Teoría de Algoritmos

(75.29) Curso Buchwald - Genender

Trabajo Práctico 2
Programación Dinámica

14 de octubre de 2024

Valentín Schneider 107964
Ian von der Heyde 107638
Juan Martín de la Cruz 109588

Índice

1. Introducción

Cuando Mateo nació, Sophia estaba muy contenta. Finalmente tendría un hermano con quien jugar. Sophi tenía 3 años cuando Mateo nació. Ya desde muy chicos, ella jugaba mucho con su hermano.

Pasaron los años, y fueron cambiando los juegos. Cuando Mateo cumplió 4 años, el padre de ambos le explicó un juego a Sophia: Se dispone una fila de n monedas, de diferentes valores. En cada turno, un jugador debe elegir alguna moneda. Pero no puede elegir cualquiera: sólo puede elegir o bien la primera de la fila, o bien la última. Al elegirla, la remueve de la fila, y le toca luego al otro jugador, quien debe elegir otra moneda siguiendo la misma regla. Siguen agarrando monedas hasta que no quede ninguna. Quien gane será quien tenga el mayor valor acumulado (por sumatoria).

Mateo ahora tiene 7 años. Los mismos años que tenía Sophia cuando comenzaron a jugar al juego de las monedas. Eso quiere decir que Mateo también ya aprendió sobre algoritmos greedy, y lo comenzó a aplicar. Esto hace que ahora quién gane dependa más de quién comience y un tanto de suerte.

Esto no le gusta nada a Sophia. Ella quiere estar segura de ganar siempre. Lo bueno es que ella comenzó a aprender sobre programación dinámica. Ahora va a aplicar esta nueva técnica para asegurarse ganar siempre que pueda.

1.1. Consigna

- Hacer un análisis del problema, plantear la ecuación de recurrencia correspondiente y proponer un algoritmo por programación dinámica que obtenga la solución óptima al problema planteado: Dada la secuencia de monedas $m_1, m_2, ..., m_n$, sabiendo que Sophia empieza el juego y que Mateo siempre elegirá la moneda más grande para sí entre la primera y la última moneda en sus respectivos turnos, definir qué monedas debe elegir Sophia para asegurarse obtener el **máximo valor acumulado posible**. Esto no necesariamente le asegurará a Sophia ganar, ya que puede ser que esto no sea obtenible, dado por cómo juega Mateo. Por ejemplo, para [1, 10, 5], no importa lo que haga Sophia, Mateo ganará.
- Demostrar que la ecuación de recurrencia planteada en el punto anterior en efecto nos lleva a obtener el **máximo valor acumulado posible**.
- Escribir el algoritmo planteado. Describir y justificar la complejidad de dicho algoritmo. Analizar si (y cómo) afecta a los tiempos del algoritmo planteado la variabilidad de los valores de las monedas.
- Realizar ejemplos de ejecución para encontrar soluciones y corroborar lo encontrado. Adicionalmente, el curso proveerá con algunos casos particulares que deben cumplirse su optimalidad también.
- Hacer mediciones de tiempos para corroborar la complejidad teórica indicada. Agregar los casos de prueba necesarios para dicha corroboración (generando sus propios sets de datos). Esta corroboración empírica debe realizarse confeccionando gráficos correspondientes, y utilizando la técnica de cuadrados mínimos.
- Agregar cualquier conclusión que parezca relevante.

1.2. ¿Qué es Programación Dinámica?

La **programación dinámica (PD)** es una técnica de optimización utilizada para resolver problemas complejos dividiéndolos en subproblemas más pequeños y manejables. Una de las características esenciales de esta técnica es que evita recalcular soluciones a subproblemas ya resueltos,

aprovechando el concepto de **memoización**. Esto implica almacenar los resultados de subproblemas en una estructura de datos para que puedan ser reutilizados cuando sea necesario, mejorando significativamente la eficiencia del algoritmo.

Un problema adecuado para ser resuelto mediante PD presenta dos propiedades clave: subproblemas superpuestos y estructura de solución óptima. Los subproblemas superpuestos se refieren a la característica de muchos problemas en los que los mismos subproblemas aparecen una y otra vez durante el proceso de resolución. La PD se asegura de que cada subproblema se resuelva solo una vez y que su solución se almacene para ser reutilizada posteriormente. Por otro lado, la estructura de solución óptima significa que la solución óptima global del problema puede construirse a partir de las soluciones óptimas de sus subproblemas.

Existen dos enfoques principales en PD: *Top-Down* y *Bottom-Up*. En el enfoque Top-Down, también conocido como programación dinámica con memoización, el problema se descompone recursivamente, y las soluciones intermedias se almacenan para evitar recalculaciones. Este enfoque sigue un estilo recursivo. En contraste, el enfoque Bottom-Up, también conocido como tabulación, resuelve primero los subproblemas más pequeños de forma iterativa y utiliza estas soluciones para construir las soluciones de los problemas más grandes. Este último enfoque es generalmente preferido por ser más eficiente en términos de uso de memoria y tiempo de ejecución.

Una de las mayores ventajas de la programación dinámica es su capacidad para reducir la complejidad temporal de problemas que, sin optimización, tendrían un coste computacional muy elevado. Al evitar la repetición de cálculos, PD puede transformar algoritmos con una complejidad exponencial en algoritmos con una complejidad polinomial o bien pseudo-polinomiales.

2. Análisis del problema

El problema plantea un escenario en el que se dispone de una fila de n monedas, cada una con un valor asignado representado por $m_1, m_2, \ldots, m_n$. Dos jugadores, Sophia y Mateo, se alternan para seleccionar una moneda, pero solo pueden elegir la primera o la última de la fila en cada turno. El objetivo de Sophia es maximizar la suma total de las monedas que recolecta al finalizar el juego.

2.1. Ejemplo del juego

Para desarrollar una breve explicación sobre el juego planteado en el problema, daremos un ejemplo de nuestros protagonistas. Para ello, supongamos que se tiene una fila de monedas con los valores $[3, 9, 2, 7]$.

- **Turno 1:** Sophia elige la última moneda, de valor 7. Ahora la fila es $[3, 9, 2]$.
- **Turno 2:** Mateo elige la primera moneda, de valor 3. Ahora la fila es $[9, 2]$.
- **Turno 3:** Sophia elige la primera moneda, de valor 9. Ahora la fila es $[2]$.
- **Turno 4:** Mateo elige la única moneda restante, de valor 2.

Resultado: Sophia tiene un total de $7 + 9 = 16$, mientras que Mateo tiene $3 + 2 = 5$. Sophia gana el juego con un total de 16.

Al fin y al cabo, el problema de las monedas se reduce a encontrar la máxima ganancia posible para Sophia, bajo la premisa de que Sophia juega de manera óptima global y Mateo de forma óptima local (greedy). En otras palabras, lo que estamos buscando es calcular el valor máximo que Sophia puede obtener siguiendo la mejor estrategia, tomando en cuenta que Mateo tomará decisiones greedy para minimizar la ganancia de Sophia.

