.UBAfiuba
FACULTAD DE INGENIERÍA

Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

Trabajo Práctico Integrador
Juegos de Hermanos

3 de diciembre de 2024

Dario Agustin Valeriano
110400

Sofia Ludmila Bica Zapata
109823

Juegos de Hermanos

1. Primera parte: Introducción y primeros años

En esta parte se abordará un problema basado en un juego con monedas de valores variados. El problema consiste en que dos jugadores, Sophia y Mateo, eligen alternadamente entre la última o la primera moneda de una fila de nn monedas, y el objetivo es obtener el mayor valor posible acumulado. Sophia, siendo muy competitiva y aprovechando que Mateo no sabe jugar (lo que le permite elegir las monedas para él, asegurándose de que el valor de las mismas sea siempre el más bajo posible), quiere asegurarse de ganar siempre. Para lograrlo, se utilizará un enfoque greedy (avaro) para resolver el problema. A lo largo de este informe se presentará el análisis teórico y empírico del algoritmo propuesto, demostrando que es capaz de obtener la solución óptima.

1.1. Análisis del problema

El juego se describe de la siguiente manera: se tiene una fila de nn monedas, cada una con un valor diferente. En cada turno, un jugador puede elegir entre dos opciones: tomar la primera moneda de la fila o la última. Al finalizar el juego, el jugador con el mayor valor acumulado de monedas es el ganador.

Sophia, quien juega primero, debe asegurarse de que siempre obtiene un valor mayor que Mateo. Para ello, se puede utilizar un algoritmo greedy. Este algoritmo toma, en cada turno, la moneda que maximice la ganancia de Sophia a corto plazo, eligiendo la de mayor valor entre la primera y la última de la fila. Además, elige la moneda que minimice la ganancia de Mateo a corto plazo, seleccionando la de menor valor entre la primera y la última para él.

1.2. Algoritmo propuesto

El algoritmo greedy propuesto se basa en la estrategia de que, en cada turno, Sophia elige la moneda de mayor valor entre la primera y la última de la fila, mientras que Mateo .elige"la de menor valor. Sophia siempre juega primero, y Mateo toma su turno siguiendo la misma regla. Este enfoque garantiza que Sophia maximice su ganancia en cada turno y, por lo tanto, asegura su victoria en todos los casos, excepto en aquellos que fueron desestimados al plantear el algoritmo: los casos en los que la cantidad de monedas es par y los valores de las monedas son iguales (en cuyo caso el resultado sería un empate).

A continuación, se presenta el código de la solución greedy para este problema.

```
def juegos_de_hermanos(monedas:deque) -> 0:
    turnos = len(monedas) // 2
    ganancia_sophia = 0
    ganancia_mateo = 0
    moneda = None
    for _ in range(turnos):
        if monedas[0] >= monedas[-1]:
            moneda = monedas.popleft()
            print("Primera moneda para Sophia -> ", moneda, "\n")
            ganancia_sophia += moneda
        else:
            moneda = monedas.pop()
            print(" ltima  moneda para Sophia -> ", moneda, "\n")
            ganancia_sophia += moneda
        if monedas[0] >= monedas[-1]:
            moneda = monedas.pop()
            print("Ultima moneda para Mateo -> ", moneda, "\n")
            ganancia_mateo += moneda
        else:
            moneda = monedas.popleft()
            print("Primera moneda para Mateo -> ", moneda, "\n")
            ganancia_mateo += moneda
    if  len(monedas) != 0:
```

```
        moneda = monedas.pop()
        print("Ultima moneda para Sophia -> ", moneda, "\n")
        ganancia_sophia += moneda
    print("Ganancia de Sophia: ", ganancia_sophia)
    print("Ganancia de Mateo: ", ganancia_mateo)
    if ganancia_sophia > ganancia_mateo:
        print("Felicidades Sophia, has ganado el juego")
    else:
        print("Parece que Mateo ha logrado hacer magia y romper el algoritmo,"
            " lo siento Sophia, no has ganado")
    return 0
```

