.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

Gian Keberlein
109585

Juan Manuel Dalmau Renella
109555

1. Introduccion

Seguimos con la misma situación planteada en el trabajo práctico anterior, pero ahora pasaron varios años. 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.

Nosotros debemos plantear ese algoritmo para Sophia.

2. Analisis del problema

Realizaremos un analisis teorico y empirico de un algoritmo de programcion dinamica para indicar que monedas debe elegir Sophia dada una secuencia de monedas para maximizar el valor acumulado en todos los casos, sabiendo que ella comienza primero, y que Mateo juega de forma Greedy.

Como intentaremos hallar la solucion con un planteo bottom-up, lo primero que vamos a hacer es buscar la ecuacion de recurrencia. Para ello, debemos hallar la forma de los subproblemas y ver como ayudan a resolver problemas mas grandes hasta llegar a la solucion.

2.1. Ecuacion de recurrencia

Supongamos que tenemos una secuencia de n monedas.

- Si n = 0: Entonces no agarramos ninguna moneda y es un empate.
- Si n = 1: Entonces Sophia debe agarrar la unica moneda y su valor es el maximo posible.
- Si n = 2: Sophia debe agarrar la moneda de mayor valor, sabiendo que Mateo elegira la que sobre. En este caso, Opt(M) = max(m0, m1).
- Si n = 3 o mayor: Hay que analizar varios casos. Que ocurre cuando Sophia elije la moneda de la izquierda, y que ocurre cuando elije la de la derecha.

Supongamos, para n = 3 o mayor, que tenemos dos indices i y j que representan el inicio y el final de la secuencia. Para que Sophia pueda elegir el orden que maximiza el valor de sus monedas, debe quedarse con el lado que optimiza las sumas, digamos:

$$\text{Opt(M)} = \max(\text{M[i]} + \text{algo}, \text{M[j]} + \text{algo})$$

Ese "algo" al que nos referimos, es el valor optimo del subproblema que aparece cuando Sophia elige la moneda primero. Pero hay que tener en cuenta que luego Mateo elegira la moneda de mayor valor de los extremos de ese subarreglo.

Por lo tanto, para nuestra ecuacion debemos analizar: si Sophia saca la moneda en i, entonces Mateo podra agarrar la moneda en i + 1 o en f, por lo que hay que buscar el optimo (la mejor opcion) entre los subarreglos (i+2, f) y (i+1, f-1), porque depende de que moneda elija este. Para el otro caso, donde Sophia agarra la moneda en f, debemos analizar la mejor opcion entre los subarreglos (i+1, f-1) y (i, f-2), por los mismos motivos que antes.

Para saber con cual de ambos subarreglos debemos quedarnos para la resolucion del problema mayor, entendimos que como Mateo siempre elegira la moneda mayor, debemos quedarnos con el lado que el no elegiria, es decir el minimo.

La ecuacion de recurrencia entonces nos queda:

$$Opt(M[i:f]) = Max(M[i]+Opt(JuegaMateo(M,i+1,f)), M[f]+Opt(JuegaMateo(M,i,f-1))) \quad (1)$$

Con JuegaMateo igual a:

```
def juega_mateo(monedas, izq, der):
    if monedas[izq] > monedas[der]:
        return monedas[izq+1:der]
    return monedas[izq:der-1]
```

