DIVIDE Y VENCERAS

KATERYNA PASTERNAK
IVÁN ANDRÉS YAGUAL MÉNDEZ

PROBLEMA 4

1. DISEÑO DEL ARGORITMO

Para diseñar una solución basada en la técnica de **divide y vencerás** para resolver el problema de encontrar la subcadena con la mayor diferencia total en los valores ASCII entre caracteres consecutivos, es esencial comprender primero el funcionamiento del algoritmo directo, que posteriormente será dividido en partes más pequeñas. Este algoritmo directo, descrito a continuación, calcula de forma secuencial la suma de las diferencias absolutas entre caracteres consecutivos en subcadenas de tamaño fijo. A partir de este enfoque, se aplica la técnica de **divide y vencerás** con el objetivo de optimizar el proceso y mejorar su eficiencia.

1.1. Diseño del Algoritmo Directo

El algoritmo directo tiene como objetivo encontrar las subcadenas de longitud "window_size" con la mayor suma de las diferencias absolutas entre los valores ASCII de caracteres consecutivos en una cadena dada. Para hacerlo, primero calculamos las diferencias entre caracteres consecutivos en la cadena y luego, para cada posible subcadena de longitud "window_size", sumamos estas diferencias y guardamos aquellas que tienen el total más alto.

Decisiones de Diseño:

- **Cálculo de diferencias entre caracteres consecutivos**: En lugar de recalcular las diferencias entre caracteres en cada subcadena, lo que sería ineficiente, primero se genera una lista de diferencias entre caracteres consecutivos en la cadena. Esto permite que, para cada subcadena, podamos simplemente sumar los valores correspondientes de la lista de diferencias sin necesidad de volver a hacer estos cálculos repetitivos. Por ejemplo, si la cadena es "abc", las diferencias serían: |ord('a') - ord('b')| y |ord('b') - ord('c')|.

- **Manejo de múltiples soluciones**: En caso de que varias subcadenas tengan la misma suma máxima de diferencias, se guardan todas estas subcadenas y se devuelven al final, junto con su suma y la posición inicial de cada una.

- **Control de validez de parámetros**: El algoritmo también verifica que los parámetros sean válidos, asegurándose de que el tamaño de la ventana "window_size" sea menor o igual a la longitud de la cadena y que sea al menos 2 (ya que una diferencia entre caracteres consecutivos requiere al menos dos caracteres).

1.2. Diseño del Algoritmo de Divide y Vencerás

El algoritmo de divide y vencerás se utiliza en este caso para encontrar la subcadena de longitud “window_size” con la mayor diferencia en los valores ASCII entre caracteres consecutivos en una cadena dada. A continuación, se describen los principales componentes del algoritmo y cómo estos se organizan para resolver el problema.

1.2.1. Descripción General

El algoritmo de divide y vencerás aborda el problema dividiendo la cadena en subproblemas más pequeños, lo que permite gestionar la solución de forma más eficiente. De manera inicial, se calcula la lista de diferencias entre caracteres consecutivos en la cadena. Luego, el problema se divide recursivamente hasta que las subcadenas alcanzan el tamaño adecuado para aplicar una solución directa.

1.2. 2. Funciones Clave

calculate_total_diff_substring(start: int) -> int

Esta función calcula la suma de las diferencias absolutas entre caracteres consecutivos en una subcadena dada, empezando desde el índice “start”. Esta función es esencial para el cálculo de la "diferencia total" de una subcadena y es utilizada tanto en la fase base como en las fases recursivas del algoritmo.

divide_and_conquer(i: int, j: int) -> Tuple[int, int]

La función “divide_and_conquer” es la función recursiva principal. Toma dos índices, “i” y “j”, que definen un rango dentro de la cadena. Esta función gestiona tres casos:

- Caso base 1: Si el rango de la subcadena es menor que el tamaño de la ventana (window_size), no se puede procesar y la función devuelve una diferencia máxima negativa (float('-inf')).
- Caso base 2: Si el tamaño de la subcadena es igual al tamaño de la ventana, se calcula la diferencia total utilizando la función “calculate_total_diff_substring”.
- Caso recursivo: Si la subcadena es mayor que el tamaño de la ventana, la función divide el problema en dos subproblemas más pequeños, correspondientes a la mitad izquierda y la mitad derecha de la subcadena, y resuelve ambos recursivamente.

frontera_case_solution(i: int, m: int, j: int) -> Tuple[int, int]

La función “frontera_case_solution” maneja los casos que involucran la "frontera" entre las mitades izquierda y derecha de la cadena. Es necesario evaluar estas subcadenas fronterizas ya que la mejor solución podría involucrar caracteres de ambas mitades. Esta función aplica una solución directa para las subcadenas que cruzan la frontera y devuelve la mejor subcadena encontrada en esa sección.

