Ficha 14

Programación Dinámica [Introducción]

1.) Introducción general.

Sabemos que los investigadores y programadores han caracterizado algunas técnicas generales que a menudo llevan a algoritmos eficientes para la resolución de ciertos problemas. Hemos realizado en fichas anteriores un breve análisis de algunas de las técnicas más importantes y conocidas como son la *recursividad*, el *backtraking*, la técnica *divide y vencerás*, los *algoritmos ávidos o devoradores*. En esta ficha en particular, haremos una introducción a los principios básicos de la *programación dinámica*.

Sin embargo, hay que recordar que existen algunos problemas para los cuales ni éstas ni otras técnicas conocidas producirán soluciones eficientes. Cuando se encuentra algún problema de este tipo suele ser útil determinar si las entradas al problema tienen características especiales que se puedan explotar en la búsqueda de una solución, o si puede usarse alguna solución aproximada sencilla, en vez de la solución exacta y más difícil de calcular.

Un conocido problema que suele usarse para mostrar la aplicación de la *programación dinámica*, es el *problema de cambio en monedas*. Se trata de plantear un algoritmo que pueda calcular la mínima cantidad de monedas en la que puede cambiarse una cierta cantidad de centavos, conociendo los valores nominales de las monedas disponibles. Supongamos que disponemos de monedas de 1, 5, 10 y 25 centavos. Nos dan una cantidad *x* de centavos, y nos piden calcular la *mínima* cantidad de esas monedas en la que puede expresarse *x*.

Así, si *x = 25* entonces la mínima cantidad de monedas (de acuerdo a nuestro conjunto de monedas) es 1, ya que disponemos de una moneda de exactamente 25 centavos. Si *x* es 31, entonces la mínima cantidad de monedas es 3: una de 25, una de 5 y una más de 1 centavo.

Podemos dar diversas soluciones a este problema, basadas en distintas técnicas algorítmicas. Una forma muy intuitiva, consiste en elegir sistemáticamente la moneda de mayor designación, devolver tantas de ella como se pueda, y luego continuar así con las monedas de mayor valor que siguen. Así, si x = 63, elegimos dos veces la moneda de 25, una vez la de 10 y tres la de 1, lo cual da un total de 6 monedas. Si las monedas disponibles son las que hemos supuesto, se puede probar que esta forma de trabajar resolverá siempre el problema en forma correcta, y constituye un ejemplo más de la conocida estrategia de resolución de problemas conocida como *algoritmos ávidos* o *devoradores* (o *greedy algorithms*).

Sabemos que *algoritmo ávido* es aquel en el que en todo momento se toma la decisión que localmente parece la más apropiada u óptima para el caso, sin importar ni analizar si esa decisión traerá malas consecuencias en el futuro. También hemos visto que lo bueno de esta clase de estrategias es que suelen ser simples de plantear y de entender. Normalmente se ajustan en forma directa al modelo de solución mental que se tiene del problema, por lo que intuitivamente suele ser la estrategia preferida si puede aplicarse. Lo malo, es que no siempre esta técnica funciona: si en nuestro ejemplo dispusiéramos de una moneda de 21 centavos, podemos ver que una estrategia de *algoritmo ávido* seguiría informando que el

mínimo es de 6 monedas, cuando es obvio que ahora la solución óptima es 3. En todo caso, el problema con las reglas ávidas es que debemos demostrar que funcionan para toda combinación posible de entradas, y esa demostración no siempre es simple de hacer.

Si el número y designación de las monedas disponibles es *arbitrario*, nos preguntamos cómo plantear entonces un algoritmo que siempre funcione. Lo primero, será fijar alguna precondición que garantice que la solución existe. Es obvio que en nuestro caso, debemos exigir que siempre existan monedas de 1 centavo: cualquiera sea la cantidad a expresar, siempre se podrá plantear como suma de monedas de 1 centavo. A partir de aquí, podríamos aplicar *recursividad*, en base a un esquema intuitivamente simple:

- ✓ Si con una sola moneda podemos cubrir la cantidad *x* pedida, devolvemos esa moneda y la solución es 1 (sólo necesitamos una moneda).

