Universidad del Valle de Guatemala
Análisis y Diseño de Algoritmos
Sección 10

Proyecto #2

21016 Javier Chavez
21032 Marco Ramírez
21108 Brian Carrillo
21116 Josué Morales

Guatemala, 04 de abril de 2024

Investigación

1. Problema elegido

Longest Common Subsequence

Dados dos strings S1 y S2 de tamaño m y n respectivamente, el problema consiste en encontrar la longitud de la de la subsecuencia más larga, presente en los dos strings (Guatam, s.f.).

Según Aho y demás contribuidores del libro *Compilers: Principles, Techniques & Tools*, una subsecuencia de un string S es cualquier string formado por eliminar cero o más (no necesariamente consecutivas) posiciones de S. Por ejemplo, **baan** es una subsecuencia de **banana**, tras haber eliminado las posiciones 2 y 5.

Ejemplos

1)
S1 = “AGGTAB”, S2 = “GXTXAYB”
Respuesta: 4

La subsecuencia más larga en común es “GTAB”, y su longitud es 4.

2)
S1 = “BD”, S2 = “ABCD”
Respuesta: 2

La subsecuencia más larga en común es “BD”, y su longitud es 2.

2. Algoritmos de solución

Algoritmo DaC

Pseudocódigo enfoque recursivo

```
i = len(A)
j = len(B)
int LCS(A,B,i,j) {
        if (i==0 | j==0 )
                return 0;
        else if (A[i-1]==B[j-1])
                return 1+LCS(A,B,i-1,j-1);
        else
                return max(LCS(A,B,i-1,j),LCS(A,B,i,j-1));
}
```

Explicación

El algoritmo con enfoque recursivo se basa en los siguientes pasos:

1. Crear una función recursiva LCS.
2. Verificar la relación entre los últimos caracteres de las cadenas.
3. Llamada recursiva según sea la relación anterior.
    a. Si los últimos caracteres de ambas cadenas son iguales, el carácter forma parte de la LCS, y por lo tanto la longitud de la subsecuencia será 1 más el resultado de la llamada recursiva a la misma función utilizando el resto de ambas cadenas (es decir, las cadenas sin el último caracter).

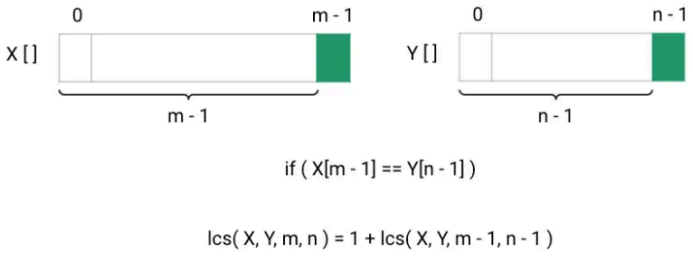

Figura 1. Caso 1 (Gautam, 2021).

    b. Si los últimos caracteres no son iguales, entonces existen dos formas de dividir el problema:

        - Ignorar el último caracter de la primera cadena y llamar a la función recursiva utilizando la primera cadena sin su último caracter y la segunda cadena de manera completa.

        - Ignorar el último caracter de la segunda cadena y llamar a la función recursiva utilizando la primera cadena completa y la segunda cadena sin su último caracter.

        La combinación de estas soluciones se da al elegir el mayor resultado de estas dos llamadas.

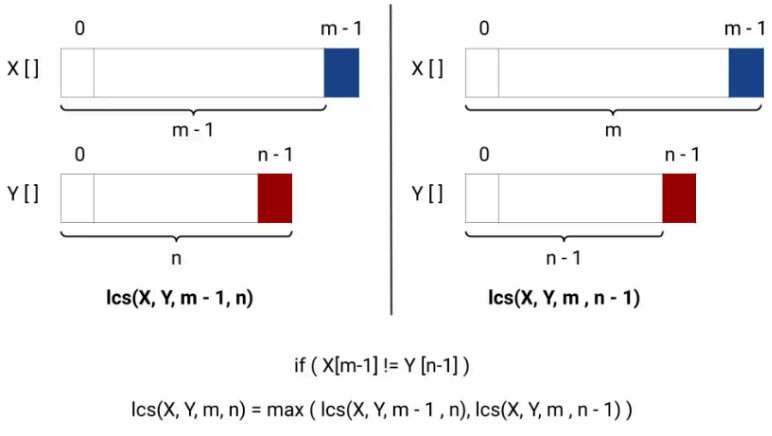

Figura 2. Caso 2 (Gautam, 2021).