1.3. Diseño del Algoritmo Directo Óptimo

El algoritmo directo óptimo, en lugar de calcular la suma para cada subcadena desde cero, utiliza un enfoque de "ventana deslizante" para optimizar el cálculo y hacerlo más eficiente. Este algoritmo es el más fácil de implementar y también el óptimo para el problema.

Decisiones de Diseño:

1. **Cálculo de diferencias entre caracteres consecutivos**: En lugar de recalcular las diferencias entre caracteres en cada subcadena, lo que sería ineficiente, primero generamos una lista de las diferencias absolutas entre caracteres consecutivos en la cadena.
2. **Enfoque de ventana deslizante**: Una vez que tenemos la lista de diferencias, podemos usar una ventana deslizante para calcular la suma de las diferencias para cada subcadena de longitud “window_size”. Esto significa que:

    1. Inicialmente, se calcula la suma de las diferencias para la primera ventana (subcadena) de tamaño “window_size”.

    2. Luego, en lugar de recalcular la suma desde cero para cada nueva ventana, se optimiza el proceso restando la diferencia que sale de la ventana (el primer valor de la diferencia) y sumando la diferencia que entra en la ventana (el siguiente valor de la diferencia).

    Este enfoque resulta más eficiente porque reduce el cálculo de la suma de diferencias a un ajuste incremental, evitando la necesidad de recomputar todo desde el principio.

2. Validación del algoritmo DV implementado

El proceso de validación del algoritmo implementado se realiza mediante la comparación de las soluciones generadas por los métodos de "Divide y Vencerás" (DV) y el método directo. Para ello, se utiliza la función “validation”, que valida los resultados de ambos métodos aplicados a

un conjunto de casos de prueba definidos en un diccionario. En cada caso, se proporcionan una cadena generada aleatoriamente y el tamaño de la ventana (window_size) a analizar. El objetivo es comparar las soluciones generadas por ambos métodos y verificar que los resultados coincidan.

La función “validation” ejecuta ambos métodos, el directo y el de divide y vencerás, y luego compara sus resultados. En caso de que haya discrepancias, las registra y las imprime para su revisión.

Al comparar los resultados del método de Divide y Vencerás con los del método directo, se verificó que todos los tests pasaron correctamente y no se encontraron discrepancias entre los resultados generados por ambos métodos.

3. Análisis teórico del tiempo de ejecución del algoritmo

3.1. direct_method_optimal

La función “direct_method_optimal” utiliza una ventana deslizante para encontrar subcadenas de longitud “window_size” en una cadena de caracteres, y calcula la suma de las diferencias absolutas entre caracteres consecutivos dentro de esas subcadenas. El análisis de complejidad se puede hacer en dos partes principales:

- **Cálculo de las diferencias entre caracteres consecutivos**: La lista “difference” se calcula utilizando una lista por comprensión, donde se evalúa la diferencia entre caracteres consecutivos de la cadena. Este paso tiene un tiempo de ejecución de **O(n)**, donde n es la longitud de la cadena.
- **Deslizamiento de la ventana**: Posteriormente, se recorre la cadena con una subcadena deslizante para calcular las diferencias totales para cada subcadena. La actualización de la suma de las diferencias entre subcadenas se realiza de manera eficiente, restando la diferencia que sale de la ventana y sumando la diferencia que entra. Este paso también tiene un tiempo de ejecución de **O(n)**, ya que el bucle recorre la cadena en su totalidad.

Tiempo total de ejecución:

- La complejidad temporal es **O(n)**, ya que el tiempo de ejecución de cada parte es lineal con respecto a la longitud de la cadena.

3.2. direct_method

La función “direct_method” es similar a “direct_method_optimal” pero no utiliza una ventana deslizante optimizada. Aquí, para cada subcadena de tamaño “window_size”, se calcula la suma de las diferencias entre caracteres consecutivos.

- **Cálculo de las diferencias entre caracteres consecutivos**: Al igual que en la función anterior, el cálculo de las diferencias en la lista “difference” se realiza en **O(n)**.
- **Cálculo de la diferencia total para cada subcadena**: En esta función, para cada subcadena de tamaño “window_size”, se calcula la suma de las diferencias mediante un bucle. La operación “sum(difference[i:i + window_size - 1])” tiene un tiempo de ejecución de **O(window_size)** para cada subcadena. Dado que el número de subcadenas posibles es **O(n)**, el tiempo de ejecución total de esta parte es **O(n * window_size)**.

