a sorted
    * sequence x1..x2 with another one y1..y2,
    * bringing the result on the sorted vector v (elementos).
    * It has to use for it two auxiliary vectors,
    * since it is impossible to do everything on v*/
    private void combinar(int x1, int x2, int y1, int y2) {
        //TODO
    }

    public void imprimirSolucion() {
        for (int i=0;i<elementos.length;i++)
            System.out.print(elementos[i]+"//");
    }
}
```

```
public static void main(String arg []) {
    int[] elementos = new int[] {2, 9, 1, 5, 3, 7, 8};
    Mezcla mezcla = new Mezcla(elementos);
    mezcla.imprimirSolucion();
}
}
```

Sabiendo que tenemos el método **combinar** implementado, escribe el código del método recursivo **mezcla**, necesario para realizar la mezcla.

**Solución:**

```
// division a=2;b=2;k=1 => O(nlogn)
private void mezcla(int izquierda, int derecha) {
    if (derecha > izquierda) {
        int centro = (izquierda + derecha)/2;
        mezcla(izquierda, centro);
        mezcla(centro+1, derecha);
        combinar(izquierda, centro, centro +1, derecha);
    }
}
```

- c) (1 punto) Indica claramente TODOS los cambios necesarios en la clase **Mezcla** para que se ejecute el código utilizando el framework Fork/Join para paralelizar la ejecución (NOTA: los métodos que no cambian no hace falta escribirlos nuevamente).

**Solución:**

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

public class MezclaForkJoin extends RecursiveAction {
    private static final long serialVersionUID = 1L;
    private int []elementos;
    private int izquierda;
    private int derecha;

    public MergesortForkJoin(int[]elementos, int izquierda, int derecha) {
        this.elementos = elementos;
        this.izquierda = izquierda;
        this.derecha = derecha;
    }

    @Override
    public void compute() {
        if (derecha > izquierda) {
            int centro = (izquierda+derecha)/2;
            invokeAll(new MezclaForkJoin(elementos, izquierda, centro),
                new MezclaForkJoin(elementos, centro+1, derecha));
            combinar(izquierda, centro, centro+1, derecha);
        }
    }

    public static void main(String arg []) {
        int[] elementos = new int[] {2, 9, 1, 5, 3, 7, 8};

        MezclaForkJoin mezcla = new MezclaForkJoin(elementos, 0, elementos.length-1);
        ForkJoinPool pool = new ForkJoinPool();
        pool.invoke(mezcla);
        mezcla.imprimirSolucion();
    }
}
```

### Pregunta 4 (2 p.)

Una empresa de traducción de libros dispone de cuatro traductores en plantilla y actualmente han llegado cuatro libros para traducir, pide presupuesto a cada traductor para cada libro y cada uno ofrece una tarifa diferente en función de su experiencia previa y preferencias. La empresa debe hacer la selección que le permita obtener un coste de traducción más bajo en conjunto.

Traductor 1 (2400, 1080, 720, 660), traductor 2 (1320, 780, 900, 840), traductor 3 (1380, 1140, 1020, 660), traductor 4 (1680, 1200, 840, 1020). Los costes están ordenados del libro A al libro D.

- a) (0,25 puntos) Explicar cómo se calcula el heurístico de ramificación. Y aplicarlo al estado en el que asignamos al traductor 1 el libro D y al traductor 4 el libro C.

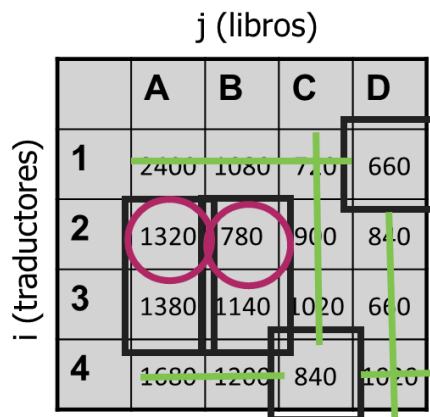
Este problema es el de las nadadoras (agentes – tareas visto en clase como ejemplo) El cálculo del heurístico de ramificación consiste en: La suma de lo ya asignado en ese estado más el caso mejor (mínimo de columnas) de lo que queda por asignar.