Listing 1: Implementación de JuegaMateo() para la ecuación de recurrencia

2.2. Implementación propuesta

Teniendo en cuenta lo anteriormente mencionado, se propone esta implementacion en Python:

```
def monedas_dinamicas(mon):
    # Inicializo la lista de optimos
    optimos =  [
        [(0,0,0)] * (len(mon)+1),
        [(i, i, mon[i]) for i in range(len(mon))]
    ]
    # Voy llenando las listas de optimos para todos los subarreglos posibles, empezando por los de longitud 2 y terminando en el de longitud n
    for i in range(2,len(mon)+1):
        local =  []
        # Recorro todos los subarreglos de longitud i combinando 2 subarreglos de longitud i-1
        for j in range(len(mon)+1-i):
            # Obtengo los extremos de mi nuevo subarreglo
            izq, der = optimos[i-1][j][0], optimos[i-1][j+1][1]

            # Simulo los proximos subarreglos si agarro la moneda de la izquierda o la de la derecha sabiendo que Mateo agarra la moneda de mayor valor
            prox_subarr_si_agarro_izq = (izq+2,der) if mon[izq+1] > mon[der] else (izq+1, der-1)
            prox_subarr_si_agarro_der = (izq,der-2) if mon[der-1] > mon[izq] else (izq+1, der-1)

            # Calculo los optimos posibles sacando la moneda de la izquierda o la de la derecha y me quedo con el maximo
            opcion_izq, opcion_der = mon[izq] + optimos[i-2][prox_subarr_si_agarro_izq[0]][2], mon[der] + optimos[i-2][prox_subarr_si_agarro_der[0]][2]
            local.append((izq, der, max(opcion_izq, opcion_der)))

        optimos.append(local)

    # Reconstruyo la solucion
    return recontruir(mon, optimos)
```

Listing 2: Implementación del algoritmo principal

Con la función de reconstrucción como:

```
def recontruir(monedas, optimos):
    # Inicializo variables
    final = []
    g_sophia = g_mateo = indice = 0

    # Recorro la lista de optimos de atras para adelante
    for i in range(len(optimos)-1, 0, -2):
        izq, der, opt = optimos[i][indice]

        # Veo si queda solo un elemento o si agarre la moneda de la izquierda con la ecuacion de recurrencia
        prox_i_si_agarro_izq = izq + 2 if izq == der or monedas[izq+1] > monedas[der] else izq + 1
```

```
        if izq == der or opt == monedas[izq] + optimos[i-2][prox_i_si_agarro_izq
    ][-1]:
            final.append(f"Sophia debe agarrar la primera ({monedas[izq]})")
            g_sophia += monedas[izq]
            if izq == der: break # Si era solo un elemento, termino
            mateo, g_mateo, indice = juega_mateo(monedas, izq+1, der, g_mateo) # Si
     no, juega Mateo
        else:
            final.append(f"Sophia debe agarrar la ultima ({monedas[der]})")
            g_sophia += monedas[der]
            mateo, g_mateo, indice = juega_mateo(monedas, izq, der-1, g_mateo)

        final.append(mateo)

    return final, g_sophia, g_mateo
```

Listing 3: Implementación del algoritmo de reconstrucción

Y la función que simula la jugada de Mateo:

```
def juega_mateo(monedas, izq, der, g_mateo):
    if monedas[izq] > monedas[der]:
        mateo = f"Mateo agarra la primera ({monedas[izq]})"
        g_mateo += monedas[izq]
        indice = izq + 1
    else:
        mateo = f"Mateo agarra la ultima ({monedas[der]})"
        g_mateo += monedas[der]
        indice = izq
    return mateo, g_mateo, indice
```

Listing 4: Implementación de JuegaMateo() para el algoritmo de reconstrucción

2.3. Ejemplo corto

Dado un arreglo de monedas M = [520, 781, 334, 568, 706, 362, 201, 482, 19, 145], el resultado del algoritmo y solución optima sería:

```
(['Sophia debe agarrar la ultima (145)', 'Mateo agarra la primera (520)', 'Sophia
    debe agarrar la primera (781)', 'Mateo agarra la primera (334)', 'Sophia debe
    agarrar la primera (568)', 'Mateo agarra la primera (706)', 'Sophia debe
    agarrar la primera (362)', 'Mateo agarra la primera (201)', 'Sophia debe
    agarrar la primera (482)', 'Mateo agarra la ultima (19)'], 2338, 1780)
```

Listing 5: Resultado de ejecutar el algoritmo en M

Con el primer elemento de la tupla siendo el desarrollo del juego óptimo, el segundo elemento siendo la ganancia total de Sophia, y el ultimo siendo la ganancia total de Mateo.

