# Sub-secuencia Común Más Larga (LCS)

**Autores**:

*Pedro Miguel Lagüera Cabrera (alu0100891485@ull.edu.es)*
*Tarun Mohandas Daryanani (alu0100891782@ull.edu.es)*
*Abián Torres Torres (alu0100887686@ull.edu.es)*

# Introducción

El **Longest common subsequence problem** también conocido como *LCS problem*, se trata de encontrar una sub-secuencia más larga que es común en dos secuencias. Es diferente del problema de sub-string común más largo; a diferencia de los *sub-strings*, las *sub-secuencias* no necesitan tener posiciones consecutivas en la secuencia original.

Sus principales usos están en el campo de la bioinformática y también es usado ampliamente para los sistemas de control de revisión como Git para reconciliar múltiples cambios sobre archivos controlados de revisión.

El problema de LCS tiene una estructura **óptima**: el problema puede ser dividido en sub-problemas más pequeños y simples, el cual puede ser dividido en sub-problemas más simples y así hasta que la solución llega a ser trivial. Estas soluciones de sub-problemas triviales son utilizadas por problemas progresivamente mayores para obtener una solución final.

Los problemas con estas dos propiedades pueden ser resueltos por una técnica de programación llamada como **programación dinámica**, en la que las soluciones de los sub-problemas se almacenan en lugar de ser calculadas una y otra vez. Para esto se necesita la técnica de memorización en la cual se tabulan las soluciones de los sub-problemas más pequeños (como si se estuviera tomando una nota "memo" en inglés) para que esas soluciones puedan ser utilizadas para resolver a los de más alto nivel.

---

## Propiedades

Sean las secuencias:

- $X = \{ x_1, x_2, \dots , x_n \}$
- $Y = \{ y_1, y_2, \dots , y_m \}$

Y una sub-secuencia común más larga de X e Y:

- $Z = \{ z_1, z_2, \dots , z_k \}$

**Primera Propiedad**

Supongamos que las dos secuencias terminan con el mismo elemento. Para encontrar su LCS, podemos acortar las dos secuencias eliminando el último elemento, encontrando el LCS de las dos secuencias acortadas y agregando el elemento eliminado.

*Por ejemplo,* tenemos dos secuencias que tienen el mismo elemento final:

- **BANANA**
- **ATANA**

Eliminamos el último elemento en las dos cadenas. Repetimos hasta que lleguemos a un elemento no común. Los elementos removidos son (ANA).
Ahora las dos secuencias son:

- **BAN**
- **AT**

El LCS de las dos secuencias es (**A**).
Ahora anexamos los elementos eliminados al LCS y obtenemos (**AANA**), el cual es LCS de las secuencias originales. Entonces podemos decir que:

$$\textbf{Si } x_n = y_m, \textbf{ LCS}(X_n, Y_m) = \textbf{append}(\textbf{LCS}(X_{n-1}, Y_{m-1}), x_n)$$

**Propiedad 2**

Supongamos que las dos secuencias **X** y **Y no** terminan con el mismo símbolo. Entonces el LCS de X y Y es la más larga de las secuencias $LCS(X_n, Y_{m-1})$ y $LCS(X_{n-1}, Y_m)$.

Para entender esta propiedad, se considera lo siguiente:

- X: **ABCDEFG** (n elementos)
- Y: **BCDGK** (m elementos)

El LCS de estas dos secuencias podría terminar en G (último elemento de X) o puede que no.

**Caso 1: LCS termina con G**

Si termina con **G** no puede terminar con **K**. Por lo tanto, no pasa nada si nosotros eliminamos K de la secuencia Y. Si K estuviera en LCS, tendría que ser el último caracter por eso nosotros sabemos que no está en LCS. Entonces podemos decir que:

$$\textbf{LCS}(X_n, Y_m) = \textbf{LCS}(X_n, Y_{m-1})$$

**Caso 2: LCS NO termina con G**

Si **no** termina con **G**, no afectaría en nada eliminar G de la secuencia de X. Por la misma razón. Entonces podemos escribir:

$$\mathbf{LCS(X_n, Y_m) = LCS(X_{n-1}, Y_m)}$$

# Implementación

Para poder implementar el algoritmo de búsqueda de la sub-secuencia común más larga, podemos afrontar el problema desde tres perspectivas:

- Un algoritmo recursivo de fuerza bruta
- Un algoritmo recursivo con memoria (Top Down)
- Un algoritmo iterativo con memoria (Bottom up)

En todos los algoritmos se hace uso de dos índices: la i para la primera cadena y la j para la segunda cadena, de modo que podemos saber en qué posición de cada cadena estamos, así como el carácter a evaluar de cada cadena.

