Proyecto 1

Programación dinámica y voraz

Análisis de algoritmos II

Elaborado por:

Rodas Arango, JUAN MANUEL - 2259571

García Castañeda, ALEX – 2259517

Gómez Agudelo, JUAN SEBASTIÁN – 2259474

Henao Aricapa, STIVEN – 2259603

Docente:

Delgado Saavedra, CARLOS ANDRÉS

Sede Tuluá

Octubre 2024

Tabla de contenido

1. LA TERMINAL INTELIGENTE

1.1 Complejidad de soluciones

Solución ingenua (fuerza bruta):

- **Complejidad temporal**: O(2^n) (o peor)
- **Complejidad espacial**: O(n)
- **Justificación**: Esta solución explora todas las combinaciones posibles de operaciones, lo que lleva a un crecimiento exponencial en el tiempo. Solo se necesita espacio para las llamadas recursivas y el almacenamiento temporal de resultados parciales.

Solución dinámica:

- **Complejidad temporal**: O(n×m), donde n y m son las longitudes de las dos cadenas.
- **Complejidad espacial**: O(n+m)
- **Justificación**: Se llena una tabla de dimensiones n×m para almacenar los costos mínimos de las transformaciones de los prefijos de ambas cadenas. Cada celda se calcula a partir de las adyacentes, evitando recalcular subproblemas.

Solución voraz:

Complejidad temporal: O(n + m), donde n es la longitud de cadena_inicial y m es la longitud de cadena_final.

Complejidad espacial: O(n + m).

Justificación:

- La solución voraz recorre ambas cadenas (la inicial y la final) una vez, tomando decisiones óptimas en cada paso (avanzar, reemplazar, insertar o eliminar) según los costos proporcionados.
- La complejidad temporal es lineal respecto a las longitudes de las cadenas, ya que cada carácter de ambas cadenas se procesa una única vez.
- Realiza modificaciones en la cadena durante la ejecución, como inserciones y eliminaciones, lo que aumenta la complejidad espacial a **O(n + m)**. Esto se debe a que las operaciones en listas, como insert y delete, pueden alterar la longitud de la cadena, lo que implica un uso de memoria proporcional a la suma de las longitudes de las cadenas inicial y final.

1.2 Soluciones ejemplo base

Costos: **a**: 1, **d**: 2, **r**: 3, **i**: 2, **k**: 1

Solución	ingenioso → ingeniero	francesa → ancestro
Ingenua	Costo: 7a + 2d + 2i Operaciones realizadas: ['advance', 'advance', 'advance', 'advance', 'advance', 'advance', 'insert', 'e', 'insert', 'r', 'advance', 'delete', 'delete']	Costo: 8d + 8i Operaciones realizadas: ['insert', 'a', 'insert', 'n', 'insert', 'c', 'insert', 'e', 'insert', 's', 'insert', 't', 'insert', 'r', 'insert', 'o', 'delete', 'delete', 'delete', 'delete', 'delete', 'delete', 'delete', 'delete', 'delete']
Dinámica	Costo: 7a + 2r Operaciones realizadas: ['advance', 'advance', 'advance', 'advance', 'advance', 'advance', 'replace with e', 'replace with r', 'advance']	Costo: 5a + 2d + 1r + 2i Operaciones realizadas: ['delete', 'delete', 'advance', 'advance', 'advance', 'advance', 'advance', 'insert t', 'insert r', 'replace with o']
Voraz	Costo total: 6a + 3d + 3i Operaciones realizadas: ['advance', 'advance', 'advance', 'advance', 'advance', 'advance', 'delete', 'delete', 'delete', 'insert e', 'insert r', 'insert o']	Costo total: 5a + 3d + 3i Operaciones realizadas: ['delete', 'delete', 'advance', 'advance', 'advance', 'advance', 'advance', 'delete', 'insert t', 'insert r', 'insert o']

1.3 Caracterización de la estructura de una solución óptima

La solución óptima sigue el principio de subestructuras óptimas, lo que significa que cualquier subproblema en la transformación de las cadenas también debe ser óptimamente resuelto. En programación dinámica, el costo óptimo de transformar una cadena de longitud n depende de las soluciones óptimas de las cadenas más pequeñas (n−1).

