J. Ángel Velázquez Iturbide
Isidoro Hernán Losada

Evaluación del Aprendizaje de la Programación Dinámica Usando Bosques de Recursión

Número 2015-06

Serie de Informes Técnicos DLSI1-URJC
ISSN 1988-8074
Departamento de Lenguajes y Sistemas Informáticos I
Universidad Rey Juan Carlos

Índice

1 Introducción 5
2 Protocolo de la Evaluación 6
3 Análisis y Resultados 7
3.1 Método de Análisis 7
3.2 Resultados 7
4 Resumen de Hallazgos 11
5 Conclusiones 12
Agradecimientos. 12
Referencias 12
Apéndice A: Enunciado de la Práctica 14
Apéndice B: Material del Grupo Experimental 17
Apéndice C: Material del Grupo de Control 1

Evaluación del Aprendizaje de la Programación Dinámica Usando Bosques de Recursión

J. Ángel Velázquez Iturbide, Isidoro Hernán Losada

Departamento de Lenguajes y Sistemas Informáticos I, Universidad Rey Juan Carlos, C/ Tulipán s/n, 28933, Móstoles, Madrid
{angel.velazquez,isidoro.hernan}@urjc.es

Resumen. SRec es un sistema para de visualización de la recursividad. Este informe presenta los resultados de una evaluación del uso docente de los bosques de recursión, soportados por SRec, para ayudar al diseño de los algoritmos con recursividad múltiple y redundante que son la base para la construcción de algoritmos de programación dinámica. Se describe el protocolo de evaluación utilizado y los resultados obtenidos. En los resultados se observan las principales dificultades de los alumnos para el diseño de esta clase de algoritmos. El informe incluye, como apéndice, el enunciado de la práctica.

Palabras clave: Programación dinámica, SRec, árbol de recursión, bosque de recursión.

1 Introducción

SRec es un sistema de visualización de la recursividad [1]. Es un sistema altamente interactivo concebido como apoyo a la docencia de los algoritmos [2]. Se han desarrollado varias extensiones, orientadas a técnicas de diseño especificas (divide y vencerás [3] y programación dinámica [4][5]).

A lo largo del ciclo de vida de SRec se han realizado diversas evaluaciones de usabilidad [6][7]. En este informe se presenta una evaluación del uso de bosques de recursión como ayuda para el diseño de los algoritmos con recursividad múltiple y redundante que son la base para construir algoritmos de programación dinámica. La evaluación se realizó como parte de una práctica. La estructura del informe es la siguiente. El apartado 2 describe el protocolo utilizado. El apartado 3 presenta el método de análisis y los resultados obtenidos. En el cuarto apartado resumimos los hallazgos de la evaluación. Finalmente, tres apéndices recogen el anunciado de la práctica y el material de estudio suministrado a los alumnos.

2 Protocolo de la Evaluación

Esta evaluación de SRec se realizó en noviembre de 2015, en la asignatura optativa "Algoritmos Avanzados", de cuarto curso del Grado en Ingeniería Informática. Participaron alumnos del grupo presencial del campus de Móstoles.

Los alumnos habían recibido un capítulo anterior sobre la eliminación de la recursividad redundante (dos sesiones de clase más otra en el aula informática para la realización de una práctica). Además, habían recibido dos sesiones de clase dedicadas a la técnica de programación dinámica. En estas sesiones habían visto los fundamentos de la técnica de diseño y la resolución de seis problemas con esta técnica (mochila 0/1, subsecuencia común más larga, alineamiento de secuencias, alineamiento global, grafo multietapa y multiplicación encadenada de matrices).

La evaluación se realizó aprovechando la sesión de laboratorio de la práctica 5a, cuyo objetivo es el diseño e implementación eficiente de un algoritmo de programación dinámica de coste mínimo [9, ejercicio 6.4]. La sesión tenía una duración de dos horas. La realización de la práctica era individual. No se esperaba que los alumnos pudieran acabar la práctica durante la sesión, por lo que tenían un plazo de una semana.

Veamos la organización de la sesión. Primero se les explicó que la participación en las evaluaciones era voluntaria y que el objetivo era mejorar la docencia de la asignatura. Ningún alumno rehusó participar. También se avisó de que la práctica era de realización individual y no se permitía que hablaran. Al final de la sesión, debían entregar el resultado de su trabajo subiendo un fichero al campus virtual. Aunque no es el informe final de la práctica, por concreción llamaremos informe a esta entrega.

Los alumnos se dividieron en dos grupos (grupo experimental y grupo de control), que se distribuyeron en aulas informáticas distintas. Los grupos se diferenciaron por el material de estudio recibido. Ambos pudieron descargarse del campus virtual un fichero sobre las primeras etapas de resolución del problema de cambio de monedas [10, cap. 8]: enunciado del problema, organización de una solución en etapas, ecuaciones recursivas, codificación en Java, y árboles de recursión. La diferencia entre ambos grupos consistió en que el grupo experimental recibió un bosque de recursión explicado, mientras que el grupo de control sólo recibió un árbol de recursión explicado.

Se presentaron 14 alumnos del grupo experimental y 17 del grupo de control. Hubo 1 alumno del grupo experimental y 2 del grupo de control que no entregaron nada pero, para el análisis de contenidos, se consideran en la misma categoría que los informes que estaban en blanco o no contenían nada constructivo. Un alumno del grupo experimental estuvo en el aula informática correspondiente al grupo de control. Al ser de realización individual, su informe se ha evaluado dentro del grupo experimental.

Incluimos el enunciado de la práctica en el Apéndice A y el material entregado a ambos grupos en los Apéndices B y C.

3 Análisis y Resultados

Primero presentamos el método de análisis para presentar a continuación los resultados del análisis.