    c. Si alguna de las posiciones de los caracteres es igual a cero, quiere decir que se ha recorrido toda la cadena, y por lo tanto la longitud a retornar es 0. Este caso es considerado el caso base de la recursión.
4. Retornar la longitud de la LCS (GeeksForGeeks, 2024).

Código en python

```
def lcs(A, B, i, j):
   if i == 0 or j == 0:
      return 0
   elif A[i-1] == B[j-1]:
      return 1 + lcs(A, B, i-1, j-1)
   else:
      return max(lcs(A, B, i-1, j), lcs(A, B, i, j-1))


# Main
if __name__ == '__main__':
   A = "AGGTAB"
   B = "GXTXAYB"
   print("Longitud de LCS: ", lcs(A, B, len(A), len(B)))
```

Algoritmo de Programación Dinámica

Pseudocódigo enfoque Top-Down con meoización

```
m = len(A)
n = len(B)
dp = [m+1][n+1]

for i=0, i<m+1, i++
        for j=0, j<n+1, j++
                dp[i][j] = -1
int LCS(A,Bi,j) {
        if (i==0 | j==0 )
                return 0;
        else if (dp[i][j] != -1)
                return dp[i][j];
        else if (A[i-1]==B[j-1])
                dp[i][j] = 1+LCS(A,B,i-1,j-1,dp);
                return dp[i][j];
        else
                dp[i][j] = max(LCS(A,B,i–1,j,dp),LCS(A,B,i,j-1,dp));
                return dp[i][j];
}
```

Explicación

Debido a que el algoritmo DaC, planteado anteriormente, presenta estas propiedades:

1. Subestructura óptima

   Por contradicción

   Suponiendo dos secuencias X e Y

   - X de longitud m tal que X = x1,x2, … ,xm
   - Y de longitud n tal que Y = y1,y2, … ,yn

     Sea Z = z1,z2, … ,zk la LCS de X e Y con longitud k.

   - X' es una subsecuencia de X
   - Y' es una subsecuencia de Y

     Sea Z' la LCS de X' e Y' con longitud k'>k.

   Existe una contradicción, puesto que Z' también sería la LCS de X y Y.

2. Subproblemas traslapados: Puesto que el algoritmo se basa en explorar todas las posibles subsecuencias en ambas cadenas utilizando recursión y comparando el último caracter, muchos subproblemas se están resolviendo más de una vez. Esto se puede visualizar de mejor manera en el árbol de recursión.

   Árbol de recursión

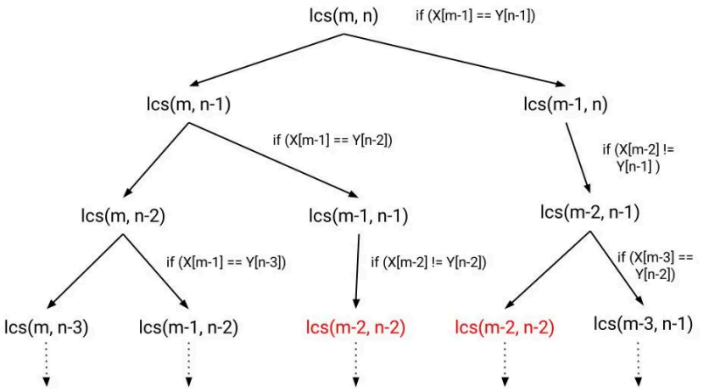

Figura 3. Recursion Tree (Gautam, 2021).

Como se muestra en este ejemplo arbitrario, subproblemas como lcs(m-2,n-2) se repiten según sean los casos desarrollados para el problema general.

