Análisis y Diseño de Algoritmos
Proyecto 2

Samuel A. Chamalé, Alejandro J. Ortega

Guatemala, marzo 2024

1. Problema Elegido

1.a. *Definición y enunciado del problema*

La Distancia de Levenshtein es una métrica ampliamente utilizada a nivel mundial para evaluar la similitud entre secuencias, especialmente en aplicaciones de lingüística computacional.

El problema de Levenshtein, también conocido como Edit Distance, consiste en determinar el número mínimo de operaciones de inserción, eliminación y reemplazo necesarias para transformar una secuencia en otra y lograr que sean idénticas.

Existen varios tipos de distancias de edición que permiten diferentes tipos de operaciones sobre las secuencias, como Hamming Distance, Jaro Distance o Longest Common Subsequence. Sin embargo, la medida más comúnmente utilizada es la Distancia de Levenshtein, por lo que a menudo se utiliza como sinónimo de distancia de edición.

En nuestro caso, nos interesa calcular la Distancia de Edición entre dos cadenas (strings), S_1 y S_2. Esta distancia se define como el número mínimo de mutaciones necesarias para transformar S_1 en S_2, donde una mutación puede ser cualquiera de las siguientes operaciones:

1. Cambiar una letra
2. Insertar una letra
3. Eliminar una letra

Consideremos dos cadenas S_1 y S_2, de tamaño m y n respectivamente. Para transformar S_1 en S_2, procedemos aplicando una secuencia de operaciones que se almacenan en una nueva cadena S_3. La transformación se realiza utilizando índices i y j para recorrer las cadenas S_1 y S_2, comenzando con $i = j = 1$. Las operaciones disponibles son las siguientes:

1. Cambiar una letra (Replace): Esta operación consiste en reemplazar un carácter en S_1 por un carácter c específico. En la cadena resultante S_3, se tiene $S_3[j] = c$ y se incrementa tanto i como j en 1 para avanzar a los siguientes caracteres en S_1 y S_2.
2. Insertar una letra (Insert): Esta operación implica insertar un carácter c en S_3. En S_3, se tiene $S_3[j] = c$, incrementando únicamente j en 1 sin modificar i.
3. Eliminar una letra (Delete): Esta operación consiste en ignorar un carácter en S_1, incrementando i en 1 sin modificar j.

Para ilustrar esto con un ejemplo, supongamos que tenemos las cadenas S_1 =çasaz S_2=çalma". La distancia de edición mínima entre estas dos cadenas sería 2, ya que podemos transformar S_1 en S_2 realizando una operación de sustitución para cambiar la "s"por "lz una operación de inserción para agregar la letra "m".

Es importante tener en cuenta que en nuestro enfoque asumimos que los costos de las operaciones de edición son constantes y no funciones. Es razonable asumir que los costos de las operaciones de edición

son constantes, es decir, que todas las operaciones tienen el mismo costo unitario. Esta simplificación se justifica por su simplicidad y practicidad en muchos casos de aplicación.

Al considerar costos constantes, el problema de la distancia de edición se vuelve más manejable y fácil de implementar, ya que no es necesario asignar valores específicos a cada operación de edición. Además, permite un enfoque más intuitivo al considerar que todas las operaciones tienen la misma importancia y contribuyen igualmente al resultado final.

Si bien existen variantes de la distancia de edición que asignan diferentes costos a cada operación, asumir costos constantes es una aproximación válida en muchas situaciones, especialmente cuando se busca una medida general de similitud entre secuencias. Esta simplificación no invalida el concepto de distancia de edición ni su utilidad en la comparación de cadenas, ya que sigue capturando la cantidad mínima de operaciones necesarias para transformar una secuencia en otra, independientemente del valor específico asignado a cada operación.

2. Algoritmos de solución

2.a. *Algoritmo DaC*

El enfoque de Divide and Conquer consiste en dividir el problema general en subproblemas más pequeños, resolverlos estos subproblemas recursivamente y luego combinar las soluciones de todos los subproblemas para obtener una solución al problema general.

Para resolver el problema de la distancia de edición (*Edit Distance*) utilizando Divide and Conquer, podemos seguir estos pasos:

Dividir

Podemos dividir las cadenas de entrada en dos mitades seleccionando un índice pivote. Esto nos permite dividir el problema en subproblemas más pequeños. Continuaremos dividiendo hasta que lleguemos a caracteres individuales o cadenas vacías.

Por ejemplo, si tenemos las cadenas “ABCD” y “ACD”, podemos elegir el índice pivote como el índice medio, lo que resulta en las subcadenas “ABC” y “ACD”.

Conquistar

Una vez que hemos dividido las cadenas, encontramos recursivamente la distancia de edición entre cada par de subcadenas. Esto implica resolver el problema de la distancia de edición para entradas más pequeñas hasta que lleguemos al caso base.

