Juan Sebastián Loaiza Ospina
B74200
Fecha: 14/11/2025

1) Investigación del problema

    a) El **problema de la Subsecuencia Común Más Larga (LCS)** consiste en encontrar la *secuencia de caracteres más larga* que aparece en **el mismo orden** en dos cadenas dadas, en este caso de ADN, aunque no necesariamente de forma continua.

    Suponiendo que existen dos cadenas **ADN1** y **ADN2**, se busca encontrar una cadena **ADNx** que posea las características:

    1) Estar formada únicamente por caracteres que aparecen tanto en **ADN1** como en **ADN2**.
    2) Mantiene el mismo orden relativo en ambas cadenas.
    3) Mayor longitud posible siguiendo las primeras dos reglas.

NOTA: Una **subsecuencia NO** es lo mismo que un **substring**. Los caracteres de la subsecuencia no tienen que aparecer seguidos uno tras otro, los caracteres de un hipotético **ADN2** pueden tener otros caracteres de por medio mientras conserven el orden de la cadena en **ADN1**.

2) ¿Por qué la fuerza bruta NO sirve como estrategia?

    La solución por fuerza bruta consiste en:

    1) Generar todas las subsecuencias posibles de **AND1**.
    2) Para cada subsecuencia posible de **ADN1**, revisar si es una subsecuencia en **ADN2**.
    3) Guardar la cadena más larga que cumpla los requisitos previos.

    El problema que encontramos con esta técnica es que la cantidad de subsecuencias que podríamos encontrar en una hipotética cadena **ADN1** de longitud **N** es $\mathbf{2^n}$**.**

NOTA: La complejidad **temporal** de esto es $\mathbf{O(N \cdot 2^n)}$ porque para cada una de las $\mathbf{2^n}$ subsecuencias generadas es necesario verificar si aparece dentro de **ADN2** y puede tomar hasta **O(N)** en el peor de los casos. En el caso de la complejidad **espacial** sería $\mathbf{O(N \cdot 2^n)}$ porque es la cantidad de caracteres en el peor de los casos y como mínimo habían $\mathbf{2^n}$ secuencias almacenadas en memoria.

3) Algoritmo eficiente (programación dinámica para LCS)

    Para usar el algoritmo de Programación Dinámica para LCS se construye una matriz donde cada posición indica la longitud de la mejor subsecuencia común entre los prefijos de las dos cadenas, y la llena comparando caracteres y usando los resultados ya calculados, permitiendo obtener la subsecuencia más larga en tiempo $\mathbf{O(n \cdot m)}$. Además, su estructura permite **paralelizar por**

diagonales, lo cual es ideal para usar MPI que es requerido por el proyecto. Esto utiliza el método wavefront, que consiste en:

a) Primero se calcula la diagonal 1 (independiente).
b) Luego la diagonal 2 (que solo necesita la 1).
c) Luego la diagonal 3 (que solo necesita la 2).
d) Y así sucesivamente con las demás diagonales.

4) Pseudocódigo pensado para resolver el problema y posteriormente paralelizar con MPI

NOTA: S1 y S2 representan Substring 1 y Substring 2.

```
wavefrontLCS( variables S1, S2 ) {

   n = longitud(S1)
   m = longitud(S2)

   Crear matriz DP[n+1][m+1] llena de ceros

   // Recorrer la matriz por diagonales wavefront
   // Cada diagonal k cumple que: i + j = k
   // Esto permitirá en la versión paralela dividir cada diagonal entre varios procesos de MPI

   Para k desde 1 hasta (n + m - 1) hacer {

      // Cálculo del rango válido de i para esta diagonal
      i_min = max(1, k - m)
      i_max = min(n, k - 1)

      // Aquí, en paralelo, cada proceso podría tomar una parte del rango
      // por ejemplo: proceso 0 primeras celdas, proceso 1 las siguientes, etc.
      Para i desde i_min hasta i_max {

         j =  k - i  // porque deben sumar k

         Si (S1[i-1] == S2[j-1]) {
            // Si los caracteres son iguales, se extiende la subsecuencia
            DP[i][j] = DP[i-1][j-1] + 1

         Sino
            // Si son distintos, pasar el máximo del de arriba o el izquierdo
            DP[i][j] =  max( DP[i-1][j], DP[i][j-1] )
         }
```

```
      }

       // En la versión MPI aquí hay que poner sincronización
       // Algo tipo una barrera
       // MPI_Barrier()??

   }

   Retornar cadena de caracteres
}
```