3.1 Método de Análisis

Para el análisis se siguió el siguiente proceso:

1. Se hizo una primera identificación de aspectos clave a partir de los elementos que los alumnos aportaban en sus informes (ecuaciones, código Java, visualizaciones, otros algoritmos, etc.).
2. Primera ronda. Durante la misma se desglosó el diseño de los algoritmos recursivos en sus elementos. También tomaron forma las categorías de los algoritmos y se incorporó la hora de entrega de los informes en el campus virtual.
3. Segunda ronda. Se suprimieron la mayor parte de los elementos de los algoritmos recursivos, ya que estaban implícitos en las categorías.
4. Tercera ronda. Durante la elaboración del informe se vio la conveniencia de analizar con más detalle la parte de las ecuaciones.

3.2 Resultados

Se identificaron 7 categorías de algoritmos recursivos:

- Algoritmo correcto. Son algoritmos que parecen ser correctos. Se organizan en etapas (meses) y por las alternativas de cada etapa (las dos sedes).
- Algoritmo casi correcto. Son algoritmos que no son correctos por pequeños detalles pero que, una vez arreglados, lo estarían. Por ejemplo, que no se consideren las dos opciones en la primera etapa (o sus costes).
- Algoritmo tipo voraz. Son algoritmos que realizan una decisión local y sólo contiene una llamada recursiva para continuar por las etapas siguientes.
- Algoritmo tipo vuelta-atrás. Son algoritmos programados que, aunque tienen un comportamiento parecido al del algoritmo recursivo, propagan la solución al modo de los algoritmos de vuelta atrás. Su estilo de programación también es similar al uso en la asignatura para esta técnica de diseño. Sin embargo, tampoco aplican bien esta técnica, sino que cometen diversos errores.
- Mezcla. Son algoritmos que no contienen una diferenciación clara entre las etapas y las alternativas de cada etapa.
- Organización. No aportan ningún algoritmo pero realizan una descripción de cómo organizar el proceso recursivo. (Un alumno del grupo experimental intentará diseñar las ecuaciones pero las deja incompletas.)
- Nada. No contienen nada aprovechable, bien porque no entregaron nada, el informe está en blanco o lo que presentan no tiene apenas valor.

La Tabla 1 muestra el número y porcentaje de informes pertenecientes a cada categoría en cada grupo.

Tabla 1. Categorías de algoritmos recursivos

	Grupo experimental		**Grupo de control**	
	#	**%**	**#**	**%**
Correcto	1	7,14%	3	17,65%
Casi correcto	3	21,43%	3	17,65%
Tipo voraz	4	28,57%	2	11,76%
Tipo vuelta atrás	1	7,14%	1	5,88%
Mezcla	3	21,43%	–	–
Organización	1	7,14%	2	11,76%
Nada	1	7,14%	6	35,29%
Total	**14**	**100%**	**17**	**100%**

Desde el punto de vista de la entrega de algoritmos, el 85'71% de alumnos del grupo experimental han entregado algún algoritmo de Java, mientras que en el grupo de control el porcentaje se reduce al 52'94%.

Las soluciones presentadas son "hacia adelante", encontrando solamente un informe con una solución "hacia atrás".

Dada la variedad de categorías, resulta clarificador agruparlas según la viabilidad de sus soluciones:

- Grupo viable: categorías "correcto" o "casi correcto". Su solución está bien organizada y, con tiempo suficiente, es probable que todos ellos corrigieran sus errores.
- Grupo intermedio: categorías "tipo voraz" o "tipo vuelta atrás". Su solución tiene problemas graves pero han sido capaces de realizar una organización del algoritmo en etapas y alternativas, y han realizado una implementación.
- Grupo con dificultades grandes: categorías "mezcla" u "organización". Su solución tiene problemas mayores porque no son capaces de realizar una organización del algoritmo en etapas y alternativas, o de implementarlo.
- Nada. No han entregado nada constructivo.

La Tabla 2 muestra la distribución de informes por grupos de categorías.

Tabla 2. Grupos de categorías de algoritmos recursivos

	Grupo experimental		**Grupo de control**	
	#	**%**	**#**	**%**
Viable	4	28,57%	6	35,29%
Intermedio	5	35,71%	3	17,65%
Dificultades grandes	4	28,57%	2	11,76%
Nada	1	7,14%	6	35,29%
Total	**14**	**100%**	**17**	**100%**

Es difícil sacar conclusiones claras. El grupo de control tiene más soluciones viables que el grupo experimental, pero no parece que la diferencia sea significativa. Sin embargo, hay más diferencias en los grupos intermedio y viable (con diferencias

de más del 15%) y, sobre todo, en la categoría de "nada" donde la diferencia es casi del 30%.

Por otro lado, hay algunos alumnos que se dan cuenta de que su solución presenta problemas e informan de ello. Dos alumnos del grupo experimental avisan de que no es óptimo (categoría tipo voraz y mezcla) y otro, de que está inacabado (mezcla). Otro alumno del grupo de control avisa de que probablemente debería haberlo enfocado de manera distinta (tipo vuelta atrás).

Veamos el rendimiento de cada grupo en los distintos elementos pedidos. La Tabla 3 presenta el número y porcentaje de alumnos que han propuesto ecuaciones recursivas.

Tabla 3. Informes con ecuaciones

	Grupo experimental		**Grupo de control**	
	#	**%**	**#**	**%**
Con ecuaciones	7	50%	7	41,18%
Sin ecuaciones	7	50%	10	58,82%
Total	**14**	**100%**	**17**	**100%**

Las ecuaciones propuestas no están exentas de errores e inexactitudes: falta el caso básico o el caso recursivo, se utilizan identificadores no explicados, se utilizan notaciones forzadas, etc. En la Tabla 4 presentamos si se suprimen los parámetros que no varían.