Es posible dar un enfoque más eficiente al algoritmo DaC a través de la meoización. Para esto, se crea una matriz en la que se almacenarán los resultados obtenidos según qué posiciones de caracteres sean evaluados. Al momento de volver a necesitar de los resultados ya calculados, el valor de retorno será el que ya se encuentra almacenado, en lugar de calcularlo nuevamente. La matriz a construir debe ser de tamaño (m+1)(n+1), puesto que esta es la cantidad total de subproblemas, considerando el caso en el que se han terminado todos los caracteres de alguna o ambas cadenas (Gautam, 2021).

Código en python

```
def lcs(A, B, i, j, dp):
    if (i == 0 or j == 0):
        return 0
    elif (dp[i][j] != -1):
        return dp[i][j]
    elif A[i-1] == B[j-1]:
        dp[i][j] = 1 + lcs(A, B, i-1, j-1, dp)
        return dp[i][j]
    else:
        dp[i][j] = max(lcs(A, B, i-1, j, dp), lcs(A, B, i, j-1, dp))
        return dp[i][j]
# Main
if __name__ == '__main__':
    A = "AGGTAB"
    B = "GXTXAYB"

    m = len(A)
    n = len(B)
    dp = [[-1 for i in range(n+1)]for j in range(m+1)]

    print("Longitud de LCS: ", lcs(A, B, m, n, dp))
```

3. Análisis teórico

a. Algoritmo DaC

Relación de recurrencia

$$T(i,j) = \begin{cases} O(1)\ si\ i = 0\ o\ j = 0 \\ 1 + T(i-1, j-1)\ si\ i,j > 0\ y\ A[i-1] = B[j-1] \\ max(T(i-1,j), T(i,j-1))\ si\ i,j > 0\ y\ A[i-1] \neq B[j-1] \end{cases}$$

Debido a la naturaleza de la relación de recurrencia, los métodos factibles a utilizar son el método de substitución y el árbol de recursión, siendo este último el aplicado en este proyecto. Por motivos de simplificación y generalización, se construyó un árbol en base a dos cadenas de tamaño n y m respectivamente, en las que ninguno de sus caracteres coincide, por lo que la longitud de la subsecuencias más larga es de 0. Este también es conocido como el peor caso, puesto que únicamente se hace uso del caso base y del caso 3 en el que se realizan dos llamadas recursivas con i-1 y j-1 respectivamente.

Árbol de recursión del peor caso para dos cadenas de tamaño n y m

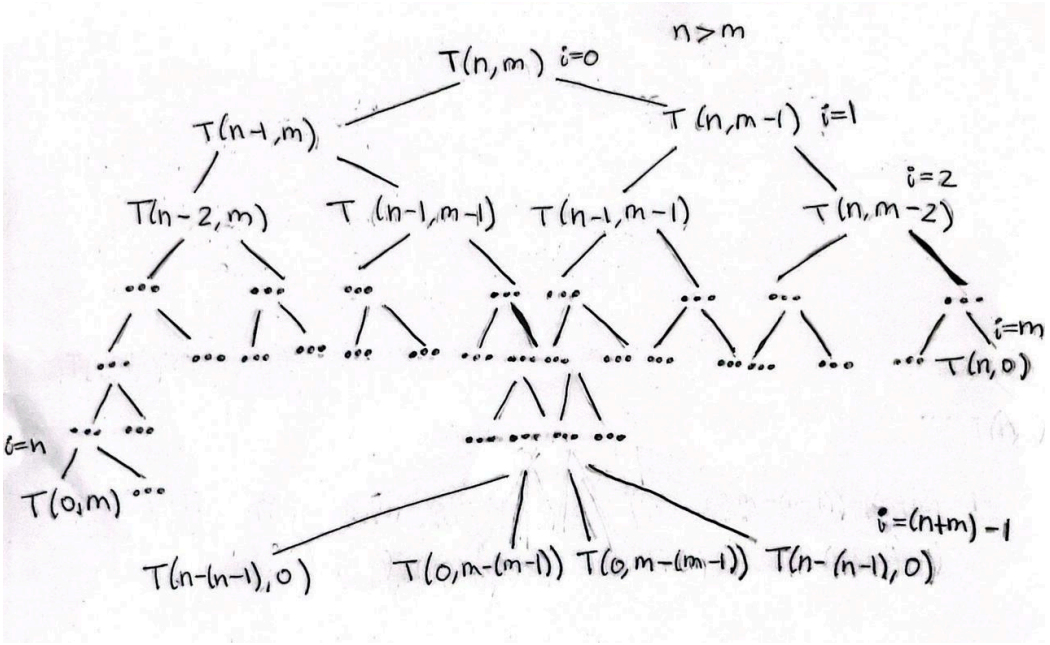

Figura 4. Árbol de recursión *Worst Case Scenario*.

