Universidad de Buenos Aires
Facultad De Ingeniería

Año 2024 - 2er Cuatrimestre

Teoría de Algoritmos

(75.29) Curso Buchwald - Genender

trabajo práctico

FECHA: 2 de diciembre de 2024

INTEGRANTES:

Miguel Ángel Fonzalida - #86125
mfonzalida@fi.uba.ar

Oliver Weber - #111138
@oweber@fi.uba.ar

Agustín Reinaldo Colman Salinas - #108804
@oweber@fi.uba.ar

1. Introducción

Este trabajo práctico tiene como objetivo evaluar el desarrollo y análisis de diferentes tipos de algoritmos a través de una narrativa de dos hermanos, Sophia y Mateo, y los juegos que comparten a lo largo de su crecimiento. A medida que avanzan en edad, los desafíos se vuelven más complejos, permitiendo explorar distintas técnicas algorítmicas como algoritmos greedy, programación dinámica y análisis de complejidad computacional.

El informe se divide en tres partes principales:

- Primera Parte: Introducción y Primeros Años En esta sección, se analiza un juego de monedas donde Sophia, utilizando un algoritmo greedy, asegura su victoria frente a su hermano menor Mateo, quien es muy pequeño para entender el juego. Se busca diseñar un algoritmo que garantice la solución óptima y demostrar su optimalidad, además de analizar su complejidad y realizar pruebas empíricas.
- Segunda Parte: Mateo Empieza a Jugar, Mateo ha crecido y comienza a jugar de manera competitiva, aplicando también estrategias greedy. Sophia, para mantener su ventaja, recurre a la programación dinámica. Se plantea y resuelve el problema mediante una ecuación de recurrencia, buscando maximizar su ganancia acumulada considerando las decisiones óptimas de Mateo. Se incluyen demostraciones de optimalidad, análisis de complejidad y validación con ejemplos y mediciones de tiempo.
- Tercera Parte: Cambios En su adolescencia, los hermanos exploran nuevos desafíos y se adentran en el juego de la Batalla Naval Individual. Esta sección aborda la complejidad computacional del problema, demostrando que es NP-Completo. Se desarrolla un algoritmo de backtracking para la versión de optimización del problema, buscando minimizar la demanda incumplida en un tablero con restricciones, y se analiza su desempeño.

A continuación, se detallará el desarrollo de cada parte, incluyendo el análisis teórico, la implementación de los algoritmos, los resultados obtenidos y las conclusiones relevantes.

2. Parte 1 - Introducción y Primeros Años

Este trabajo aborda el análisis y diseño de algoritmos aplicados a un juego de monedas entre dos hermanos, Sophia y Mateo. A través de este juego, se exploran conceptos fundamentales de algoritmos *greedy*, demostrando cómo pueden utilizarse para garantizar soluciones óptimas en problemas de toma de decisiones secuenciales.

3. Descripción del Problema

El juego consiste en una fila de n monedas colocadas en orden, cada una con un valor específico. En cada turno, un jugador puede tomar una moneda de uno de los extremos de la fila (la primera o la última). Los jugadores se turnan para seleccionar monedas hasta que no queda ninguna. El objetivo es maximizar la suma de los valores de las monedas recolectadas.

En esta primera parte, Sophia juega contra su hermano menor Mateo, quien es demasiado pequeño para entender el juego. Sophia debe elegir las monedas tanto para ella como para Mateo, pero siendo competitiva, busca asegurarse la victoria seleccionando las mejores monedas para sí misma y las peores para Mateo.

4. Análisis del Problema

El problema presenta características que permiten aplicar un algoritmo *greedy*:

- **Decisiones locales óptimas**: En cada turno, Sophia puede seleccionar la moneda de mayor valor disponible entre los extremos para maximizar su ganancia inmediata.
- **Control sobre las elecciones de Mateo**: Sophia elige por Mateo, pudiendo asignarle siempre la moneda de menor valor disponible.
- **Solución construida a partir de decisiones óptimas**: La solución óptima del problema se logra mediante la combinación de las mejores decisiones en cada paso. Esto significa que al tomar la mejor elección en cada turno, Sophia construye una solución óptima global.

Dado que Sophia controla las elecciones de ambos jugadores, puede aplicar una estrategia que maximice su ganancia total y minimice la de Mateo.

5. Diseño del Algoritmo *Greedy*

El algoritmo propuesto sigue estos pasos:

1. **Turno de Sophia**:
   - Sophia elige la moneda de mayor valor entre los dos extremos (inicio y fin de la fila).
   - Actualiza los índices de la fila según la elección realizada.
2. **Turno de Mateo**:
   - Sophia selecciona para Mateo la moneda de menor valor entre los dos extremos disponibles.
   - Actualiza los índices de la fila según la elección realizada.

Este proceso se repite hasta que no quedan monedas en la fila.
El código en Python es el siguiente:

```
def juego_monedas(coins):
    """
    Función que simula el juego de las monedas entre Sophia y Mateo.
    Sophia siempre elige la moneda de mayor valor para sí misma y la de menor valor
    para Mateo.

    Parámetros:
    - coins: Lista de enteros que representa los valores de las monedas en fila.

    Retorna:
    - S_Sophia: Suma total acumulada por Sophia.
```

```
    - S_Mateo: Suma total acumulada por Mateo.
    """
    # Inicialización de índices y sumas acumuladas
    i = 0                       # Índice inicial
    j = len(coins) - 1          # Índice final
    S_Sophia = 0                # Suma total de Sophia
    S_Mateo = 0                 # Suma total de Mateo

    # Mientras queden monedas por elegir
    while i <= j:
        # Turno de Sophia
        if coins[i] >= coins[j]:
            S_Sophia += coins[i]
            i += 1
        else:
            S_Sophia += coins[j]
            j -= 1

        # Verificar si ya no quedan monedas después del turno de Sophia
        if i > j:
            break

        # Turno de Mateo (Sophia elige por él)
        if coins[i] <= coins[j]:
            S_Mateo += coins[i]
            i += 1
        else:
            S_Mateo += coins[j]
            j -= 1

    return S_Sophia, S_Mateo
```

Python Code 1: Algoritmo *Greedy* para el juego de monedas

6. Optimalidad del Algoritmo

Como se mencionó anteriormente este algoritmo *greedy* garantiza la solución óptima (excepto en el caso de una cantidad par de monedas de igual valor), consideremos lo siguiente:

- **Elecciones de Sophia**: Al elegir siempre la moneda de mayor valor disponible, Sophia maximiza su ganancia inmediata en cada turno.
- **Control sobre las elecciones de Mateo**: Sophia asigna a Mateo la moneda de menor valor, minimizando su ganancia.
- **Independencia de decisiones futuras**: Las decisiones tomadas en cada turno no limitan las opciones futuras de Sophia, ya que siempre opta por la mejor opción disponible en ese momento.

Dado que Sophia controla ambas elecciones y siempre maximiza su ganancia mientras minimiza la de Mateo, la estrategia es óptima para ella. La única excepción es cuando hay una cantidad par de monedas con valores iguales, donde es inevitable un empate, independientemente de la estrategia.

7. Análisis de Complejidad

7.1. Complejidad Temporal

El algoritmo recorre la fila de monedas una sola vez, moviendo los índices hacia el centro. En cada iteración del bucle `while`, se realizan comparaciones y actualizaciones de variables que toman tiempo constante.

Por lo tanto, la complejidad temporal es:

$$\mathcal{O}(n)$$

donde n es el número de monedas.

7.2. Complejidad Espacial

El algoritmo utiliza un espacio adicional constante para las variables de índices y sumas acumuladas, independientemente del tamaño de la entrada.

Por lo tanto, la complejidad espacial es:

$$\mathcal{O}(1)$$

7.3. Impacto de la Variabilidad de los Valores de las Monedas en el Tiempo de Ejecución

La variabilidad de los valores de las monedas no afecta la complejidad temporal o espacial del algoritmo, ya que:

- El número de iteraciones del ciclo depende únicamente de la cantidad de monedas n, no de sus valores.
- Las operaciones realizadas en cada iteración son independientes de los valores específicos de las monedas (comparaciones y sumas de valores).

7.4. Impacto de la Variabilidad de los Valores de las Monedas en la Optimalidad del Algoritmo

La optimalidad del algoritmo no se ve afectada por la variabilidad en los valores de las monedas. Independientemente de si los valores son muy dispares o similares, la estrategia de Sophia garantiza que:

- **Maximiza su ganancia**: Siempre elige la moneda de mayor valor disponible.
- **Minimiza la ganancia de Mateo**: Le asigna la moneda de menor valor disponible.

