PROYECTO

PEDRO BERNAL LONDOÑO - 2259548-3743

JOTA EMILIO LÓPEZ - 2259394-3743

ESMERALDA RIVAS GUZMÁN - 2259580-3743

DIRECTOR

CARLOS ANDRÉS DELGADO

UNIVERSIDAD DEL VALLE SEDE TULUÁ

FACULTAD DE INGENIERÍA

PROGRAMA ACADÉMICO DE INGENIERÍA DE SISTEMAS

ANÁLISIS Y DISEÑO DE ALGORITMOS II

TULUÁ – VALLE DEL CAUCA

2024

Este proyecto tiene como objetivo principal resolver dos problemas mediante la aplicación de estrategias de fuerza bruta, programación dinámica y programación voraz. Los problemas a abordar son la transformación de cadenas de texto en "La terminal inteligente", donde se deben minimizar los costos de operación al modificar una cadena, y "El problema de la subasta pública", que determina la distribución de acciones entre diferentes oferentes para maximizar los ingresos.

1. La terminal inteligente

1.1. Entender el problema

Costos: a = 1, d = 2, r = 3, i = 2, k = 1.

Transformar la cadena **ingenioso** en **ingeniero**.

Operación	Cadena
-	i n g e n i o s o
advance	i n g e n i o s o
advance	i n g e n i o s o
advance	i n g e n i o s o
advance	i n g e n i o s o
advance	i n g e n i o s o
advance	i n g e n i o s o
insert e	i n g e n i e o s o
insert r	i n g e n i e r o s o
advance	i n g e n i e r o s o
kill	i n g e n i e r o _
Costo = 7a + 2i + k = 7 (1) + 2 (2) + 1 = 7 + 4 + 1= 12	

Operación	Cadena
-	i n g e n i o s o
advance	i n g e n i o s o
advance	i n g e n i o s o
advance	i n g e n i o s o
advance	i n g e n i o s o

advance	i n g e n i o s o
advance	i n g e n i o s o
kill	i n g e n i _
insert e	i n g e n i e _
insert r	i n g e n i e r _
insert o	i n g e n i e r o _
Costo = 6a + k + 3i = 6 (1) + 1 + 3 (2) = 6 + 1 + 6 = 13	

Transformar la cadena **francesa** en **ancestro**.

Operación	Cadena
-	f r a n c e s a
delete	r a n c e s a
delete	a n c e s a
advance	a n c e s a
advance	a n c e s a
advance	a n c e s a
advance	a n c e s a
advance	a n c e s a
replace with t	a n c e s t _
insert r	a n c e s t r _
insert o	a n c e s t r o _
Costo = 2d + 5a + r + 2i = 2(2) + 5(1) + 3 + 2(2) = 16	

Operación	Cadena
-	f r a n c e s a
delete	r a n c e s a
delete	a n c e s a

advance	a n c e s a
advance	a n c e s a
advance	a n c e s a
advance	a n c e s a
advance	a n c e s a
kill	a n c e s _
insert t	a n c e s t _
insert r	a n c e s t r _
insert o	a n c e s t r o _
Costo = 2d + 5a + k + 3i = 2(2) + 5(1) + 1 + 3(2) = 16	

1.2 Caracterizar la estructura de una solución óptima

La estructura de una solución óptima para la transformación de una cadena base $x[1..n]$ en otra cadena objetivo $y[1..n]$ está compuesta por soluciones óptimas de subproblemas más pequeños, esto significa que cualquier secuencia óptima de operaciones para transformar una parte de x en una parte de y también será óptima para esas subsecuencias. Ahora bien, en el problema de la terminal inteligente, esto implica que para cualquier prefijo $x[1..i]$ de la cadena base y cualquier prefijo $y[1..i]$ de la cadena objetivo, la secuencia de operaciones mínima para convertir $x[1..i]$ en $y[1..i]$ debe estar compuesta de la secuencia óptima de las transformaciones para sus subcadenas, esta propiedad de la programación dinámica nos permite abordar el problema de la terminal inteligente de forma contundente.

