Programación Dinámica y Voraz

Integrantes

Alejandro Marin Hoyos
2259353-3743
Manuel Antonio Vidales Duran
2155481-3743
Yessica Fernanda Villa Nuñez
2266301-3743
Karen Jhulieth Grijalba Ortiz
2259623-3743

Análisis y Diseño de Algoritmos II

Universidad del Valle
Tuluá, Valle del Cauca
Octubre 2024

1.Terminal Inteligente

1.1 Entender el problema

- [x] ~~Muestre dos soluciones que permitan transformar la cadena ingenioso en ingeniero.Indique el costo de cada solución.~~
- [x] ~~Muestre dos soluciones que permitan transformar la cadena francesa en ancestro. Indique el costo de cada solución.~~

1.2 Caracterizar la estructura de una solución óptima

- [x] ~~Sea x[1..n] la cadena a transformar en y[1..n], caracterice la estructura de una solución óptima, esto es, indique cómo la solución óptima está compuesta de otras soluciones óptimas.~~

1.3 Definir recursivamente el valor de una solución óptima

- [x] ~~Utilice la recurrencia. Explique con ejemplos cada uno de los valores que calcula para la matriz M. Puede suponer los siguientes costos para las operaciones: a = 1, d = 2, r = 3, i = 2, k = 1.~~

1.4 Calcular el valor de una solución óptima

- [x] ~~Explique cuál es la forma correcta de completar la matriz M. Indique en qué posición de~~
- [x] ~~la matriz quedará la solución al problema original.~~
- [x] ~~Desarrolle un algoritmo para calcular el valor de la solución óptima. Indique su complejidad.~~
- [x] ~~Implemente el algoritmo.~~

1.5 Construir una solución óptima

- [x] ~~Desarrolle un algoritmo que despliegue la secuencia de operaciones óptima que permite transformar una cadena x[1..n] en y[1..n]. Indique su complejidad.~~

- ☑ ~~Implemente el algoritmo. El programa debe permitir modificar los costos de las operaciones e ingresar las cadenas x y y.~~

1.1 Entender el problema

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

  Solución 1: Estrategia por fuerza bruta.

  El enfoque consistió en probar todas las posibles secuencias de operaciones (como **avance**, **delete**, **replace**, **insert**, **kill**) para convertir **ingenioso** en **ingeniero**. Luego, se evalúa la mejor secuencia de operaciones y se halló el costo total.

Para este ejemplo asignamos los siguientes costos para las operaciones: a = 1, d = 2, r = 3, i = 2, k = 1.

```
----------------------FUERZA BRUTA----------------------
Fuerza Bruta - Transformar 'ingenioso' en 'ingeniero':
Costo total: 12
Mejor secuencia de operaciones: ['advance', 'advance', 'advance', 'advance', 'advance'
, 'advance', 'replace with e', 'replace with r']
```

a= 6 y r=2
Cálculo del costo: 6+2(3)=12

La determinación del costo de transformación requirió la evaluación de todas las secuencias posibles de operaciones, lo que resulta extremadamente costoso en términos computacionales cuando las cadenas o operaciones son más complejas.

Solución 2: Estrategia de programación dinámica.
La programación dinámica optimiza el cálculo mediante la memorización y la división del problema en subproblemas más pequeños.

Para este ejemplo asignamos los siguientes costos para las operaciones: a = 1, d = 2, r = 3, i = 2, k = 1.

```
----------------------PROGRAMACION DINAMICA----------------------
Programación Dinámica - Transformar 'ingenioso' en 'ingeniero':
Costo total: 12
Secuencia de operaciones: ['advance', 'advance', 'advance', 'advance', 'advance', 'adv
ance', 'kill', 'insert e', 'insert r', 'advance']
```

a=7, i= 2, y k=1

Cálculo del costo: 7+2(2)+1 = 12

```
Matriz de costos (dp)
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]

[2, 1, 3, 5, 7, 9, 11, 13, 15, 17]

[4, 2, 2, 4, 6, 8, 10, 12, 14, 16]

[6, 2, 3, 3, 5, 7, 9, 11, 13, 15]

[8, 2, 3, 4, 4, 6, 8, 10, 12, 14]

[10, 2, 3, 4, 5, 5, 7, 9, 11, 13]

[12, 2, 3, 4, 5, 6, 6, 8, 10, 12]

[14, 2, 3, 4, 5, 6, 7, 9, 11, 11]

[16, 2, 3, 4, 5, 6, 7, 9, 11, 12]

[18, 2, 3, 4, 5, 6, 7, 9, 11, 12]
```

Matriz de Costos: El algoritmo de programación dinámica construye una matriz donde compara los caracteres de "ingenioso" y "ingeniero" en cada paso. En cada celda de la matriz, evalúa si es más barato **advance**, **insert** o **kill**. Estas decisiones se basan en los costos asociados a cada operación.

La programación dinámica permite calcular el costo mínimo de esta transformación, haciendo un seguimiento eficiente de todas las posibles secuencias de operaciones (**advance**, **insert**, **kill**) y eligiendo la de menor costo para cada subproblema.

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

Solución 1: Estrategia por fuerza bruta.

El enfoque consistió en probar todas las posibles secuencias de operaciones (como **avance**, **delete**, **replace**, **insert**, **kill**) para convertir la cadena **francesa** en **ancestro**. Luego, se evalúa la mejor secuencia de operaciones y se halló el costo total.