2.2. Consideraciones del Problema

Para cada turno, Sophia tiene dos opciones: tomar la moneda en la posición i o la moneda en la posición j. Dependiendo de la elección que haga, Mateo tomará su mejor decisión en el turno siguiente. La estrategia de Sophia debe considerar las siguientes condiciones:

Si Sophia elige la moneda m_i, entonces en el siguiente turno Mateo jugará con las monedas restantes en el intervalo $[i+1, j]$. Si Sophia elige la moneda m_j, Mateo jugará con el intervalo $[i, j-1]$.

Además, consideremos las siguientes variables para modelar el problema:

- n: número total de monedas en la fila.
- m_i: valor de la moneda en la posición i, donde $i \in [1, n]$.
- $M(i, j)$: valor máximo que puede obtener Sophia si le toca jugar en una fila que comienza en la posición i y termina en la posición j.

3. Análisis para encontrar la solución con Programación Dinámica

Para encontrar una solución usando programación dinámica, es necesario seguir un proceso sistemático que permita descomponer el problema original en subproblemas más pequeños y manejables. Esto implica identificar tres componentes clave, que son fundamentales para aplicar correctamente esta técnica:

1. Casos Base
2. Forma de los Subproblemas
3. Composición de Subproblemas

3.1. Casos base

Los casos base representan las situaciones más simples del problema, donde la solución es inmediata y no requiere mayor descomposición. En el contexto de nuestro problema de las monedas, un caso base se da cuando solo queda una moneda en la fila. Si Sophia debe elegir entre una única moneda, la elección es trivial: Sophia simplemente tomará esa moneda, ya que no tiene otras opciones. Matemáticamente, esto puede expresarse como:

$$M(i, i) = m_i$$

Donde $M(i, i)$ indica que, si la fila contiene solo la moneda en la posición i, el valor que Sophia obtiene es simplemente el valor de esa moneda, m_i.

Otro caso base ocurre cuando quedan exactamente dos monedas de distinto valor, es decir, m_i y m_j con $i \neq j$. En este caso, Sophia tomará la moneda de mayor valor:

$$M(i, j) = \text{máx}(m_i, m_j)$$

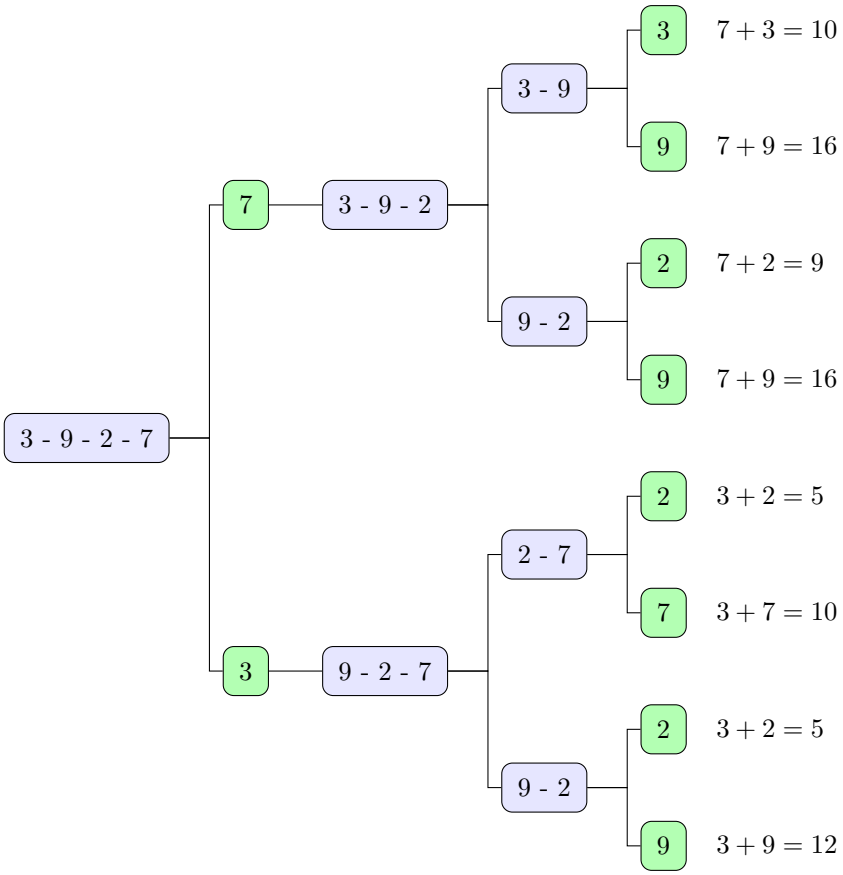
Aquí, Sophia optará por la moneda más valiosa entre las dos disponibles, maximizando su ganancia en esa jugada.

3.2. Forma de los subproblemas

Para descomponer el problema original en subproblemas, es necesario considerar las decisiones que toma Sophia en cada turno. En cada movimiento, Sophia puede elegir la primera moneda de la fila (m_i) o la última (m_j). Después de que Sophia elige una moneda, el juego continúa con Mateo eligiendo de las monedas restantes.

La clave está en ver cómo el problema de n monedas se reduce a un subproblema más pequeño cada vez que Sophia o Mateo toma una moneda. Si Sophia elige la moneda m_i, el problema se reduce a encontrar la mejor estrategia para el intervalo $[i+1, j]$. Si elige m_j, el subproblema que queda es el intervalo $[i, j-1]$. En ambos casos, el tamaño del problema se reduce, y continuamos resolviendo estos subproblemas hasta llegar a los casos base.

Veamos este planteo de forma más explícita y con el ejemplo usado anteriormente.



Esto significa que los subproblemas son de la forma $M(i, j)$, donde se trata de maximizar la ganancia de Sophia considerando únicamente las monedas entre las posiciones i y j.