3. Analisis de complejidad

3.1. Complejidad teórica

- **juega_mateo():** La complejidad de este algoritmo es $O(1)$, ya que solo hace operaciones en tiempo constante.
- **recontruir():** La complejidad de este algoritmo depende en su totalidad de la cantidad de monedas/optimos que se le pasen. Recorre la matriz de optimos accediendo a un elemento de cada fila y haciendo operaciones en tiempo constante. Dado esto, la complejidad es $O(n)$, donde n es la cantidad de monedas.
- **monedas_dinamicas():** El algoritmo depende en su totalidad del tamaño del arreglo de monedas. Éste genera una matriz triangular de base n, donde n es la cantidad de monedas, y

luego hace operaciones en tiempo constante. Por lo tanto, se hacen $(n^2)/2$ operaciones, pero como el 2 es constante, la complejidad queda $O(n^2)$. Luego de esto está la reconstrucción de la solución, que es lineal en la cantidad de monedas, por lo que la cota superior de la complejidad es $O(n^2)$.

3.2. Analisis de variabilidad

Para comprobar cómo afecta la variabilidad de los valores de las monedas a los tiempos de ejecución, se realizaron pruebas en las que se calculó la duración del algoritmo actuando sobre conjuntos de monedas con distintas características. Cada conjunto tiene una cantidad N de monedas representadas por valores enteros aleatorios. El primero, de baja variabilidad, está conformado por monedas de valores entre 1 y 10, el segundo, de mediana variabilidad, contiene valores entre 1 y 5,000 (Cinco mil), y el tercero, de alta variabilidad, contiene valores entre 1 y 100,000,000,000 (Cien mil millones). Para visualizar las diferencias temporales, se le tomó el tiempo que tardaba el algoritmo en terminar de procesar cada conjunto, haciendo variar el tamaño N de los mismos. Se obtuvieron los resultados que se ven a continuación. Estos se presentan en modo de tabla para exactitud y figura para mejor visualizacion.

N. Monedas	T. Poco Variado (s)	T. Medianamente Variado (s)	T. Muy Variado (s)
1	0.0	0.0	0.0
10	0.0	0.0	0.0
100	0.0	0.0	0.0
1000	0.393899	0.408155	0.388392
6000	15.218278	15.240525	14.95258
10000	42.669973	43.034896	43.72174
15000	96.702918	97.0801	98.068642
20000	244.527302	240.371374	295.179528

Cuadro 1: Tabla de tiempos de ejecución de arreglos con distinta variabilidad de valores