Para este ejemplo asignamos los siguientes costos para las operaciones: a = 1, d = 2, r = 3, i = 2, k = 1.

```
----------------------FUERZA BRUTA----------------------
Fuerza Bruta - Transformar 'francesa' en 'ancestro':
Costo total: 16
Mejor secuencia de operaciones: ['delete', 'delete', 'advance', 'advance', 'advance',
'advance', 'advance', 'insert t', 'insert r', 'replace with o']
```

a=5, d=2, r=1 y i=2
Cálculo del costo: 5+2(2)+3+2(2)= 16

Solución 2: Estrategia de programación dinámica.
Este método facilita la solución eficaz de problemas complejos de manera eficiente al:

- Determinar y solucionar subproblemas que se repiten.
- Guardar los hallazgos de estos subproblemas para prevenir cálculos repetitivos.
- Emplear las soluciones guardadas para elaborar la solución al problema inicial de forma progresiva.

La programación dinámica disminuye considerablemente el tiempo de ejecución en comparación con métodos más sencillos como la fuerza bruta, particularmente en situaciones con una estructura repetitiva o recursiva.

Para este ejemplo asignamos los siguientes costos para las operaciones: a = 1, d = 2, r = 3, i = 2, k = 1.

```
----------------------PROGRAMACION DINAMICA----------------------
Programación Dinámica - Transformar 'francesa' en 'ancestro':
Costo total: 14
Secuencia de operaciones: ['insert a', 'kill', 'advance', 'advance', 'advance', 'advan
ce', 'insert t', 'insert r', 'replace with o']
```

a=4, r=1, i=3, k=1
Cálculo del costo: 4+3+3(2)+1= 14

```
Matriz de costos (dp)
[0, 2, 4, 6, 8, 10, 12, 14, 16]

[2, 3, 5, 7, 9, 11, 13, 15, 17]

[4, 3, 5, 7, 9, 11, 13, 14, 16]

[6, 3, 5, 7, 9, 11, 13, 15, 17]

[8, 3, 4, 6, 8, 10, 12, 14, 16]

[10, 3, 5, 5, 7, 9, 11, 13, 15]

[12, 3, 5, 6, 6, 8, 10, 12, 14]

[14, 3, 5, 6, 7, 7, 9, 11, 13]

[16, 3, 5, 6, 7, 8, 10, 12, 14]
```

Esta matriz nos permite reconstruir la secuencia óptima de operaciones para transformar una cadena en otra, siguiendo el camino más barato desde la esquina inferior derecha hasta la esquina superior izquierda.

1.2 Caracterizar la estructura de una solución óptima

			1	2	3	4	5	6	7	8	9
			i	n	g	e	n	i	e	r	o
		0	2	4	6	8	10	12	14	16	18
1	i	2	1	3	5	7	9	11	13	15	17
2	n	4	2	2	4	6	8	10	12	14	16
3	g	6	2	3	3	5	7	9	11	13	15
4	e	8	2	3	4	4	6	8	10	12	14
5	n	10	2	3	4	5	5	7	9	11	13
6	i	12	2	3	4	5	6	6	8	10	12
7	o	14	2	3	4	5	6	7	9	11	11
8	s	16	2	3	4	5	6	7	9	11	12
9	o	18	2	3	4	5	6	7	9	11	12

La composición de una solución óptima para transformar x[1..n] en y[1..n] se compone utilizando el principio de optimización:

Para cada posición (i,j), la solución óptima se deriva de seleccionar el mínimo costo entre las siguientes subestructuras óptimas:

a) Si x[i] = y[j]:

- La solución óptima incluye hacer advance (costo 1).
- Más la solución óptima del subproblema para transformar x[i+1..n] en y[j+1..n].

b) Si x[i] ≠ y[j], la solución óptima será el mínimo entre:

1. **Replace:**

- Reemplazar el carácter actual (costo 3)
- Más la solución óptima para x[i+1..n] en y[j+1..n]

2. **Delete:**

- Eliminar el carácter actual (costo 2)
- Más la solución óptima para x[i+1..n] en y[j..n]

3. **Insert:**

- Insertar el carácter necesario (costo 2)
- Más la solución óptima para x[i..n] en y[j+1..n]

4. **Kill:**

- Eliminar desde la posición actual hasta el final (costo 1)
- Más la solución óptima para x[1..k] en y[1..n], donde k < i

Esta estructura garantiza que la solución óptima global se construye a partir de soluciones óptimas de subproblemas más pequeños.

Solución de la matriz M

Primera Fila (Fila 0)

Esta fila representa el costo de transformar una cadena vacía `""` en los primeros caracteres de la cadena destino. El único tipo de operación que puedes hacer aquí es **insertar** caracteres, porque partimos de una cadena vacía.

Los costos de inserción están acumulados de la siguiente manera:

- `dp[0][0] = 0`: Transformar una cadena vacía en otra cadena vacía tiene un costo 0.
- `dp[0][1] = 2`: Insertar el primer carácter del destino ("i") cuesta 2 (costo de inserción).
- `dp[0][2] = 4`: Insertar los dos primeros caracteres del destino ("in") cuesta 2 + 2 = 4.
- `dp[0][3] = 6`: Insertar los tres primeros caracteres del destino ("ing") cuesta 2 + 2 + 2 = 6.
- Y así sucesivamente, cada valor en la primera fila se incrementa en 2 (el costo de insertar un carácter), acumulando el costo total para cada longitud de prefijo del destino.

Primera Columna (Columna 0)

Esta columna representa el costo de transformar la cadena origen en una cadena vacía `""`. La única operación que puedes hacer aquí es **eliminar** caracteres, porque debes vaciar la cadena origen.

