Programación Dinámica (I): Fundamentos

aprende.olimpiada-informatica.org/algoritmia-dinamica-1

Analicemos esta función recursiva (véase: manual de recursividad) para calcular los números de Fibonacci:

```
int fib(int n) {
      if (n < 2) {
          return 1;
      }
      return fib(n-1) + fib(n-2);
}
```

```
def fib(n):
        if n < 2: return 1
        return fib(n-1) + fib(n-2)
```

Nos preguntamos cómo de eficiente es esta función. Para ello, vamos a calcular el número total de sumas que se hace en una llamada a `fib(n)` en términos de `n` . Para `n=0, 1` no se hace ninguna suma. Para `n=2` se hace una suma; para `n=3` se hacen tres sumas; para `n=4` , se hacen 5 sumas (las 3 que se hacen con la llamada a `fib(3)` , la que se hace con la llamada a `fib(2)` y la que se hace sumando los dos valores que retornan estas funciones). Vemos que este número de sumas satisface una relación de recurrencia similar a la de los números de Fibonacci: \(S(n) = S(n-1) + S(n-2) + 1\). Eso significa que el número de sumas crece exponencialmente, y por tanto nuestro programa tardará bastante en calcular números de Fibonacci grandes. ¿Cómo podemos hacer más eficiente el cálculo de números de Fibonacci? Una ineficiencia que podemos observar es que calculamos muchas veces los mismos números. Por ejemplo, la llamada a `fib(n)` llama a `fib(n-1)` y a `fib(n-2)` , y a su vez `fib(n-1)` llama a `fib(n-2)` también, de forma que el cálculo del (n-2)-ésimo número de Fibonacci se realiza dos veces.

Una forma natural de tratar de solucionar este inconveniente es guardar el valor de los números de fibonacci que ya hemos calculado:

```
vector<int> f; //todos los valores inicializados a -1, excepto f[0] = f[1] = 1
int fib(int n) {
         if (f[n] == -1) { f[n] = fib(n-1) + fib(n-2); }
         return f[n];
 }
```

```
MAX_N = 1000
f = [-1 for i in range(MAX_N)]

def fib(n):
        if n < 2: return 1
        if f[n] != -1: return f[n]

        f[n] = fib(n-1) + fib(n-2)
        return f[n]
```

De esta forma, cada número de Fibonacci sólo se calcula una sola vez, y en las llamadas subsecuentes a `fib(n)` simplemente se retorna el valor que se tiene guardado. Esto mejora la complejidad temporal del programa de exponencial a lineal.

Esta idea tan sencilla es la base de la técnica conocida como programación dinámica: evitar volver a realizar cálculos que ya se han hecho. El ejemplo que hemos puesto es muy simple y puede dar la impresión de ser una trivialidad, pero lo cierto es que esta técnica es muy poderosa para resolver una gran variedad de problemas y, más allá de almacenar los resultados de una función para valores que ya se han calculado, existen ciertos trucos menos intuitivos también basados en la idea de “no volver a resolver problemas ya resueltos” que permiten optimizar programas de forma sorprendente.

Vamos a ver otro ejemplo sencillo de aplicación de la técnica de Programación Dinámica. Revisamos el problema del cambio en monedas, que ya comentamos en el manual de algoritmos voraces y dimos la solución en el caso particular de las monedas de euros. Aquí trataremos el caso general:

Ejemplo. (Cambio de monedas) *Dada una cantidad de dinero \(N\) y una lista de denominaciones de monedas \(\{x_1, x_2, \ldots, x_m\}\), dar el mínimo número de monedas cuyo valor sume \(N\).*

Pensamos cómo podemos calcular la solución recursivamente. Una cantidad de dinero N puede “provenir” de una cantidad de dinero $N-x$ a la que se le añade una moneda de valor x. Es decir, una colección de monedas óptima (es decir, con el menor número posible de monedas) que sume N y que contenga una moneda de valor x se construye a partir de una colección de monedas óptima que sume $N-x$ a la que se le añada una moneda de valor x. No podemos presuponer que una colección óptima contenga ningún tipo de moneda en concreto, así que para calcular el mínimo número de monedas necesarias para sumar N calculamos el mínimo entre los mínimos números de monedas necesarias para sumar $N-x_i$ para $i=1, \ldots, m$ y le sumamos 1. En código:

```
int cambio(int n) {
    if (n == 0)  {
         return 0;
    }
    int res = -1;
    for(int i=0; i < m; ++i) {
        if (x[i] <= n and cambio(n-x[i]) != -1) {
            if (res == -1) {
                res = cambio(n-x[i])+1;
            } else {
                res = min(res, cambio(n-x[i])+1);
            }
        }
    }
    return res; //si no se puede obtener el cambio, retorna -1
}

def cambio(n):
    if n == 0:
         return 0

    res = -1
    for i in range(m):
        if x[i] <= n and cambio(n-x[i]) != -1:
            if res == -1:
                res = cambio(n-x[i])+1
            else:
                res = min(res, cambio(n-x[i])+1)

    return res; #si no se puede obtener el cambio, retorna -1
```

La forma de optimizar esta función recursiva es la misma que para el cálculo de números de Fibonacci: almacenamos los valores de `cambio(n)` para sólo calcularlos una vez. Así, cada uno de los valores desde `cambio(1)` a `cambio(n)` lo calculamos como mucho una vez (como mucho porque, dependiendo de los valores de x_i, algunos no los necesitaremos), y como se emplea $\mathcal{O}(m)$ en calcular cada uno la complejidad temporal total del algoritmo es $\mathcal{O}(mn)$. Los ejemplos que hemos visto de programación dinámica siguen la idea de “coger una función recursiva y optimizarla almacenando valores ya calculados anteriormente”. Este tipo de planteamiento recibe el nombre de dinámica “de arriba a abajo” (*top-down*) o memoización.