1.2.1. Elecciones de Sophia

Las elecciones de Sophia siguen lo planteado anteriormente y se reflejan en el siguiente fragmento del código, donde además se aprecia cómo se va contabilizando, jugada a jugada, su ganancia:

```
    if monedas[0] >= monedas[-1]:
        moneda = monedas.popleft()
        print("Primera moneda para Sophia -> ", moneda, "\n")
        ganancia_sophia += moneda
    else:
        moneda = monedas.pop()
        print(" ltima  moneda para Sophia -> ", moneda, "\n")
        ganancia_sophia += moneda
```

1.2.2. Elecciones de Mateo

Las .elecciones"de Mateo también siguen lo planteado anteriormente y se reflejan en el siguiente fragmento del código, donde también se va contabilizando, jugada a jugada, su ganancia:

```
    if monedas[0] >= monedas[-1]:
        moneda = monedas.pop()
        print(" ltima  moneda para Mateo -> ", moneda, "\n")
        ganancia_mateo += moneda
    else:
        moneda = monedas.popleft()
        print("Primera moneda para Mateo -> ", moneda, "\n")
        ganancia_mateo += moneda
```

1.2.3. Cantidad par o impar de monedas

Si la cantidad de monedas es par, todas las decisiones del juego son tomadas dentro del bucle `for` del código:

```
for _ in range(turnos):
    if monedas[0] >= monedas[-1]:
        moneda = monedas.popleft()
        print("Primera moneda para Sophia -> ", moneda, "\n")
        ganancia_sophia += moneda
    else:
        moneda = monedas.pop()
        print(" ltima  moneda para Sophia -> ", moneda, "\n")
        ganancia_sophia += moneda
    if monedas[0] >= monedas[-1]:
        moneda = monedas.pop()
        print(" ltima  moneda para Mateo -> ", moneda, "\n")
        ganancia_mateo += moneda
    else:
        moneda = monedas.popleft()
        print("Primera moneda para Mateo -> ", moneda, "\n")
        ganancia_mateo += moneda
```

Por el contrario, si la cantidad de monedas es impar, la última moneda va dirigida a Sophia y el fragmento del algoritmo que le entrega la última es el siguiente:

```
if len(monedas) != 0:
    moneda = monedas.pop()
    print(" ltima  moneda para Sophia -> ", moneda, "\n")
    ganancia_sophia += moneda
```

1.2.4. A modo de broma

Dado que Sophia siempre será la ganadora, como probaremos más adelante, se añadió un comentario haciendo alusión a una "victoria"de Mateo:

```
if ganancia_sophia > ganancia_mateo:
    print("Felicidades Sophia, has ganado el juego")
else:
    print("Parece que Mateo ha logrado hacer magia y romper el algoritmo,"
        " lo siento Sophia, no has ganado")
return 0
```

1.3. Sophia siempre gana: Demostración

Vamos a demostrar que Sophia siempre gana utilizando un método de inducción matemática.

1.3.1. Caso Base:

El caso base corresponde a cuando la cantidad de monedas, n, es igual a 1. Es decir, tenemos una fila con solo una moneda disponible. Bajo esta condición, la única moneda que existe debe ser elegida por Sophia, ya que ella siempre juega primero. En este caso, la ganancia de Sophia será el valor de esa moneda, mientras que la ganancia de Mateo será 0, ya que no hay más monedas disponibles para él.

Entonces, si $n = 1$, la ganancia de Sophia es mayor que la de Mateo, y por lo tanto, Sophia gana. Esto establece que la afirmación es cierta para $n = 1$.

1.3.2. Paso Inductivo:

Ahora, asumimos que la afirmación es cierta para algún $n = k$, es decir, que si tenemos k monedas, Sophia siempre gana el juego. Nuestra tarea es demostrar que la afirmación también es válida para $n = k + 1$.

Supongamos que tenemos $k + 1$ monedas. De acuerdo con las reglas del juego, en cada turno, Sophia debe elegir la moneda de mayor valor entre la primera y la última de la fila, y Mateo elige la de menor valor entre las dos. En este escenario, podemos dividir el problema en dos casos:

1.3.3. Caso 1:

Sophia elige la primera moneda**: Si Sophia elige la primera moneda, entonces el número de monedas restantes es k, y el problema se reduce al caso de k monedas, que, por nuestra hipótesis inductiva, sabemos que Sophia ganará. La ganancia de Sophia en este caso será la ganancia obtenida en el caso k más el valor de la primera moneda que eligió.

