21

Estrategia algorítmica

Contenido

Objetivos del capítulo

- Descubrir características comunes que a los diferentes algoritmos, que nos permitan reconocer la existencia de patrones para clasificarlos según su estrategia de resolución.
- Conocer los principales modelos de estrategia algorítmica: “divide y conquista”, “*greddy*” y “programación dinámica”.

Competencias específicas

- Identificar las características comunes de diferentes algoritmos, para reconocer patrones y facilitar la creación de programas.

21.1 Introducción

Sin saberlo, durante los diferentes capítulos de este libro hemos analizado y resuelto problemas aplicando soluciones cuya estrategia algorítmica responde a ciertos modelos preestablecidos.

En este capítulo estudiaremos algunos de estos modelos de estrategia algorítmica, conocidos como "divide y conquista", "*greddy*" y "programación dinámica", con el objetivo de identificar sus características para reconocer sus patrones.

21.2 Divide y conquista

Tal vez "divide y conquista" sea la técnica natural de la programación estructurada ya que consiste en dividir un problema grande en varios problemas de menor tamaño tantas veces como sea necesario hasta hacer que la resolución de cada uno de ellos sea trivial.

Los algoritmos recursivos también son casos de "divide y conquista", aunque lo que realmente se divide en una llamada recursiva es la cantidad de datos que el algoritmo debe procesar.

Algunos casos típicos de esta técnica son la búsqueda binaria, el Árbol Binario de Búsqueda (ABB) y el algoritmo de ordenamiento *quicksort*. Todos estos algoritmos dividen el conjunto de datos en mitades, luego vuelven a dividir cada una de las mitades y así sucesivamente hasta trivializar la solución.

Recordemos que tanto la búsqueda binaria como el ABB tienen una complejidad de orden logarítmico mientras que la complejidad del ordenamiento *quicksort* es cuasilineal.

21.3 Greddy, algoritmos voraces

Los algoritmos *greddy* encuentran la solución del problema evaluando cada uno de los elementos de un conjunto que llamaremos "conjunto de candidatos".

El algoritmo consiste en una serie de iteraciones, en cada una de las cuales se evalúa un elemento del conjunto de candidatos para determinar si corresponde incluirlo en el conjunto solución o no.

Luego, aquellos elementos no incluidos serán descartados definitivamente. Un algoritmo *greddy* no reconsidera una opción. Simplemente la toma o deja.

En este tipo de algoritmos podemos identificar cuatro funciones:

- La función de *selección* se ocupa de seleccionar el elemento más apropiado del conjunto de candidatos.
- La función de *factibilidad* que, una vez seleccionado un elemento, evalúa si es factible incluirlo en el conjunto solución.
- La función *solución* evalúa si el conjunto solución efectivamente constituye una solución para el problema.
- La función *objetivo* que busca optimizar la solución encontrada.

Esta técnica es simple y básica; y para demostrarlo analizaremos el siguiente problema.

21.3.1 Seleccionar billetes de diferentes denominaciones

Determinar cuántos billetes de 100, 50, 20, 10, 5, 2 y 1 se necesitan para abonar un determinado importe. Se pide también que se utilice la menor cantidad de billetes, dando prioridad a los de mayor denominación.

En este problema el conjunto de candidatos está compuesto por las diferentes denominaciones de los billetes:

```
D = {100, 50, 20, 10, 5, 2, 1}
```

El conjunto solución estará compuesto por pares (*c*,*d*) donde *c* representa una cantidad y *d* representa una denominación, es decir, un elemento del conjunto de candidatos.