1.3 Definir recursivamente el valor de una solución óptima

El valor de una solución óptima puede ser definido recursivamente utilizando los costos asociados a las operaciones *(advance, delete, replace, insert, kill)* , sea $dp[i][j]$ el costo mínimo para transformar el prefijo $x[1..i]$ de la cadena base en el prefijo $y[1..i]$ de la cadena objetivo, la relación de recurrencia para este valor es:

$$dp[i][j] = \min \begin{cases} dp[i+1][j+1] + \min(C_{advance}, C_{replace}) \\ dp[i+1][j+1] + C_{replace} \\ dp[i+1][j] + C_{delete} \\ dp[i][j+1] + C_{insert} \\ dp[i+1][j] + C_{kill} + (n-j) \cdot C_{insert} \end{cases}$$

Supongamos que tenemos las cadenas *x="a"* y *y="ab"*, donde *m* representa la cantidad de caracteres de *x*, y *n* la cantidad de caracteres de *y*, vamos a crear una matriz de tamaño *[n+1,m+1]*, inicializada con el valor numérico más grande posible, que representaremos como "*infinito*":

infinito	*infinito*	*infinito*
infinito	*infinito*	*infinito*

A continuación, establecemos el valor de *dp[m][n]* en 0, lo que representa el costo de transformar dos cadenas vacías, este valor se convierte en nuestro caso base primario y servirá como punto de partida para el llenado de la matriz, comenzaremos a llenar la matriz de abajo hacia arriba (bottom-up), donde cada celda se calculará utilizando los resultados de subproblemas previamente resueltos, asegurando que cada operación posterior esté fundamentada en los costos acumulados de las transformaciones

infinito	*infinito*	*infinito*
infinito	*infinito*	*0*

Una vez establecido el caso base, el algoritmo maneja situaciones donde una de las cadenas, ya sea la cadena base o la cadena objetivo, está vacía, en estos casos, solo existe una opción, insertar o eliminar caracteres (incluso una eliminación total, "kill"), tras llenar estos casos, la matriz se verá de la siguiente forma:

infinito	*infinito*	*1*
4	*2*	*0*

A continuación, el algoritmo procede a evaluar cada carácter de ambas cadenas, si los caracteres son iguales, se consideran dos posibles opciones: avanzar al siguiente carácter en ambas cadenas, con un costo definido por la operación *advance*, o realizar un reemplazo del carácter por sí mismo, con el costo correspondiente a replace, la opción con menor costo es la seleccionada, ahora bien, cuando los caracteres no coinciden, se evalúan las operaciones disponibles: *replace* para cambiar el carácter actual de la cadena base por el correspondiente de la cadena objetivo, *delete* para eliminar el carácter de la cadena base, o i*nsert* para insertar el carácter de la cadena objetivo en la cadena base, también se evalúa la opción *kill*, que elimina todos los caracteres restantes de la cadena base y luego inserta los caracteres faltantes de la cadena objetivo, el algoritmo selecciona la operación con el menor costo y actualiza las matriz dp con el costo, después de haber evaluado todos los casos y haber llenado la tabla completa, obtenemos una representación visual de los costos asociados a cada transformación.

En la siguiente tabla se muestra cómo se ve la matriz luego de completar todos los pasos del algoritmo:

3	*3*	*1*
4	*2*	*0*

Finalmente, la solución óptima se obtiene en la celda *dp[0][0]*, que contiene el costo mínimo para transformar la cadena base en la cadena objetivo.

1.4 Implementación de las soluciones y complejidad temporal

Estrategia por fuerza bruta

La función *transform_string_brute_force* utiliza una función recursiva interna llamada *brute_force*, que evalúa los diferentes casos posibles en cada paso (*advance, delete, insert, replace, kill)* y compara los costos de las operaciones para seleccionar el más bajo. Esta función se llama con índices *i* y *j* para iterar sobre las posiciones de la cadena base (*base_string)* y la cadena objetivo (*target_string)*, respectivamente.

La complejidad temporal del algoritmo en el peor caso es $O(4^{min(m,n)})$, donde *m* es la longitud de la cadena base y *n* la longitud de la cadena objetivo. Este comportamiento se debe a que la función recursiva evalúa todas las posibles combinaciones de operaciones para cada par de índices lo que implica una exploración exhaustiva en cada paso, lo que termina en un aumento exponencial en función de las longitudes de las cadenas y de la cantidad de operaciones.

