Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

Trabajo Práctico 2
Programación Dinámica

28 de diciembre de 2024

Lucas Araujo
109867

Leandro Brizuela
109842

José Rafael Patty Morales
109843

Matías Rea
99770

Marcos Tomás Weng Xu
109153

Índice

1. Planteamiento del problema y formulación de la ecuación de recurrencia

En este trabajo de ejemplo realizaremos el análisis teórico y empírico de diferentes algoritmos que sirven para resolver el mismo problema: obtener el valor del máximo elemento de un arreglo desordenado de n elementos.

1.1. Descripción del Problema

Se requiere encontrar el subconjunto de máxima suma de un conjunto , bajo ciertas condiciones. Esto está enmarcado en el modelado de un juego por turnos, donde compiten dos jugadores para obtener la máxima suma al finalizar sus turnos, el que obtenga el máximo gana la partida. Las reglas del juego son:

- Hay un conjunto de monedas, ordenadas en una fila de forma aleatoria.
- Cada jugador, en su turno, puede elegir una moneda, la cual solo puede estar en los extremos de la fila, ya sea encabezando la fila o en la parte posterior de esta. Al elegir la moneda la quita de la fila y la guarda en el conjunto propio del jugador.
- Al no haber más monedas, cada jugador suma el total de las monedas que tiene y gana el que posea el mayor puntaje.

Llamando a los dos jugadores con los nombres Mateo y Sophia. El orden de los turnos es invariable, primero juega Sophia y luego Mateo. Mateo en su turno siempre tomará la moneda de mayor valor entre las dos posibles, entre la del frente y la de atrás de la fila. En la situación que las dos sean de igual valor, tomara la primera. Debemos diseñar un algoritmo que permita a Sophia ganar todas las veces que sea posible, utilizando la técnica de programación dinámica.

2. Formulación de la Ecuación de Recurrencia

Para formular la ecuación de recurrencia empezamos desde los problemas mas chicos a los mas grandes. Definimos como problemas mas chicos a los cuales tienen menos cantidad de monedas. Como necesitaremos guardar los resultados óptimos en todos los intervalos [i,j] de monedas posibles usaremos una matriz bidimensional donde i representara la primera moneda y j la ultima moneda. Cada celda dp_sophia[i][j] almacena la solución óptima para el subproblema correspondiente a ese intervalo.

Problema con cero monedas:
La decisión óptima de Sophia es no agarrar ninguna moneda. Obtiene el máximo posible.

Problema con una moneda:
La decisión óptima de Sophia es agarrar la moneda.

Problema con dos monedas:
La decisión óptima de Sophia es agarrar la moneda con el máximo valor entre las dos. optS = max(moneda número 1, moneda número 2)

Problema con 3 monedas:

Primera opción: Sophia toma la primera moneda

Si monedas $[i+1] \geq monedas[j]$, Mateo elige la moneda $monedas[i+1]$, dejando el subproblema con la moneda restante: $[i+2, j]$. Si monedas $[i+1] < monedas[j]$, Mateo elige la moneda $monedas[j]$, dejando el subproblema con la moneda restante: $[i+1, j-1]$, que corresponde a

$monedas[i+1]$.

Segunda opción: Sophia toma la ultima moneda Este caso es similar a si Sophia toma la primera moneda. Si $monedas[j-1] > monedas[i]$, Mateo elige la moneda $monedas[j-1]$, dejando el subproblema con la moneda restante: $[i, j-2]$. Si $monedas[j-1] < monedas[i]$, Mateo elige la moneda $monedas[i]$, dejando el subproblema con la moneda restante: $[i+1, j-1]$, que corresponde a $monedas[j-1]$.

Problema con 4 monedas El caso para 4 monedas es lo mismo que para 3. Podemos armar entonces nuestra ecuación de recurrencia para n monedas.

$$\text{dp\_sophia}[i][j] = \text{máx}\left(\begin{array}{l} \text{monedas}[i] + \begin{cases} \text{dp\_sophia}[i+2][j], & \text{si monedas}[i+1] \geq \text{monedas}[j] \\ \text{dp\_sophia}[i+1][j-1], & \text{en otro caso} \end{cases}, \\ \text{monedas}[j] + \begin{cases} \text{dp\_sophia}[i][j-2], & \text{si monedas}[j-1] \geq \text{monedas}[i] \\ \text{dp\_sophia}[i+1][j-1], & \text{en otro caso} \end{cases} \end{array}\right)$$

Como se ve en la ecuación de recurrencia se busca encontrar el máximo entre si Sophia agarra la primera moneda o si Sophia agarra la última moneda.