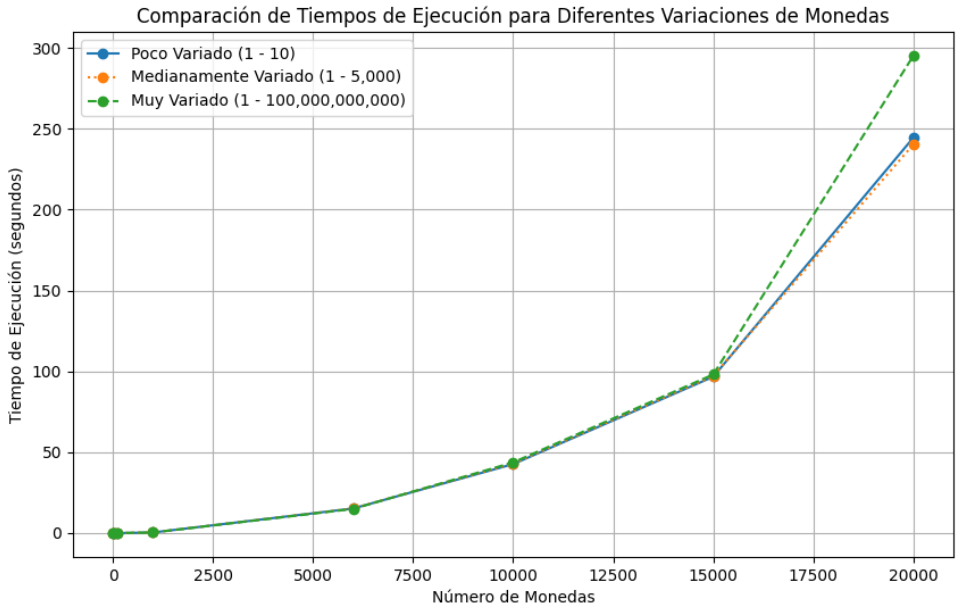

Figura 1: Lineplot del Cuadro 1

Se puede observar que hasta los 100 elementos, los tiempos de ejecución eran tan pequeños que ni se pudieron medir. Luego a partir de los 1,000 elementos, los tiempos de ejecución incrementan a una velocidad cada vez mayor, lo cual se corresponde con la complejidad teórica analizada previamente. Esta diferencia llega a su máximo a las 20,000 monedas, entre las cuales hay una diferencia de más de 50 segundos.

Como era de esperarse, el arreglo de monedas con menor variabilidad tarda menos en procesarse. Esto puede deberse a que los números pequeños se almacenan y manejan de forma más eficiente en memoria, lo que afectará mucho los tiempos de nuestro algoritmo de programación dinámica que guarda las soluciones óptimas de todos los subarreglos posibles. Las operaciones con números pequeños implican un menor costo computacional que los números grandes, como los de cien mil millones, ya que estos últimos requieren más recursos de procesamiento y memoria. Esto afecta desde el lado de la memoria caché, ya que al ser datos más grandes, baja su eficiencia de guardado en caché, y puede ocurrir más frecuentemente un 'Cache Miss' y que se tenga que ir a buscar el dato a la memoria principal, y tardar un poco más. Además, en Python, las operaciones con números pequeños (que caben en una palabra de máquina) son mucho más rápidas que las realizadas con enteros grandes. Esto significa que las sumas, restas y comparaciones entre monedas del conjunto de poca variabilidad se ejecutan más eficientemente.

4. Ejemplos

Para demostrar que el algoritmo elegido siempre obtiene la solucion optima realizamos algunas pruebas.

Como primer prueba utilizamos los datos provistos por la catedra, de los cuales ya sabıamos cual era el resultado esperado.

Para los 10 archivos de prueba, las ganancias fueron las esperadas:

- 5.txt: Sophia obtiene 1.483 y Mateo 1.268.
- 10.txt: Sophia obtiene 2.338 y Mateo 1.780.
- 20.txt: Sophia obtiene 5.234 y Mateo 4.264.
- 25.txt: Sophia obtiene 7.491 y Mateo 6.523.
- 50.txt: Sophia obtiene 14.976 y Mateo 13.449.
- 100.txt: Sophia obtiene 28.844 y Mateo 22.095.
- 1000.txt: Sophia obtiene 1.401.590 y Mateo 1.044.067.
- 2000.txt: Sophia obtiene 2.869.340 y Mateo 2.155.520.
- 5000.txt: Sophia obtiene 9.939.221 y Mateo 7.617.856.
- 10000.txt: Sophia obtiene 34.107.537 y Mateo 25.730.392.

Otro ejemplo que usamos para validar que el funcionamiento es el correcto es el caso donde la secuencia de monedas es [1, 12, 5, 1].

Aunque es un caso simple, se puede ver que si Sophia falla al elegir la primera moneda cuando ambos extremos tienen el mismo valor, entonces no esta analizando que extremo maximiza su valor acumulado final. Sophia debe bloquear a Mateo de agarrar la moneda de valor 12, y eso lo lograria eligiendo primero la ultima moneda forzando a Mateo a elegir la de valor 5.

Con nuestro algoritmo, el resultado final es:

- Sophia debe agarrar la ultima (1).

- Mateo agarra la ultima (5).
- Sophia debe agarrar la ultima (12).
- Mateo agarra la ultima (1).

Ganancia Sophia: 13

Ganancia Mateo: 6

Vemos que efectivamente, Sophia esta maximizando su valor.

5. Verificacion de la complejidad temporal

Para corroborar que la complejidad temporal del algoritmo es efectivamente del orden lineal, realizaremos mediciones de los tiempos de ejecucion del mismo para diferentes cantidades de mo nedas. Luego, realizaremos un ajuste de recta utilizando la tecnica de cuadrados minimos para verificar que, aproximadamente, los tiempos obtenidos son resultados de una funcion lineal.

