.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

Benjamin Castellano	Joaquin Eduardo	Lionel Gabriel
Bogdan	Embon	Maydana Gonzalez
111519	111292	106512

1. Introduccion

Sophia y Mateo son hermano que aún son niños pero ya tienen la madurez necesaria para querer competir entre ambos, quieren jugar un juego explicado por su padre: 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).

Lo que buscamos en este trabajo práctico es encontrar 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.

Primero que nada, para un algoritmo de programación dinámica, debemos buscar la **ecuación de recurrencia**. ¿Cómo podemos pensarla? Tenemos que hacer que gane Sophia a toda costa, o en caso contario, que pueda tener el máximo valor acumulable posible porque hay que tener en cuenta que Mateo también quiere ganar y va a hacer todo lo posible siempre (elegir la moneda más grande entre la primera y la última cada vez que le toque jugar). Si sólo pensamos en elegir la moneda de mayor valor en el turno de Sophia, nos puede pasar, que la moneda que reemplace la misma sea de un valor muy alto, por ejemplo, Sophia elegió la moneda '7' teniendo el arreglo [7, **100**, 1, 5], en este caso le estamos dejando la moneda de mayor valor a Mateo.

Para cada turno de Sophia, la elección será entre la primer moneda o la última pero, para ambas, sumada la máxima sumatoria del subarreglo que las acompaña, es decir, el valor de la moneda + máximo del subarreglo. ¿Qué quiere decir ésto? Que tendremos en cuenta las monedas que le dejamos a Mateo para elegir. Teniendo en cuenta cada arreglo con una posición *inicio* y fin, la ecuación de recurrencia quedaría:

opción_izquierda = Valor primer moneda + Máximo del subarreglo correspondiente
opción_derecha = Valor última moneda + Máximo del subarreglo correspondiente

Opt[ini][fin] = max(opción_izquierda, opción_derecha)

2. Algoritmo

2.1. Codigo

```
1  def juego_sophia_mateo(monedas):
2      cant_monedas = len(monedas)
3
4      matriz = [[0] * cant_monedas for _ in range(cant_monedas)]
5
6      # Caso base: cuando solo hay una moneda (Sophia toma la unica moneda)
7      for i in range(cant_monedas):
8          matriz[i][i] = monedas[i]
9
10     # Vamos llenando la tabla desde casos menores.
11     for longitud in range(2 + cant_monedas % 2, cant_monedas + 1, 2):
12         for ini in range(cant_monedas - longitud + 1):
13             fin = ini + longitud - 1
14
15             # Sophia toma la moneda en el extremo izquierdo
16             opcion_izquierda = monedas[ini]
17             # Sumamos valores dependiendo de la decision de Mateo
18             if monedas[ini + 1] > monedas[fin]:
19                 opcion_izquierda += matriz[ini + 2][fin] if ini + 2 <= fin else 0
20             else:
21                 opcion_izquierda += matriz[ini + 1][fin - 1] if ini + 1 <= fin - 1
       else 0
22
23             # Mismo procedimiemto pero para la moneda del extremo derecho
24             opcion_derecha = monedas[fin]
25
26             if monedas[ini] > monedas[fin - 1]:
27                 opcion_derecha += matriz[ini + 1][fin - 1] if ini + 1 <= fin - 1
       else 0
28             else:
29                 opcion_derecha += matriz[ini][fin - 2] if ini <= fin - 2 else 0
30
31             # Guardar el valor maximo en matriz
32             matriz[ini][fin] = max(opcion_izquierda, opcion_derecha)
33
34     return rearmar_solucion(matriz, monedas)
```

Nuestro algoritmo de programación dinámica está planteado de manera "bottom up", guardando primero los resultados conseguidos de los subproblemas anteriores, para ir utilizandolos luego para construir el problema principal.

Para trabajar con las monedas, vamos a tomar un arreglo con estas monedas, en el orden con el que se nos las dan.

Para la parte de "memoization", caracteristica de este tipo de algoritmo, las sumatorias maximas de monedas para cada subproblema, seran guardados en una matriz, creada en la linea 4 del codigo anterior. Esta decisión se da por las variables del problema, que en este problema identificamos dos. Por un lado "*ini*", como la posicion relativa de la moneda del extremo izquierdo, y por el otro "fin", como la posición relativa de la moneda del extremo derecho.

