.UBAfiuba
FACULTAD DE INGENIERÍA

Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

Trabajo Práctico 2
Programación Dinámica For The Win

15 de octubre de 2024

Victoria Avalos	108434
Santiago Tissoni	103856
Facundo Monpelat	92716

1. Introducción

Los algoritmos de programación dinámica se caracterizan por descomponer problemas en subproblemas más pequeños y solapados, almacenando las soluciones de estos para evitar cálculos redundantes. Este enfoque asegura la obtención de soluciones óptimas al garantizar que se consideran, de forma implícita, todas las posibles combinaciones de decisiones previas. La programación dinámica es particularmente efectiva en problemas que exhiben subestructuras óptimas y solapamiento de subproblemas. Algunos ejemplos conocidos son el algoritmo de Floyd-Warshall para caminos más cortos, el problema de la mochila, y la sucesión de Fibonacci.

El objetivo de este trabajo práctico es aplicar y analizar el uso de un algoritmo de programación dinámica en un escenario específico: un juego de toma de decisiones en el que se posee una fila de monedas de diferentes valores, donde se debe seleccionar una de alguno de los extremos con el objetivo de acumular la mayor puntuación.

A lo largo del trabajo, se evaluará si es posible garantizar una solución óptima utilizando un enfoque de programación dinámica, así como los factores que podrían afectar la optimalidad y la eficiencia del algoritmo propuesto.

1.1. Descripción del Problema

Se dispone de una fila de n monedas, cada una con un valor distinto. En cada turno, el jugador puede elegir solo entre la primera o la última moneda de la fila, removiendo la elegida y cediendo el turno al otro jugador. Los jugadores son dos hermanos, Sophia y Mateo. Sophia aplicará sus conocimientos en programación dinámica para asegurarse de ganar siempre que pueda, mientras que Mateo aplicará una estrategia greedy para maximizar su ganancia inmediata.

Respecto de los valores de las monedas, se considerará que son siempre positivos y no se admite una cantidad par de monedas con el mismo valor.

A lo largo de este trabajo, se propone un algoritmo que permite a Sophia seleccionar las monedas de forma tal que asegure la victoria siempre que pueda, teniendo en cuenta que Mateo siempre eligirá la moneda máxima de ambos extremos.

1.2. Objetivos

Los objetivos del presente trabajo práctico son los siguientes:

- Definir la ecuación de recurrencia y proponer un algoritmo de programación dinámica que determine las monedas que Sophia debe elegir para asegurarse el mayor valor acumulado posible teniendo en cuenta la estrategia greedy de Mateo. Aunque Sophia no siempre pueda ganar, se busca optimizar su resultado.
- Probar la optimalidad de la ecuación de recurrencia propuesta: Probar que la ecuación planteada lleva efectivamente al máximo valor acumulado posible para Sophia.
- Implementar el algoritmo y justificar su complejidad. Analizar el impacto de la variabilidad de los valores de las monedas en su rendimiento.
- Realizar ejemplos de ejecución y validar la optimalidad del algoritmo.
- Medir el rendimiento: Verificar empíricamente la complejidad del algoritmo mediante gráficos y mínimos cuadrados, utilizando conjuntos de datos generados.

Resolución

2. Algoritmo que garantice la ganancia máxima posible para Sophia

En este trabajo realizaremos el análisis teórico y empírico de un algoritmo de programación dinámica que resuelve el problema de determinar las monedas que escogerá Sophia. Nuestro programa permite ejecutar dicho algoritmo con los casos de prueba que se le pasen en formato txt con el comando

```
python3 problema.py <monedas separadas por ;>
python3 problema.py "10;40"
```

También posee pruebas automatizadas que se ejecutan con

```
python3 tests.py
```

y soporta la generación automática y ejecuccion de nuevos casos de prueba con los comandos

```
python3 generador.py <tam_maximo_arreglo> <valor_maximo_elementos>
python3 tests.py --tests-autogenerated
```

Para más información, leer el repositorio con el código

2.1. Ecuación de recurrencia

Para resolver este problema mediante programación dinámica, comenzamos planteando la ecuación de recurrencia correspondiente. Para ello, primero definimos dos aspectos clave:

La forma de los subproblemas: Cada subproblema representa el valor máximo acumulado que Sophia puede obtener en un intervalo reducido de la fila de monedas. En cada turno, Sophia puede elegir tomar una moneda del extremo izquierdo o derecho. Dependiendo de su elección, el subproblema se reduce al intervalo que queda disponible después del turno de Mateo.