Estas pruebas se realizaron en una misma computadora para desestimar el error causado por diferencias en entornos de desarrollo.

Con el siguiente programa, leimos las mediciones obtenidas en el analisis de variabilidad e hicimos un ajuste por cuadrados minimos para los tres casos de tiempos posibles para cada cantidad de monedas.

```
import csv
import matplotlib.pyplot as plt
import numpy as np

# ========= Aproximacion por cuadrados minimos ========= #

def ajuste_cuadratico(cantidades, tiempos):
    cantidades = np.array(cantidades)
    tiempos = np.array(tiempos)

    A = np.vstack([cantidades**2, cantidades, np.ones(len(tiempos))]).T
    AtA = A.T @ A
    Atb = A.T @ tiempos
    x = np.linalg.solve(AtA, Atb)

    a, b, c = x
    tiempos_predecidos = a * cantidades**2 + b * cantidades + c
    errores_residuales = list(map(abs, tiempos - tiempos_predecidos))

    return x, errores_residuales

# ========== Graficos ========== #

def graficar_benchmark(cantidades, tiempos):
    plt.plot(cantidades, tiempos, marker='o', linestyle='none')
    plt.title("Tiempos de ejecucion del algoritmo")
    plt.xlabel("Cantidad de monedas")
    plt.ylabel("Tiempo de ejecucion (segundos)")
    plt.grid(True)
    plt.show()

def graficar_ajuste_cuadratico(cantidades, tiempos, coeficiente_2, coeficiente_1,
    coeficiente_0):
    tiempos_ajustados = coeficiente_2 * np.array(cantidades)**2 + coeficiente_1 *
    np.array(cantidades) + coeficiente_0
    plt.plot(cantidades, tiempos, 'o', label='Tiempos medidos')  # Datos originales
    plt.plot(cantidades, tiempos_ajustados, '-', label=f'Ajuste cuadratico: t(n) =
    {coeficiente_2:.5e} * n^2 + {coeficiente_1:.5e} * n + {coeficiente_0:.5e}')
    plt.xlabel('Cantidad de monedas')
    plt.ylabel('Tiempo de ejecucion (segundos)')
```

```
    plt.title('Ajuste por cuadrados minimos usando $A^T Ax = A^T b$')
    plt.legend()
    plt.grid(True)
    plt.show()

def graficar_error(errores):
    plt.plot(errores, marker='o')
    plt.title("Errores residuales del ajuste")
    plt.xlabel("Cantidad de monedas")
    plt.ylabel("Error residual")
    plt.grid(True)
    plt.show()

if __name__ == "__main__":
    with open('./tablas/tabla_variabilidad.csv', 'r') as csvfile:
        reader = csv.reader(csvfile)
        next(reader)

        cantidades = []
        tiempos_poco_variadas = []
        tiempos_medianamente_variadas = []
        tiempos_muy_variadas = []
        for row in reader:
            cantidades.append(int(row[0]))
            tiempos_poco_variadas.append(float(row[1]))
            tiempos_medianamente_variadas.append(float(row[2]))
            tiempos_muy_variadas.append(float(row[3]))

    # Graficamos los tiempos medidos
    x, errores = ajuste_cuadratico(cantidades, tiempos_poco_variadas)
    graficar_ajuste_cuadratico(cantidades, tiempos_poco_variadas, x[0], x[1], x[2])
    graficar_error(errores)

    x, errores = ajuste_cuadratico(cantidades, tiempos_medianamente_variadas)
    graficar_ajuste_cuadratico(cantidades, tiempos_medianamente_variadas, x[0], x
    [1], x[2])
    graficar_error(errores)

    x, errores = ajuste_cuadratico(cantidades, tiempos_muy_variadas)
    graficar_ajuste_cuadratico(cantidades, tiempos_muy_variadas, x[0], x[1], x[2])
    graficar_error(errores)
```

Con los coeficientes aproximados obtenidos en las resoluciones de los sistemas para cada caso, pudimos graficar estas curvas que, por el bajo error, validan la complejidad de nuestro algoritmo.