Estrategia por programación dinámica

La función *transform_string_dp*, utiliza una matriz $dp[i][j]$ para almacenar el costo mínimo de transformar los primeros *i* caracteres de la cadena base en los primeros *j* caracteres de la cadena objetivo. Además, usa una matriz $op[i][j]$ que registra la operación que se debe realizar en ese punto de la transformación.

El algoritmo llena estas matrices de abajo hacia arriba y de derecha a izquierda, evaluando el costo en cada paso. Una vez que la matriz *dp* está completa, se utiliza la función *build_operation_steps* para reconstruir la secuencia de operaciones realizadas, basándose en las decisiones almacenadas en la matriz *op*.

La complejidad temporal se ve optimizada, debido a que se almacenan los resultados parciales, evitando la repetición de cálculos. La matriz *dp* tiene dimensiones $(m + 1) * (n + 1)$, donde *m* y *n* son las longitudes de las cadenas base y objetivo. Cómo se evalúan todas las celdas una vez, la complejidad temporal es $O(m * n)$.

Estrategia por programación voraz

La función *transform_string_greedy* elige la operación con el menor costo en cada paso sin considerar el costo global a largo plazo, a medida que avanza evalúa las siguientes opciones:

- Si $i = m$, es decir, hemos procesado toda la cadena base, el algoritmo inserta los caracteres restantes de la cadena objetivo.
- Si $j = n$, es decir, hemos procesado toda la cadena objetivo, el algoritmo elige entre *kill* y *delete*.
- Si los caracteres en ambas cadenas coinciden, el algoritmo elige entre *advance* y *replace*.
- Si los caracteres no coinciden, el algoritmo elige entre *replace, delete e insert*, teniendo en cuenta el costo de las operaciones y sus combinaciones.

La complejidad temporal es $O(m + n)$, debido a que el algoritmo solo recorre las cadenas una vez, evaluando una operación en cada paso, tomando decisiones mínimas sin considerar el costo total de la secuencia, razón por la que no se garantiza la solución óptima global.

2. El problema de la subasta pública

2.1. Entender el problema

El problema consiste en asignar *A* acciones a distintos oferentes para maximizar el valor total obtenido por el gobierno, mientras se respetan las restricciones impuestas por cada oferta:

- Cada oferta tiene la forma $< precio, mínimo, máximo >$
- El gobierno venderá las acciones sobrantes a un precio mínimo *B*
- El objetivo es determinar la asignación *X* de acciones a cada oferente tal que se maximice el valor total $vr(X)$, cumpliendo que todas las acciones *A* sean asignadas.

$A = 1000, B = 100, n = 2$

Ofertas: $< 500, 100, 600 >, < 450, 400, 800 >$

Oferta del gobierno: $< 100, 0, 1000 >$

Asignaciones:

$X = < 600, 400, 0 >, Vr = 600(500) + 400(450) + 0 = 480000$
$X = < 200, 800, 0 >, Vr = 200(500) + 800(450) + 0 = 460000$

2.2. Una primera aproximación

Dado un número $A = 1000$, un precio mínimo $B = 100$ y $n = 4$ con las siguientes ofertas:

Oferta 1: $< 500, 400, 600 >$

Oferta 2: $< 450, 100, 400 >$

Oferta 3: $< 400,\ 100,\ 400 >$

Oferta 4: $< 200,\ 50,\ 200 >$

Oferta del gobierno: $< 100,\ 0,\ 1000 >$

Asignación:

$X = < 600, 400, 0, 0, 0 >, Vr = 600(500) + 400(450) + 0 + 0 + 0 = 480000$

2.3. Caracterizar la estructura de una solución óptima:

La estructura de una solución óptima para la subasta pública se basa en descomponer el problema original en subproblemas más pequeños. Esto significa que:

- La solución óptima $Vr(x)$ para asignar *A* acciones puede ser obtenida considerando las soluciones óptimas para subproblemas donde se asigna una subconjunto de acciones a los primeros i oferentes.
- Cada subproblema puede ser resuelto de forma independiente, lo que hace que el enfoque de programación dinámica sea adecuado.

Caracterización del subproblema:

- Sea $dp[i][a]$ el valor máximo que se puede obtener asignando exactamente *a* acciones utilizando las primeras *i* ofertas. Entonces, la solución óptima para el problema general estará en $dp[n][A]$.

