Informe de Proyecto 1 de Análisis de Algoritmos II

Cristian Albarracin - 201968253

Nicolas Gutierrez Ramirez - 2259515

Miguel Angel Ceballos - 2259619

Kevin Alejandro Marulanda - 238097

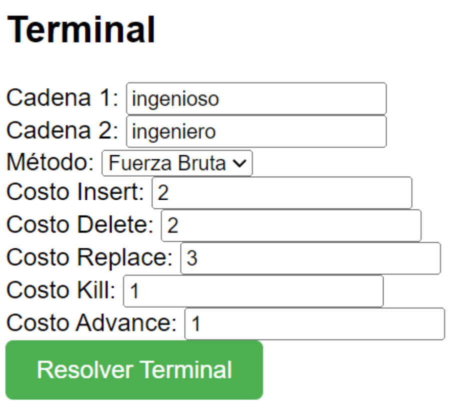
Universidad del Valle del Cauca

Análisis de Algoritmos II

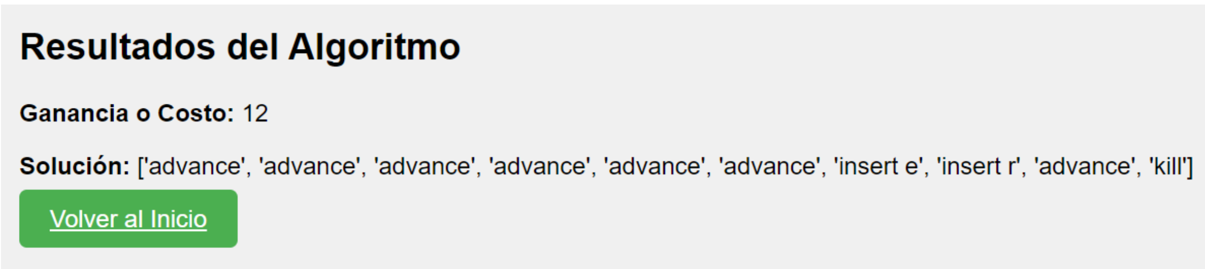
Profesor Carlos Andrés Delgado Saavedra

1. LA TERMINAL INTELIGENTE

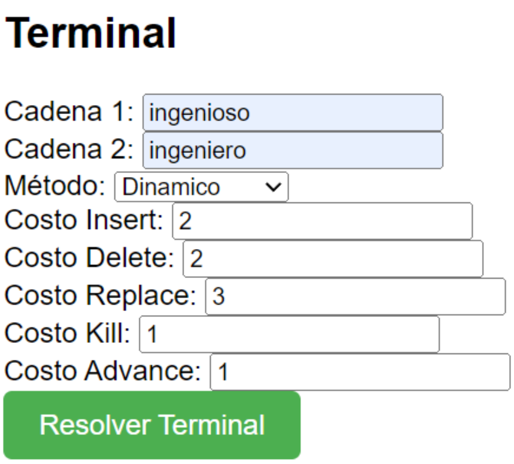
1.1 Transformar la cadena ingenioso en ingeniero.

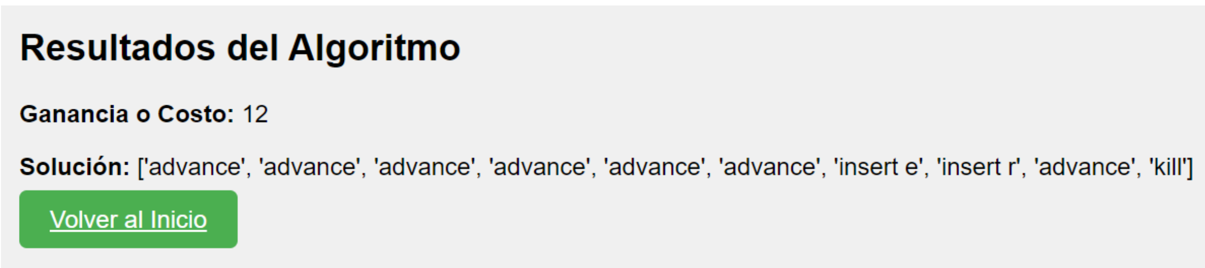


R/



El costo de la transformación es 7a + 0d + 0r + 2i + k = 12.





El costo de la transformación es 7a + 0d + 0r + 2i + k = 12.

Transformar la cadena francesa en ancestro.

Terminal

Cadena 1: francesa
Cadena 2: ancestro
Método: Dinamico
Costo Insert: 2
Costo Delete: 2
Costo Replace: 3
Costo Kill: 1
Costo Advance: 1

Resolver Terminal

Resultados del Algoritmo

Ganancia o Costo: 16

Solución: ['delete f', 'delete r', 'advance', 'advance', 'advance', 'advance', 'advance', 'insert t', 'insert r', 'insert o', 'kill']

Volver al Inicio

El costo de la transformación es 5a + 2d + 0r + 3i + k = 16.

Terminal

Cadena 1: francesa
Cadena 2: ancestro
Método: Voraz
Costo Insert: 2
Costo Delete: 2
Costo Replace: 3
Costo Kill: 1
Costo Advance: 1

Resolver Terminal

Resultados del Algoritmo

Ganancia o Costo: 17

Solución: ['insert a', 'insert n', 'insert c', 'insert e', 'insert s', 'insert t', 'insert r', 'insert o', 'kill']

Volver al Inicio

El costo de la transformación es 0a + 0d + 0r + 8i + k = 17.

