Práctica 4

Eliminación de la recursividad redundante

Algoritmos Avanzados

Víctor Fernández Fernández
Mikayel Mardanyan Petrosyan

11 de Noviembre de 2019

0 Problema

Dado el siguiente método:

```
public static int f (int x, int y) {
    if (y==0)
            return x;
    else
            return f(x,y-1) + f(x+1,y-1);
}
```

El objetivo de la práctica es eliminar la redundancia existente en este algoritmo recursivo múltiple y obtener dos algoritmos optimizados: uno mediante *memoización*[1] y otro mediante *tabulación*.

1 Análisis de la redundancia y diseño de la tabla

El primer paso para la eliminación de la recursividad redundante es el análisis de la redundancia, seguido por el diseño de una tabla para la eliminación de dicha redundancia. Para ello hemos usado el sistema SRec.

[1] No confundir con *memorización*

1.1 Análisis de la redundancia

Vamos a representar gráficamente la recursividad de la función f de dos maneras distintas: mediante un árbol de recursión y mediante el grafo de dependencia correspondiente al árbol mencionado.

El árbol de recursión muestra las llamadas recursivas realizadas por la función. Cada nodo corresponde con una llamada y muestra los datos de entrada (azul) y de salida (rojo), obviando el identificador de la función porque solo hay una.

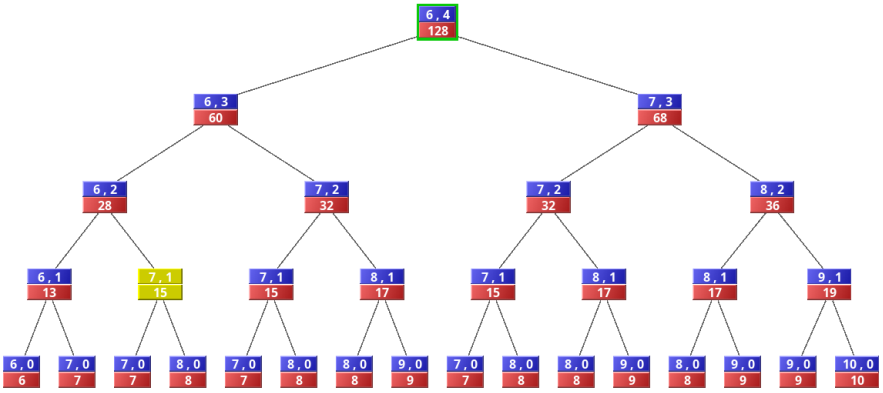

Figure 1: Árbol de recursión

Vemos que en el árbol aparecen varios nodos (llamadas recursivas) que se repiten; por ejemplo, el nodo que aparece resaltado, que se repite hasta dos veces. Esto nos indica que hay redundancia y que las técnicas de eliminación de recursividad ofrecerán resultados notables. Como el método inicial realiza dos llamadas recursivas, cada nodo tendrá dos arcos a dos nuevos nodos (hasta llegar a los casos base).

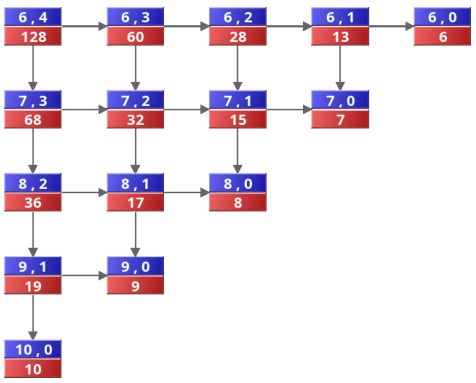

Figure 2: Grafo de dependencia

En este grafo vemos más fácilmente las dependencias de cada llamada recursiva. La llamada inicial (6,4) hará dos llamadas recursivas y estas se propagarán hasta llegar a los nodos base.

1.2 Tabla

Las técnicas de optimización estudiadas consisten en la eliminación de las llamadas recursivas redundantes mediante el almacenamiento de los resultados que producen estas. Para ello, vamos a necesitar una tabla que almacene las soluciones de los nodos (que corresponden a llamadas) que vemos en el grafo de dependencia.