Aplicándolo al estado propuesto, el valor del heurístico sería:  $660 + 840 + 1320 + 780 = 3600$

j (libros)

	A	B	C	D
1	2400	1080	720	660
2	1320	780	900	840
3	1380	1140	1020	660
4	1680	1200	840	1020

i (traductores)



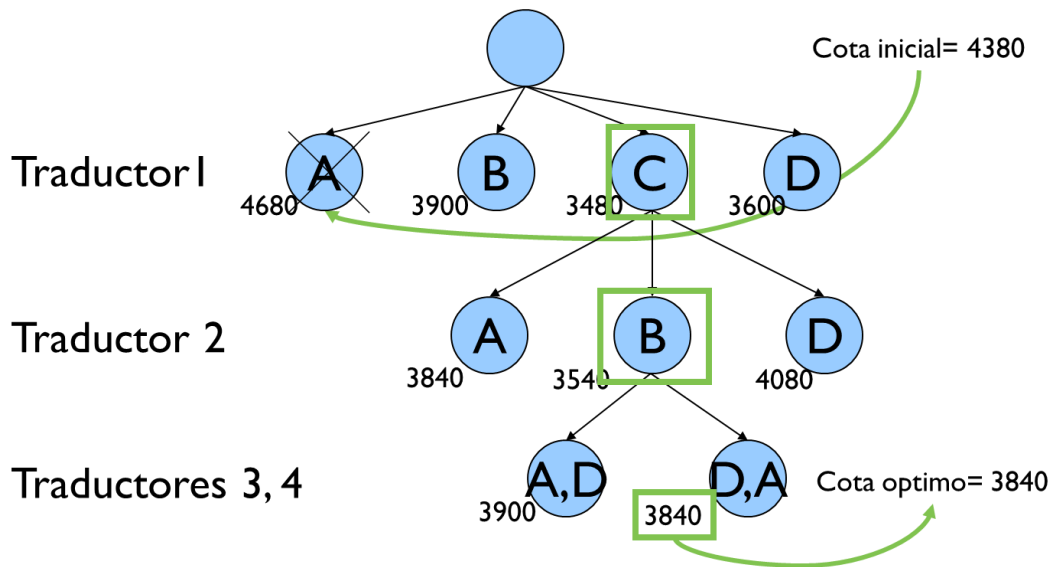
- b) (0,25 puntos) Calcular la cota inicial de poda y razonar cuándo se produce el cambio de esta cota.

Inicialmente, es la menor de la sumas de las dos diagonales de la matriz de costes.

Valor cota inicial:  $\min(5220, 4380) = 4380$ .

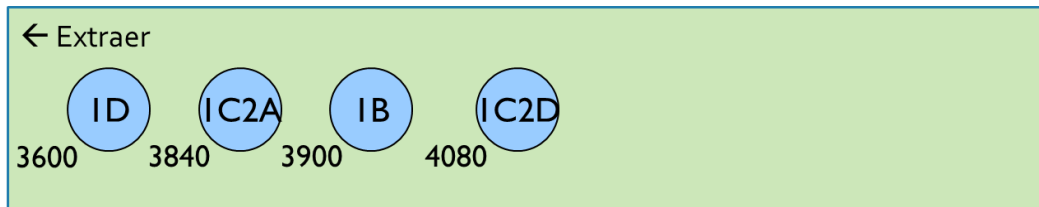
Cuando en el desarrollo de los estados del árbol, lleguemos a una solución válida cuyo valor sea menor que el de la cota actual, se cambiará la cota a este nuevo valor.

- c) (1 punto) Representar el árbol de estados hasta encontrar dos estados solución, teniendo en cuenta la cota de poda inicial.



Igual que en clase se desarrollan los dos últimos estados conjuntamente; evidentemente sería correcto desarrollarlos por separado, pero el desarrollo cambia un poco.

- d) (0,5 puntos) Representar de forma ordenada los estados que quedan en la cola de prioridad en la situación descrita en el punto c) indicando el estado con el valor de su heurístico. Razonar si todos estos estados llegarán a desarrollarse.



No todos los estados se llegarán a desarrollar de forma efectiva, ya que los tres últimos superan la nueva cota de poda establecida al encontrar una nueva solución, por tanto, estos tres no se desarrollarán nunca.