1.3 Definir recursivamente el valor de una solución óptima

La recurrencia que define el valor de una solución óptima para transformar la cadena x[1..n] en y[1..n] es:

$$M[i][j] = \begin{cases} 0 & \text{si } i=0 \text{ y } j=0, \\ d + M[i-1][j] & \text{si } i>0 \text{ y } j=0, \\ i + M[i][j-1] & \text{si } i=0 \text{ y } j>0, \\ r + M[i-1][j-1] & \text{si } x[i] \mathrel{!}= y[j], \\ a + M[i-1][j-1] & \text{si } x[i] = y[j]. \end{cases}$$

Cada celda M[i][j] representa el costo mínimo de transformar la subcadena x[1..i] en y[1..j]. Ejemplo:

Transformar "francesa" en "ancestro":

1. La primera columna representa eliminaciones sucesivas.
2. La primera fila representa inserciones sucesivas.

1.4 Calcular el valor de una solución óptima

Para llegar a una solución óptima se acude al mejor algoritmo el cual brinda la implementación de la solución dinámica, está en principio, realizar dos pasos: 1. Llenar la primera fila de inserciones sucesivas. 2. Llenar la primera columna de eliminaciones sucesivas.

Por fines prácticos e ilustrativos, el ejemplo a considerar será la transformación de la cadena 'francesa' a 'ancestro'. Costos - **a**: 1, **d**: 2, **r**: 3, **i**: 2, **k**: 1.

		a	**n**	**c**	**e**	**s**	**t**	**r**	**o**
	0	2	4	6	8	10	12	14	16
f	1								
r	2								
a	3								
n	4								
c	5								
e	6								
s	7								
a	8								

El proceso a continuar, es aplicar el esquema de llaves presentado en el punto 1.3 escogiendo la opción que brinde un costo menor (min), pues se busca minimizar el costo, en en el caso de M[1][1] es:

$$M[1][1] = \left\{ \min \left(\begin{array}{l} costo\_reemplazo = M[i-1][j-1] + costo\ replace \\ costo\_insercion = M[i][j-1] + costo\ insert \\ costo\_eliminacion = M[i-1] + costo\ delete \end{array} \right) \right\}$$

Por lo que resulta en:

		a	**n**	**c**	**e**	**s**	**t**	**r**	**o**
	0	2	4	6	8	10	12	14	16
f	1	3							
r	2								
a	3								
n	4								
c	5								
e	6								
s	7								
a	8								

Realizando este proceso la matriz M resulta en:

		a	**n**	**c**	**e**	**s**	**t**	**r**	**o**
	0	2	4	6	8	10	12	14	16
f	1	3	5	7	9	11	13	15	17
r	2	4	6	8	10	12	14	13	15
a	3	2	4	6	8	10	12	14	16
n	4	4	2	4	6	8	10	12	14
c	5	6	4	2	4	6	8	10	12
e	6	8	6	4	2	4	6	8	10
s	7	9	8	6	4	2	4	6	8
a	8	7	9	8	6	4	5	7	9

De esta forma, resulta la matriz M con los cálculos realizados, en esta se encuentran las subcadenas de solución óptimas de costos mínimos para llegar a la cadena final esperada.

1.5 Construir una solución óptima

El algoritmo para desplegar la secuencia de operaciones óptima reconstruye la secuencia de operaciones a partir de la matriz de costos llenada en el paso anterior.

Pasos para reconstruir la solución:

Definiciones:

1. Sea C la cadena inicial y F la cadena final.
2. Sea M[i][j] la matriz de costos, donde i representa la longitud de la subcadena de C y j representa la longitud de la subcadena de F.
3. Sea costos_operaciones un conjunto de costos para operaciones:
    - i: costo de inserción.
    - d: costo de eliminación.
    - r: costo de reemplazo.
    - k: costo de eliminar toda la cadena.