Para encontrar la distancia de edición entre dos subcadenas, consideramos tres posibles operaciones:

- Si los caracteres en las posiciones actuales coinciden, avanzamos a las siguientes posiciones sin realizar operaciones adicionales.
- Si los caracteres en las posiciones actuales no coinciden, tenemos tres opciones: inserción, eliminación o sustitución. Realizamos cada operación y calculamos la distancia de edición para las subcadenas resultantes.
- Elegimos la operación que produzca la mínima distancia de edición y continuamos recursivamente hasta que lleguemos al caso base.

Combinar

Después de resolver los subproblemas y obtener la distancia de edición para cada par de subcadenas, fusionamos los resultados para obtener la distancia de edición final para las cadenas originales. Combinamos las distancias de edición de las subcadenas según las operaciones realizadas.

Pseudocódigo

```
Edit Distance Divide And Conquer(x, y):
    Si longitud(x) = 0:
        Devolver length(y)
    Fin Si
    Si longitud(y) = 0:
        Devolver length(x)
    Fin Si
    Si x[0] = y[0]:
        Devolver Edit Distance Divide And Conquer(x[1:], y[1:])
    Fin Si
    Devolver 1 + min(
        Edit Distance Divide And Conquer(x, y[1:]),
        Edit Distance Divide And Conquer(x[1:], y),
        Edit Distance Divide And Conquer(x[1:], y[1:])
    )
Fin Algoritmo
```

2.b. *Programa que implemente el algoritmo DaC*

El programa que implementa el algoritmo con enfoque divide and conquer se encuentra en el siguiente enlace: `https://github.com/OrtegaRehbach/Proyecto02ADA/blob/main/editDistanceDAC.py`

2.c. *Algoritmo de DP*

Al utilizar el enfoque *bottom-up* en la resolución de problemas con programación dinámica, es necesario resolver subproblemas más pequeños antes de abordar los problemas más grandes. Este enfoque se basa en la idea de que la solución óptima para un problema se puede construir a partir de las soluciones óptimas de sus subproblemas más pequeños.

El primer paso para resolver un problema utilizando este enfoque consiste en identificar una subestructura óptima. Esto implica encontrar las decisiones necesarias para producir subproblemas. En el caso del problema *Edit Distance*, es necesario definir los índices i y j sobre X y Y para formar subcadenas y abordar los subproblemas.

Consideremos una secuencia de operaciones $(o_1, \ldots, o_k)$ que se considera óptima. Después de aplicar una de estas operaciones, podemos afirmar que se debe haber aplicado una secuencia de operaciones óptima para transformar las subcadenas de X y Y.

Si la granularidad mínima son los caracteres individuales, el término *'formulación de subcadenas'* puede resultar ambiguo. Específicamente, para generar los subproblemas, la estrategia debe ser sólida y podríamos considerar como buen acercamiento comenzar evaluando los prefijos o sufijos más cortos.

Durante la ejecución del algoritmo, dado que es necesario calcular los costos de las operaciones de los subproblemas, se puede comenzar de forma arbitraria desde los prefijos más cortos hasta los prefijos más largos.

El siguiente paso en la resolución de un problema mediante programación dinámica es encontrar el caso base, en nuestro problema ocurre cuando no se necesitan funciones de transformación. En otras palabras, esto sucede cuando una de las dos cadenas tiene una longitud de 0. Dado que una de las cadenas está vacía, simplemente debemos insertar todos los elementos de una cadena en la otra o eliminar todos los elementos de la otra cadena.

Para abordar el problema de manera dinámica, podemos utilizar una matriz T para almacenar los resultados de los subproblemas. Las dimensiones de la matriz T están determinadas por n y m, donde n representa la longitud de la cadena X y m representa la longitud de la cadena Y.

De esta forma cada celda $T[i][j]$ representaría la secuencia de operaciones óptima en termino de costo para transformar el prefijo de $X[i]$ en el prefijo $Y[j]$.

	Y[0]	Y[1]	Y[2]	...	Y[n-1]	Y[n]
X[0]						
X[1]						
X[2]						
...						
X[m-1]						
X[m]						

Representación de la matriz T que almacena los resultados de los subproblemas durante la ejecución del algoritmo. *bottom-up*.

Recordemos que durante la ejecución del algoritmo, se irían calculando los valores de las celdas de manera iterativa, empezando desde los prefijos más cortos (es decir, las celdas en la esquina superior izquierda de la tabla) hasta los prefijos más largos (es decir, las celdas en la esquina inferior derecha de la tabla).

Una vez que todos los valores de las celdas se hayan sido calculados, el valor de la celda $T[m][n]$ representará la distancia de edición mínima para transformar X en Y.

De forma programática, si tenemos $T[i][j]$ para todos los pares (i, j) donde $i \leq m$ y $j \leq n$, podemos construir $T[m][n]$ utilizando la siguiente observación:

- Si el último carácter de $A[i]$ es igual al último carácter de $B[j]$, entonces no necesitamos hacer nada (ninguna operación de transformación) por lo que el costo (*minimo edit distance*) sería igual a $T[i-1][j-1]$.
- Al contrario, si el último carácter de $A[i]$ es distinto al último carácter de $B[j]$, podemos realizar una de las operaciones para transformar la secuencia. Entonces aquí podríamos calcular el costo de cada operación y elegir la operación con el costo mínimo (de forma recurrente). Después de realizar la operación, consideramos el costo mínimo para los prefijos restantes.

