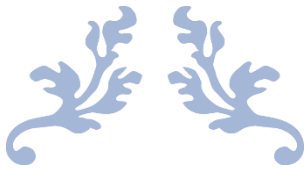


PRÁCTICA 4:

ELIMINACIÓN DE LA RECURSIVIDAD Y PROGRAMACIÓN DINÁMICA

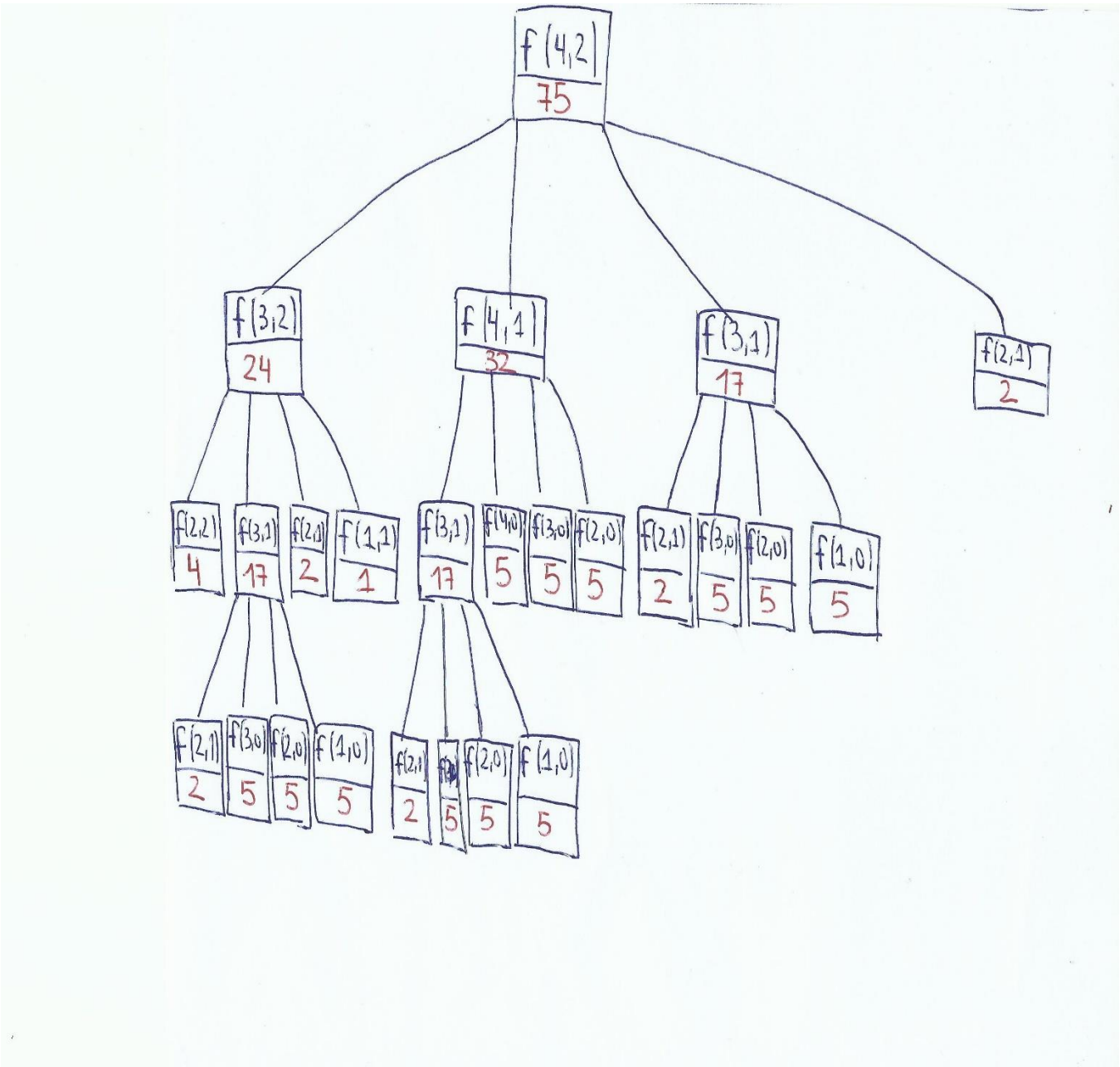
Algoritmos Avanzados. Grado en Ingeniería Informática