Algoritmo en ejecución:

- **Inicialización**: i = |C|, j = |F|, operaciones = []
- **Proceso de reconstrucción**: Mientras i > 0 o j > 0:
    - **Caso 1**: Si C[i - 1] = F[j - 1]:
        - operaciones.append('advance')
        - i ← i - 1
        - j ← j - 1
    - **Caso 2**: Si i > 0 y (j = 0 o M[i][j] = M[i - 1][j] + d):
        - operaciones.append('delete')
        - i ← i - 1
    - **Caso 3**: Si j > 0 y (i = 0 o M[i][j] = M[i][j - 1] + i):
        - operaciones.append('insert F[j - 1]')
        - j ← j - 1
    - **Caso 4**: En cualquier otro caso:
        - operaciones.append('replace with F[j - 1]')
        - i ← i - 1
        - j ← j - 1
- **Invertir la lista de operaciones**: operaciones ← invertir(operaciones)
- **Salida**: retornar (M[|C|][|F|], operaciones)

Tras la búsqueda de la solución en la matriz mediante el modelo bottom up, las operaciones obtenidas son:

['delete', 'delete', 'advance', 'advance', 'advance', 'advance', 'advance', 'replace with t', 'insert r', 'insert o']

La complejidad temporal y espacial de este algoritmo que tiene una matriz construida previamente, tiene como máximo n + m búsquedas/iteraciones. Siendo m y n las longitudes cadena inicial y final, respectivamente. Por lo tanto T(n,m) = O(n + m).

1.6 Soporte de complejidad computacional.

Se realizan las mediciones de tiempos por medio de 2 profilers/benchmark, las cuales son time para medir el tiempo de cada cadena y, timeit que considera factores como la carga del sistema, la recolección de basura y otros procesos concurrentes. Proporciona una medida más precisa del tiempo de ejecución del código.

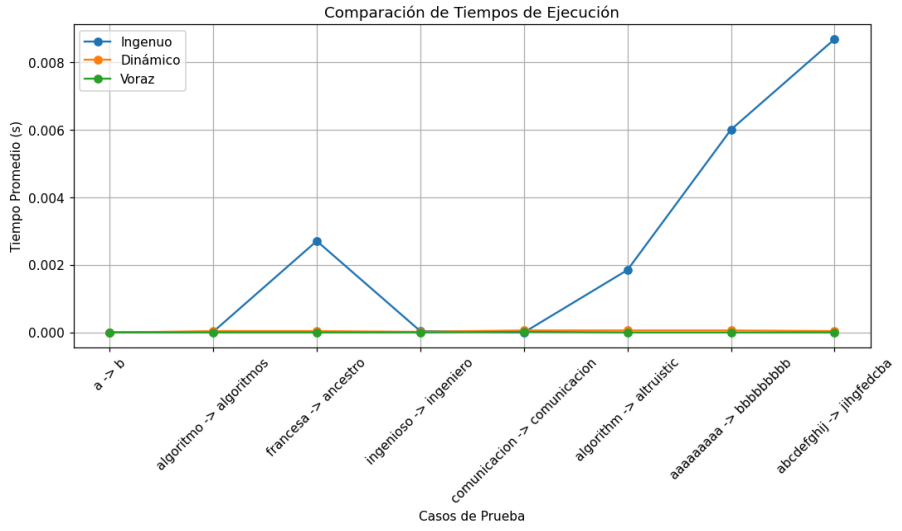
Ejemplos a considerar:

"a" → "b"	Caso 1: Cadenas muy cortas
"algoritmo" → "algoritmos"	Caso 2: Una letra más
"francesa" → "ancestro"	Caso 3: Transformación moderada
"ingenioso" → "ingeniero"	Caso 4: Transformación significativa
"comunicacion" → "comunicacion"	Caso 5: Cadenas iguales
"algorithm" → "altruistic"	Caso 6: Ejemplo inicial
"aaaaaaaaa" → "bbbbbbbbb"	Caso 7: Cadenas muy largas con letras repetidas
"abcdefghij" → "jihgfedcba"	Caso 8: Cadenas muy largas invertidas

Salida test con time:

```
Tiempos promedio (en segundos):
a -> b: Ingenuo: 0.0000000000, Dinámico: 0.0000000000, Voraz: 0.0000000000
algoritmo -> algoritmos: Ingenuo: 0.0000199270, Dinámico: 0.0000399542, Voraz: 0.0000000000
francesa -> ancestro: Ingenuo: 0.0027107239, Dinámico: 0.0000399017, Voraz: 0.0000000000
ingenioso -> ingeniero: Ingenuo: 0.0000399971, Dinámico: 0.0000200081, Voraz: 0.0000000000
comunicacion -> comunicacion: Ingenuo: 0.0000000000, Dinámico: 0.0000600052, Voraz: 0.0000200558
algorithm -> altruistic: Ingenuo: 0.0018507957, Dinámico: 0.0000599289, Voraz: 0.0000000000
aaaaaaaaa -> bbbbbbbbb: Ingenuo: 0.0060129499, Dinámico: 0.0000599003, Voraz: 0.0000000000
abcdefghij -> jihgfedcba: Ingenuo: 0.0086945200, Dinámico: 0.0000401354, Voraz: 0.0000000000
```