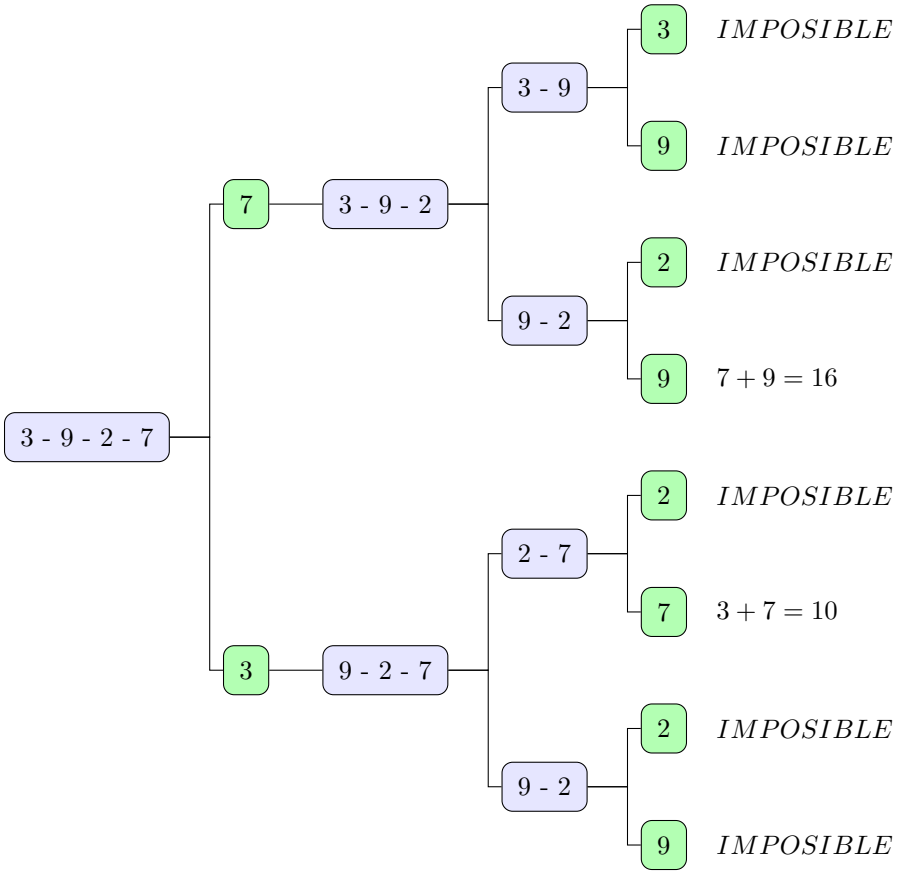
3.2.1. Consideración del oponente sabio

Dado que Mateo es sabio y nunca tomará una moneda menor si tiene una opción mayor, esto significa que ciertas configuraciones de elecciones no son posibles. Por ejemplo:

- Si $m_i > m_j$, Mateo siempre elegirá m_i, dejando a Sophia enfrentarse al subproblema restante.
- Si $m_j > m_i$, Mateo optará por m_j, y Sophia deberá resolver el subproblema con las monedas restantes.

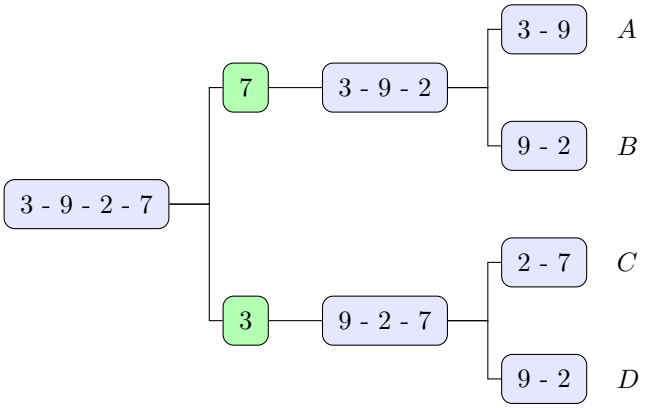
Con esta consideración, se debe actualizar la estructura de los subproblemas considerando las elecciones que Mateo hará. Para cada subproblema $M(i, j)$, Sophia debe maximizar su ganancia

mientras asume que Mateo minimizará su ganancia futura eligiendo la moneda de mayor valor disponible en su turno. Esto introduce nuevas restricciones en el problema, ya que ciertas combinaciones de elecciones no serán posibles para Sophia debido a la naturaleza óptima de las decisiones de Mateo. Veamos cómo se reducen las posibilidades en base a lo mencionado anteriormente con el ejemplo dado:



3.3. Composición de Subproblemas

Entonces, podemos observar que la elección de Sophia reduce el tamaño del subproblema en cada turno, y la secuencia de elecciones lleva finalmente a un caso base donde solo quedan dos monedas. En ese punto, el jugador actual elige la mayor. Por lo tanto, los subproblemas más pequeños resuelven situaciones donde quedan menos monedas. A medida que los subproblemas más pequeños se resuelven, la solución óptima se propaga hacia los subproblemas más grandes hasta llegar al problema completo.



La secuencia completa de monedas es $[3, 9, 2, 7]$, denotada como `monedas`(i, j), donde i y j son los índices de los extremos de la secuencia. Sophia es la primera en jugar, y Mateo sigue siempre tomando la moneda de mayor valor entre los extremos.

Sophia, al ser la primera en jugar, tiene dos opciones:

- Tomar la moneda de la derecha (valor 7): Esto lleva a una nueva subconfiguración, es decir, la secuencia $[3, 9, 2]$.
- Tomar la moneda de la izquierda (valor 3): Esto lleva a una nueva subconfiguración de monedas, es decir, la secuencia $[9, 2, 7]$.

Al tomar una moneda, el subproblema que enfrentará el siguiente jugador (Mateo) es uno de los dos subproblemas generados. Mateo también jugará óptimamente, es decir, tomará siempre la moneda de mayor valor en los extremos.

Primera Rama (Derecha - Elegir 7)

Si Sophia elige la moneda de valor 7, la nueva configuración para Mateo es $[3, 9, 2]$. Dependiendo el lado que escoga Mateo, tendremos dos nuevos posibles juegos:

- `monedas`(i, j-2), es decir $[3, 9]$, a la cual denotaremos como A
- `monedas`(i+1, j-1), es decir $[9, 2]$, a la cual denotaremos como B

Ahora bien, como sabemos que el hermano de Sophia es listo, **buscará maximizar su ganancia tomando la moneda de mayor valor**. Por ende, es completamente imposible que se juegue la rama A. Esto deja la secuencia $[9, 2]$ como subproblema para Sophia (rama B).

En este caso, el subproblema se representa como:

$$M_j + OPT(B)$$

donde M_j es la moneda que Sophia eligió en la primera jugada.

Segunda Rama (Izquierda - Elegir 3)

Como vimos en el anterior punto, podemos aplicar la misma lógica para la segunda rama. Después de que Sophia elige la moneda de valor 3, la nueva configuración para Mateo es $[9, 2, 7]$. Dependiendo la moneda que escoga Mateo, Sophia tiene dos opciones:

- `monedas`(i+2, j), es decir $[2, 7]$, a la cual denotaremos como C
- `monedas`(i+1, j-1), es decir $[9, 2]$, a la cual denotaremos como D

Al igual que sucedía con la otra rama, **buscará maximizar su ganancia tomando la moneda de mayor valor**. En tal caso, la rama D es imposible y esto deja la secuencia $[9, 2]$ como subproblema para Sophia (rama C). A esto lo podemos representar como:

$$M_i + OPT(C)$$

donde M_i es la moneda que eligió Sophia en la primera jugada.

Cabe mencionar que pueden haber arreglos donde si o si pierda Sophia. Sophia analizará con esta ecuación qué moneda tomar pero Mateo, aplicando su algoritmo Greedy, podría forzarla a enfrentarse a subproblemas desfavorables. Es decir, la elección de uno de los jugadores puede generar

subproblemas que no resultan en la solución óptima para el otro jugador en turnos posteriores. Un ejemplo claro de esto sería una secuencia de monedas como [3,9,2].

Además, al intentar maximizar la suma de monedas que se obtienen, nos encontramos con que muchas configuraciones parciales del juego se repiten a medida que los jugadores alternan sus elecciones. Por ejemplo, cuando Sophia toma una moneda del extremo izquierdo o derecho, los subproblemas resultantes, que consisten en las secuencias más cortas de monedas, pueden aparecer en diferentes etapas de la toma de decisiones. Esto se ve reflejado en el árbol dado anteriormente (ramas B y D).

Entonces vemos que es importante utilizar programación dinámica por el hecho de que los mismos subproblemas aparezcan en múltiples ramas del árbol de decisiones para almacenar (*memorizar*) los resultados de estos subproblemas en lugar de recalcularlos cada vez. Al hacerlo, reducimos la complejidad computacional al evitar cálculos redundantes y aceleramos la solución del problema general, lo que es una clara ventaja en términos de eficiencia.