1.3.4. Caso 2:

Sophia elige la última moneda**: Si Sophia elige la última moneda, de nuevo el número de monedas restantes es k, y el problema se reduce a k monedas, donde, por la hipótesis inductiva, Sophia ganará. Al igual que en el primer caso, la ganancia de Sophia será la ganancia obtenida en el caso k más el valor de la última moneda.

En ambos casos, el valor de las monedas restantes para k monedas garantiza que, después de que Sophia haya tomado su decisión (tomando la de mayor valor en cada turno), siempre tendrá una ganancia mayor que la de Mateo, ya que Mateo elige la moneda de menor valor.

Por lo tanto, en el caso de $k + 1$ monedas, Sophia sigue ganando, lo que completa el paso inductivo.

1.3.5. conclusión

Como hemos demostrado que si la afirmación es cierta para $n = k$, también lo es para $n = k+1$, y hemos mostrado que la afirmación es cierta para el caso base $n = 1$, podemos concluir que, por inducción matemática, Sophia siempre ganará el juego, sin importar el número de monedas n (siempre y cuando las monedas tengan valores diferentes y mayores que cero).

Por lo tanto, Sophia siempre gana.

1.4. Complejidad del Algoritmo

La complejidad temporal y espacial del algoritmo propuesto pueden analizarse de la siguiente manera:

1.4.1. Complejidad Temporal

El algoritmo realiza un número de operaciones proporcional al número de monedas, n, ya que cada turno implica una comparación y una eliminación de un elemento de la estructura de datos (una cola o deque). En cada iteración del juego, Sophia y Mateo seleccionan una moneda y la eliminan de la cola, lo cual ocurre hasta que no queden más monedas.

El tiempo que toma cada operación de comparación y eliminación es constante, $O(1)$, ya que las operaciones de acceso a los primeros y últimos elementos de un deque son $O(1)$. Como el algoritmo realiza estas operaciones en un número total de n turnos (uno por cada moneda), la complejidad temporal total del algoritmo es $O(n)$.

1.4.2. Complejidad Espacial

En cuanto a la complejidad espacial, el algoritmo utiliza una estructura de datos de tipo deque para almacenar las monedas. Esta estructura ocupa espacio proporcional al número de monedas que se almacenan, es decir, $O(n)$, ya que se almacenan n monedas en el deque a lo largo de la ejecución del algoritmo.

Por lo tanto, la complejidad espacial es también $O(n)$.

1.4.3. Fragmentos de Código Relacionados con la Complejidad

Aquí mostramos un fragmento de código donde se realizan las operaciones de comparación y eliminación de elementos:

```
# Comparacion y eliminacion de la moneda de mayor valor
if monedas[0] >= monedas[-1]:
    moneda = monedas.popleft()  # Operacion O(1)
    ganancia_sophia += moneda
else:
    moneda = monedas.pop()  # Operacion O(1)
    ganancia_sophia += moneda
```

En este fragmento, se realizan las comparaciones y las eliminaciones de las monedas en cada iteración. Cada una de estas operaciones toma un tiempo constante $O(1)$.

1.4.4. Aplicación del Teorema Maestro

Para analizar más a fondo la complejidad temporal, podemos usar el Teorema Maestro. El algoritmo puede ser modelado como una recurrencia, donde en cada paso tomamos una moneda de la cola (esto es, un número constante de operaciones) y reducimos el tamaño del problema en una unidad. Esto se puede describir con la siguiente recurrencia:

$$T(n) = T(n-1) + O(1)$$

Esta recurrencia expresa que en cada paso se hace un trabajo constante ($O(1)$), y el tamaño del problema se reduce en 1 (es decir, una moneda es eliminada). Aplicando el Teorema Maestro, el tiempo de ejecución total es:

$$T(n) = O(n)$$

Por lo tanto, la complejidad temporal del algoritmo es $O(n)$.