Por lo tanto, la ecuación de recurrencia estaría dada por:

$$T[i][j] = \begin{cases} max(i,j), & \text{si } min(i,j) = 0 \\ T[i-1][j-1], & \text{si } X[i-1] = Y[j-1] \\ min \begin{cases} cost(replace) + T[i-1][i-1] \\ cost(delete) + T[i-1][j] \\ cost(insert) + T[i][j-i] \end{cases} & \text{si } X[i-1]! = Y[j-1] \end{cases}$$

Consiguientemente el algoritmo ser definido como:

```
Edit Distance Dynamic Programming(x, y)
  m <- longitud(x)
  n <- longitud(y)
  Crear una matriz T de tamaño (m+1) x (n+1) inicializada con ceros

  Para i desde 0 hasta m inclusive hacer
    Para j desde 0 hasta n, inclusive hacer
      Si min(i, j) = 0 entonces
        T[i][j] <- max(i, j)
      Si no, si x[i-1] = y[j-1] entonces
        T[i][j] <- T[i-1][j-1]
```

```
      Si no
        T[i][j] <- min( costo(replace) + T[i-1][j-1],
                        costo(delete) + T[i-1][j],
                        costo(insert) + T[i][j-1] )
    Fin Para
  Fin Para
  Devolver T[m][n]
Fin Algoritmo
```

2.d. *Programa que implemente el algoritmo DP*

El programa que implementa el algoritmo de programación dinámica se encuentra en el siguiente enlace: `https://github.com/OrtegaRehbach/Proyecto02ADA/blob/main/editDistanceDP.py`

3. Análisis teórico

3.a. *Algoritmo DaC*

3.a.1. Árbol de recursión

Podemos analizar el tiempo de ejecución construyendo el árbol de recursión para el algoritmo. Definamos $k = max(longitud(s_1), longitud(s_2))$, de modo que k representa la longitud de la cadena más larga ingresada al algoritmo. Debido a que en cada llamada a la función se decrementa una de las dos cadenas, podemos intuir que la altura máxima del árbol será igual a k. En cada llamada al algoritmo se ejecutan tres llamadas recursivas, por lo que cada nodo del árbol tendrá tres ramas. Esto continuará hasta que se alcance el caso base, es decir, cuando la longitud de alguna de las cadenas sea 0.

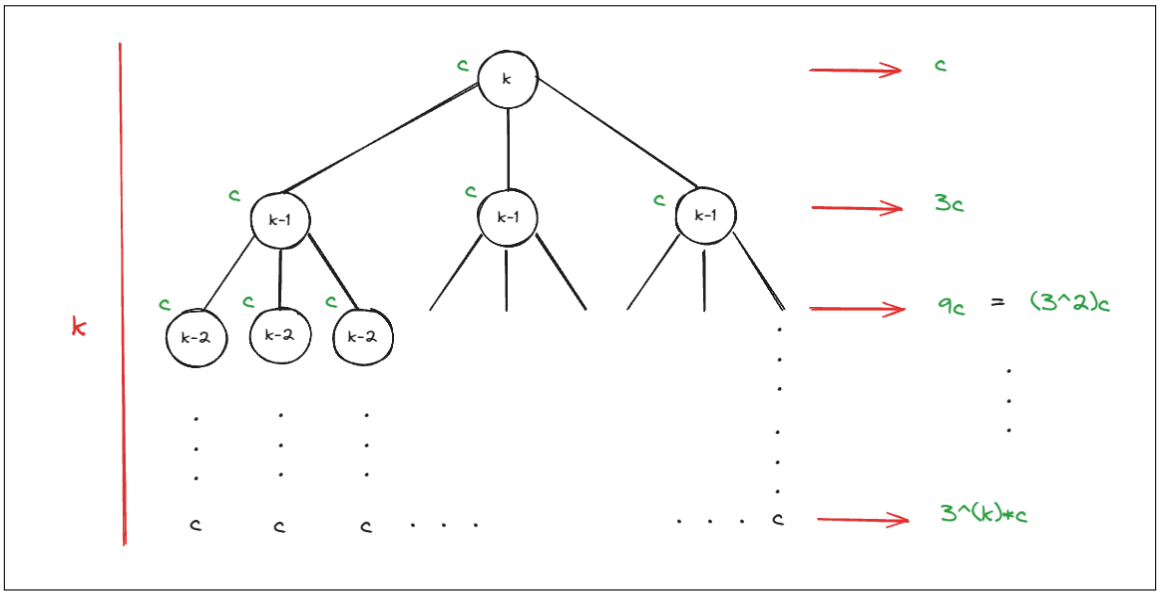


Imagen 1: Árbol de recursión del algoritmo *Edit Distance* con enfoque Divide and Conquer.