4. Ecuación de Recurrencia

Llegados a este punto, dedujimos que la elección que Sophia realiza depende de cuál sea la que maximice su ganancia, sabiendo que Mateo jugará con un algoritmo Greedy al tomar la moneda del mayor valor de los extremos. Ella toma la decisión inicial dependiendo de las opciones generadas por los subproblemas, maximizando su resultado final.

Entonces, Sophia tomará la decisión que maximice su ganancia y se podría modelar con la siguiente fórmula recursiva:

$$\text{OPT}[i,j] = \text{máx} \begin{cases} m_i + \text{OPT}[i+2][j] & \text{si } m_{i+1} \geq m_j, \\ m_i + \text{OPT}[i+1][j-1] & \text{si no;} \\ \\ m_j + \text{OPT}[i+1][j-1] & \text{si } m_i \geq m_{j-1}, \\ m_j + \text{OPT}[i][j-2] & \text{si no} \end{cases}$$

4.1. Términos de la ecuación

- $\mathbf{OPT}[i,j]$: Esta función representa la ganancia óptima de Sophia si está jugando sobre el subarreglo de monedas desde el índice i hasta el índice j. Sophia siempre intenta maximizar su ganancia.
- $m_i + \mathbf{OPT}[i+2][j]$: Esta expresión corresponde a la situación en la que Sophia elige la moneda del extremo izquierdo m_i. Si la siguiente moneda m_{i+1} es mayor o igual que m_j (la moneda del extremo derecho), Mateo jugará de manera *greedy* y tomará m_{i+1}, dejando a Sophia con el subarreglo desde $i+2$ hasta j, es decir, $\text{OPT}[i+2][j]$.
- $m_i + \mathbf{OPT}[i+1][j-1]$: En este caso, Sophia también toma la moneda m_i, pero Mateo selecciona m_j (porque $m_j > m_{i+1}$). Esto deja a Sophia con el subarreglo de tamaño $i+1$ a $j-1$, es decir, $\text{OPT}[i+1][j-1]$.
- $m_j + \mathbf{OPT}[i+1][j-1]$: Aquí, Sophia elige la moneda del extremo derecho m_j, y Mateo selecciona m_i (si $m_i \geq m_{j-1}$). El subarreglo restante para Sophia es $\text{OPT}[i+1][j-1]$.
- $m_j + \mathbf{OPT}[i][j-2]$: En este caso, Sophia elige m_j, pero Mateo selecciona m_{j-1} (porque $m_{j-1} > m_i$). Esto deja a Sophia con el subarreglo desde i hasta $j-2$, es decir, $\text{OPT}[i][j-2]$.

4.2. Demostración de optimalidad

Para demostrar rigurosamente por qué esta ecuación recursiva nos lleva a obtener el máximo valor acumulado, vamos a descomponer el proceso en varios pasos lógicos y matemáticos, utilizando

inducción matemática para validar la fórmula y argumentar por qué efectivamente proporciona la solución óptima.

4.3. Hipótesis de optimalidad (Principio de optimalidad)

El principio fundamental detrás de la programación dinámica es el *principio de optimalidad*, que establece que cualquier solución óptima de un problema contiene soluciones óptimas a los subproblemas.

En este contexto, dado un subarreglo de monedas $\{m_i, m_{i+1}, ..., m_j\}$, si Sophia juega de manera óptima (maximiza su ganancia), entonces la subsolución óptima a partir de i y j implica que después de su elección, las subsecuentes elecciones también deben ser óptimas. De lo contrario, Sophia no estaría maximizando su ganancia.

4.4. Definición de los subproblemas planteados

La función $\text{OPT}[i, j]$ representa la ganancia máxima que Sophia puede obtener al jugar óptimamente sobre el subarreglo $\{m_i, m_{i+1}, ..., m_j\}$. La clave de la recurrencia es que Sophia tiene dos elecciones posibles:

- Tomar m_i
- Tomar m_j

Después de tomar una moneda, Mateo jugará usando una estrategia *greedy*, seleccionando la moneda de mayor valor de los extremos del subarreglo restante. Esto restringe las subsecuentes opciones de Sophia, pero al estar considerando todos los subproblemas posibles, la elección que maximice su ganancia puede modelarse con la fórmula obtenida:

$$\text{OPT}[i,j] = \text{máx} \begin{cases} m_i + \text{OPT}[i+2][j] & \text{si } m_{i+1} \geq m_j, \\ m_i + \text{OPT}[i+1][j-1] & \text{si no;} \\ \\ m_j + \text{OPT}[i+1][j-1] & \text{si } m_i \geq m_{j-1}, \\ m_j + \text{OPT}[i][j-2] & \text{si no} \end{cases}$$

4.5. Inducción Matemática sobre el Tamaño del Subarreglo

Vamos a demostrar que la fórmula recursiva nos lleva al valor óptimo utilizando inducción sobre el tamaño del subarreglo $n = j - i + 1$.

4.5.1. Base de Inducción

- Si $n = 1$, es decir, solo hay una moneda, Sophia simplemente toma esa única moneda m_i, y su ganancia es m_i. Esto es claramente óptimo.

$$\text{OPT}[i, i] = m_i$$

- Si $n = 2$, hay dos monedas m_i y m_j. Sophia tomará la de mayor valor, ya que Mateo tomará la otra. Entonces:

$$\text{OPT}[i, i+1] = \text{máx}(m_i, m_{i+1})$$

  Esto es claramente óptimo.

4.5.2. Paso Inductivo

Supongamos que la fórmula es válida para cualquier subarreglo de longitud $k \leq n-1$, y probemos para $k = n$.

Para un subarreglo $\{m_i, ..., m_j\}$, Sophia tiene dos opciones iniciales:

- **Opción 1**: Si toma m_i, Mateo jugará de manera greedy y seleccionará m_{i+1} si $m_{i+1} \geq m_j$, lo que deja a Sophia con el subproblema $\text{OPT}[i+2, j]$. En caso contrario, Mateo tomará m_j, dejando el subproblema $\text{OPT}[i+1, j-1]$.
- **Opción 2**: Si toma m_j, Mateo jugará de manera greedy, tomando m_i si $m_i \geq m_{j-1}$, lo que deja a Sophia con el subproblema $\text{OPT}[i+1, j-1]$. Si no, Mateo toma m_{j-1}, y el subproblema restante será $\text{OPT}[i, j-2]$.

Por el principio de inducción, sabemos que $\text{OPT}[i+2, j]$, $\text{OPT}[i+1, j-1]$, y $\text{OPT}[i, j-2]$ son subproblemas resueltos óptimamente (porque su longitud es menor que n). Por lo tanto, Sophia elegirá la opción que le proporciona la mayor ganancia, dado que ambos subproblemas se resuelven de manera óptima. Esto está reflejado en la fórmula dada anteriormente.

Entonces podemos decir que la ecuación recursiva asegura que en cada paso, Sophia toma la decisión que maximiza su ganancia considerando las decisiones óptimas de los subproblemas. Este razonamiento, junto con la validez de la hipótesis inductiva, nos garantiza que el algoritmo propuesto siempre encontrará la ganancia óptima para Sophia.