1.4.5. Conclusión

En resumen, tanto la complejidad temporal como la espacial del algoritmo son O(n)O(n). Esto se debe a que el algoritmo realiza un número de operaciones proporcional al número de monedas y utiliza una estructura de datos (deque) que almacena esas monedas. La eficiencia del algoritmo es adecuada para este tipo de problemas, ya que se asegura de que la ejecución crezca linealmente con el tamaño de la entrada.

2. Segunda parte: Mateo empieza a jugar

En esta parte se abordará un problema basado en el mismo juego. El problema consiste en que ambos jugadores han crecido: Mateo ha aprendido a aplicar el algoritmo codicioso (greedy), mientras que Sophia no se ha quedado atrás y ha aprendido programación dinámica. Ahora, Sophia debe aplicar esta nueva estrategia para asegurarse de obtener la mayor ganancia posible y así intentar vencer a Mateo.

2.1. Análisis del Problema

En este caso, tenemos una secuencia de monedas $m_1, m_2, \ldots, m_n$, donde Sophia y Mateo juegan turnándose para elegir monedas. Sophia siempre juega primero, y Mateo aplica una estrategia codiciosa, eligiendo siempre la moneda más grande entre la primera y la última disponible.

Sophia quiere asegurarse de obtener el valor máximo posible, lo que nos lleva a un problema de optimización. En lugar de tomar decisiones codiciosas, Sophia utilizará programación dinámica para determinar qué monedas debe elegir, con el objetivo de maximizar su valor acumulado. Esto debe hacerse teniendo en cuenta las elecciones que Mateo hará durante el juego, ya que él también juega de manera óptima. Para ello se planteó el siguiente algoritmo.

```
def juegos_de_hermanos(monedas:list) -> 0:
    primera_moneda = 0
    ultima_moneda = len(monedas) - 1
    raiz = Nodo(None,0,None,[primera_moneda,ultima_moneda])
    arbol_binario = ArbolBinario(raiz)
    nodos = cargar_arbol(raiz,monedas)
    ganancia_sophia = 0
    sumatoria_monedas = sum(monedas)
    nodo_max = None
    jugadas = []
    for nodo in nodos:
```

```
12          if nodo.suma >= ganancia_sophia:
13              nodo_max = nodo
14              ganancia_sophia = nodo.suma
15      if nodo_max.peso is None:
16          jugadas.append(["Sophia debe agarrar la ultima ->", nodo_max.valor])
17          nodo_max = nodo_max.padre
18      while nodo_max.padre is not None:
19          jugadas.append([nodo_max.peso[0] + " ->", nodo_max.peso[1]])
20          if nodo_max.padre.hijo_izq == nodo_max:
21              jugadas.append(["Sophia debe agarrar la primera ->", nodo_max.valor])
22          else:
23              jugadas.append(["Sophia debe agarrar la ultima ->", nodo_max.valor])
24          nodo_max = nodo_max.padre
25      for jugada in jugadas[::-1]:
26          print(jugada[0],jugada[1] ,"\n")
27      print("Ganancia Sophia: " ,ganancia_sophia)
28      print("Ganancia Mateo: ", sumatoria_monedas - ganancia_sophia)
29      if ganancia_sophia > sumatoria_monedas - ganancia_sophia:
30          print("Felicidades Sophia, has ganado el juego")
31      else:
32          print("Parece que Mateo ha logrado hacer magia y romper el algoritmo, lo
        siento Sophia, no has ganado")
33      return ganancia_sophia
```

2.2. Ecuación de Recurrencia

En este problema, el objetivo es encontrar la estrategia óptima para Sophia al jugar un juego de monedas contra Mateo, quien utiliza una estrategia codiciosa. La estrategia óptima para Sophia puede ser modelada mediante un enfoque de programación dinámica.

En cada momento del juego, Sophia tiene dos opciones: tomar la primera moneda o tomar la última moneda de la secuencia disponible. Mateo, por su parte, siempre elegirá la moneda que maximice su ganancia entre las opciones disponibles. Esto hace que Sophia deba anticipar las jugadas de Mateo y tomar decisiones en consecuencia.

Definimos el estado del juego mediante dos índices: `primera` y `ultima`, que representan la posición de la primera y la última moneda disponibles. El valor de `DP(primera, ultima)` es el valor máximo que Sophia puede obtener a partir de ese estado, considerando las decisiones óptimas que se toman a lo largo del juego.