A partir de este árbol se logran apreciar características importantes de este problema.

a. No todas las ramas poseen la misma altura, algunas alcanzan el caso base más rápido, por lo que no existe una altura general del árbol, sino diferentes alturas según las ramas obtenidas.
b. Si empezamos con T(m,n) y sólo disminuimos i en cada paso, después de n pasos, llegamos a T(0,m), el cual es un caso base.
c. Si empezamos con T(m,n) y sólo disminuimos j en cada paso, después de m pasos, llegamos a T(n,0), el cual también es un caso base.
d. Muchos casos se repiten, por lo que el problema es potencialmente mejorable con un enfoque con meoización.

Por lo tanto, el peor camino que podríamos tomar desde la raíz T(n,m) hasta un caso base, en términos de llamadas recursivas, sería el camino que consiste en disminuir i en n pasos y disminuir j en m pasos. Es decir que el peor camino posee una altura de n+m.

Para cada nivel l del árbol hay dos llamadas recursivas, por lo que la cantidad de llamadas recursivas en el nivel l es 2^l, a excepción de los casos bases que no continúan bifurcándose. Para calcular el número total de llamadas, se tendría que sumar sobre todos los niveles del árbol, lo cual no incluiría todas las llamadas debido a la característica anteriormente mencionada, en la algunas ramas terminan antes de llegar a un caso base. Sin embargo, es posible acotar el tiempo de ejecución del problema según la altura máxima encontrada según el análisis realizado. En este caso, el número total de llamadas es menor que 2^{n+m}, por lo que el tiempo de ejecución expresado en notación asintótica es el siguiente:

$$T(n,m) = O(2^{n+m})$$

Podemos concluir que este problema crece exponencialmente.

b. Algoritmo Top-Down

El algoritmo de programación dinámica para LCS crear una matriz M de tamaño $(n + 1)(m + 1)$. La adición de 1 en filas y columnas se realiza para el manejo de cadenas vacías. Esta matriz es utilizada para almacenar los resultados de los subproblemas LCS para diferentes combinaciones de prefijos de las cadenas. Para analizar la tasa de crecimiento de este enfoque se listan los siguientes pasos:

1. Inicializar la matriz M con -1 en todas las posiciones.
2. Aplicación de la relación de recurrencia: Si el resultado para la llamada recursiva i,j no se encuentra registrado en la matriz, se calcula. En caso contrario, únicamente se obtiene el valor de dicha matriz. A diferencia del enfoque bottom up en el que se llena toda la matriz de antemano, esta se llena únicamente cuando se necesita.

En el peor de los casos, se visitarán $n \times m$ celdas. Si la recursión termina antes, es posible que no se necesite llenar toda la matriz, como en el caso de cadenas con una cantidad significativa de coincidencias. Por lo tanto, el tiempo de ejecución del enfoque top-down puede ser expresado en notación asintótica de la siguiente manera:

$$T(n, m) = O(n \times m)$$

Podemos concluir que el algoritmo Top-Down mejora considerablemente el tiempo de ejecución del enfoque recursivo.

4. Análisis empírico

a. Listado de entradas de prueba

Cuadro 1. Cadenas sin repetición de caracteres.

Cadenas sin repetición de caracteres	
Cadena A	**Cadena B**
Y	E
LR	BY
MXR	VYC
QXGE	SDFM
KFPDZ	WXQTJ
BMEGLU	SJPIWV
SDAYKCI	WVENPHO
TFNYBSUA	VJLDXQIR
CSFDNMROZ	LUWVPHXQJ
LWVBZFIXQO	JHKDRGPTCS
XEOVCBQNZGS	WTUDYRIPHMA
WBAMKJCQEPNF	TZGVYOUIDLRH
AJSGODIHXFYNE	LZCUTWBRVQPKM
INFCAMBORQZEVT	KUHXPYDSJWGLab
ASKXDIYGRLPQFVB	JZHNOUCMEWTabcd

Cuadro 2. Cadenas con repetición de caracteres.