Sin embargo, en el caso particular donde hay una cantidad par de monedas con valores iguales, el juego puede resultar en un empate, ya que Sophia no puede obtener una ventaja sobre Mateo siguiendo esta estrategia.

8. Ejemplos y Casos de Prueba

8.1. Ejemplo 1

Monedas: [3, 9, 2, 4, 8]

8.1.1. Ejecución Paso a Paso

1. **Inicialización**:
   - $i = 0$, $j = 4$
   - $S_{\text{Sophia}} = 0$, $S_{\text{Mateo}} = 0$
   - `elecciones` = []
2. **Turno de Sophia**:
   - Opciones: $coins[i] = 3$, $coins[j] = 8$
   - Como $3 < 8$, Sophia elige la última moneda (8).
   - $S_{\text{Sophia}} += 8 \implies S_{\text{Sophia}} = 8$
   - $j -= 1 \implies j = 3$
   - `elecciones.append("Última moneda para Sophia")`
3. **Verificación**:
   - $i = 0$, $j = 3$ =>Continuar.
4. **Turno de Mateo**:

- Opciones: $coins[i] = 3$, $coins[j] = 4$
- Como $3 \leq 4$, Mateo toma la primera moneda (3).
- $S_{\text{Mateo}} += 3 \implies S_{\text{Mateo}} = 3$
- $i += 1 \implies i = 1$
- `elecciones.append("Primera moneda para Mateo")`

5. **Verificación**:
   - $i = 1$, $j = 3$ =>Continuar.
6. **Turno de Sophia**:
   - Opciones: $coins[i] = 9$, $coins[j] = 4$
   - Como $9 \geq 4$, Sophia elige la primera moneda (9).
   - $S_{\text{Sophia}} += 9 \implies S_{\text{Sophia}} = 17$
   - $i += 1 \implies i = 2$
   - `elecciones.append("Primera moneda para Sophia")`
7. **Verificación**:
   - $i = 2$, $j = 3$ =>Continuar.
8. **Turno de Mateo**:
   - Opciones: $coins[i] = 2$, $coins[j] = 4$
   - Como $2 \leq 4$, Mateo toma la primera moneda (2).
   - $S_{\text{Mateo}} += 2 \implies S_{\text{Mateo}} = 5$
   - $i += 1 \implies i = 3$
   - `elecciones.append("Primera moneda para Mateo")`
9. **Verificación**:
   - $i = 3$, $j = 3$ =>Continuar.
10. **Turno de Sophia**:
    - Única opción: $coins[i] = 4$
    - Sophia toma la primera moneda (4).
    - $S_{\text{Sophia}} += 4 \implies S_{\text{Sophia}} = 21$
    - $i += 1 \implies i = 4$
    - `elecciones.append("Primera moneda para Sophia")`
11. **Fin del Juego**:
    - $i = 4$, $j = 3$ =>$i > j$, termina el juego.

8.1.2. Resultados

- **Ganancia de Sophia**: 21
- **Ganancia de Mateo**: 5
- **Elecciones**:
  - Última moneda para Sophia
  - Primera moneda para Mateo
  - Primera moneda para Sophia
  - Primera moneda para Mateo
  - Primera moneda para Sophia

8.1.3. Salida del Programa

Resultado al correr el código en Python: *Para poder ejecutar el código tenemos que abrir un terminal y navegar a la carpeta Parte_1 donde se encuentra el archivo Parte1.py y ejecutar el siguiente comando:*

```
python Parte1.py TP1_data/ejemplo1.txt
```

- Última moneda para Sophia; Primera moneda para Mateo; Primera moneda para Sophia; Primera moneda para Mateo; Primera moneda para Sophia
- Ganancia de Sophia: 21
- Ganancia de Mateo: 5

8.2. Ejemplo 2

Monedas: [5, 5, 5, 5]

8.2.1. Ejecución Paso a Paso

1. **Inicialización**:
   - $i = 0$, $j = 3$
   - $S_{\text{Sophia}} = 0$, $S_{\text{Mateo}} = 0$
   - `elecciones` = []
2. **Turno de Sophia**:
   - Opciones: 5 y 5
   - Sophia elige la primera moneda (5).
   - $S_{\text{Sophia}} += 5 \implies S_{\text{Sophia}} = 5$
   - $i += 1 \implies i = 1$
   - `elecciones.append("Primera moneda para Sophia")`
3. **Verificación**:
   - $i = 1$, $j = 3$ =>Continuar.
4. **Turno de Mateo**:
   - Opciones: 5 y 5
   - Mateo toma la primera moneda (5).
   - $S_{\text{Mateo}} += 5 \implies S_{\text{Mateo}} = 5$
   - $i += 1 \implies i = 2$
   - `elecciones.append("Primera moneda para Mateo")`
5. **Verificación**:
   - $i = 2$, $j = 3$ =>Continuar.
6. **Turno de Sophia**:
   - Opciones: 5 y 5
   - Sophia elige la primera moneda (5).
   - $S_{\text{Sophia}} += 5 \implies S_{\text{Sophia}} = 10$
   - $i += 1 \implies i = 3$
   - `elecciones.append("Primera moneda para Sophia")`
7. **Verificación**:
   - $i = 3$, $j = 3$ =>Continuar.

8. **Turno de Mateo**:
   - Única opción: 5
   - Mateo toma la primera moneda (5).
   - $S_{\text{Mateo}} += 5 \implies S_{\text{Mateo}} = 10$
   - $i += 1 \implies i = 4$
   - `elecciones.append("Primera moneda para Mateo")`
9. **Fin del Juego**:
   - $i = 4$, $j = 3 => i > j$, termina el juego.

8.2.2. Resultados

- **Ganancia de Sophia**: 10
- **Ganancia de Mateo**: 10
- **Elecciones**:
  - Primera moneda para Sophia
  - Primera moneda para Mateo
  - Primera moneda para Sophia
  - Primera moneda para Mateo

8.2.3. Salida del Programa

Resultado al correr el código en Python: *Para poder ejecutar el código tenemos que abrir un terminal y navegar a la carpeta Parte_1 donde se encuentra el archivo Parte1.py y ejecutar el siguiente comando:*

```
python Parte1.py TP1_data/ejemplo2.txt
```

- Primera moneda para Sophia; Primera moneda para Mateo; Primera moneda para Sophia; Primera moneda para Mateo
- Ganancia de Sophia: 10
- Ganancia de Mateo: 10

8.3. Ejemplo 3

Monedas: [1, 100, 1]

8.3.1. Ejecución Paso a Paso

1. **Inicialización**:
   - $i = 0$, $j = 2$
   - $S_{\text{Sophia}} = 0$, $S_{\text{Mateo}} = 0$
   - `elecciones` = []
2. **Turno de Sophia**:
   - Opciones: 1 y 1
   - Sophia elige la primera moneda (1).
   - $S_{\text{Sophia}} += 1 \implies S_{\text{Sophia}} = 1$
   - $i += 1 \implies i = 1$
   - `elecciones.append("Primera moneda para Sophia")`
3. **Verificación**:

- $i = 1$, $j = 2$ =>Continuar.

4. **Turno de Mateo**:
   - Opciones: 100 y 1
   - Como $100 > 1$, Mateo toma la última moneda (1).
   - $S_{\text{Mateo}} += 1 \implies S_{\text{Mateo}} = 1$
   - $j -= 1 \implies j = 1$
   - `elecciones.append("Última moneda para Mateo")`
5. **Verificación**:
   - $i = 1$, $j = 1$ =>Continuar.
6. **Turno de Sophia**:
   - Única opción: 100
   - Sophia toma la primera moneda (100).
   - $S_{\text{Sophia}} += 100 \implies S_{\text{Sophia}} = 101$
   - $i += 1 \implies i = 2$
   - `elecciones.append("Primera moneda para Sophia")`
7. **Fin del Juego**:
   - $i = 2$, $j = 1$ =>$i > j$, termina el juego.

8.3.2. Resultados

- **Ganancia de Sophia**: 101
- **Ganancia de Mateo**: 1
- **Elecciones**:
  - Primera moneda para Sophia
  - Última moneda para Mateo
  - Primera moneda para Sophia

8.3.3. Salida del Programa

Resultado al correr el código en Python: *Para poder ejecutar el código tenemos que abrir un terminal y navegar a la carpeta Parte_1 donde se encuentra el archivo Parte1.py y ejecutar el siguiente comando:*

```
python Parte1.py TP1_data/ejemplo3.txt
```

- Primera moneda para Sophia; Última moneda para Mateo; Primera moneda para Sophia
- Ganancia de Sophia: 101
- Ganancia de Mateo: 1

9. Mediciones Empíricas y Análisis de Complejidad

En esta sección, realizamos mediciones de tiempos para corroborar la complejidad teórica del algoritmo `juego_monedas` implementado en la Primera Parte.