2.4. Definir recursivamente el valor de una solución óptima:

La recurrencia consiste en las decisiones que se toman para cada oferta, si se asigna una cantidad *x* de acciones a la oferta *i*, entonces el valor total dependerá de lo que se haya obtenido al resolver el subproblema de las primeras $i - 1$ ofertas, con $a - x$ acciones restantes

Enfoque General

Supongamos que tenemos:

- *A:* Número total de acciones disponibles.
- *B:* Precio mínimo al que el gobierno comprará cualquier acción que no sea asignada a un oferente.
- *n:* Número de oferentes.
- Cada oferente *i* tiene una oferta en forma de una tripleta (p_i, m_i, M_i), donde:
    - p_i : Precio que el oferente está dispuesto a pagar por cada acción.
    - m_i : Número mínimo de acciones que el oferente debe comprar.
    - M_i : Número máximo de acciones que el oferente puede comprar.

El objetivo es maximizar el valor total recibido $Vr(x)$ de asignar acciones a los oferentes de forma que:

$$vr(X) = \sum_{i=1}^{n} x_i p_i + (A - \sum_{i=1}^{n} x_i)B$$

donde x_i representa el número de acciones asignadas al oferente *i*.

Definir Recursivamente la Solución Óptima

Sea $dp[i][a]$ el valor máximo que se puede obtener al asignar exactamente *a* acciones entre los primeros *i* oferentes. Para calcular el valor de $dp[i][a]$, tenemos dos opciones:

- No asignar ninguna acción al *i-ésimo* oferente: $dp[i][a] = dp[i-1][a]$.
- Asignar x_i acciones al *i-ésimo* oferente, donde $m_i \leq x_i \leq min(M_i, a)$.

$$dp[i][a] = max\left(dp[i-1][a],\ max_{x_i \in [m_i, min(M_i, a)]}\left(dp[i-1][a - x_i] + x_i * p_i\right)\right)$$

2.5 Calcular el valor de una solución

Utilizamos la relación de recurrencia para llenar la tabla *dp* de abajo hacia arriba (enfoque bottom-up). La solución óptima estará en la celda $dp[n][A]$, que representa el valor máximo posible utilizando todos los oferentes y asignando exactamente *A* acciones, y se recupera la mejor asignación retrocediendo desde $dp[n][A]$ y siguiendo las decisiones tomadas en la tabla.

Supongamos un problema con los siguientes parámetros: $A = 10,\ B = 50,\ n = 2,$ ofertas: $< 200, 2, 5 >,\ < 150, 3, 7 >$.

Llenemos una matriz *dp* para este caso, con dimensiones $[n + 1][A + 1]$, es decir $3 * 11$. Inicializamos toda la tabla con 0, ya que al principio no se ha asignado ninguna acción. La tabla inicial será:

$$\text{dp} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Para $i = 1$, el mínimo de acciones que se pueden asignar es 2, y el máximo 5. Para $a = 0, 1$, no podemos asignar ninguna acción, por lo que $dp[1][0]$ y $dp[1][1] = 0$.

Para $a = 2, 3, 4, 5$, tenemos que:

$a = 2,\ dp[1][2] = max(0, 2 * 200) = 400$

$a = 3,\ dp[1][3] = max(0, 3 * 200) = 600$

$a = 4,\ dp[1][4] = max(0, 4 * 200) = 800$

$a = 5,\ dp[1][5] = max(0, 5 * 200) = 1000$

dp = $\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 400 & 600 & 800 & 1000 & 1000 & 1000 & 1000 & 1000 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$

Para $i = 2$, el mínimo de acciones que se pueden asignar es 3, y el máximo 7. Para $a = 0, 1, 2$, no podemos asignar ninguna acción de parte de la oferta dos.