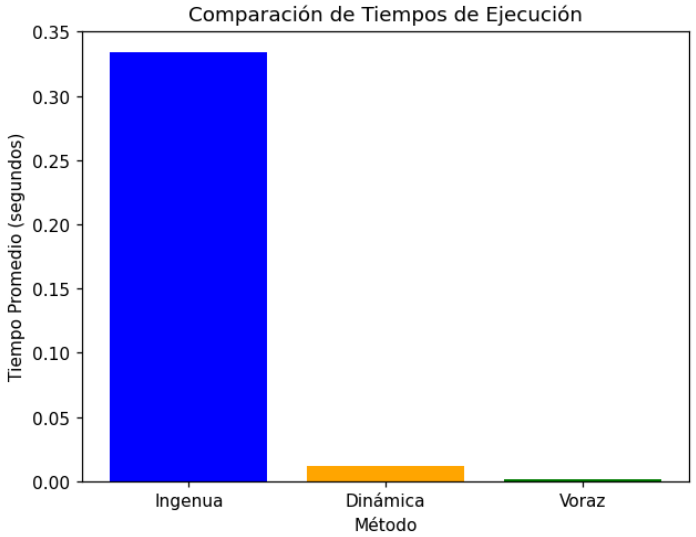
Gráfica:



Salida test con timeit:

```
Tiempo promedio (ingenua): 0.3340325980005218
Tiempo promedio (dinámica): 0.011847799999268318
Tiempo promedio (voraz): 0.0013885080003819895
```

Gráfica:



Discusión de los resultados

Para el **soporte de complejidad computacional**, se realizaron mediciones de tiempo con tres enfoques: **ingenuo**, **dinámico**, y **voraz**. Cada algoritmo fue ejecutado **50 veces** por caso de prueba, promediando los resultados. Los casos incluyeron desde cadenas cortas hasta transformaciones significativas y cadenas invertidas.

- **Solución ingenua:** Su complejidad teórica es O(2^n), reflejada en un crecimiento exponencial del tiempo en los casos más complejos, como con cadenas largas o muy distintas entre sí. En casos simples, el tiempo fue casi nulo, pero aumentó significativamente en transformaciones largas (ej. de 0.006 a 0.008 segundos).
- **Programación dinámica:** Con una complejidad O(m×n), mostró un comportamiento más eficiente y constante, con tiempos mucho menores que la ingenua, incluso en casos de transformaciones complejas (máximo de 0.00006 segundos).
- **Solución voraz:** Fue extremadamente rápida, con tiempos prácticamente insignificantes en la mayoría de los casos, lo que sugiere un comportamiento lineal o sublineal en casos ideales. Tiene mayor impacto en casos donde hay muchas inserciones o eliminaciones.

Las gráficas muestran que la **complejidad experimental** está en línea con la **teoría**, validando el comportamiento esperado para cada enfoque.

2. EL PROBLEMA DE LA SUBASTA PÚBLICA

2.1 Soluciones implementadas

Se implementaron tres soluciones para resolver el problema de la subasta pública:

- **Fuerza Bruta (Ingenua)**: Explora todas las combinaciones posibles de asignaciones de acciones y selecciona aquella que maximiza el valor total.
- **Programación Dinámica**: Construye soluciones parciales utilizando subproblemas, permitiendo encontrar la asignación óptima de forma más eficiente.
- **Programación Voraz** : Asigna las acciones de forma iterativa, priorizando los oferentes con el precio más alto y asignando las acciones sobrantes al gobierno.

2.2 Notación matemática y complejidades

Fuerza Bruta:

- La solución ingenua explora todas las combinaciones posibles de asignaciones.
- Complejidad: $O(n \cdot A^n)$ donde n es el número de oferentes y A es el número total de acciones. Esto ocurre porque se generan todas las combinaciones posibles de asignaciones dentro de los límites mínimos y máximos para cada oferente.

