Escuela de Ingeniería Informática
Universidad de Oviedo

Algoritmia
2015-2016

Control 2 – 18 de abril de 2016

Apellidos, nombre _____ NIF: _____

Pregunta 1 (4 puntos)

Queremos crear un algoritmo para resolver el juego del Sudoku. Para diseñarlo utilizaremos un algoritmo que utiliza la técnica de **Backtracking**.

a) (2,5 p.) Rellena los huecos para que el algoritmo de backtracking proporcione una solución:

```
public class SudokuPrimera {
	static int n;                    // Tamaño del Sudoku
	static int[][] tablero; // Tablero para el juego del Sudoku
	static boolean solEncontrada;

	static void backtracking(int x, int y) {
		// x==n no quedan posiciones libres --> solución
		if (x==n) {
			solEncontrada = true;
			mostrarTablero();
		}
		else
			for (int k=1; k<=9; k++) { // n posibilidades
				if(!solEncontrada && comprobarFila(x,k) &&
comprobarColumna(y,k) && comprobarRegion(x,y,k)) {
					tablero[x][y] = k; // guardar número

					int[] posVacia = new int[2];
					// Buscamos siguiente posición rellenar
					posVacia = buscarVacia(x,y);
					backtracking(posVacia[0], posVacia[1]);

					tablero[x][y] = 0; // borrar número
			}
		 }
	}

	/* Comprueba si se puede poner el número k en la fila x */
	static boolean comprobarFila(int x, int k) { … }

	/**
	 * Comprueba si se puede poner k en la columna y
	 */
	static boolean comprobarColumna(int y, int k) {
		boolean b = true;
		for (int i=0; i<n; i++)
			if (tablero[i][y] == k) b=false;
		return b;
	}

	/**
	 * Comprueba si se puede poner k en una region del sudoku
	 */
	static boolean comprobarRegion(int x, int y, int k) {
		boolean b = true;
```

```
            int sectorX = x/3; // si x=7 => sectorX = 2
            int sectorY = y/3; // si y=1 => sectorY = 0
            //that is, we only check the region from [6,0] to (9, 3)
            //cada region tiene tamaño 3x3 (working with a 9x9 Sudoku)
            for (int i= sector*3; i<sector*3+3; i++)
                  for (int j= sector*3; j<sector*3+3; j++)
                        if (tablero[i][j] == k) b=false;
            return b;
      }

      static void mostrarTablero() {…}
       /* Busca la siguiente posición vacía para poner un número */
      static int[] buscarVacia(int x, int y) {…}

}
```

b) (0,5 p.) ¿Cuántos hijos genera cada estado en este problema? Explica tu respuesta.

Se generan 9 posibles hijos (uno por cada valor posible que puede contener en la celda). Sin embargo, algunos de estos hijos se eliminarán al comprobar las condiciones y comprobar que no son válidos, debido tanto al tablero inicial como a las celdas que se generaron anteriormente.

c) (1 p.) Trabajando con backtracking, normalmente tenemos dos tipos diferentes de complejidades. ¿Cuáles son estas complejidades? Explica claramente cuando obtenemos cada una de ellas haciendo referencia al tipo de árbol generado.

Si el número de posibles valores para cada x_i es m_i, tendremos un árbol en el que todos los nodos tengan m_i hijos (grado del árbol) entonces se generan:

- m_1 nodos en el nivel 1
- $m_1 \cdot m_2$ nodos en el nivel 2
- $m_1 \cdot m_2 \cdot \ldots \cdot m_n$ nodos en el nivel n

Para un problema $m_i = 2$ (grado 2, sólo hay dos posibilidades de variación y se genera un árbol de llamadas binario). El número de nodos generados es:

- $t(n) = 2 + 2^2 + 2^3 + \ldots + 2^n = 2n+1 - 2 \in O(2^n)$

Para un problema en el que el grado del árbol se reduce en uno en cada nivel. Es decir en el primer nivel tenemos n hijos, cada nodo de este nivel tiene n-1 y así sucesivamente hasta llegar a 1.

- $t(n) = n + n \cdot (n-1) + n \cdot (n-1) \cdot (n-2) + \ldots + n! \in O(n!)$

En general, órdenes de complejidad factoriales o exponenciales.

Pregunta 2 (3 puntos)

La función de *Ackermann* es un ejemplo clásico de función recursiva, es notable especialmente porque no es una función con recursión primitiva. En computabilidad fue un contraejemplo a lo que se creía en 1900 que cada función computable era recursiva primitiva. Esta función se utiliza como benchmark para medir la habilidad de los compiladores para optimizar la recursión.

Normalmente se define como:

$$A(m,n) = \begin{cases} n+1, & \text{si } m = 0; \\ A(m-1,1), & \text{si } m > 0 \text{ y } n = 0; \\ A(m-1, A(m,n-1)), & \text{si } m > 0 \text{ y } n > 0 \end{cases}$$

a) (0,5 p.) Escribe el código fuente en Java de un método que resuelva el problema utilizando un esquema recursivo para cualquier valor de *m* y *n*.

```
/**
 * Función de Ackermann
 * @param m debería ser mayor que 0
 * @param n debería ser mayor que 0
 * @return
 */
private static long ackermann(long m, long n) {
      if (m==0)
            return n+1;
      else if (n==0)  // no hace falta la condición m>0
            return ackermann(m-1,1);
      else  // no hace falta la condición n>0
            return ackermann(m-1,ackermann(m,n-1));
}
```

b) (2 p.) Diseña un algoritmo para resolver Ackermann para *A(3,5)* usando la estrategia de programación dinámica. No hace falta escribir el código, crea la tabla que necesitamos y rellena los valores. ¿Cuál es el resultado final?