9.1. Metodología

El algoritmo tiene una complejidad teórica de $\mathcal{O}(n)$, donde n es el número de monedas. Para validar esta complejidad, realizamos mediciones empíricas de los tiempos de ejecución del algoritmo con diferentes tamaños de entrada.

9.1.1. Generación de Datos

Se generaron listas de monedas de tamaños variables desde $n = 100$ hasta $n = 1\,000\,000$, incrementando en 20 pasos equidistantes. Los valores de las monedas fueron generados aleatoriamente utilizando una distribución uniforme en el rango de 1 a 100.

9.1.2. Medición de Tiempos

Para cada tamaño n, se midió el tiempo de ejecución del algoritmo `juego_monedas` utilizando la función `time.perf_counter()` de Python, que proporciona una medición de tiempo de alta resolución.

9.2. Resultados

Los tiempos de ejecución medidos se presentan en la Figura 5.

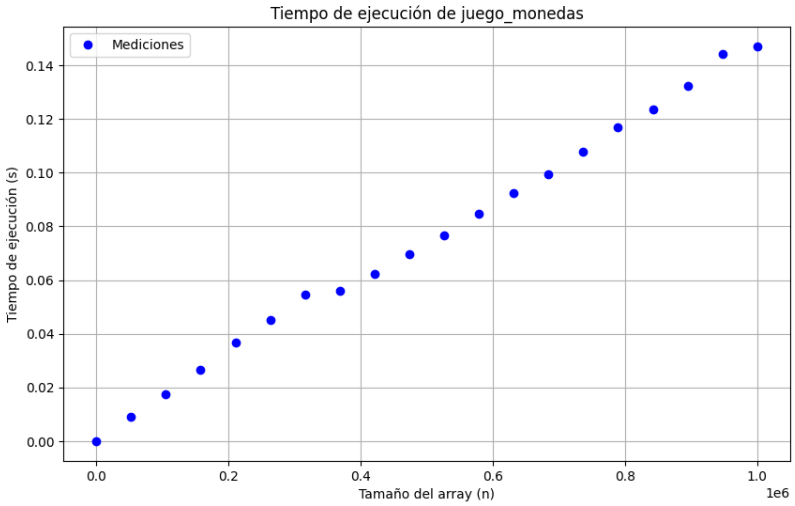

Figura 1: Tiempo de ejecución de `juego_monedas` en función del tamaño de entrada n.

9.3. Ajuste por Cuadrados Mínimos

Para corroborar que la complejidad es lineal, realizamos un ajuste por cuadrados mínimos de los datos medidos a una función lineal de la forma:

$$t(n) = c_1 n + c_2$$

donde $t(n)$ es el tiempo de ejecución en segundos, n es el tamaño de la entrada, y c_1, c_2 son coeficientes a determinar.

Utilizamos la función `curve_fit` del módulo `scipy.optimize` para obtener los coeficientes c_1 y c_2. Los resultados del ajuste fueron:

$$c_1 = 1{,}441\,249\,533\,825\,821\,7 \times 10^{-7}$$
$$c_2 = 0{,}003\,035\,661\,918\,730\,953$$

El error cuadrático total del ajuste fue de $0.000\,125\,611\,926\,081\,806\,97$.

En la Figura 6, se muestra el ajuste lineal sobre los datos medidos.

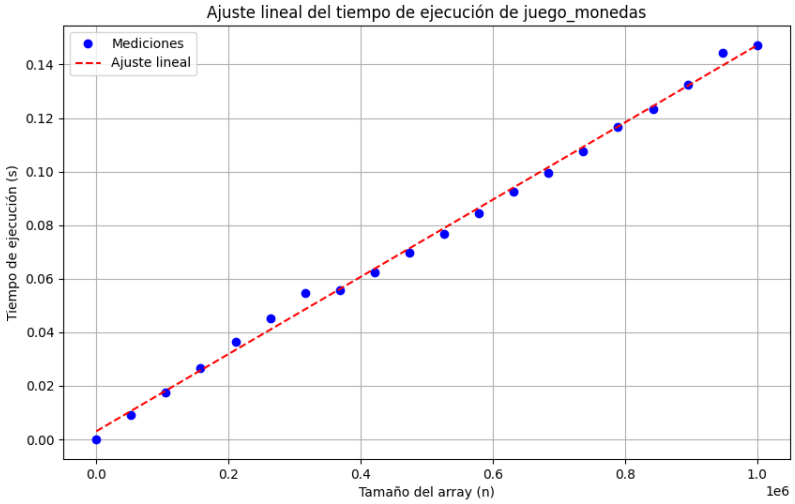

Figura 2: Ajuste lineal del tiempo de ejecución de `juego_monedas`.

9.4. Análisis de Resultados

Como se puede observar en las figuras y los resultados del ajuste:

- El tiempo de ejecución del algoritmo crece linealmente con el tamaño de la entrada n, lo cual es consistente con la complejidad teórica $\mathcal{O}(n)$.
- El bajo error cuadrático total indica que el ajuste lineal es apropiado y que los datos empíricos siguen de cerca la tendencia esperada.
- Los coeficientes c_1 y c_2 proporcionan información sobre el rendimiento del algoritmo en términos prácticos.

9.5. Código Utilizado

El código Python utilizado para realizar las mediciones y el ajuste es el siguiente:

```
    # Definición del algoritmo juego_monedas
def juego_monedas(coins):
    """
    Función que simula el juego de las monedas entre Sophia y Mateo.
    Sophia siempre elige la moneda de mayor valor para sí misma y la de menor valor
    para Mateo.
    """
    i = 0
    j = len(coins) - 1
    S_Sophia = 0
    S_Mateo = 0
    while i <= j:
        # Turno de Sophia
        if coins[i] >= coins[j]:
            S_Sophia += coins[i]
            i += 1
        else:
            S_Sophia += coins[j]
            j -= 1
        if i > j:
            break
        # Turno de Mateo
        if coins[i] <= coins[j]:
            S_Mateo += coins[i]
            i += 1
        else:
            S_Mateo += coins[j]
            j -= 1
    return S_Sophia, S_Mateo

```

```python
# Función para generar una lista de monedas aleatorias
def generar_monedas(n):
    return np.random.randint(1, 100, size=n)

# Función para medir el tiempo de ejecución del algoritmo
def medir_tiempo(n_values):
    tiempos = []
    for n in n_values:
        coins = generar_monedas(n)
        start_time = time.perf_counter()
        juego_monedas(coins)
        end_time = time.perf_counter()
        tiempos.append(end_time - start_time)
    return tiempos

# Código principal
if __name__ == "__main__":
    # Generamos valores de n desde 100 hasta 1,000,000 en 20 puntos
    n_values = np.linspace(100, 1_000_000, 20, dtype=int)
    tiempos = medir_tiempo(n_values)
```

Python Code 2: Código de las mediciones para el juego de monedas de la Parte 1

10. Conclusiones de la Primera Parte

El algoritmo *greedy* diseñado permite a Sophia garantizar su victoria en el juego de monedas al maximizar su ganancia y minimizar la de Mateo. La estrategia es óptima y eficiente, con una complejidad lineal respecto al número de monedas.

La variabilidad en los valores de las monedas no afecta la optimalidad del algoritmo, excepto en casos especiales donde las monedas tienen valores iguales y el juego puede resultar en empate.

Este estudio demuestra la efectividad de los algoritmos *greedy* en problemas donde las decisiones locales óptimas conducen a soluciones globales óptimas, especialmente cuando un jugador tiene control sobre las elecciones del oponente.

11. Parte 2: Mateo Empieza a Jugar

En esta segunda parte, Mateo ha crecido y comienza a jugar por sí mismo, aplicando estrategias *greedy* al igual que Sophia. Esto introduce un nuevo desafío para Sophia, quien ahora debe considerar las elecciones óptimas de Mateo para maximizar su propio valor acumulado. Para lograrlo, Sophia recurre a la programación dinámica.

11.1. Descripción del Nuevo Escenario

El juego sigue las mismas reglas básicas: una fila de monedas $m_1, m_2, \ldots, m_n$ y en cada turno, un jugador puede tomar la primera o la última moneda. Sophia comienza el juego. Mateo, en cada uno de sus turnos, siempre elige la moneda de mayor valor disponible entre los dos extremos, aplicando una estrategia *greedy*.

El objetivo de Sophia es maximizar su valor acumulado total, sabiendo que Mateo juega de manera óptima en su propio beneficio.