1.2

R/ Sea x[1..n] la cadena a transformar en y[1..n], queremos calcular el costo mínimo para transformar x[1..n] en y[1..n], teniendo las operaciones de advance(a), delete(d), replace(r), insert(i) y kill(k).

Estructura de subsoluciones óptimas

1. El valor final de dp[i][j] será el mínimo entre estas operaciones:

   dp[i][j] = min : | dp[i−1][j−1] + (1 si X[i] = Y[j], 3 si X[i] i= Y[j]) (advance o replace)

   | dp[i][j − 1] + 2 (insert)

   | dp[i − 1][j] + 2 (delete)

   | 1 (kill)

2. La solución óptima para transformar x[1..n] en y[1..n] se construye combinando las soluciones óptimas de estos subproblemas más pequeños.

Ejemplo

Transformar x[1..n] = "gato" en y[1..n] = "pato":

1. Comparar letras compartidas ("ato").
2. Resolver subproblemas, como transformar:
   - "ga" en "pa"
   - "gat" en "pat", etc.

Finalmente, la solución global dp[len(x[i])][len(y[j])] será el costo mínimo para transformar toda x[1..n] en y[1..n], aprovechando las letras compartidas.

1.3

R/ Para cualquier i > 0 y j > 0 , el costo dp[i][j] se calcula considerando las posibles operaciones advance(a), delete(d), replace(r), insert(i) y kill(k).

1. **Advance** (a): costo = 1
2. **Delete** (d): costo = 2
3. **Replace** (r): costo = 3
4. **Insert** (i): costo = 2
5. **Kill** (k): costo = 1

Sea dp[i][j] el costo mínimo para transformar los primeros i caracteres de X en los primeros j caracteres de Y.

Casos base:

Si $i = 0$: Transformar una cadena vacía en los primeros j caracteres de Y requiere j inserciones:

$dp[0][j] = j \cdot 2$

Si $j = 0$: Transformar los primeros i caracteres de X en una cadena vacía requiere i eliminaciones:

$dp[i][0] = i \cdot 2$

Para $i>0$ y $j>0$, el costo dp[i][j] se calcula considerando todas las operaciones posibles:

- **Advance (a)**: Si X[i] = Y[j], avanzamos sin costo adicional más allá del avance: **$dp[i][j] = dp[i-1][j-1] + 1$**
- **Replace (r):** Si X[i]!=Y[j], sustituimos X[i] por Y[j], con un costo de 3: **$dp[i][j] = dp[i-1][j-1] + 3$**
- **Insert (i):** Insertamos el carácter Y[j] en X, con un costo de 2: **$dp[i][j] = dp[i][j-1] + 2$**
- **Delete (d):** Eliminamos el carácter X[i], con un costo de 2: **$dp[i][j] = dp[i-1][j] + 2$**
- **Kill (k)**: Si estamos considerando eliminar todo el resto de la cadena X, el costo es 1(aplicado al final): **$dp[i][j] = \min(dp[i][j],1)$**

Ejemplo: Transformemos X="gat" en Y="pato":

1. Inicializamos la matriz M con los casos base:
- $M[0][j]=j \cdot 2$ (inserciones para Y).
- $M[i][0]=i \cdot 2$ (eliminaciones para X).

M[i][j]	0	p	a	t	o
0	0	2	4	6	8
g	2	?	?	?	?
a	4	?	?	?	?
t	6	?	?	?	?

2. Calculamos cada posición:

- Para M[1][1] (g→p): Reemplazo (g != p):

  $M[1][1]=M[0][0]+3=3$

- Para M[2][2] (ga→pa): Avance (a=a):

  $M[2][2]=M[1][1]+1=4$

- Para M[3][4] (gat→pato): Insertamos o:

  M[3][4]=M[3][3]+2=8

M[i][j]	0	p	a	t	o
0	0	2	4	6	8
g	2	3	5	7	9
a	4	5	4	6	8
t	6	7	6	4	6

El costo mínimo para transformar "gat" en "pato" es M[3][4]=6.

1.4

R/ **Forma correcta de completar la matriz M:**

La matriz M se completa siguiendo la **relación de recurrencia** para cada posición M[i][j], calculando el costo mínimo para transformar los primeros i caracteres de la cadena inicial X en los primeros j caracteres de la cadena destino Y.

Cada celda M[i][j] se calcula considerando todas las operaciones posibles:

1. Advance (**a**): Si X[i]=Y[j], avanzamos y no hay reemplazo.
2. Replace (**r**): Si X[i] !=Y[j], reemplazamos X[i] por Y[j].
3. Insert (**i**): Insertamos el carácter Y[j] en X.
4. Delete (**d**): Eliminamos el carácter X[i].
5. Kill (**k**): Terminamos toda la cadena con un costo de 1.