Utilizando el "ancho" $(3^k c)$ del árbol podemos obtener una buena aproximación para una cota superior del tiempo de ejecución del algoritmo. Podemos probarlo si asumimos que el tiempo de ejecución del algoritmo está dado por la siguiente ecuación de recurrencia:

$$T(k) = 3T(k-1) + k$$

Podemos probar que nuestro aproximación es válida sustituyendo esta desigualdad $T(k) \leq c \cdot 3^k$ en el término recursivo de la ecuación de recurrencia que definimos anteriormente:

$$T(k) = 3T(k-1) + k$$
$$T(k) \leq 3(c \cdot 3^{k-1}) + k$$
$$T(k) \leq c \cdot 3^k + k$$

Si aplicamos notación *Big-O* al lado derecho de la desigualdad resultante obtenemos $O(3^k)$, recordando que $k = max(longitud(s_1), longitud(s_2))$.

3.a.2. Análisis por partes

Para realizar el análisis asintótico del algoritmo de distancia de edición utilizando el enfoque de Divide and Conquer, primero debemos comprender cómo funciona el algoritmo y luego determinar cuántas operaciones realiza en función del tamaño de las cadenas de entrada.

El algoritmo Divide and Conquer divide el problema original en subproblemas más pequeños, los resuelve recursivamente y luego combina las soluciones de los subproblemas. En este caso, las operaciones básicas del algoritmo son las llamadas recursivas.

Al analizar las distintas partes del algoritmo podemos obtener la siguiente información:

1. **Dividir:** En cada llamada recursiva, las cadenas de entrada se reducen en longitud en al menos una unidad en al menos una de las cadenas (o en ambas si los últimos caracteres son diferentes). Esto implica que el número de llamadas recursivas será proporcional a la longitud de las cadenas de entrada.
2. **Conquistar y Combinar:** En cada nivel de recursión, se realizan un número constante de llamadas recursivas, exactamente tres. Esto se debe a que, independientemente de las longitudes de las cadenas, siempre se hacen tres llamadas recursivas para considerar las tres operaciones posibles (inserción, eliminación y sustitución).

Dado que en cada llamada recursiva se reduce al menos una de las cadenas en longitud y siempre se realizan tres llamadas recursivas, el tiempo de ejecución del algoritmo está dominado por el número de niveles de recursión que tiene que realizar.

Supongamos que las longitudes de las cadenas de entrada son m y n. En cada nivel de recursión, se reducen ambas cadenas en al menos una unidad, por lo que la profundidad máxima de la recursión será $max(m, n)$. Además, como en cada nivel se realizan un número constante de llamadas recursivas, el tiempo de ejecución total del algoritmo será proporcional al número total de nodos en el árbol de recursión.

Por lo tanto, el tiempo de ejecución del algoritmo de distancia de edición utilizando el enfoque de Divide and Conquer puede describirse en notación Big-O como $O(3^{max(m,n)})$, donde m y n son las longitudes de las cadenas de entrada. Sin embargo, esto es una simplificación, ya que en realidad el número de llamadas recursivas puede ser menor debido a la poda de subproblemas repetidos.

3.b. *Algoritmo DP*

En este caso, al utilizar un enfoque *bottom-up*, se ejecuta de manera constructiva, generando cada caso posible en lugar de utilizar recursión o ramificación. Esto proporciona un tiempo de complejidad estable y dependiente del tamaño de los casos necesarios.

Debido a esto, el análisis de la complejidad temporal se vuelve sencillo. Al no haber recursión real, sino solo llamadas a la estructura de datos, la operación relevante en nuestro algoritmo son los ciclos anidados. Dado que tenemos dos ciclos, uno que recorre n y otro que recorre m, el tiempo se puede definir como $O(m \times n)$.

4. Análisis Empírico

4.a. *Entradas de prueba*

En este escenario, se emplearon 30 entradas para el algoritmo basado en el enfoque de programación dinámica. Esta elección se realizó debido a que dicho enfoque tiene una complejidad temporal asintótica de $O(n^2)$. Por otro lado, utilizar las mismas entradas para el enfoque de Divide and Conquer resultaba virtualmente imposible, ya que este algoritmo presenta una complejidad polinomial de base 3 dependiente