- ✓ De otro modo, para cada posible valor de $j < x$ calculamos en forma independiente (con dos llamadas recursivas) la cantidad mínima de monedas a usar para cubrir cantidades iguales a j y $(x - j)$ y retornamos el valor de j para el cual la suma de ambos valores sea la mínima.

En el proyecto que acompaña a esta ficha, mostramos implementada esta solución por medio del siguiente método en la clase *Change*:

```
public int recursive_change(int values[], int amount)
{
    // la cantidad de valores nominales disponibles...
    int n = values.length;

    // empezamos suponiendo que el mínimo es igual a tantas monedas de
    // valor 1 como sea el valor de amount...
    int min = amount;

    // buscamos una moneda que sea ella misma el cambio exacto...
    for(int i = 0; i < n; i++)
    {
        // si es el caso, con una sóla cubrimos el valor de amount...
        if(values[i] == amount) { return 1; }
    }

    // no encontramos esa moneda... resolvemos recursivamente...
    for(int j = 1; j <= amount / 2; j++)
    {
        int candidate = recursive_change(values, j) + recursive_change(values, amount - j);
        if(candidate < min)
        {
            min = candidate;
        }
    }

    // ... y retornamos el mínimo valor que hayamos visto...
    return min;
}
```

Estrictamente hablando, este algoritmo es *muy ineficiente* ya que puede verse que ambas invocaciones recursivas eventualmente procesan los *mismos conjuntos de datos*. Eso también implica que si se ejecuta este sencillo algoritmo recursivo para un valor pequeño en la entrada, funcionará correctamente y su tiempo de ejecución será razonable. Pero si pide tanto como un valor inicial a cambiar de 63 centavos, el algoritmo demorará una *cantidad extraordinaria de tiempo*, debido al enorme tamaño del árbol de llamadas recursivas.

Si bien pueden plantearse algunas mejoras que hagan que este algoritmo recursivo mejore su rendimiento, el hecho es que mediante *recursividad* no podrán obtenerse mejoras significativas: simplemente, las soluciones recursivas para este problema son ineficientes debido a que el problema no es esencialmente recursivo: no es un solución natural plantear un algoritmo que se invoque a si mismo dos veces para dar la respuesta óptima, y tampoco necesita este problema que se use una pila para guardar el camino de retorno de la recursión.

Una solución mucho mejor consiste en usar una *tabla* (un arreglo) en la cual se almacenen los resultados ya calculados, y luego se use esa tabla para calcular nuevos casos. Al fin y al cabo, si queremos saber cuántas monedas hacen falta para cambiar 63 centavos, es posible que sirva saber cuántas monedas hacen falta para cambiar 30 o 60, y si esos cáclulos ya fueron hechos con anterioridad sería útil poder reusar sus resultados sin tener que volver a calcular. La idea: no calcular dos o más veces lo mismo, sino que cada vez que se tenga un nuevo caso, se guarde su resultado en una tabla e ir llenando esa tabla a medida que el proceso avanza sobre nuevos casos. Esta técnica o estrategia de solución basada en tablas de resultados previos, se conoce como *programación dinámica*.

En nuestro caso, dado el valor *amount* a cambiar en monedas, queremos reutilizar los cálculos que hayamos realizado para cualquier valor menor que amount durante el proceso, y para ello usaremos un arreglo *used* que tendrá exactamente *amount + 1* casilleros. De esta forma, el arreglo *used* tednrá efectivamente su última casilla numerada con el valor *amount*, y la idea es depositar en esa casilla el valor final: la cantidad mínima de monedas para cubrir el valor *amount*.