Tabla 4. Ecuaciones con los parámetros que no varían eliminados

	Grupo experimental		**Grupo de control**	
	#	**%**	**#**	**%**
Sin parámetros constantes	4	57,14%	2	28,57%
Con parámetros constantes	3	42,86%	5	71,43%
Total	**7**	**100%**	**7**	**100%**

Asimismo, en Tabla 5 presentamos si las ecuaciones son coherentes con los algoritmos Java correspondientes (no siempre están presentes ambos). En un caso no hay información suficiente para determinar si hay coherencia ya que las ecuaciones no incluyen el caso recursivo.

Tabla 5. Coherencia entre ecuaciones y algoritmos en Java

	Grupo experimental		**Grupo de control**	
	#	**%**	**#**	**%**
Coherencia	4	66,67%	5	71,43%
Incoherencia	2	33,33%	1	14,29%
Información insuficiente	–	–	1	14,29%
Total	**6**	**100%**	**7**	**100%**

Solamente nos ha parecido interesante ver dos aspectos adicionales de las soluciones de Java: si se incluye la llamada inicial desde el método principal al algoritmo recursivo y si se utiliza un parámetro acumulador para propagar la solución

mejor entre llamadas recursivas. La Tabla 6 muestra quiénes han hecho un mal uso de estos elementos, entre los que han entregado algún algoritmo en Java.

Tabla 6. Algoritmos en Java sin llamada inicial o con parámetro acumulador

	Grupo experimental		Grupo de control	
	#	%	#	%
Sin llamada inicial	1 (de 12)	8,33%	1 (de 9)	11,11%
Con parámetro acumulador	2 (de 12)	16,67%	3 (de 9)	33,33%

Veamos algunos elementos adicionales. Algunos alumnos presentaron sus informes sin explicación alguna. La Tabla 7 muestra los resultados sobre este aspecto.

Tabla 7. Informes con explicaciones

	Grupo experimental		Grupo de control	
	#	%	#	%
Con explicaciones	6	42,86%	11	64,71%
Sin explicaciones	8	57,14%	6	35,29%
Total	**14**	**100%**	**17**	**100%**

Un alumno del grupo de control (categoría correcto) realiza una explicación de su diseño en términos de tareas y actividades, más propios de otros problemas.

La Tabla 8 muestra quiénes presentaron un árbol de recursión correspondiente al algoritmo recursivo diseñado.

Tabla 8. Informes con árbol de recursión

	Grupo experimental		Grupo de control	
	#	%	#	%
Con árbol de recursión	6	42,86%	4	23,53%
Sin árbol de recursión	8	57,14%	13	76,47%
Total	**14**	**100%**	**17**	**100%**

Uno de los informes del grupo experimental presenta un árbol de recursión tras invocar la función de búsqueda de nodos, para comprobar que hay redundancia.

Por último, la Tabla 9 muestra las apariciones de grafos de dependencia.

Tabla 9. Informes con grafos de dependencia

	Grupo experimental		Grupo de control	
	#	%	#	%
Con grafo de dependencia	5	35,71%	4	23,53%
Sin grafo de dependencia	9	64,29%	13	76,47%
Total	**14**	**100%**	**17**	**100%**

Hay incluso 2 alumnos del grupo experimental (con solución tipo voraz) que llegan a codificar un algoritmo tabulado, analizan su complejidad y lo amplían para que determine las decisiones óptimas.

Un último dato es el tiempo medio de entrega del informe parcial por medio en el campus virtual, que es las 16:47 para el grupo experimental y las 16:54 para el grupo de control.

4 Resumen de Hallazgos

Podemos resumir los resultados de la evaluación como sigue:

- Han entregado un algoritmo en Java el 85'71% de alumnos del grupo experimental y el 52'94% del grupo de control.
- Hemos identificado 7 categorías de algoritmos, con distribución dispar entre ambos grupos: correcto, casi correcto, tipo voraz, tipo vuela atrás, mezcla, organización y nada.
- Las categorías pueden agruparse en 4 grupos de viabilidad, definidos según la capacidad para especificar el algoritmo. El grupo de control tiene un porcentaje algo mayor que el experimental de categorías viables (35,29% vs. 28,57%) y bastante mayor de alumnos sin ninguna solución (35,29% vs. 7,14%). El grupo experimental tiene un porcentaje mayor que el de control de alumnos con viabilidad intermedia (35,71% vs. 17,65%) o con grandes dificultades (28,57% vs. 11,76%).
- Han entregado ecuaciones el 50% de alumnos del grupo experimental y 41,18% del grupo de control.
- El diseño de las ecuaciones presenta problemas, sobre todo de claridad por no suprimir los parámetros constantes (42'86% del grupo experimental y 71,43% del grupo de control). Además, cerca de un tercio de los alumnos de cada grupo no mantienen la coherencia entre las ecuaciones y el algoritmo de Java.
- El porcentaje de alumnos que han explicado sus productos es medio: 42,86% del grupo experimental y 64,71% del grupo de control.
- El porcentaje de alumnos que han continuado obteniendo visualizaciones del algoritmo recursivo (árboles de recursión y grafos de dependencia) oscila entre 23,53% y 42,86%. Aún menor es el porcentaje de alumnos que han desarrollado un algoritmo tabulado.
- Los alumnos del grupo experimental han realizado sus entregan en un tiempo medio inferior en 7 minutos que los alumnos del grupo de control.

Dando una interpretación cualitativa a estos hallazgos, podríamos afirmar lo siguiente:

- La mayor parte de los alumnos han sido capaces de desarrollar un algoritmo recursivo. Porcentualmente, ha sido menor el número de alumnos que han diseñado ecuaciones o explicado sus diseños.
- Las principales dificultades de los alumnos para el diseño de algoritmos recursivos de programación dinámica se deben a dos factores: organización del algoritmo en etapas y alternativas, y codificación del algoritmo.
- Los resultados de la evaluación no permiten plantear conclusiones definitivas sobre las ventajas de uso docente de los bosques de recursión, por lo que

convendría realizar más evaluaciones. Los resultados más claros parecen ser que hay un porcentaje significativamente mayor de alumnos sometidos a árboles de recursión (en lugar de bosques) que han dejado su solución en blanco. Una posible interpretación es que los alumnos sometidos a los bosques de recursión han podido apreciar mejor las diferencias entre los distintos árboles, identificando en consecuencia mejor la organización en etapas y alternativas.