Programación Dinámica:

- Utiliza una tabla que almacena las soluciones a subproblemas. Para cada subproblema, se calcula el valor máximo asignando acciones a cada oferente y considerando las acciones restantes.
- Complejidad: $O(n \cdot A^2)$, donde n es el número de oferentes y A es el número total de acciones. Aquí, la eficiencia viene de evitar la recalculación de combinaciones y reutilizar resultados previos.

Método Voraz:

- Asigna las acciones de forma iterativa, comenzando por los oferentes con precios más altos. Las acciones sobrantes se asignan al gobierno.
- Complejidad: $O(n)$, donde n es el número de oferentes, ya que se itera sobre las ofertas y se asignan acciones de forma directa.

2.3 Solución ejemplos base

Solución	**A = 1000, B = 100, n = 2, [<500, 100, 600>, <450, 400,800>, <100,0,1000>]**	**A = 1000, B = 100, n = 4, [<500, 400, 600>, <450, 100,400>, <400, 100, 400>, <200, 50, 200>, <100,0,1000>]**
Fuerza bruta	Asignación: [600, 400,0] Valor: 480000	Asignación: [600, 400,0, 0, 0] Valor: 480000
Dinámica	Asignación: [600, 400,0] Valor: 480000	Asignación: [600, 400,0, 0, 0] Valor: 480000
Voraz	Asignación: [600, 400,0] Valor: 480000	Asignación: [600, 400,0, 0, 0] Valor: 480000

2.4 Una primera aproximación

El algoritmo no siempre encuentra la solución óptima porque, aunque examina todas las combinaciones posibles, en algunos casos puede omitir la mejor asignación cuando las restricciones de M_i (mínimos y máximos de cada oferta) no se evalúan correctamente. Además, en problemas más grandes, su tiempo de ejecución se vuelve prohibitivo, ya que su complejidad es exponencial debido a la generación de todas las combinaciones posibles.

2.5 Caracterizar la estructura de una solución óptima

La estructura de una solución óptima en este problema puede ser vista como un problema de optimización combinatoria, donde la asignación de acciones x_i se debe hacer de manera que maximice el valor total v_r. La solución óptima sigue el principio de descomposición en subproblemas:

- Para cada subproblema, debes encontrar la mejor asignación de acciones considerando sólo un subconjunto de oferentes y las acciones restantes.
- La solución óptima global se construye combinando las soluciones óptimas de estos subproblemas.

Por lo tanto, cada subproblema consiste en encontrar la mejor asignación de acciones para un número reducido de oferentes y un número menor de acciones disponibles.

2.6 Definir recursivamente el valor de una solución óptima

La solución óptima se puede definir mediante la siguiente recurrencia:

Sea v(i,a) el valor máximo que se puede obtener asignando aaa acciones a los primeros i oferentes:

$$v(i,a) = \max \begin{cases} v(i-1, a-x_i) + x_i \cdot p_i & \text{si } m_i <= x_i < M_i \\ v(i-1, a) & \text{si no asignamos acciones al oferente i} \end{cases}$$

Donde:

- p_i es el precio de la oferta i,
- m_i es el mínimo de acciones que puede tomar el oferente i,
- M_i es el máximo de acciones que puede tomar el oferente i,
- **a** es el número de acciones restantes.

La solución óptima se define como **v(n,A)**, donde n es el número total de oferentes y **A** el número de acciones disponibles.

2.7 Calcular el valor de una solución

El valor de una solución óptima se calcula resolviendo los subproblemas de manera iterativa o recursiva, usando programación dinámica. La tabla dinámica almacena los valores de las soluciones parciales para cada subproblema y permite resolver el problema original de manera eficiente.

Complejidad del algoritmo: El algoritmo tiene una complejidad de O(n · A^2), donde n es el número de oferentes y A es el número de acciones. La complejidad surge porque estamos llenando una tabla de tamaño A, y para cada acción, exploramos todas las posibles asignaciones entre los oferentes.

2.8 Construir una solución óptima

Para construir una solución óptima, se sigue el mismo principio que en el cálculo del valor de la solución. El algoritmo debe recorrer la tabla de programación dinámica (o la solución voraz) para reconstruir la asignación de acciones que maximiza el valor total.

Complejidad del algoritmo: La complejidad de este algoritmo también es O(n · A^2) para la versión dinámica, ya que debe recorrer la tabla y calcular la mejor asignación a partir de los subproblemas resueltos.