Tiempo total de ejecución:

- La complejidad temporal de esta función es **O(n * window_size)**, que puede ser menos eficiente cuando “window_size” es grande.

3.2. divide_and_conquer_method

La función “divide_and_conquer_method” utiliza el enfoque de divide y vencerás para dividir la cadena en subproblemas más pequeños y resolverlos recursivamente.

3.2.1. División de la cadena:

El algoritmo divide la cadena en subproblemas más pequeños. En el caso de la recursión, la cadena se divide en dos mitades, lo que significa que el tamaño de los subproblemas se reduce a la mitad en cada paso. Por lo tanto, en cada llamada recursiva, el tamaño de los subproblemas disminuye en un factor de b=2.

3.2.2. Número de subproblemas:

En cada paso, el número de subproblemas aumenta en un factor de a=2, ya que la cadena se divide en dos subproblemas más pequeños.

3.2.3. Tiempo de combinación (f(n)):

La parte de "combinación" en el algoritmo se refiere a la evaluación de las soluciones de los subproblemas y la fusión de estas soluciones en un resultado final. En este caso, el algoritmo evalúa la "frontera" entre las dos mitades de la cadena. El proceso de evaluación de la frontera

involucra iterar sobre una pequeña cantidad de subcadenas que se encuentran entre las dos mitades y calcular la diferencia total de las subcadenas. Esto toma un tiempo proporcional al tamaño de la subcadena o O(n).

Entonces, el tiempo de combinación f(n) es O(n) en el peor caso.

3.2.4. Estructura recursiva:

Podemos modelar la complejidad del algoritmo usando la relación de recurrencia:

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + O(n)$$

Aquí, tenemos a=2, b=2 y f(n)=O(n).

3.2.5. Aplicación de la formula maestra:

Ahora, aplicando la formula maestra obtenemos la complejidad temporal:

$$T(n) = O(n \log n)$$

Conclusión:

La complejidad temporal de la función “divide_and_conquer_method” es O(n * log n).

4. Estudio experimental del tiempo de ejecución

Para realizar el estudio experimental, se implementó la función “time_test”, que evalúa el tiempo de ejecución de tres métodos diferentes con distintos tamaños de entrada. Los métodos evaluados son:

1. **Método directo básico**
2. **Método directo óptimo**
3. **Método basado en divide y vencerás**

La función “time_test” toma como entrada un diccionario donde las claves son cadenas aleatorias y los valores son los tamaños de ventana. Para cada caso, se mide el tiempo de ejecución de los tres métodos y se almacenan los resultados para su posterior análisis.

El gráfico resultante muestra cómo el tiempo de ejecución varía con el tamaño de la entrada para cada uno de los métodos. Se ajusta una curva a los resultados para cada uno, lo que permite observar de manera visual las diferencias en la eficiencia de los métodos a medida que aumenta el tamaño de los casos.

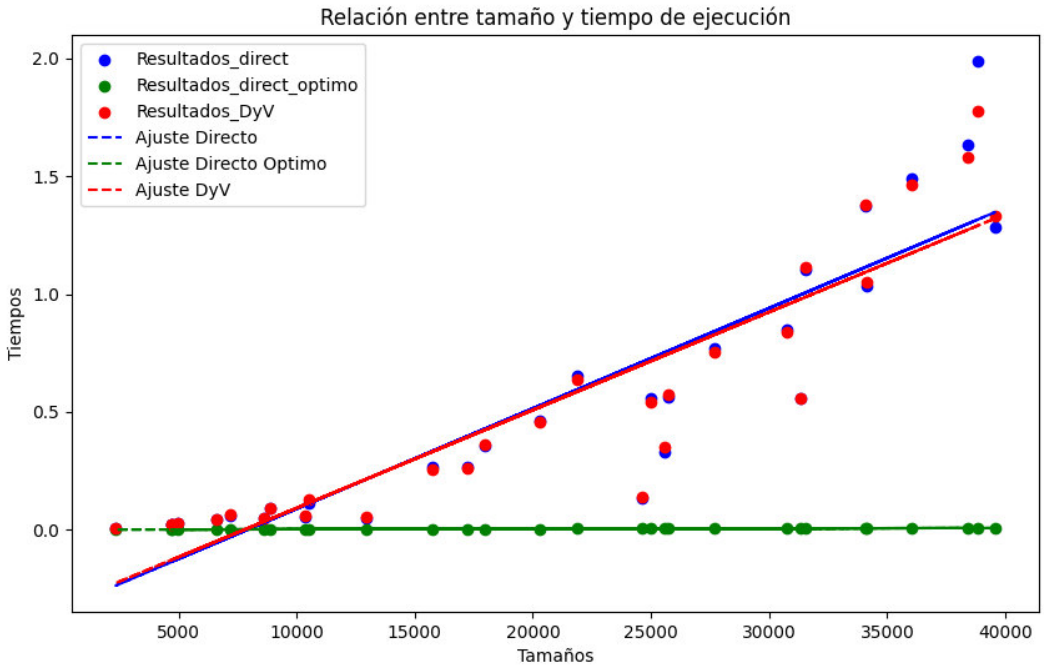
En cuanto al **método óptimo directo**, los resultados experimentales confirmaron que es, de hecho, el más eficiente. A medida que el tamaño de la entrada aumenta, la complejidad de este