Los costos de eliminación se calculan de la siguiente manera:

- `dp[0][0] = 0`: Transformar una cadena vacía en otra cadena vacía tiene un costo 0.
- `dp[1][0] = 2`: Eliminar el primer carácter de la cadena origen ("i") cuesta 2 (costo de eliminación).
- `dp[2][0] = 4`: Eliminar los dos primeros caracteres de la cadena origen ("in") cuesta 2 + 2 = 4.
- `dp[3][0] = 6`: Eliminar los tres primeros caracteres de la cadena origen ("ing") cuesta 2 + 2 + 2 = 6.
- Y así sucesivamente, cada valor en la primera columna se incrementa en 2 (el costo de eliminar un carácter), acumulando el costo total para vaciar la cadena origen.

Secuencia de operaciones para transformar la cadena (ingenioso) en (ingeniero):

advance (Fila 1, Columna 1): La letra "i" es igual en ambas cadenas.

advance (Fila 2, Columna 2): La letra "n" es igual en ambas cadenas.

advance (Fila 3, Columna 3): La letra "g" es igual en ambas cadenas.

advance (Fila 4, Columna 4): La letra "e" es igual en ambas cadenas.

advance (Fila 5, Columna 5): La letra "n" es igual en ambas cadenas.

advance (Fila 6, Columna 6): La letra "i" es igual en ambas cadenas.

kill (Fila 7, Columna 6): Eliminamos la "o" en "ingenioso".

insert e (Fila 7, Columna 7): Insertamos la letra "e" de "ingeniero".

insert r (Fila 8, Columna 8): Insertamos la letra "r" de "ingeniero".

advance (Fila 9, Columna 9): La letra "o" es igual en ambas cadenas.

La matriz refleja en cada celda el costo mínimo para transformar el prefijo de la cadena origen hasta esa posición en el prefijo correspondiente de la cadena destino, considerando todas las operaciones posibles (advance, delete, replace, insert, kill) y sus costos respectivos.

1.3 Definir recursivamente el valor de una solución óptima.

			1	2	3	4	5	6	7	8	9
			i	n	g	e	n	i	e	r	o
		0	2	4	6	8	10	12	14	16	18
1	i	2	1	3	5	7	9	11	13	15	17
2	n	4	2	2	4	6	8	10	12	14	16
3	g	6	2	3	3	5	7	9	11	13	15
4	e	8	2	3	4	4	6	8	10	12	14
5	n	10	2	3	4	5	5	7	9	11	13
6	i	12	2	3	4	5	6	6	8	10	12
7	o	14	2	3	4	5	6	7	9	11	11
8	s	16	2	3	4	5	6	7	9	11	12
9	o	18	2	3	4	5	6	7	9	11	12

Costos para las operaciones: **a = 1, d = 2, r = 3, i = 2, k = 1.**

Fila 0 [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]

- [0]: Caso base - ningún carácter que transformar
- [2]: Insertar 'i' (costo=2)
- [4]: Insertar 'n' (2+2=4)
- [6]: Insertar 'g' (4+2=6)
- [8]: Insertar 'e' (6+2=8)
- [10]: Insertar 'n' (8+2=10)
- 12: Insertar 'i' (10+2=12)
- [14]: Insertar 'e' (12+2=14)
- [16]: Insertar 'r' (14+2=16)
- [18]: Insertar 'o' (16+2=18)

Fila 1 [2, 1, 3, 5, 7, 9, 11, 13, 15, 17]

- [2]: Borrar 'i' (costo=2)
- [1]: Advance por 'i' igual (1)
- [3]: Insert 'n' (1+2=3)
- [5]: Insert 'g' (3+2=5)
- [7]: Insert 'e' (5+2=7)
- [9]: Insert 'n' (7+2=9)
- [11]: Advance por 'i' (9+2=11)
- [13]: Insert 'e' (11+2=13)
- [15]: Insert 'r' (13+2=15)
- [17]: Insert 'o' (15+2=17)

Fila 2 [4, 2, 2, 4, 6, 8, 10, 12, 14, 16]

- [4]: Borrar 'in' (2x2=4)
- [2]: Kill desde 'n' (1+1=2)
- [2]: Advance por 'in' (1+1=2)
- [4]: Insert 'g' (2+2=4)
- [6]: Insert 'e' (4+2=6)
- [8]: Insert 'n' (6+2=8)
- [10]: Insert 'i' (8+2=10)
- [12]: Insert 'e' (10+2=12)
- [14]: Insert 'r' (12+2=14)
- [16]: Insert 'o' (14+2=16)

Fila 3 [6, 2, 3, 3, 5, 7, 9, 11, 13, 15]

- [6]: Borrar 'ing' (3×2=6)
- [2]: Kill desde 'g' (1+1=2)
- [3]: Kill (2+1=3)
- [3]: Advance (2+1=3)
- [5]: Insert 'e' (3+2=5)
- [7]: Insert 'n' (5+2=7)
- [9]: Insert 'i' (7+2=9)
- [11]: Insert 'e' (9+2=11)

- [13]: Insert 'r' (11+2=13)
- [15]: Insert 'o' (13+2=15)

Fila 4 [8, 2, 3, 4, 4, 6, 8, 10, 12, 14]

- [8]: Borrar 'inge' (4×2=8)
- [2]: Kill desde 'e' (1+1=2)
- [3]: kill (2+1=3)
- [4]: Kill (3+1=4)
- [4]: Advance (3+1=4)
- [6]: Insert 'n' (4+2=6)
- [8]: Insert 'i' (6+2=8)
- [10]: Insert 'e' (8+2=10)
- [12]: Insert 'r' (10+2=12)
- [14]: Insert 'o' (12+2=14)

Fila 5 [10, 2, 3, 4, 5, 5, 7, 9, 11, 13]