- Los alumnos parecen tener mayor dificultad para diseñar ecuaciones recursivas que algoritmos recursivos en Java. Una posible explicación es que su diseño exige un mayor nivel de abstracción, por lo que resulta más difícil que el algoritmo correspondiente (sin embargo, para un experto podría ser al revés).
- El uso de SRec y una metodología de eliminación de la recursividad redundante permiten que una minoría destacada de alumnos puedan llegar a obtener visualizaciones e incluso algoritmos tabulados. Desde un punto de vista docente, conviene resaltar aún más (si cabe) que la recursividad lineal en propia de la técnica voraz, no de la programación dinámica. Por tanto, si se diseña un algoritmo recursivo lineal, al menos deberían ser conscientes de que no es óptimo.

5 Conclusiones

Hemos presentado de forma detallada una evaluación del uso de bosques de recursión como ayuda al diseño de algoritmos de programación dinámica realizada en noviembre de 2015. Se ha incluido el procedimiento usado, el método de análisis, los resultados y una breve valoración de los mismos. Los resultados han sido positivos para conocer las dificultades de los alumnos relacionadas con el diseño de esta clase de algoritmos.

Agradecimientos. Este trabajo se ha financiado con los proyectos TIN2011-29542-C02-01 del Ministerio de Economía y Competitividad de España y S2013/ICE-2715 de la Comunidad Autónoma de Madrid.

Referencias

1. Velázquez-Iturbide, J.Á., Pérez-Carrasco, A., Urquiza-Fuentes, J.: SRec: An animation system of recursion for algorithm courses. En: Proceedings of the 13th Annual Conference on Innovation and Technology in Computer Science Education, ITiCSE 2008. ACM Press, New York (2008) 225–229, DOI 10.1145/1384271.1384332
2. Velázquez-Iturbide, J.Á., Pérez-Carrasco, A.: InfoVis interaction techniques in animation of recursive programs. Algorithms 3, 1 (2010) 76-91, DOI 10.3390/a3010076
3. Velázquez-Iturbide, J.Á., Pérez-Carrasco, A., Urquiza-Fuentes, J.: A design of automatic visualizations for divide-and-conquer algorithms. Electronic Notes in Theoretical Computer Science 224 (2009) 159-167, DOI 10.1016/j.entcs.2008.12.060

4. Velázquez-Iturbide, J.Á., Pérez-Carrasco, A.: Familias de visualizaciones de los árboles de recursión. En: SIIE13 XV International Symposium on Computers in Education – Proceedings, M.J. Marcelino, M.C. Azebedo Gomes y A.J. Mendes (eds.) (2013) 18-23
5. Velázquez-Iturbide, J.Á., Pastor Herranz, D., Pérez-Carrasco, A.: La visualización interactiva como apoyo al desarrollo de algoritmos de programación dinámica. En: Atas do XVII Simpósio Internacional de Informática Educativa (SIIE'15), M.R. Rodrigues, M. Llamas Nistal y M. Figueiredo (eds.) (2015) 195-201
6. Velázquez-Iturbide, J.Á., Pérez-Carrasco, A., Urquiza-Fuentes, J.: Multiple usability evaluations of a program animation tool. En: The 10th IEEE International Conference on Advanced Learning Technologies, M. Jemni, Kinshuk, D. Sampson y J.M. Spector (eds.) (2010) IEEE Computer Society, 452-454, DOI 10.1109/ICALT.2010.131
7. Velázquez Iturbide, J.Á., Pérez Carrasco, A., Debdi, O.: Experiences in usability evaluation of educational programming tools. En: Student Usability in Educational Software and Games: Improving Experiences, C. González (ed.), IGI Global (2013) 241-260, DOI 10.4018/978-1-4666-1987-6
8. Velázquez Iturbide, J.Á.: Evaluaciones sexta y séptima de usabilidad de SRec. En: Serie de Informes Técnicos DLSI1-URJC, nº 2015-04, Departamento de Lenguajes y Sistemas Informáticos I, Universidad Rey Juan Carlos, 2015, 41 págs.
9. Kleinberg, J., Tardos, É.: Algorithm Design. Pearson Addison-Wesley, 2006
10. Brassard, G., Bratley, P.: Fundamentals of Algorithmics. Englewood Cliffs, NJ: Prentice-Hall, 1996

Apéndice A: Enunciado de la Práctica

Grado en Ingeniería Informática
Asignatura *Algoritmos Avanzados*
Curso 2015/2016
Práctica nº 5.a

Objetivo

El objetivo de la práctica es que el alumno practique la técnica de programación dinámica.

Carácter

La práctica es voluntaria. Debe realizarse individualmente.

Enunciado

En la práctica 2 se planteó el *problema del plan de sedes de coste mínimo*. Suponiendo que la planificación se realiza para 2 sedes ($s0$ y $s1$) y n meses, el objetivo consiste en decidir en qué sede debe el negocio concentrar su actividad durante cada mes de forma que se minimicen los costes de operación.

Por ejemplo, los costes de 4 meses pueden ser c_0={1,3,20,30} y c_1={50,20,2,4}, con un coste de traslado f=10. Un plan con coste mínimo sería {$s0$,$s0$,$s1$,$s1$}, con un coste igual a 1+3+10+2+4=20.

