Algoritmos y Estructuras de Datos III
Práctica: Programación Dinámica

Guido Tagliavini Ponce

03/09/2014

1. Repaso

¿Qué es programación dinámica (de ahora en más PD)? Es una técnica que consiste en resolver un problema que tenga una estructura recursiva, sin repetir cómputos. Que *el problema tenga una estructura recursiva* significa que su solución se puede expresar en términos de subproblemas similares más pequeños. *No repetir cómputos* significa que la solución de cada subproblema sólo se computa una vez. Por este motivo, la técnica es útil cuando existe un solapamiento de subproblemas.

Un esquema básico de una solución que usa PD es el siguiente:

1. Dividir un problema en uno o más subproblemas.
2. Resolver recursivamente cada uno de estos subproblemas. Si ya habíamos computado dicha subsolución, la devolvemos directamente.
3. Usando estas subsoluciones, computar la solución al problema y guardarla.

Notemos que cada vez que terminamos de computar una solución, la guardamos, y cada vez que preguntamos por una subsolución, la devolvemos en caso de haberla computado previamente. Por esto es que nunca repite un cómputo. Este esquema lleva el nombre de *top-down*, puesto que primero intenta resolver los problemas más grandes, dividiéndolos sucesivamente en subproblemas, hasta llegar a un caso base. La técnica de almacenar las soluciones ya computadas a modo de caché, recibe el nombre de *memoización*.

Existe otro esquema llamado *bottom-up*, que consiste comenzar resolviendo las instancias más chicas, y terminando en las más grandes. Este esquema permite realizar el cómputo iterativamente.

¿Y qué es mejor? ¿Bottom-up o top-down? Depende del problema. En términos de eficiencia, una solución bottom-up computa *todas* las subsoluciones, mientras que una top-down sólo computa aquellas necesarias, con lo cual, si la solución al problema en realidad requiere sólo una pequeña parte de las subsoluciones, entonces top-down resulta mejor. En caso contrario, bottom-up suele superar a top-down, puesto que es iterativo y sus ventajas frente a una función recursiva son bien sabidas. En términos asintóticos, la complejidad de ambas versiones es, en general, la misma.

Por otro lado, el esquema top-down es un poco más sencillo, ya que para dar un esquema bottom-up se requiere conocer la dependencia entre los subproblemas. Esto es, dado un problema cualquiera, necesitamos saber de antemano cuáles subsoluciones necesitamos tener para computar la solución al problema. De todos modos suele ser fácil determinar un tal orden en que debemos realizar los cómputos.

1.1. PD en problemas de optimización

PD no sólo sirve en problemas de optimización, aunque es cierto que la mayoría de los problemas en los que aparece son de este tipo. En el contexto de un problema de optimización, para que exista una estructura recursiva es necesario que se cumpla el *Principio de Optimalidad*. Esto es, el problema debe poder resolverse utilizando únicamente soluciones *óptimas* a subproblemas.

2. El problema: Longest Common Subsequence

Definición. Dada una secuencia de caracteres $X = \langle x_1, \ldots, x_n \rangle$, una *subsecuencia de* X es una secuencia $Z = \langle z_1, \ldots, z_k \rangle$ que cumple que existen índices $1 \leq i_1 < \ldots < i_k \leq n$ tal que $z_j = x_{i_j}$ para todo $j = 1, \cdots, k$. Intuitivamente, Z es el resultado de tomar X y quitarle cero o más caracteres.

Dadas dos secuencias de caracteres $X = \langle x_1, \ldots, x_n \rangle$ e $Y = \langle y_1, \ldots, y_m \rangle$, se desea encontrar la longitud de una subsecuencia común más larga. Este problema se conoce como *Longest Common Subsequence* (o LCS).

Por ejemplo, si $X = \langle a, b, b, a \rangle$ e $Y = \langle b, a, b, a \rangle$, entonces $\langle a \rangle$, $\langle a, a \rangle$ y $\langle b, b \rangle$ son subsecuencias comunes de X e Y. Una LCS de X e Y es $\langle a, b, a \rangle$. Tiene longitud máxima puesto que una subsecuencia común es, en particular, una subsecuencia de X, con lo cual tiene longitud a lo sumo $|X| = 4$. Sin embargo no hay subsecuencias comunes de longitud 4, puesto que la única subsecuencia de X de longitud 4 es la misma secuencia, pero X no es subsecuencia de Y. Observemos que la LCS no siempre es única. Por ejemplo, $\langle b, b, a \rangle$ es otra LCS en este caso.