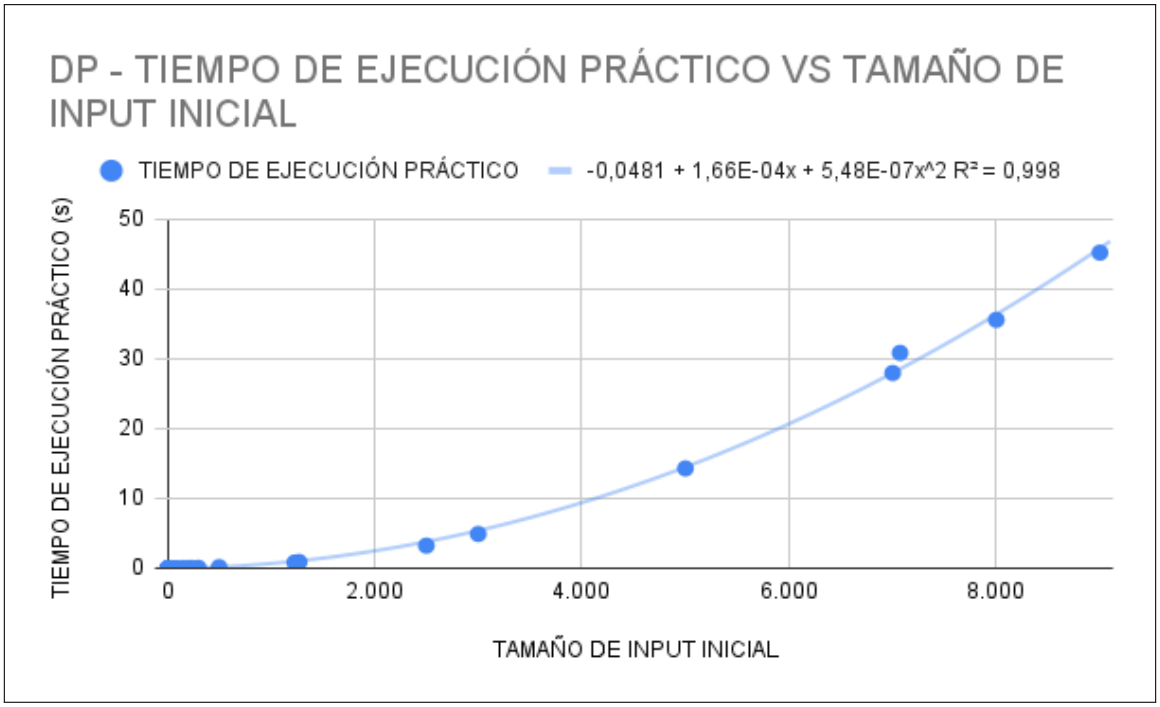
del tamaño de la entrada. Como consecuencia, su ejecución requeriría tiempos de cómputo extremadamente largos.

Por lo tanto, se optó por utilizar dos conjuntos de pruebas diferentes. El primero consistió en una serie de cadenas de prueba que variaban en longitud desde 8 hasta 9000 caracteres. Se realizaron un total de 30 pruebas en este conjunto. Por otro lado, el segundo conjunto constaba de 12 pruebas con cadenas cuya longitud iba desde 2 hasta 13 caracteres.

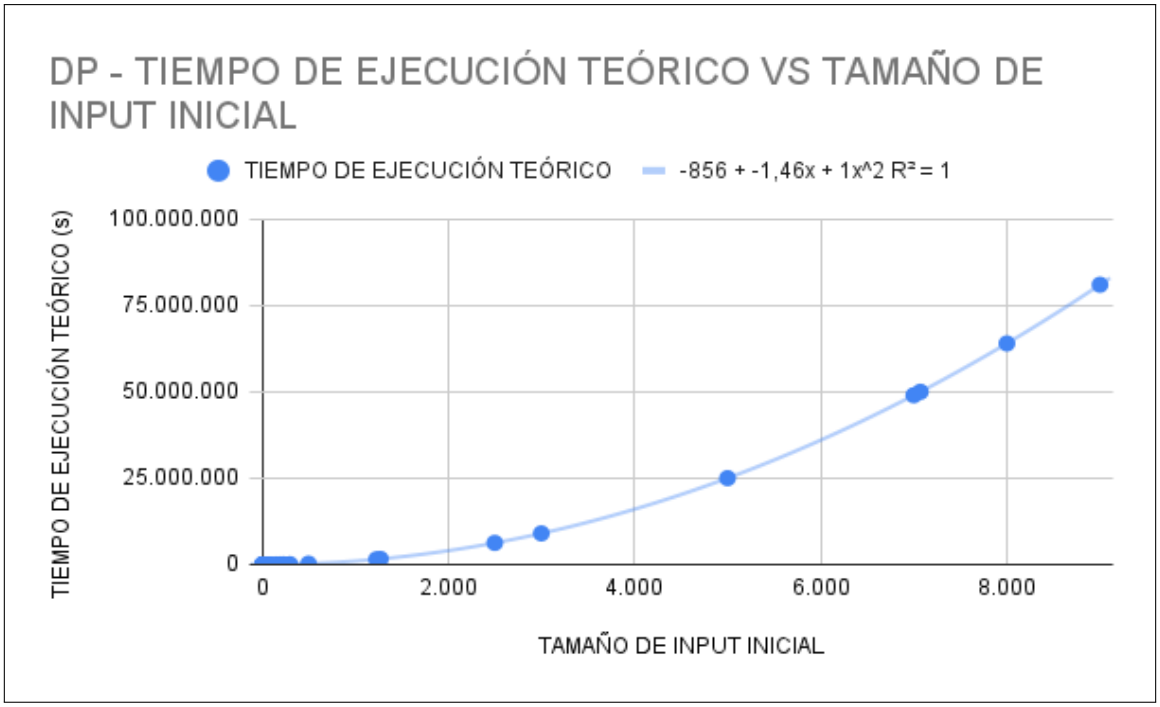
El primer conjunto mencionado puede ser hayado en el siguiente enlace (30 pruebas): `https://github.com/OrtegaRehbach/Proyecto02ADA/tree/main/sequence_hard`

