# Taller 6

Carlos Barón - Andrés Cocunubo

07/03/2018

1. Punto 1 Subsecuencia creciente más larga

   (a) Descripción problema

   La subsecuencia creciente más larga es una secuencia (no necesariamente de elementos contiguos) que sigue la definición de una relación de orden parcial $<$.

   (b) Formalización

   Dada una secuencia $S$ de $n$ elementos, encontrar un subsecencia $R$ , que poseen una relacion de orden parcial.

   i. Entradas: Una secuencia $S$ de números.
   $S =< a_i, ..., a_j > |i < j \in \mathbb{N}$

   ii. Salidas: Una subsecuencia $R$ tal que:
   $R =< r_i, ..., r_j > \in S \wedge r_i < k < r_j$.

   (c) Algoritmos

**Algorithm 1** Subsecuencia creciente más larga

procedure LIS(S)
  let $M \in \mathbb{N}^{|S|x|S|} \leftarrow [1]$
  for $i \leftarrow 0$ to $|S|$ do
    for $j \leftarrow 0$ to $i$ do
      if $S[i] < S[i+1]$ then
        $M[i][j] \leftarrow M[i][j] + 1$
      end if
      $M[i+1][j] \leftarrow M[i][j]$
    end for
  end for
end procedure

**Algorithm 2** Backtracking Subsecuencia creciente más larga

```
procedure Backtracking(S)
    let R be an empty sequence
    j ← 0
    while j! = len(S) do
        if M[len(S)][j] > M[len(S)][j + 1] then
            insert S[j] into R
        end if
        j ← j + 1
    end while
    if S[len(S)] < S[j − 1] then
        insert S[j − 1] into R
    else
        insert S[j] into R
    end if
    return R
end procedure
```

(d) Análisis del diseño

Se partió del siguiente diseño recurrente para la resolción del problema:

$$LIS(i,j) = \begin{cases} 1 & si \quad i = j \\ max(LIS(i+1,k)+1) & si \quad S[i] < S[i+1] \\ max(LIS(i+1,k)) & si \quad S[i] > S[i+1] \end{cases}$$

La tabla de memoización $M$ se llena desde la última posición(esquina inferior derecha), y de arriba para abajo, dejando la solución al problema en la primera columna y última fila, una vez sabido esto se procede a armar la secuencia resultante para esto se toma la última fila y se crea un contador $j$ para moverse a través de las columnas, si el valor que se encuentra en la tabla $M[|S|][j]$ es mayor a $M[|S|][j+1]$ el valor actual en la secuencia se agrega a la respuesta, adicionalmente se tiene en cuenta si las últimas posiciones de la tabla son iguales, para luego comparar y decidir si los valores de la secuencia son mayores para agregar a la respuesta.

(e) Invariante

Para el algoritmo de bottom-up todos los valores que se encuentren en la parte inferior a la diagonal de la matriz de memoización $M$, indican el tamaño de la secuencia más larga hasta ese momento.

(f) Complejidad

- Para el algoritmo de bottom-up por inspección se determina que la complejidad del algoritmo es $O(|S|^2)$.
- En cuanto al algoritmo de backtracking puesto que tiene un ciclo while, no siempre se recorrerá en diagonal la matriz, por lo cual su complejidad vendría dada por $\Theta(|S|)$

2. Punto 2 Edición de Distancias

   (a) Problema

   Encontrar la menor diferencia entre dos secuencias de carácteres. La menor difrencia se define como la cantidad de ediciones, inserciones o eliminaciones para hacer que una cadena sea igual a otra.

   (b) Formalizacion

   Dadas dos secuencias X y Y, encontrar una secuencia S de comandos, donde S indique como convertir X en Y

      i. Entrada: Una secuencia $X$ de $n$ elementos $X = < x_n \in T >$ y una secuencia $Y$ de $m$ elementos $Y = < y_m \in T >$ donde $X \wedge Y$ pueden poseer una relación de equivalencia sobre una subsecuencia $T$ de $X \vee Y$
      $X =< x_0, ..., x_n > \wedge Y =< y_0, ..., y_m > | \exists (x_i = y_j \vee x_i = x_k \vee y_i = y_k)$

      ii. Salida: Una secuencia $S$ de $k$ elementos $S =< s_k \in C >$donde $s_k$ puede ser igual a
      $InsertCommand$ que posee un $y_j$ (elemento a insertar) y una posicion (posición donde insertar) $i$ en $X$
      $\vee$ $DeleteCommand$ que posee una posicion (posición donde eliminar) $i$ en $X$
      $\vee$ $ChangeCommand$ que posee un $y_j$ (elemento para cambio), un $x_i$ (elemento a cambiar) y una posicion (posición donde cambiar) $i$ en $X$