## Algoritmo de fuerza bruta

Es el algoritmo más sencillo de implementar por la facilidad a la hora de afrontarlo e implementarlo, pero es también el algoritmo menos eficiente y más lento. Esto es, es el que más recursos consume y además más tarda en solucionar el problema. Esto se debe a que resuelve cada problema varias veces, sin importar si ya lo habíamos resuelto anteriormente.

```
15  private static int subproblem(int i, int j) {
16    if(i == -1 || j == -1)
17      return 0;
18    if((i >= 0) && (j >= 0) && (getA().charAt(i) == getB().charAt(j)))
19      return subproblem(i - 1, j - 1) + 1;
20    else
21      return Math.max(subproblem(i - 1, j), subproblem(i, j - 1));
22  }
```

Podemos observar que el algoritmo tiene en cuenta dos casos:

- Si en la posición i j de las respectivas cadenas coinciden los caracteres, por lo que devolvemos el sub-problema de 1 + sub-problema (pero esta vez quitando ambos caracteres).
- Si no coinciden, se devuelve el máximo entre quitar un carácter u otro. Sin embargo, en ningún momento recordamos lo que hemos calculado, por lo que cada llamada se realiza de la misma manera sin tener un control.

- Es posible que ejecutemos la posición i-1, j; Retornemos y ahora ejecutemos con las posiciones i, j-1; En este caso podrían coincidir ambos caracteres y ejecutar sub-problemas con i-1, (j-1)-1; pero, i-1 ya había sido resuelto previamente, por lo que estamos volviendo a calcular un valor que ya se había calculado.

## Algoritmo recursivo con memoizacion: Top Down

En este algoritmo tratamos de buscar la solución también de manera recursiva, pero en este caso existen algunas diferencias:

- Al ser Top Down vamos de arriba hacia abajo en el árbol de recursividad, por lo que empezamos con los índices a 0 en vez de empezar por el final de cada cadena.
- Rellenamos una tabla inicial que nos servirá para no tener que solucionar un problema varias veces.

El resultado es un algoritmo parecido al anterior, pero que esta vez soluciona los sub-problemas estrictamente necesarios, que serán n x m sub-problemas.

La primera fase del algoritmo rellena la tabla de n x m elementos con un "-1" en cada posición. De este modo, si en algún momento consultamos una posición de la matriz y el valor es mayor que -1, significa que ya hemos resuelto ese problema, por lo que no debemos volver a solucionarlo. Esta comprobación se realiza en la línea 21, por lo que, si esta condición no se cumple, no lo volvemos a solucionar, sino que retornamos el valor.

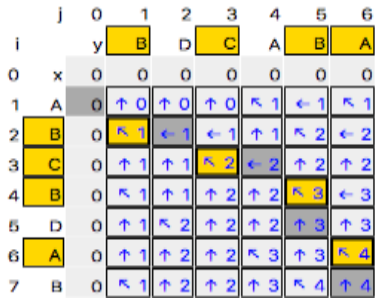
```
20  private static int subproblem(int i, int j) {
21    if(table.get(i, j) < 0){
22      if(i == a.length() || j == b.length())
23        table.set(i, j, 0);
24      else if(a.charAt(i) == b.charAt(j))
25        table.set(i, j, subproblem(i + 1, j + 1) + 1);
26      else
27        table.set(i, j, Math.max(subproblem(i + 1, j), subproblem(i, j + 1)));
28    }
29    return table.get(i, j);
30  }
```

- Si se nos presenta el caso de que debemos solucionar el sub-problema, actuamos de la misma manera que el algoritmo de fuerza bruta:
  - ❖ Si coinciden los caracteres, avanzamos los dos índices y sumamos 1 a la solución.
  - ❖ Si no coinciden los caracteres, devolvemos el máximo entre avanzar un índice u otro.
  - ❖ Cada vez que hacemos una llamada a un sub-problema volvemos a comprobar si ya lo hemos resuelto.