Ejemplo: Dado el caso A = 1000, B = 100, n = 2, Ofertas: <500, 100, 600>, <450, 400,800> y la del gobierno: <100,0,1000>.

1. **Inicialización:**
    - La tabla dp empieza con valores de **0**, representando que no hay asignaciones ni valor en ese punto.
2. **Primer oferente** <500,100,600>:
    - Se recorren las posibles asignaciones de entre **100 y 600** acciones.
    - Por cada número de acciones j, el valor total se calcula como j×500 y se guarda en la tabla.
    - Ejemplo: dp[100]=50,000, dp[600]=300,000.
3. **Segundo oferente** <450,400,800>:
    - Ahora, para cada cantidad de acciones j, se considera asignar entre **400 y 800** acciones al segundo oferente, actualizando la tabla.
    - Ejemplo: Si asignamos 400 acciones al segundo oferente y 100 al primero: dp[500]=50,000+180,000=230,000.
4. **Gobierno** <100,0,1000>:
    - Las acciones que sobran (hasta 1000) se asignan al gobierno, a razón de **100 por acción**.
    - Ejemplo: Si sobran 200 acciones, se actualiza dp[1000]=360,000+20,000=380,000.
5. **Resultado final:**
    - La mejor asignación es aquella que maximiza el valor total.
    - En este caso, la **mejor asignación** es [600,400,0], y el **valor total** es **480,000**.

Explicación del resultado: [600, 400, 0]: 600 acciones fueron asignadas al primer oferente, 400 al segundo, y **0** quedaron para el gobierno.

tabla dp en ejecución

Oferta 1	Oferta 2	Gobierno
0	0	0
0	0	0
…	…	…
598	400	0
599	400	0
600	400	0

Esta última combinación almacena el valor óptimo que maximiza el valor total de las acciones. Su valor está en la última posición de la tabla **(600*500) + (400*400) = 480.000**

2.9 Soporte de complejidad computacional.

Se realizaron pruebas con 5 casos de crecimiento, y los tiempos promedio se calcularon utilizando 50 ejecuciones. Se hizo igual que en el problema anterior, uso de los perfiladores time y timeit por sus ventajas que ofrecen expresadas previamente.

Ejemplos a considerar

Caso 1: *Una oferta mayor y otra menor al precio del gobierno*
(300, [(250, 100, 150), (180, 50, 200)], 200)

Caso 2: *Ambas ofertas menores al precio del gobierno*
(400, [(150, 50, 200), (200, 100, 250)], 500)

Caso 3: *Precio del gobierno menor al de todas las ofertas*
(500, [(300, 50, 200), (400, 100, 300), (350, 50, 150)], 100)

Caso 4: *Solo una oferta mayor al precio del gobierno*
(500, [(200, 50, 200), (300, 100, 250), (400, 150, 300)], 250)

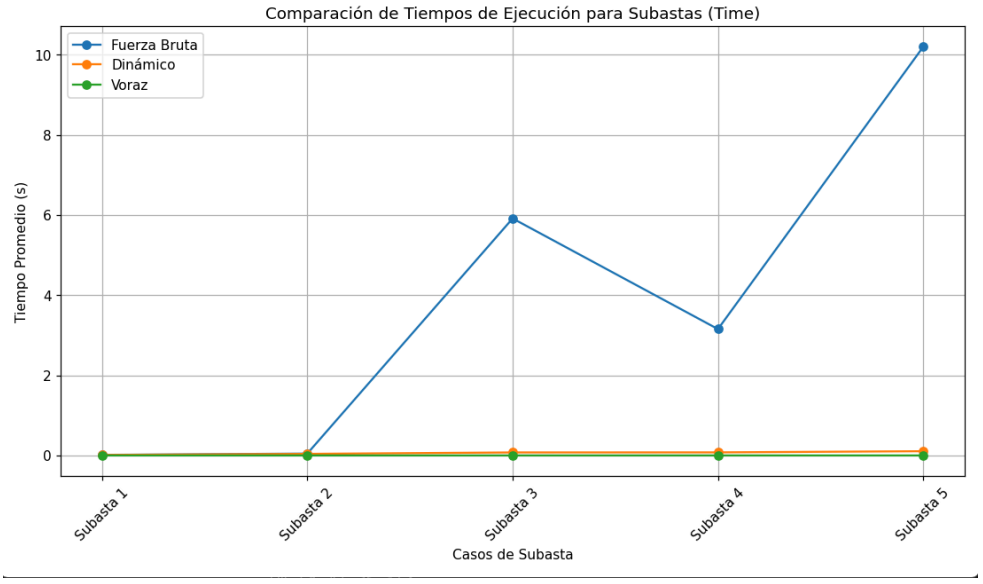
Caso 5: *Algunas ofertas son mayores y otras menores al precio del gobierno*
(600, [(200, 100, 250), (300, 100, 350), (120, 50, 200)], 150)