algoritmo parece mantenerse constante, mostrando un comportamiento de tiempo de ejecución cercano a O(1) en la mayoría de los casos. Esto sugiere que la optimización aplicada ha reducido significativamente el tiempo de ejecución, independientemente del tamaño de la entrada.

En cuanto a los métodos directo básico y divide y vencerás, los resultados muestran que ambos algoritmos tienen un comportamiento similar. Sin embargo, el método de divide y vencerás no logra la mejora de eficiencia esperada a medida que crece el tamaño de la entrada. Esto podría deberse a que, aunque esta técnica es eficiente en ciertos escenarios, no se aprovechó completamente en este caso debido a solapamientos de subproblemas o a una mayor complejidad en la combinación de soluciones. Estas operaciones pueden ser más costosas que la resolución directa, especialmente para tamaños de entrada pequeños o intermedios. Es posible que el algoritmo de divide y vencerás muestre una mejora más significativa cuando tanto el tamaño de la entrada como el de la ventana sean considerablemente grandes.

Conclusión:

- **Método óptimo directo** es claramente el mejor, además en muchos casos, muestra una complejidad de tiempo O(1) debido a su eficiencia.
- **Método directo básico y divide y vencerás** presentan un comportamiento similar, con el segundo mostrando una ligera mejora de eficiencia cuando el tamaño de la entrada es grande. No obstante, los resultados experimentales no muestran un incremento tan significativo en la eficiencia como se esperaba teóricamente, posiblemente debido a factores como la superposición de subproblemas o la complejidad de la combinación de soluciones.



5. Conclusiones

Como grupo, consideramos que esta actividad ha sido una excelente oportunidad para aplicar y poner en práctica los conocimientos adquiridos sobre diferentes enfoques algorítmicos, especialmente en lo que respecta a la comparación entre métodos directos y de divide y vencerás. A través de la experimentación, hemos podido entender mejor cómo el tamaño de los datos influye en el rendimiento de los algoritmos y cómo optimizar la solución en función de estas variaciones.

El **método directo óptimo** ha sido el que ha demostrado mejor rendimiento, con tiempos de ejecución más bajos, lo que indica que la optimización de los algoritmos es crucial para mejorar la eficiencia. Por otro lado, el **método de divide y vencerás** no mostró el aumento de eficiencia que esperábamos en los casos experimentales. A pesar de la teoría que sugiere una mejora significativa al dividir el problema en subproblemas más pequeños, la implementación no resultó ser tan eficiente como esperábamos, probablemente debido a la sobrecarga del proceso de combinación de soluciones de los subproblemas, lo que generó un costo adicional. Esto refuerza la idea de que, en algunos casos, el enfoque más sencillo y directo puede ser más eficiente que una estrategia más compleja.

En cuanto a las pruebas experimentales, hemos aprendido a medir el tiempo de ejecución de manera precisa y a interpretar los resultados, lo cual es clave para entender la escalabilidad de los algoritmos. Los gráficos generados a partir de los tiempos de ejecución nos dieron una

visión clara de cómo los distintos algoritmos se comportan con diferentes tamaños de entrada, permitiéndonos comparar sus rendimientos de manera efectiva.

Como grupo, nos sentimos satisfechos con el resultado final, ya que no solo logramos implementar los tres métodos, sino que también conseguimos obtener un análisis detallado de sus comportamientos y limitaciones.

Estimación del tiempo invertido por cada miembro del grupo:

- Kateryna:
    - Desarrollo del algoritmo y pruebas iniciales: 4 horas
    - Análisis teórico y estudio de la complejidad: 1 hora
    - Validación experimental y ajuste de gráficos: 2 horas
    - Documentación: 1 hora
    - Total: 8 horas
- Iván:
    - Mejora del algoritmo: 3 horas
    - Diseño de las pruebas y estudio experimental: 3 horas
    - Redacción de conclusiones: 1 hora
    - Total: 7 horas