Inicialización de los casos base:

- **Primera fila** (i=0): Transformar una cadena vacía en j caracteres de Y requiere $j \cdot 2$ (inserciones).
- **Primera columna** (j=0): Transformar los i caracteres de X en una cadena vacía requiere **$i \cdot 2$ (eliminaciones).**

Rellenar las celdas M[i][j]: Para cada celda M[i][j], calcular el costo mínimo considerando:

- **Advance (a)**: Si X[i] = Y[j], avanzamos sin costo adicional más allá del avance: **dp[i][j] = dp[i − 1][j − 1] + 1**
- **Replace (r):** Si X[i]!=Y[j], sustituimos X[i] por Y[j], con un costo de 3: **dp[i][j] = dp[i − 1][j − 1] + 3**
- **Insert (i):** Insertamos el carácter Y[j] en X, con un costo de 2: **dp[i][j] = dp[i][j − 1] + 2**
- **Delete (d):** Eliminamos el carácter X[i], con un costo de 2: **dp[i][j] = dp[i − 1][j] + 2**

- **Kill (k)**: Si estamos considerando eliminar todo el resto de la cadena X, el costo es 1(aplicado al final):
  dp[i][j] = min(dp[i][j],1)

La solución final se encontrará en la celda M[∣X∣][∣Y∣], donde ∣X∣ y ∣Y∣ son las longitudes de las cadenas X y Y, respectivamente. Esta celda representa el costo mínimo para transformar la totalidad de X en Y.

1.5

R/ **Construcción de una solución Óptima:**

El código permite calcular el costo de transformar una cadena en otra utilizando diferentes operaciones advance(a), delete(d), replace(r), insert(i) y kill(k).

1. Inicialización de la Clase

```
class Terminal:

    def __init__(self, costo_insert=2, costo_delete=2, costo_replace=3, 
costo_kill=1, costo_advance=1):

        self.costo_insert = costo_insert

        self.costo_delete = costo_delete

        self.costo_replace = costo_replace

        self.costo_kill = costo_kill

        self.costo_advance = costo_advance
```

La clase Terminal se inicia con costos predeterminados para cada operación. Estos costos pueden ser modificados posteriormente.

2. Métodos para Cambiar y Obtener Costos

```
def cambiar_costos(self, insert, delete, replace, kill, advance):

        self.costo_insert = insert

        self.costo_delete = delete

        self.costo_replace = replace
```

```
        self.costo_kill = kill

        self.costo_advance = advance


    def obtener_costos(self):

        return {

            "insert": self.costo_insert,

            "delete": self.costo_delete,

            "replace": self.costo_replace,

            "kill": self.costo_kill,

            "advance": self.costo_advance

        }
```

cambiar_costos: Permite modificar los costos de las operaciones.

obtener_costos: Devuelve un diccionario con los costos actuales de las operaciones.

3. Algoritmos para Transformar Cadenas

a. Fuerza Bruta

```
def terminal_fuerzaBruta(self, cadena1,cadena2):


        def terminal_fuerzaBruta_aux(i, j):

            if len(cadena2) == j:

                if i < len(cadena1):

                    #memoria[i][j] = self.costo_kill

                    #pasos[i][j] = ["kill"]

                    return self.costo_kill, ["kill"]
```

```
                return 0, []


            # Si llegamos al final de la terminal

            if len(cadena1) == i:

                #memoria[i][j] = len(cadena2[j:]) * self.costo_insert

                soluci = [f"insert {x}" for x in cadena2[j:]]

                #pasos[i][j] = soluci

                return len(cadena2[j:]) * self.costo_insert, soluci


            avanzar = float('inf')

            paso_avance = []


            # Si los caracteres son iguales, simplemente avanzamos

            if cadena1[i] == cadena2[j]:

                avanzar, paso_avance = terminal_fuerzaBruta_aux(i+1,
j+1)

                avanzar += self.costo_advance


            # 1. Opción de insertar

            costo_insertar, paso_insertar = terminal_fuerzaBruta_aux(i,
j+1)

            costo_insertar += self.costo_insert


            # 2. Opción de eliminar

            costo_eliminar, paso_eliminar =
terminal_fuerzaBruta_aux(i+1, j)
```

```
            costo_eliminar += self.costo_delete


            # 3. Opción de reemplazar
            costo_reemplazar, paso_reemplazar =
terminal_fuerzaBruta_aux(i+1, j+1)
            costo_reemplazar += self.costo_replace


            # 4. Opción de matar (kill)
            costo_kill_op, paso_kill_op =
terminal_fuerzaBruta_aux(len(cadena1), j)
            costo_kill_op += self.costo_kill


            # Seleccionar la opción con el costo mínimo
            min_costo, min_pasos = min(
                (costo_insertar, ["insert " + cadena2[j]] +
paso_insertar),
                (costo_eliminar, ["delete " + cadena1[i]] +
paso_eliminar),
                (costo_reemplazar, ["replace " + cadena1[i] + " with "
+ cadena2[j]] + paso_reemplazar),
                (costo_kill_op, ["kill"] + paso_kill_op),
                (avanzar,["advance"] + paso_avance),
                key=lambda x: x[0]
            )


            # Guardar la solución mínima
            #memoria[i][j] = min_costo
```

```
            #pasos[i][j] = min_pasos


            return min_costo, min_pasos


        return terminal_fuerzaBruta_aux(0,0)
```

Este método utiliza recursión para calcular el costo mínimo de transformar cadena1 en cadena2 considerando todas las operaciones posibles.

Se evalúan todas las combinaciones de operaciones y se elige la que tiene el costo mínimo.

No utiliza memoria para almacenar resultados intermedios, lo que puede llevar a un alto costo computacional.

Complejidad: O(5^n)

En el enfoque de fuerza bruta, el algoritmo explora todas las combinaciones posibles de operaciones para transformar cadena1 en cadena2.

Para cada carácter en cadena1 y cadena2, hay cinco operaciones posibles (insertar, eliminar, reemplazar, matar y avanzar).