Cadenas con repetición de caracteres	
Cadena A	**Cadena B**
ABCDEF	BCDEFG
HELLO	WORLD
CAT	DOG
PYTH	JAVA
OPE	GPT
ALGORITHM	DATASTRUC
BANANA	ANANAS
APPLEABCD	PINEAPPLE
BLUEE	GREEN
STACK	QUEUE
ELEPHANT	HIPPOPOT
WATE	FIRE
ROBOT	SMART
SOON	MOON
FLOWER	GARDEN
BEACH	SEASS
RIVER	STRIE
MOUNTA	VALLEY
CICLE	BIKES
CLOUD	SKYES
FISH	BIRD
BOOK	PEEK
HOUSE	HOMES
KEYS	LOCK
PIZZAS	BURGER
COFFEE	APPLES
CHOCOLATES	VANILLASSS
SOONEST	NOODLES
TRAIN	PLANE
SHOES	SOCKS

b. Gráficas de los tiempos de ejecución de cada algoritmo en función de las entradas de prueba.

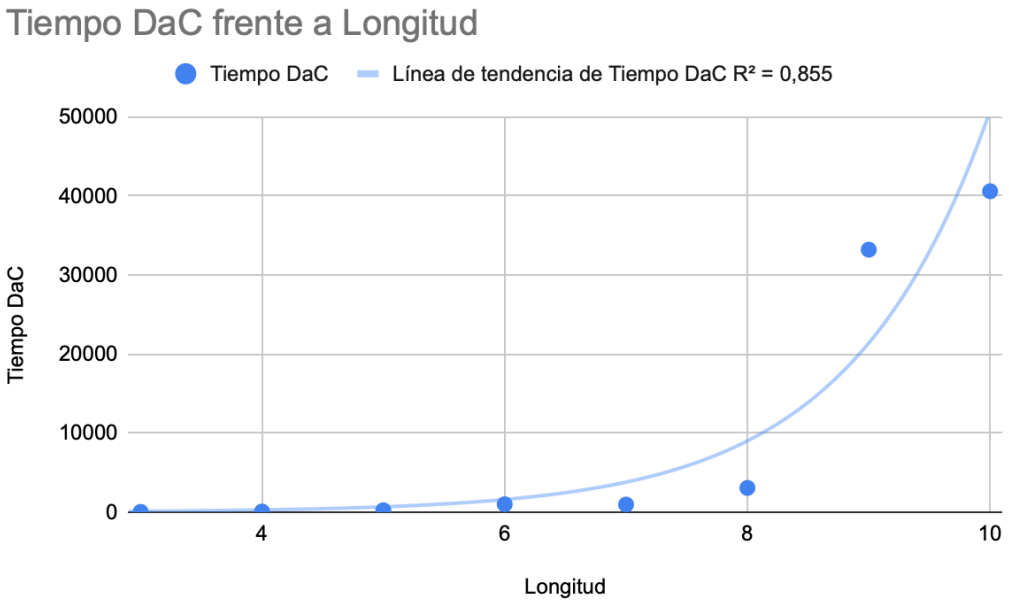

Figura 5. Tiempo DaC vs Longitud de cadenas con repetición.