Los pares (*c*,*d*) los implementaremos con instancias de la clase `CantBillete` que simplemente tendrá dos atributos (`cant` y `denom`), un constructor y los métodos de acceso.

```
package libro.cap21.greddy;

public class CantBillete
{
   private int denom;
   private int cant;

   public CantBillete(int d, int c)
   {
      denom = d;
      cant = c;
   }

   // :
   // setters y getters
   // :
}
```

Luego, implementaremos el algoritmo como una función (o método) que recibe el importe que se va a pagar y un *array* que contenga las diferentes denominaciones, y retorna una `Collection<CantBillete>` con tantos elementos como billetes de diferentes denominaciones sea necesario utilizar.

```
package libro.cap21.greddy;

import java.util.ArrayList;
import java.util.Collection;

public class Billetes
{
   public static Collection<CantBillete> pagar(int monto,int[] denom)
   {
      int i=0;
      int suma=0;
      int cant=0;

      ArrayList<CantBillete> ret = new ArrayList<CantBillete>();
```

```
            while( i<denom.length && suma<=monto )
            {
               // -- FUNCION DE FACTIBILIDAD --
               // es factible agregar un billete mas de esta denominacion?
               if( suma+denom[i]<=monto )
               {
                  suma+=denom[i];
                  cant++;
               }
               else // no lo es...
               {
                  if( cant>0 )
                  {
                     ret.add( new CantBillete(denom[i],cant) );
                     cant=0;
                  }

                  // -- FUNCION DE SELECCION --
                  // seleccionamos el proximo candidato
                  i++;
               }
            }

            return ret;
         }

         public static void main(String[] args)
         {
            int denom[] = { 100,50,20,10,5,2,1 };

            int importeAPagar = 2543; // hardcodeamos un valor
            Collection<CantBillete> ret = pagar(importeAPagar,denom);

            for(CantBillete a:ret)
            {
               System.out.println(a.getCant()+" billetes de: "+a.getDenom());
            }
         }
      }
```

En este algoritmo la función de factibilidad está dada por la condición:

```
if( suma+denom[i]<=monto )
```

El `if` determina si es factible agregar un billete más de la denominación actual. Si no corresponde entonces, por el `else`, la acción de incrementar el valor de `i` representará a la función de selección. Al incrementar el valor de `i` estamos descartando definitivamente la posibilidad de incorporar más billetes de la denominación actual y pasaremos a evaluar la cantidad de billetes de la próxima denominación. Notemos que es fundamental el hecho de que el *array* esté ordenado decrecientemente.

Aquí podemos considerar que función de selección es la que, luego de cada iteración, incrementa el valor de `i` para pasar a evaluar la siguiente denominación.

21.3.2 Problema de la mochila

Se tiene una mochila cuya capacidad máxima es de *P* kilogramos y un conjunto de objetos o_i, cada uno de los cuales tiene un peso p_i (expresado en kilogramos) y un valor monetario v_i (expresado en pesos).

Se desea desarrollar un algoritmo que indique cuáles de estos objetos deben cargarse en la mochila de forma tal que el peso total no exceda su capacidad *P* y, además, se maximice el valor monetario que se va a transportar.

Los objetos pueden transportarse enteros o pueden ser fraccionados. Así, se trata de maximizar $\sum x_i . v_i$ con la restricción de que $\sum x_i . p_i <= P$ siendo x_i un valor mayor que 0 y menor o igual a 1.

Veamos un ejemplo: disponemos de una mochila con capacidad para transportar hasta 40 kilos. En ella queremos cargar los elementos que se describen en la siguiente tabla de forma tal que el valor monetario que se va a transportar sea máximo.

Elemento	Valor ($)	Peso (Kg)
A	50	36
B	48	30
C	30	20

Para que los objetos resulten comparables entre sí debemos analizar cuál es su valor monetario por cada kilogramo. Esta relación valor/peso se expone en la siguiente tabla.

Elemento	Valor ($)	Peso (Kg)	$/Kg
A	50	36	1.38
B	48	30	1.66
C	30	20	1.50

Ahora ordenaremos decrecientemente las filas de la tabla según la relación valor/peso.

Elemento	Valor ($)	Peso (Kg)	$/Kg
B	48	30	1.66
C	30	20	1.50
A	50	36	1.38

Luego, para aprovechar al máximo la capacidad de 40 kilos con una combinación de elementos óptima debemos cargar en la mochila 1 elemento B y aún tendremos disponibles 10 kg. No podemos meter un elemento C entero porque sobrepasaría la capacidad máxima pero si lo fraccionamos podremos meter ½ de elemento C. Con esto, quedará colmada la capacidad de la mochila. No queda lugar para el elemento A.