5. Implementación del algoritmo óptimo

Presentamos la implementación del algoritmo propuesto utilizando el lenguaje de programación *Python*. El código ha sido estructurado de manera que siga una lógica clara y modular, facilitando su lectura.

La elección de *Python* se debe a su legibilidad y a las bibliotecas avanzadas que ofrece para la manipulación de estructuras de datos y la optimización de procesos. El enfoque utilizado para la implementación refleja fielmente el análisis presentado anteriormente, asegurando que se mantengan las propiedades de optimalidad y eficiencia.

5.1. Algoritmo funcional

```python
def jugar(coins):
    n = len(coins)
    dp = [[0] * n for _ in range(n)]

    for i in range(n):
        dp[i][i] = coins[i]

    for length in range(2, n + 1):
        for i in range(n - length + 1):
            j = i + length - 1
            if not i + 2 <= j:
                dp[i][j] = max(coins[i], coins[j])
                continue
            if coins[i+1] >= coins[j]:
                option1 = coins[i] + (dp[i+2][j])
            else:
                option1 = coins[i] + (dp[i+1][j-1])

            if coins[i] >= coins[j-1]:
                option2 = coins[j] + (dp[i+1][j-1])
            else:
                option2 = coins[j] + (dp[i][j-2])
            dp[i][j] = max(option1, option2)
    return dp
```

5.1.1. Grilla de seguimiento

A continuación, mostraremos cómo se desarrolla la ecuación de recurrencia utilizando una representación en forma de grilla. Esta grilla almacenará los resultados de cada subproblema correspondiente a las diferentes combinaciones de monedas, tal como lo explicamos previamente.

Presentamos la misma secuencia que utilizamos en las explicaciones anteriores:

Index	0	1	2	3
Coin	3	9	2	7

En la diagonal principal se encuentran los casos base, donde solo hay una moneda disponible. Según el enunciado del problema, Sophia siempre comienza el juego, por lo que en estos casos base Sophia tomará la única moneda disponible.

Para facilitar el análisis, utilizamos las siguientes notaciones:

- **monedas(i,j)**: Representa el subjuego correspondiente a un arreglo de monedas desde el índice i hasta el índice j.
- **Valores de las celdas**: Cada celda contiene una tupla que indica las ganancias máximas de cada jugador en ese subjuego. El primer valor corresponde a la ganancia de Sophia, mientras que el segundo valor corresponde al minímo entre los subproblemas óptimos, como se muestra en la ecuación de recurrencia. *monedas(i, j)*.

Dado que no es posible tener un subjuego donde el índice inicial sea mayor que el índice final (i.e., $i > j$), las celdas por debajo de la diagonal principal quedan vacías.

Index	0	1	2	3
0	monedas(0,0)			
1		monedas(1,1)		
2			monedas(2,2)	
3				monedas(3,3)

Cuando el subjuego se extiende a un conjunto de dos o más monedas, aplicamos la lógica definida por la ecuación de recurrencia. Por ejemplo, para el subjuego *monedas(0, 1)*, Sophia puede optar por tomar la moneda de valor 3 o la moneda de valor 9. Al ser un caso base, Sophia seleccionará el máximo de estos dos valores, mientras que Mateo se quedará con el juego restante.

Index	0	1	2	3
0	3	9		
1		9	9	
2			2	7
3				7

Consideremos el escenario **monedas(0, 2)**, con tres monedas cuyos valores son 3, 9 y 2. Sophia, al ser la primera en elegir, dispone de dos opciones:

$$\text{OPT}[0,2] = \text{máx} \begin{cases} m_0 + \text{OPT}[2][2], & \text{si } m_1 \geq m_2, \\ m_0 + \text{OPT}[1][1], & \text{si no,} \\ \\ m_2 + \text{OPT}[1][1], & \text{si } m_0 \geq m_1, \\ m_2 + \text{OPT}[0][0], & \text{si no} \end{cases}$$

Ahora, vamos a desarrollar los posibles resultados:

- Si Sophia elige la moneda del extremo izquierdo $m_0 = 3$, entonces Mateo evaluará entre $m_1 = 9$ y $m_2 = 2$. Como $m_1 > m_2$, Mateo tomará m_1, y quedará el subproblema OPT[2][2]. En este caso, Sophia tendrá que jugar con la única moneda restante $m_2 = 2$.

  Por lo tanto:
  $$\text{OPT}[0, 2] = m_0 + \text{OPT}[2][2] = 3 + 2 = 5$$

- Si Sophia elige la moneda del extremo derecho $m_2 = 2$, entonces Mateo evaluará entre $m_0 = 3$ y $m_1 = 9$. Como $m_1 > m_0$, Mateo tomará m_1, y el subproblema que queda es OPT[0][0], donde Sophia se quedará con la única moneda restante $m_0 = 3$.

  Por lo tanto:
  $$\text{OPT}[0, 2] = m_2 + \text{OPT}[0][0] = 2 + 3 = 5$$

Finalmente, observamos que, independientemente de la opción que elija Sophia, la ganancia máxima que puede obtener es 5.

Index	0	1	2	3
0	3	9	5	
1		9	9	
2			2	7
3				7

La misma lógica se aplica para el juego de **monedas(1,3)**, donde la secuencia de monedas es [9,2,7].

Index	0	1	2	3
0	3	9	5	
1		9	9	16
2			2	7
3				7

Finalmente, evaluamos el escenario **monedas(0, 3)**, donde Sophia se enfrenta a la secuencia completa de monedas [3, 9, 2, 7]. Al igual que en el caso anterior, Sophia tiene dos opciones:

$$\text{OPT}[0, 3] = \text{máx} \begin{cases} m_0 + \text{OPT}[2][3], & \text{si } m_1 \geq m_3, \\ m_0 + \text{OPT}[1][2], & \text{si no,} \\ \\ m_3 + \text{OPT}[1][2], & \text{si } m_0 \geq m_2, \\ m_3 + \text{OPT}[0][1], & \text{si no} \end{cases}$$

Vamos a desarrollar los posibles casos:

- **Si Sophia elige $m_0 = 3$:**

  Mateo elige la moneda de mayor valor de los extremos entre $m_1 = 9$ y $m_3 = 7$. Como $m_1 > m_3$, Mateo toma m_1, y el subproblema restante es OPT[2][3], con las monedas $m_2 = 2$ y $m_3 = 7$.

  Ahora, en la tabla vemos que el OPT[2][3]:
  $$\text{OPT}[2, 3] = m_3 = 7$$

  Ya que no quedan más monedas después de esta jugada, Sophia obtendrá 7 al tomar m_3. Por lo tanto:
  $$\text{OPT}[0, 3] = m_0 + \text{OPT}[2][3] = 3 + 7 = 10$$

- **Si Sophia elige $m_3 = 7$:**

  Mateo elegirá entre $m_0 = 3$ y $m_1 = 9$. Como $m_1 > m_0$, Mateo tomará m_1, y el subproblema que queda es OPT[0][1], con las monedas $m_0 = 3$ y $m_1 = 9$.