11.2. Análisis del Problema y Ecuación de Recurrencia

Teniendo en cuenta que el objetivo de Sophia es maximizar el valor total acumulado de las monedas que recoge, sabiendo que Mateo juega de forma greedy, el desafío radica en que Mateo siempre tomará la mejor opción para sí mismo en cada turno. Por lo tanto, Sophia debe planificar sus movimientos considerando las respuestas óptimas de Mateo para maximizar su propio puntaje.

Planteamiento de la Ecuación de Recurrencia:

Definimos $F(i, j)$ como el valor máximo que Sophia puede acumular al jugar óptimamente con las monedas desde la posición i hasta la j en la secuencia. El estado del juego se define por los índices i y j, que representan los extremos actuales de la fila de monedas.

La recurrencia se define considerando las dos opciones que Sophia tiene en su turno:

- 1. Opción 1: Sophia elige la moneda en la posición i (la primera moneda).
  - Mateo luego elige la moneda de mayor valor entre m_{i+1} y m_j.
    - Si $m_{i+1} \geq m_j$, Mateo elige m_{i+1}, y el juego continúa con $F(i+2, j)$.
    - Si $m_{i+1} < m_j$, Mateo elige m_j, y el juego continúa con $F(i+1, j-1)$.
  - Valor acumulado para Sophia en esta opción: $m_i +$ resultado del subjuego correspondiente.
- 2. Opción 2: Sophia elige la moneda en la posición j (la última moneda).
  - Mateo luego elige la moneda de mayor valor entre m_i y m_{j-1}.
    - Si $m_i \geq m_{j-1}$, Mateo elige m_i y el juego continúa con $F(i+1, j-1)$.
    - Si $m_i < m_{j-1}$, Mateo elige m_{j-1}, y el juego continúa con $F(i, j-2)$.
  - Valor acumulado para Sophia en esta opción: $m_j +$ resultado del subjuego correspondiente.

La ecuación de recurrencia es:

$$F(i,j) = \text{máx} \begin{cases} m_i + \begin{cases} F(i+2,j), & \text{si } m_{i+1} \geq m_j \\ F(i+1,j-1), & \text{si } m_{i+1} < m_j \end{cases} \\ m_j + \begin{cases} F(i+1,j-1), & \text{si } m_i \geq m_{j-1} \\ F(i,j-2), & \text{si } m_i < m_{j-1} \end{cases} \end{cases}$$

Casos Base:

- Si $i > j$, no hay monedas restantes: $F(i, j) = 0$.
- Si $i = j$, solo queda una moneda, que Sophia toma: $F(i, j) = m_i$.

11.3. Diseño del Algoritmo de Programación Dinámica

El algoritmo utiliza una matriz de programación dinámica DP para almacenar los valores de $F(i, j)$, evitando cálculos redundantes. Se llena la matriz de manera iterativa, considerando intervalos crecientes de monedas.

```
    def juego_monedas(coins):
    """
    Función que simula el juego de las monedas entre Sophia y Mateo utilizando
    programación dinámica.
    Sophia busca maximizar su ganancia sabiendo que Mateo siempre elige la moneda de
    mayor valor
    entre las opciones disponibles en sus turnos.
    Parámetros:
    - coins: Lista de enteros que representa los valores de las monedas en fila.
    Retorna:
    - S_Sophia: Suma total acumulada por Sophia.
    - S_Mateo: Suma total acumulada por Mateo.
    - elecciones: Lista de elecciones en el orden solicitado.
    """
    n = len(coins)
    # Matrices para programación dinámica y para rastrear las elecciones
    DP = [[-1] * n for _ in range(n)]       # Matriz para almacenar las ganancias má
    ximas de Sophia
    move = [[None] * n for _ in range(n)]   # Matriz para almacenar las elecciones de
    Sophia ('i' o 'j')

    #imprimir_dp(DP, n)

    for intervalo in range(1, n + 1):
        for i in range(n - intervalo + 1):
            j = i + intervalo - 1  # j es el final del subintervalo

            if i == j:
                DP[i][j] = coins[i]
                move[i][j] = 'i'
            else:
                # Opción 1: Sophia elige la moneda en posición i
                # Simulamos la elección de Mateo
                if i + 1 <= j:
                    if coins[i + 1] >= coins[j]:
                        # Mateo elige la moneda en posición i + 1
                        new_i, new_j = i + 2, j
                    else:
                        # Mateo elige la moneda en posición j
                        new_i, new_j = i + 1, j - 1
                else:
                    # No hay monedas para que Mateo elija
                    new_i, new_j = i + 1, j

                if new_i >= n:
                    new_i = new_i - n

                if new_j >= n:
                    new_j = new_j - n

                option1 = coins[i] + DP[new_i][new_j]

                # Opción 2: Sophia elige la moneda en posición j
                # Simulamos la elección de Mateo
                if i <= j - 1:
                    if coins[i] >= coins[j - 1]:
                        # Mateo elige la moneda en posición i
                        new_i2, new_j2 = i + 1, j - 1
                    else:
                        # Mateo elige la moneda en posición j - 1
                        new_i2, new_j2 = i, j - 2
                else:
                    # No hay monedas para que Mateo elija
                    new_i2, new_j2 = i, j - 1

                if new_i2 >= n:
                    new_i2 = new_i2 - n

                if new_j2 >= n:
```

```
                    new_j2 = new_j2 - n

                option2 = coins[j] + DP[new_i2][new_j2]

                # Elegimos la mejor opción para Sophia
                if option1 >= option2:
                    DP[i][j] = option1
                    move[i][j] = 'i'
                else:
                    DP[i][j] = option2
                    move[i][j] = 'j'


    # Reconstruimos las elecciones de Sophia y Mateo
    elecciones = []
    i, j = 0, n - 1
    S_Sophia, S_Mateo = 0, 0

    while i <= j:
        if move[i][j] == 'i':
            # Sophia elige la moneda en posición i
            S_Sophia += coins[i]
            elecciones.append(f"Primera moneda para Sophia")
            # Turno de Mateo
            if i + 1 <= j:
                if coins[i + 1] >= coins[j]:
                    # Mateo elige la moneda en posición i + 1
                    S_Mateo += coins[i + 1]
                    elecciones.append(f"Primera moneda para Mateo")
                    i += 2
                else:
                    # Mateo elige la moneda en posición j
                    S_Mateo += coins[j]
                    elecciones.append(f"Última moneda para Mateo")
                    i += 1
                    j -= 1
            else:
                # No hay monedas para Mateo
                i += 1
        else:
            # Sophia elige la moneda en posición j
            S_Sophia += coins[j]
            elecciones.append(f"Última moneda para Sophia")
            # Turno de Mateo
            if i <= j - 1:
                if coins[i] >= coins[j - 1]:
                    # Mateo elige la moneda en posición i
                    S_Mateo += coins[i]
                    elecciones.append(f"Primera moneda para Mateo")
                    i += 1
                    j -= 1
                else:
                    # Mateo elige la moneda en posición j - 1
                    S_Mateo += coins[j - 1]
                    elecciones.append(f"Última moneda para Mateo")
                    j -= 2
            else:
                # No hay monedas para Mateo
                j -= 1

    #imprimir_dp(DP, n)

    return S_Sophia, S_Mateo, elecciones
```

Python Code 3: Algoritmo de Programación Dinámica para el juego de monedas

12. Demostración para la Ecuación de Recurrencia

La ecuación de recurrencia planteada es esencial para garantizar que Sophia obtenga el máximo valor acumulado posible, considerando que Mateo juega de forma *greedy*. A continuación, demostraremos que esta recurrencia efectivamente conduce al valor óptimo para Sophia.

12.1. Definición Formal del Problema

Consideremos una secuencia de monedas $m_1, m_2, \ldots, m_n$. Sophia y Mateo alternan turnos para elegir monedas de los extremos de la secuencia. Sophia juega primero y busca maximizar su ganancia total, mientras que Mateo, en su turno, siempre elige la moneda de mayor valor disponible entre los extremos (estrategia *greedy*).

Nuestro objetivo es encontrar $F(i, j)$, el valor máximo que Sophia puede acumular al jugar óptimamente desde la posición i hasta la j, sabiendo que Mateo responde de manera *greedy*.

12.2. Demostración por Inducción

Demostraremos que la ecuación de recurrencia propuesta calcula correctamente $F(i, j)$ para todos los subintervalos $[i, j]$, lo que garantiza que Sophia obtenga el máximo valor posible.

Base de la Inducción Para el caso base, cuando $i > j$, no hay monedas restantes, por lo que:

$$F(i, j) = 0$$

Cuando $i = j$, solo queda una moneda, y Sophia la toma:

$$F(i, j) = m_i$$

Estos casos base son correctos y sirven como punto de partida para la inducción.