La composición de los subproblemas para resolver problemas más grandes: Si Sophia elige una moneda de un extremo en particular, la solución óptima debe considerar la mejor opción que tomará Mateo, quien juega de manera *greedy*. Si, en cambio, Sophia toma la moneda del otro extremo, nuevamente debemos tener en cuenta las acciones de Mateo para componer la solución.

Es importante destacar que las acciones de Mateo son predecibles, ya que siempre elegirá la moneda de mayor valor entre las disponibles en los extremos.

Con este análisis, la ecuación de recurrencia que utilizamos es la siguiente:

$$\mathrm{OPT}(i,j) = \text{máx} \left(\begin{array}{l} \mathrm{coins}[i] + \mathrm{OPT}(i+2,j) \text{ if } \mathrm{coins}[i+1] \geq \mathrm{coins}[j] \text{ else } 0, \\ \mathrm{coins}[i] + \mathrm{OPT}(i+1,j-1) \text{ if } \mathrm{coins}[i+1] < \mathrm{coins}[j] \text{ else } 0, \\ \mathrm{coins}[j] + \mathrm{OPT}(i+1,j-1) \text{ if } \mathrm{coins}[i] \geq \mathrm{coins}[j-1] \text{ else } 0, \\ \mathrm{coins}[j] + \mathrm{OPT}(i,j-2) \text{ if } \mathrm{coins}[i] < \mathrm{coins}[j-1] \text{ else } 0 \end{array} \right)$$

Donde:

- i: índice correspondiente al extremo izquierdo de la fila de monedas,
- j: índice correspondiente al extremo derecho de la fila de monedas,
- coins: fila de monedas.

Dicha ecuación contempla todas las posibles situaciones derivadas de las decisiones de Sophia, evaluando cada configuración de las monedas y anticipando las acciones de Mateo en cada escenario.

- Sophia escoge la moneda de la izquierda ($coins[i]$). Si la moneda de la izuierda de la fila resultante ($coins[i+1]$) es mayor a la del extremo opuesto ($coins[j]$), sabemos que Mateo la escogerá. Sumamos la moneda de Sophia al óptimo acumulado para la fila de monedas con los índices actualizados ($OPT(i+2, j)$), que corresponde al siguiente turno de Sophia.
- Sophia escoge la moneda de la izquierda ($coins[i]$). Si la moneda de la derecha de la fila resultante ($coins[j]$) es mayor a la del extremo opuesto ($coins[i+1]$), sabemos que Mateo la escogerá. Sumamos la moneda de Sophia al óptimo acumulado para la fila de monedas con los índices actualizados ($OPT(i+1, j-1)$), que corresponde al siguiente turno de Sophia.
- Sophia escoge la moneda de la derecha ($coins[j]$). Si la moneda de la izuierda de la fila resultante ($coins[i]$) es mayor a la del extremo opuesto ($coins[j-1]$), sabemos que Mateo la escogerá. Sumamos la moneda de Sophia al óptimo acumulado para la fila de monedas con los índices actualizados ($OPT(i+1, j-1)$), que corresponde al siguiente turno de Sophia.
- Sophia escoge la moneda de la derecha ($coins[j]$). Si la moneda de la derecha de la fila resultante ($coins[j-1]$) es mayor a la del extremo opuesto ($coins[i]$), sabemos que Mateo la escogerá. Sumamos la moneda de Sophia al óptimo acumulado para la fila de monedas con los índices actualizados ($OPT(i, j-2)$), que corresponde al siguiente turno de Sophia.

Entonces, la ecuación de recurrencia se queda con la situación que maximice la ganancia de Sophia.

2.2. Algoritmo de Programación Dinámica

El algoritmo que utilizamos para implementar la ecuación de recurrencia emplea memorización para evitar recalcular subproblemas más pequeños, utilizando un enfoque "bottom-up". Esto significa que resuelve primero los subproblemas más simples, comenzando desde los índices más pequeños y avanzando de manera incremental hasta alcanzar los índices superiores, construyendo las soluciones más grandes a partir de las más pequeñas.