  Ahora, en la tabla vemos que el OPT[0][1]:

  $$\text{OPT}[0,1] = m_1 = 9$$

  Por lo tanto:

  $$\text{OPT}[0,3] = m_3 + \text{OPT}[0][1] = 7 + 9 = 16$$

Finalmente, observamos que el máximo que Sophia puede obtener en el juego completo es 16 si toma $m_3 = 7$ en su primer turno. Esta es la solución óptima para el problema y **siempre se verá reflejado en la esquina superior derecha**.

Index	0	1	2	3
0	3	9	5	**16**
1		9	9	16
2			2	7
3				7

5.2. Reconstruccion de resultados

```python
def reconstruccion(dp_table, coins):
    i, j = 0, len(coins) - 1
    monedas_sophia = []
    turno = 0

    while i <= j:
            # Si esta en las primeras 2 diagonales (casos base) Sophia elige la moneda mas grande
            if not i + 2 <= j:
                option1 = coins[i]
            # Sino Sophia elige segun la ecuacion de recurrencia
            else:
                if coins[i+1] >= coins[j]:
                    option1 = coins[i] + (dp[i+2][j])
                else:
                    option1 = coins[i] + (dp[i+1][j-1])
            # Verificar si la matriz de optimos coincede con la opcion 1. En caso contrario coincide con la opcion 2
            if dp[i][j] == option1:
                monedas_sophia.append(coins[i])
                turno += 1
                if coins[i+1] >= coins[j]:
                    # Mateo toma la moneda en coins[i+1]
                    i += 2
                else:
                    # Mateo toma la moneda en coins[j]
                    i += 1
                    j -= 1
            else:
                monedas_sophia.append(coins[j])
                if coins[i] >= coins[j-1]:
                    # Mateo toma la moneda en coins[i]
                    i += 1
                    j -= 1
                else:
                    # Mateo toma la moneda en coins[j-1]
                    j -= 2
    return monedas_sophia
```

Este código devuelve un arreglo con las monedas que tomó Sophia. Para reconstruir las decisiones de Sophia usando la matriz `dp` y el arreglo de monedas C, debemos entender cómo funciona

la estrategia de Sophia en cada turno. Al usar la matriz `dp`, Sophia busca siempre maximizar su ganancia futura considerando la mejor elección que puede hacer en cada subarreglo $[i, j]$.

5.2.1. Reconstrucción de las monedas tomadas por Sophia

La función `reconstruir_monedas_tomadas_por_sophia` toma el arreglo de monedas `coins` y la matriz de resultados óptimos `dp`, y usa la estrategia de la ecuación de recurrencia para seleccionar las monedas que maximicen la ganancia de Sophia. A continuación, se detalla cada paso de la reconstrucción:

1. **Inicialización**:
   - Se comienza con $i = 0$ y $j = \text{len(coins)} - 1$, apuntando a los extremos del arreglo.
   - `monedas_sophia` es una lista vacía donde guardaremos las monedas que Sophia elige.
2. **Iteración sobre el rango de monedas**:
   - Mientras $i \leq j$, revisamos qué moneda le conviene a Sophia, usando las decisiones preestablecidas en la matriz `dp` y en la ecuación de recurrencia.
3. **Selección de la moneda para Sophia**:
   - Si el rango i a j es pequeño (menos de 3 posiciones, es decir, los casos base en las primeras diagonales de `dp`), Sophia elige la moneda de mayor valor.
   - Si hay más de dos posiciones, la elección de Sophia depende de cuál de las dos opciones maximiza su ganancia:
     - *Opción 1*: Sophia elige la moneda en coins$[i]$ y Mateo juega de acuerdo a la estrategia *greedy*.
     - *Opción 2*: Sophia elige la moneda en coins$[j]$ y Mateo también juega *greedy*.
4. **Verificación en la matriz `dp`**:
   - Comparamos `dp[i][j]` con `option1` para verificar si la *opción 1* corresponde a la solución óptima.
   - Si `dp[i][j]` es igual a `option1`, entonces Sophia elige coins$[i]$, y actualizamos los índices i y j según la elección de Mateo.
   - Si `dp[i][j]` no es igual a `option1`, significa que la opción óptima para Sophia es elegir coins$[j]$, y nuevamente actualizamos los índices.
5. **Repetición del proceso**:
   - Repetimos los pasos hasta que todos los índices hayan sido procesados y $i > j$, terminando el juego con el registro de las monedas que Sophia tomó.

5.2.2. Ejemplo con `coins = (3, 9, 2, 7)`

A continuación, se muestra un ejemplo de reconstrucción con el arreglo de monedas `coins = (3, 9, 2, 7)`. Para ilustrar cómo Sophia selecciona las monedas utilizando la matriz de resultados óptimos `dp`, consideremos el arreglo de monedas `coins = (3, 9, 2, 7)`.

- **Primera Iteración:** Inicialmente, $i = 0$ y $j = 3$, con las monedas $(3, 9, 2, 7)$.
  - *Opción 1*: Sophia elige $m_0 = 3$.
    - Mateo sigue la estrategia *greedy* y elige $m_1 = 9$, dejando el subarreglo $(2, 7)$ para Sophia.
    - `option1` $= 3 +$ `dp[2][3]`.
  - *Opción 2*: Sophia elige $m_3 = 7$.

- Mateo sigue la estrategia *greedy* y elige $m_0 = 3$, dejando el subarreglo $(9, 2)$.
- `option2` $= 7 +$ `dp[0][1]`.

- Comparando `dp[0][3]`, Sophia elige la opción con mayor valor. Si `dp[0][3]` es igual a `option2`, Sophia elige 7, que se añade a `monedas_sophia`.
- Luego, se actualizan $i = 0$ y $j = 1$ tras la elección de Mateo.

- **Segunda Iteración:** Con $i = 1$ y $j = 2$, las monedas restantes son $(9, 2)$.
  - *Opción 1*: Sophia elige $m_1 = 9$, quedando `option1` $= 9$ pues Mateo tomaría $m_2 = 2$.
  - *Opción 2*: Sophia elige $m_2 = 2$, y Mateo tomaría $m_1 = 9$, dejando `option2` $= 2$.
  - Dado que `dp[1][2]` es igual a `option1`, Sophia elige 9, que se añade a `monedas_sophia`.
  - El juego termina ya que $i > j$.

Al final, el arreglo `monedas_sophia` contiene las monedas $(7, 9)$, que son las que Sophia eligió siguiendo la estrategia óptima.