El objetivo de la práctica es desarrollar de forma sistemática un algoritmo de programación dinámica que resuelva el problema planteado. El algoritmo tendrá la misma cabecera usada en las prácticas anteriores para el método principal

```
public static int sedes (int[] c0, int[] c1, int f)
```

donde $c0$ y $c1$ son los vectores de costes mensuales de las sedes $s0$ y $s1$, respectivamente, y f es el coste fijo de cada cambio de sede. Para el ejemplo anterior, la llamada del método principal será *sedes*({1,3,20,30},{50,20,2,4},10) y su resultado, 20.

Se sugiere seguir el siguiente método de trabajo:

1. Leer cuidadosamente el enunciado de la práctica, incluyendo las instrucciones de entrega.
2. Analizar el proceso de diseño del ejemplo suministrado como material complementario del enunciado de la práctica.
3. Aplicar dicho proceso de diseño al problema aquí planteado del plan de sedes de coste mínimo.

Entrega
Debe entregarse un informe completo elaborado siguiendo el índice detallado a continuación. El informe debe enviarse por medio del apartado de Evaluación del campus virtual. Si se tienen dificultades, puede enviarse por el correo del campus virtual con el asunto "Práctica 5a". El plazo de entrega del informe es el domingo 29 de noviembre de 2015, incluido.

Esta práctica contiene una participación voluntaria en una actividad docente, que el profesor explicará. Tal y como aparece en las normas de la asignatura, los alumnos que acepten participar tendrán un incremento de 0'25 puntos sobre la nota de la asignatura ya aprobada. Dichos alumnos deberán entregar, antes de abandonar la clase, un **informe parcial** con los puntos que hayan podido realizar del informe detallado a continuación. La entrega se realizará por medio del apartado de Evaluación del campus virtual.

Informe
El alumno debe entregar un informe con la siguiente estructura:

1. **Algoritmo recursivo.** Se incluirá una solución recursiva, expresada en dos formatos distintos:
   a. Ecuaciones recursivas, con notación libre, acompañadas de una breve explicación.
   b. Algoritmo recursivo codificado en Java.
2. **Análisis de la redundancia.** Se incluirá:
   a. Un árbol de recursión.
   b. Grafo de dependencia correspondiente.

   Ambos gráficos deben ser representativos del comportamiento de la función *sedes*. Pueden producirse con el sistema SRec (disponible en el campus virtual) o de cualquier otra forma (manualmente o con otro programa).
3. **Tabulación.** Se incluirá:
   a. Representación gráfica de una tabla adecuada para la función recursiva, de forma que se aprecie claramente: el número de celdas en cada dimensión y el subproblema que se almacenará en cada celda de la tabla. Si se considera conveniente, puede incluirse la declaración en Java de la tabla, expresada en función de los parámetros.
   b. Código del algoritmo de programación dinámica, resultante de la tabulación.
   c. Análisis de complejidad del algoritmo de programación dinámica en tiempo y en espacio.
4. **Determinación de decisiones.** Se ampliará el algoritmo del apartado anterior de forma que, junto al valor del beneficio óptimo, imprima las actividades correspondientes.
5. **Conclusiones.** Se explican las conclusiones obtenidas tras realizar la práctica. Estas conclusiones pueden consistir en una valoración de la técnica de programación dinámica, de alguna fase concreta o cualquier otro comentario sobre la práctica

(incidencias que han dificultado la realización de la práctica, sus aspectos más atractivos o más difíciles, sugerencias sobre cómo mejorar la práctica, etc.)

Evaluación

Se evaluará la calidad y claridad de todos los apartados del informe.

Apéndice B: Material del Grupo Experimental

Problema del cambio de monedas

Enunciado del problema

Se quiere cambiar una cantidad de dinero *c* en un número mínimo de monedas de cierto conjunto *u* de unidades monetarias. Por ejemplo, si *u* = {25,5,1} y *c* = 10, el cambio óptimo es {0,2,0}, es decir, 2 monedas de 5.

Si las unidades monetarias usadas cumplen ciertas condiciones, puede usarse el algoritmo voraz que considera las unidades monetarias en orden decreciente de valor, tomando todas las monedas posibles de cada una. Ahora bien, el algoritmo voraz no da la solución óptima para algunos conjuntos de unidades monetarias. Por ejemplo, si *u*={7,5,1} y *c*=10, el criterio voraz calcularía el desglose de 4 monedas {1,0,3}, cuando el desglose óptimo es {0,2,0} con 2 monedas.

Se quiere diseñar un algoritmo de *programación dinámica* que resuelva el problema del cambio de moneda en general. La unidad monetaria más pequeña u_n vale 1 para que al menos haya un cambio posible.

Organización de la solución y representación de los problemas

La solución se va construyendo paso a paso:

- Las etapas son las unidades monetarias, desde las unidades monetarias mayores a la menores. En el ejemplo anterior, se considerarán sucesivamente las unidades monetarias de valor 7, 5 y 1.
- En cada etapa, probamos todas las posibilidades de cambio con la unidad monetaria correspondiente: no tomar ninguna moneda, tomar una y así sucesivamente hasta probar el máximo posible. Por ejemplo, si quedan 5 céntimos por cambiar con monedas de dos céntimos, hay tres posibilidades: 0, 1 y 2 monedas.

En este proceso, va variando la unidad monetaria a considerar en cada paso y la cantidad de dinero que queda por cambiar. Por tanto, definimos una función numérica *cm*(*i*,*x*) que representa el número mínimo de monedas con las que se puede cambiar una cantidad *x* cuando las unidades monetarias que falta por considerar son las comprendidas entre *i* y *n* ($0 \leq i \leq n$), siendo *n* = *u.length*-1.

La llamada inicial debe ser *cm*(0,*c*) ya que se desea cambiar una cantidad total *c* considerando todas las unidades monetarias, es decir, desde la primera.

Algoritmo recursivo

La función *cm*(*i*,*x*) se define de la forma siguiente:

- Caso básico. Corresponde a la última unidad monetaria u_n, de valor uno. Se cambian tantas monedas de un céntimo como dinero *x* quede.