El segundo conjunto de pruebas mencionado puede ser hayado en el siguiente enlace (12 pruebas): `https://github.com/OrtegaRehbach/Proyecto02ADA/tree/main/sequence_easy`

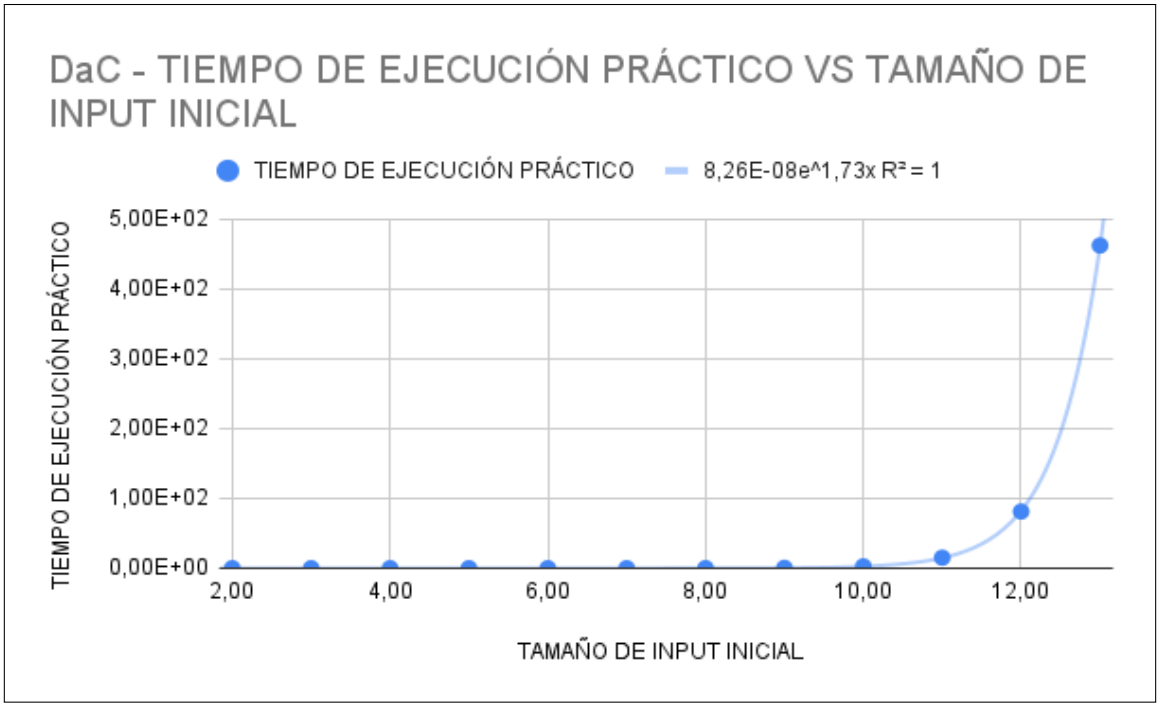
4.b. *Gráficas*



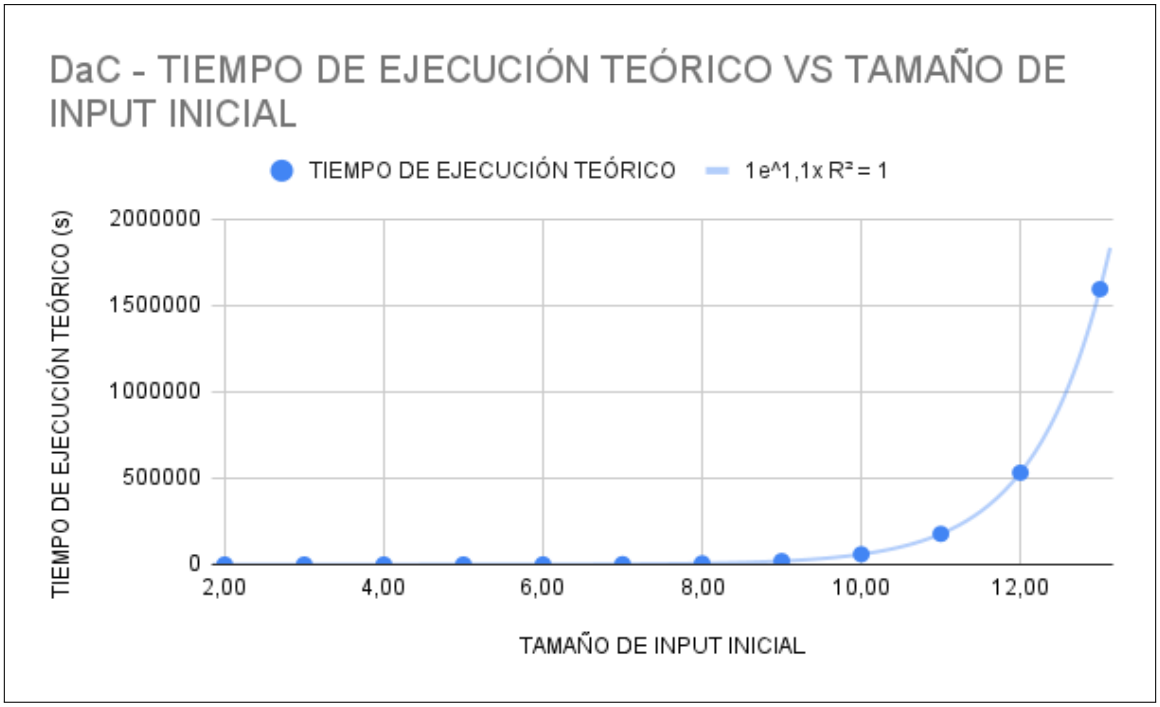
Gráfica 1: Programación Dinámica, Tiempo Práctico vs Tamaño de Input.