11 DE DICIEMBRE DE 2017
JOSE VICENTE BAÑULS GARCÍA
JORGE ARANDA GARCÍA

Tabla de contenido

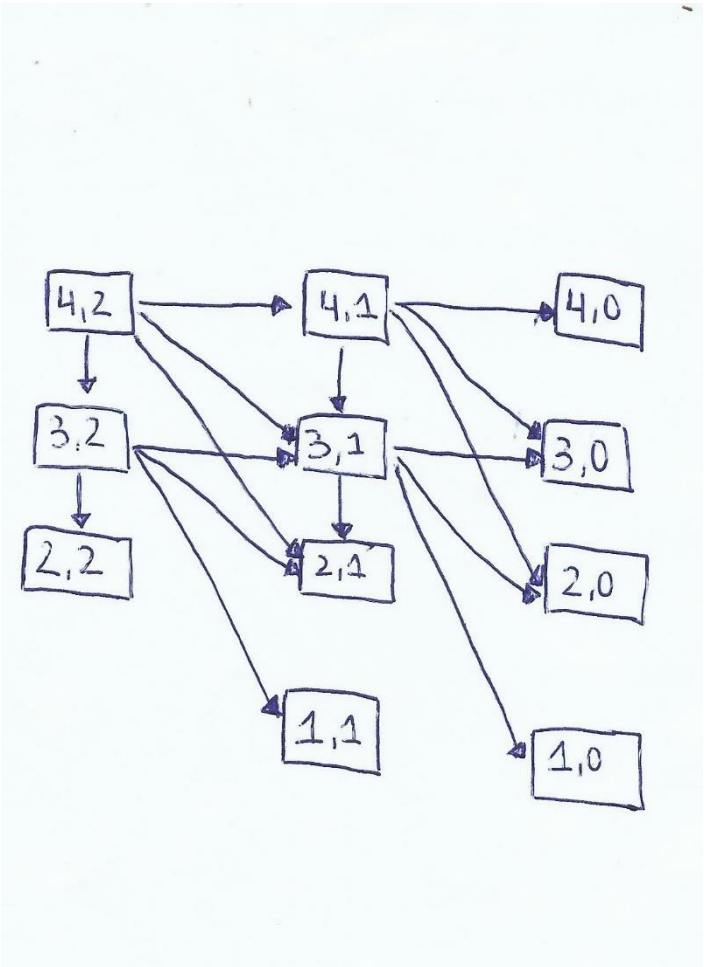
1. ELIMINACIÓN DE LA RECURSIVIDAD

a) Dibujar y analizar el árbol de recursión para la llamada f(4,2). Añadir la solución para cada subproblema.



En cada uno de los nodos, se muestra la llamada recursiva que se hace en azul, y en rojo se muestra el resultado de dicha llamada. Si un nodo en concreto no tiene hijos, es porque las condiciones de la llamada recursiva se corresponden con alguno de los casos base, bien multiplicar x*y si x<=2, y>0, bien 5, si y<0.

b) Dibujar y analizar el grafo de dependencia asociado al árbol anterior.



Cada celda se corresponde con alguna de las llamadas necesarias para la resolución del problema planteado.

Como se puede observar, las llamadas que no son casos base, es decir cuando la x es menor que 2 o la y es 0, dependen siempre de otras cuatro posiciones de la tabla, que se corresponden con las llamadas recursivas.

f(x,y) = f(x-1,y) + f(x,y-1) + f(x-1,y-1) + f(x-2,y-1).

Por ejemplo: **f(3,2)** = f(2,2)+ f(3,1)+ f(2,1)+ f(1,1).

c) Dibujar la tabla asociada especificando las dimensiones de la misma y los subproblemas que se almacenan en cada celda de la tabla.

f(1,0) 5	f(1,1) 1	
f(2,0) 5	f(2,1) 2	f(2,2) 4
f(3,0) 5	f(3,1) 17	f(3,2) 24
f(4,0) 5	f(4,1) 32	f(4,2) 75

El tamaño de la tabla es de **m * n+1**. Cada posición de la tabla almacena el resultado (en verde) de la llamada correspondiente (en negro). Si es un caso base se rellena sin más y si no, se rellena con el resultado de las posiciones que le sean necesarias. Cabe destacar que la tercera posición de la primera fila no está rellena porque para el caso que nos concierne no es necesaria.

d) Código del algoritmo de memorización utilizado para eliminar la recursividad.