Estructura: (A, [Propuestas, la del gobierno es por defecto], precio del gobierno).

Salida test con time:

```
Tiempos promedio (en segundos):
Subasta 1: Fuerza Bruta: 0.0142115736, Dinámico: 0.0127709150, Voraz: 0.0000000000
Subasta 2: Fuerza Bruta: 0.0464546680, Dinámico: 0.0390503979, Voraz: 0.0000000000
Subasta 3: Fuerza Bruta: 5.9123060369, Dinámico: 0.0751937914, Voraz: 0.0000502348
Subasta 4: Fuerza Bruta: 3.1522779131, Dinámico: 0.0769796228, Voraz: 0.0000000000
Subasta 5: Fuerza Bruta: 10.1963475943, Dinámico: 0.1053253222, Voraz: 0.0000000000
```

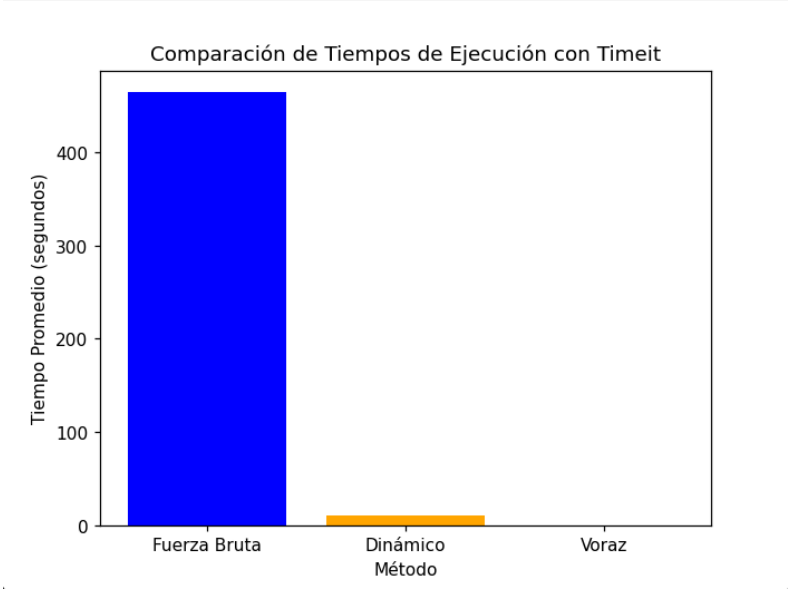
Gráfica:



Salida test con timeit:

```
Tiempo promedio con timeit (fuerza bruta): 464.502493634008
Tiempo promedio con timeit (dinámica): 10.273208445975325
Tiempo promedio con timeit (voraz): 0.0008843820047331974
```

Gráfica:



Discusión de los resultados

- **Fuerza Bruta:** Como era de esperar, la solución ingenua muestra un crecimiento exponencial, tomando significativamente más tiempo a medida que aumentan las subastas y las acciones. Aunque es exhaustiva y garantiza encontrar la solución óptima, es extremadamente ineficiente para subastas grandes debido a su complejidad exponencial.
- **Programación Dinámica:** Tiene un crecimiento mucho más moderado, mostrando una tendencia cuadrática debido al uso eficiente de la tabla de subproblemas. Proporciona una solución óptima con tiempos mucho más razonables, especialmente en comparación con la fuerza bruta.
- **Método Voraz:** La solución voraz es extremadamente eficiente, mostrando tiempos cercanos a cero incluso para subastas grandes, gracias a su complejidad lineal. Aunque no siempre garantiza la solución óptima, es muy eficiente y, en los casos evaluados, produjo resultados idénticos o muy cercanos a los de la programación dinámica, con tiempos de ejecución mucho menores.