5.3. Complejidad

La complejidad teórica de nuestra solución es cuadratica. Vamos a analizarlo más a detalle sección a sección:

```
2     n = len(coins)
3     dp = [[0] * n for _ in range(n)]
4
5     for i in range(n):
6         dp[i][i] = coins[i]
```

Primero construimos la matriz y rellenamos la diagonal con los valores de las monedas. Como se observa en la linea 3, la minima complejidad que tenemos desde el principio es $\mathcal{O}(n^2)$, al construir una matriz de n x n. Luego rellenar la diagonal nos cuesta un $\mathcal{O}(n)$, que queda despreciable por la operacion anterior.

```
8     for length in range(2, n + 1):
9         for i in range(n - length + 1):
10            j = i + length - 1
11            if not i + 2 <= j:
12                dp[i][j] = max(coins[i], coins[j])
13                continue
14            if coins[i+1] >= coins[j]:
15                option1 = coins[i] + (dp[i+2][j])
16            else:
17                option1 = coins[i] + (dp[i+1][j-1])
18
19            if coins[i] >= coins[j-1]:
20                option2 = coins[j] + (dp[i+1][j-1])
21            else:
22                option2 = coins[j] + (dp[i][j-2])
23            dp[i][j] = max(option1, option2)
24    return dp
```

Luego recorremos la matriz de forma diagonal, lo que por si solo cuesta $\mathcal{O}(n^2)/2$

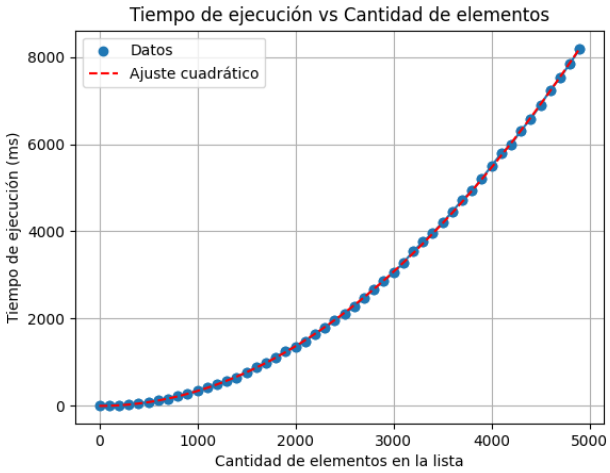
Finalmente el resto de las operaciones dentro del for son constantes, simplmente comparamos valores y accedemos al arreglo de monedas y la matriz mediante indices, resultando en una complejidad final de $\mathcal{O}(n^2)$

6. Mediciones

Se llevaron a cabo mediciones sistemáticas utilizando arreglos de distintas dimensiones, donde los elementos de cada arreglo fueron proporcionados por la cátedra. De esta forma, obtuvimos una variedad de datos representativos, facilitando un análisis del desempeño del algoritmo bajo estudio.

6.1. Tiempo de ejecución vs. Cantidad de elementos

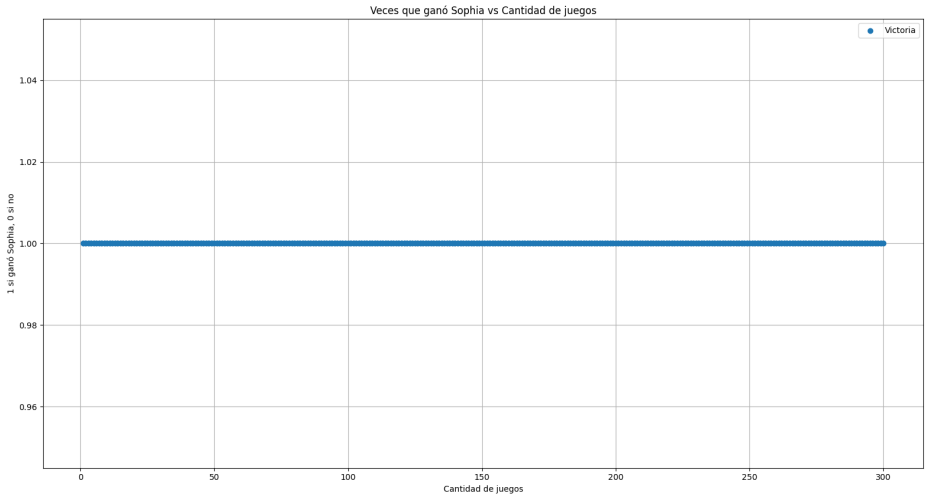
En el siguiente grafico, mostramos como distintas cantidades de monedas afectan los tiempos de ejecucion.



El gráfico confirma la complejidad cuadrática teórica del algoritmo principal $\mathcal{O}(n^2)$, que implica llenar y operar sobre la matriz `dp` de tamaño $n \times n$. El ajuste cuadrático (línea roja) sigue de cerca los datos empíricos, lo que valida que el número de operaciones escala con el cuadrado de la cantidad de elementos en la lista de monedas.

6.2. Cantidad de victorias vs. Cantidad de juegos

En esta seccion muestra una funcion que grafica 1 si gano Sophia, 0 si gano Mateo. Lo realizamos con mil simulaciones del juego con valores generados aleatoriamente.



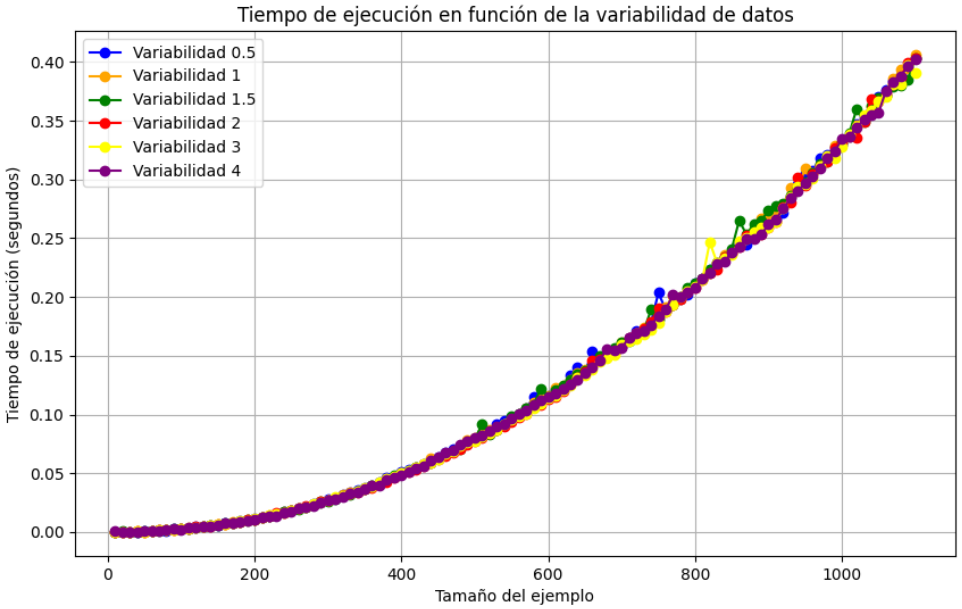
Observandose que si bien puede que no gane todas las partidas, gana la mayoria. Lo que muestra la mejora entre jugar con programacion dinamica *vs.* jugar con greedy.

6.3. Impacto en la variabilidad

La variabilidad en los gráficos se refiere a la dispersión de los valores de las monedas con respecto la cantidad de las mismas en un juego dado.

Cuando la variabilidad es alta, se generan datos más distantes de los valores promedio, lo que implica que algunos puntos pueden estar más alejados, creando una mayor dispersión. Por otro lado, cuando la variabilidad es baja, los datos están más concentrados en torno a un valor central, lo que indica una distribución más homogénea y menos dispersa.