Para $a = 3, 4, 5, 6, 7$, tenemos que:

$a = 3,\ dp[2][3] = max(dp[1][3],\ dp[1][0] + 3 * 150) = max(600, 450) = 600$

$a = 4,\ dp[2][4] = max(dp[1][4],\ dp[1][0] + 4 * 150) = max(800, 600) = 800$

$a = 5,\ dp[2][5] = max(dp[1][5],\ dp[1][0] + 5 * 150) = max(1000, 750) = 1000$

$a = 6,\ dp[2][6] = 1050$

Se evalúa asignar 3 acciones a la oferta 2: Valor previo: 1000
$dp[2][6] = (600 + 3 * 150) = 1050$

Se evalúa asignar 4 acciones a la oferta 2: Valor previo: 1050

$dp[2][6] = (400 + 4 * 150) = 1000$

Se evalúa asignar 5 acciones a la oferta 2: Valor previo: 1050

$dp[2][6] = (0 + 5 * 150) = 750$

Se evalúa asignar 6 acciones a la oferta 2: Valor previo: 1050

$dp[2][6] = (0 + 6 * 150) = 900$

$a = 7,\ dp[2][7] = 1250$

Se evalúa asignar 3 acciones a la oferta 2: Valor previo: 1000
$dp[2][7] = (800 + 3 * 150) = 1250$

$a = 8,\ dp[2][8] = 1450$

Se evalúa asignar 3 acciones a la oferta 2: Valor previo: 1000
$dp[2][8] = (1000 + 3 * 150) = 1450$

$a = 9,\ dp[2][9] = 1600$

Se evalúa asignar 3 acciones a la oferta 2: Valor previo: 1000
$dp[2][9] = (1000 + 4 * 150) = 1600$

$a = 10,\ dp[2][10] = 1750$

Se evalúa asignar 3 acciones a la oferta 2: Valor previo: 1000
$dp[2][10] = (1000 + 5 * 150) = 1750$

$$\text{dp} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 400 & 600 & 800 & 1000 & 1000 & 1000 & 1000 & 1000 & 1000 \\ 0 & 0 & 400 & 600 & 800 & 1000 & 1050 & 1250 & 1450 & 1600 & 1750 \end{bmatrix}$$

La celda $dp[2][10]$, contiene el valor máximo generado, que consiste en la asignación de 5 acciones para la oferta 1, y 5 acciones para la oferta 2.

Asignaciones: $< 5, 5, 0 >$, Vr: 1750

2.6 Implementación de las soluciones y complejidad temporal

Estrategia por fuerza bruta

La función *auction_brute_force* usa la función recursiva *combine*, la cual explora todas las posibles combinaciones de asignación de acciones para cada oferta. En cada paso, se intenta asignar entre el número mínimo y máximo y se realiza una llamada recursiva para asignar acciones al resto de ofertas. Adicionalmente, se calcula el valor total de cada combinación mediante la función *calculate_value*.

La complejidad temporal está dada por $O(n * (m^n)$, donde *n* es el número de ofertas y *m* es el número máximo de acciones que se pueden asignar a una oferta. Esta complejidad se da debido a que el algoritmo explora para cada una de las ofertas todas las posibles combinaciones, entre el rango de acciones lo que genera un número exponencial y un costo que no es viable para problemas grandes.

Estrategia por programación dinámica

La función *auction_dp* utiliza una matriz *dp*, con dimensiones $(n + 1) * (A + 1)$, donde *n* es el número de ofertas y *A* es el total de acciones disponibles. El llenado de la matriz se realiza iterando sobre cada oferta, y para cada una, sobre la cantidad posible de acciones restantes.

Para cada celda $dp[i][a]$, que representa el valor máximo posible al asignar acciones a las primeras ofertas, se actualiza considerando las posibles asignaciones para la oferta *i*. Una vez que la matriz está llena, el valor óptimo se encuentra en la última celda $dp[n][A]$, y la mejor asignación de acciones se recupera recorriendo la matriz hacia atrás.