3. Solución

Una solución naïve sería computar la lista de todas las subsecuencias de X y, para cada una de ellas, verificar si es una subsecuencia de Y, quedándonos con la más larga que lo sea. Como hay tantas subsecuencias de X como subconjuntos de $\{1, \ldots, n\}$, resulta que son 2^n subsecuencias a testear, con lo cual el algoritmo es $\Omega(2^n)$, así que hay que pensar en algo mejor.

3.1. Formulación recursiva

Lo primero que debemos hacer es encontrar una formulación recursiva del problema. Vamos a ver que lo que sirve en este caso es mirar el problema de LCS sobre los prefijos de ambas cadenas.

Definición. Dada una secuencia de caracteres $X = \langle x_1, \cdots, x_n \rangle$, un *prefijo de* X es una secuencia $X(i) = \langle x_1, \cdots, x_i \rangle$, con $0 \leq i \leq n$. Definimos $X(0)$ como la secuencia vacía.

La pregunta que nos hacemos es: ¿cómo es una LCS de $X(i)$ e $Y(j)$? Concretamente, para cada $0 \leq i \leq n$ y $0 \leq j \leq m$, queremos computar el valor de

$$f(i, j) = \text{longitud de una LCS de } X(i) \text{ e } Y(j)$$

¿Por qué queremos computar esto? ¿Cómo sabemos que esto se puede computar más o menos fácilmente? Esto les puede sonar a galerazo, y lo es. Pero ahora cada vez que vean un problema que involucre secuencias de caracteres, van a pensar en LCS y la idea de considerar el problema sobre los prefijos (aunque esto no significa que todo problema de este tipo se resuelva de esta forma). De todas formas, uno podría intentar buscarle una explicación al por qué de considerar esta función, y no cualquier otra. Dado que queremos una formulación recursiva, necesitamos ser capaces de expresar cierto problema en función de subproblemas, para lo cual necesitamos definir qué es un subproblema. Para poder hablar de un subproblema, tiene que haber algún parámetro del problema que decrezca de una instancia a otra más pequeña. Es razonable pensar que dichos parámetros sean las longitudes de ciertas subcadenas de X e Y.

En cualquier caso, es evidente que esta función está muy relacionada con nuestro problema de calcular la LCS de X e Y. De hecho $f(n, m)$ es exactamente la longitud de una LCS de $X(n) = X$ e $Y(m) = Y$, el resultado que buscamos.

El resultado clave es el siguiente.

Teorema (LCS sobre prefijos cumple el Principio de Optimalidad)**.** *Sean* $X(i) = \langle x_1, \ldots, x_i \rangle$ *e* $Y(j) = \langle y_1, \ldots, y_j \rangle$ *dos prefijos. Sea* $Z = \langle z_1, \ldots, z_k \rangle$ *una LCS cualquiera de* $X(i)$ *e* $Y(j)$*.*

1. *Si* $x_i = y_j$ *entonces* $z_k = x_i = y_j$ *y* $Z(k-1)$ *es una LCS de* $X(i-1)$ *e* $Y(j-1)$*.*
2. *Si* $x_i \neq y_j$ *y* $x_i \neq z_k$ *entonces* Z *es una LCS de* $X(i-1)$ *e* $Y(j)$*.*
3. *Si* $x_i \neq y_j$ *e* $y_j \neq z_k$ *entonces* Z *es una LCS de* $X(i)$ *e* $Y(j-1)$*.*

Demostración. 1. Supongamos, por el absurdo, que $z_k \neq x_i = y_j$. Entonces Z es una subsecuencia común de $X(i-1)$ e $Y(j-1)$ de longitud, obviamente, máxima. Entonces agregando x_i al final de Z obtenemos una subsecuencia de $X(i)$ e $Y(j)$, que es más larga que Z, lo cual es una contradicción. Luego, debe ser $z_k = x_i = y_j$.

Entonces, es claro que $Z(k-1)$ es una subsecuencia de $X(i-1)$ e $Y(j-1)$. Si no tuviera longitud máxima, entonces existiría una subsecuencia W de longitud $|W| > k-1$, común a $X(i-1)$ e $Y(j-1)$.

Pero entonces al agregar $x_i = y_j$ al final de W, obtenemos una subsecuencia común de $X(i)$ e $Y(j)$, de longitud $|W| + 1 > k = |Z|$, lo cual es absurdo, pues supusimos que Z tenía longitud máxima entre todas las subsecuencias de $X(i)$ e $Y(j)$.