- [10]: Borrar 'ingen' (5×2=10)
- [2]: Kill desde 'n' (1+1=2)
- [3]: Advance (2+1=3)
- [4]: kill (3+1=4)
- [5]= kill (4+1=5)
- [5]: Advance por 'ingen' (4+1=5)
- [7]: Insert 'i' (5+2=7)
- [9]: Insert 'e' (7+2=9)
- [11]: Insert 'r' (9+2=11)
- [13]: Insert 'o' (11+2=13)

Fila 6 [12, 2, 3, 4, 5, 6, 6, 8, 10, 12]

- 12: Borrar 'ingeni' (6×2=12)
- [2]: Kill desde 'i' (1+1=2)
- [3]: Kill (2+1=3)
- [4]: Kill (3+1=4)
- [5]: Kill (4+1=5)
- [6]: kill (5+1=6)
- [6]: Advance por 'ingeni' (5+1=6)
- [8]: Insert 'e' (6+2=8)
- [10]: Insert 'r' (8+2=10)
- [12]: Insert 'o' (10+2=12)

Fila 7 [14, 2, 3, 4, 5, 6, 7, 9, 11, 11]

- [14]: Borrar 'ingenio' (7×2=14)
- [2]: Kill desde 'o' (1+1=2)
- [3]: kill (2+1=3)
- [4]: kill (3+1=4)
- [5]:kill (4+1=5)
- [6]: kill (5+1=6)
- [7]: kill (6+1=7)
- [9]: Insert 'e' (7+2=9)
- [11]: Insert 'r' (9+2= 11)
- [11]: Insert 'o' (10+1=11)

Fila 8 [16, 2, 3, 4, 5, 6, 7, 9, 11, 12]

- [16]: Borrar 'ingenios' (8×2=16)
- [2]: Kill desde 's' (1+1=2)
- [3]: kill (2+1=3)
- [4]: kill (3+1=4)
- [5]: kill (4+1=5)
- [6]: kill (5+1=6)
- [7]: Kill (6+1=7)
- [9]: Insert 'e' (7+2=9)
- [11]: Insert 'r' (9+2=11)
- [12]: kill (11+1=12)

Fila 9 [18, 2, 3, 4, 5, 6, 7, 9, 11, 12]

- [16]: Borrar 'ingenios' (8×2=16)
- [2]: Kill desde 's' (1+1=2)
- [3]: kill (2+1=3)
- [4]: kill (3+1=4)
- [5]: kill (4+1=5)
- [6]: kill (5+1=6)
- [7]: Kill (6+1=7)
- [9]: Insert 'e' (7+2=9)
- [11]: Insert 'r' (9+2=11)
- [12]: Adavance (11+1=12)

1.4 Calcular el valor de una solución óptima.

- El método adecuado para completar la matriz **M** en un algoritmo de programación dinámica, para el problema de transformaciones de cadenas, consiste en un proceso iterativo donde se evalúan las posibles operaciones (**avance**, **delete**, **replace**, **insert**, **kill**) en cada celda de la matriz. En cada posición **M[i][j]**, se almacena el costo mínimo para transformar los primeros **i** caracteres de la cadena origen en los primeros **j** caracteres de la cadena destino.

La solución al problema original tiene como objetivo determinar el costo mínimo para cambiar completamente la cadena de origen a la cadena de destino, se puede encontrar en la esquina inferior derecha de la matriz de costos. Esto se puede expresar de la siguiente manera:

solución = **M[longitud de la cadena de origen][longitud de la cadena de destino].**

Origen: 'ingenioso'
Destino: 'ingeniero'
La longitud de la cadena origen **m** es 9, y la longitud de la cadena destino **n** es 9. La solución al problema se encuentra en la posición **M[9][9] .**

			1	2	3	4	5	6	7	8	9
			i	n	g	e	n	i	e	r	o
		0	2	4	6	8	10	12	14	16	18
1	i	2	1	3	5	7	9	11	13	15	17
2	n	4	2	2	4	6	8	10	12	14	16
3	g	6	2	3	3	5	7	9	11	13	15
4	e	8	2	3	4	4	6	8	10	12	14
5	n	10	2	3	4	5	5	7	9	11	13
6	i	12	2	3	4	5	6	6	8	10	12
7	o	14	2	3	4	5	6	7	9	11	11
8	s	16	2	3	4	5	6	7	9	11	12
9	o	18	2	3	4	5	6	7	9	11	12

Complejidad:

def calcular_solucion_optima(origen, destino):

m, n = len(origen), len(destino)

T(n) = O(1)

Porque se están asignando las longitudes m y n de las cadenas.

dp = [[float('inf')] * (n + 1) for _ in range(m + 1)]

operations = [[None] * (n + 1) for _ in range(m + 1)] # Para almacenar las operaciones

La matriz dp tiene (m+1)×(n+1) celdas, y cada celda requiere espacio constante para almacenar `float('inf')`.

La matriz `operations` también tiene (m+1)×(n+1) celdas, y cada una almacena `None`, lo cual también ocupa espacio constante.

Costo = (m+1)×(n+1)=mxn+m+n+1

El término m · n crece mucho más rápido que los otros términos Los términos (m, n, y 1) se consideran insignificantes, ya que se vuelven relativamente pequeños frente a m · n.

T(n)= O(m×n)

```
# Inicializar la matriz dp

dp[0][0] = 0

operations[0][0] = []
```

T(n) = O(1) porque Inicializa lista vacía de operaciones para el caso base.

```
for i in range(1, m + 1): # Inicializar la primera columna
```

Costo = m y tiene m iteraciones válidas.

```
    dp[i][0] = dp[i - 1][0] + COSTOS['delete'] # Costo de eliminar
```

Costo = 1 Porque inicializa la matriz dp con el costo eliminar.

```
    operations[i][0] = operations[i - 1][0] + ['delete'] # Operación de eliminar
```

T(n)= 1

Costo = 1 Porque inicializa la matriz operations con la Operación de eliminar.