La ecuación de recurrencia que describe este problema es la siguiente:

$$DP(\text{primera}, \text{ultima}) = \max\left(\text{monedas}[\text{primera}] + \min\left(DP(\text{primera}+2, \text{ultima}), DP(\text{primera}+1, \text{ultima-1})\right),\right.$$
$$\left.\text{monedas}[\text{ultima}] + \min\left(DP(\text{primera}+1, \text{ultima-1}), DP(\text{primera}, \text{ultima-2})\right)\right)$$

Esta ecuación refleja dos posibles escenarios:

2.2.1. 1. Sophia elige la primera moneda

En este caso, el valor máximo que puede obtener es la moneda `monedas[primera]` más el valor mínimo entre los siguientes dos estados: - $DP(\text{primera}+2, \text{ultima})$: El estado después de que Mateo juega de forma óptima al elegir entre `monedas[primera+1]` y `monedas[ultima]`. - $DP(\text{primera}+1, \text{ultima-1})$: El estado después de que Mateo juega de forma óptima al elegir entre `monedas[primera+1]` y `monedas[ultima-1]`.

2.2.2. 2. Sophia elige la última moneda

En este caso, el valor máximo que puede obtener es la moneda `monedas[ultima]` más el valor mínimo entre los siguientes dos estados: - $DP(\text{primera}+1, \text{ultima-1})$: El estado después de

que Mateo juega de forma óptima al elegir entre `monedas[primera]` y `monedas[ultima-1]`. - $DP(\text{primera}, \text{ultima-2})$: El estado después de que Mateo juega de forma óptima al elegir entre `monedas[primera]` y `monedas[ultima-2]`.

2.2.3. Condiciones Base

Las condiciones base del problema ocurren cuando el rango de monedas se reduce a un solo valor, es decir, cuando `primera == ultima`. En ese caso, no hay más elecciones que hacer, y el valor acumulado de Sophia es simplemente el valor de la moneda restante:

$$DP(\text{primera}, \text{primera}) = \text{monedas}[\text{primera}]$$

Este enfoque de programación dinámica resuelve el problema de forma eficiente al calcular el valor máximo que Sophia puede obtener en cada posible estado del juego, considerando todas las jugadas posibles de Mateo.

A continuación, se muestra el fragmento de código donde se refleja esta lógica de la ecuación de recurrencia:

```
    for nodo in nodos:
        if nodo.suma >= ganancia_sophia:
            nodo_max = nodo
            ganancia_sophia = nodo.suma
    if nodo_max.peso is None:
        jugadas.append(["Sophia debe agarrar la ultima ->", nodo_max.valor])
        nodo_max = nodo_max.padre
    while nodo_max.padre is not None:
        jugadas.append([nodo_max.peso[0] + " ->", nodo_max.peso[1]])
        if nodo_max.padre.hijo_izq == nodo_max:
            jugadas.append(["Sophia debe agarrar la primera ->", nodo_max.valor])
        else:
            jugadas.append(["Sophia debe agarrar la ultima ->", nodo_max.valor])
        nodo_max = nodo_max.padre
```

Este fragmento de código muestra cómo se simula el juego y cómo se calcula la ganancia de Sophia en función de las decisiones óptimas tomadas en cada momento del juego. El ciclo de decisiones está basado en la lógica descrita en la ecuación de recurrencia, donde Sophia elige entre la primera o última moneda disponible y Mateo responde de manera codiciosa.

2.3. Análisis de Complejidad

El algoritmo propuesto para resolver el problema de las monedas tiene una complejidad que depende de la forma en que el árbol de decisiones crece durante el proceso de recursión. A continuación, realizamos un análisis detallado de la complejidad temporal y espacial.

2.4. Recurrencia del Algoritmo

El algoritmo resuelve el problema de forma recursiva, evaluando dos subproblemas en cada paso: uno para el caso en que el jugador toma la primera moneda y otro para el caso en que toma la última. Esto da lugar a una recurrencia en la forma:

$$T(n) = 2T(n-1) + O(n)$$

donde $T(n)$ representa el tiempo necesario para resolver el problema de tamaño n, y $O(n)$ es el trabajo realizado en cada nivel de la recursión, debido a las comparaciones y actualizaciones de las sumas acumuladas de las monedas.

