1. Longest Common Subsequence

1.1. Enunciado

Sean A y C dos cadenas de letras (strings) de tamaño n y k respectivamente, decimos que C es una subsecuencia de A si existen $i_1 < \ldots < i_k$ tales que $A_{i_1} = C_1, \ldots, A_{i_k} = C_k$.

El problema de *Longest Common Subsequence* (LCS) consiste en dadas dos cadenas A y B de longitud n y m, encontrar la subsecuencia C común a A y B más larga.

1.2. Análisis

Una subsecuencia C de A y B (de tamaño n y m) la identificamos por los índices $i_1 < \ldots < i_k$ y $j_1 < \ldots < j_k$ de A y B tal que $A_{i_1} \ldots A_{i_k} = B_{j_1} \ldots B_{j_k} = C$.

Para toda instancia (A, B) se cumple lo siguiente:

1. Si $A_1 = B_1$ entonces existe C^* subsecuencia más larga de A y B tal que $i_1 = j_1 = 1$.
2. Si $A_1 \neq B_1$ entonces en toda C^* subsecuencia más larga de A y B o bien $i_1 > 1$ o $j_1 > 1$.

Demostración

1. Por absurdo, supongamos que $A_1 = B_1$ pero $i_1 > 1$ o $j_1 > 1$. Si $i_1 > 1$ y $j_1 > 1$ entonces podemos armarnos C' agregando los índices de A_1 y B_1 al principio de la subsecuencia y obtenemos una mejor, absurdo.
   Si $i_1 > 1$ entonces $j_1 = 1$, pero podemos tomar $i_1 = 1$ y C' es una solución óptima con $i_1 = j_1 = 1$, absurdo. El caso con $j_1 > 1$ es análogo.
2. Trivial.

1.3. Formulación recursiva

Usando la propiedad demostrada anteriormente podemos plantear una formulación recursiva. La idea es tomar como caso base cuándo alguno de las dos cadenas es vacía, donde sabemos que la subsecuencia común más larga es la vacía. Luego, tenemos dos casos recursivos, uno cuando $A_1 = B_1$ y otro cuando $A_1 \neq B_1$.
Definimos una función f que dados dos string A y B nos dice la longitud de la subsecuencia común más larga.

$$f(A,B) = \begin{cases} 0 & si\ |A| = 0\ ó\ |B| = 0 \\ 1 + f(A_2 \ldots A_n, B_2 \ldots B_m) & si\ A_1 = B_1 \\ \text{máx}(f(A_2 \ldots A_n, B), f(A, B_2 \ldots B_m)) & si\ A_1 \neq B_1 \end{cases} \quad (1)$$

1.4. Solapamiento de subproblemas

Un subproblema en este contexto es un par de strings (A', B'). Si analizamos la función recursiva, entonces vemos que para cada subproblema (A', B') se puede necesitar resolver alguno de los tres subproblemas $(A'_2 \ldots A'_n, B'_2 \ldots B'_m), (A'_2 \ldots A'_n, B'), (A', B'_2 \ldots B'_m)$. En todos los casos, los subproblemas que se generan tienen sufijos de A' y B'. Es por esto que podemos afirmar que los únicos subproblemas que se van a llamar en la recursión son aquellos de la forma (A', B') con A' sufijo de A y B' sufijo de B.

Como A tiene $n+1$ sufijos y B tiene $m+1$ entonces a lo sumo hay $(n+1)(m+1)$ posibles subproblemas distintos que se puede tener que resolver. Sin embargo, la recursión en el peor caso es exponencial ya que puede hacer dos llamados recursivos en cada paso. Esto es un indicio de que hay solapamiento de subproblemas.