T(n)= m+1+1

T(n)= O(m)

La notación Big-O ignora constantes y términos menores en términos de crecimiento.

```
    for j in range(1, n + 1): # Inicializar la primera fila
```

Costo = n y tiene n iteraciones válidas.

```
        dp[0][j] = dp[0][j - 1] + COSTOS['insert'] # Costo de insertar
```

Costo = 1 Porque inicializa la matriz dp con el costo de insertar.

```
        operations[0][j] = operations[0][j - 1] + [f'insert {destino[j-1]}'] # Operación de insertar
```

Costo = 1 Porque inicializa la matriz operations con la Operación de insertar.

T(n)= n+1+1

T(n)= O(n)

```
    # Llenar la matriz
    for i in range(1, m + 1):
```

Costo = m.

```
        for j in range(1, n + 1):
```

Costo = n.

```
            if origen[i - 1] == destino[j - 1]:
                dp[i][j] = dp[i - 1][j - 1] + COSTOS['advance']
                operations[i][j] = operations[i - 1][j - 1] + ['advance']
                print(f"dp[{i}][{j}] = {dp[i][j]} (advance)")
```

Compara los caracteres correspondientes y si son iguales, aplica operación advance.

Costo = 1.

T(n) = mxn + 1

T(n) =O (mxn)

```
else:
    # Reemplazar
    if dp[i][j] > dp[i - 1][j - 1] + COSTOS['replace']:
        dp[i][j] = dp[i - 1][j - 1] + COSTOS['replace']
        operations[i][j] = operations[i - 1][j - 1] + [f'replace with {destino[j-1]}']
        print(f"dp[{i}][{j}] = {dp[i][j]} (replace)")
    # Insertar
    if dp[i][j] > dp[i][j - 1] + COSTOS['insert']:
        dp[i][j] = dp[i][j - 1] + COSTOS['insert']
        operations[i][j] = operations[i][j - 1] + [f'insert {destino[j-1]}']
        print(f"dp[{i}][{j}] = {dp[i][j]} (insert)")

    # Eliminar
    if dp[i][j] > dp[i - 1][j] + COSTOS['delete']:
        dp[i][j] = dp[i - 1][j] + COSTOS['delete']
        operations[i][j] = operations[i - 1][j] + ['delete']
        print(f"dp[{i}][{j}] = {dp[i][j]} (delete)")
```

Evalúa y aplica operaciones de reemplazo, inserción y eliminación debido a que si el condicional se cumple entraría a ejecutar alguna de estas operaciones. El costo también está asociado a las asignaciones a las matrices que componen dichas decisiones.

Costo: 1

```
        # Considerar la operación "kill"
        for k in range(i):
```

Costo: m

```
            if dp[i][j] > dp[k][j] + COSTOS['kill']:
```

Costo:1

```
                dp[i][j] = dp[k][j] + COSTOS['kill']
                operations[i][j] = operations[k][j] + ['kill']
                print(f"dp[{i}][{j}] = {dp[i][j]} (kill)"
```

Evalúa la operación kill para todas las posiciones anteriores. Donde k representa el número de iteraciones del bucle kill en cada celda.

Por eso decimos que "la operación kill añade O(m)" a cada celda, porque para cada posición (i,j) en la matriz, estamos haciendo hasta m operaciones adicionales con el bucle k.

Costo: m+1

T(n)= O(m)

Costo total de los 3 bucles

Costo: (**mxn)m +1**

T(n) = O (m^2xn)

```
#imprimir el costo dp
costo = calcular_costo(operations[m][n])
print("Costo de la transformacion: ", costo) # Aqui tengo el costo de las operaciones

return costo, operations[m][n], dp   # Retorna el costo y la lista de operaciones
```

Costo total:

T(n): 1 + (mxn) + m + n + (m^2xn)

T(n): O(m^2xn)

1.5 Construir una solución óptima.

```
def algoritmo_voraz(x, y, costos=COSTOS):
    m, n = len(x), len(y)
```

Costo: 1 Porque se están asignando las longitudes m y n de las cadenas.

```
    i, j = 0, 0
```

Costo: 1 Porque asignamos longitudes en i y j

```
    operaciones = []
```

Costo: 1 Creamos la lista

```
    costo_total = 0
```

Costo: 1 Inicializamos la variable.

Costo total: 1+1+1+1

T(n)= O(1)

```
while i < m or j < n:
```

Costo: m + n

El bucle avanza a través de los índices i y j, que recorren las cadenas x y y. En cada iteración, se incrementa i,j o ambos, lo que hace que el número total de iteraciones esté limitado por la suma de m y n de las longitudes de x y y.

```
    if i < m and j < n and x[i] == y[j]:
        # Si los caracteres coinciden, avanzamos
        operaciones.append('advance')
        costo_total += costos['advance']
        i += 1
        j += 1
    elif i < m and j < n:
        # Si no coinciden, hacemos un reemplazo
        operaciones.append(f'replace {x[i]} with {y[j]}')
        costo_total += costos['replace']
        i += 1
        j += 1
    elif i < m:
        # Si hay más caracteres en x, eliminamos
        operaciones.append(f'delete {x[i]}')
        costo_total += costos['delete']
        i += 1
```

```
        elif j < n:
            # Si hay más caracteres en y, insertamos
            operaciones.append(f'insert {y[j]}')
            costo_total += costos['insert']
            j += 1
```

Costo : 1

Todas las operaciones dentro son O(1)

Se ejecuta una sola rama de los if/elif

Cada iteración tiene complejidad O(1)

```
    return costo_total, operaciones
```

Costo total: 1 + m + n + 1

T(n)= O(m+n)

2. EL PROBLEMA DE LA SUBASTA PÚBLICA

2.1 Entender el problema

- ☐ Muestre dos asignaciones de las acciones para A = 1000, B = 100, n = 2, la oferta <500, 100, 600 >, la oferta < 450, 400, 800 > y la oferta del gobierno < 100, 0, 1000 >. Indique el valor vr para la solución.