Hipótesis de Inducción Supongamos que la ecuación de recurrencia calcula correctamente $F(i', j')$ para todos los subintervalos tales que $j' - i' < k$, donde k es un entero positivo.

Paso Inductivo Consideremos el subintervalo $[i, j]$ donde $j - i = k$.

Sophia tiene dos opciones:

1. **Elegir** m_i:

   Mateo elige de manera *greedy* entre m_{i+1} y m_j.

   - Si $m_{i+1} \geq m_j$, Mateo elige m_{i+1}, y el juego continúa en el subintervalo $[i + 2, j]$.
   - Si $m_{i+1} < m_j$, Mateo elige m_j, y el juego continúa en el subintervalo $[i + 1, j - 1]$.

   El valor acumulado para Sophia es:

   $$\text{Opción 1} = m_i + F_{\text{resto}}$$

   Donde F_{resto} es el mínimo entre $F(i+2, j)$ y $F(i+1, j-1)$, dependiendo de la elección de Mateo.

2. **Elegir** m_j:

   Mateo elige de manera *greedy* entre m_i y m_{j-1}.

   - Si $m_i \geq m_{j-1}$, Mateo elige m_i, y el juego continúa en el subintervalo $[i + 1, j - 1]$.
   - Si $m_i < m_{j-1}$, Mateo elige m_{j-1}, y el juego continúa en el subintervalo $[i, j - 2]$.

   El valor acumulado para Sophia es:

   $$\text{Opción 2} = m_j + F_{\text{resto}}$$

   Donde F_{resto} es el mínimo entre $F(i+1, j-1)$ y $F(i, j-2)$, dependiendo de la elección de Mateo.

   Sophia elige la opción que maximiza su ganancia:

   $$F(i, j) = \max(\text{Opción 1},\ \text{Opción 2})$$

Por la hipótesis de inducción, los valores de F en subintervalos más pequeños son correctos. Por lo tanto, $F(i, j)$ se calcula correctamente.

Conclusión Por inducción, la ecuación de recurrencia calcula correctamente el valor máximo que Sophia puede obtener en cada subintervalo, considerando las respuestas *greedy* de Mateo. Por lo tanto, la recurrencia nos lleva a obtener el máximo valor acumulado posible para Sophia.

13. Análisis de Complejidad del Algoritmo

El algoritmo implementado utiliza programación dinámica para calcular $F(i, j)$ para todos los subintervalos posibles de la secuencia de monedas. A continuación, analizamos la complejidad temporal y espacial.

13.1. Complejidad Temporal

El algoritmo consiste en los siguientes pasos:

1. **Inicialización de las matrices** `DP` y `move`:

   Ambas matrices son de tamaño $n \times n$, y su inicialización tiene una complejidad de:

   $$T_{\text{init}} = \mathcal{O}(n^2)$$

2. **Llenado de las matrices**:

   Se llena la matriz `DP` considerando todos los subintervalos posibles. El número total de subintervalos en una secuencia de longitud n es:

   $$\sum_{k=1}^{n}(n-k+1) = \frac{n(n+1)}{2}$$

   Para cada subintervalo, realizamos operaciones constantes (comparaciones y asignaciones). Por lo tanto, la complejidad es:

   $$T_{\text{llenado}} = \mathcal{O}\left(\frac{n(n+1)}{2}\right) = \mathcal{O}(n^2)$$

3. **Reconstrucción de las elecciones**:

   El proceso de reconstrucción recorre desde $i = 0$ hasta $i = n$, avanzando uno o dos pasos en cada iteración. Por lo tanto, la complejidad es lineal:

   $$T_{\text{reconstrucción}} = \mathcal{O}(n)$$

Complejidad Total Sumando todas las etapas:

$$T_{\text{total}} = T_{\text{init}} + T_{\text{llenado}} + T_{\text{reconstrucción}} = \mathcal{O}(n^2) + \mathcal{O}(n^2) + \mathcal{O}(n) = \mathcal{O}(n^2)$$

13.2. Complejidad Espacial

El algoritmo utiliza dos matrices de tamaño $n \times n$:

$$S_{\text{total}} = \mathcal{O}(n^2)$$

14. Impacto de la Variabilidad de los Valores de las Monedas

La variabilidad en los valores de las monedas no afecta la complejidad temporal o espacial del algoritmo. Esto se debe a que:

- El número de operaciones realizadas por el algoritmo depende únicamente del tamaño de la entrada n, es decir, del número de monedas.
- Las operaciones realizadas en cada paso (comparaciones y sumas) son de tiempo constante, independientemente del valor de las monedas.

- La estructura de la matriz `DP` y el número de subintervalos a considerar permanecen iguales sin importar los valores específicos de las monedas.

Por lo tanto, la complejidad del algoritmo es $\mathcal{O}(n^2)$, y la variabilidad en los valores de las monedas no tiene un impacto significativo en los tiempos de ejecución.

14.1. Ejemplos y Casos de Prueba

14.1.1. Ejemplo 1

Monedas: [1, 10, 5]
Aplicación del algoritmo y seguimiento de las elecciones:

- **Paso 1**: Sophia puede elegir entre 1 (posición 0) y 5 (posición 2).
- **Opción 1**: Sophia elige 1.
  - Mateo elige el máximo entre 10 y 5, que es 10.
  - Sophia toma 5 en su siguiente turno.
  - Total de Sophia: $1 + 5 = 6$
- **Opción 2**: Sophia elige 5.
  - Mateo elige el máximo entre 1 y 10, que es 10.
  - Sophia toma 1 en su siguiente turno.
  - Total de Sophia: $5 + 1 = 6$

Resultado al correr el código en Python: *Para poder ejecutar el código tenemos que abrir un terminal y navegar a la carpeta Parte_2 donde se encuentra el archivo Parte2.py y ejecutar el siguiente comando:*

```
python Parte2.py TP2_data/ejemplo1.txt
```

- Elecciones: Primera moneda para Sophia; Primera moneda para Mateo; Primera moneda para Sophia
- Ganancia de Sophia: 6
- Ganancia de Mateo: 10

14.1.2. Ejemplo 2

Monedas: [3, 9, 1, 2]
Aplicación del algoritmo:

- **Paso 1**: Sophia puede elegir entre 3 y 2.
- **Opción 1**: Sophia elige 3.
  - Mateo elige el máximo entre 9 y 2, que es 9.
  - Sophia juega con [1, 2].
  - Total de Sophia: $3 + 2 = 5$
- **Opción 2**: Sophia elige 2.
  - Mateo elige el máximo entre 3 y 1, que es 3.
  - Sophia juega con [9, 1].
  - Sophia puede obtener un total mayor en este camino.
- **Resultado**: Sophia debe elegir 2 en su primer turno para maximizar su total a 11.

Resultado al correr el código en Python: *Para poder ejecutar el código tenemos que abrir un terminal y navegar a la carpeta Parte_2 donde se encuentra el archivo Parte2.py y ejecutar el siguiente comando:*

```
python Parte2.py TP2_data/ejemplo2.txt
```

- Elecciones: Última moneda para Sophia; Primera moneda para Mateo; Primera moneda para Sophia; Primera moneda para Mateo
- Ganancia de Sophia: 11
- Ganancia de Mateo: 4

14.2. Análisis Detallado del Algoritmo y Ejemplo Práctico

14.2.1. Análisis de la Ecuación de Recurrencia

La ecuación de recurrencia se puede visualizar como un árbol binario balanceado, donde cada nodo representa el valor máximo que Sophia puede obtener a partir de un intervalo específico de monedas. El nodo raíz corresponde al intervalo completo de monedas disponibles. Para calcular el valor en cada nodo, se suma el valor de la moneda seleccionada por Sophia (ya sea la primera o la última) al mínimo valor que puede obtener de los subintervalos resultantes, considerando las elecciones óptimas de Mateo.

En cada nivel del árbol:

- **Nodo actual**: Representa el intervalo de monedas desde i hasta j.
- **Opciones de Sophia**:
  - Tomar la primera moneda (m_i): Luego, Mateo toma la mejor opción para él, y el juego continúa en un subintervalo reducido.
  - Tomar la última moneda (m_j): Mateo responde de manera similar, y el juego continúa en otro subintervalo.

Los nodos hoja corresponden a intervalos de una sola moneda o a situaciones donde no quedan monedas, sirviendo como casos base de la recurrencia.

14.2.2. Descripción Detallada del Algoritmo

El algoritmo se divide en tres pasos principales:

Inicialización Se crean dos matrices de tamaño $n \times n$:

- **Matriz** `DP`: Almacena los valores máximos que Sophia puede obtener para cada subintervalo $[i, j]$.
- **Matriz** `decisiones`: Registra las elecciones de Sophia en cada subintervalo, indicando si debe tomar la primera ('i') o la última moneda ('j').