2. Si $z_k \neq x_i$ entonces Z tiene que ser una subsecuencia común de $X(i-1)$ e $Y(j)$. Si no tuviera longitud máxima, podríamos tomar una mejor, obteniendo así una subsecuencia común de $X(i-1)$ e $Y(j)$, y por lo tanto de $X(i)$ e $Y(j)$, que es más larga que Z, llegando a un absurdo.

3. Análogo a 2.

□

Este resultado muestra que una LCS de $X(i)$ e $Y(j)$ se puede construir a partir de una LCS de prefijos de $X(i)$ e $Y(j)$. Es decir, una subsecuencia común más larga se construye a partir de subsecuencias comunes de los prefijos de $X(i)$ e $Y(j)$, pero no son subsecuencias cualesquiera, sino que *son de longitud máxima*. Por eso decimos que se cumple el Principio de Optimalidad.

El teorema también nos indica que para calcular una LCS, a lo sumo necesitamos conocer dos subproblemas. Si $x_i = y_j$, entonces una LCS de $X(i)$ e $Y(j)$ se obtiene agregándole x_i a una LCS de $X(i-1)$ e $Y(j-1)$. Si $x_i \neq y_j$ entonces una LCS de $X(i)$ e $Y(j)$ es una LCS de $X(i-1)$ e $Y(j)$, o bien es una LCS de $X(i)$ e $Y(j-1)$. A priori no sabemos cuál, pero necesariamente es la más larga de esas dos opciones. Concluimos que si $i, j \geq 1$ entonces

$$f(i,j) = \begin{cases} 1 + f(i-1, j-1) & \text{si } x_i = y_j \\ \max\{f(i-1,j), f(i,j-1)\} & \text{si } x_i \neq y_j \end{cases}$$

Los casos base son

$$f(0,j) = f(i,0) = f(0,0) = 0$$

¿Cómo se refleja el Principio de Optimalidad en esta fórmula recursiva? Esta fórmula expresa $f(i,j)$ en términos de $f(i-1,j-1)$, $f(i-1,j)$ y $f(i,j-1)$. Estas instancias más pequeñas son *subsoluciones óptimas*. Si una LCS de $X(i)$ e $Y(j)$ se construyera a partir de subsecuencias comunes de prefijos no necesariamente óptimas, entonces no tendría sentido hablar de $f(i-1,j-1)$, $f(i-1,j)$ y $f(i,j-1)$, ya que f *sólo habla de óptimos*. En definitiva, si no valiera el Principio de Optimalidad, no habría fórmula recursiva.

3.2. Solapamiento de subproblemas

Observen que al calcular $f(i,j)$, podríamos usar $f(i-1,j)$, y esta instancia a su vez posiblemente use $f(i-1,j-1)$. Ahora bien, también usaríamos $f(i,j-1)$ y este subproblema a su vez podría usar $f(i-1,j-1)$. Esto muestra que $f(i-1,j-1)$ es un subproblema compartido, y es evidente que lo mismo sucede para subproblemas más pequeños.

Tenemos una formulación recursiva y los subproblemas se solapan. Entonces PD es una buena idea para resolver este problema.

3.3. Algoritmo

Supongamos que queremos construir un primer algoritmo recursivo, basándonos solamente en la fórmula recursiva que calculamos antes, por más ineficiente que resulte. El algoritmo sería el siguiente.

```
1  LCS-Naive(X = ⟨x_1, ··· , x_i⟩, Y = ⟨y_1, ··· , y_j⟩)
2  begin
3      if i = 0 or j = 0 then
4          return 0
5      end
6      if x_i = y_j then
7          return 1 + LCS-Naive(X(i − 1), Y(j − 1))
8      end
9      return máx{LCS-Naive(X, Y(j − 1)), LCS-Naive(X(i − 1), Y)}
10 end
```

Pasar de este algoritmo a una versión top-down de PD es muy fácil. Sólo hay que agregarle una estructura que vaya almacenando los valores que vamos computando, y cada vez que se necesite un valor que ya fue computado, lo devolvemos directamente.