Para estos tres últimos apartados se ha creado una clase en Java, que se entrega junto con este informe, además una clase main.java para ejecutar cada uno de los métodos. Esta clase se llama EliminacionRecursividad.java.

En concreto, la cabecera del método que implementa el algoritmo de memorización es la siguiente:

```
public static int mem(int m, int n){…}
```

e) Código del algoritmo de tabulación utilizado para eliminar la recursividad.

La cabecera que se corresponde con este algoritmo es:

```
public static int tab (int m ,int n){…}
```

f) Código del algoritmo de tabulación con minimización de memoria utilizado para eliminar la recursividad.

La cabecera que se corresponde con este algoritmo es:

```
public static int tabOpt(int m, int n){…}
```

La diferencia principal entre este método y el anterior es la optimización de la memoria. Para ello, se cambia la tabla utilizada en el apartado anterior por un vector de tamaño m, en el que, junto con algunas variables auxiliares, se van almacenando los valores obtenidos, hasta obtener la solución final.

2. PROGRAMACIÓN DINÁMICA

a) Definición de la función global.

Beneficio(t) = Cantidad máxima de beneficio que se puede obtener al invertir durante t meses.

b) Demostración del principio de optimalidad.

En el instante i la cantidad se ha escogido de forma óptima puesto que en todas las etapas anteriores se ha escogido de forma óptima.

c) Definición de los casos base.

Beneficio(0)= C, donde C es la cantidad de la que disponemos.

d) Definición de la función recursiva.

Beneficio(i)=

máx {beneficio (i-1), (beneficio (i-1) – GCD[i-1]) * RCD[i-1]}

si 1 <= i < 6

máx {beneficio(i-1), (beneficio(i-1)-GCD[i-1]) * RCD[i-1], (beneficio(i-6)-GBT[i-6]) * RBT[i-6]}

si i >= 6

e) Construcción del árbol recursivo para una llamada concreta.

Para unos datos de entrada:

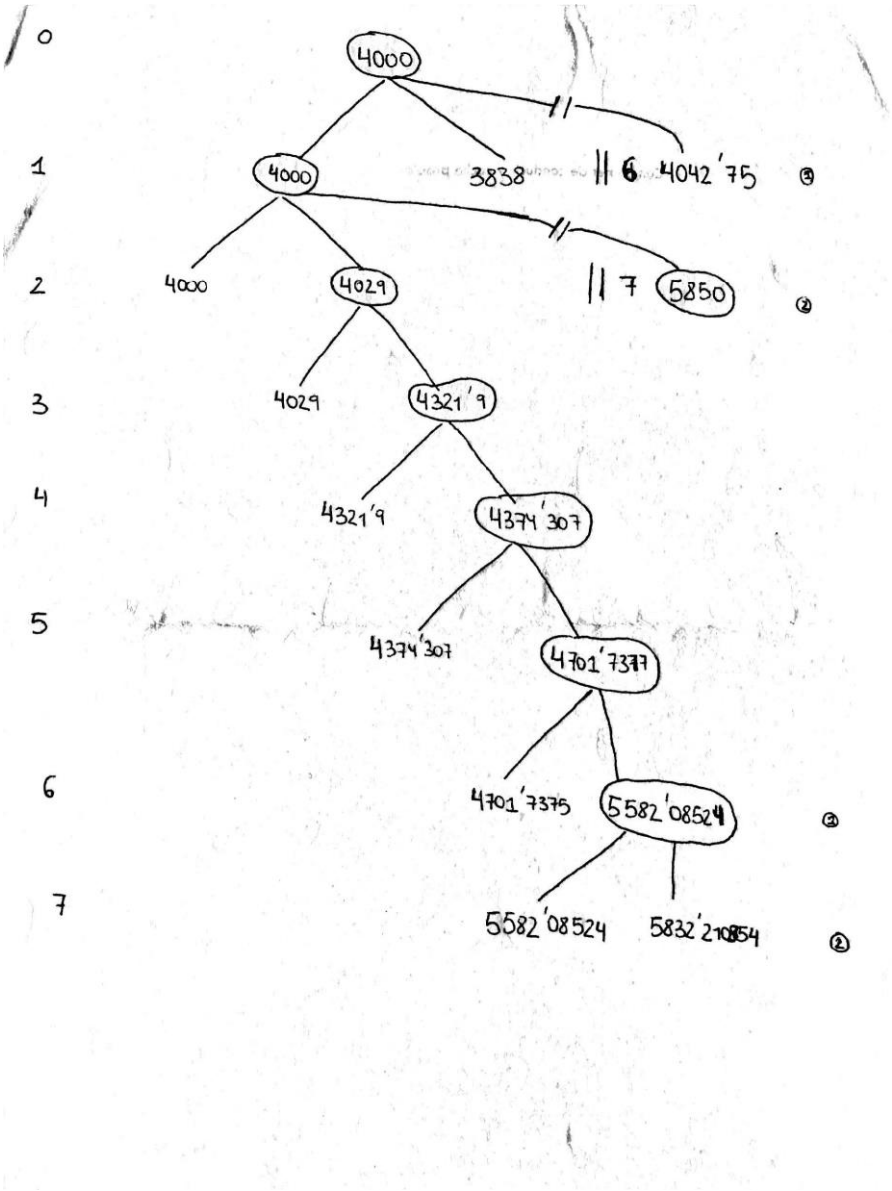
GCD: {200,50,100,75,100,50,80}

RCD: {1.01,1.02,1.1,1.03,1.1,1.2,1.06}

GBT: {75,100,200,50,200,50,75}

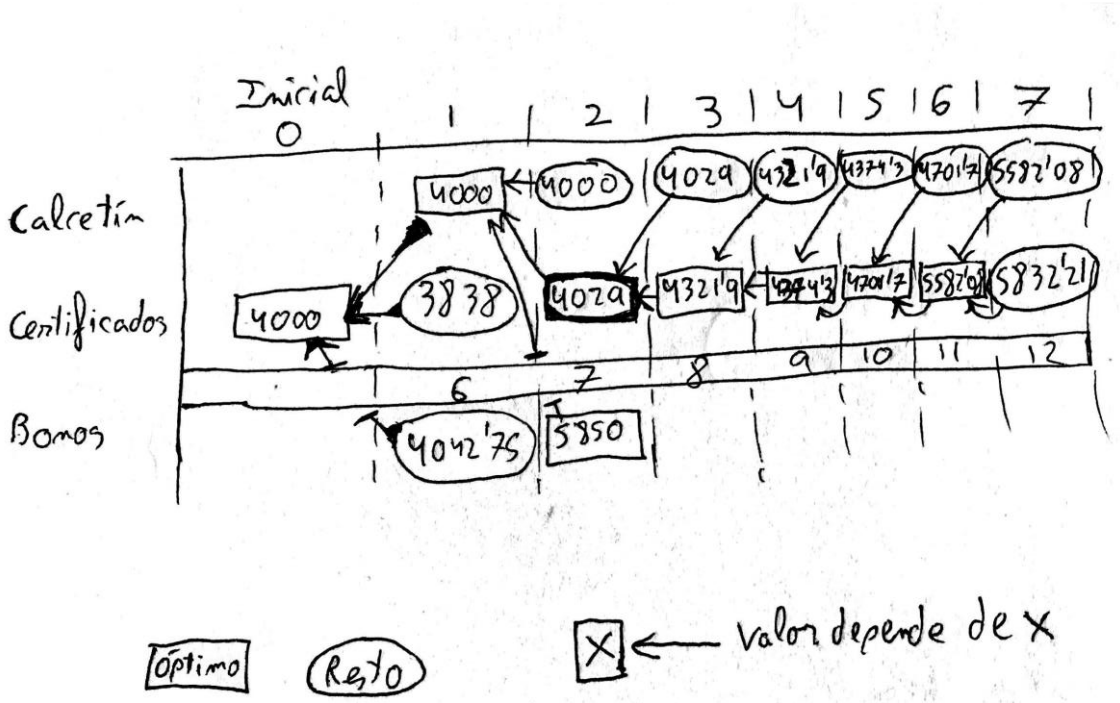
RBT: {1.03,1.5,1.3,1.2,1.02,1.2,1.05}

Y una cantidad inicial de 4000, el árbol que se obtiene es:



f) Construcción del grafo de dependencia para dicha llamada.

Para los mismos datos de entrada:



g) Diseño y construcción de la tabla necesaria para eliminar la recursividad de la llamada anterior.

Para los mismos datos de entrada:

Beneficio obtenible en N meses(0 = inversión inicial)							
0	1	2	3	4	5	6	7
4000	Max(4000, 3838) = 4000	Max(4000, 4029) = 4029	Max(4029, 4321.9) = 4321.9	Max(4321.9, 4374.3) = 4374.3	Max(4374.3, 4701.7) = 4701.7	Max(4701.7, 5582.08, 4042.75) = 5582.08	Max(5582.08, 5832.21, 5850) = 5850

h) Implementación en Java del algoritmo iterativo completo.