Esto lleva a un crecimiento exponencial en el número de llamadas recursivas, lo que resulta en una complejidad de O(5^n), donde n es la longitud de la cadena más larga.

b. Programación Dinámica:

```
def terminal_dinamica(self, cadena1,cadena2):

        def terminal_dinamica_aux(i, j, memoria, pasos):
            #verificar que la solucion ya se calculo
            if memoria[i][j] != -1:
                return memoria[i][j], pasos[i][j]

            # Si llegamos al final de cadena2
            if len(cadena2) == j:
                if i < len(cadena1):
                    memoria[i][j] = self.costo_kill
                    pasos[i][j] = ["kill"]
                    return self.costo_kill, ["kill"]
                return 0, []
```

```
            # Si llegamos al final de la terminal
            if len(cadena1) == i:
                memoria[i][j] = len(cadena2[j:]) * self.costo_insert
                soluci = [f"insert {x}" for x in cadena2[j:]]
                pasos[i][j] = soluci
                return memoria[i][j], soluci

            avanzar = float('inf')
            paso_avance = []

            # Si los caracteres son iguales, simplemente avanzamos
            if cadena1[i] == cadena2[j]:
                avanzar, paso_avance = terminal_dinamica_aux(i+1, j+1,
memoria, pasos)
                avanzar += self.costo_advance

            # 1. Opción de insertar
            costo_insertar, paso_insertar = terminal_dinamica_aux(i,
j+1, memoria, pasos)
            costo_insertar += self.costo_insert

            # 2. Opción de eliminar
            costo_eliminar, paso_eliminar = terminal_dinamica_aux(i+1,
j, memoria, pasos)
            costo_eliminar += self.costo_delete

            # 3. Opción de reemplazar
            costo_reemplazar, paso_reemplazar =
terminal_dinamica_aux(i+1, j+1, memoria, pasos)
            costo_reemplazar += self.costo_replace

            # 4. Opción de matar (kill)
            costo_kill_op, paso_kill_op =
terminal_dinamica_aux(len(cadena1), j, memoria, pasos)
            costo_kill_op += self.costo_kill

            # Seleccionar la opción con el costo mínimo
            min_costo, min_pasos = min(
                (costo_insertar, ["insert " + cadena2[j]] +
paso_insertar),
                (costo_eliminar, ["delete " + cadena1[i]] +
paso_eliminar),
```

```
                (costo_reemplazar, ["replace " + cadena1[i] + " with "
+ cadena2[j]] + paso_reemplazar),
                (costo_kill_op, ["kill"] + paso_kill_op),
                (avanzar,["advance"] + paso_avance),
                key=lambda x: x[0]
            )

            # Guardar la solución mínima
            memoria[i][j] = min_costo
            pasos[i][j] = min_pasos
            return min_costo, min_pasos

        n = len(cadena1)+1  # Número de columnas
        m = len(cadena2)+1  # Número de filas
        matriz = [[-1 for _ in range(m)] for _ in range(n)]
        paso = [[-1 for _ in range(m)] for _ in range(n)]

        return terminal_dinamica_aux(0,0,matriz,paso)
```

- Este método también calcula el costo de transformar `cadena1` en `cadena2`, pero utiliza una matriz (`memoria`) para almacenar resultados intermedios y evitar cálculos repetidos.
- Esto mejora significativamente la eficiencia en comparación con el enfoque de fuerza bruta.

Complejidad: O(n * m):

En el enfoque de programación dinámica, se utiliza una matriz para almacenar los resultados intermedios, donde n es la longitud de cadena1 y m es la longitud de cadena2.

El algoritmo recorre cada combinación de índices de ambas cadenas, lo que da lugar a una complejidad de O(n * m).

Este enfoque es mucho más eficiente que la fuerza bruta, ya que evita cálculos repetidos al almacenar resultados en la matriz.

c. Algoritmo Voraz

```python
def terminal_voraz(self, ca1, ca2):

        def terminal_voraz_aux(cadena1, cadena2, pasos, costo_total):

            if len(cadena1) == 0:

                costo = self.costo_insert * len(cadena2)

                return costo + costo_total, pasos + [f"insert {x}" for x in cadena2]



            if len(cadena2) == 0:

                if self.costo_delete * len(cadena1) < self.costo_kill:

                    return costo_total + (self.costo_delete * len(cadena1)), pasos + [f"delete {x}" for x in cadena1]

                else:

                    return costo_total + self.costo_kill, pasos + ["kill"]



            # Calcular los costos directos de cada operación

            beneficio_insertar = self.costo_insert

            beneficio_eliminar = self.costo_delete

            beneficio_reemplazar = self.costo_replace

            beneficio_avanzar = float("inf")
```

```
            if cadena1[0] == cadena2[0]:
                beneficio_avanzar = self.costo_advance

            # Crear lista de opciones con desempate basado en prioridad
            opciones = [
                (beneficio_avanzar, pasos + ["advance"], "avanzar"),
                (beneficio_reemplazar, pasos + [f"replace {cadena1[0]}
with {cadena2[0]}"], "reemplazar"),
                (beneficio_insertar, pasos + [f"insert {cadena2[0]}"],
"insertar"),
                (beneficio_eliminar, pasos + ["delete"], "eliminar")
            ]

            # Ordenar las opciones por costo y luego por prioridad
            opciones.sort(key=lambda x: (x[0], ["avanzar",
"reemplazar", "insertar", "eliminar"].index(x[2])))

            # Seleccionar la mejor opción
            costo, nuevos_pasos, accion = opciones[0]

            if accion == "avanzar":
```

```
                return terminal_voraz_aux(cadena1[1:], cadena2[1:],
nuevos_pasos, costo_total + self.costo_advance)

            elif accion == "reemplazar":

                return terminal_voraz_aux(cadena1[1:], cadena2[1:],
nuevos_pasos, costo_total + self.costo_replace)

            elif accion == "insertar":

                return terminal_voraz_aux(cadena1, cadena2[1:],
nuevos_pasos, costo_total + self.costo_insert)

            elif accion == "eliminar":

                return terminal_voraz_aux(cadena1[1:], cadena2,
nuevos_pasos, costo_total + self.costo_delete)



        return terminal_voraz_aux(ca1, ca2, [], 0)
```