2.2 Una primera aproximación

Considere el algoritmo que lista las posibles asignaciones:

$$< y_1, y_2, \ldots, \sum_{i=1}^{n-1} y_i = A >$$

donde yi ∈ {0, Mi }, luego calcula el vr para cada asignación y selecciona la mejor.

- ☐ Utilice el algoritmo anterior para la entrada A = 1000, B = 100, n = 4, < 500, 400, 600 >, < 450, 100, 400 >, < 400, 100, 400 >, < 200, 50, 200 >, la oferta del gobierno < 100, 0, 1000 >.
- ☐ Indique si el algoritmo anterior siempre encuentra la solución óptima.

2.3 Caracterizar la estructura de una solución óptima.

- ☐ Caracterice la estructura de una solución óptima, esto es, indique la forma de los subproblemas y cómo la solución óptima la componen.

2.4 Definir recursivamente el valor de una solución óptima

- ☐ Muestre la recurrencia que define el costo de la solución óptima.

2.5 Calcular el valor de una solución.

- ☐ Por medio de un algoritmo, muestre cómo se calcula el costo de la solución óptima a través de subproblemas hasta llegar a dar la solución del problema original.
- ☐ Indique la complejidad del algoritmo.
- ☐ Implemente el algoritmo.

2.6 Construir una solución óptima

- ☐ Desarrolle un algoritmo que permita conocer la asignación de acciones X =< x1 , x2 , ..., xn , xn+1 > tal que vr(X) sea máximo. Indique su complejidad.
- ☐ Implemente el algoritmo. El programa debe permitir ingresar los valores para A, B, n y cada oferta < pi , mi , Mi >.

2.1 Entender el problema

Transformar las ofertas dadas en asignaciones óptimas:

Parámetros iniciales:

- A=1000: Número total de acciones.
- B=100: Precio mínimo por acción.
- n=2: Número de ofertas.
- Ofertas: <500,100,600>,<450,400,800>,<100,0,1000> (oferta del gobierno).

Solución 1: Estrategia por Fuerza Bruta

Descripción: La estrategia de fuerza bruta explora todas las posibles combinaciones de asignaciones que cumplan con las restricciones dadas ($m_i \leq x_i \leq M_i$) para maximizar el valor total vr(X).

Asignación:

1. x1=600: Se asignan 600 acciones a la primera oferta (<500,100,600>).
2. x2=400: Se asignan 400 acciones a la segunda oferta (<450,400,800>).

Costo total:

$$vr(X)=(600 \cdot 500)+(400 \cdot 450)=300{,}000+180{,}000=480{,}000$$

Justificación: La fuerza bruta evalúa todas las combinaciones posibles y selecciona la que maximiza el valor total. En este caso, asignar el máximo posible a la primera oferta y el restante a la segunda es óptimo.

Solución 2: Estrategia por Programación Dinámica

Descripción: La estrategia de programación dinámica divide el problema en subproblemas más pequeños y calcula el valor máximo acumulado para transformar las primeras iii ofertas con un total de aaa acciones asignadas.

Asignación:

Pasos del algoritmo:

- Inicializa una matriz *dp* con ceros donde *dp[i][a]* representa el valor máximo para las primeras *i* ofertas y *a* acciones asignadas.

- Itera sobre cada oferta y cada posible cantidad de acciones asignadas:
  - Si se selecciona una oferta, se evalúa asignar entre *mi* y *min(Mi,a)* acciones.
  - Para cada asignación:
    - Se calcula el valor acumulado sumando $x \cdot pi$ y actualizando *dp[i][a]*.
- Al final, *dp[n][A]* contiene el valor máximo para transformar todas las ofertas.

Asignación óptima obtenida:

- x1=600: Se asignan 600 acciones a la primera oferta (<500,100,600>).
- x2=400: Se asignan 400 acciones a la segunda oferta (<450,400,800>).

Costo total:

$$vr(X)=(600 \cdot 500)+(400 \cdot 450)=300{,}000+180{,}000=480{,}000$$

Justificación: La programación dinámica utiliza una matriz para calcular el costo mínimo acumulado en cada posición, considerando todas las combinaciones de asignaciones posibles. En este caso, los valores de *dp* permiten identificar la combinación óptima directamente sin necesidad de explorar todas las combinaciones exhaustivamente, como en la fuerza bruta.

```
Fuerza Bruta
Asignación: [600, 400, 0]
Valor: 480000

Programación Dinámica:
Asignación: [600, 400, 0]
Valor: 480000
```

2.2 Una primera aproximación

Descripción del Enfoque

El algoritmo que lista todas las posibles asignaciones para el problema es el de **fuerza bruta**, que explora exhaustivamente todas las combinaciones de asignaciones posibles para encontrar la que maximice vr(X).