3. Demostración de la optimalidad

Para demostrar la optimalidad empezamos desde los problemas mas chicos a los mas grandes. Definimos como problemas mas chicos a los cuales tienen menos cantidad de monedas.

Problema con cero monedas:
La decisión óptima de Sophia es no agarrar ninguna moneda. Obtiene el máximo posible.

Problema con una moneda:
La decisión óptima de Sophia es agarrar la moneda.

Problema con dos monedas:
La decisión óptima de Sophia es agarrar la moneda con el máximo valor entre las dos. optS = max(moneda número 1, moneda número 2)

Problema con n monedas:
La primera opción de Sophia: puede decidir tomar la moneda del frente de la fila. En este caso se debe desechar la moneda que tomará Mateo. Luego queda la fila con n-2 monedas, y se reduce a un problema más pequeño, aplicando la decisión óptima para este. Por ejemplo, en el caso de 3 monedas, Sophia toma la primera moneda, posición 1. Se desecha la moneda de mayor valor entre la segunda y la tercera. Queda el problema de 1 moneda y se resuelve como tal.

La otra opción de Sophia: Puede decidir tomar la moneda posterior de la fila. En este caso se debe desechar la moneda que tomará Mateo. Luego queda la fila con n-2 monedas, y se reduce a un problema más pequeño, aplicando la decisión óptima para este. Por ejemplo, en el caso de 3 monedas, Sophia toma la última moneda, posición 1. Se desecha la moneda de mayor valor entre la primera y la segunda. Queda el problema de 1 moneda y se resuelve como tal.

En cada turno, Sophia evalúa ambos extremos, monedas[i] y monedas[j]

Para cada extremo (monedas[i] o monedas[j]), Sophia evalúa cuál es la mejor opción posible para la cantidad de monedas restantes, considerando de antemano cuál será la jugada óptima de

Mateo.

La decisión de Sophia en cada turno se basa en los resultados óptimos de subproblemas más pequeños, que ya han sido calculados. Esto garantiza que cada elección que tome sea parte de una estrategia globalmente óptima.

De este modo, Sophia selecciona la opción que maximiza su ganancia total en ese paso, asegurando que las decisiones actuales conduzcan al resultado más favorable al finalizar el juego.

De esta forma el algoritmo a cada paso utiliza el óptimo de un subproblema menor.

4. Desarrollo y análisis del algoritmo de programación dinámica

Partiendo de la ecuación de recurrencia desarrollada para resolver el problema de maximizar las ganancias de Sophia al elegir monedas de una fila, implementamos un enfoque basado en programación dinámica. Al analizar la ecuación y como se completa la matriz utilizada, decidimos completarla

```
1  def optimal_strategy(coins):
2      n = len(coins)
3      dp_sophia = [[0 for _ in range(n)] for _ in range(n)]
4
5      # Caso Base
6      for i in range(n):
7          dp_sophia[i][i] = coins[i]
8
9      is_odd = n % 2
10
11     for length in range(2+ is_odd, n+1, 2):
12         for i in range(n - length + 1):
13             j = i + length - 1
14             # ecuacion de recurrencia
15             dp_sophia[i][j] = max(coins[i] + (dp_sophia[i+2][j] if i+2 <= j and
       coins[i+1] >= coins[j] else dp_sophia[i+1][j-1]),
16                                   coins[j] + (dp_sophia[i][j-2] if j-2 >= i and
       coins[j-1] > coins[i] else dp_sophia[i+1][j-1]))
17
18     return dp_sophia
```

4.1. Análisis de Complejidad

Recorremos el código línea por línea para calcular la complejidad, ya sea temporal como espacial.

Complejidad temporal:

- Línea 3: se crea una matriz de NxN para todos los problemas de N monedas. $O(n^2)$.
- Línea 6 y 7: Se recorre la diagonal de la matriz (n) para completar con el valor de una moneda. El coste por completar el índice [i,i] de la matriz es k, de valor constante. $O(n \cdot k) = O(n)$, al ser n las operaciones en tiempo constante.
- Línea de 11-16:
  - Línea 11, se recorren n elementos con un for.
  - Línea 12, se recorre con un for n elementos, dentro del for precedente.
  - Línea 13 a 16: se completa la matriz realizando operaciones de tiempo constante, como comparaciones de números, sumas, y operadores max, dentro de los dos for.

Entonces el tiempo total es

$$O(n \cdot n \cdot k) = O(n^2)$$

El costo total del algoritmo es: Complejidad temporal = $O(n^2)+O(n)+O(n^2) = 2 \cdot O(n^2)+O(n)$

Basándonos estrictamente en la notación O, y como los términos de mayor valor vuelven despreciables a los de menor, asi como las constantes multiplicativas: Complejidad temporal final = $O(n^2)$.