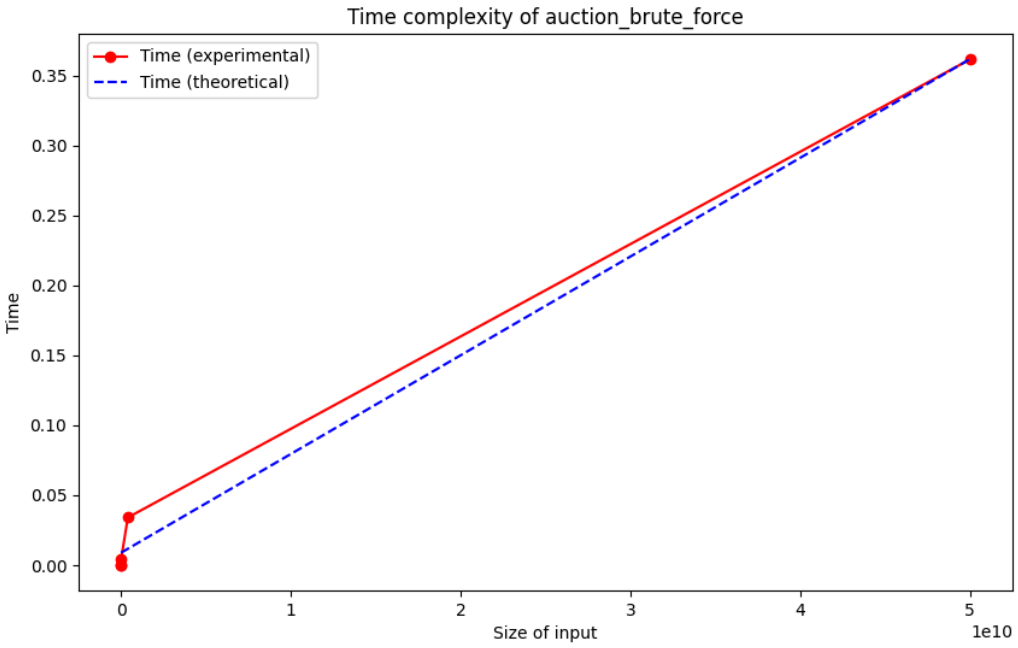
La complejidad temporal es $O(n * A^2)$, debido a que recorre las ofertas y, para cada una, evalúa posibles asignaciones en un rango de hasta A acciones, teniendo en cuenta que en el peor de los casos, alguna de las ofertas, distinta a la del gobierno, tiene como máximo A acciones y como mínimo cero. Aunque el algoritmo es más eficiente que la estrategia por fuerza bruta, puede ser costoso si el valor de A es grande.

Estrategia por programación voraz

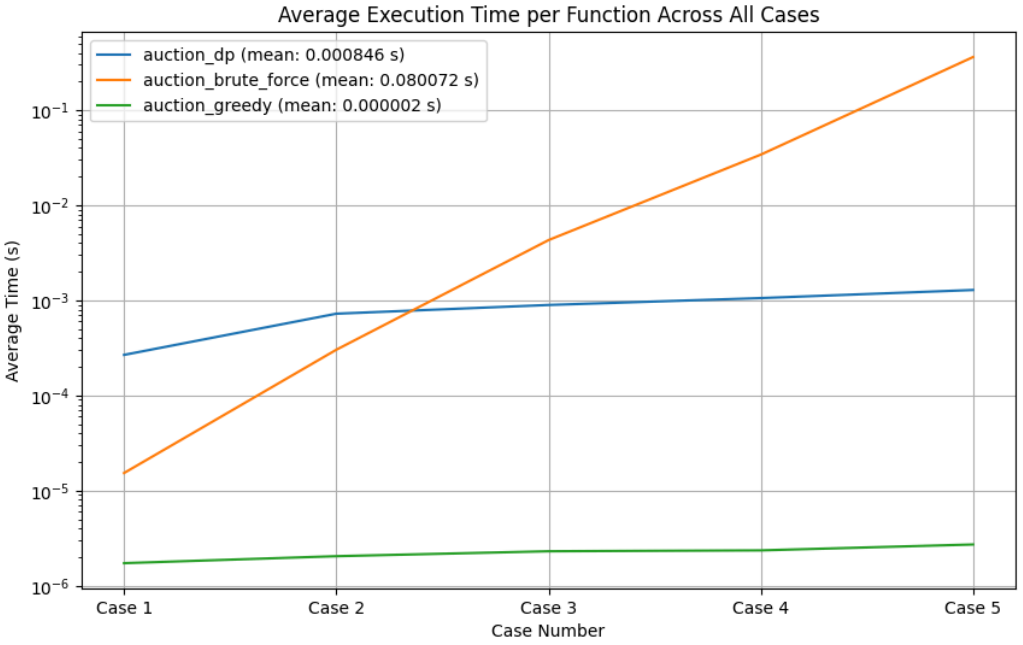
La función *auction_greedy* filtra las ofertas que no cumplen con el precio mínimo y las ordena en orden descendente según su precio. Luego, asigna las acciones disponibles a las ofertas con mayor precio, respetando el rango del mínimo y máximo de acciones que puede aceptar cada una.