## Algoritmo iterativo con memoizacion: Bottom Up

El algoritmo bottom up es significativamente distinto a los dos anteriores, principalmente debido a que, en vez de ser recursivo, obtiene una solución de forma iterativa.
Además, otra diferencia es que la tabla se va rellenando dinámicamente. Esto significa que no tenemos un paso previo en que el rellenamos con un valor predeterminado, sino que vamos recorriendo ambas cadenas y revisando las coincidencias ("matchings") de caracteres. Así, sabemos que, a la hora de tratar de elaborar una solución, todos los problemas han sido previamente resueltos.
De igual modo, seguimos teniendo dos fases:

- Primera fase: durante esta fase rellenamos la tabla, mediante dos bucles (n, m).
  - Si existe un "matching", sumamos 1 en diagonal, es decir, cogemos el valor en i-1, j-1 y le sumamos 1.
  - Si no existe un "matching", se escoge el valor mayor entre el de la izquierda y el de encima, es decir, (i - 1, j) y (i, j - 1).
- Segunda fase: backtracking
  - Debemos comenzar en la esquina inferior derecha de la tabla creada e ir escalando hacia la esquina superior izquierda. Podemos realizar tres movimientos: izquierda, arriba y diagonal.
    - Primero comprobamos el movimiento hacia la izquierda, para lo que el número de la izquierda debe ser igual.
    - Luego comprobamos el movimiento hacia arriba, para lo que el número de la izquierda debe ser igual.
    - Por último, si no podemos realizar los dos movimientos anteriores, significa que en la primera fase sumamos en diagonal, por lo que hubo un "matching". De este modo, nos movemos en diagonal hacia arriba a la izquierda.
  - La condición de parada es detener el algoritmo en cuanto llegamos al principio de alguna de las cadenas, de modo que no quedan más caracteres que comparar con la otra cadena.

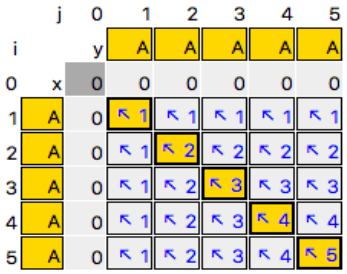


# Análisis: Algoritmo de fuerza bruta vs Bottom Up

La variante Bottom Up permite, de manera muy intuitiva, obtener la complejidad este, simplemente debemos llevar a cabo un análisis paso a paso del pseudocódigo descrito previamente.
Sabemos que, para cualquier par de cadenas, los dos bucles anidados situados en el método, ejecutará una única instrucción tantas veces como, la longitud de la primera cadena más uno, por la longitud de la segunda cadena más uno:

$$T(n) = (n+1) * (m+1)$$

Por otro lado, debemos contemplar el proceso de backtracking. En el peor caso, vamos a tener que recorrer la matriz tantas posiciones como unidades resultantes de sumar las longitudes de las dos cadenas. Sabemos que es así porque el algoritmo de reconstrucción de la solución solo contempla movimientos hacia arriba o izquierda. En cambio, si las dos cadenas son iguales, únicamente visitaremos la diagonal principal es su totalidad.



| | j | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|---|
| i | | y | A | A | A | A | A |
| 0 | x | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | A | 0 | ↖ 1 | ↖ 1 | ↖ 1 | ↖ 1 | ↖ 1 |
| 2 | A | 0 | ↖ 1 | ↖ 2 | ↖ 2 | ↖ 2 | ↖ 2 |
| 3 | A | 0 | ↖ 1 | ↖ 2 | ↖ 3 | ↖ 3 | ↖ 3 |
| 4 | A | 0 | ↖ 1 | ↖ 2 | ↖ 3 | ↖ 4 | ↖ 4 |
| 5 | A | 0 | ↖ 1 | ↖ 2 | ↖ 3 | ↖ 4 | ↖ 5 |