Podemos observar que el número de filas y columnas depende únicamente del parámetro y: se harán llamadas recursivas hasta que el parámetro y sea 0. En las columnas la x permanece constante y en las filas, se incrementa, pero en ambas el parámetro y decrece hasta 0, por lo que la tabla necesitará tener $y + 1$ filas e $y + 1$ columnas.

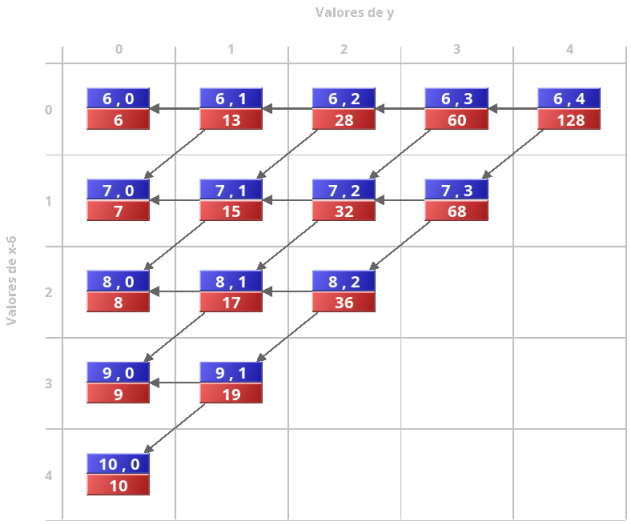

Figure 3: Tabla

Organizando los nodos por orden ascendiente de sus parámetros obtenemos lo siguiente. Vemos que los valores de y decrecen hasta 0, por lo que se pueden asociar fácilmente con los índices de la tabla. Sin embargo, los valores de x van creciendo. Por tanto, si asociamos los valores de este parámetro con los índices de la tabla, tendremos un espacio desaprovechado. Para solucionar este problema vamos a restar al valor del parámetro x el valor de dicho parámetro en la llamada inicial. Así, no habrá espacio desaprovechado.

La implementación en Java de la tabla t mostrada sería:

```
int[][] t = new int[y+1][y+1];
```

donde $f(m, n)$ se almacena en $t[m - x][n]$, siendo x e y los valores de los parámetros en la llamada inicial, y m y n los valores que toman dichos parámetros.

2 Memoización

Esta técnica consiste en almacenar en una tabla los resultados de las llamadas recursivas ya realizadas. La nueva función equivalente a la inicial va a crear esta tabla, llenarla con un valor que marque que está vacía y después llamar a una función auxiliar. Esta función auxiliar comprobará si la solución ya ha sido calculada y, si no es así, hará las llamadas recursivas correspondientes.

2.1 Código

```
public static int f1(int x, int y) {
     // Tabla t
     int[][] t = new int[y + 1][y + 1];
     // Llenamos la tabla con un valor inicial
     for (int i = 0; i <= y; i++)
         for (int j = 0; j <= y; j++)
             t[i][j] = -1;
     // Llamamos al algoritmo de memoizacion
     fMemo(x, x, y, t);
     // Devolvemos la solucion de la llamada inicial
     return t[0][y];
 }

 private static void fMemo(int x, int m, int n, int[][]
t) {
     // Si la solucion de la llamada no se ha calculado
     if (t[m - x][n] == -1)
         // Si es un caso base
         if (n == 0)
             t[m - x][0] = m;
             // Si es un caso recursivo
         else {
             // Calculamos los dos operandos
             fMemo(x, m, n - 1, t);
```

```
                fMemo(x, m + 1, n - 1, t);
                // Operamos y almacenamos la solucion
                t[m - x][n] = t[m - x][n - 1] + t[m - x +
1][n - 1];
            }
    }
```

La tabla t será la que almacene los resultados de las llamadas recursivas realizadas. Las celdas van a tener un valor inicial de -1 para marcar estas como no calculadas. Siempre vamos a llamar a la función de memoización, pero solo ejecutaremos su cuerpo cuando la tabla no tenga un valor para la llamada actual. En caso de que ya hayamos calculado esa llamada, saldremos del método y volveremos al primer método, y buscaremos dicha solución en la tabla. En el caso de que no la tengamos, llenaremos la celda con el valor del primer parámetro si es un caso base, o calcularemos los operandos y después los sumaremos y almacenaremos en la tabla.