El algoritmo comienza creando una matriz dp de tamaño $n \times n$, donde n es la cantidad de monedas, y utiliza dos bucles anidados para llenarla. En estos bucles, los índices i y j representan los límites de un subarreglo de monedas: i es el inicio del subarreglo y j es el final. El objetivo es calcular el valor máximo que Sophia puede obtener considerando solo las monedas desde la posición i hasta la posición j.

El primer bucle `for` controla el tamaño del subarreglo, usando la variable `length`, que varía desde 1 (subarreglos de una moneda) hasta n (el subarreglo completo). Para cada valor de `length`, el segundo bucle recorre todas las posiciones de inicio posibles para los subarreglos de esa longitud, variando i desde 0 hasta $n-$`length`. A partir de i, el índice final j se calcula como $j = i+$`length`-1, cubriendo así todos los subarreglos de tamaño `length`.

En cada iteración, se evalúan tres casos:

- Si el subarreglo tiene solo una moneda ($i == j$), el valor en $dp[i][j]$ es simplemente el valor de esa moneda.
- Si el subarreglo tiene dos monedas ($i+1 == j$), se almacena en $dp[i][j]$ el valor de la moneda de mayor valor entre las dos.
- Para subarreglos con más de dos monedas ($i < j$), se evalúan varias posibles decisiones que Sophia puede tomar, es decir, si toma la moneda en i o en j, y se consideran las jugadas óptimas de Mateo en respuesta. Para cada opción, se utilizan los subproblemas previamente resueltos en la matriz dp (como $dp[i+2][j]$ o $dp[i][j-2]$), y se almacena el máximo valor posible en $dp[i][j]$.

De esta manera, el algoritmo recorre la matriz dp de forma incremental, comenzando por los subarreglos más pequeños y utilizando esas soluciones para calcular los subarreglos más grandes, hasta completar el subarreglo que abarca todas las monedas.

```python
def valor_max_sophia(coins):
    n = len(coins)

    dp = [[0] * n for _ in range(n)]

    for length in range(1, n + 1):
        for i in range(n - length + 1):
            j = i + length - 1

            # Caso base: una moneda
            if i == j:
                dp[i][j] = coins[i]
            # Caso base: dos monedas
            elif i + 1 == j:
                dp[i][j] = max(coins[i], coins[j])

            else:
                a = (
                    coins[i] + dp[i + 2][j]
                    if coins[i + 1] >= coins[j]
                    else 0
                )
                b = (
                    coins[i] + dp[i + 1][j - 1]
                    if coins[i + 1] < coins[j]
                    else 0
                )
                c = (
                    coins[j] + dp[i + 1][j - 1]
                    if coins[i] >= coins[j - 1]
                    else 0
                )
                d = (
                    coins[j] + dp[i][j - 2]
                    if coins[i] < coins[j - 1]
                    else 0
                )
                dp[i][j] = max(a, b, c, d)

    return dp
```

2.3. Reconstrucción de la solución

El algoritmo de reconstrucción de la solución, es decir recopilar las decisiones que debe tomar Sophia para llegar al óptimo, toma como entrada la lista de monedas ‘coins‘ y la matriz de soluciones F previamente calculada. Su objetivo es reconstruir las decisiones que toma Sophia durante el juego para maximizar su ganancia.

Se inicializan dos índices, $i = 0$ y $j = n - 1$, que representan los límites del subarreglo de monedas aún disponibles, es decir, la primera moneda en la posición i y la última moneda en la posición j. A medida que avanza el bucle, Sophia y Mateo alternan turnos para seleccionar monedas, y se registran las decisiones de ambos jugadores en una lista llamada ‘decisiones‘.

El bucle continúa mientras $i \leq j$. En cada iteración, primero juega Sophia. Si el valor almacenado en $F[i][j]$ corresponde a haber tomado la última moneda ($coins[j]$), se añade esa decisión a la lista ‘decisiones‘ y se reduce j en 1. Si en cambio el valor en $F[i][j]$ corresponde a haber tomado la primera moneda ($coins[i]$), se añade esa decisión y se incrementa i en 1.