4.2. Análisis espacial

El algoritmo que utiliza la técnica de programación dinámica, solo aloca en memoria una matriz de tamaño NxN, por lo tanto: Complejidad espacial = NxN
En cambio el algoritmo que construye la solución, utiliza dos vectores con las monedas que agarra Sophia y Mateo respectivamente, es decir, aloca 2 vectores de tamaño $\frac{n}{2}$, por lo tanto: Complejidad espacial = $2 * \frac{n}{2} = n$.

4.3. Analisis de Variablidad

Analizamos si la variabilidad de las monedas afecta a los tiempo del algoritmo planteado. Lo hicimos resolviendo el problema con 10000 monedas. En la cual hicimos variaciones por el rango de valores de las monedas. Los rangos varian de 1-10, 1-1000 y de 1-100000. Tambien hicimos variar el tipo de dato, entre un arreglo solo de monedas de tipo entero y un arreglo solo de monedas de tipo flotante. Hicimos los ensayos 3 veces y sacamos un promedio. Que esta representado en las siguientes tablas:

Cantidad de monedas	Variabilidad por rango	Tiempo 1	Tiempo 2	Tiempo 3	Promedio de tiempo
10000	1-10	29,92	27,32	27,24	28,16
10000	1-1000	28,01	28,37	27,87	28,08
10000	10-10000	27,50	28,41	27,73	27,88

Cantidad de monedas	Variabilidad por tipo	Tiempo 1	Tiempo 2	Tiempo 3	Promedio de tiempo
10000	int	27,59	27,31	27,49	28,16
10000	float	27,13	27,52	27,34	28,08

Figura 1: Timpo del algoritmo segun variabilidad

Variabilidad por rango de valores:
Como se observa en la primera tabla, la variabilidad en el rango de valores (1-10, 1-1000, y 10-10000) no tuvo un impacto significativo en los tiempos de resolución del algoritmo. Esto tiene sentido, ya que el rango de valores de las monedas no afecta directamente el tiempo de ejecucion ya que el algoritmo únicamente compara los valores de las monedas y realiza cálculos aritméticos simples para determinar la ganancia máxima. El tiempo de ejecucion del algoritmo aumentaria si agrego mas cantidad de monedas a procesar. El tiempo del algoritmo no se ve afectado por los valores especificos de las monedas.

Variabilidad por tipo de dato:
En la segunda tabla, se compararon los tiempos de ejecución al usar valores de tipo int frente a valores de tipo float. Nuevamente, los resultados muestran que el tipo de dato utilizado no afectó de manera significativa el tiempo de ejecución del algoritmo. Esto puede ser por muchas razones como el tipo de lenguaje utilizado o la arquitectura del computador que se usa para ejecutar el algoritmo, siendo esta ultima la mas relevante.

5. Ejecución de ejemplos y verificación de resultados

5.1. Creacion de un script para generar un set de datos

Hemos creado un script para la generación de ejemplos aletorios de N monedas, que permite guardar los resultados en un txt. Este script utiliza el algoritmo de programacion dinamica que realizamos anteriormente.

```python
import random
import sophia_and_mateo_game as pd

def generate_random_coins(n, min_value=1, max_value=1000):
    return [random.randint(min_value, max_value) for _ in range(n)]


def write_results_to_file(sophia_choices, mateo_choices, filename="resultados.txt"):
    with open(filename, "w") as f:
        min_length = min(len(sophia_choices), len(mateo_choices))

        for i in range(min_length):
            f.write(f"Sophia agarra ({sophia_choices[i]}), Mateo agarra ({mateo_choices[i]})\n")

        f.write(f"Ganancia de Sophia: {sum(sophia_choices)}\n")
        f.write(f"Ganancia de Mateo: {sum(mateo_choices)}")


def write_coins_to_file(random_coins, filename="random_coins.txt"):
    with open(filename, "w") as f:
        for i in range(len(random_coins)):
            f.write(f"{random_coins[i]};")

def generar_ejemplos_de_ejecucion():

    # Ejemplo de uso
    n = 5  # Numero de monedas del arreglo
    random_coins = generate_random_coins(n, 1, 1000)

    dp_sophia = pd.optimal_strategy(random_coins)
    sophia_choices, mateo_choices = pd.reconstruction(random_coins, dp_sophia, len(random_coins))

    write_coins_to_file(random_coins)
    write_results_to_file(sophia_choices, mateo_choices)

generar_ejemplos_de_ejecucion()
```

5.2. Analisis de una instancia de 5 monedas

Hemos ejecutado el script para una instancia de 5 monedas, el cual generó dos archivos: uno que contiene la secuencia inicial de valores en formato .txt y otro que detalla las jugadas realizadas por Sophia y Mateo en cada paso. Además, este segundo archivo incluye el valor óptimo obtenido por Sophia al finalizar la partida.