Así completamos la mochila de 40 kilos y maximizamos el valor monetario de la carga transportada valuada en 1*$48 + ½*$30 = $63.

Desarrollaremos la clase `Elemento` con los atributos `nombre`, `peso` y `valor` más un atributo extra que nos permitirá indicar qué cantidad de ese elemento debemos cargar en la mochila.

```
package libro.cap21.greddy;

public class Elemento
{
   private String nombre;
   private int peso;
   private int valor;
   private double cantidadACargar;
```

```
   public Elemento(String nombre, int peso, int valor)
   {
      this.nombre = nombre;
      this.peso = peso;
      this.valor = valor;
   }

   // :
   // setters y getters
   // toString
   // :
}
```

También necesitaremos una implementación de `Comparator` para ordenar el conjunto de elementos decrecientemente según la relación valor/peso.

```
package libro.cap21.greddy;
import java.util.Comparator;

public class RelacionValorPeso implements Comparator<Elemento>
{
   public int compare(Elemento e1, Elemento e2)
   {
      double d1 = (double)e1.getValor()/e1.getPeso();
      double d2 = (double)e2.getValor()/e2.getPeso();

      return -1*(d1<d2?-1:d1>d2?1:0);
   }
}
```

Ahora sí, veamos el programa que implementa el algoritmo de la mochila.

```
package libro.cap21.greddy;

import java.util.Arrays;

public class Mochila
{
   public static void procesarCarga(int capacidad, Elemento[] elementos)
   {
      // ordenamos el array descendentemente segun la relacion valor/peso
      Arrays.sort(elementos, new RelacionValorPeso());

      int suma=0;
      for(int i=0; i<elementos.length;i++)
      {
         if( suma<=capacidad )
         {
            // peso de este elemento
            int peso = elementos[i].getPeso();

            // capacidad de carga disponible
            int disponible = capacidad-suma;
```

```
            // cantidad que se va a agregar de este elemento
            double agregar = Math.min(1, (double)disponible/peso);
            elementos[i].setCantidadACargar(agregar);

            // sumamos el peso recientemente agregado, disminuye la capacidad disponible
            suma+= agregar*peso;
         }
      }
   }

   public static void main(String[] args)
   {
      int capacidad = 40; // hardcodeamos los datos...
      Elemento elementos[] = { new Elemento("A",36,50)
                              ,new Elemento("B",30,48)
                              ,new Elemento("C",20,30) };

      // procesamos los elementos
      procesarCarga(capacidad, elementos);

      for(Elemento e:elementos)
      {
         System.out.println(e);
      }
   }
}
```

La salida de este programa será:

```
Elemento [nombre=B, peso=30, valor=48, cantidadACargar=1.0]
Elemento [nombre=C, peso=20, valor=30, cantidadACargar=0.5]
Elemento [nombre=A, peso=36, valor=50, cantidadACargar=0.0]
```

21.4 Programación dinámica

Como introducción a este tema será conveniente recordar la implementación recursiva de la función de Fibonacci que analizamos en el Capítulo 17.

```
public static double fib(int n)
{
   return n<3?:1:fib(n-1)+fib(n-2);
}
```

En ese capítulo vimos también que una computadora estándar y contemporánea es incapaz de resolver esta función cuando el valor del parámetro `n` supera a 50.

La incapacidad se origina en la doble invocación recursiva que requiere que, para resolver `fib(8)`, tengamos que resolver primero `fib(7)` y `fib(6)`. Y para resolver `fib(7)` tengamos que resolver `fib(6)` y `fib(5)`; del mismo modo, para resolver `fib(6)` tenemos que resolver `fib(5)` y `fib(4)` y así sucesivamente. Nos vemos en la situación de resolver varias veces el mismo cálculo.

La implementación recursiva de la función de Fibonacci tiene una complejidad exponencial del orden de 2^n, exageradamente ineficiente.