2.4.1. Expansión Recursiva

Al expandir la recurrencia, obtenemos lo siguiente:

$$T(n) = 2T(n-1) + O(n)$$
$$T(n-1) = 2T(n-2) + O(n-1)$$
$$T(n-2) = 2T(n-3) + O(n-2)$$

y así sucesivamente.

Como cada subproblema genera otros dos subproblemas de tamaño decreciente, el número de subproblemas crece exponencialmente con el tamaño de la entrada.

2.4.2. Complejidad Temporal

Este tipo de recurrencia se expande de forma que el número total de operaciones crece de manera exponencial. En particular, en cada nivel de la recursión, se generan dos subproblemas, y cada uno de ellos requiere $O(n)$ trabajo para ser resuelto. Como resultado, la complejidad total es:

$$T(n) = O(2^n)$$

Por lo tanto, la complejidad temporal del algoritmo es exponencial, específicamente $O(2^n)$. Esto indica que el algoritmo tiene un rendimiento muy pobre para entradas grandes, ya que el tiempo de ejecución aumenta rápidamente a medida que aumenta el tamaño de la entrada.

2.4.3. Complejidad Espacial

La complejidad espacial del algoritmo está relacionada con el tamaño del árbol de decisiones generado. Dado que en cada paso del algoritmo se almacenan dos subproblemas, y el número total de subproblemas crece exponencialmente con el tamaño de la entrada, la complejidad espacial también es $O(2^n)$.

2.5. Conclusión

El algoritmo propuesto tiene una complejidad temporal y espacial exponencial, $O(2^n)$. Este comportamiento es característico de algoritmos recursivos que no están optimizados para evitar la resolución repetida de subproblemas, como ocurre en este caso con la exploración exhaustiva de decisiones en el árbol de recursión. Debido a esta complejidad, el algoritmo no es eficiente para entradas de gran tamaño y podría beneficiarse de técnicas como la programación dinámica para reducir la cantidad de subproblemas redundantes y mejorar la eficiencia.

2.6. Análisis de la Variabilidad de los Valores de las Monedas

La variabilidad de los valores de las monedas tiene un impacto significativo en la optimalidad del algoritmo planteado. Para entender cómo influye, es necesario considerar tanto el comportamiento de la estrategia de Sophia como la de Mateo y cómo sus decisiones dependen de los valores de las monedas disponibles en cada turno.

2.7. Estrategia de Juego y la Influencia de los Valores de las Monedas

En el juego, Sophia y Mateo alternan turnos eligiendo una moneda, y Mateo siempre elige la moneda de mayor valor disponible, ya sea la primera o la última. Mientras que Sophia también

busca maximizar su ganancia, ella debe anticipar las elecciones de Mateo, dado que su estrategia también depende de los valores de las monedas.

Si los valores de las monedas están equilibrados (es decir, las monedas tienen valores relativamente cercanos entre sí), el algoritmo es capaz de tomar decisiones que maximizan las ganancias de Sophia, ya que las elecciones de Mateo no cambian radicalmente el valor acumulado de las elecciones de Sophia. En este caso, la programación dinámica debería ser eficiente y la solución obtenida sería generalmente cercana a la óptima.

2.8. Impacto de la Alta Variabilidad en los Valores de las Monedas

Cuando existe una gran variabilidad entre los valores de las monedas (por ejemplo, monedas muy pequeñas intercaladas con monedas muy grandes), el algoritmo puede enfrentar mayores dificultades para garantizar que Sophia obtenga el máximo valor posible. Esto se debe a que las decisiones de Mateo pueden ser mucho más perjudiciales para Sophia en ciertos escenarios:

- Si una de las monedas más grandes está al principio o al final de la secuencia, Mateo la elegirá en su turno, dejando a Sophia con una selección menos beneficiosa en su siguiente turno.
- En casos donde la diferencia entre las monedas más pequeñas y más grandes es considerable, Sophia podría verse forzada a tomar decisiones subóptimas para evitar que Mateo obtenga una ganancia mayor.