La estructura que vamos a usar es una matriz de enteros dimensión $(n+1) \times (m+1)$, que llamamos dp. Supondremos que es una variable global, y además que $dp[i][j] = -1 \Leftrightarrow f(i,j)$ no fue calculado aún. Usamos el valor -1 para distinguir lo calculado de lo no calculado, puesto que $f(i,j) \geq 0$ para todo i, j.

```
1  INIT()
2  begin
3      for i = 0 to n do
4          for j = 0 to m do
5              dp[i][j] = −1
6          end
7      end
8  end
```

```
1  LCS-TOP-DOWN(X = ⟨x_1, ··· , x_i⟩, Y = ⟨y_1, ··· , y_j⟩)
2  begin
3      if dp[i][j] ≠ −1 then
4          return dp[i][j]
5      end
6      if i = 0 or j = 0 then
7          dp[i][j] = 0
8          return 0
9      end
10     if x_i = y_j then
11         dp[i][j] = 1+LCS-TOP-DOWN(X(i − 1), Y(j − 1))
12     else
13         dp[i][j] = máx{LCS-TOP-DOWN(X, Y(j − 1)), LCS-TOP-DOWN(X(i − 1), Y)}
14     end
15     return dp[i][j]
16 end
```

Primero debemos llamar a INIT(), puesto que inicialmente no hay ninguna entrada de dp calculada, y luego a LCS-TOP-DOWN(X, Y).

Finalmente queremos dar una versión bottom-up. Para esto tenemos que deducir en qué orden tenemos que computar los valores de f, de modo tal que al intentar computar un valor cualquiera $f(i,j)$, los subproblemas de los que depende ya hayan sido resueltos. Recordemos que $f(i,j)$ depende de $f(i-1, j-1)$, $f(i-1, j)$ y $f(i, j-1)$. Acá conviene pensar a f en su forma de matriz dp. Lo que está sucediendo es que un elemento $dp[i][j]$ depende de su casillero izquierdo, su casillero superior, y su casillero diagonal superior izquierdo. Por lo tanto, podemos completar la matriz dp de forma natural, es decir, fila a fila, comenzando por la primera de todas, y recorriendo cada una de izquierda a derecha. Notemos que esta no es la única forma de llenar la matriz. Otra opción es, por ejemplo, recorrer por columnas, comenzando desde la primera, y recorriendo cada una de arriba a abajo.

```
1  LCS-Bottom-Up(X = ⟨x_1, ··· , x_n⟩, Y = ⟨y_1, ··· , y_m⟩)
2  begin
3      dp[0][0] = 0
4      for i = 1 to n do
5          dp[i][0] = 0
6      end
7      for j = 1 to m do
8          dp[0][j] = 0
9      end
10     for i = 1 to n do
11         for j = 1 to m do
12             if x_i = y_j then
13                 dp[i][j] = 1 + dp[i − 1][j − 1]
14             else
15                 dp[i][j] = máx{dp[i][j − 1], dp[i − 1][j]}
16             end
17         end
18     end
19     return dp[n][m]
20 end
```

3.4. Complejidad

Calculemos el costo temporal y espacial de las dos versiones que usan PD. Se puede ver que ambas tienen un costo espacial $O(nm)$, proporcional al tamaño de la matriz dp que usan.

Desde el punto de vista temporal, la más fácil de analizar es la versión bottom-up, que es evidente que tiene un costo $O(nm)$. Afirmamos que la versión top-down corre en ese mismo tiempo, y vamos a dar un argumento de esto. En primer lugar, Init es $O(nm)$. Resta ver que LCS-Top-Down también lo es.

Podemos pensar al costo total de LCS-Top-Down(X, Y), como la suma del costo de cada instrucción que ejecuta, sin tener en cuenta el costo de las llamadas recursivas que dicha instrucción pueda hacer. Como cada una consume tiempo $O(1)$, basta contar la cantidad de instrucciones ejecutadas en total.

Llamemos *estado* a cada par (i, j) para el que queremos computar $dp[i][j]$ a lo largo de la ejecución. Un estado es *computado* cuando computamos su entrada asociada en dp. Podemos dividir a la función en dos partes:

(I1) El costo de computar un nuevo estado (lineas 6 a 15).

(I2) El costo de retornar el valor para un estado ya computado (lineas 3 a 5).

Queremos contar la cantidad de veces que se ejecutan las instrucciones de cada parte.

Notemos que en total son $O(nm)$ estados. Como cada estado se computa una sóla vez, entonces las instrucciones de la parte (I1) se ejecutan $O(nm)$ veces, y por lo tanto, como son $O(1)$ instrucciones, el costo aportado por todas ellas es $O(nm)$.