Identificar un sufijo de un string es sencillo y se puede hacer indicando desde qué posición arranca.
Por lo tanto cada subproblema lo podemos identificar con un par de números (i, j) con $1 \leq i \leq n+1, 1 \leq j \leq m+1$.
De este modo el sufijo 1 de A es $A_1 \ldots A_n$, el 2 es $A_2 \ldots A_n$ y el $(n+1)$-ésimo es el string vacío. Podemos reescribir f del siguiente modo.

$$f(A,B,i,j) = \begin{cases} 0 & si\ i = |A| + 1\ ó\ j = |B| + 1 \\ 1 + f(A,B,i+1,j+1) & si\ A_i = B_j \\ \text{máx}(f(A,B,i+1,j), f(A,B,i,j+1)) & si\ A_i \neq B_j \end{cases} \quad (2)$$

1.5. Memoización

Una manera de evitar repetir el cálculo es agregar un diccionario *Memoria* y para cada subproblema preguntar primero si ya se calculó su solución. Si no es así, computarla y guardarla.

---

function LCS(Memoria, A, B, i, j)
  if NoEstaGuardado(Memoria, i, j) **then**
    if $|A| = 0$ ó $|B| = 0$ **then**
      Guardar(Memoria, i, j, 0)
    else if $A_i = B_j$ **then**
      Guardar(Memoria, i, j, 1+LCS(Memoria, A, B, i+1, j+1))
    else if $A_i \neq B_j$ **then**
      Guardar(Memoria, i, j, Max(LCS(Memoria, A, B, i+1, j), LCS(Memoria, A, B, i, j+1)))

  devolver Obtener(Memoria, i, j)

De este modo nos aseguramos resolver a lo sumo una vez cada subproblema, y por lo tanto como tenemos $(n+1)(m+1)$ subproblemas, tenemos un algoritmo que en peor caso tarda $O(nm)$. Una aclaración es que este algoritmo tiene esa complejidad porque el diccionario *Memoria* se puede implementar con las operaciones *NoEstaGuardado*, *Guardar*, y *Obtener* en tiempo constante. Eso se logra usando una matriz como implementación, y en su posición (i, j) se guarda la solución al problema (i, j).

1.6. Orden de subproblemas

Decimos que un subproblema $(i', j') < (i, j)$ si necesitamos que el subproblema (i', j') esté resuelto para poder resolver (i, j). En nuestro ejemplo en particular dado un (i, j) al llamar a f tenemos cuatro opciones. Una opción es que (i, j) es un caso base $i = n + 1$ o $j = m + 1$, y en dicho caso no depende de ningún subproblema. Otra opción es que $A_i = B_j$, en dicho caso el problema $(i + 1, j + 1)$ tiene que estar resuelto para poder resolver (i, j). Por último, si $A_i \neq B_j$ entonces los subproblemas $(i+1, j)$ y $(i, j+1)$ deben estar resueltos de antemano.

Por lo tanto, si supiesemos que los problemas $(i + 1, j + 1), (i + 1, j), (i, j + 1)$ ya están resueltos (en caso de que $i \neq n + 1, j \neq m + 1$), podemos resolver (i, j) usando esos cálculos.

1.7. Algoritmo Bottom-Up

Primero podemos resolver los problemas $(n + 1, j)$, $(i, m + 1)$ que tienen valor 0. Después podemos ir resolviéndolos en orden decreciente de i, y luego j. De este modo, todos los subproblemas necesarios ya van a estar resueltos cuando se necesiten.

---

function LCS(A, B)
  $M \leftarrow Vacío()$
  $M_{n+1,j} \leftarrow 0, \forall j \in 1 \dots m + 1$
  $M_{i,m+1} \leftarrow 0, \forall i \in 1 \dots n + 1$
  for $i \leftarrow n \dots 1$ **do**
    for $j \leftarrow m \dots 1$ **do**
      if $A_i = B_j$ **then**
        $M_{ij} \leftarrow M_{i+1,j+1}$
      else if $A_i \neq B_j$ **then**
        $M_{ij} \leftarrow \text{máx}(M_{i+1,j}, M_{i,j+1})$

  devolver M_{11}