Después del turno de Sophia, juega Mateo, siempre que queden monedas ($i \leq j$). Mateo selecciona la moneda de mayor valor entre las que quedan, comparando entre la primera ($coins[i]$) y la última ($coins[j]$). Dependiendo de su elección, se actualizan los índices i o j y se añade la decisión de Mateo a la lista ‘decisiones‘.

Este proceso se repite hasta que todas las monedas han sido seleccionadas por Sophia y Mateo, momento en el cual el algoritmo devuelve la lista ‘decisiones‘ que contiene la secuencia completa de jugadas realizadas por ambos jugadores.

```python
def reconstruccion(coins, F):
    decisiones = []
    i, j = 0, len(coins) - 1

    while i <= j:
        if F[i][j] == coins[j] + F[i][j - 2] or F[i][j] == coins[j] + F[i + 1][j - 1]:
            decisiones.append(f"Sophia debe agarrar la ultima {coins[j]}")
            j -= 1
        else:
            decisiones.append(f"Sophia debe agarrar la primera {coins[i]}")
            i += 1

        # Despues del juego de Sophia, juega Mateo
        if i <= j:
            if coins[i] >= coins[j]:
                decisiones.append(f"Mateo agarra la primera {coins[i]}")
                i += 1
            else:
                decisiones.append(f"Mateo agarra la ultima {coins[j]}")
                j -= 1

    return decisiones
```

2.4. Análisis de optimalidad

En esta sección nos dedicaremos a demostrar que la ecuación de recurrencia planteada obtiene siempre el máximo valor posible acumulado. Ya que Mateo juega greedy, no podemos asegurar que Sophia siempre gane, ya que de hecho hay situaciones donde no lo hará. Por eso, demostraremos que acumulará lo máximo posible, no que siempre ganará.

Definiciones:

- $coins[q]$): El valor de la moneda q.
- i: índice de la moneda del extremo izquierdo
- j: índice de la moneda del extremo derecho

Estrategia a seguir por cada jugador en un determinado turno:

$$Sophia = OPT(i, j)$$
$$Mateo = max(V(i), V(j))$$

Caso base

El caso base ocurre cuando solo queda una moneda, es decir, cuando $i = j$. En este caso, Sophia debe tomar esa única moneda, ya que no hay más opciones. Entonces, tenemos:

$$Sophia = OPT(i, i) = coins[i]$$

Esto es trivialmente correcto, ya que no hay otras monedas que elegir ni decisiones que tomar. El valor acumulado es simplemente el valor de la única moneda disponible.

Hipótesis inductiva

Supongamos que la ecuación de recurrencia es correcta para cualquier subfila con hasta k monedas. Es decir, para cualquier subfila (i, j) tal que $j - i + 1 \leq k$, la ecuación de recurrencia nos da el valor máximo posible que Sophia puede acumular al elegir de forma óptima.

Paso inductivo

Ahora, demostraremos que la ecuación es correcta para una subfila con $k+1$ monedas, es decir, $j - i + 1 = k + 1$.

Sophia tiene dos opciones principales:

- **Elegir la moneda en el extremo izquierdo**, es decir, $coins[i]$.
- **Elegir la moneda en el extremo derecho**, es decir, $coins[j]$.

Después de que Sophia elija una moneda, Mateo tomará una de las dos restantes de manera greedy (eligiendo siempre el mayor extremo). Analizaremos cada opción de Sophia en función de la decisión que tomaría Mateo.

Caso 1: Sophia elige $coins[i]$

Si Sophia elige $coins[i]$, Mateo se queda con las monedas en el rango $(i+1, j)$. Como Mateo juega de forma greedy, elegirá la moneda más grande entre $coins[i+1]$ y $coins[j]$. Tenemos dos posibilidades:

- **Mateo elige** $coins[i+1]$ **(si** $coins[i+1] \geq coins[j]$**)**. Esto deja a Sophia con la subfila $(i+2, j)$. El valor que Sophia puede acumular en esta situación es:

$$Sophia = coins[i] + OPT(i+2, j)$$

- **Mateo elige** $coins[j]$ **(si** $coins[i+1] < coins[j]$**)**. Esto deja a Sophia con la subfila $(i+1, j-1)$. El valor que Sophia puede acumular es:

$$Sophia = coins[i] + OPT(i+1, j-1)$$

Caso 2: Sophia elige $coins[j]$

Si Sophia elige $coins[j]$, Mateo se queda con las monedas en el rango $(i, j-1)$. Mateo, de nuevo, tomará la moneda más grande entre $coins[i]$ y $coins[j-1]$. Tenemos dos posibilidades:

- **Mateo elige** $coins[i]$ **(si** $coins[i] \geq coins[j-1]$**)**. Esto deja a Sophia con la subfila $(i+1, j-1)$. El valor que Sophia puede acumular es:

$$Sophia = coins[j] + OPT(i+1, j-1)$$

- **Mateo elige** $coins[j-1]$ **(si** $coins[i] < coins[j-1]$**)**. Esto deja a Sophia con la subfila $(i, j-2)$. El valor que Sophia puede acumular es:

$$Sophia = coins[j] + OPT(i, j-2)$$

Conclusión del paso inductivo

De este modo, Sophia maximiza su valor acumulado considerando todas las posibles elecciones que Mateo hará de forma greedy después de su decisión inicial. Esto queda reflejado en la ecuación de recurrencia:

$$OPT(i,j) = \text{máx} \begin{pmatrix} coins[i] + OPT(i+2, j) & \text{si } coins[i+1] \geq coins[j], \\ coins[i] + OPT(i+1, j-1) & \text{si } coins[i+1] < coins[j], \\ coins[j] + OPT(i+1, j-1) & \text{si } coins[i] \geq coins[j-1], \\ coins[j] + OPT(i, j-2) & \text{si } coins[i] < coins[j-1] \end{pmatrix}$$

La hipótesis inductiva nos permite concluir que la ecuación de recurrencia maximiza el valor acumulado para Sophia para cualquier subfila de longitud $k+1$. Por lo tanto, por inducción, la ecuación es válida para cualquier cantidad de monedas en la fila.

Conclusión final

La ecuación de recurrencia es correcta y garantiza que Sophia, jugando de forma óptima usando programación dinámica, maximiza su valor acumulado frente a un oponente (Mateo) que juega de manera greedy.

2.4.1. Variabilidad del valor de las monedas para los tiempos de ejecucion

Los diferentes niveles de variabilidad no afectan los tiempos de ejecución de algoritmos. Para un array de tamaño n de variabidad baja, el tiempo de ejecución del algoritmo es similiar a un array de tamaño m con $m = n$ de variabilidad alta, debido a que cada iteración dentro del algoritmo, los valores de las monedas son únicamente comparados con complejidad $O(1)$ constante, y comparar números de alta variabilidad tiene la misma complejidad y tiempo de ejecución que comparar numeros "más cercanos"

Veamos una ejecución para comprobarlo

```
[4]  ✓ 0.0s
```

```python
from utils.utils import time_algorithm
import numpy as np
from problema import valor_max_sophia
```

```python
# Exectute the algorithm with random arrays and measure the time it takes to run

def get_random_array(size, max_coin_value):
    return list(np.random.randint(1, max_coin_value, size))

# Run the algorithm n times, with input sizes from min_size to max_size
min_size = 1000
max_size = 1000
n = 1
x = np.linspace(min_size, max_size, n).astype(int)

# Obtain results
results_baja_variabilidad = time_algorithm(valor_max_sophia, x, lambda s: [get_random_array(s, 100)])
print(results_baja_variabilidad)

results_alta_variabilidad = time_algorithm(valor_max_sophia, x, lambda s: [get_random_array(s, 10000)])
print(results_alta_variabilidad)
```

```
[7]  ✓ 3.0s
```

```
... {np.int64(1000): 0.27790131568908694}
    {np.int64(1000): 0.2773725509643555}
```

Para *partida_baja_variabilidad* utilizamos valores entre 1 y 100 y para *partida_alta_variabilidad* utilizamos valores entre 1 y 10.000, y para ambos usamos 1000 de monedas, y ejecutamos cada algoritmo una única vez. Se puede observar que para ambos casos los tiempos de ejecución no varían significativamente.