La complejidad temporal es $O(n \log n)$, si $n > 1$, de lo contrario la complejidad está dada por $O(n)$, donde n es el número de ofertas válidas. En este caso, el paso más costoso es ordenar las ofertas. Este algoritmo es eficiente y su implementación permite que tome las decisiones más beneficiosas, debido a que el punto siempre será obtener el valor máximo, ordenar las ofertas, asegura que este valor pueda ser obtenido.

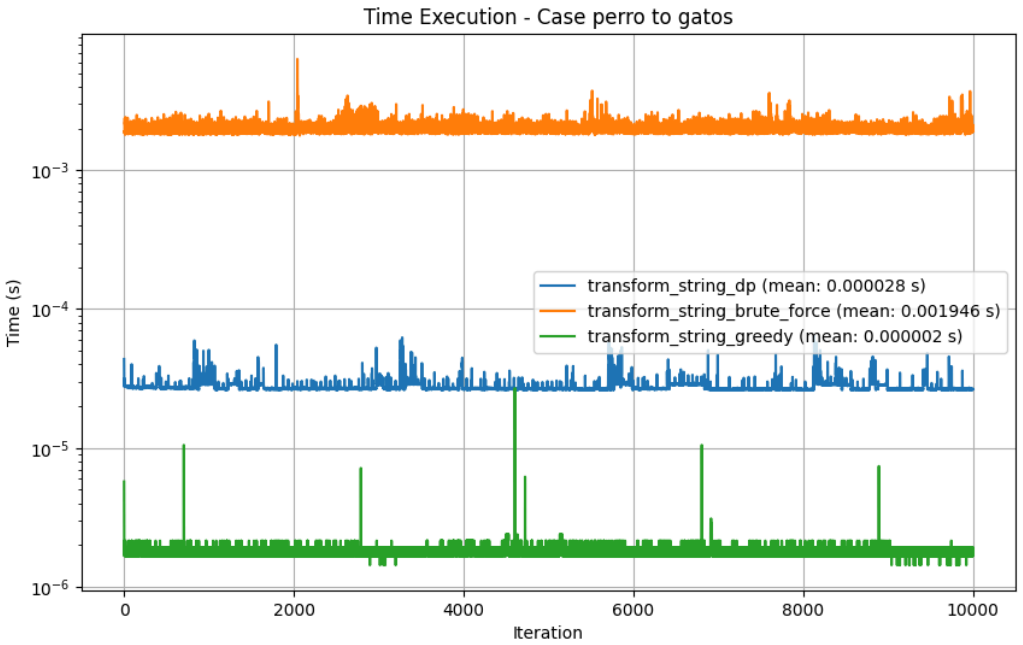
Resultados:



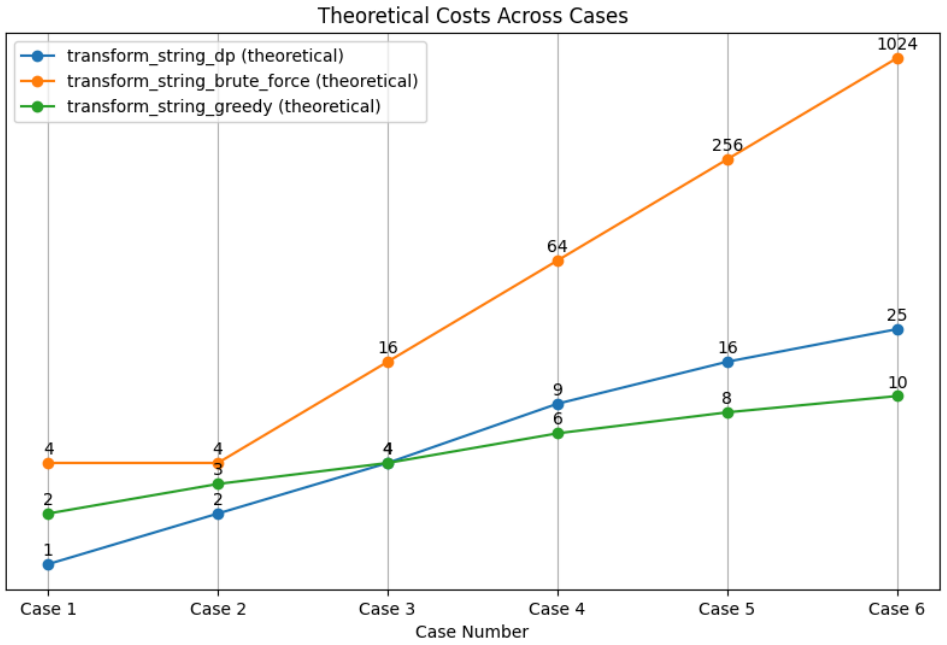
Ambas líneas exhiben una tendencia lineal ascendente, indicando que el tiempo de ejecución del algoritmo aumenta proporcionalmente con el tamaño de la entrada. La proximidad entre la línea experimental y la teórica sugiere que los resultados experimentales se ajustan bien a las predicciones teóricas.