Algoritmo: Fuerza Bruta

1. **Entrada:**
    - A=1000: Total de acciones.
    - B=100: Precio mínimo aceptado por acción.
    - n=4: Número de ofertas.
    - Ofertas:
        - <500,400,600>
        - <450,100,400>
        - <400,100,400>
        - <200,50,200>
        - <100,0,1000> (oferta del gobierno).
2. **Algoritmo:**
    - Enumerar todas las combinaciones posibles de asignaciones X=<y1,y2,...,y4>, donde:
        - $yi \in [mi,\ Mi]$
        - $\sum_{i=1}^{n} yi \ = \ A$
    - Calcular vr(X) para cada asignación.
    - Seleccionar la asignación con el mayor vr(X).

Evaluación de Ejemplo

Se listan las combinaciones posibles cumpliendo las restricciones. Aquí mostramos algunas asignaciones relevantes:

1. **Asignación 1:** X=[600,400,0,0]

    vr(X)=(600 · 500)+(400 · 450) = 300,000+180,000 = 480,000.

2. **Asignación 2:** X=[500,400,100,0]

    vr(X)=(500 · 500)+(400 · 450)+(100 · 400) = 250,000+180,000+40,000 = 470,000.

3. **Asignación 3:** X=[400,400,200,0]

    vr(X)=(400 · 500)+(400 · 450)+(200 · 400) = 200,000+180,000+80,000 = 460,000.

4. **Asignación 4:** X=[400,400,100,100]

$$vr(X)=(400 \cdot 500)+(400 \cdot 450)+(100 \cdot 400)+(100 \cdot 200)$$
$$=200,000+180,000+40,000+20,000 = 440,000.$$

Mejor Asignación: La asignación óptima encontrada por el algoritmo es X=[600,400,0,0] con un costo total de vr(X)=480,000.

¿El Algoritmo Encuentra la Solución Óptima?

Sí, el algoritmo de fuerza bruta encuentra siempre la solución óptima, ya que evalúa todas las combinaciones posibles. Sin embargo, su principal limitación es el tiempo de ejecución.

Justificación del Rendimiento

1. **Complejidad Computacional:**
   - Dado que el algoritmo evalúa todas las combinaciones posibles de asignaciones dentro de los límites permitidos, su complejidad es exponencial:

     $$O(M1 \cdot M2 \cdot M3 \cdot M4)$$

     Donde $Mi=(Mi-mi+1)$ es el rango de valores para la i-ésima oferta.

   - En este caso:
     - $M1=600-400+1=201$
     - $M2=400-100+1=301$
     - $M3=400-100+1=301$
     - $M4=200-50+1=151$
   - Número de combinaciones aproximadas: $201 \cdot 301 \cdot 301 \cdot 151 = 2,751,662,401$ combinaciones.
2. **Limitaciones:**
   - Este enfoque se vuelve **poco práctico para grandes valores de A y n**, ya que el tiempo de ejecución aumenta rápidamente.
3. **Ventajas:**
   - Garantiza encontrar la solución óptima porque evalúa todas las combinaciones posibles sin ninguna heurística.

Conclusión

Aunque el algoritmo de fuerza bruta encuentra la solución óptima, su rendimiento es muy lento debido a la naturaleza exponencial de la exploración. Por esta razón, es preferible utilizar enfoques más eficientes, como **programación dinámica** o **algoritmos voraces**, especialmente cuando el número de ofertas o el rango de asignaciones es grande.

2.3 Caracterizar la estructura de una solución óptima.

Descripción

La solución óptima para el problema de la subasta pública se puede construir utilizando programación dinámica. Esta técnica nos permite dividir el problema en subproblemas más pequeños, resolver cada uno de estos subproblemas óptimamente y luego utilizar estas soluciones para construir la solución global.

Descomposición Recursiva del Problema

1. **Cada oferta se considera como un subproblema**:
    - **Subproblema i**: Determina la mejor asignación de xi acciones de la oferta i para maximizar el valor vr(X).
    - Se busca asignar el mayor número de acciones posibles dentro de los límites de mi ≤ xi ≤ Mi.
    - La elección de xi se hace optimizando el valor acumulado en vr(X) considerando las asignaciones previas.
2. **Construcción de la matriz dp**:
    - Utilizamos una matriz *dp* donde *dp[i][a]* representa el valor máximo para asignar *a* acciones considerando las primeras iii ofertas.
    - **Caso base**: *dp[0][a]=0* para todo *a*, ya que no hay ofertas para asignar.
    - **Caso recursivo**: Se determina *dp[i][a]* como el máximo entre no seleccionando la oferta *i* o seleccionándola y sumando el valor $dp[i-1][a-xi]+xi \cdot pi$, donde *xi* es el número de acciones asignadas de la oferta *i*.
3. **Solución óptima**:
    - La solución óptima se obtiene cuando *dp[n][A]* proporciona el valor máximo acumulado para A acciones distribuidas entre las n ofertas.
    - Esta matriz se construye iterativamente:
        - Por cada oferta *i*, y para cada *a* acciones asignadas posibles:
            - Se evalúa si asignar *xi* acciones a la oferta *i* incrementa el valor acumulado vr(X) de manera óptima.
            - La selección de *xi* se realiza optimizando el valor de vr(X).

Ejemplo

Considerando las ofertas:

- <500,400,600>
- <450,100,400>
- <400,100,400>
- <200,50,200>
- <100,0,1000> (oferta del gobierno)

La matriz *dp* se llena iterativamente:

- **Inicialización**: *dp[0][a]=0* para todo *a* ya que no hay ofertas.
- **Iteración**:
  - Se evalúa cada oferta *i*:
    - Se consideran las asignaciones posibles para *xi* y se actualiza *dp[i][a]* con el valor máximo encontrado.
    - Se compara:
      - No asignar *xi (dp[i−1][a])*.
      - Asignar *xi* y sumarlo a *dp[i−1][a−xi]+xi · pi*.
- **Resultado**:
  - *dp[n][A]* proporciona la solución óptima al problema.