Por lo tanto, podemos afirmar que, la complejidad resultante de la variante Bottom Up estará acotada por:

$$\Omega((n * m) + (n))$$
$$O((n * m) + (n + m)))$$

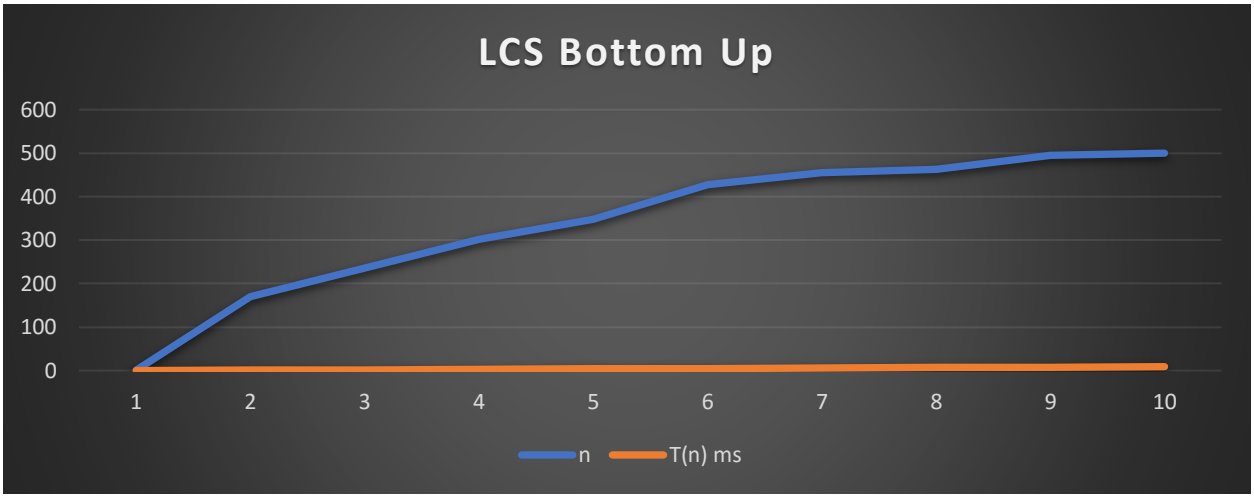
En cuanto a la variante recursiva sin memoization, deberemos analizar más detenidamente el pseudocódigo. Es evidente que, en el peor caso, las cadenas son totalmente distintas, por lo que se el algoritmo siempre ejecutará la instrucción correspondiente a la tercera condición:

```
else
  return Math.max(subproblem(i - 1, j), subproblem(i, j - 1));
```

Esto conlleva que se realicen **dos llamadas recursivas** en cada iteración del método en pila. Suponiendo que las dos cadenas son del **mismo tamaño** (n=m), podemos inferir que la complejidad, en cota superior, será de:

$$O(2^n)$$

Nótese que el tiempo, en el caso de la variante recursiva, excede notablemente el tiempo estimado para el algoritmo Bottom Up.
Llegados a este punto, comprobemos de manera experimental si es cierto lo que proponemos:




Como planteábamos, efectivamente, el algoritmo Bottom Up crece, "prácticamente", de forma **lineal**. Vemos como para una entrada de dos cadenas de tamaño n, el tiempo es de 10 ms, lo cual indica que el algoritmo para cadenas de tamaños elevados, funciona de manera eficiente.

En cuanto a la variante recursiva, inicialmente la relación (tamaño) / (tiempo de ejecución) se mantiene estable, prácticamente en un tiempo equivalente a 0. Pero a partir de valores superiores a 20, comenzamos a ver una breve subida, hasta llegar a tomar forma **exponencial**. Por lo tanto, en ese punto, vemos como los problemas re-calculados comienzan a ser un inconveniente para nuestro algoritmo.
En conclusión, la programación dinámica muchas veces puede ser nuestra salvación a la hora de diseñar algoritmos, pues puede hacer que, desde el punto de vista de la eficiencia, nuestros algoritmos den ese salto de salto de calidad que queremos.

# Bibliografía


1. Simulador LCS online, el cual puede ser de gran utilidad a la hora de testear los resultados de nuestras implementaciones.
2. Recurso interesante acerca de la implementación del algoritmo LCS en todas sus variantes, así como su complejidad.
3. Otro recurso complementario sobre LCS proveniente de un artículo de Wikipedia.