Secuencia Inicial
`918; 712; 407; 5; 564;`

Estrategia

1. **Sophia** toma `918`, **Mateo** toma `712`.
2. **Sophia** toma `564`, **Mateo** toma `407`.

Resultados

Jugador	**Ganancia**
Sophia	1487
Mateo	1119

Esto son los resultados que genera nuestro script en formato de texto. Ahora fijemonos paso a paso si esto da la solucion exacta

Sabemos que en primera instancia tenemos una matriz donde guardamos los valores con la solucion para una sola moneda.

i\j	j=0	j=1	j=2	j=3	j=4
i=0	918				
i=1		712			
i=2			407		
i=3				5	
i=4					564

Figura 2: Primera instancia de nuestra matriz

Para ahorrarnos varios pasos, supongamos que viene messi y me da todos los resultados con el optimo en cada jugada que va a tener la suma maxima de monedas para cada subarreglo de monedas estando a un paso de obtener mi solucion tal como refleja en la tabla. No hay duda de que messi me va a dar el resultado optimo de los pasos anteriores.

i\j	j=0	j=1	j=2	j=3	j=4
i=0	918	918	1325	1325	?
i=1		712	712	717	1119
i=2			407	407	569
i=3				5	564
i=4					564

Figura 3: Ultima instancia antes de obtener mi solucion final

Apliquemos la ecuacion de recurrecia para obtener nuestra solucion final. Teniamos que nuestra ecuacion de recurrencia es

$$\text{dp\_sophia}[i][j] = \text{máx}\left(\begin{array}{l} \text{monedas}[i] + \begin{cases} \text{dp\_sophia}[i+2][j], & \text{si monedas}[i+1] \geq \text{monedas}[j] \\ \text{dp\_sophia}[i+1][j-1], & \text{en otro caso} \end{cases}, \\ \\ \text{monedas}[j] + \begin{cases} \text{dp\_sophia}[i][j-2], & \text{si monedas}[j-1] \geq \text{monedas}[i] \\ \text{dp\_sophia}[i+1][j-1], & \text{en otro caso} \end{cases} \end{array}\right)$$

Para la primera parte de la ecuacion de recurrencia donde se agarra la moneda[i] (la primera moneda). El primer posible maximo se calcula de la siguiente forma como se muestra en la ilustracion:

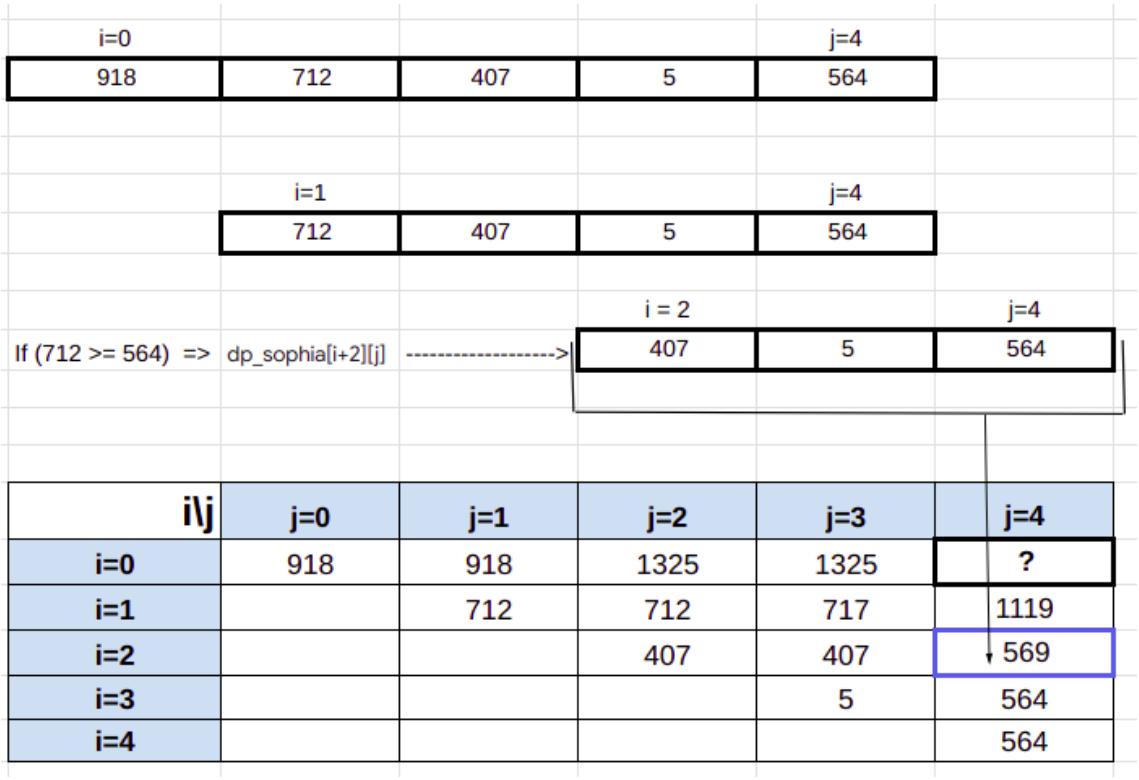

Figura 4: Ilustracion de como se calcula la primera parte de la ecuacion de recurrencia

La ilustracion muestra como calcula el primer posible maximo, estando en el ultimo paso, tengo el arreglo de monedas completo y me quedo con el maximo de dos opciones. En este caso nosotros calculamos el maximo de la primer opcion que basicamente es agarro la primera moneda y me fijo sin esa moneda si el extremo izquierdo es mayor al extremo derecho, Entonces significa que mateo agarrara la moneda del extremo izquierdo, tal cual paso en la ilustracion. Si esto pasa entonces debo quedarme con la suma de la moneda que agarre + la mejor solucion del subarreglo sin la moneda que agarre yo y la que agarro mateo (es lo que esta recuadrado en azul).
Nos queda la primera parte de la ecuacion de recurrencia calculada

$$\text{dp\_sophia}[i][j] = \text{máx}(918 + 569, \text{segundo posible maximo de la ecuación de recurrencia})$$

Analogamente resolvemos para la segundo posible maximo de la ecuacion de recurrencia Que nos da 564+717

$$\text{dp\_sophia}[i][j] = \text{máx}(918 + 569,\ 564 + 717)$$

Sumando nos queda:

$$\text{dp\_sophia}[i][j] = \text{máx}(1487,\ 1281)$$

Entonces la mayor cantidad de monedas que puede obtener Sophia para este juego es de 1487 De esta manera nuestra tabla nos queda completa:

i\j	j=0	j=1	j=2	j=3	j=4
i=0	918	918	1325	1325	1487
i=1		712	712	717	1119
i=2			407	407	569
i=3				5	564
i=4					564

Figura 5: Resultado final de la tabla

Efectivamente este es el resultado que esperábamos y es el mismo que nos genero nuestro script aleatorio

5.3. Explicación de la Reconstrucción

La reconstrucción es necesaria para saber que monedas debe agarrar Sophia. Ya que hasta el momento solo hemos calculado la sumatoria total de monedas para cada problema y solo sabemos que el resultado optimo de Sophia es de 1487 pero no sabemos como lo hace.

El siguiente código implementa una función llamada reconstructivo, que se utiliza para reconstruir las elecciones óptimas realizadas por Sophia y Mateo.

```
def reconstruction(coins, dp_sophia):
    i = 0
    j = len(coins) - 1
    sophia_choices = []
    mateo_choices = []

    while i <= j:
        #caso borde
        if i==j:
            sophia_choices.append((coins[i], PRIMERA)
            break

        choose_first = coins[i] + (dp_sophia[i+2][j] if i+2 <= j and coins[i+1] >= coins[j] else dp_sophia[i+1][j-1]) >= \
                       coins[j] + (dp_sophia[i][j-2] if j-2 >= i and coins[j-1] > coins[i] else dp_sophia[i+1][j-1])

        if choose_first:
            sophia_choices.append((coins[i], PRIMERA)
            if i+2 <= j and coins[i+1] >= coins[j]:
                mateo_choices.append((coins[i+1],PRIMERA)
                i += 2 #agarra mateo salteo [i+1]
            else:
                i += 1
                mateo_choices.append((coins[j], ULTIMA))
                j -= 1 #agarra mateo salteo [j]
        else:
            sophia_choices.append((coins[j], ULTIMA))
            if j-2 >= i and coins[j-1] > coins[i]:
                mateo_choices.append((coins[j-1], ULTIMA))
                j -= 2 #agarra mateo salteo [j-1]
            else:
                mateo_choices.append((coins[i], PRIMERA))
                i += 1 #agarra mateo salteo [i]
                j -= 1

    return sophia_choices, mateo_choices
```