2.4 Definir recursivamente el valor de una solución óptima

Recurrencia para *dp[i][a]*:

La recurrencia para el valor acumulado *dp[i][a]* se define de la siguiente manera:

$$dp[i][a] = \begin{cases} dp[i-1][a] & \text{si no se asigna ninguna acción de la oferta } i \\ dp[i-1][a-x_i] + x_i \cdot p_i & \text{si } m_i \leq x_i \leq \min(M_i, a) \end{cases}$$

- **Explicación**:
  - **Caso 1**: *dp[i−1][a]* es el valor acumulado sin tomar ninguna acción de la oferta *i*. Esto nos da el valor si ignoramos la oferta *i*.

- **Caso 2**: $dp[i-1][a-xi]+xi \cdot pi$ toma en cuenta la oferta i asignando xi acciones, sumando su costo y el valor acumulado de las ofertas anteriores con $a-xi$ acciones restantes.
- Esta fórmula permite acumular los valores óptimos paso a paso, generando una solución global a partir de decisiones locales.

2.5 Calcular el valor de una solución.

Descripción del Algoritmo

El objetivo del algoritmo es calcular el costo de la solución óptima para la asignación de acciones en la subasta pública utilizando programación dinámica. Este algoritmo explora las combinaciones posibles de asignaciones a través de subproblemas y acumula los valores para determinar la solución óptima.

Paso a Paso para Calcular el Valor de la Solución

1. **Construcción de la matriz dp**:
   - La matriz dp tiene dimensiones (n+1)×(A+1):
     - $dp[i][a]$ representa el valor acumulado máximo para las primeras i ofertas y a acciones asignadas.
   - **Inicialización**:
     - $dp[0][a]=0$ para todo a: No hay ofertas para asignar.
     - $dp[i][0]=0$ para todo i: No se han asignado acciones.
2. **Iteración sobre las ofertas**:
   - Por cada oferta i:
     - Se evalúa cada posible a acciones asignadas:

       Inicialmente, copia el valor sin considerar la oferta i: $dp[i][a]=dp[i-1][a]$

       Si el precio de la oferta iii es mayor o igual a B:
       Se evalúan todas las cantidades x de acciones asignables dentro del rango $mi \leq xi \leq min(Mi,a)$, $dp[i][a]$ se actualiza como:

       $dp[i][a] = max(dp[i][a],dp[i-1][a-x]+x \cdot pi)$

       Esta fórmula asegura que estamos eligiendo la mejor asignación para maximizar el valor acumulado.

3. **Resultado final**:
    - La solución óptima se encuentra en *dp[n][A]*, que proporciona el valor máximo acumulado para nnn ofertas y A acciones.
    - Se puede reconstruir la solución óptima mediante backtracking utilizando las asignaciones de las matrices *dp* y '`asignaciones`'.

Complejidad del Algoritmo

- **Tiempo**: O(n · A)
    - El algoritmo itera sobre todas las ofertas y todas las posibles asignaciones de acciones.
    - Para cada oferta *i*, se evalúan *min(Mi,A) − mi+1* posibles asignaciones de acciones.
    - La complejidad total se da por la iteración sobre las ofertas y las acciones asignadas.
- **Espacio**: O(n · A)
    - Se necesita espacio para almacenar la matriz *dp* que guarda los valores acumulados de las ofertas y acciones asignadas.

2.6 Construir una solución óptima

Descripción del Problema

El objetivo es construir una asignación óptima de acciones X=[x1,x2,...,xn,xn+1] que maximice vr(X) para las ofertas dadas en la subasta pública, utilizando un enfoque voraz.

Construcción de la Solución Óptima con el Algoritmo Voraz

1. **Ordenar las ofertas**:
    - Comenzamos por ordenar las ofertas en orden descendente según el precio *pi*. Este orden permite seleccionar primero las ofertas con mayor valor por acción, las cuales tienen mayor impacto en el valor acumulado de vr(X).
2. **Asignación de acciones**:
    - **Inicialización**:
        - `acciones_restantes` es igual a A (total de acciones disponibles).
        - `mejor_asignacion` se inicializa como un arreglo de ceros de longitud n + 1 para almacenar las cantidades de acciones asignadas a cada oferta.
        - `valor_total` comienza en 0, ya que no se ha asignado ninguna acción.

- Iterar sobre las ofertas:
  - Para cada oferta *i* en la lista ordenada:
    - **Condición de selección**:

      Si el precio *pi* es mayor o igual a B (el precio mínimo aceptado por acción), consideramos la oferta.

    - **Asignación de acciones**:

      `max_asignacion` se define como *min(Mi,acciones_restantes)*, donde *Mi* es el máximo número de acciones permitidas para la oferta *i*.

      Si `max_asignacion` es mayor o igual a *mi* (el mínimo número de acciones requerido):

      - Asigna `max_asignacion` acciones de la oferta *i* a `mejor_asignacion[i]`.
      - Actualiza `acciones_restantes` restando `max_asignacion` de las acciones restantes.
      - Incrementa `valor_total` sumando *maxasignacion × pi* al valor total acumulado.
  - **Asignación de las acciones restantes**:
    - Una vez que se han considerado todas las ofertas disponibles, las acciones restantes se asignan al gobierno.
    - `mejor_asignacion[n]` es la cantidad de acciones restantes asignadas al gobierno.
    - `valor_total` se incrementa con *acciones_restantes×pn*, donde pn es el precio de la última oferta.

3. **Resultado final**:
   - `mejor_asignacion` contiene las asignaciones óptimas de acciones para cada oferta.
   - `valor_total` es el valor acumulado máximo de vr(X).