      $S =< s_0, ..., s_p > | s_i = InsertCommand \wedge s_i = DeleteCommand \wedge s_i = ChangeCommand$

   (c) Pseudocódigo

**Algorithm 3** Distancias de Edición

procedure ED($X$,$Y$)
    let $M \in \mathbb{N}^{|X|x|Y|} \leftarrow [0]$
    let $B \in \mathbb{N}^{|X|x|Y|} \leftarrow [0, 0]$
    for $i \leftarrow 1$ to $|X|$ do
        for $j \leftarrow 1$ to $|Y|$ do
            if $i = 0$ then
                $M[i][j]\leftarrow j$
                $B[i][j]\leftarrow [0, -1]$
            else if $j = 0$ then
                $M[i][j]\leftarrow i$
                $B[i][j]\leftarrow [-1, 0]$
            else
                det$\leftarrow M[i-1][j] + 1$
                ins$\leftarrow M[i][j-1] + 1$
                chg$\leftarrow M[i-1][j+1] + d(X, Y, i, i)$
                if $det < ins \wedge det < chg$
                    $M[i][j]\leftarrow det$
                    $B[i][j]\leftarrow [-1, 0]$
                else if $ins < det \wedge ins < chg$
                    $M[i][j]\leftarrow ins$
                    $B[i][j]\leftarrow [0, -1]$
                else
                    $M[i][j]\leftarrow chg$
                    $B[i][j]\leftarrow [-1, -1]$

**Algorithm 4** Bactracking distacias de edición

```
procedure ED_Backtracking(X,Y,B)
    let Output as an empty list
    i ← |X|, j ← |Y|
    while i ≥ 0 ∧ j ≥ 0
        C ← B[i][j]
        if X[i] ≠ Y[j]
            if C[0] = −1 ∧ C[1] = 0
                if i = j
                    insert "ChangeCommand" into Output
                else
                    insert "DeleteCommand" into Output
            else if C[0] = 0 ∧ C[1] = −1
                if i = j
                    insert "ChangeCommand" into Output
                else
                    insert "InsertCommand" into Output
            else if C[0] = −1 ∧ C[1] = −1
                insert "ChangeCommand" into Output
        i ← i + C[0], j ← j + C[1]
    return Output
```

(d) Análisis del diseño

Para solucionar el problema se partío del siguiente diseño recurrente:

$$ed(|X|,|Y|) = \begin{cases} |X| & si \quad i = 0 \\ |Y| & si \quad j = 0 \\ min([1 + ed(i-1,j)],[1 + ed(i,j-1)],[d(i,j) + ed(i-1,j-1)]) & sino \end{cases}$$

En seguida se prodeció a memoizar el problema con la tabla $M$, guardando cada resultado en una posición $[i, j]$. Para realizar el bottom-up, se identificó que para calcular cada elemento se necesitaba del valor en una posición menos en $i$, donde se encontraba el costo de eliminar, el valor de una posición menos en $j$, donde se encontraba el costo de insertar y el valor de una posición menos en $i$ y $j$, donde se encontraba el costo de cambiar, para asi calcular cual era el menor y guardarlo en la posicón actual. Con base en lo anterior, se pudo identificar que para construir la tabla, era necesario realizar un recorrido normal a la matriz, primero recorrer las filas y por cada fila recorrer las columnas.

(e) Invariante

La invariante del algoritmo de bottom-up es que cada elemento de la tabla en una posición $[i, j]$ representa el minimo numero de cambios que se tienen que hacer, hasta la posicion $i$ en $X$ y la posicion $j$ en $Y$ , para convertir $X$ en $Y$.

(f) Complejidad

- Por inspección de código la complejidad del algoritmo ED es $O(n * m)$, siendo $n$ el tamaño de la primera secuencia y $m$ el tamaño de la segunda secuencia, debido a que se realiza un recorrido completo y ordenado en cuanto a filas y columnas a un matriz de dimesiones $nm$.
- En cuanto al backtracking su complejidad es $\Theta(n)$ debido a que forzosamente se hace un recorrido sesgado a la diagonal de la matriz.