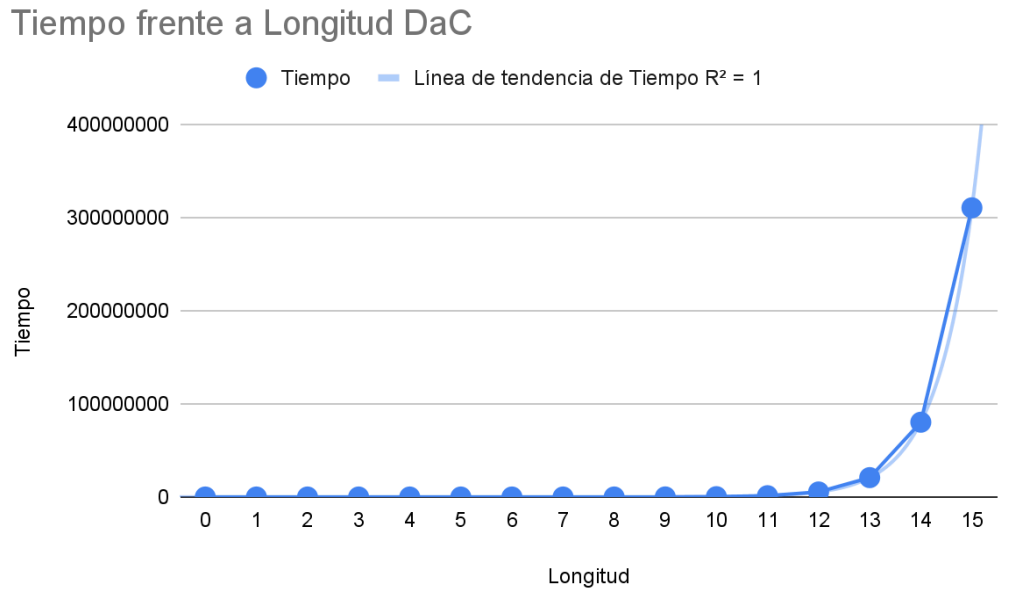

Figura 6. Tiempo DaC vs Longitud de cadenas sin repetición.

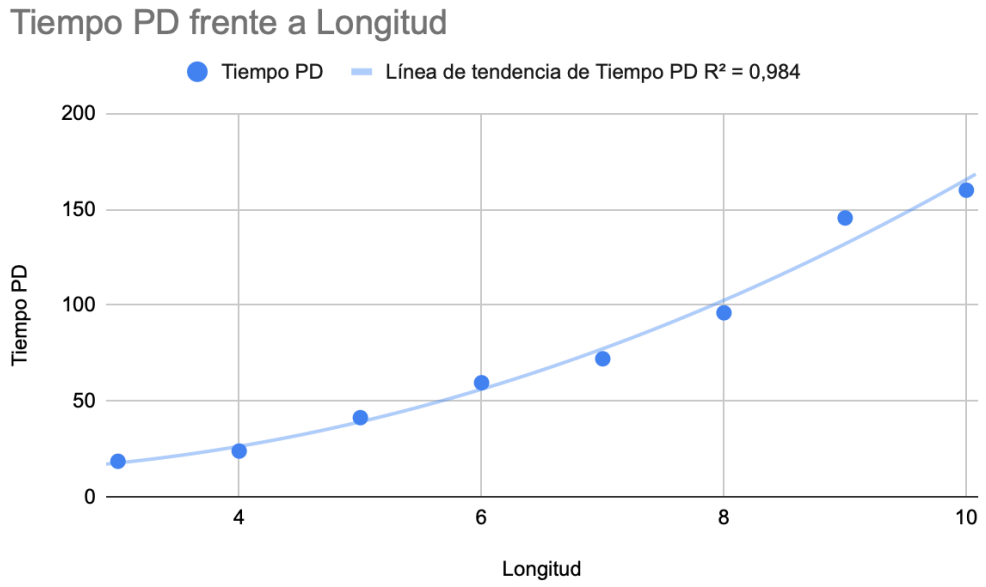

Figura 7. Tiempo PD vs Longitud de cadenas con repetición.