Luego debemos ir llenando esa tabla con los resultados parciales de los subproblemas o valores *anteriores* al cálculo para *amount*, para lo cual tendremos que plantear el mecanismo de llenado para cada casilla: esto se conoce como el planteo de la *recurrencia* a emplear para el llenado de la tabla. En todo proceso de programación dinámica se parte de uno o más casos obvios, llamados *casos base* de la recurrencia: en nuestro caso, el caso base es directo: si tenemos que cubrir un valor de *amount = 0*, entonces necesitamos *0* monedas y sabemos entonces que la casilla *used[0]* valdrá *0*.

A partir de allí, en cada casilla *used[v]* (con *1 <= v <= amount*) deberemos almacenar la mínima cantidad de monedas posible para llegar al valor v, usando las monedas disponibles. El cálculo podría parecer complicado, pero un ejemplo mostrará en forma clara la ecuación de recurrencia a emplear:

Suponga que los *n* valores nominales de las monedas disponibles están almacenados en un arreglo *values*. Sea *int values[] = {1, 5, 10, 25, 50}* con *n = 5* y sea *v = 13*: esto es, queremos llenar la casilla *used[13]* de la tabla, suponiendo que ya hemos llenado todas las anteriores. Está claro entonces que los posibles valores para *used[13]* saldrán del *mínimo* entre todas las siguientes combinaciones:

- ✓ 13 monedas de 1 centavo.
- ✓ 1 moneda de 1 centavo + el mínimo para cubrir 12 centavos = 1 + used[12]
- ✓ 1 moneda de 5 centavos + el mínimo para cubrir 8 centavos = 1 + used[8]
- ✓ 1 moneda de 10 centavos + el mínimo para cubrir 3 centavos = 1 + used[3]

- ✓ Y no podemos usar monedas de 25 o 50 centavos, ya que ambas son mayores a v = 13.

De allí surge con sencillez la expresión completa de la recurrencia a usar:

used[v] = minimo(v, mininimo(1 + used[ui]))

donde ui = v – values[j] con j = 0, 1, 2, … n - 1

Finalmente, el planteo del algoritmo completo sería entonces:

Entradas:

- Un arreglo *values*, con los *n* valores nominales de las monedas disponibles (de forma que el valor nominal 1 exista).
- Un valor entero *amount* que será el valor a cubrir con las monedas disponibles.

Salidas:

- ✓ Un número entero que indique la mínima cantidad de monedas a devolver para cubrir el valor *amount*.

Proceso (algoritmo básico – recurrencia a emplear):

int change(int values[], int amount):

- Sea *n* = *values*.length;
- Sea la tabla *used[]* con *amount* + 1 casilleros: used = new int[amount + 1]
- Sea *used[0]* = 0
- Para todo v en [1 . . amount]:
  - used[v] = min(v, min(1 + used[ui])) donde ui = v – values[j] (con j = 0, 1, 2, …, n-1)
- Retorne *used[amount]*

Un replanteo del problema del cambio de monedas, ahora usando programación dinámica, se muestra en el mismo proyecto que acompaña a esta ficha, pero ahora en el siguiente método de la misma clase *Change*:

```
public int change(int values[], int amount)
{
        // la cantidad de valores nominales disponibles...
        int n = values.length;

        // la tabla para los resultados de los subproblemas anteriores...
        int used[] = new int[amount + 1];

        used[0] = 0;
        for(int v = 1; v <= amount; v++ )
        {
             int min = v;
```

```
            for(int j = 0; j < n; j++)
            {
                // si values[j] se pasa, seguir el ciclo...
                if(values[j] > v) { continue; }

                int ui = v - values[j];
                if(used[ui] + 1 < min)
                {
                    min = used[ui] + 1;
                }
            }

            used[v] = min;
        }

        return used[amount];
}
```

El método aplica en forma directa el algoritmo basdado en la recurrencia explicada más arriba. Se deja para el alumno el estudio de los detalles.