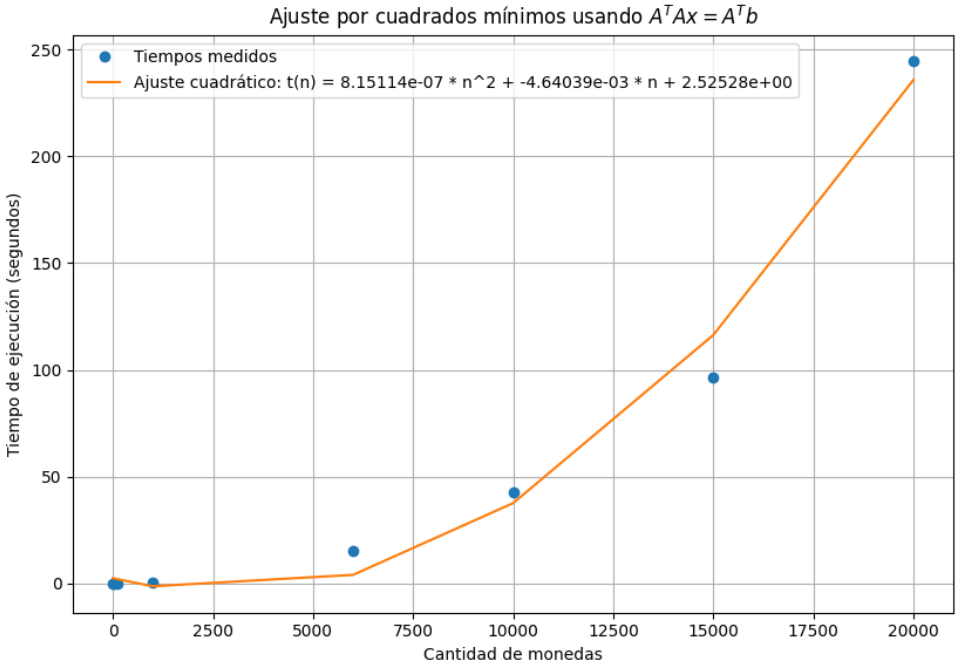

Figura 2: Ajuste por cuadrados minimos para valores poco variados

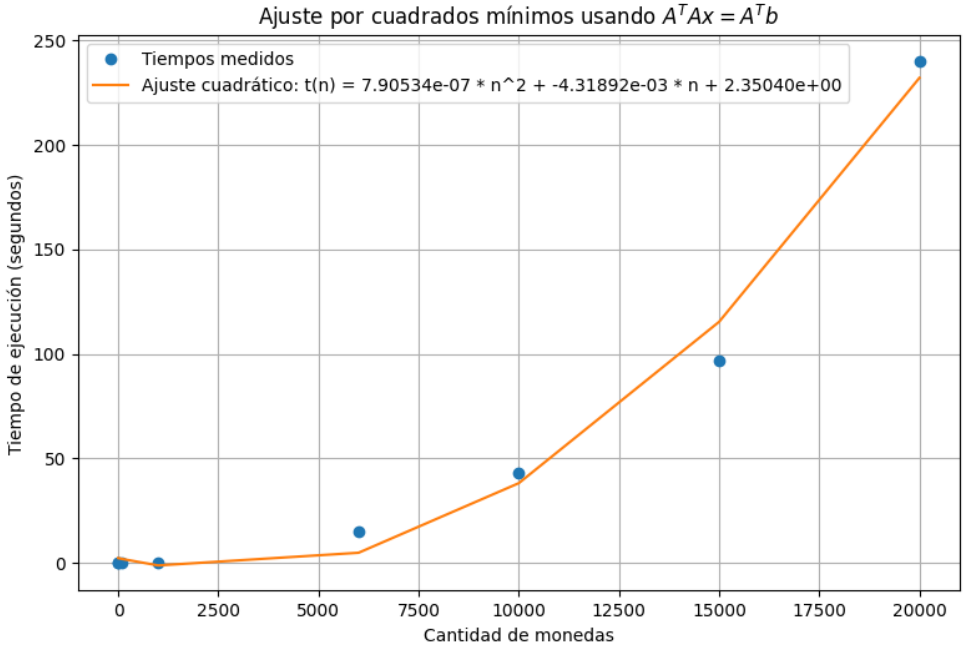

Figura 3: Ajuste por cuadrados minimos para valores medianamente variados