Parece más difícil estimar el tiempo que agregan las instrucciones (I2). Estas instrucciones sólo se ejecutan en llamadas que simplemente retornan un valor de dp ya computado, y tienen costo $O(1)$. Además provienen de otra instancia de la función que se encontraba computando un estado, i. e., ejecutando instrucciones (I1). Por lo tanto, podemos pensar dicho costo $O(1)$, como parte del costo de la instancia llamadora. Esto no modifica el costo $O(nm)$ sobre todos los costos de instrucciones (I1), puesto que cada cómputo de un valor de dp realiza, a lo sumo, 2 llamadas recursivas.

En definitiva, LCS-Top-Down(X, Y) corre en tiempo $O(nm)$.

Como regla general, podemos decir que un algoritmo que use PD tendrá un costo $O(SC)$ donde S es la cantidad de estados y C es el costo de computar cada uno. Pero esto es sólo una regla informal. Sólo deben usarla para hacer estimaciones, y siempre que se les pida deben probar que la complejidad sea la afirmada.

4. Comentarios finales

- Recordemos que en el repaso inicial dijimos que "una solución bottom-up computa todas las subsoluciones, mientras que una top-down solo computa aquellas necesarias". ¿Se cumple eso en este caso?

  En principio, es evidente que LCS-Bottom-Up computa absolutamente todos los valores de f, puesto que llena toda la matriz dp. Para ver que LCS-Top-Down sólo computa los valores que necesita, pensemos qué pasa, por ejemplo, en el caso $X = Y$ (y por ende $n = m$). Al llamar a LCS-Top-Down(X, Y), tendremos $i = j$ y $x_i = y_j$, con lo cual se realiza la llamada recursiva LCS-Top-Down($X(n-1)$, $Y(n-1)$). En esta nueva instancia de la función, vuelve a suceder que el último elemento de los dos prefijos del input coincide, con lo cual se llama a LCS-Top-Down($X(n-2)$, $Y(n-2)$). Así sucesivamente, terminando en el momento en que ambos argumentos sean la secuencia vacía. En términos de la matriz dp que vamos llenando, sólo estamos computando las entradas de la diagonal.

  Sin embargo, esto *no* se traduce en un mejor costo temporal. Si bien, en este caso particular, LCS-Top-Down corre en tiempo $O(n)$, la complejidad de todo el esquema top-down es $\Omega(nm)$ debido a Init. Por esta razón, en este y en cualquier otro caso, la complejidad del algoritmo top-down es $O(nm)$ y no mejor, i. e. , es $\Theta(nm)$.

- Muchas veces les van a pedir que prueben que el algoritmo que proponen es correcto. En este caso (y en general) la demostrar la correctitud es, escencialmente, probar que la expresión dada de f es correcta. El algoritmo propiamente dicho sólo se limita a computar en base a lo indicado por dicha función f, y es evidente que lo hace correctamente.

  Si implementan un esquema bottom-up, deben probar que realizan los cómputos respetando las dependencias. De todos modos esto, en general, es sencillo.

- Nunca hablamos de la terminación de los algoritmos. La versión bottom-up termina claramente. ¿Por qué termina la versión top-down? Una justificación sencilla es que en cada llamada recursiva, alguno de los dos argumentos *decrece* y cuando alguno es vacío llegamos a un caso base. Formalmente, estamos considerando la función variante $v(\langle x_1, \ldots, x_i\rangle, \langle y_1, \ldots, y_j\rangle) = i + j \geq 0$, de modo tal que en cada instancia, si no se ha llegado a un caso base, entonces en la siguiente llamada recursiva v decrece estrictamente. Cuando v vale cero, necesariamente llegamos a un caso base.

5. Tarea

1. Programar las dos versiones del algoritmo que usan PD. Para verificar sus programas, pueden usar el siguiente juez online:

   `http://uva.onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=16&page=show_problem&problem=1346`

2. ¿Cómo harían para efectivamente dar una LCS de las secuencias de entrada?

3. En la versión bottom-up, para llenar cada fila de la matriz sólo estamos usando la anterior. Dar una nueva versión del algoritmo que use memoria $O(\text{mín}\{n, m\})$.

4. Usando las ideas presentadas, hacer el Ejercicio 3.11.

Referencias

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2001.

[2] Robert A. Wagner and Michael J. Fischer. The string-to-string correction problem. *J. ACM*, 21(1):168–173, January 1974.