Richard Ernest Bellman (1920–1984). Fue un matemático aplicado, su mayor contribución fue la programación dinámica. La IEEE le otorgó la medalla de honor, en 1979, por su contribución a la teoría de los sistemas de control y de los procesos de decisión, en especial por su aporte con la programación dinámica y por la ecuación de Bellman.

También vimos, dentro de ese mismo capítulo, una versión mejorada de la implementación recursiva en la que utilizamos una *hashtable* para almacenar los términos de la serie a medida que los calculábamos. Luego, en cada llamada a la función, antes de hacer una invocación recursiva, verificamos si la tabla contenía el resultado que estábamos buscando. Con esto, evitamos hacer llamadas recursivas innecesarias.

Esta modificación hizo que la función se comportara de manera óptima llevándola de una complejidad exponencial a una complejidad lineal. Claro que, como contrapartida, la nueva versión incrementó el uso de la memoria, es decir, aumentó la complejidad espacial.

Una vez refrescado este panorama, podemos pasar a definir el concepto de programación dinámica que se basa en el llamado "principio de optimalidad" enunciado por Bellman en 1957, que dice:

"En una secuencia de decisiones óptima toda subsecuencia debe ser óptima también"

Los algoritmos de programación dinámica dividen el problema principal en varios subproblemas más pequeños. La diferencia entre "programación dinámica" y "divide y conquista" está dada en que aquí los subproblemas no son independientes entre sí.

En la programación dinámica el resultado de cada subproblema que se resuelve debe ser almacenado en una tabla para evitar tener que volverlo a calcular.

En resumen, la programación dinámica implica:

- Menor cantidad de operaciones.
- Mayor uso de memoria.

Problema de los billetes según la programación dinámica

21.4.1 Problema de los billetes según la programación dinámica

El problema de calcular la cantidad de billetes que se necesita para abonar un importe determinado, tal como lo hemos resuelto y dependiendo de los datos de entrada, puede llegar a arrojar soluciones incorrectas o no optimizadas.

Recordemos que el objetivo del problema era, en primer lugar, minimizar la cantidad de billetes que se debía entregar y, dado que esta cantidad será mínima entonces priorizar la entrega de billetes de mayor denominación.

El algoritmo llegará a una conclusión errónea si pretendemos abonar un importe de $80 utilizando billetes de las siguientes denominaciones: $10, $40 y $60. En este caso nos recomendará utilizar 1 billete de $60 y 2 billetes de $10: tres billetes en total cuando la solución óptima sería utilizar solo 2 billetes de $40.

La solución, utilizando la técnica de programación dinámica, consiste en completar una tabla de *n* filas y *v*+1 columnas donde *n* es la cantidad de denominaciones diferentes y *v* es el monto que queremos abonar.

Para simplificar el análisis, en lugar de pretender abonar un monto de $80 con billetes de $10, $40 y $60, pensaremos en abonar un monto de $8 con billetes de $1, $4 y $6.

Llamaremos *d* al conjunto de las diferentes denominaciones. Este conjunto tendrá *n* elementos que, según los datos de nuestro ejemplo, serán: $d = \{1, 4, 6\}$

Consideraremos dos índices:

- *i* representará la posición de cada una de las diferentes denominaciones; por lo tanto, será $0 <= i < n$.
- *j* representará las columnas; por esto será $0 <= j <= v$.

Llamaremos *m* a la tabla o matriz descrita más arriba y la utilizaremos para asignar en $m[i][j]$ la menor cantidad de billetes necesarios para pagar un monto de $\$j$ con billetes de $\$d[i]$ o de menor denominación.

Por ejemplo, si *j* fuese 6 e *i* fuese 1 entonces en *m*[*i*][*j*] tendremos que asignar la mínima cantidad de billetes que se requiere utilizar para abonar $6 utilizando billetes de $4 y/o de $1. En este caso las opciones son: 6 billetes de $1 o 1 billete de $4 y 2 billetes de $1. El mínimo entre 6 y 3 es 3; por lo tanto, este será el valor que debemos asignar.

A continuación, haremos un análisis paso a paso para ilustrar el criterio que utilizaremos para completar cada una de las celdas de la matriz *m* que, recordemos, tendrá *n* filas y *v*+1 columnas.