Para facilitar el desarrollo de estos tres últimos apartados, se ha implementado una clase, ProgramacionDinamica.java, además de una clase main.java, para la ejecución de cada uno de los métodos.

En concreto, la cabecera del método que se corresponde con este apartado es:

```
public double beneficio (int c_inicial, int[]GCD, double[] RCD, int [] GBT, double [] RBT ){…}
```

i) Implementación en Java del algoritmo para recuperar la solución.

La cabecera de este método es:

```
public double[] beneficioSolucion (int c_inicial, int[]GCD, double[] RCD, int [] GBT, double [] RBT ){…}
```

La única diferencia con respecto al apartado anterior es que, en este apartado en lugar de devolver el beneficio total obtenido, se devuelve un array donde en cada posición se almacena el beneficio tras haber elegido una de las dos opciones (tres a partir del sexto mes) cada mes, de forma que se puede obtener de forma secuencial la solución.

j) Justificar si es posible reducir la memoria en el algoritmo iterativo. En caso afirmativo, implementar dicho algoritmo en Java

No hemos sido capaces de encontrar una forma de reducir la memoria, puesto que es necesario el array que almacena el beneficio de cada mes. Si no se incluyese la opción de comprar bonos del tesoro de seis meses, bastaría con almacenar en una variable el beneficio del mes anterior, pero como no es así hace falta almacenar el beneficio de todos los meses puesto que, a priori, no sabes si vas a necesitar la cantidad que tenías hace 6 meses para calcular la actual.

3. CUESTIONES TEÓRICAS

a) Comparación de las técnicas de memorización y tabulación para la eliminación de la recursividad.

Técnica de memorización: Aplicación más sencilla vs más consumo de memoria. La idea es guardar todas las soluciones ya calculadas en una tabla para no volver a calcularlas. Cada celda de la tabla se inicializa con un valor v distinto a los que puede devolver nuestra función.

Técnica de tabulación: técnica más eficiente que la de memorización, pero menos directa. La idea es guardar las soluciones ya calculadas necesarias el menor tiempo posible. No se basa en una regla de conversión sino en un método: reordenar los cálculos de forma que los valores f(y) se almacenan durante el tiempo que se necesiten, pero consumiendo el mínimo de memoria.

Básicamente, la diferencia principal de ambas técnicas es el orden de llamadas.

b) ¿Qué condiciones se tienen que cumplir para poder utilizar la Programación Dinámica?

Existen problemas solapados: Hay que formular el problema como algoritmo recursivo.

La solución ha de ser alcanzada a través de una secuencia de decisiones.

Si el principio de Optimalidad no es aplicable, entonces no se puede usar la Programación Dinámica.

c) ¿Qué similitudes hay con la Técnica de Vuelta Atrás?

Como se ha comentado en el apartado en la Programación Dinámica, así como también ocurría en la Técnica de Vuelta Atrás, la solución debe ser alcanzada a través de una secuencia de decisiones.

d) Enuncia el principio de optimalidad de Bellman. Pon dos ejemplos: uno en el que se cumpla el principio y otro donde no, justificando tu respuesta.

La solución óptima de cualquier caso no trivial de un problema es una combinación de soluciones óptimas de algunos subcasos.

Un ejemplo en el que se cumple este principio es el del camino más corto entre A y B, pasando por C. Si la distancia entre A y C es mínima, entonces la distancia de C a B también será mínima.

Un ejemplo en el que no se cumple es el camino más rápido entre A y B pasando por C. No se puede garantizar que siendo la subsolución entre A y C óptima lo sea también la subsolución entre C y B.

4. CONCLUSIONES

Tras resolver esta práctica nos hemos dado cuenta que la presunta complejidad que presumíamos que íbamos a tener no ha sido tal, si no que una vez entendidos los problemas, la resolución de los mismos salió sola. Respecto al consumo de memoria, sí que es cierto que al eliminar la recursividad se mejoran notablemente la cantidad de memoria utilizada. Sin embargo, bien es cierto que con los costes actuales de memoria Y los costes actuales de capacidad de procesamiento no vemos tan claro la mejora que supone ahorrarnos unos cuantos kilobytes frente al consumo de capacidad de cómputo que aplicaría realizar esa comprobación.