Ambas matrices se inicializan con valores apropiados (ceros y `None`, respectivamente).

Llenado de las Matrices Se procede a llenar las matrices de manera iterativa, considerando intervalos crecientes de monedas:

1. **Intervalos de una moneda ($i = j$)**:
   - `DP[i][j]` se asigna el valor de la única moneda disponible (m_i).
   - `decisiones[i][j]` se establece en 'i', ya que Sophia solo puede tomar esa moneda.
2. **Intervalos de dos o más monedas ($i < j$)**:
   - Para cada intervalo $[i, j]$, se calculan las dos opciones posibles para Sophia:
     - **Opción 1**: Tomar la primera moneda (m_i), y luego considerar la respuesta óptima de Mateo.
     - **Opción 2**: Tomar la última moneda (m_j), y proceder de manera similar.
   - Se simula la elección de Mateo, quien siempre toma la moneda de mayor valor disponible entre los extremos.

- Se calcula el valor acumulado para Sophia en cada opción, sumando el valor de la moneda elegida y el resultado del subintervalo correspondiente.
- Se elige la opción que maximiza el valor para Sophia y se actualizan las matrices `DP` y `decisiones` en la posición $[i][j]$.

Este proceso se repite para todos los subintervalos, llenando completamente las matrices. El valor máximo que Sophia puede obtener se encuentra en `DP[0][n-1]`.

Reconstrucción de las Elecciones Utilizando la matriz `decisiones`, se reconstruye la secuencia de elecciones de Sophia y Mateo:

- Se comienza desde el intervalo $[0, n-1]$.
- En cada paso, se verifica si `decisiones[i][j]` es 'i' o 'j':
  - Si es 'i', Sophia toma la primera moneda, y el intervalo se actualiza a $[i+1, j]$.
  - Si es 'j', Sophia toma la última moneda, y el intervalo se actualiza a $[i, j-1]$.
- Se simula la elección de Mateo en cada turno, quien toma la moneda de mayor valor entre las disponibles.
- Se registran las elecciones y se acumulan las ganancias de ambos jugadores.

14.2.3. Ejemplo Práctico

Consideremos el siguiente conjunto de monedas:

$$\text{Monedas} : [96,\ 594,\ 437,\ 674,\ 950]$$

Inicialización Las matrices se inicializan de la siguiente manera:

- `DP`:

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

- `decisiones`:

$$\begin{bmatrix} \text{None} & \text{None} & \text{None} & \text{None} & \text{None} \\ \text{None} & \text{None} & \text{None} & \text{None} & \text{None} \\ \text{None} & \text{None} & \text{None} & \text{None} & \text{None} \\ \text{None} & \text{None} & \text{None} & \text{None} & \text{None} \\ \text{None} & \text{None} & \text{None} & \text{None} & \text{None} \end{bmatrix}$$

Iteración 1: Intervalos de una moneda Se llenan las diagonales principales:

$$\begin{aligned} &\texttt{DP[0][0]} = 96, && \texttt{decisiones[0][0]} = i \\ &\texttt{DP[1][1]} = 594, && \texttt{decisiones[1][1]} = i \\ &\texttt{DP[2][2]} = 437, && \texttt{decisiones[2][2]} = i \\ &\texttt{DP[3][3]} = 674, && \texttt{decisiones[3][3]} = i \\ &\texttt{DP[4][4]} = 950, && \texttt{decisiones[4][4]} = i \end{aligned}$$

Las matrices actualizadas son:

- `DP`:

$$\begin{bmatrix} 96 & 0 & 0 & 0 & 0 \\ 0 & 594 & 0 & 0 & 0 \\ 0 & 0 & 437 & 0 & 0 \\ 0 & 0 & 0 & 674 & 0 \\ 0 & 0 & 0 & 0 & 950 \end{bmatrix}$$

- `decisiones`:

$$\begin{bmatrix} 'i' & \text{None} & \text{None} & \text{None} & \text{None} \\ \text{None} & 'i' & \text{None} & \text{None} & \text{None} \\ \text{None} & \text{None} & 'i' & \text{None} & \text{None} \\ \text{None} & \text{None} & \text{None} & 'i' & \text{None} \\ \text{None} & \text{None} & \text{None} & \text{None} & 'i' \end{bmatrix}$$

Iteración 2: Intervalos de dos monedas Ejemplo para el intervalo $[0, 1]$ (monedas $[96,\ 594]$):

- **Opción 1** (Sophia toma 96):
  - Mateo toma 594 (mayor entre 594 y 594).
  - Valor para Sophia: $96 + 0 = 96$ (no quedan más monedas).
- **Opción 2** (Sophia toma 594):
  - Mateo toma 96.
  - Valor para Sophia: $594 + 0 = 594$.
- **Elección**: Sophia elige la opción 2, ya que $594 > 96$.
- Actualización de las matrices:

$$\texttt{DP[0][1]} = 594, \quad \texttt{decisiones[0][1]} = 'j'$$

Este proceso se repite para todos los intervalos de dos monedas.

Iteraciones Subsiguientes Se continúa llenando las matrices para intervalos más largos, aplicando la misma lógica en cada paso.

Al finalizar, las matrices quedan:

- `DP`:

$$\begin{bmatrix} 96 & 594 & 533 & 1268 & \mathbf{1483} \\ 0 & 594 & 594 & 1111 & 1544 \\ 0 & 0 & 437 & 674 & 1387 \\ 0 & 0 & 0 & 674 & 950 \\ 0 & 0 & 0 & 0 & 950 \end{bmatrix}$$

- `decisiones`:

$$\begin{bmatrix} 'i' & 'j' & 'i' & 'j' & 'j' \\ \text{None} & 'i' & 'i' & 'j' & 'j' \\ \text{None} & \text{None} & 'i' & 'j' & 'j' \\ \text{None} & \text{None} & \text{None} & 'i' & 'j' \\ \text{None} & \text{None} & \text{None} & \text{None} & 'i' \end{bmatrix}$$

El valor máximo que Sophia puede obtener es **1483**, ubicado en `DP[0][4]`.

Reconstrucción de las Elecciones Siguiendo la matriz `decisiones`:

1. **Intervalo** $[0, 4]$, `decisiones[0][4]` = **'j'**:
   - Sophia toma la última moneda (950).
   - Monedas restantes: [96, 594, 437, 674].
   - Mateo elige entre $m_i = 96$ y $m_{j-1} = 674$. Como $674 > 96$, Mateo toma 674.
   - Actualizamos el intervalo:
     - Sophia tomó m_j, y Mateo tomó m_{j-1}.
     - Nuevo intervalo: $[i, j-2] = [0, 2]$.
2. **Intervalo** $[0, 2]$, `decisiones[0][2]` = **'i'**:
   - Sophia toma la primera moneda (96).
   - Monedas restantes: [594, 437].
   - Mateo elige entre $m_{i+1} = 594$ y $m_j = 437$. Como $594 > 437$, Mateo toma 594.
   - Actualizamos el intervalo:
     - Sophia tomó m_i, y Mateo tomó m_{i+1}.
     - Nuevo intervalo: $[i+2, j] = [2, 2]$.
3. **Intervalo** $[2, 2]$, `decisiones[2][2]` = **'i'**:
   - Sophia toma la moneda restante (437).
   - No quedan monedas para Mateo.

Resultados

- **Ganancia de Sophia**: $950 + 96 + 437 = 1483$.
- **Ganancia de Mateo**: $674 + 594 = 1268$.
- **Ganadora**: Sophia.

Resultado al correr el código en Python:

```
python Parte2.py TP2_data/ejemplo3.txt
```

- Última moneda para Sophia; Última moneda para Mateo; Primera moneda para Sophia; Primera moneda para Mateo; Primera moneda para Sophia
- Ganancia de Sophia: 1483
- Ganancia de Mateo: 1268

14.2.4. Conclusiones del Ejemplo

Este ejemplo práctico ilustra cómo el algoritmo permite a Sophia maximizar su ganancia total considerando las elecciones óptimas de Mateo. A través del llenado iterativo de las matrices y la posterior reconstrucción de las elecciones, se demuestra la eficacia de la programación dinámica en este tipo de problemas.

14.3. Observaciones Finales

Es importante destacar que la elección de Sophia en cada paso está influenciada por las posibles respuestas de Mateo. Al considerar las estrategias del oponente, Sophia puede planificar sus movimientos para garantizar el máximo beneficio posible. Este enfoque resalta la importancia de la programación dinámica en la resolución de problemas donde las decisiones futuras dependen de las elecciones actuales.