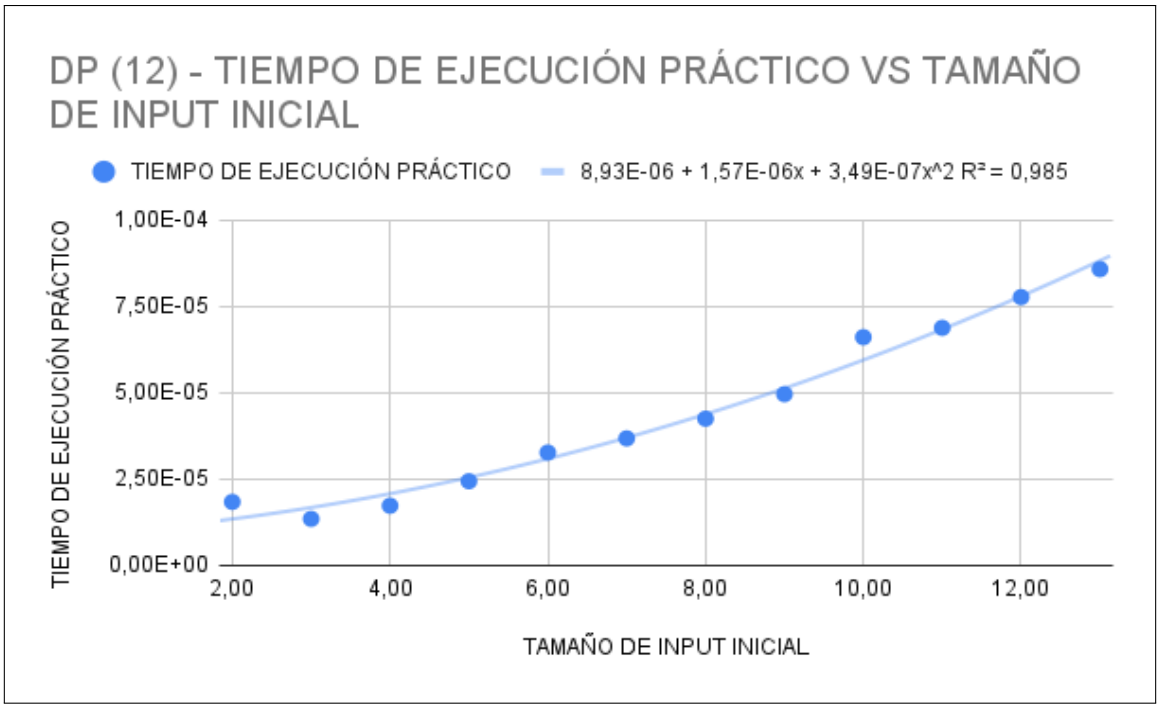
Gráfica 2: Programación Dinámica, Tiempo Teórico vs Tamaño de Input.