En nuestro ejemplo la matriz tendrá 3 filas, una por cada denominación, y 9 columnas. La primera representará la necesidad de abonar un valor, absurdo, de $0. La segunda representará la necesidad de abonar $1 y la última representará la necesidad de abonar un valor de $8.

	0	1	2	3	4	5	6	7	8
	$	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
0	**1**									
1	**4**									
2	**6**									

Fig. 21.1 Tabla de programación dinámica vacía.

21.4.1.1 Análisis de la primera fila, que representa al billete de $1

La fila 0 (la primera fila) representa al billete de $1, entonces debemos completar cada celda *m*[0,*j*] con la cantidad mínima necesaria de billetes de $1 que se requiere para abonar un valor de $*j*. En este caso particular, dado que *i* representa la unidad, el valor de cada celda coincidirá con el importe $*j* representado por la columna salvo, obviamente, la columna 0 que representa un valor de $0 y que resulta imposible de abonar con billetes de cualquier denominación.

Veamos:

	0	1	2	3	4	5	6	7	8
	$	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
0	**1**	0	1	2	3	4	5	6	7	8
1	**4**									
2	**6**									

Fig. 21.2 Tabla de programación dinámica.

Es decir, para pagar $1 con billetes de $1 necesitamos 1 billete. Para pagar $2 con billetes de $1 necesitamos usar 2 billetes, etcétera. Obviamente, no utilizaremos ningún billete de $1 para pagar $0. La columna 0 siempre tendrá valores cero.

21.4.1.2 Análisis de la segunda fila, que representa al billete de $4

Ahora completaremos la segunda fila. En este caso el criterio será el siguiente: si el valor que se va a pagar (representado por cada columna) es menor que la denominación del billete, completaremos la celda con el valor de la celda ubicada en la fila anterior. Esto se debe a que no podemos pagar $*j* con billetes de la denominación representada por la fila *i*. En cambio, si la denominación del billete (en este caso $4) es menor o igual al valor representado por la columna *j*, entonces completaremos la celda con el mínimo entre las siguientes opciones:

$$j/d[i] + m[i\text{-}1][j\%d[i]]$$

y

$$m[i\text{-}1,j]$$

Esto puede resultar un poco complicado de entender. Veremos la matriz con la segunda fila completa y luego explicaremos en detalle el porqué de cada valor.

		0	1	2	3	4	5	6	7	8
	$	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
0	**1**	0	1	2	3	4	5	6	7	8
1	**4**	0	1	2	3	1	2	3	4	2
2	**6**									

Fig. 21.3 Tabla de programación dinámica.

Recordemos que la segunda fila representa al billete de $4. Por esto no podremos utilizar este billete para pagar montos de $1, $2 o $3. Es decir, la mejor (y única) opción para abonar estos montos está ubicada en la fila anterior que representa la denominación de menor valor más próxima.

Para abonar $4 con billetes de $4 (o menores) resulta obvio que solo necesitaremos utilizar 1 billete de $4.

Para abonar $5 con billetes de $4 (o menores) tendremos que usar 1 billete de $4 y abonar el residuo de $1 con 1 billete de $1: en total 2 billetes.

Para abonar $6 utilizaremos 1 billete de $4 y 2 billetes de $1: tres billetes en total.

Para abonar $7 utilizaremos 1 billete de $4 y 3 billetes de $1: cuatro billetes en total.

Para abonar $8 utilizaremos 2 billetes de $4 ya que al dividir 8 por 4 no obtendremos ningún valor residual.

21.4.1.3 Análisis de la tercera fila, que representa al billete de $6

Siguiendo el criterio mencionado más arriba, completaremos la tercera fila y luego analizaremos cada caso por separado.

		0	1	2	3	4	5	6	7	8
	$	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
0	**1**	0	1	2	3	4	5	6	7	8
1	**4**	0	1	2	3	1	2	3	4	2
2	**6**	0	1	2	3	1	2	1	2	2

Fig. 21.4 Tabla de programación dinámica.

