Universidad de Valladolid

Programación Dinámica

Subsecuencia Común Más Larga

Sergio García Prado

Seguimiento del trabajo en:
https://github.com/garciparedes/Longest-Common-Subsequence

November 11, 2015

1 Introducción

1.1 Definición del problema

El problema analizado que se pretende resolver consiste en lo siguiente:

Primeramente describiremos el concepto secuencia para después explicar a qué nos referimos con subsecuencia, ya que es una de las ideas fundamentales que habrá que tener claras para entender las especificaciones que se nos piden para resolver el problema.

Secuencia: Es una colección ordenada de elementos en la cual la repetición está permitida.
Ejemplo: A, B, C, A, E, F

Subsecuencia: Es una secuencia que se obtiene a partir de otra de igual o mayor longitud mediante la supresión de algunos elementos manteniendo el orden de los elementos restantes. Por ejemplo, la secuencia A, C, A es una subsecuencia de A, B, C, A, E, F.
No debe confundirse con el término subcadena, que además impone la restricción de que los elementos han de ser contiguos. Un ejemplo de subcadena es C, A, E.

Ahora que ya tenemos clara la definición de subsecuencia modelizaremos el problema a resolver:

Dadas dos secuencias de longitud arbitraria, nuestro objetivo es encontrar la subsecuencia común de mayor longitud entre ambas.

1.2 Aplicaciones

Este algoritmo tiene una gran cantidad de aplicaciones. Uno de los sectores donde su uso está más extendido es en el de la informática: se usa en software de control de versiones como **git** y en el comando **diff** de linux, que muestra las diferencias entre ficheros. También se utiliza en el sector de la bioinformática (aplicación de la tecnología de computadores a la gestión y análisis de datos biológicos.) para **secuenciación de ADN**.

2 Programación Dinámica

2.1 Definición

La programación dinámica es un patrón de diseño de algoritmos basado en la división del problema base en subproblemas de menor tamaño y complejidad que se resolverán una única vez, ya que se almacenará la solución de cada uno de ellos para luego reutilizarlo en el caso de que fuera necesario. Al proceso de almacenar las soluciones de los subproblemas se lo denomina **memoization**. Para que un problema pueda ser resuelto mediante programación dinámica tiene que cumplir dos propiedades: solapamiento de los subproblemas y subestructura óptima.

2.2 Solapamiento de Problemas

Se dice que un problema tiene esta propiedad si se puede subdividir en problemas de menor tamaño cuyos resultados se pueden reutilizar para resolver sucesivos subproblemas que lo contienen. El ejemplo típico de esta propiedad es la Sucesión de Fibonacci, que se define como:

$$F(n) = \begin{cases} n & \text{if } n \leq 1 \\ F(n-1) + F(n-2) & \text{otherwise} \end{cases}$$

Gráficamente esto se puede representar como un árbol binario, en el cual, para calcular un nodo, tenemos que recurrir a los resultados de sus dos hijos, y así recursivamente hasta llegar a tener como hijos$F(0), F(1)$, que son nuestros casos base:

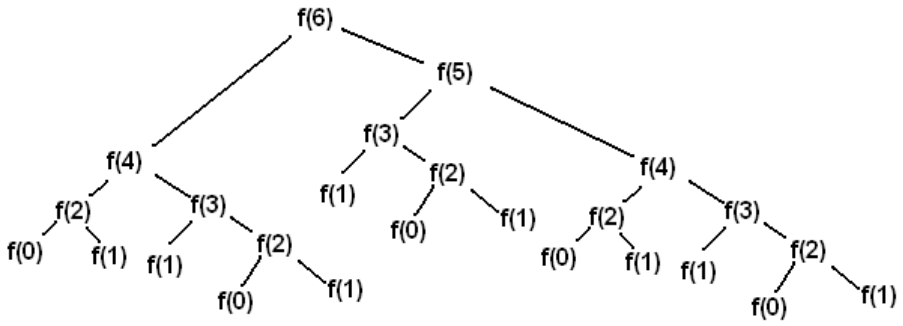

Figure 1: Sucesión de Fibonacci

2.3 Subestructura Óptima

La segunda propiedad que debe cumplir el problema es la de subestructura óptima, que consiste en lo siguiente: La solución al problema contiene (o depende) de las soluciones óptimas a sus subproblemas. Esta es una propiedad que la programación dinámica comparte con los algoritmos voraces.