También hay otra forma de trabajar con programación dinámica. Volvamos a pensar el problema de cambio de monedas, ahora sin tener en mente la idea de la recursividad. Otra forma de enfocar el problema es ir ‘construyendo’ las colecciones mínimas de monedas que suman una determinada cantidad. Por ejemplo, si para la cantidad de dinero x_1 sabemos que sólo necesitamos una moneda, entonces para las cantidades de dinero x_1+x_i para $i=1, \ldots, m$ sabemos que como mucho necesitamos dos monedas. En general, si tenemos el mínimo número de monedas necesario para una cantidad de dinero determinada N, entonces sabemos que para las cantidades $N+x_i$ el número mínimo de monedas necesario es como máximo uno más. La gracia de esta forma de pensar el

problema es que podemos ir calculando los valores de forma iterativa: una vez ya hemos procesado todos los valores anteriores a una cantidad N, ya sabemos cuál es el número mínimo de monedas para sumar N. El código es este:

```
vector<int> cambio; //Inicializado a -1, tamaño suficientemente grande
cambio[0] = 0;
for(int n=0; n <= N; ++n) {
    if(cambio[n] != -1) {
        for(int i=0; i < m; ++i) {
            if (cambio[n+x[i]] == -1) {
                cambio[n+x[i]] = cambio[n]+1;
            } else {
                cambio[n+x[i]] = min(cambio[n+x[i]], cambio[n]+1);
            }
        }
    }
}
```

```
cambio = [-1 for i in range(N)]
cambio[0] = 0

for n in range(N):
    if cambio[n] != -1:
        for i in range(m):
            if cambio[n+x[i]] == -1:
                cambio[n+x[i]] = cambio[n]+1
            else:
                cambio[n+x[i]] = min(cambio[n+x[i]], cambio[n]+1)
```

En cierto modo estamos haciendo lo mismo que antes, ya que basamos los cálculos del valor de `cambio[n]` en función de los valores anteriores, sólo que ahora en lugar de ir calculando los valores conforme los va “pidiendo” la recursividad, los vamos construyendo “desde abajo”. Por eso, este planteamiento se llama “de abajo a arriba” (*bottom-up*) o constructivo. Nótese que, mientras que con la implementación *top-down* algunos valores de `cambio[n]` no se calculaban, aquí calculamos todos desde 0 hasta N. Volvamos al ejemplo del cálculo de números de Fibonacci. La forma de calcularlos con una dinámica *bottom-up* es así:

```
vector<int> fib(n+1);
fib[0] = 1;
fib[1] = 1;
for (int i=2; i <= n; ++i) {
      fib[i] = fib[i-1] + fib[i-2];
}
```

Con este código nos guardamos los valores de todos los números de Fibonacci desde 0 hasta n, pero si sólo queremos calcular el enésimo la forma obvia de hacerlo es la siguiente:

```
int fib1 = 1;
int fib2 = 1;
for (int i=2; i <= n; ++i) {
    int t = fib1;
    fib1 = fib1 + fib2;
    fib2 = t;
} //fib1 contiene el n-ésimo número fibonacci
```

En general, en un problema de programación dinámica planteado de forma *bottom-up* se puede ahorrar memoria almacenando solo los resultados necesarios para la recurrencia. Volviendo al ejemplo de el cambio de las monedas: si \(x_{\text{max}}\) es el valor máximo de una moneda, entonces sólo necesitamos que nuestro vector de valores de cambio tenga una longitud de \(x_{\text{max}}+1\) . La implementación es la siguiente:

```
vector<int> cambio; //Inicializado a -1, tamaño igual a xmax+1
cambio[0] = 0;
for(int n=0; n < N; ++n) {
    if(cambio[0] != -1) {
        for(int i=0; i < m; ++i) {
            if (cambio[x[i]] == -1) {
                cambio[x[i]] = cambio[0]+1;
            } else {
                cambio[x[i]] = min(cambio[x[i]], cambio[0]+1);
            }
      }
   }
//cambio[0] ya no nos interesa; desplazamos los valores
   for(int i=0; i < xmax; ++i) {
      cambio[i] = cambio[i+1];
   }
   cambio[xmax] = -1;
}
//cambio[0] contiene el minimo numero de monedas que suman N
```

Estos ejemplos que hemos trabajado no ilustran todas las diferencias entre ambos métodos y las ventajas e inconvenientes de cada uno, pero su simplicidad permite entender los fundamentos de los dos: la idea principal es que el método *top-down* está basado en la recursividad, mientras que el método *bottom-up* está basado en la iteratividad.

Etiquetas

Dinámica

Añadir nuevo comentario

Acerca de formatos de texto

Texto sin formato

- No se permiten etiquetas HTML.

- Saltos automáticos de líneas y de párrafos.
- Las direcciones de correos electrónicos y páginas web se convierten en enlaces automáticamente.