2.5. Complejidad

Nuestro algoritmo crea una matriz, donde los indices i de las filas representa la primer moneda de la fila, y j de columnas representa a la ultima moneda de la fila. La matriz inicializa con todos los valores en 0 y luego recorremos la matriz de forma diagonal donde si i=j (diagonal principal) asignamos con el valor de la moneda, luego si tenemos dos monedas elegimos el máximo de ambas y sino hallamos el maximo siguiendo la ecuacion de recurrencia planteada donde este último como se compara 4 valores siempre su complejidad es $\mathbf{O(1)}$. Ya que el algoritmo itera la matriz en todas sus posiciones, y en cada una realiza solo comparaciones y asignaciones las cuales son todas de complejidad $\mathbf{O(1)}$ el algoritmo queda con complejidad temporal de $\mathbf{O}(n^2)$. Como se utiliza la matriz para guardar información y no hay llamadas recursivas la complejidad espacial nos queda como $\mathbf{O}(n^2)$.

Para la reconstrucción de la solucion, el algoritmo se "para.[en] el extremo superior derecho de la matriz y después va realizando n saltos (n: cantidad de las monedas) y cada salto se realiza mediante una corroboración de $\mathbf{O(1)}$ donde evalua la ecuación de recurrencia para saber si se tomo la moneda izquierda o derecha. Las soluciones se guardan en una lista por ende la complejidad

espacial queda como **O(n)**.

2.6. Ejemplos de ejecución

A continuación mostraremos el seguimiento de la ecuación de recurrencia sobre una matriz de nxn de forma iterativa. Las filas corresponden a i (extremo derecho de la fila de monedas), y las columnas a j (extremo izquierdo de la fila de monedas). La fila de mondeas utilizada es [5, 10, 13, 2] y recordando la ecuación de recurrencia:

$$\text{OPT}(i,j) = \text{máx} \left(\begin{array}{l} \text{coins}[i] + \text{OPT}(i+2,j) \text{ if coins}[i+1] \geq \text{coins}[j] \text{ else } 0, \\ \text{coins}[i] + \text{OPT}(i+1,j-1) \text{ if coins}[i+1] < \text{coins}[j] \text{ else } 0, \\ \text{coins}[j] + \text{OPT}(i+1,j-1) \text{ if coins}[i] \geq \text{coins}[j-1] \text{ else } 0, \\ \text{coins}[j] + \text{OPT}(i,j-2) \text{ if coins}[i] < \text{coins}[j-1] \text{ else } 0 \end{array} \right)$$

Comenzamos con i=0 y j=0. Observamos que los casos base donde i=j definen la diagonal principal con los valores de la moneda, los anotamos. Además, las posiciones donde i ¿j las completamos con 0 ya que no tiene sentido que el extremo inicial sea un índice mayor al final.

	0	1	2	3	
0	5				
1	0	10			
2	0	0	13		caso base: i = j
3	0	0	0	2	

Luego tenemos el caso base donde hay solo dos monedas i+1 = j por lo tanto buscamos el máximo entre esos valores.

	0	1	2	3	
0	5	**max(5,10)=10**			
1	0	10	**max(13,10)=13**		
2	0	0	13	**max(13,2)=13**	caso base: en este caso hay solo dos monedas porque i+1 = j
3	0	0	0	2	

Para las demás posiciones, como ya no tenemos casos base, utilizamos la ecuación de recurrencia:

	0	1	2	3	
0	5	10	**18**		
1	0	10	13		
2	0	0	13	13	i=0 j=2
3	0	0	0	2	coins[0] + OPT(2,2) solo si coins[1](10) >= coins[2](13) => 0
					coins[0] + OPT(1,1) solo si coins[1](10) < coins[2](13) => 15
					coins[2] + OPT(1, 1) solo si coins[0](5) >= coins[1](10) => 0
					coins[2] + OPT(0,0) solo si coins[0](5) < coins[1](10) => 18