3 Solución

3.1 Enfoques Disponibles

Para encontrar la subsecuencia común más larga entre $N[n_1...n_n]$ secuencias existen distintos enfoques:

Primeramente podríamos pensar en un enfoque por fuerza bruta, es decir, comparando todas las subsecuencias posibles. Esto tendría un crecimiento asintótico de $O(2^{n_1} \sum_{i=2}^{N} n_i)$ que corresponde al coste de obtener todas las subsecuencias de la primera secuencia $O(2^{n_1})$ y compararlo con todas las demás (sumatorio de todas las restantes), es decir, de compararlas todas con todas.

Otra solución es utilizar un enfoque dinámico, ya que como demostraremos posteriormente, nuestro problema posee las propiedades necesarias para ser resuelto de esta manera. Con este enfoque obtenemos un crecimiento asintótico de $O(N \prod_{i=1}^{N} n_i)$ que corresponde al coste de generar la matriz que almacena las longitudes de subsecuencias comunes (memoization).

Nótese que para valores muy grandes de N el problema se transforma en uno de tipo **NP-hard**, es decir, problemas de decisión que son como mínimo tan difíciles como un problema de NP.

3.2 Enfoque dinámico

Veamos como encontrar la subsecuencia común más larga se puede resolver por programación dinámica. Abordaremos el problema para dos secuencias, pero este es extrapolable a cualquier número de secuencias fácilmente incrementando el tamaño de la entrada de nuestro algoritmo.

3.2.1 Notación

Denominaremos a las dos secuencias $X[0...m-1]$ e $Y[0...n-1]$, cuya longitud será m y n respectivamente. Sean $i \in (0, m-1)$ y $j \in (0, n-1)$
A modo de ejemplo supondremos $X = abcde, Y = aert$ por lo que $m = 5, n = 4$.

También que tenemos una matriz de L[0...m-1][0...n-1] de tamaño m x n, la cual utilizaremos para almacenar los resultados de cada subproblema

Sea S una secuencia, l su longitud e $i \in (1, l)$, la secuencia S_i es la correspondiente a los i primeros elementos.
Ejemplo: $X_1 = a$, $X_3 = abc$ y $X_5 = abcde = X$

3.2.2 Subestructura óptima

En el caso de que X[m-i] sea igual a Y[n-j] (las últimas posiciones de cada secuencia), rellenaremos la matriz L con el valor de las secuencias sin esos últimos elementos + 1 (el elemento común encontrado):

$$L(X[0..m-i], Y[0..n-j]) = 1 + L(X[0..m-1-i], Y[0..n-1-j])$$

Ejemplo:

$$L(abcde, aert) = 1 + L(abcd, aer)$$

Por contra, si los dos últimos elementos no fueran iguales, es decir, si X[m-i] fuese distinto de Y[n-1] tendríamos que obtener el máximo de eliminar el último carácter de la primera secuencia y de la segunda, es decir:

$$L(X[0..m-i], Y[0..n-j]) = max(L(X[0..m-1-i], Y[0..n-j]), L(X[0..m-i], Y[0..n-1-j])$$

Ejemplo:

$$L(abcd, aer) = max(L(abc, aer), L(abcd, ae))$$

La formulación completa del problema es la siguiente:

$$LCS(X_i, Y_j) = \begin{cases} 0 & \text{if } i = 0 \vee j = 0 \\ LCS(X_{i-1}, Y_{j-1}) + 1 & \text{if } x_i = y_j \\ max(LCS(X_i, Y_{j-1}), LCS(X_{i-1}, Y_j)) & \text{if } x_i \neq y_j \end{cases}$$

Para rellenar la matriz L aplicaremos la función $LCS(X_i, Y_j)$ a cada una de las entradas L_{ij}.