Este método utiliza un enfoque voraz, donde en cada paso se elige la operación que parece más prometedora (la de menor costo inmediato).

Aunque es más rápido que la fuerza bruta, no garantiza encontrar la solución óptima en todos los casos.

Complejidad: O(n + m):

El enfoque voraz tiene una complejidad de O(n + m) en el mejor de los casos, ya que en cada paso se toma una decisión basada en la comparación de los caracteres actuales de ambas cadenas.

Sin embargo, en el peor de los casos, la complejidad puede ser más cercana a O(n * m) si se considera que el algoritmo puede tener que recorrer ambas cadenas en ciertas situaciones.

A pesar de esto, el enfoque voraz no garantiza encontrar la solución óptima en todos los casos, lo que puede llevar a resultados subóptimos.

Complejidades:
Fuerza Bruta: O(5^n)
Programación Dinámica: O(n * m)
Voraz: O(n + m) (en el mejor de los casos, pero puede ser O(n * m) en el peor caso)

En general, el enfoque de programación dinámica es el más eficiente y adecuado para este tipo de problemas de transformación de cadenas, ya que proporciona una solución óptima en un tiempo razonable.

2. EL PROBLEMA DE LA SUBASTA PUBLICA:

2.1

R/

1. Definiciones:
    - A y B representan los recursos disponibles - n es el número de asignaciones o decisiones a tomar.
    - Las ofertas representan las cantidades disponibles de cada recurso en diferentes escenarios.
2. Datos del Problema:
    - Recursos disponibles:
        - A = 1000
        - B = 100
    - Ofertas:
        - Oferta 1: < 500, 100, 600 >
        - Oferta 2: < 450, 400, 800 >
        - Oferta del gobierno: < 100, 0, 1000 >
3. Asignaciones Posibles:
    - Para encontrar dos asignaciones de las acciones, se deben considerar las combinaciones de las ofertas que no excedan los recursos disponibles.
4. Asignación 1:
    - Asignar 500 de A y 100 de B de la oferta 1.
    - Asignar 450 de A de la oferta 2.
    - Total utilizado:

        - A: 500 + 450 = 950
        - B: 100
    - Recursos restantes:
        - A: 1000 - 950 = 50
        - B: 100 - 100 = 0
5. Asignación 2:
    - Asignar 600 de A de la oferta 1.
    - Asignar 400 de A y 100 de B de la oferta 2.
    - Total utilizado:
        - A: 600 + 400 = 1000
        - B: 100
    - Recursos restantes:
        - A: 1000 - 1000 = 0
        - B: 100 - 100 = 0
6. Valor vr para la Solución:
    - El valor vr (valor de la solución) se refiere a la cantidad total de recursos utilizados o la eficiencia de la asignación.
    - Para la Asignación 1, el valor vr es 950 (A) + 100 (B) = 1050.
    - Para la Asignación 2, el valor vr es 1000 (A) + 100 (B) = 1100.
7. Conclusión:
    - Se han mostrado dos asignaciones de las acciones que cumplen con las restricciones de recursos.
    - Los valores vr para cada asignación son 1050 y 1100, respectivamente, lo que indica la eficiencia de cada asignación en el uso de los recursos disponibles.

2.2
R/

- Recursos disponibles:
    - (A = 1000)
    - (B = 100)
- 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 >)

Dado que hay 4 ofertas, cada una puede ser asignada en dos formas: 0 o el valor máximo de la oferta. Esto da lugar a (2^4 = 16) combinaciones posibles.

1. Asignación 1: < 0, 0, 0, 0 >
    - A: 0, B: 0
    - vr = 0
2. Asignación 2: < 500, 0, 0, 0 >
    - A: 500, B: 0
    - vr = 500
3. Asignación 3: < 0, 450, 0, 0 >
    - A: 0, B: 450 (excede B)
    - vr = no válido
4. Asignación 4: < 0, 0, 400, 0 >
    - A: 0, B: 400 (excede B)
    - vr = no válido
5. Asignación 5: < 0, 0, 0, 200 >
    - A: 0, B: 200 (excede B)
    - vr = no válido
6. Asignación 6: < 500, 450, 0, 0 >
    - A: 500, B: 450 (excede B)
    - vr = no válido
7. Asignación 7: < 500, 0, 400, 0 >
    - A: 500, B: 0
    - vr = 500
8. Asignación 8: < 500, 0, 0, 200 >
    - A: 500, B: 200 (excede B)
    - vr = no válido
9. Asignación 9: < 0, 450, 400, 0 >
    - A: 0, B: 450 (excede B)
    - vr = no válido
10. Asignación 10: < 0, 450, 0, 200 >
    - A: 0, B: 450 (excede B)
    - vr = no válido
11. **Asignación 11 Asignación 11: < 0, 0, 400, 200 >