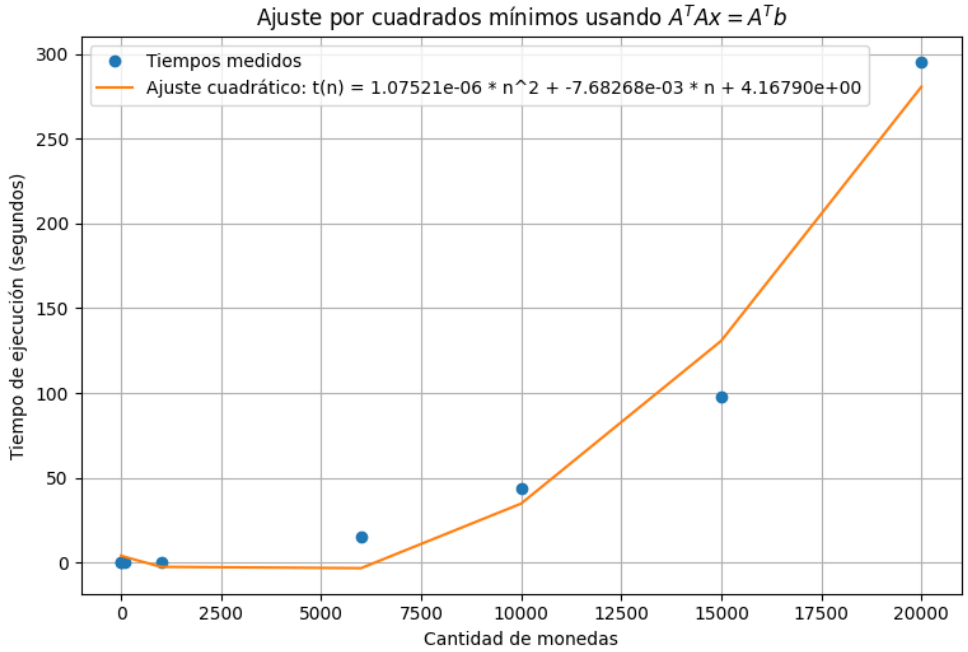

Figura 4: Ajuste por cuadrados minimos para valores muy variados

Y en las siguientes figuras podemos ver los graficos de errores de los ajustes para cada caso.