2.2 Árbol de recursión

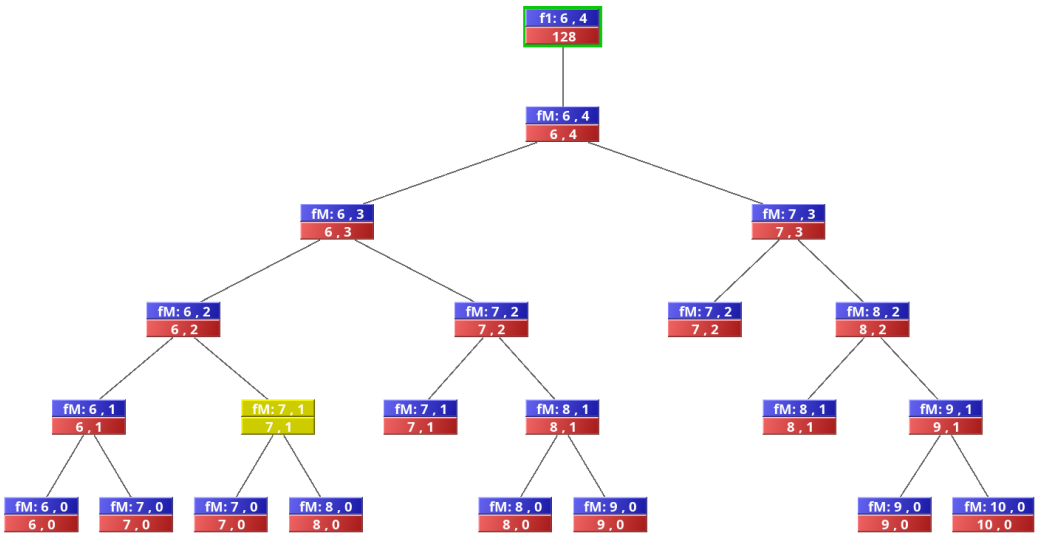

Figure 4: Vista global del árbol de recursión

En la vista global vemos que el árbol tiene menos nodos que el del método original. El nodo que antes se repetía dos veces ahora se repite una, debido a que ya se tiene almacenada la solución del nodo padre que llama a este. La rama izquierda y la rama derecha que sale de cada nodo de mencionada rama izquierda son las llamadas que se calculan por primera vez, y las diagonales que acompañan a dichas ramas derechas son accesos a la tabla.

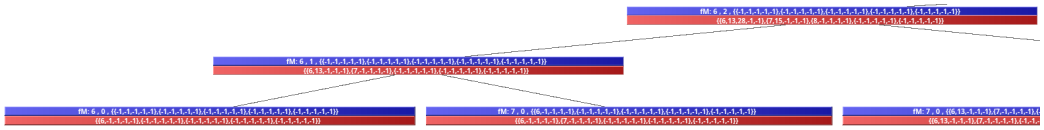

Figure 5: Detalle de llamadas del árbol de recursión

En el detalle de una sección del árbol vemos cómo los valores de entrada de la tabla (en azul) se modifican (en rojo). En el primer nodo hoja la tabla llega vacía y se introduce el valor de la primera variable. En el segundo ocurre lo mismo y se devuelve la solución al nodo padre, que suma estos dos valores y almacena la suma en otra celda para que sea usada por su nodo padre más adelante.

2.3 Análisis de complejidad

$$
\begin{aligned}
T(n) &= (n+2) + \sum_{i=1}^{n} (i) - 1 + \sum_{i=1}^{n-1} (i) = \\
&= (n+2) + \left(\frac{n^2+n}{2} - 1 \right) + \left(\frac{n^2+n}{2} - n \right) = \\
&= (2n+4) + \left(n^2 + n - 2 \right) + \left(n^2 + n - 2n \right) = \\
&= 2n^2 + 2n + 2 = \\
&= \Theta\left(n^2 \right)
\end{aligned}
$$

$$
\begin{aligned}
M(n) &= (n+1)(n+1) = \\
&= n^2 + 2n + 1 = \\
&= \Theta\left(n^2 \right)
\end{aligned}
$$

donde n es el segundo parámetro de la llamada inicial , $T()$ es la función de tiempo y $M()$ es la función de espacio en memoria.