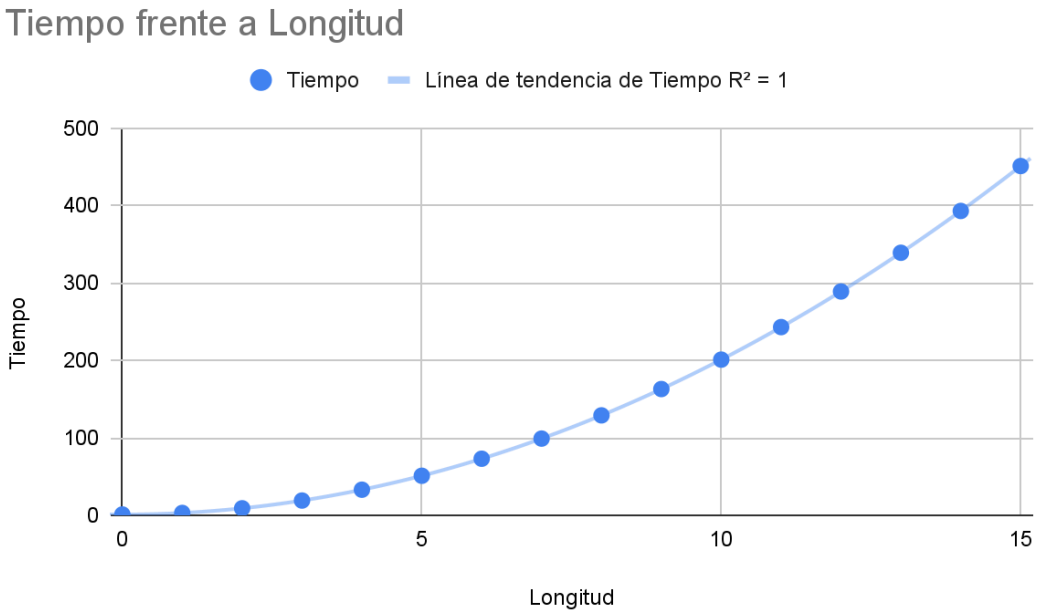

Figura 8. Tiempo PD vs Longitud de cadenas sin repetición.

c. Resultados

Análisis del enfoque DaC:

- **Tiempo de Ejecución Exponencial:** Como se anticipó en el análisis teórico, el enfoque DaC muestra un crecimiento exponencial en el tiempo de ejecución, particularmente evidente en casos con cadenas más largas por ejemplo "ALGORITHM" vs. "DATASTRUC". Esto se debe a la naturaleza recursiva del algoritmo, que genera una gran cantidad de subproblemas repetidos, especialmente en casos donde las cadenas no tienen muchos caracteres en común.
- **Casos Base Rápidos:** Para pares con poca o ninguna coincidencia por ejemplo., "CAT" vs. "DOG", el algoritmo llega rápidamente a los casos base, lo que resulta en tiempos de ejecución más cortos. Sin embargo, incluso en estos casos, la eficiencia es considerablemente menor en comparación con el enfoque PD.

Análisis del enfoque PD:

- **Tiempo de Ejecución Polinomial:** Los resultados empíricos confirman el análisis teórico que predice un crecimiento polinomial para el enfoque de PD. Esto es evidente en cómo el tiempo de ejecución aumenta de manera más controlada y predecible a medida que se incrementa la longitud de las cadenas. Incluso para pares con longitudes significativas por ejemplo, "CHOCOLATES" vs. "VANILLASSS".
- **Eficiencia en la Memorización:** La mejora significativa en los tiempos de ejecución para el enfoque PD se debe al uso de memorización, que evita cálculos redundantes almacenando los resultados de los subproblemas. Esto reduce drásticamente el número de operaciones necesarias para encontrar la LCS.

Comparación y discrepancias:

- **Confirmación de Análisis Teóricos:** Los datos empíricos respaldan firmemente los análisis teóricos, demostrando el tiempo de ejecución exponencial del enfoque DaC y el polinomial del enfoque PD.
- **Discrepancias:** Se observa mayor discrepancia en los casos en los que las cadenas poseen una o más coincidencias entre sí. Esto es normal considerando que el análisis teórico se hizo en base al peor caso.
- **Importancia de la Elección del Algoritmo:** La elección del algoritmo adecuado puede tener un impacto dramático en la eficiencia, especialmente a medida que aumenta el tamaño de las entradas. Los resultados subrayan la importancia de la programación dinámica para problemas con subestructuras óptimas y subproblemas traslapados.

Referencias

Bari, A. (2018). 4.9 Longest Common Subsequence (LCS) - Recursion and Dynamic Programming. Obtenido de https://www.youtube.com/watch?v=sSno9rV8Rhg

Gautam, S. (2021). Longest Common Subsequence. Obtenido de https://www.enjoyalgorithms.com/blog/longest-common-subsequence

GeeksForGeeks. (2024). Longest Common Subsequence (LCS). Obtenido de https://www.geeksforgeeks.org/longest-common-subsequence-dp-4/

Aho, A., Lam, M., Sethi, R. & Ullman, J. (2007). Compilers: Principles, Techniques & Tools. Second Edition.