La gráfica muestra que la función "auction_greedy" es la más eficiente en tiempo de ejecución, seguida por "auction_dp". En contraste, "auction_brute_force" es significativamente más lenta. Estos resultados son esenciales para evaluar la eficacia de los algoritmos, especialmente en contextos donde el tiempo de procesamiento es crucial.



"transform_string_greedy" es el algoritmo más eficiente, con un tiempo de ejecución muy bajo, indicando alta optimización para este problema. "transform_string_dp" es menos eficiente que el voraz, pero sigue siendo rápido y útil cuando el enfoque voraz no es aplicable o no garantiza una solución óptima. "transform_string_brute_force" es significativamente más lento y poco práctico para grandes volúmenes de datos o situaciones donde el tiempo de ejecución es crítico.



"transform_string_brute_force": Tiene un costo constante de 4 en los primeros tres casos,

pero aumenta exponencialmente a 1024 en el caso 6, mostrando una alta ineficiencia para casos grandes.

"transform_string_dp": Muestra un aumento gradual y lineal en los costos, de 1 en el caso 1 a 25 en el caso 6, siendo más eficiente que la fuerza bruta.

"transform_string_greedy": Presenta costos generalmente más bajos que la programación dinámica, de 2 en el caso 1 a 10 en el caso 6, destacando como el método más eficiente de los tres.

Para este proyecto, implementamos una versión de un sistema de benchmarking diseñado para evaluar el desempeño de distintos algoritmos en escenarios definido, el propósito principal de esta implementación es medir y analizar cómo se comportan las funciones a medida que incrementamos la complejidad de los casos de prueba, lo que permite obtener datos concretos para comparar los tiempos de ejecución con las complejidades teóricas esperadas, estos resultados serán la base para los análisis y conclusiones del proyecto.

Ahora bien, la clase principal, denominada Benchmark, está diseñada para automatizar este proceso de evaluación, se enfoca en medir el tiempo de ejecución de las funciones a través de múltiples iteraciones para obtener datos “precisos”, posteriormente, genera gráficos que representan el comportamiento temporal de cada algoritmo y cómo varía en función de las características del problema, este comportamiento se complementa con la función adjust_and_graph_times y permite generar un análisis de los tiempos experimentales al ajustarlos a funciones teóricas de complejidad y generar representaciones gráficas que facilitan la interpretación de los resultados.

Este proceso comienza al recibir los tiempos medidos experimentalmente (times) junto con los tamaños de entrada correspondientes (values) y una función teórica (theoretical_func) que modela la complejidad esperada del algoritmo evaluado, el ajuste se realiza utilizando el método de mínimos cuadrados, proporcionado por curve_fit de la librería scipy.optimize, que permite encontrar los parámetros óptimos de la función teórica para que esta se alinee lo más posible con los datos experimentales, estos parámetros ajustados (params) son el resultado del proceso de optimización y representan los coeficientes necesarios para que la función teórica describa mejor los tiempos medidos. Una vez ajustada, la función teórica se evalúa con los tamaños de entrada originales, generando los valores estimados (estimated_times) que se compararán visualmente con los valores experimentales.

La segunda parte del proceso consiste en la generación de un gráfico que muestra la comparación entre los tiempos experimentales y los estimados, en este gráfico, los puntos rojos representan los tiempos medidos, mientras que la línea azul discontinua ilustra la predicción de la función teórica ajustada, este contraste visual permite evaluar qué tan bien se adapta la función teórica a los datos reales, identificando posibles diferencias o confirmando la validez de la aproximación teórica.