– Caso recursivo. Para la unidad monetaria x_i, existe un límite en el número de monedas que pueden tomarse. Por tanto, se consideran todas las posibilidades, desde 0 hasta dicho máximo, quedándonos con la mejor opción, es decir, la que conduce a un número mínimo de monedas.

Combinando ambos casos en una definición recursiva de *cm*, queda:

$$cm(n,x) = x \qquad para\ 0 \le x \le c$$
$$cm(i,x) = \min\{j + cm(i+1, x - j\cdot u_i\} \qquad para\ 1 \le i \le n-1,\ 0 \le j \le x/u_i$$

Algoritmo recursivo en Java

El algoritmo recursivo anterior puede codificarse fácilmente en Java. Suponemos que las unidades monetarias se representan como un vector *u* ordenado decrecientemente. Por ejemplo, las unidades del sistema monetario del euro, en céntimos, son {200, 100, 50, 20, 10, 5, 2, 1}.

El algoritmo queda:

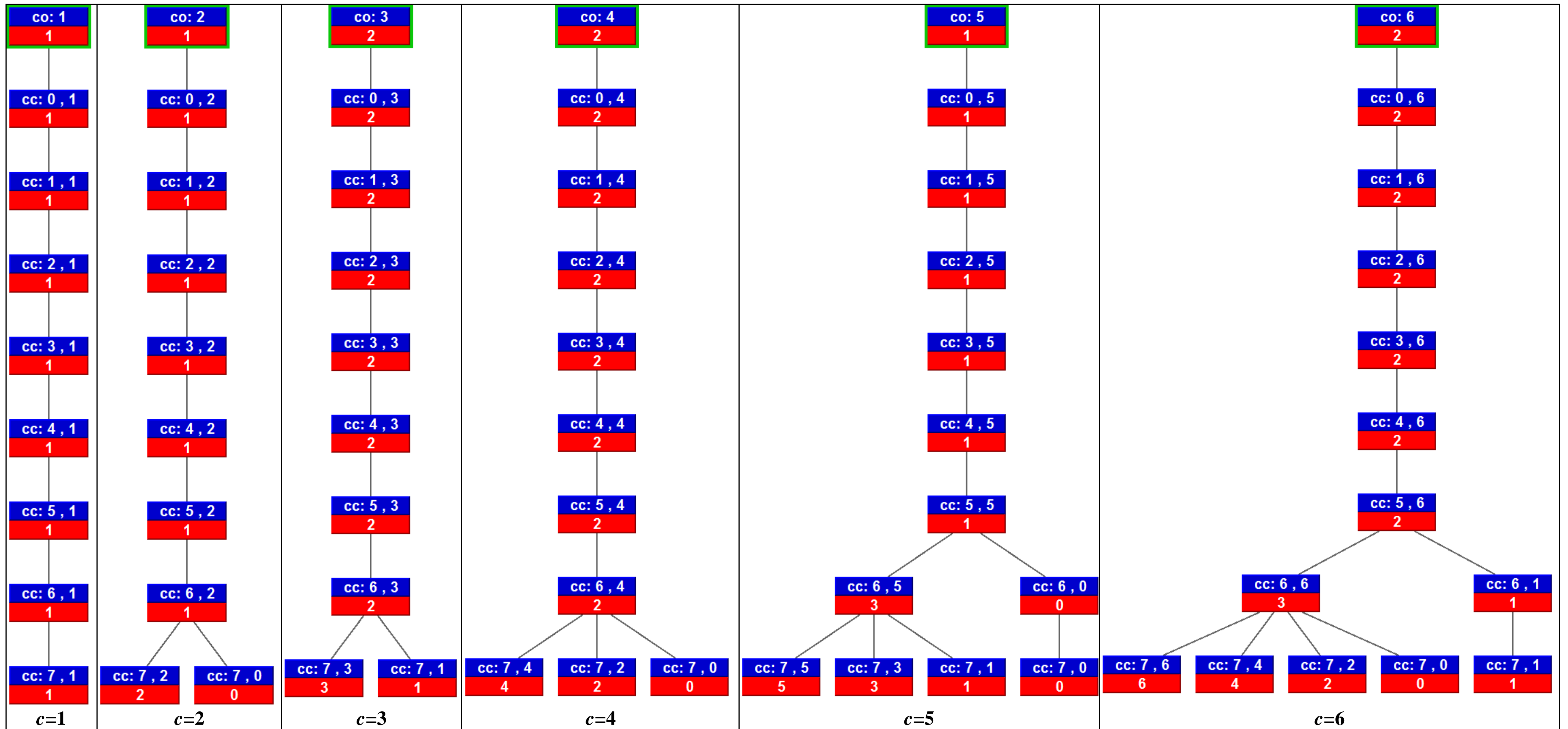
```
public static int cambiarMonedas (int c, int[] u) {
   return cm(0,c,u);
}

private static int cm (int i, int x, int[] u) {
   if (i==u.length-1)
      return x;
   else {
      int menor = Integer.MAX_VALUE;
      for (int j=0; j<=x/u[i]; j++) {
         int result = j + cm(i+1,x-j*u[i],u);
         if (result < menor)
            menor = result;
      }
      return menor;
   }
}
```

Árboles de recursión

Se muestran los árboles de recursión que surgen al cambiar varias cantidades, desde 1 hasta 6 céntimos de euro. El nodo superior de cada árbol corresponde a la llamada al método principal *cambiarMonedas* (etiquetado *ca* por brevedad) y los nodos que hay por debajo representan llamadas al método auxiliar *cm*. No se muestra el vector de unidades monetarias porque su contenido no varía. Cada unidad monetaria se considera en un nivel distinto, desde el nivel superior con *i*=0 (monedas de 200 céntimos, es decir, 2€) hasta el nivel inferior con *i*=7 (monedas de 1 céntimo).