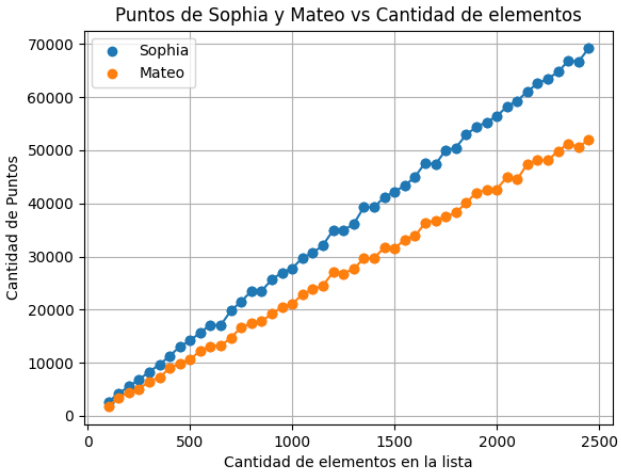
Decidimos analizar estos niveles de variabilidad, para evaluar el comportamiento del algoritmo bajo distintos escenarios de dispersión.



Observamos que aunque la variabilidad de los valores de las monedas afecta la distribución de puntos entre Sophia y Mateo, no altera el tiempo de ejecución ni la complejidad del algoritmo sustancialmente. Todos los valores de variabilidad siguen un patrón similar en términos de tiempo de ejecución, ya que el número de operaciones realizadas permanece constante para un tamaño de lista determinado. Esto confirma que la complejidad del algoritmo depende del tamaño de la lista de monedas, no de la variabilidad de sus valores.

6.4. Puntos vs. Cantidad de elementos

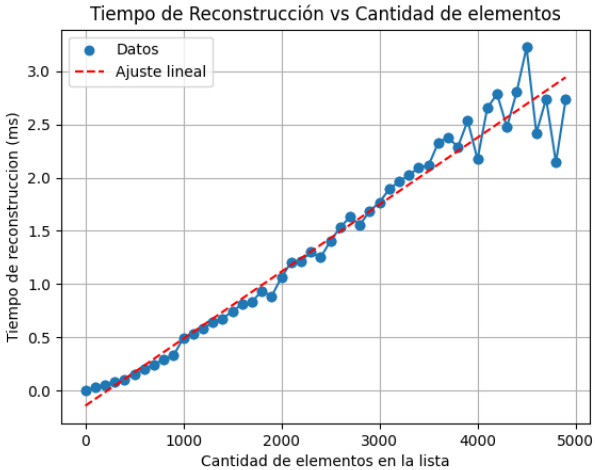
Al igual que en el trabajo anterior, graficamos la cantidad de puntos ganados por cada jugador *versus* la cantidad de monedas en cada partida. En este caso se observa que Sophia acumula la maxima cantidad posible por partida, que, como analizamos anteriormente, no necesariamente la hace ganar siempre.



Aunque el algoritmo asegura que Sophia maximice su ganancia, la cantidad de puntos que obtienen Sophia y Mateo también depende de los valores específicos de las monedas. La variabilidad de los puntos entre Sophia y Mateo tiende a crecer con el tamaño de la lista, pero el patrón general de que Sophia obtenga más puntos se mantiene, ya que ella elige de manera óptima mientras Mateo sigue una estrategia *greedy*.

6.5. Tiempo de Reconstruccion vs. Cantidad de Elementos

A continuacion graficamos el tiempo que toma reconstruir las monedas que eligio Sophia con distintas cantidades de monedas



El gráfico muestra un ajuste que parece aproximadamente lineal (representado por la línea de ajuste en rojo). El tiempo de reconstrucción aumenta de manera proporcional a la cantidad de elementos en la lista, lo cual indica que la fase de reconstrucción del resultado tiene una complejidad de $O(n)$. Esto es consistente con el algoritmo utilizado, donde para reconstruir las decisiones de Sophia, accedemos n veces a ciertas posiciones de la matriz **dp** para verificar las decisiones óptimas. Además, recorremos el arreglo de monedas una única vez desde el principio hasta el final, seleccionando las monedas tomadas por Sophia y Mateo. Este enfoque permite determinar las decisiones de manera eficiente, en tiempo lineal respecto a la cantidad de monedas.

Las ligeras oscilaciones en la parte alta del gráfico podrían deberse a variaciones en el sistema

durante la ejecución o a pequeñas ineficiencias del código, pero no son significativas como para cambiar la tendencia general.

Es así como podemos decir que el proceso de reconstrucción es eficiente y tiene una complejidad lineal respecto a la cantidad de monedas. Esto implica que, una vez que tenemos la solución calculada, el tiempo para reconstruir las decisiones es proporcional al número de monedas, lo cual es óptimo.

Aunque el gráfico solo analiza la reconstrucción, es importante recordar que el tiempo total de resolución del juego, incluyendo el llenado de la tabla de programación dinámica, sigue siendo $O(n^2)$. Sin embargo, este gráfico confirma que la parte de la reconstrucción de las elecciones de Sophia tiene una complejidad menor, específicamente $O(n)$.

7. Conclusiones

A lo largo de este trabajo, nos enfrentamos al desafío de resolver el juego de las monedas entre Sophia y Mateo utilizando programación dinámica, lo cual nos permitió desarrollar una solución óptima que, aunque de complejidad cuadrática, garantiza la victoria de Sophia siempre que sea posible. En comparación con un enfoque **greedy**, que únicamente toma decisiones locales sin considerar futuras consecuencias, nuestra solución dinámica evalúa cada posible secuencia de jugadas, asegurando que Sophia tome decisiones estratégicas en función de todas las posibles respuestas de Mateo.

El desarrollo de este algoritmo no estuvo exento de complejidades. Pasamos varios días analizando matrices y gráficos detallados, buscando patrones y estudiando múltiples variantes hasta encontrar una ecuación de recurrencia que capturara con precisión el comportamiento del problema.

Inicialmente, llegamos a una ecuación de recurrencia que parecía correcta:

$$\text{OPT}(i,j) = \text{máx}\begin{pmatrix} \text{monedas}[i] + \text{mín}\left(\text{OPT}(i+1,j-1), \text{OPT}(i+2,j)\right), \\ \text{monedas}[j] + \text{mín}\left(\text{OPT}(i+1,j-1), \text{OPT}(i,j-2)\right) \end{pmatrix}$$

Esta fórmula, si bien era óptima en todos los casos analizados, solo funcionaba bajo el supuesto de que Mateo también empleara el mismo algoritmo para tomar sus decisiones. Fue gracias a la observación de nuestro profesor, Martín Buchwald, que nos dimos cuenta de este detalle no menor. No estabamos teniendo en cuenta la variable del algortimo empleado por Mateo. Mas bien, estabamos modelando algo que busca minimizar cuanto puede conseguir. Su consejo fue sumamente importante para corregir nuestra aproximación y ajustar el algoritmo.

Posteriormente, en cada etapa del desarrollo de la implementación, buscamos optimizar no solo la complejidad del algoritmo, sino también su implementación práctica, asegurándonos de que el código fuera eficiente y capaz de manejar grandes volúmenes de datos.

Podemos decir que hacer este trabajo fue desafiante en muchos aspectos. No solo nos ayudó a aplicar y entender mejor los conceptos de programación dinámica, sino también poner a prueba nuestra capacidad para resolverlos de manera organizada y en equipo. Lo más increíble fue que, una vez que dimos con la ecuación de recurrencia correcta, logramos avanzar rápidamente y en poco tiempo pudimos terminar el desarrollo, lo que hizo que el esfuerzo inicial valiera la pena.