15. Mediciones Empíricas

En esta sección, realizamos mediciones de tiempos para corroborar la complejidad teórica del algoritmo de programación dinámica implementado en la Parte 2.

15.1. Metodología

El algoritmo tiene una complejidad teórica de $\mathcal{O}(n^2)$, donde n es el número de monedas. Para validar esta complejidad, realizamos mediciones empíricas de los tiempos de ejecución del algoritmo con diferentes tamaños de entrada.

15.1.1. Generación de Datos

Se generaron listas de monedas de tamaños variables desde $n = 100$ hasta $n = 10\,000$, incrementando en 20 pasos equidistantes. Los valores de las monedas fueron generados aleatoriamente utilizando una distribución uniforme en el rango de 1 a 100.

15.1.2. Medición de Tiempos

Para cada tamaño n, se midió el tiempo de ejecución del algoritmo `juego_monedas_dp` utilizando la función `time.perf_counter()` de Python, que proporciona una medición de tiempo de alta resolución.

15.2. Resultados

Los tiempos de ejecución medidos se presentan en la Figura 3.

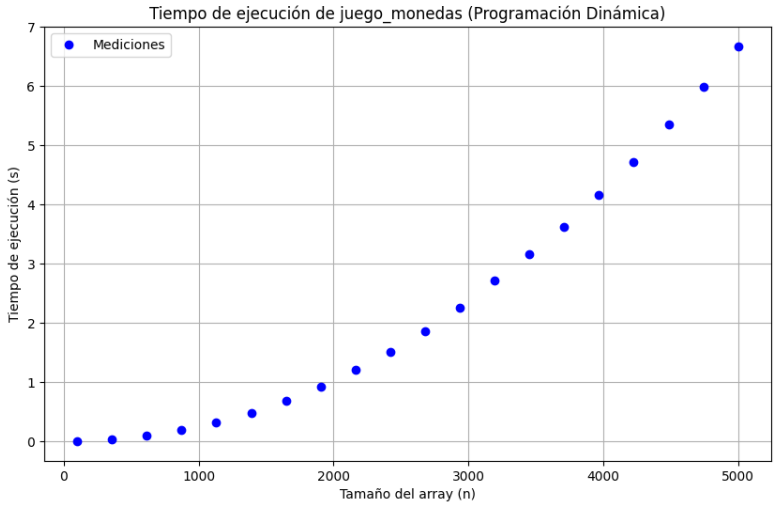

Figura 3: Tiempo de ejecución de `juego_monedas_dp` en función del tamaño de entrada n.

15.3. Ajuste por Cuadrados Mínimos

Para corroborar que la complejidad es cuadrática, realizamos un ajuste por cuadrados mínimos de los datos medidos a una función cuadrática de la forma:

$$t(n) = c_1 n^2 + c_2$$

donde $t(n)$ es el tiempo de ejecución en segundos, n es el tamaño de la entrada, y c_1, c_2 son coeficientes a determinar.

Utilizamos la función `curve_fit` del módulo `scipy.optimize` para obtener los coeficientes c_1 y c_2. Los resultados del ajuste fueron:

$$c_1 = 2{,}668\,539\,228\,140\,595 \times 10^{-7}$$
$$c_2 = -0{,}026\,639\,818\,115\,517\,37$$

El error cuadrático total del ajuste fue de 0.005 648 423 353 986 891.
En la Figura 4, se muestra el ajuste cuadrático sobre los datos medidos.

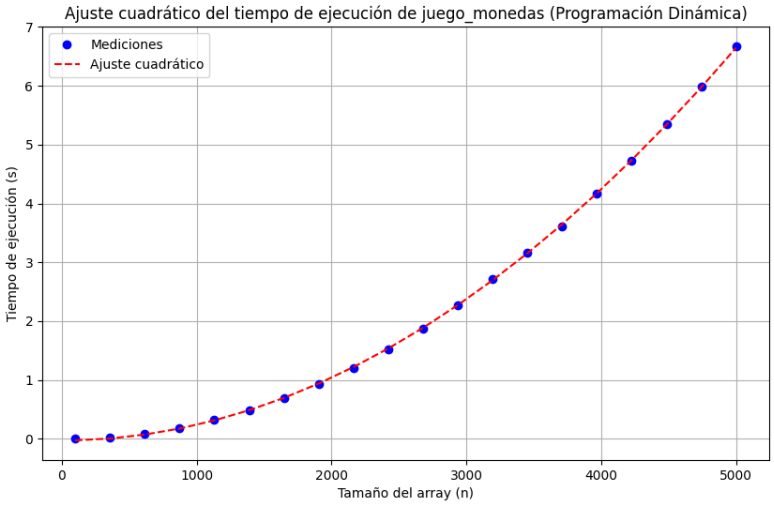

Figura 4: Ajuste cuadrático del tiempo de ejecución de `juego_monedas_dp`.

15.4. Análisis de Resultados

Como se puede observar en las figuras y los resultados del ajuste:

- El tiempo de ejecución del algoritmo crece cuadráticamente con el tamaño de la entrada n, lo cual es consistente con la complejidad teórica $\mathcal{O}(n^2)$.
- El bajo error cuadrático total indica que el ajuste cuadrático es apropiado y que los datos empíricos siguen de cerca la tendencia esperada.
- Los coeficientes c_1 y c_2 proporcionan información sobre el rendimiento del algoritmo en términos prácticos.

15.5. Código Utilizado

El código Python utilizado para realizar las mediciones y el ajuste es el siguiente:

```
import numpy as np
import time
import matplotlib.pyplot as plt
import scipy.optimize as opt
import Parte2  # Importamos tu archivo Parte2.py

# Función para generar una lista de monedas aleatorias
def generar_monedas(n):
    return np.random.randint(1, 100, size=n)

# Función para medir el tiempo de ejecución del algoritmo
def medir_tiempo(n_values):
    tiempos = []
    for n in n_values:
        coins = generar_monedas(n)
        # Medimos el tiempo de ejecución de juego_monedas
        start_time = time.perf_counter()
        Parte2.juego_monedas(coins)
        end_time = time.perf_counter()
        tiempos.append(end_time - start_time)
```

```
        return tiempos

    # Código principal
    if __name__ == "__main__":
        # Generamos valores de n desde 100 hasta 5,000 en 20 puntos
        n_values = np.linspace(100, 5000, 20, dtype=int)
        tiempos = medir_tiempo(n_values)
```

Python Code 4: Código para las mediciones para el juego de monedas

15.6. Conclusiones de la Segunda Parte

El uso de programación dinámica permite a Sophia maximizar su ganancia total en el juego, considerando las elecciones óptimas de Mateo. El algoritmo es óptimo y tiene una complejidad temporal y espacial de $\mathcal{O}(n^2)$.

Este enfoque muestra cómo la programación dinámica es efectiva en problemas donde las decisiones futuras dependen de las elecciones actuales y donde un enfoque *greedy* no es suficiente para garantizar la optimalidad.

16. Parte 3: Cambios

A medida que los hermanos crecen, exploran nuevos juegos y desafíos. En esta tercera parte, se analiza el problema de la Batalla Naval Individual, un juego que presenta interesantes retos desde el punto de vista de la complejidad computacional y el diseño de algoritmos.

17. Demostración Problema de la Batalla Naval en NP

Para demostrar que el problema de la Batalla Naval Individual (**BNI**) pertenece a la clase NP, debemos mostrar que:

- Dado una solución candidata, podemos verificar en tiempo polinomial si es una solución válida.

17.1. Definición del Problema

Dado un tablero de $n \times m$ casilleros, una lista de demandas para cada fila y columna, y una lista de barcos con sus respectivos tamaños, determinar si existe una colocación de los barcos en el tablero que cumpla con las siguientes condiciones:

1. Las demandas de las filas y columnas se satisfacen exactamente.
2. Los barcos se colocan horizontal o verticalmente sin exceder los límites del tablero.
3. Los barcos no están adyacentes entre sí, ni siquiera en diagonal.

17.2. Verificación en Tiempo Polinomial

Dado un certificado que consiste en una asignación de posiciones para cada barco:

1. **Comprobación de límites del tablero**: Verificar que cada barco está dentro del tablero. Esto se realiza en tiempo $O(k)$, donde k es el número de barcos.
2. **Verificación de no superposición**: Asegurarse de que los barcos no se superponen y no están adyacentes. Se puede crear una matriz auxiliar y marcar las casillas ocupadas y adyacentes. Esto toma tiempo $O(k \cdot \text{máx}(b_i))$, siendo b_i el tamaño del barco i.
3. **Validación de demandas**: Sumar las casillas ocupadas en cada fila y columna y compararlas con las demandas dadas. Esto requiere tiempo $O(nm)$.