Como ya sabemos, no hay manera de utilizar billetes de $6 para pagar montos inferiores a este valor, así que la mejor opción siempre será la que determinamos cuando analizamos la fila que representa a la denominación anterior que, en este caso, es la de $4.

Ahora, para abonar $6 con billetes de $6 (o inferiores) es evidente que alcanzará con un único billete (ver *m*[2][6]).

Para abonar $7 con billetes de $6 (o inferiores) necesitaremos 1 billete de $6 y 1 billete de $1 (ver *m*[2][7]).

Para abonar $8 con billetes de $6 (o inferiores), aplicamos la fórmula detallada más arriba:

$8/$6 = 1 más un valor residual de $2. La cantidad de billetes óptima para abonar este residuo está ubicada en *m*[*i*-1],2]. Esto suma un total de 3 billetes. Sin embargo, en la celda ubicada en la fila anterior vemos que si utilizamos billetes de $4 (o inferiores) solo necesitaremos 2 billetes. Entonces escogeremos el mínimo valor entre 3 y 2, que es 2.

21.4.1.4 Recorrer la tabla de cantidades mínimas para indicar qué cantidad de billetes de cada denominación debemos utilizar

La tabla o matriz *m* que analizamos y completamos más arriba solo indica cuál es la cantidad óptima de billetes para abonar el importe especificado pero no lleva el rastro sobre cómo se compone esa cantidad.

Es decir, el valor 2 que asignamos en *m*[2][8] indica que esa es la menor cantidad de billetes que podemos utilizar para pagar $8 con billetes de hasta $6 pero no indica el tipo o la denominación de cada uno de esos dos billetes.

Por esto, debemos complementar la estructura de datos con una matriz auxiliar booleana que llamaremos *u* (de "utiliza") que tendrá las mismas dimensiones que *m*.

Recordemos que en *m*[*i*][*j*] asignamos el mínimo entre las siguientes dos opciones:

$$op1 = j/d[i] + m[i\text{-}1][j\%d[i]]$$

y

$$op2 = m[i\text{-}1,j]$$

La opción 2 indica que no utilizaremos billetes de la denominación representada por *i*. En este caso completaremos *u*[*i*][*j*] con *false*. De lo contrario asignaremos *true*.

Así, la matriz *u* que se generará a la par de la matriz *m* según nuestro ejemplo se verá de la siguiente manera:

	0	1	2	3	4	5	6	7	8
	$	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
0	**1**	0	1	2	3	4	5	6	7	8
1	**4**	0	1	2	3	1	2	3	4	2
2	**6**	0	1	2	3	1	2	1	2	2

Fig. 21.5 Matriz de cantidades mínimas.

	0	1	2	3	4	5	6	7	8
	$	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
0	**1**	*true*	*true*	*true*	*true*	*true*	*true*	*true*	*true*	*true*
1	**4**	*false*	*false*	*false*	*false*	*true*	*true*	*true*	*true*	*true*
2	**6**	*false*	*false*	*false*	*false*	*false*	*false*	*true*	*true*	*false*

Fig. 21.6 Matriz de utilización.

El algoritmo para recuperar la denominación de cada uno de los billetes que componen la solución óptima (mínima) es el siguiente:

Sea *i*=*n*-1 (la posición de la última fila de la matriz) y *j*=*v* la posición de la última columna, entonces evaluamos *u*[*i*][*j*]: si es *true* será porque utilizamos un billete de la denominación *d*[*i*]. Lo guardamos en un *array* y luego restamos:

$$quedan = v - d[i]$$

Luego decrementamos *i* y asignamos *j* = *quedan*.

En cambio, si *u*[*i*][*j*] = *false* será porque no utilizamos ningún billete de la denominación *d*[*i*]. Luego, simplemente decrementamos el valor de *i*.

Todo esto mientras el valor de *quedan* sea mayor que cero.