En el árbol de recursión encontramos $n+2$ nodos, correspondientes con la rama izquierda y los dos nodos hijo; una progresión aritmética desde 2 hasta n (no desde 1 hasta n, por eso restamos 1), correspondiente con las ramas derechas que salen de la izquierda; y una progresión aritmética desde 1 hasta $n-1$, correspondiente con las diagonales mencionadas anteriormente. En cuanto a la memoria, almacenamos los resultados en un array bidimensional.

3 Tabulación

Esta técnica no es recursiva como la memoización, sigue un esquema iterativo con un orden lineal de cómputo. Consiste en llenar la tabla primero con unos valores iniciales o valores base y después seguir un orden lineal para rellenar el resto de la tabla usando los valores calculados anteriormente.

Además, esta técnica admite una minimización de la memoria. Los valores pueden almacenarse solamente durante el tiempo que se necesiten y luego usar este espacio para contener otros valores, ahorrando espacio. La cuestión entonces es encontrar el número mínimo de variables suficientes para contener los resultados necesarios. Estas variables se consiguen definiendo un corte en el grafo de dependencia. La disminución en el espacio de memoria requiere definir una nueva estructura de datos para contener las variables.

3.1 Código

```
public static int f3(int x, int y) {
     // Vector t
     int[] t = new int[y + 1];
     // Llenamos los casos base
     for (int i = 0; i <= y; i++)
         t[i] = x + i;
     // Calculamos las llamadas recursivas
     for (int j = 1; j <= y; j++)
         for (int i = 0; i <= y; i++)
             if (i + j <= y)
                 t[i] = t[i] + t[i + 1];
     // Devolvemos el resultado de la llamada inicial
     return t[0];
 }
```

En este caso no necesitamos la tabla entera para calcular el problema, nos basta con un vector, ya que empleamos un corte en el grafo de dependencia (figura 3) para utilizar solo las columnas de esta tabla. Así, llenamos el vector t con los casos base, que corresponden con la primera columna de la tabla. En el bucle for externo iteramos tantas veces como columnas hay en la tabla y en el interno recorremos la columna sobrescribiendo cada nodo con la suma del resultado de las llamadas.

3.2 Análisis de complejidad

$$\begin{aligned} T(n) &= 1 + (n+1) + n\,(n+1) = \\ &= 1 + n + 1 + n^2 + n = \\ &= n^2 + 2n + 2 = \\ &= \Theta\left(n^2\right) \end{aligned}$$

$$\begin{aligned} M(n) &= (n+1) = \\ &= \Theta\,(n) \end{aligned}$$

donde n es el segundo parámetro de la llamada inicial , $T()$ es la función de tiempo y $M()$ es la función de espacio en memoria.

Para el tiempo de ejecución nos fijamos en el primer bucle for simple y el segundo (anidado) con otro bucle for dentro. En cuanto a la memoria, los resultados se almacenan en un vector de $n+1$ posiciones.

4 Conclusiones

En esta práctica hemos aprendido a mejorar algoritmos recursivos por medio de la eliminación de la redundancia. Los algoritmos recursivos son muy útiles para implementar algoritmos de una manera muy similar a su especificación, y esto puede ayudar a entenderlos mejor, pero es necesario modificarlos para obtener mejores resultados en cuanto a eficiencia. Hemos visto cómo hemos pasado de una complejidad en tiempo $O(2^n)$ en el algoritmo inicial a una $O(n^2)$ en los algoritmos optimizados y cómo hemos conseguido una complejidad en memoria $O(n)$ muy eficiente con la tabulación.

La herramienta SRec nos ha ayudado bastante a visualizar los problemas recursivos, su redundancia y la dependencia de las soluciones.

La parte más interesante ha sido la memoización, pues con un esquema muy simple y parecido al del algoritmo inicial obtenemos mayor eficiencia. La parte más complicada ha sido la tabulación y la visualización de los cortes, aunque también ha sido muy interesante.