Comienza con los índices i y j apuntando al primer y último índice de la lista monedas, respectivamente. Los arreglos `sophia_choices` y `mateo_choices` se usan para almacenar las monedas que eligen Sophia y Mateo. Utilizando la misma ecuación de recurrencia podemos saber el camino correcto hacia la reconstrucción. La elección se basa en las soluciones pre-calculadas en la matriz `dp_sophia`. Con la ecuación de recurrencia, la matriz de soluciones pre-calculadas y el arreglo original de monedas. Manejando un poco los indices i y j podemos ir reconstruyendo a partir de mi arreglo original de monedas yendo por el camino correcto de elecciones de Sophia para que la reconstrucción represente las mejores elecciones que pudo haber hecho Sophia. Esto se logra de la siguiente manera:

1. Si Sophia elige la moneda i , Mateo tomará la moneda con mayor valor entre la siguiente (i+1) y la última (j), luego se incrementa el indice i y el indice i o j incrementa o decrementa respectivamente, dependiendo lo que haya agarrado Mateo.
2. Si Sophia elige la moneda j, luego se moverán los indices según lo que eligió Mateo.

Este proceso se repite hasta que todos los elementos de monedas han sido procesados.

En las siguientes imágenes se muestran el funcionamiento de la reconstrucción paso a paso

	i\j	j=0	j=1	j=2	j=3	j=4
	i=0	918	918	1325	1325	1487
	i=1		712	712	717	1119
	i=2			407	407	569
	i=3				5	564
	i=4					564
i = 0 , j = 4	0	1	2	3	4	
Monedas	918	712	407	5	564	
Elecciones_Sophia	918	i aumenta 1				
Elecciones_Mateo	712	i aumenta 1				

Figura 6: Primer paso de la reconstrucción

Como dijimos anteriormente se comienza la reconstruccion en el $i = 0$ y en $j = 4$ que corresponde a la primer y ultima moneda, respecitvamente, de mi arreglo original de monedas. Utilizando la ecuacion de recurrencia y la matriz de soluciones agarramos el optimo para el arreglo original de monedas que seria agarrar la moneda 918. Movemos los indices correspondientes a lo que agarro cada uno y reptimos el proceso

	i\j	j=0	j=1	j=2	j=3	j=4
	i=0	918	918	1325	1325	1487
	i=1		712	712	717	1119
	i=2			407	407	569
	i=3				5	564
	i=4					564
i = 2 , j = 4	2	3	4			
Monedas	407	5	564			
Elecciones_Sophia	918	564	j decrementa 1			
Elecciones_Mateo	712	407	i incrementa 1			

Figura 7: Segundo paso de la reconstrucción

Notar que ahora estoy en la instancia $i = 2$ y $j = 4$ que esta recuadrada en rojo en la matriz de soluciones. Otra vez repito el proceso anterior que me tomara la decisión optima para ese conjunto de monedas

	i\j	j=0	j=1	j=2	j=3	j=4
	i=0	918	918	1325	1325	1487
	i=1		712	712	717	1119
	i=2			407	407	569
	i=3				5	564
	i=4					564
i = 3 , j = 3	3					
Monedas	5					
Elecciones_Sophia	918	564	5			
Elecciones_Mateo	712	407				

Figura 8: Ultimo paso de la reconstrucción

Finalmente se me acabaron las monedas y tengo las elecciones de Sophia y Mateo. Siendo las elecciones de Sophia las optimas para ese juego.

6. Corroboración empírica de la complejidad y medición de tiempos

Para realizar la corroboración, se crearon 20 set de datos de monedas, los cuales tienen una cantidad de monedas que varían entre 5 y 15 mil. Los sets son generados con monedas de valor aleatorio, con valores posibles entre 0 y mil. Luego se tomo el tiempo que tarda en obtenerse las monedas de Sophia y Mateo, calcular la matriz de óptimos de subproblemas y la reconstrucción. Los datos obtenidos se reflejan en el gráfico:

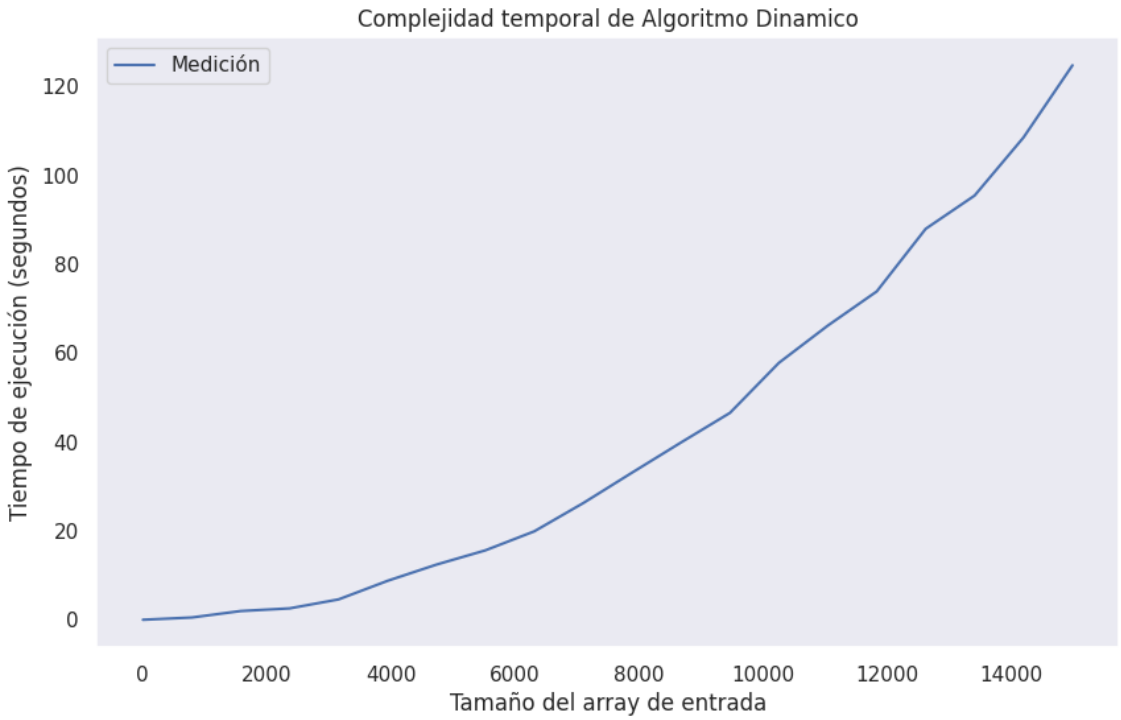

Figura 9: Complejidad temporal

7. Análisis de Complejidad con la Técnica de Cuadrados Mínimos

El método de mínimos cuadrados se utiliza para ajustar una función a un conjunto de datos buscando minimizar la suma de los cuadrados de las diferencias (errores) entre los valores observados (datos) y los valores predichos por la función ajustada. Ademas se utilizo el Coeficiente de Determinación, ya que es independiente de la escala de los datos. Un valor cercano a 1 indica un ajuste fuerte. Se utilizo como curva a comparar una funcion cuadratica:

$$f(x) = c_1 \cdot x^2 + c_2 \tag{1}$$

Se obtuvo:

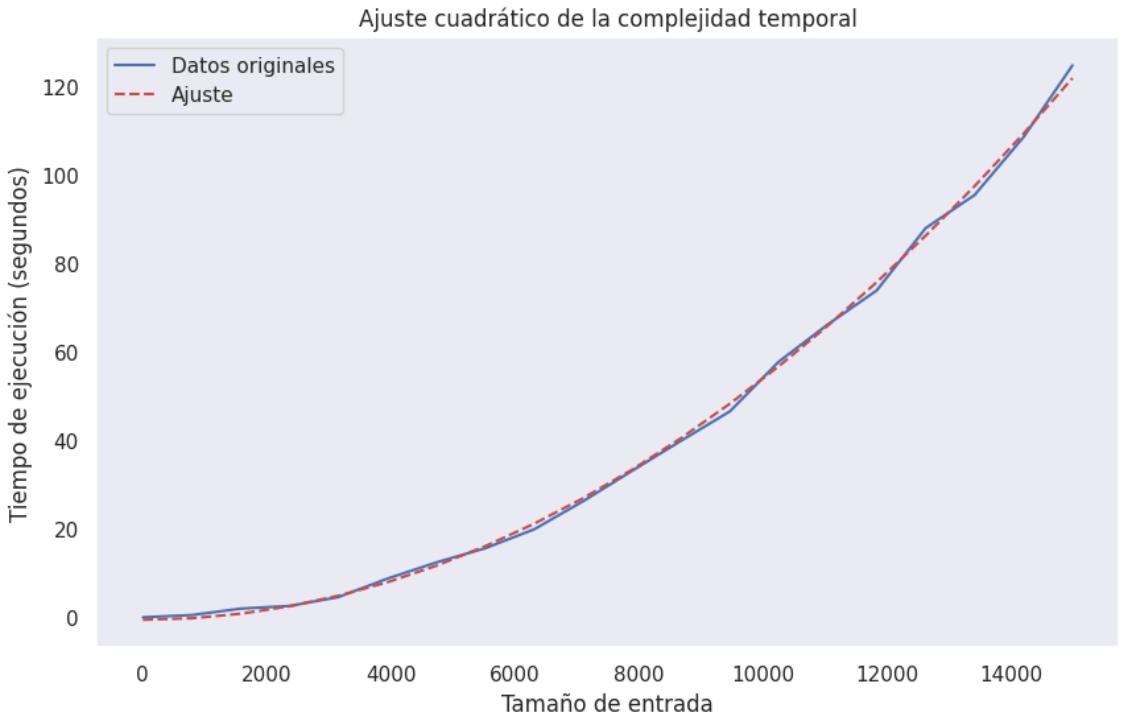

Figura 10: Complejidad temporal

Los coeficientes obtenidos de la ecuación cuadrática, el error cuadrático y coeficiente de determinación son:

```
#Calculo del error cuadratico con un ajuste cuadratico
f = lambda x, c1, c2: c1 * x**2 + c2

c, pcov = sp.optimize.curve_fit(f, x, results)

print(f"c_1 = {c[0]}, c_2 = {c[1]}")
r = np.sum((c[0] * x**2 + c[1] - results)**2)
print(f"Error cuadrático total: {r}")
```

```
c_1 = 5.43907322786802e-07, c_2 = -0.5550192600444329
Error cuadrático total: 29.761926930934557
```

```
[47] ss_res = np.sum((results - (c[0] * x**2 + c[1]))**2)  # Suma de residuos
     ss_tot = np.sum((results - np.mean(results))**2)      # Suma total
     r_squared = 1 - (ss_res / ss_tot)
     print(f"R^2: {r_squared}")
```

```
R^2: 0.9989955532752999
```

Figura 11: Complejidad temporal

Aunque el error cuadrático, valor cercano a 29, se puede decir que es grande, se verifica por medio del gráfico de ajuste y el coeficiente de determinación que es correcto afirmar que el algoritmo tiene complejidad cuadrática.

8. Conclusiones y Observaciones

Un vector para estos casos de programación dinámica no nos sirvió porque perdíamos mucha información de posibles ramas potenciales a mi solución. Ademas en cada jugada se crean exponenciales ramas de posibles jugadas que con un vector no puede capturar toda esa infromacion.

Hicimos dos versiones de este problema. Tuvimos que elegir entre una y elegimos la mas legible. La otra version era aproximadamente era un 20 % mas rapida pero gastaba el doble de memoria. Esa rapidez se vio compensada por mas uso de la memoria. Esto era asi porque ya precalculaba los movimientos de Mateo y los guardaba en una matriz. Cuando necesitaba saber que moneda agarro mateo simplemente iba a la matriz a buscar el indice del movimiento de mateo en tal paso.

Cuando hicimos la reconstruccion pensamos en agarrar solo el maximo de mi arreglo precalculado como se hacia en el problema del laberinto. Nos dimos cuenta que estabamos ignorando cómo esa elección afectaba las posibilidades en turnos posteriores y la reconstrucción se iba por un camino no optimo donde ya no se puede retroceder. (Borrar)

El problema de Sophia y Mateo es mas complejo de lo que parece. Ya que se generan nuevas permutaciones que afectan a los subproblemas a medida que se agranda el arreglo de monedas. Este problema se podria pensar como un caso especifico del problema ya conocido Max Subset Sum donde queremos obtener la suma maxima de un sub-conjunto de elementos. Este sub conjunto puede ser un grafo comun con conexiones variadas. En el caso de Sophia nuestro grafo se podria dibujar como la fila de monedas donde cada vertice es una moneda y las conexiones entre las aristas las adyacencias entre las monedas. Tendremos tambien un vertice especial que podriamos conectarlo con aristas a ambos extremos de la fila donde representara la eleccion que puede tomar Sophia.

La diferencia es que: En "Max Subset Sum", se deben incluir o excluir elementos.
En el problema de Sophia, las elecciones de monedas están limitadas a los extremos de la fila. Tambien tenemos la restriccion de que la siguiente moneda no la podremos usar (sabiendo cual va a ser esta moneda). El caso se Sophia es un caso especifico de Max Subset Sum. Lo cual podria reducirse polinomialmente a:

$$\text{Juego Sophia} \leq_p \text{Max Subset Sum}$$