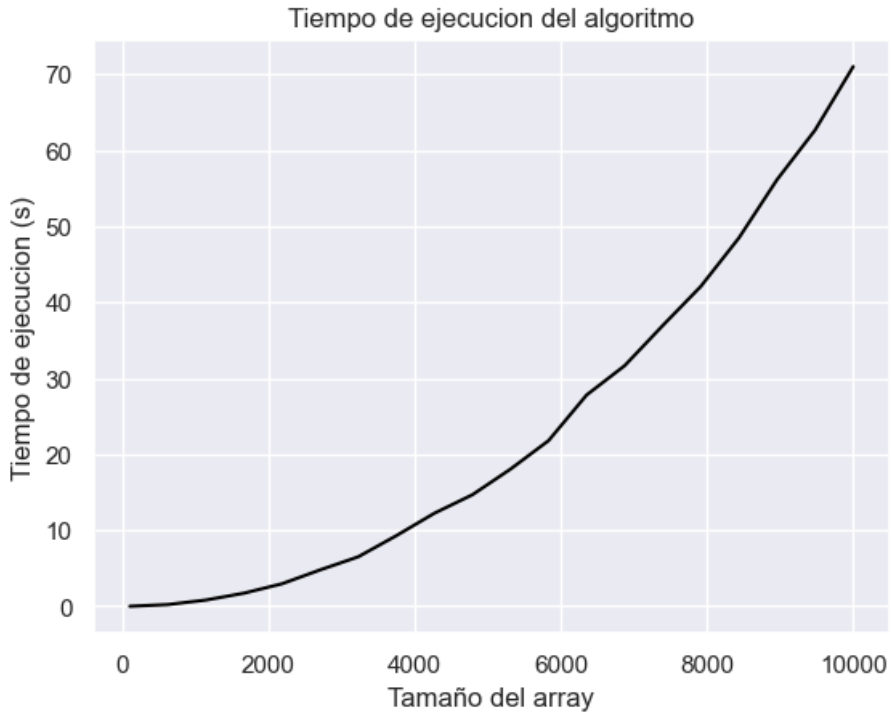
	0	1	2	3	
0	5	10	18		
1	0	10	13	**12**	
2	0	0	13	13	i=1 j=3
3	0	0	0	2	coins[1] + OPT(3,3) solo si coins[2](13) >= coins[3](2) => 12
					coins[1] + OPT(2,2) solo si coins[2](13) < coins[3](2) => 0
					coins[3] + OPT(2,2) solo si coins[1](10) >= coins[2](13) => 0
					coins[3] + OPT(1,1) solo si coins[1](10) < coins[2](13) => 12

	0	1	2	3	
0	5	10	18	**18**	
1	0	10	13	12	
2	0	0	13	13	i=0 j=3
3	0	0	0	2	coins[0] + OPT(2,3) solo si coins[1](10) >= coins[3](2) => 18
					coins[0] + OPT(1,2) solo si coins[1](10) < coins[3](2) => 0
					coins[3] + OPT(1,2) solo si coins[0](5) >= coins[2](13) => 0
					coins[3] + OPT(0,1) solo si coins[0](5) < coins[2](13) => 12

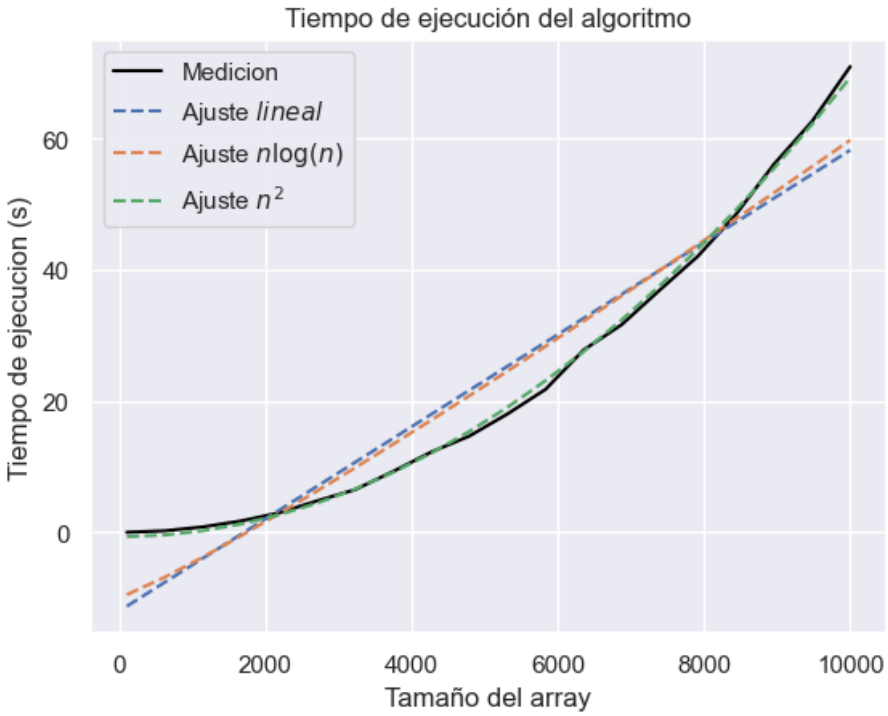
Se puede apreciar que para este ejemplo, Sophia gana con con un valor de 18 escogiendo las monedas de valor 5 y de 13.