Obsérvese que el árbol más sencillo, para *c*=1, muestra un solo cambio válido: 1 moneda de un céntimo. Para *c*=2, hay dos alternativas: cambiar en 1 moneda de dos céntimos o en 2 monedas de un céntimo. Para *c*=3, sigue habiendo dos alternativas (aunque distintas), pero para *c*=4 hay tres, ya que podemos elegir entre 0, 1 ó 2 monedas de dos céntimos. En el último caso mostrado, con *c*=6, hay cinco formas de

cambiar (en monedas: 1-1-1-1-1-1, 2-1-1-1-1, 2-2-1-1, 2-2-2 ó 5-1), siendo óptimo el último de estos cambios, con sólo dos monedas.



co: 1
1
cc: 0 , 1
cc: 1 , 1
cc: 2 , 1
cc: 3 , 1
cc: 4 , 1
cc: 5 , 1
cc: 6 , 1
cc: 7 , 1
c=1
co: 2
cc: 0 , 2
cc: 1 , 2
cc: 2 , 2
cc: 3 , 2
cc: 4 , 2
cc: 5 , 2
cc: 6 , 2
cc: 7 , 2
cc: 7 , 0
c=2
co: 3
cc: 0 , 3
cc: 1 , 3
cc: 2 , 3
cc: 3 , 3
cc: 4 , 3
cc: 5 , 3
cc: 6 , 3
cc: 7 , 3
cc: 7 , 1
c=3
co: 4
cc: 0 , 4
cc: 1 , 4
cc: 2 , 4
cc: 3 , 4
cc: 4 , 4
cc: 5 , 4
cc: 6 , 4
cc: 7 , 4
cc: 7 , 2
cc: 7 , 0
c=4
co: 5
cc: 0 , 5
cc: 1 , 5
cc: 2 , 5
cc: 3 , 5
cc: 4 , 5
cc: 5 , 5
cc: 6 , 5
cc: 6 , 0
cc: 7 , 5
cc: 7 , 3
cc: 7 , 1
cc: 7 , 0
c=5
co: 6
cc: 0 , 6
cc: 1 , 6
cc: 2 , 6
cc: 3 , 6
cc: 4 , 6
cc: 5 , 6
cc: 6 , 6
cc: 6 , 1
cc: 7 , 6
cc: 7 , 4
cc: 7 , 2
cc: 7 , 0
cc: 7 , 1
c=6

Apéndice C: Material del Grupo de Control

Problema del cambio de monedas

Enunciado del problema

Se quiere cambiar una cantidad de dinero *c* en un número mínimo de monedas de cierto conjunto *u* de unidades monetarias. Por ejemplo, si *u* = {25,5,1} y *c* = 10, el cambio óptimo es {0,2,0}, es decir, 2 monedas de 5.

Si las unidades monetarias usadas cumplen ciertas condiciones, puede usarse el algoritmo voraz que considera las unidades monetarias en orden decreciente de valor, tomando todas las monedas posibles de cada una. Ahora bien, el algoritmo voraz no da la solución óptima para algunos conjuntos de unidades monetarias. Por ejemplo, si *u*={7,5,1} y *c*=10, el criterio voraz calcularía el desglose de 4 monedas {1,0,3}, cuando el desglose óptimo es {0,2,0} con 2 monedas.

Se quiere diseñar un algoritmo de *programación dinámica* que resuelva el problema del cambio de moneda en general. La unidad monetaria más pequeña u_n vale 1 para que al menos haya un cambio posible.

Organización de la solución y representación de los problemas

La solución se va construyendo paso a paso:

- Las etapas son las unidades monetarias, desde las unidades monetarias mayores a la menores. En el ejemplo anterior, se considerarán sucesivamente las unidades monetarias de valor 7, 5 y 1.
- En cada etapa, probamos todas las posibilidades de cambio con la unidad monetaria correspondiente: no tomar ninguna moneda, tomar una y así sucesivamente hasta probar el máximo posible. Por ejemplo, si quedan 5 céntimos por cambiar con monedas de dos céntimos, hay tres posibilidades: 0, 1 y 2 monedas.

En este proceso, va variando la unidad monetaria a considerar en cada paso y la cantidad de dinero que queda por cambiar. Por tanto, definimos una función numérica *cm*(*i*,*x*) que representa el número mínimo de monedas con las que se puede cambiar una cantidad *x* cuando las unidades monetarias que falta por considerar son las comprendidas entre *i* y *n* ($0 \leq i \leq n$), siendo *n* = *u.length*-1.

La llamada inicial debe ser *cm*(0,*c*) ya que se desea cambiar una cantidad total *c* considerando todas las unidades monetarias, es decir, desde la primera.

Algoritmo recursivo

La función *cm*(*i*,*x*) se define de la forma siguiente:

- Caso básico. Se da al alcanzar la última unidad monetaria u_n, de valor uno. Se cambian tantas monedas de un céntimo como dinero *x* quede.

– Caso recursivo. Para la unidad monetaria u_i, existe un límite en el número de monedas que pueden tomarse. Por tanto, se consideran todas las posibilidades, desde 0 hasta dicho máximo, quedándonos con la mejor opción, es decir, la que conduce a un número mínimo de monedas.

Combinando ambos casos en una definición recursiva de *cm*, queda:

$$cm(n,x) = x \qquad para\ 0 \le x \le c$$
$$cm(i,x) = \min\left\{ j + cm(i+1, x - j{\cdot}u_i \right\} \qquad para\ 1 \le i \le n-1,\ 0 \le j \le x/u_i$$

Algoritmo recursivo en Java

El algoritmo recursivo anterior puede codificarse fácilmente en Java. Suponemos que las unidades monetarias se representan como un vector *u* ordenado decrecientemente. Por ejemplo, las unidades del sistema monetario del euro, en céntimos, son {200, 100, 50, 20, 10, 5, 2, 1}.