Todas estas comprobaciones se pueden realizar en tiempo polinomial respecto al tamaño del tablero y la cantidad de barcos. Por lo tanto, el problema de la Batalla Naval Individual está en NP.

18. Demostración de que el Problema de la Batalla Naval es NP-Completo

Para demostrar que el problema es NP-Completo, debemos:

- Mostrar que está en NP (ya se hizo).
- Probar que es NP-Completo, mediante una reducción polinomial.

18.1. Reducción desde el Problema 3-Partition

El problema **3-Partition** es conocido por ser NP-Completo incluso en su versión unaria. Consiste en decidir si un conjunto de $3m$ números positivos puede ser dividido en m subconjuntos de 3 elementos cada uno, de manera que la suma de los números en cada subconjunto sea igual.

18.1.1. Idea de la Reducción

La idea es transformar una instancia de 3-Partition en una instancia del problema de la Batalla Naval Individual de tal manera que:

- Cada número en la instancia de 3-Partition corresponde a un barco con tamaño proporcional.
- Las demandas de filas y columnas se establecen de manera que sólo una distribución que corresponda a una partición válida satisfaga las demandas.

18.1.2. Construcción de la Reducción

Sea $A = \{a_1, a_2, \ldots, a_{3m}\}$ un conjunto de números positivos tal que la suma total es $T = mB$, donde B es la suma objetivo para cada subconjunto.

1. Crear un tablero con $n = m$ filas y m columnas.
2. Establecer las demandas de cada fila y columna en B.
3. Para cada número a_i en A, crear un barco de tamaño a_i.
4. Las restricciones del juego obligarán a que los barcos se coloquen sin superposición ni adyacencia, y las demandas exigirán que en cada fila y columna se coloque exactamente un subconjunto de barcos que sumen B.

18.1.3. Argumento de Correctitud

- **Si** existe una partición válida en 3-Partition, entonces podemos colocar los barcos en el tablero de manera que satisfagan las demandas. - **Si** podemos colocar los barcos en el tablero cumpliendo las demandas, entonces existe una partición de los números en subconjuntos de suma B.

18.1.4. Conclusión

Dado que hemos realizado una reducción polinomial al problema de la Batalla Naval Individual, y como ya se demostró que está en NP, el problema es NP-Completo.

19. Mediciones y Análisis Experimental

En esta sección se presentan las mediciones realizadas para evaluar el tiempo de ejecución del algoritmo de backtracking implementado para el problema de la Batalla Naval Individual. Se utilizó el script 'mediciones_parte3.py' para generar instancias del problema y medir los tiempos de ejecución para distintos tamaños de entrada.

19.1. Configuración de las Mediciones

El script 'mediciones_parte3.py' genera instancias aleatorias del problema para tamaños de tablero n que varían desde 2 hasta 11, debido a la complejidad exponencial del algoritmo. Para cada tamaño n, se generaron:

- Demandas de filas y columnas aleatorias entre 0 y n.
- Una lista de barcos con largos aleatorios entre 1 y un máximo determinado (por ejemplo, 5).

19.2. Resultados Obtenidos

Los tiempos de ejecución medidos para cada tamaño de tablero n se muestran en la Tabla 1.

Tamaño del tablero (n)	Tiempo de ejecución (s)
2	t_2
3	t_3
4	t_4
5	t_5
6	t_6
7	t_7
8	t_8
9	t_9
10	t_{10}
11	t_{11}

Cuadro 1: Tiempos de ejecución medidos para distintos tamaños de tablero.

19.2.1. Gráfico de los Tiempos Medidos

En la Figura 5, se presenta el gráfico de los tiempos de ejecución en función del tamaño del tablero n.

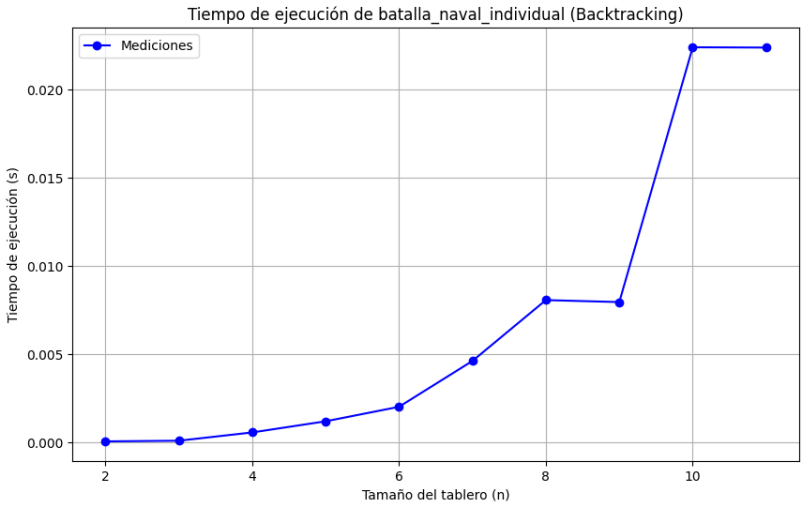

Figura 5: Tiempo de ejecución de `batalla_naval_individual` en función del tamaño del tablero n.

19.3. Análisis de los Resultados

Para analizar la complejidad temporal del algoritmo, se ajustó una función exponencial de la forma:

$$t(n) = a \cdot e^{b \cdot n}$$

donde $t(n)$ es el tiempo de ejecución para un tablero de tamaño n, y a y b son coeficientes a determinar.

Utilizando el método de mínimos cuadrados, se obtuvieron los siguientes coeficientes:

- $a = 0.0002688472297981305$
- $b = 0.4116650968683956$

El error cuadrático total del ajuste fue:

$$\text{Error cuadrático total} = 5.463264976329234\text{e-}05$$

19.3.1. Gráfico del Ajuste Exponencial

En la Figura 6, se muestra el gráfico del ajuste exponencial superpuesto a los datos medidos.

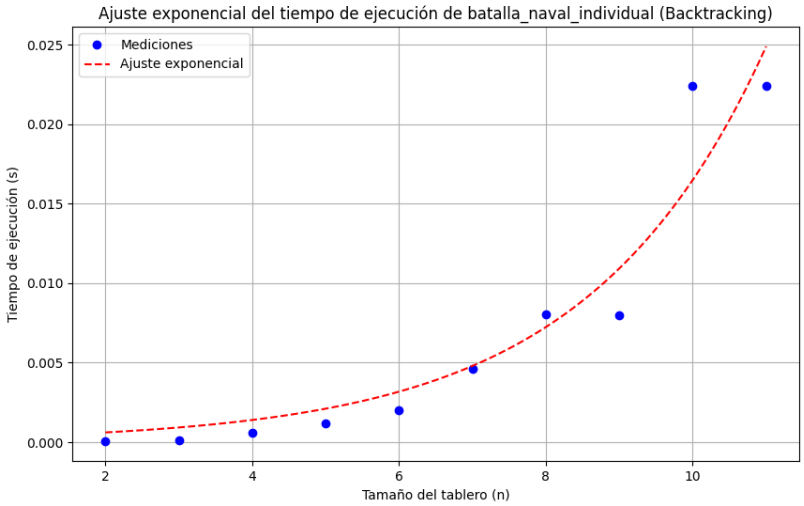

Figura 6: Ajuste exponencial del tiempo de ejecución de `batalla_naval_individual`.

19.4. Interpretación de los Resultados

Los resultados obtenidos confirman que el tiempo de ejecución del algoritmo crece exponencialmente con el tamaño del tablero n, lo cual es consistente con la complejidad esperada para un algoritmo de backtracking en un problema NP-Completo.

El valor positivo del coeficiente b en el ajuste exponencial indica un crecimiento exponencial, y el error cuadrático total bajo sugiere que el modelo se ajusta bien a los datos medidos.

19.5. Conclusiones del Análisis Experimental

El análisis experimental realizado muestra que el algoritmo, si bien es correcto y encuentra la solución óptima, no es escalable para valores grandes de n debido al crecimiento exponencial del tiempo de ejecución. Esto limita su aplicabilidad práctica a tableros de tamaño pequeño o mediano.

Para instancias de mayor tamaño, sería recomendable explorar técnicas de optimización adicionales.

19.6. Conclusiones de la Tercera Parte

El problema de la Batalla Naval Individual es NP-Completo, lo que implica que no se conoce un algoritmo polinomial para resolver todos los casos. El algoritmo de backtracking permite encontrar soluciones óptimas para problemas de tamaño moderado, pero su aplicabilidad es limitada para instancias grandes.

Este análisis resalta la importancia de reconocer problemas NP-Completo en la práctica y considerar alternativas como algoritmos aproximados o heurísticas para abordar instancias más grandes.