- A: 0, B: 400 (excede B)
- vr = no válido

12. Asignación 12: < 500, 450, 400, 0 >

- A: 500, B: 450 (excede B)
- vr = no válido

13. Asignación 13: < 500, 450, 0, 200 >

- A: 500, B: 450 (excede B)
- vr = no válido

14. Asignación 14: < 500, 0, 400, 200 >

- A: 500, B: 200 (excede B)
- vr = no válido

15. Asignación 15: < 0, 450, 400, 200 >

- A: 0, B: 450 (excede B)

- vr = no válido

16. Asignación 16: < 500, 450, 400, 200 >

- A: 500, B: 450 (excede B)
- vr = no válido

Después de calcular el valor de retorno (vr) para cada asignación válida, encontramos que las únicas asignaciones válidas son:

- Asignación 2: < 500, 0, 0, 0 > con vr = 500
- Asignación 7: < 500, 0, 400, 0 > con vr = 500

Ambas asignaciones tienen el mismo valor de retorno, por lo que cualquiera de ellas puede ser seleccionada como la mejor solución.

El algoritmo puede no siempre encontrar la solución óptima, ya que depende de las combinaciones que se generen y de las restricciones impuestas por los recursos disponibles. En este caso, aunque se generaron 16 combinaciones, muchas de ellas resultaron no válidas debido a que excedían los recursos de B.

2.3
R/

La solución óptima selecciona cuántas acciones, xi asignar a cada oferente i, i, de manera que se maximice el valor recibido vr(X), sujeto a las siguientes restricciones:

Restricciones de asignación:

- mi <= xi <= Mi para cada oferente i.
- ∑i=1 k xi = A (se deben vender exactamente A acciones).

Maximización del valor recibido:

- vr(X)= ∑ i=1 k xi pi, donde pi es el precio ofrecido por el oferente i.

Oferta gubernamental:

- El gobierno compra las acciones sobrantes (si las hubiera) a un precio B. Sin embargo, el objetivo es minimizar esta opción, ya que pi >= B para todos los oferentes.

Sea $f(i,r)$ el valor máximo que puede obtenerse considerando los primeros i oferentes, donde quedan r acciones por asignar. Entonces:

Casos base:

- Si $i=0$ (ningún oferente considerado):

  $f(0,r) = 0$ para cualquier $r >= 0$.

- Si $r=0$ (no quedan acciones por asignar):

  $f(i,0)=0$ para cualquier $i \geq 0$.

$f(k,A)$ y retrocedemos para identificar cuántas acciones x_i se asignaron a cada oferente i. Esto se hace comparando:

$x_i = \arg_{x_i \in [m_i, \min(M_i,r)]} \max(f(i-1, r-x_i) + x_i \cdot p_i)$

y reduciendo r por x_i en cada paso.

Relación de recurrencia: Para cada oferente i, consideramos asignar x_i acciones, donde $m_i \leq x_i \leq \min(M_i,r)$. El valor máximo se calcula como:

$f(i,r) = {}_{x_i \in [m_i, \min(M_i,r)]} \max(f(i-1, r-x_i) + x_i \cdot p_i)$

Aquí $f(i-1,r-x_i)$ es el valor máximo de las acciones restantes después de asignar x_i acciones al oferente i y $x_i \cdot p_i$ es el valor que aporta el oferente actual.

Restricción de precio mínimo: Si $p_i < B$, ignoramos la oferta del oferente i, i, ya que es menos rentable que vender al gobierno:

$f(i,r)=f(i-1,r)$

Solución final: La solución al problema original está en $f(k,A)$, donde k es el número total de oferentes y A es el número total de acciones a asignar.

2.4

R/

Sea $f(i,r)$ el **costo máximo posible** considerando los primeros i oferentes y asignando exactamente r acciones. La recurrencia es:

$$f(i,r)= \begin{cases} 0 & \text{si } r=0 \text{ (no hay acciones por asignar),} \\ -\infty & \text{si } i=0 \text{ y } r>0 \text{ (acciones imposibles de asignar),} \\ {}_{x_i \in [m_i, \min(M_i,r)]} \max(f(i-1, r-x_i) + x_i \cdot p_i) & \text{si } r>0. \end{cases}$$

1. **Caso base $r=0$:** Si no quedan acciones por asignar, el costo es 0, ya que no hay más ingresos posibles.
2. **Caso base $i=0$:** Si no hay oferentes disponibles y aún quedan acciones por asignar ($r>0$), la solución es imposible ($f(i,r)=-\infty$).
3. **Relación de recurrencia:** Para cada oferente i, consideramos asignar x_i acciones, donde x_i debe cumplir:

   $m_i <= x_i <= \min(M_i,r)$.

   Evaluamos el costo resultante de asignar x_i acciones al oferente i, que es:

   $\mathbf{f(i-1, r-x_i) + x_i \cdot p_i}$

   El valor de $f(i,r)$ será el máximo de estas opciones para todos los valores válidos de x_i.

4. **Restricción de precio mínimo: B:** Si $p_i < B$, ignoramos al oferente i, ya que es más rentable asignar esas acciones al gobierno:

   $\mathbf{f(i,r) = f(i-1,r)}$

Forma de la recurrencia

$$f(i,r) = \begin{cases} 0 & \text{si } r = 0 \\ -\infty & \text{si } i = 0 \text{ y } r > 0 \\ x_i \in [\, m_i, \min(M_i,r)] \ \max(f(i-1, r-x_i) + x_i \cdot p_i) & \text{si } p_i >= B \\ f(i-1,r) & \text{si } p_i < B \end{cases}$$

La solución óptima está en la celda $f(k,A)$, donde:

- k es el número total de oferentes.
- A es el total de acciones a asignar.