3.2.3 Solapamiento de problemas

Como se puede apreciar, el resultado de cada subproblema se reutiliza para obtener el de los de nivel superior. Por tanto podemos almacenar el resultado para calcularlo una única vez al igual que en el caso de la Sucesión de Fibonacci.

3.2.4 Obtención de los resultados

Para obtener la longitud de la subsecuencia común más larga tan solo tendremos que obtener el valor de L_{mn}. Este valor se encuentra en esa posición dado que es la última entrada de la matriz que se rellena, es decir, el corresponde a las secuencias completas.

Para obtener la subsecuencia común más larga el proceso es algo más complicado. A grandes rasgos consiste en comenzar en la entrada L_{mn} e ir decrementando los valores de i y j añadiendo a la secuencia el elemento correspondiente a dicha posición en el caso de que $X_i = Y_j$ o en caso contrario decrementar i o j según sea mayor X_i o Y_j hasta que $i = 0$ o $j = 0$. La secuencia resultante será la subsecuencia común más larga.

Este algoritmo se puede adaptar fácilmente para encontrar todas las subsecuencias comunes más largas en caso de que esta no fuera única simplemente recorriendo todos los caminos posibles (ya que este puede no ser único) que cumplan las condiciones anteriormente descritas.

3.2.5 Ejemplo de ejecución

Sean $X = abcde, Y = aert$ por lo que $m = 5, n = 4$.

Lo primero que haremos es aplicar el algoritmo para generar la matriz L.
Nota: Para mejorar la eficiencia del algoritmo aumentamos en 1 el tamaño de la matriz, es decir, $L[0...m][0...n]$ para añadir 0 en las entradas L_{ij} tales que $i = 0$ o $j = 0$, lo que reduce el número de comprobaciones respecto de los límites de la lista.

	∅	a	b	c	d	e
∅						
a						
e						
r						
t						

	∅	a	b	c	d	e
∅	0	0	0	0	0	0
a	0					
e	0					
r	0					
t	0					

	∅	a	b	c	d	e
∅	0	0	0	0	0	0
a	0	1				
e	0					
r	0					
t	0					

	∅	a	b	c	d	e
∅	0	0	0	0	0	0
a	0	1	1			
e	0					
r	0					
t	0					

	∅	a	b	c	d	e
∅	0	0	0	0	0	0
a	0	1	1	1	1	1
e	0	1				
r	0					
t	0					

	∅	a	b	c	d	e
∅	0	0	0	0	0	0
a	0	1	1	1	1	1
e	0	1	1	1	1	2
r	0					
t	0					

	∅	a	b	c	d	e
∅	0	0	0	0	0	0
a	0	1	1	1	1	1
e	0	1	1	1	1	2
r	0	1	1	1	1	2
t	0					

	∅	a	b	c	d	e
∅	0	0	0	0	0	0
a	0	1	1	1	1	1
e	0	1	1	1	1	2
r	0	1	1	1	1	2
t	0	1	1	1	1	2

Una vez rellenada la matriz ya conocemos la longitud de la subsecuencia común más larga, que, como dijimos antes se aloja en L_{mn}, es decir, L_{65} ya que en nuestro caso la matriz es más grande (sino sería L_{54}).

Para obtener la subsecuencia común más larga utilizaremos el algoritmo expuesto anteriormente:

	∅	a	b	c	d	e
∅	0	0	0	0	0	0
a	0	1	1	1	1	1
e	0	1	1	1	1	2
r	0	1	1	1	1	2
t	0	1	1	1	1	2

	∅	a	b	c	d	e
∅	0	0	0	0	0	0
a	0	1	1	1	1	1
e	0	1	1	1	1	2
r	0	1	1	1	1	2
t	0	1	1	1	1	2

	∅	a	b	c	d	e
∅	0	0	0	0	0	0
a	0	1	1	1	1	1
e	0	1	1	1	1	2
r	0	1	1	1	1	2
t	0	1	1	1	1	2