Esto hace que la optimalidad del algoritmo se vea afectada, ya que Sophia podría no ser capaz de obtener la ganancia máxima debido a la estrategia greedy de Mateo. En escenarios con alta variabilidad en los valores, el algoritmo tiende a ser menos efectivo al optimizar la estrategia de Sophia, ya que la predicción de los movimientos de Mateo se vuelve más incierta.

2.9. Optimalidad en Función de los Valores de las Monedas

La optimalidad del algoritmo depende en gran medida de cómo las monedas están distribuidas en términos de valor. Si la secuencia de monedas tiene un patrón predecible o equilibrado, la programación dinámica puede aprovechar la estructura del problema para generar soluciones óptimas. Sin embargo, cuando los valores son altamente dispares, el algoritmo puede no encontrar la solución óptima debido a la estrategia de Mateo y la necesidad de anticipar sus movimientos.

2.10. Conclusión

La variabilidad de los valores de las monedas tiene un impacto directo en la capacidad del algoritmo para garantizar una solución óptima. En escenarios con monedas de valores muy dispares, el algoritmo puede no ser capaz de ofrecer una solución óptima para Sophia debido a las decisiones de Mateo. Sin embargo, en secuencias más equilibradas de monedas, el algoritmo será más efectivo y ofrecerá una estrategia de selección más cercana a la óptima para Sophia. En estos casos, la programación dinámica mejora la toma de decisiones al considerar las diferentes posibilidades y anticipar las elecciones de Mateo.

3. Tercera parte (cambios): La Batalla Naval

3.1. Demostración de que el Problema de la Batalla Naval está en NP

El problema de la Batalla Naval se encuentra en la clase NP, ya que, dado un tablero de tamaño $n \times m$ y una posible solución (es decir, una colocación de los barcos en el tablero), podemos verificar si esa solución cumple con las restricciones de filas y columnas, y las restricciones de adyacencia, en tiempo polinómico.

Para verificar una solución, realizamos dos pasos:

- **Verificación de las restricciones de filas y columnas**: Comprobamos que la cantidad de casilleros ocupados en cada fila y cada columna coincida con las restricciones dadas. Esto se puede hacer en tiempo $O(n \times m)$.
- **Verificación de las restricciones de adyacencia**: Comprobamos que los barcos no estén adyacentes (ni horizontalmente, ni verticalmente, ni diagonalmente). Esto también se puede hacer en tiempo $O(n \times m)$.

Dado que ambos pasos se realizan en tiempo polinómico, podemos concluir que el problema está en NP.

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

Para demostrar que el problema de la Batalla Naval es NP-Completo, debemos mostrar dos cosas:

1. El problema está en NP, lo cual ya hemos demostrado en la sección anterior.
2. El problema es NP-Hard, lo cual significa que podemos reducir un problema conocido NP-Completo al problema de la Batalla Naval en tiempo polinómico.

Vamos a realizar una reducción desde el problema de Bin-Packing al problema de la Batalla Naval.

3.2.1. Reducción del problema de Bin-Packing

El problema de Bin-Packing es NP-Completo. Dado un conjunto de objetos con longitudes $b_1, b_2, \ldots, b_k$ y una capacidad de bin C, debemos asignar estos objetos a bins sin exceder la capacidad C. Ahora, construimos un tablero de la Batalla Naval donde cada objeto b_i es un barco con longitud b_i, y las restricciones de filas y columnas corresponden a las capacidades de los bins.

Si podemos resolver el problema de la Batalla Naval, podemos resolver el problema de Bin-Packing, ya que colocar los barcos en el tablero es equivalente a asignar los objetos a los bins de manera válida.

Por lo tanto, como el problema de Bin-Packing es NP-Completo, podemos concluir que el problema de la Batalla Naval es también NP-Completo.

3.3. Conclusión

Hemos demostrado que el problema de la Batalla Naval:

- Está en la clase NP, ya que podemos verificar una solución en tiempo polinómico.
- Es NP-Completo, ya que hemos realizado una reducción polinómica desde el problema de Bin-Packing.