Partimos de la función recursiva; pero a partir de ella creamos una tabla y la forma de rellenarla para obtener el resultado iterativamente.

Al tener 2 parámetros creamos la tabla con dos dimensiones y los límites están claramente definidos: 0 como límite inferior de las dimensiones y los valores de los parámetros como límite superior.

La primera fila (fila 0) se puede rellenar directamente con los valores proporcionados por la función de forma directa.

La segunda condición nos da la forma de calcular los valores de la columna 0.

La tercera condición después de obtener el valor de la función anidada nos da la celda de la que tenemos que extraer el valor del resto de las celdas.

La única dificultad que tenemos que superar es que a partir de ciertos valores, la celda de la cual dependemos está fuera de la tabla definida. Aquí podemos extrapolar lo que hicimos por ejemplo en el problema de la mochila para eliminar los valores base de la tabla; pero aquí como son valores de la función habrá que descubrir el "patrón" para calcularlos. Es fácil al menos para las filas 0 y 1, algo más complejo para la fila 2, de esta forma, en función de la columna n podemos obtener el valor correspondiente de la celda y calcular la siguiente fila.

n / m	**0**	**1**	**2**	**3**	**4**	**5**	Generación del valor para n
0	1	2	3	4	5	6	n+1
1	2	A(0,2)=3	A(0,3)=4	A(0,4)=5	6	7	n+2
2	3	5	7	9	11	13	2n+3
3	5	13	29	61	125	**253**	

Resultado de A(3,5)= 253

c) (0,5 p.) Explica las principales diferencias entre Divide y Vencerás y Programación dinámica tanto en el diseño como en la complejidad.

Divide y vencerás

- Es un método de refinamiento progresivo (descendente):
- Atacamos el caso completo que vamos dividiendo en subcasos más pequeños mediante un esquema **recursivo**.
- La complejidad para es **no polinómica → exponencial** si se resuelve con DV con sustracción o crece el número de subproblemas.

Programación dinámica

- Es una técnica ascendente:
- Se empieza por los subcasos más pequeños y combinando las soluciones **iterativamente** se obtienen respuestas a subcasos cada vez mayores, hasta que llegamos al caso completo.
- La complejidad viene dada por el recorrido de la tabla: **polinómica**.

Pregunta 3 (3 puntos)

En una compañía que vende productos de madera se ha creado un sistema que permite construir listones de cualquier longitud (múltiplo del listón pequeño) a partir de pequeños listones prefabricados de determinadas medidas que disponen de un sistema en encaje entre sí.

El proveedor sólo suministra un conjunto de listones con unas longitudes dadas a elegir entre los dos tipos siguientes, del que se puede encargar el número que sea necesario:

1. Conjunto 1: Se dispone de listones prefabricados con las siguientes longitudes: 10, 30 y 40,90.
2. Conjunto 2: Se dispone de listones prefabricados con las siguientes longitudes: 10, 20, 40, 80.

Teniendo en cuenta esto se pide:

d) (1 p.) Indica un heurístico que podamos utilizar en un algoritmo voraz para calcular qué listones necesitamos y describe cómo se aplica.

Para obtener el mínimo de listones para obtener la longitud pedida. Utilizar el listón prefabricado con más longitud que no supere la longitud que queda para completar el listón a construir.

Esto se repite sucesivamente hasta obtener el listón final de la longitud pedida.

e) (2 p.) Qué conjunto de listones encargará la compañía si quiere utilizar siempre la menor cantidad de listones, independientemente de la longitud del listón a construir. Demuéstralo con ejemplos.

Lo que se quiere averiguar es para cuál de los dos conjuntos el heurístico se comporta como óptimo, es decir siempre nos proporciona el mínimo número de listones dentro de los posibles para ese conjunto concreto.

Con el primer conjunto de listones el heurístico NO es óptimo. Sólo hay que plantear el caso de intentar crear un listón de longitud 60 o 150 (y más casos).

Listones	90	40	30	10	
Heurístico		1		2	
Alternativa			2		

Con el segundo conjunto, independientemente del listón que queramos construir no encontramos otra forma mejor de cogerlos que la que nos da el heurístico.

Por tanto, la compañía debe escoger el segundo conjunto para utilizar siempre el mínimo de listones dentro de ese conjunto.

Si optamos por comparar los dos conjuntos, va a depender del caso la resolución, y no hay un mejor conjunto general para dos. Esto se puede comprobar con distintas medidas donde en algunas se comportan los dos igual, por ejemplo para 120 (2 listones), en otras se comporta mejor el conjunto 2, por ejemplo para 160 (conj 1: 90+40+30, conj 2: 80+80) y en otras se comporta mejor el conjunto 1, por ejemplo para 180 (conj 1: 90+90, conj 2: 80+80+20). No

disponemos de información de que tipo de listones se quiere fabricar así que no podemos seleccionar un conjunto en base a esto.