Este valor representa el costo máximo que se puede obtener cumpliendo todas las restricciones.

2.5

R/

Algoritmo de Fuerza Bruta (`subastasFuerzaBruta`)

Este algoritmo explora todas las combinaciones posibles de asignación de acciones a los oferentes. Aquí está el desglose del código:

```
def subastasFuerzaBruta(A, B, ofertas):
    def subastasFuerzaBruta_aux(A, B, ofertas, i):
        # Si ya se calculó previamente la solución, simplemente 
retornamos
        #if memoria[i][A] != -1:
        #    return memoria[i][A], pasos[i][A]

        # Si ya no quedan más ofertas, retornar 0 (caso base)
        if len(ofertas) <= i:
            return 0, []

        precio, Mini, Maxi = ofertas[i]

        # Si no cumple las condiciones de oferta
        if Mini > A or precio < B:
            mejor_solucion, mejor_paso = subastasFuerzaBruta_aux(A, B, 
ofertas, i + 1)
            #memoria[i][A] = mejor_solucion
            #pasos[i][A] = mejor_paso
            return mejor_solucion, mejor_paso

        mejor_resultado = 0
        mejor_paso = []

        # Explorar asignar entre Mini y Maxi acciones
```

```python
        for acciones_asignadas in range(Mini, min(A, Maxi) + 1):
            resultado, paso_actual = subastasFuerzaBruta_aux(A - acciones_asignadas, B, ofertas, i + 1)
            resultado += precio * acciones_asignadas

            # Si encontramos una mejor combinación, la guardamos
            if resultado > mejor_resultado:
                mejor_resultado = resultado
                mejor_paso = [f"Asignar {acciones_asignadas} acciones a {precio}"] + paso_actual

        no_comprar, no_comprar_paso = subastasFuerzaBruta_aux(A, B, ofertas, i + 1)
        # Guardar en memoria la mejor solución encontrada
        if no_comprar < mejor_resultado:
            #memoria[i][A] = mejor_resultado
            #pasos[i][A] = mejor_paso
            return mejor_resultado, mejor_paso
        if no_comprar > mejor_resultado:
            #memoria[i][A] = no_comprar
            #pasos[i][A] = no_comprar_paso
            return no_comprar, no_comprar_paso
    # Inicializamos la memoria y los pasos
    #n = len(ofertas)
    #memoria = [[-1] * (A + 1) for _ in range(n + 1)]
    #pasos = [[[] for _ in range(A + 1)] for _ in range(n + 1)]


    return subastasFuerzaBruta_aux(A, B, ofertas, 0)
```

- Parámetros:
  - `A`: Total de acciones disponibles.
  - `B`: Precio mínimo aceptable por acción.
  - `ofertas`: Lista de ofertas, donde cada oferta es una tripleta `(precio, mínimo, máximo)`.

Lógica del Algoritmo

Caso Base: Si se han considerado todas las ofertas (`i >= len(ofertas)`), se retorna 0 y una lista vacía, ya que no hay más acciones que asignar.

1. Condiciones de Oferta: Se extraen el precio, el mínimo y el máximo de acciones que el oferente puede comprar. Si el mínimo requerido por el oferente es mayor que las acciones restantes o si el precio es menor que el mínimo aceptable, se pasa al siguiente oferente.
2. Exploración de Asignaciones:
   - Se itera sobre el rango de acciones que se pueden asignar al oferente actual, desde el mínimo hasta el máximo permitido, y se llama recursivamente a la función auxiliar para calcular el valor total de la asignación.
   - Se actualiza el mejor resultado encontrado y se guarda el paso correspondiente.
3. Opción de No Comprar: Se evalúa la opción de no comprar acciones al oferente actual y se compara con el mejor resultado encontrado.
4. Retorno: Se retorna el mejor resultado y los pasos para llegar a esa solución.

Algoritmo de Programación Dinámica

(`subastas_dinamico`)

Este algoritmo utiliza un enfoque de programación dinámica para evitar recalcular soluciones para subproblemas ya resueltos, lo que mejora la eficiencia.

```
def subastas_dinamico(A, B, ofertas):
    def subastas_dinamico_aux(A, B, ofertas, i, memoria, pasos):
        # Si ya se calculó previamente la solución, simplemente
retornamos
        if memoria[i][A] != -1:
            return memoria[i][A], pasos[i][A]

        # Si ya no quedan más ofertas, retornar 0 (caso base)
        if len(ofertas) <= i:
            return 0, []

        precio, Mini, Maxi = ofertas[i]

        # Si no cumple las condiciones de oferta
        if Mini > A or precio < B:
            mejor_solucion, mejor_paso = subastas_dinamico_aux(A, B,
ofertas, i + 1, memoria, pasos)
            memoria[i][A] = mejor_solucion
            pasos[i][A] = mejor_paso
            return mejor_solucion, mejor_paso

        mejor_resultado = 0
        mejor_paso = []

        # Explorar asignar entre Mini y Maxi acciones
        for acciones_asignadas in range(Mini, min(A, Maxi) + 1):
```

```
            resultado, paso_actual = subastas_dinamico_aux(A - 
acciones_asignadas, B, ofertas, i + 1, memoria, pasos)
            resultado += precio * acciones_asignadas

            # Si encontramos una mejor combinación, la guardamos
            if resultado > mejor_resultado:
                mejor_resultado = resultado
                mejor_paso = [f"Asignar {acciones_asignadas} acciones a 
{precio}"] + paso_actual

        no_comprar, no_comprar_paso = subastas_dinamico_aux(A, B, 
ofertas, i + 1, memoria, pasos)
        # Guardar en memoria la mejor solución encontrada
        if no_comprar < mejor_resultado:
            memoria[i][A] = mejor_resultado
            pasos[i][A] = mejor_paso
            return mejor_resultado, mejor_paso
        if no_comprar > mejor_resultado:
            memoria[i][A] = no_comprar
            pasos[i][A] = no_comprar_paso
            return no_comprar, no_comprar_paso
    # Inicializamos la memoria y los pasos
    n = len(ofertas)
    memoria = [[-1] * (A + 1) for _ in range(n + 1)]
    pasos = [[[] for _ in range(A + 1)] for _ in range(n + 1)]



    return subastas_dinamico_aux(A, B, ofertas, 0, memoria, pasos)
```