### Pregunta 5 (2 p.)

Un camión sale del almacén (*origen*) y tiene que recorrer  $n$  puntos que se conocen a priori, para realizar el reparto, debemos encontrar el *camino* con el menor *coste* que recorra todos esos puntos desde el almacén hasta volver de nuevo al almacén, sin repetir ninguno. Se dispone de la matriz de costes  $w$ , que contiene la distancia de ir de cada nodo a cualquier otro.

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

```
public class Reparto {

    static int n; // número de puntos
    static int[][] w; // matriz de costes del camino entre todos los puntos
    static int origen; // identifica nodo origen
    static boolean [] marca;
```

```
static int[] camino;           // guarda el camino
static int coste;             // coste del camino
static int longitud;          // número de nodos recorridos en el camino

static int[] caminoMejor;     // para anotar el ciclo mejor
static int costeMejor;        // coste del ciclo mejor

static void backtracking (int actual) {
    if (longitud==n && actual==origen) {
        if (coste<costeMejor)
        {
            for (int l=0;l<=longitud;l++)
                caminoMejor[l]=camino[l];
            costeMejor=coste;
        }
    }
    else
        for (int j=0;j<n;j++)
            if (!marca[j]
                && coste+w[actual][j]<costeMejor) {
                longitud++;
                coste=coste+w[actual][j];
                marca[j]=true;
                camino[longitud]=j;
                backtracking(j);
                longitud--;
                coste=coste-w[actual][j];
                marca[j]=false;
            }
} // fin de backtracking
}
```

- b) (0,4 puntos) Qué código se debe añadir y dónde (señalar con un flecha), para realizar una poda que descarte los estados que no llevan a una mejor solución.

```
→ if (!marca[j]
    && coste+w[actual][j]<costeMejor)
```

### Pregunta 6 (1 p.)

El material genético consiste en secuencias de pequeños elementos denominados nucleótidos. Los nucleótidos que forman el ADN son secuencias de adenina (A), citosina (C) timina (T) y guanina (G). Como paso previo a multitud de análisis biológicos, se nos pide encontrar una **subsecuencia común máxima** (*longest common subsequence, LCS*) de dos secuencias de ADN utilizando para ello la técnica de la **programación dinámica**, ya que se puede prestar a este tipo de problemas.

**NOTA:** Los caracteres en la subsecuencia no tienen por qué ser consecutivos (p.e., ADE es una subsecuencia, aunque no una subcadena, de ABCDE).

Para facilitarnos el trabajo, alguien nos explica una estrategia recursiva que puede ayudarnos a encontrar una solución. Consideremos:

- C1 es el elemento más a la derecha de S1
- C2 es el elemento más a la derecha de S2

- $S1'$  es  $S1$  sin  $C1$
  - $S2'$  es  $S2$  sin  $C2$
  - Así, para obtener la LCS de las secuencias  $S1$  y  $S2$ , tenemos tres problemas recursivos:
    - $L1 = \text{LCS}(S1', S2)$
    - $L2 = \text{LCS}(S1, S2')$
    - $L3 = \text{LCS}(S1', S2')$
  - La solución será el valor más grande de los siguientes 3 casos:
    - $L1$
    - $L2$
    - $L3 + 1$  si  $C1$  es igual a  $C2$ , o  $L3$  si  $C1$  no es igual a  $C2$
  - El caso base es cuando  $S1$  o  $S2$  son una cadena de tamaño 0
    - En ese caso la LCS de  $S1$  o  $S2$  es 0.
- a) (0,75 puntos) Diseña y rellena una tabla que nos permita obtener el tamaño de una LCS mediante programación dinámica. Utiliza los datos del ejemplo:  $S1 = \text{GCCCTA}$  y  $S2 = \text{GCGCA}$

**Solución:**

		G	C	C	C	T	A
	0	0	0	0	0	0	0
G	0	1	1	1	1	1	1
C	0	1	2	2	2	2	2
G	0	1	2	2	2	2	2
C	0	1	2	3	3	3	3
A	0	1	2	3	3	3	4

- b) (0,25 puntos) Indica y explica cuál es la complejidad del algoritmo utilizado para rellenar la tabla.

La solución con programación dinámica tiene una complejidad de  $O(m*n)$ , siendo  $m$  y  $n$  el tamaño de las dos secuencias de ADN