3. Mediciones

Para estimar la complejidad de nuestro algoritmo, hemos ejecutado 20 partidas que van desde 100 hasta 10000 monedas. Además, cada partida es ejecutada 10 veces para obtener una estimación más precisa de los tiempos de ejecución del programa, a partir de la media de tiempos tomados por cada partida.



Como se puede apreciar, el tiempo de ejecución del algoritmo crece de forma cuadratica a medida que aumentamos el tamaño del arreglo de entrada del algoritmo.



Realizamos un ajuste lineal, un ajuste logratimico y uno cuadrático (n^2). Como podemos ver, el ajuste cuadratico es el que mejor se aproxima a nuestra curva de complejidad, tal como lo habíamos argumentado previamente en este informe. De la misma forma, se puede observar el gran error que presenta el ajuste lineal y cuadratico en comparación.

De esta forma, podemos apreciar de forma empirica que el algoritmo tiene complejidad O(n^2), ya que el tiempo de ejecución del algoritmo crece de forma cuadratica con respecto al tamaño de la entrada.

4. Conclusiones

Analizando el problema y la implementación del algoritmo mediante programación dinámica, demostró ser efectivo en encontrar el óptimo valor para Sophia, aunque en algunos casos Sophia no consigue ganar la partida. A lo largo de las pruebas de variabilidad se vio que no afecta al resultado final del algoritmo. Esto se debe a que la estrategia donde elige izquierda o derecha sin importar la variabilidad de la moneda, independientemente de cómo estén distribuidos los valores restantes.

Con base en el análisis inductivo, podemos concluir que la ecuación de recurrencia planteada es correcta y garantiza que Sophia siempre obtendrá el valor máximo posible, dadas las condiciones del juego y el comportamiento Greedy de Mateo. Aunque no podemos asegurar que Sophia gane en todos los casos (ya que Mateo también juega de manera óptima para sí mismo), la ecuación asegura que Sophia acumulará el máximo valor posible con las decisiones que tome en cada turno.

Este resultado se logra al considerar todas las posibles elecciones de Sophia y anticipar las decisiones que tomará Mateo de forma Greedy en cada escenario. Así, Sophia siempre selecciona la opción que maximiza su valor a largo plazo, sin importar las acciones de Mateo. Por lo tanto, la estrategia óptima para Sophia queda reflejada en la ecuación de recurrencia, que se implementa eficazmente mediante Programación Dinámica.

El algoritmo de programación dinámica presentado tiene una complejidad de tiempo $O(n^2)$, donde n es el número de monedas. Esto se debe a que llenamos una tabla $dp[i][j]$ de tamaño $n \times n$, donde cada celda se calcula en tiempo constante $O(1)$, considerando las dos posibles elecciones de Sophia y las decisiones Greedy de Mateo.

Además, la reconstrucción de las decisiones óptimas de Sophia también se realiza en $O(n)$, ya que solo recorremos una vez la tabla de decisiones para determinar las monedas que Sophia debe elegir.

En resumen, el algoritmo es eficiente, ya que resuelve el problema en tiempo cuadrático $O(n^2)$, lo cual es razonable para valores de n moderados. Esto lo hace adecuado para casos prácticos donde se trabaja con un número considerable de monedas.

Para finalizar, concluimos que el algoritmo tiene alta complejidad, pero es un algoritmo óptimo que permite encontrar la solución al problema de Sophia, y no una aproximación *buena* como hacía el algoritmo Greedy.