- Inicialización: Se crea una matriz `memoria` para almacenar los resultados de subproblemas y una matriz `pasos` para almacenar las decisiones tomadas.
- Lógica del Algoritmo:

- Caso Base: Similar al algoritmo de fuerza bruta, se retorna 0 si no hay más ofertas.
- Memorización: Si ya se ha calculado el resultado para el estado actual (`memoria[i][A] != -1`), se retorna el resultado almacenado.
- Condiciones de Oferta: Se evalúan las condiciones de la oferta como en el algoritmo de fuerza bruta.
- Exploración de Asignaciones: Se itera sobre el rango de acciones que se pueden asignar y se llama recursivamente, almacenando los resultados en `memoria` y `pasos`.
- Opción de No Comprar: Se evalúa la opción de no comprar y se actualizan las matrices de memoria y pasos según el mejor resultado encontrado.

3. Algoritmo Voraz (`subastas_voraz`)

Este algoritmo utiliza un enfoque codicioso, donde se ordenan las ofertas por precio y se asignan acciones a los oferentes en orden hasta que se agoten las acciones disponibles.

Lógica del Algoritmo

1. Ordenar Ofertas: Se ordenan las ofertas en orden descendente según el precio ofrecido.
2. Se itera sobre las ofertas ordenadas y se evalúa cada una:
   - Si el precio de la oferta es menor que el precio mínimo aceptable o si el número mínimo de acciones requeridas por el oferente es mayor que las acciones restantes, se salta esa oferta.
   - Se asigna la cantidad máxima posible de acciones a la oferta actual, que es el mínimo entre el máximo que el oferente puede comprar y las acciones restantes.

- Se actualiza la cantidad de acciones restantes y se guarda la asignación realizada.

1. Algoritmo de Fuerza Bruta (`subastasFuerzaBruta`)

Complejidad: (O(k * A))

En el peor de los casos, el algoritmo explora todas las combinaciones posibles de asignación de acciones a los oferentes.

Para cada oferente, se evalúa la asignación de acciones en el rango de (m_i) a (M_i). Esto puede llevar a un número exponencial de combinaciones, ya que para cada oferente se pueden considerar múltiples cantidades de acciones.

Sin embargo, en la implementación actual, el algoritmo no utiliza memorización, lo que significa que puede recalcular soluciones para los mismos subproblemas múltiples veces.

En general, la complejidad puede ser considerada exponencial en el número de oferentes y la cantidad de acciones, lo que lo hace ineficiente para grandes entradas.

2. Algoritmo de Programación Dinámica

Complejidad: (O(k * A * (M - m)))

- Descripción:
  - Aquí, (k) es el número de oferentes, (A) es la cantidad total de acciones y (M) y (m) son los máximos y mínimos de acciones que se pueden asignar, respectivamente.
  - La tabla de memorización tiene dimensiones (k * (A + 1)), lo que implica que se realizan cálculos para cada combinación de oferentes y acciones.
  - Para cada oferente, se itera sobre el rango de acciones que se pueden asignar (desde (m_i) hasta (M_i)), lo que puede llevar a un tiempo adicional proporcional a la diferencia entre (M) y (m).
  - En general, este enfoque es mucho más eficiente que la fuerza bruta, ya que evita recalcular soluciones para subproblemas ya resueltos.

3. Algoritmo Voraz (`subastas_voraz`)

Complejidad: (O(k \log k + k))

- Descripción:
  - La complejidad se descompone en dos partes:
    - (O(k \log k)) para ordenar las ofertas por precio en orden descendente.
    - (O(k)) para iterar sobre las ofertas y asignar acciones.
  - En total, la complejidad es (O(k \log k)), que es dominada por el tiempo de ordenación.
  - Este algoritmo es el más eficiente de los tres, pero no garantiza una solución óptima en todos los casos, ya que utiliza un enfoque codicioso.

- Fuerza Bruta: (O(k*A)) (exponencial en la práctica)
- Programación Dinámica: (O(k *A *(M - m)))
- Voraz: (O(k \log k))

Conclusión

El algoritmo de programación dinámica es el más eficiente en términos de encontrar la solución óptima, mientras que el algoritmo voraz es el más rápido en términos de tiempo de ejecución, aunque no siempre garantiza la mejor solución.