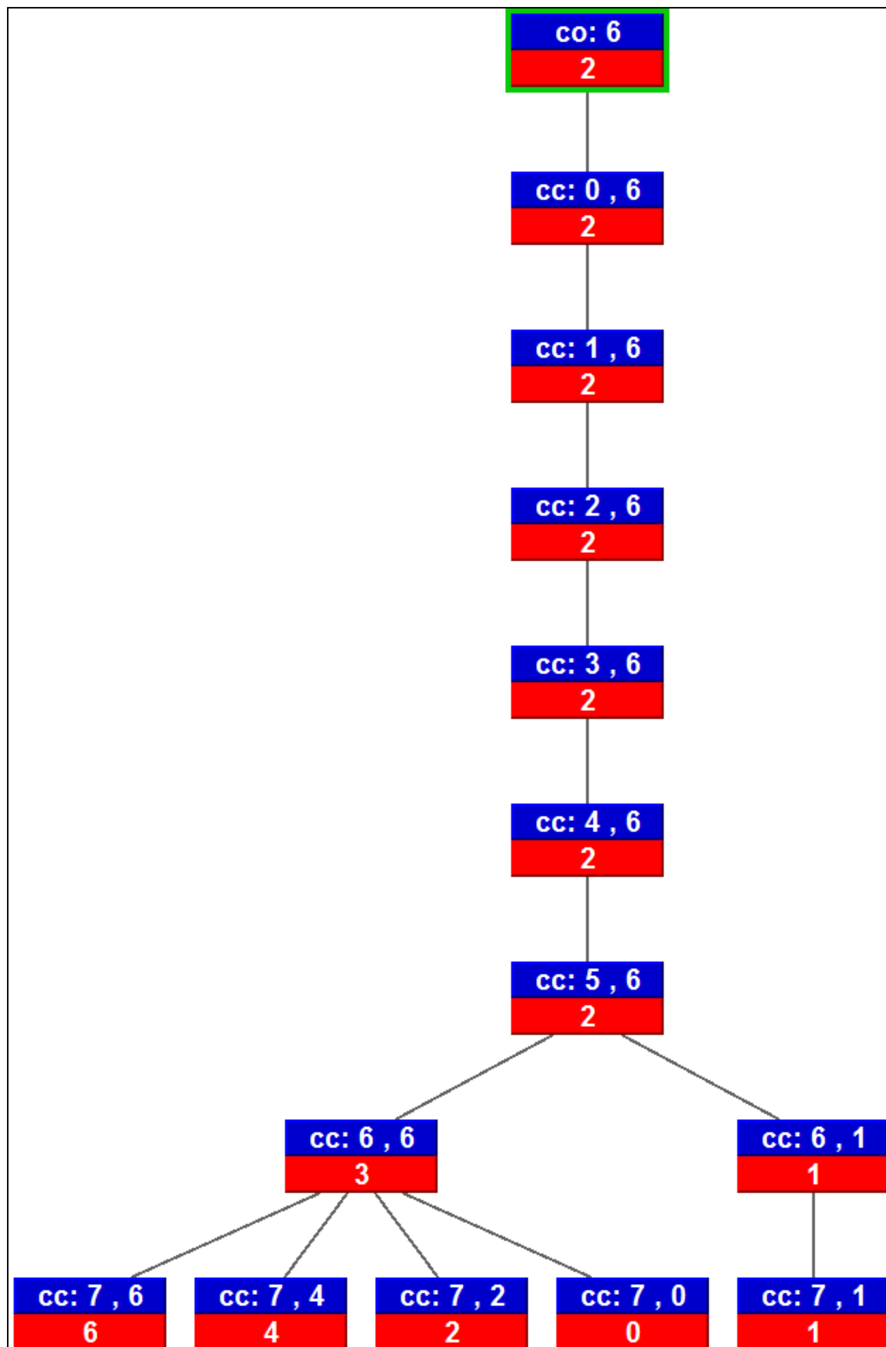
El algoritmo queda:

```
public static int cambiarMonedas (int c, int[] u) {
   return cm(0,c,u);
}

private static int cm (int i, int x, int[] u) {
   if (i==u.length-1)
      return x;
   else {
      int menor = Integer.MAX_VALUE;
      for (int j=0; j<=x/u[i]; j++) {
         int result = j + cm(i+1,x-j*u[i],u);
         if (result < menor)
            menor = result;
      }
      return menor;
   }
}
```

Árboles de recursión

Se muestra el árbol de recursión que resulta al cambiar 6 céntimos de euro. El nodo superior del árbol representa la llamada al método principal *cambiarMonedas* (etiquetado *ca* por brevedad) y los nodos que hay por debajo representan llamadas al método auxiliar *cm*. No se muestra el vector de unidades monetarias porque su contenido no varía. Cada unidad monetaria se considera en un nivel distinto, desde *i*=0 (monedas de 200 céntimos, es decir, 2€) hasta *i*=7 (monedas de un céntimo).

El resultado óptimo para este caso es 2 porque basta con 1 moneda de cinco céntimos y 1 de un céntimo. Hay 5 formas distintas de cambiar, como puede verse en el nivel inferior. Así, en la alternativa de la izquierda aún no se ha cambiado ninguna moneda, por lo que debe cambiarse con 6 monedas de un céntimo. Sin embargo, la alternativa de la derecha corresponde a la solución óptima (2 monedas) porque ya se ha cambiado 1 moneda de cinco céntimos y sólo queda por cambiar 1 céntimo.


co: 6
2
cc: 0 , 6
2
cc: 1 , 6
2
cc: 2 , 6
2
cc: 3 , 6
2
cc: 4 , 6
2
cc: 5 , 6
2
cc: 6 , 6
3
cc: 6 , 1
1
cc: 7 , 6
6
cc: 7 , 4
4
cc: 7 , 2
2
cc: 7 , 0
0
cc: 7 , 1
1