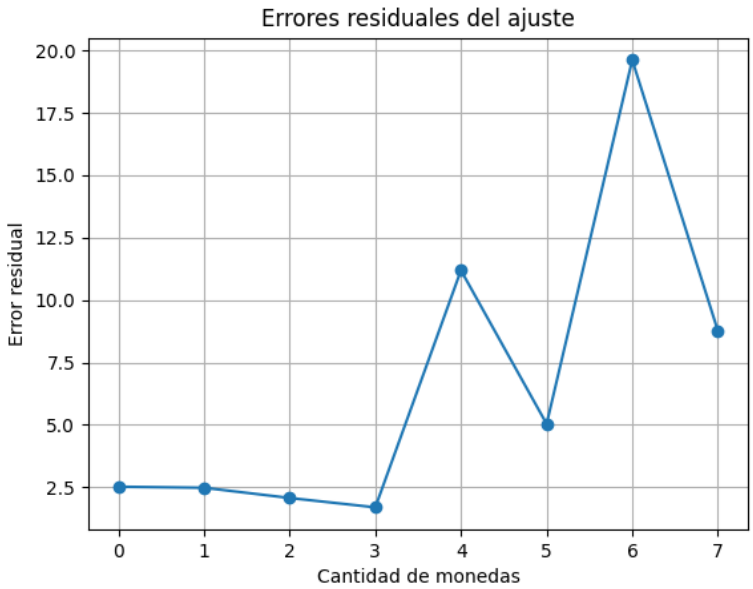

Figura 5: Errores en el ajuste para valores poco variados

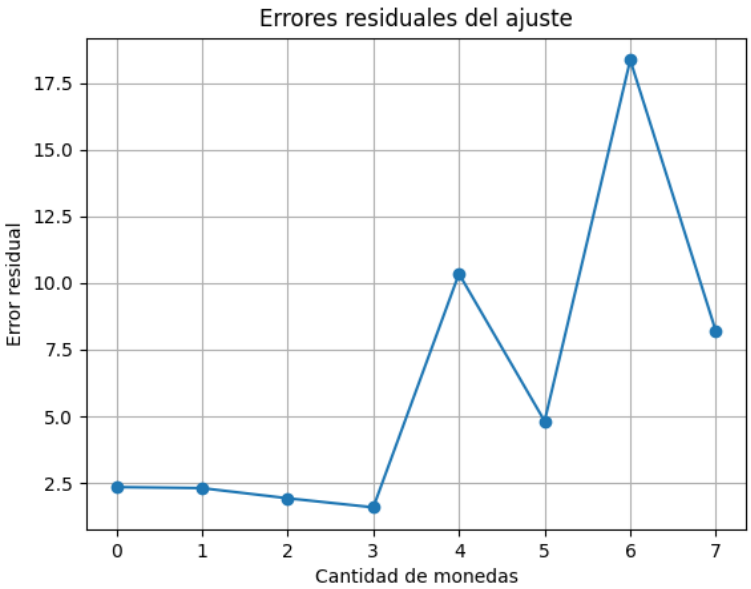

Figura 6: Errores en el ajuste para valores medianamente variados

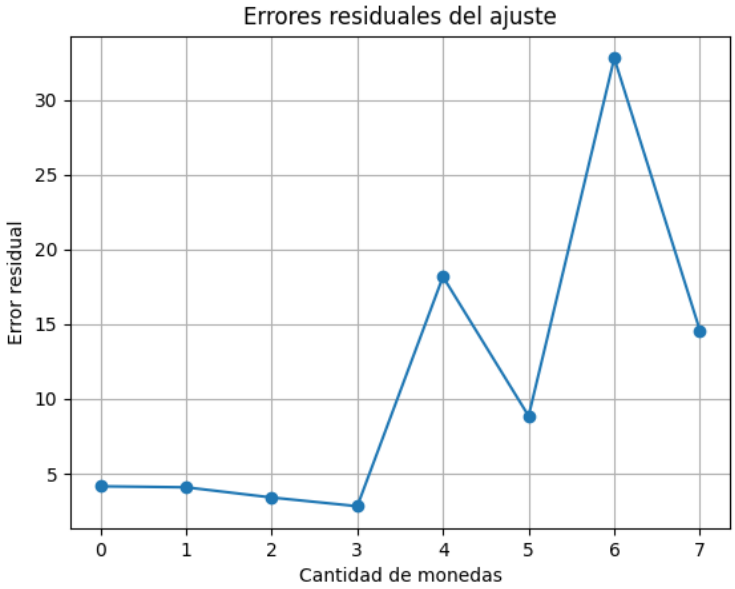

Figura 7: Errores en el ajuste para valores muy variados

6. Conclusión

Luego del desarrollo de este informe, queda demostrado que la implementación propuesta logra maximizar el valor acumulado para Sophia en todos los casos, considerando la estrategia greedy de Mateo. Los resultados de las pruebas con diversos conjuntos de datos, incluyendo los proporcionados por la cátedra, confirman la optimalidad de las soluciones obtenidas.

La complejidad temporal del algoritmo, teóricamente establecida como $O(n^2)$, se verifica mediante el análisis de los tiempos de ejecución para diferentes tamaños de entrada. Los ajustes por cuadrados mínimos realizados corroboran esta complejidad, con errores residuales bajos que validan la precisión del ajuste. Además, se observa que la variabilidad en los valores de las monedas tiene un impacto en el rendimiento, siendo los conjuntos con menor variabilidad los que se procesan más rápidamente, lo cual puede deberse a la eficiencia en el manejo de números pequeños por parte de Python y los sistemas de memoria.