	∅	a	b	c	d	e
∅	0	0	0	0	0	0
a	0	1	1	1	1	1
e	0	1	1	1	1	2
r	0	1	1	1	1	2
t	0	1	1	1	1	2

	∅	a	b	c	d	e
∅	0	0	0	0	0	0
a	0	1	1	1	1	1
e	0	1	1	1	1	2
r	0	1	1	1	1	2
t	0	1	1	1	1	2

	∅	a	b	c	d	e
∅	0	0	0	0	0	0
a	0	1	1	1	1	1
e	0	1	1	1	1	2
r	0	1	1	1	1	2
t	0	1	1	1	1	2

Los resultados de la ejecución para las secuencias *abcde* y *aert* son los siguientes; $LCS = ae$ y su longitud 2.

3.3 PseudoCódigo

Algorithm 1 lcs

```
1:  function LCS(X, Y, m, n))
2:      X : secuencia de m elementos [0...m-1]
3:      Y : secuencia de n elementos [0...n-1]
4:      m : longitud de X
5:      n : longitud de Y
6:
7:      L ← array[0...m][0...n]                                   ▷ O(1)
8:
9:      for i ← 0, i < m + 1, i + + do                            ▷ O(m ∗ n)
10:         for j ← 0, i < n + 1, j + + do                        ▷ O(n)
11:             if i = 0 ∨ j = 0 then                             ▷ O(1)
12:                 L[i][j] ← 0                                   ▷ O(1)
13:             else if X[i − 1] = Y [j − 1] then                 ▷ O(1)
14:                 L[i][j] ← L[i − 1][j − 1] + 1                 ▷ O(1)
15:             else
16:                 L[i][j] ← max(L[i − 1][j], L[i][j − 1])       ▷ O(1)
17:             end if
18:         end for
19:     end for
20:
21:     index ← L[m][n]                                           ▷ O(1)
22:     LCS ← array[0...index − 1]                                ▷ O(1)
23:
24:     i ← m                                                     ▷ O(1)
25:     j ← n                                                     ▷ O(1)
26:     while i > 0 ∧ j > 0 do                                    ▷ O(n + m)
27:         if X[i − 1] = Y [j − 1] then                          ▷ O(1)
28:             LCS[index − 1] ← X[i − 1]                         ▷ O(1)
29:             index ← index − 1                                 ▷ O(1)
30:             j ← j − 1                                         ▷ O(1)
31:             j ← j − 1                                         ▷ O(1)
32:         else if L[i − 1][j] > L[i][j − 1] then                ▷ O(1)
33:             i ← i − 1                                         ▷ O(1)
34:         else
35:             j ← j − 1                                         ▷ O(1)
36:         end if
37:     end while
38:
39:     return LCS                                                ▷ O(1)
40: end function
```

3.4 Análisis asintótico

Como vemos, el orden de crecimiento asintótico de la función depende de la longitud de las secuencias de entrada, es decir, del coste de rellenar y recorrer la matriz L. El analisis sobre el crecimiento se ha hecho a partir del peor caso.

$$T(n) = O(lenX * lenY) + O(m + n) = O(lenX * lenY)$$

Con esto llegamos a la conclusión de que este algoritmo tienen un coste de $O(n^2)$ si suponemos que las dos secuencias de entrada tienen la misma longitud.

4 Referencia Bibliográfica

- https://en.wikipedia.org/wiki/Sequence
- https://en.wikipedia.org/wiki/Subsequence
- https://en.wikipedia.org/wiki/Bioinformatics
- https://en.wikipedia.org/wiki/Dynamic_programming
- https://www.cs.berkeley.edu/~vazirani/algorithms/chap6.pdf
- https://en.wikibooks.org/wiki/Algorithms/Dynamic_Programming
- http://faculty.ksu.edu.sa/Alsalih/CSC%20529/5.2_DynamicProgramming.pdf
- http://www.geeksforgeeks.org/dynamic-programming-set-4-longest-common-subsequence/