Veamos el código implementado en la clase `PagaImporte` que recibe como parámetros en el constructor el *array* de denominaciones y el importe que se va a pagar.

```
package libro.cap21.dinamica;

import java.util.ArrayList;

public class PagaImporte
{
   // matriz de minimas cantidades
   private int matriz[][];

   // matriz que indica si se utilizan o no billetes de la denominacion
   private boolean utiliza[][];

   // array de denominaciones
   private int[] denominaciones;

   // importe que se va a abonar
   private int v;

   public PagaImporte(int[] denom, int v)
   {
      this.denominaciones = denom;
      this.v = v;

      // arma las dos matrices
      armarMatriz();
   }

   // sigue mas abajo
   // :
```

Ahora veamos el método `armarMatriz` que será el encargado de crear y completar las dos matrices.

```
   // :
   // viene de mas arriba

   private void armarMatriz()
   {
      // determinamos las dimensiones
      int filas = denominaciones.length;
      int columnas = v + 1;

      // instanciamos las matrices
      matriz = new int[filas][columnas];
      utiliza = new boolean[filas][columnas];

      // completamos la columna de $0
      for(int i = 0; i<filas; i++)
      {
         matriz[i][0]=0;
         utiliza[i][0]=false;
      }
```

```
      // completamos la primera fila
      for(int j = 0; j<columnas; j++)
      {
         matriz[0][j]=j;
         utiliza[0][j]=true;
      }

      // completamos las otras celdas
      for(int i=1; i<filas; i++)
      {
         for(int j=1; j<columnas; j++)
         {
            int op1 = j/denominaciones[i]+matriz[i-1][j%denominaciones[i]];
            int op2 = matriz[i-1][j];

            matriz[i][j] = Math.min(op1, op2);
            utiliza[i][j] = Math.min(op1, op2)==op1;
         }
      }
   }

   // sigue mas abajo
   // :
```

Veamos ahora el método `obtenerResultados` que retorna un `ArrayList<CantBillete>` donde `CantBillete` es una clase básicamente con dos atributos: `cantidad` y `denominación`. Veámosla:

```
package libro.cap21.dinamica;

public class CantBillete
{
   private int cantidad;
   private int denominacion;

   public CantBillete(int c, int d)
   {
      setCantidad(c);
      setDenominacion(d);
   }
   public String toString()
   {
      String x = cantidad>1?"billetes":"billete";
      return cantidad+" "+x+" de $"+denominacion;
   }

   // :
   // setters y getters
   // :
}
```

Y, ahora sí, el método `obtenerResultados`:

```
   // :
   // viene de mas arriba

   public ArrayList<CantBillete> obtenerResultados()
   {
      ArrayList<CantBillete> a = new ArrayList<CantBillete>();

      int i = denominaciones.length-1;
      int j = matriz[0].length-1;

      while(i>=0)
      {
         if( utiliza[i][j] )
         {
            int d = denominaciones[i];
            int c = 1;
            int queda = j-d;
            a.add(new CantBillete(c, d));

            if( queda==0 )
            {
               break;
            }

            j = queda;
         }
         else
         {
            i--;
         }

      }

      return a;
   }

   // sigue mas abajo
   // :
```

Por último, veamos el método `main` que prueba el algoritmo.

```
   // :
   // viene de mas arriba

   public static void main(String[] args)
   {
      int denom[] = { 1, 4, 6 };
      int valor = 8;

      PagaImporte t = new PagaImporte(denom, valor);
      ArrayList<CantBollete> res = t.obtenerResultados();

      System.out.println(res);
   }
}
```

21.5 Resumen

En este capítulo analizamos las estrategias de solución de algunos algoritmos para identificar patrones comunes en función de que cumplan o no con determinadas características. Vimos también que los métodos de programación dinámica solucionan problemas que los métodos *greddy*, en ocasiones, no pueden resolver.

En el próximo capítulo estudiaremos grafos y algunos algoritmos que nos permitirán realizar operaciones sobre grafos analizando sus versiones *greddy* y de programación dinámica.

21.6 Contenido de la página Web de apoyo

El material marcado con asterisco (*) solo está disponible para docentes.

21.6.1 Mapa conceptual

21.6.2 Autoevaluaciones

21.6.3 Videotutorial

21.6.3.1 Problema de los billetes por programación dinámica

21.6.4 Presentaciones*