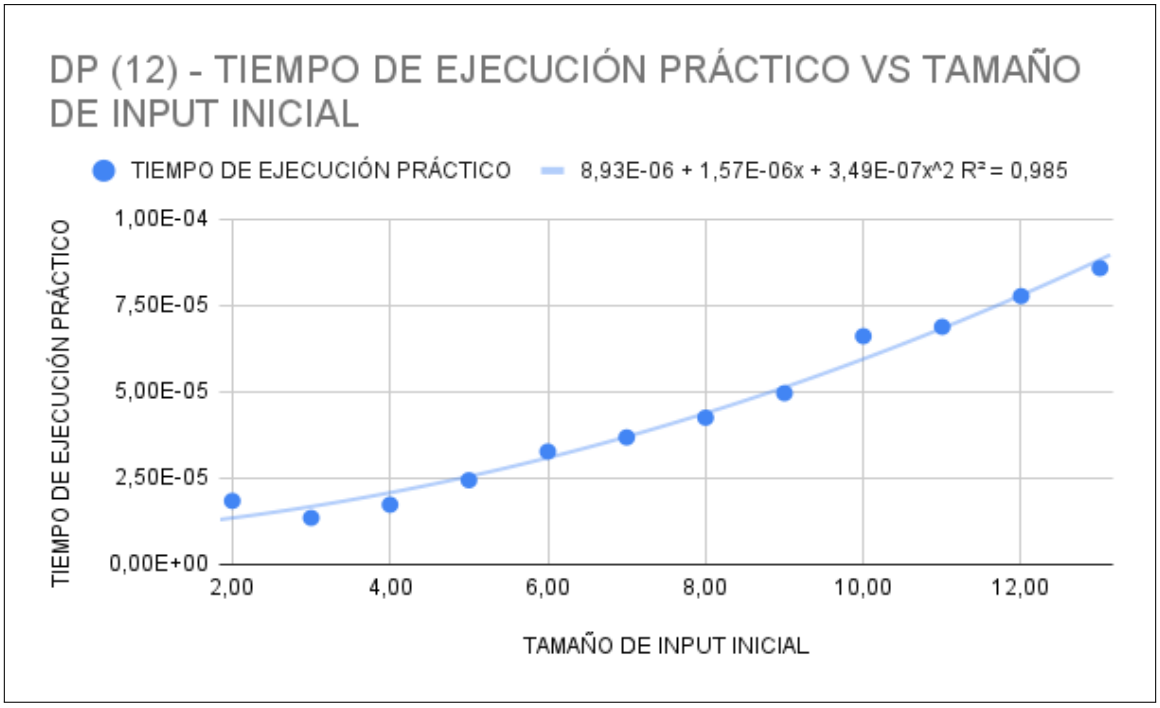
Gráfica 3: Divide and Conquer (12 Pruebas), Tiempo Práctico vs Tamaño de Input.



Gráfica 4: Divide and Conquer (12 Pruebas), Tiempo Teórico vs Tamaño de Input.



Gráfica 5: Programación Dinámica (12 Pruebas), Tiempo Práctico vs Tamaño de Input.



Gráfica 6: Programación Dinámica (12 Pruebas), Tiempo Teórico vs Tamaño de Input.

4.c. *Comentarios*

Las Gráficas 1 y 2 representan los resultados obtenidos para el primer conjunto de pruebas al utilizar el algoritmo de enfoque de "divide and conquer". Se observa que los resultados prácticos se ajustan con un coeficiente R cercano a 1 en una regresión polinomial de segundo grado. Esto concuerda con la complejidad teórica medida en $O(mn)$, donde en la mayoría de las pruebas las cadenas tenían tamaños similares, por lo que la complejidad se puede aproximar a $O(n^2)$. En base a esto, podemos sugerir que los resultados empíricos respaldan y concuerdan con las predicciones teóricas.

Las Gráficas 3 y 4 muestran los resultados obtenidos para el segundo conjunto de pruebas al utilizar el algoritmo de enfoque de programación dinámica. Los resultados del tiempo de ejecución práctico coinciden con las expectativas teóricas de complejidad, lo cual se evidencia al aplicar una regresión exponencial correspondiente a una medida teórica de $O(3^n)$. No se observa ninguna discrepancia significativa entre los resultados empíricos y los resultados teóricos. En base a esto, podemos sugerir que los resultados empíricos respaldan y concuerdan con las predicciones teóricas, indicando una coherencia entre ambos. Es importante destacar que debido a la alta complejidad de este algoritmo, no fue posible realizar simulaciones con cadenas más largas, ya que se evidencia que a partir de 12 caracteres, como se observa en la Gráfica 3, el tiempo de ejecución crece rápidamente.

Las Gráficas 5 y 6 representan nuevamente los resultados de pruebas al utilizar el algoritmo de enfoque de programación dinámica, pero con el segundo conjunto de pruebas. Al igual que en las gráficas anteriores (Gráficas 1 y 2), se observa que no existe una discrepancia significativa entre las predicciones teóricas y los resultados empíricos.

Referencias

[1] Thomas H. Cormen (2011), *Introduction to Algorithms*. Massachusetts: The MIT Press.

[2] Th. Ottmann, *Edit distance and approximate string matching*. Recuperado de `https://dbis.informatik.uni-freiburg.de/content/courses/SS09/Kursvorlesung/Theory%20I/Slides/11-Edit_Distance.pdf`

[3] Christopher D. Manning (2008), *Introduction to Algorithms*. Cambridge University Press. Recuperado de `https://www-nlp.stanford.edu/IR-book/`

[4] CodingDrills. (s.f.), *Edit Distance with DC*. Recuperado de https://www.codingdrills.com/tutorial/introduction-to-divide-and-conquer-algorithms/edit-distance-dc

[5] COMP 200 COMP 130. (s.f.), *Edit Distance Algorithms*. Rice University. Recuperado de https://www.clear.rice.edu/comp130/12spring/editdist/

[6] Lloyd Allison, Edit distance. (s.f.), *Dynamic Programming Algorithm (DPA) for Edit-Distance* . Monash University, Recuperado de https://users.monash.edu/~lloyd/tildeAlgDS/Dynamic/Edit/