Escuela de Ingeniería Informática
Universidad de Oviedo

Algoritmia
2014-2015

Examen final – 25 de junio 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 2 segundos para n=2, calcula el tiempo que tardará para n=6.

n1 = 2 – t1= 2 segundos

n2 = 6 – t2 = ?

n1 * k = n2 => k = n2/n1 = 6/2 = 3

$t2 = (3^{n2}) / (3^{n1}) * t1 = 3^{n2-n1} * t1 = 3^{n1(k-1)} * 2 = 3^{2(3-1)} * 2 = 3^4 * 2 =$ **162 = t2 = segundos**

b) (0,5 puntos) Considere de nuevo un algoritmo con complejidad $O(3^n)$. 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?

t1= 2 – n1= 100

t2= 54 – n2 = ?

$t_2 = Kt_1 \Rightarrow K = \frac{t_2}{t_1} = \frac{54}{2} = 27$

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

$Log_3 3n^2 = log_3 27 + log_3 3n^1$ → n2 = 3 + n1 → **n2 = 103**

Pregunta 2 (1 p.)

Indica la complejidad temporal de los siguientes fragmentos de código:

a) Divide & Vencerás por división con a = 4, b = 2 y k = 2. Como $a = b^k$ entonces la complejidad es $O(n^k \log n)$ = **$O(n^2 \log n)$**

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

b) En este caso, no hay llamadas recursivas, tenemos 3 bucles, el primero simple y en secuencia los otros dos anidados:

- El primero tiene una complejidad $O(\log_3 m) = O(\log m)$
- El segundo tiene una complejidad O(1) ¿por qué? porque como mucho se ejecutará un número constante de veces = 100, debido al valor con el que queda la variable i a la salida del bucle anterior.
- El tercero tiene una complejidad $O(\log_2 n) = O(\log n)$

En resumen, la complejidad total es **O(logm) + O(logn)**

```
public static void foo(int n, int m)
{
   int i = m;
   while (i > 100)
      i = i / 3;

   for (int k = i;  k >= 0; k--)
   {
        For (int j = 1; j < n; j *= 2)
              System.out.print(k + "/t" + j);
        System.out.println();
   }
}
```

Pregunta 3 (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.

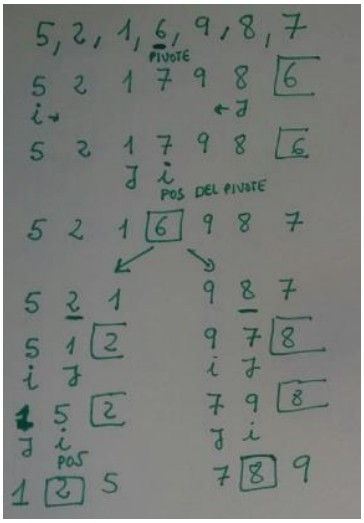
```
public class LaberintoUna {
    static int n;               //tamaño del laberinto (n*n)
    static int[][] lab;         //representación de caminos y muros
    static boolean haySolucion;     //se encontró una solución
      // arriba, abajo, izquierda, derecha
    static int[] movx= {0,  0,  -1,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

```
static int desy;            //coordenada y de la posici□estino

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 SOLUCIм");
}
```

Pregunta 4 (2 p.)

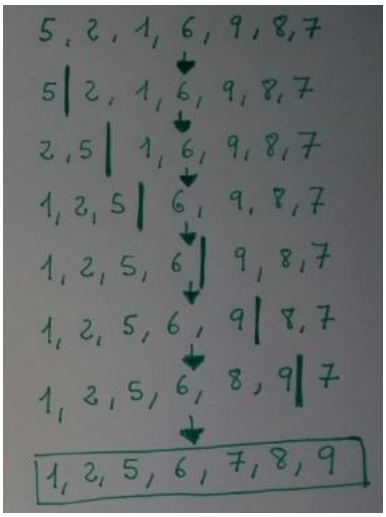
Teniendo en cuenta la siguiente secuencia de números: 5, 2, 1, 6, 9, 8, 7.

a) (0,75 puntos) Ordénalos por el algoritmo de ordenación llamado Rápido utilizando como pivote el **elemento central**. Indica la traza para ver la solución paso por paso.



b) (0,25 puntos) Justifica y explica la complejidad del algoritmo que has aplicado para ordenar los números indicados.
El algoritmo Rápido, utilizando como pivote el elemento central, tiene una complejidad de O(nlogn) en el caso medio. O(logn) porque estamos ante una estructura de árbol y O(n) porque hay que realizar un particionado en cada nivel del árbol.

c) (0,75 puntos) Ordénalos por el algoritmo de ordenación llamado Inserción directa. Indica la traza para ver la solución paso por paso.



d) (0,25 puntos) Justifica y explica la complejidad del algoritmo anterior para el caso mejor, peor y medio.
Caso mejor: O(n), cuando están todos los números ordenados, sólo hace falta una iteración
Caso peor: $O(n^2)$
Caso medio: $O(n^2)$

Pregunta 5 (2 p.)

Sea el esquema general de la técnica de ramificación y poda:

```
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
```

(a)

```
                    cola.insertar(estadoHijo);
                }
            } //while
        }
```

Explicar (respuesta corta) qué ocurre en términos de funcionamiento y tiempo de ejecución si realizamos los siguientes cambios en este esquema:

a) Qué modificaciones habría que realizar para que este esquema buscase la primera solución (tacha lo que sobre y escribe lo que falta sobre el código anterior).

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

                        }
                    }
                    else
                        cola.insertar(estadoHijo);
                }
        } //while
    }
```

Realmente no sería imprescindible quitar todo el código tachado. Simplemente indicamos que este código sobra, porque no se necesita ninguna poda, pero podría funcionar sin eliminarlo.

b) 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.

c) 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.

d) Si eliminamos el fragmento de código marcado como (a)

Cuando cambiamos la cota de poda al encontrar una solución, podamos todos los nodos que tengan un heurístico mayor desarrollados a partir de ese momento. Este funcionamiento al eliminar este fragmento de código. Pero si había algún nodo con mayor heurístico en la cola de prioridad este permanece allí y por tanto al final del proceso de ejecución se desarrollará sin necesidad empeorando el tiempo de ejecución.

Pregunta 6 (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 gastar más de una cantidad dada. 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	325	12
Arroz	362	14
Aceite de oliva	900	34
Filete ternera	92	90
Manzana	45	6
Leche	63	5
Pan	270	12
Huevos	280	17

a) (0,75 p.) Plantear un heurístico para resolver este problema mediante un algoritmo voraz.

Se pide el heurístico que permite obtener el menú con la **mayor cantidad de calorías posible**: esto significa que tenemos que conseguir el óptimo.
Recordar el problema de la *mochila* visto en clase, en la que se fragmentan los objetos.
Calcular la relación Calorías / precio.
Ordenar los alimentos de mayor a menor respecto a la relación calculada anteriormente.
Ir cogiendo la máxima cantidad de los alimentos en el orden obtenido cuando no tengamos dinero para obtener la cantidad máxima del último alimento fraccionarlo; para que se ajuste al dinero que nos queda.

b) (1,25 p.) Dar una solución al problema, con la tabla dada, aplicando el heurístico, suponiendo que como mucho podemos gastar 1 €.

Alimento	Calorías (100gr.)	Precio (100gr.)		Calorias /cent	Orden	Proporción	Precio	Calorias
Sopa	336	14		24,00	4	1	14	336
Lentejas	325	12		27,08	1	1	12	325
Arroz	362	14		25,86	3	1	14	362
Aceite de oliva	900	34		26,47	2	1	34	900
Filete ternera	92	90		1,02	9	0	0	0
Manzana	45	6		7,50	8	0	0	0
Leche	63	5		12,60	7	0	0	0
Pan	270	12		22,50	5	1	12	270
Huevos	280	17		16,47	6	0,82	14	230,6
Total	2673,0	204,0					100,0	2423,6

En la columna proporción aparece la cantidad de cada alimento que cogemos respecto a los 100 gr dados en la tabla.

En la columna orden, indicamos el orden de ordenación respecto a la relación Calorias / centimos. De todos los alimentos tomamos la cantidad máxima (100gr) hasta que no tenemos suficiente dinero. Con el dinero restante, cogemos justo la cantidad que podemos pagar del siguiente alimento (mochila con fraccionamiento): de huevos cogemos 0,82, lo que supone 230,6 Kcal y 14 céntimos.