Para llenar esta matriz con los valores que nos interesan, primero vamos al caso base, el cual seria un arreglo con una sola moneda (ini == fin), simplemente llenamos esas posiciones con la única moneda del arreglo. Notar que tambien esta cubierto el caso de un arreglo vacio (la matriz tendría largo cero, y los siguientes pasos que recorren esta matriz no harian nada, el resultado seria 0).

Luego, vamos a ir llenando la matriz incrementando el tamaño de cada subproblema, cada vez que hayamos calculado su sumatoria maxima de monedas (Ciclo que comienza en linea 11). Estos subproblemas los vamos a resolver de acuerdo a la siguiente ecuación de recurrencia:

$$\text{matriz[ini][fin]} = \max(\text{opcion\_izquierda}, \text{opcion\_derecha})$$

Donde:

- matriz[ini][fin] es la sumatoria maxima que Sophia puede tener con un las monedas desde la posicion ini hasta la posicion fin.
- opcion_izquierda es el valor que obtiene Sophia si elige la moneda en el extremo izquierdo(posicion ini). Este valor se calcula como: opcion_izquierda = $monedas[ini]$+$ganancia\_segun\_mateo$. Donde $ganancia\_segun\_mateo$ es: $matriz[ini+2][fin]$ (si $ini+2 <= fin$)si mateo elige la moneda en $ini+1$, o $matriz[ini+1][fin-1]$ (si $ini+1 <= fin-1$) si mateo elige la moneda fin.
- opcion_derecha es el valor que obtiene Sophia si elige la moneda en el extremo derecho(posicion fin). Este valor se calcula como: opcion_izquierda = monedas[ini] + $ganancia\_segun\_mateo$. Donde $ganancia\_segun\_mateo$ es: $matriz[ini+1][fin-1]$ (si $ini+1 <= fin-1$) si mateo elige la moneda en ini, o $matriz[ini][fin-2]$ (si $ini <= fin-2$) si mateo elige la moneda fin

Entonces, la ecuacion de recurrencia completa seria:

matriz[ini][fin] = max(monedas[ini] + min(matriz[ini + 2][fin], matriz[ini + 1][fin - 1]),
monedas[fin] + min(matriz[ini + 1][fin - 1], matriz[ini][fin - 2]))

Finalmente, devolvemos la función rearmar_solucion() (El codigo aparece más abajo), que lo que hace es devolver dos arreglos, uno que tiene las posiciones de las monedas que Sophia tiene que agarrar, y otro que tiene las posiciones de las monedas que Mateo tiene que agarrar(las posiciones son respecto de el arreglo de monedas principal)

```
    def rearmar_solucion(matriz, monedas):
    solucion_sophia = []
    solucion_mateo = []
    cant_monedas = len(monedas)
    ini = 0
    fin = cant_monedas - 1
    valor_maximo = matriz[ini][fin]
    while valor_maximo > 0:
        if ini == fin:
            solucion_sophia.append(ini)
            break
        siguiente_ini, siguiente_fin = siguientes_indices_eligiendo_mateo(ini + 1,
    fin, monedas)
        if valor_maximo == monedas[ini] + matriz[siguiente_ini][siguiente_fin]:
            solucion_sophia.append(ini)
            valor_maximo -= monedas[ini]
            if siguiente_ini == ini + 1:
                solucion_mateo.append(fin)
            else:
                solucion_mateo.append(ini + 1)
        else:
            siguiente_ini, siguiente_fin = siguientes_indices_eligiendo_mateo(ini,
    fin - 1, monedas)
            solucion_sophia.append(fin)
            valor_maximo -= monedas[fin]
            if siguiente_ini == ini:
                solucion_mateo.append(fin - 1)
            else:
                solucion_mateo.append(ini)
        ini = siguiente_ini
        fin = siguiente_fin
    return solucion_sophia, solucion_mateo

def siguientes_indices_eligiendo_mateo(ini, fin, monedas):
    if monedas[fin] >= monedas[ini]:
        return ini, fin - 1
    return ini + 1, fin
```

Mediante un ciclo, y utilizando la ecuacion de recurrencia a la inversa por cada iteración, podemos rearmar el camino y conocer bien cuales son los pasos que Sophia tiene que seguir para conseguir el maximo valor acumulado.

Nos parece importante aclarar explicitamente, que a partir de todas las caracteristicas que este algoritmo presenta, y teniendo en cuenta la forma en la que está escrito, este algoritmo termina resolviendo el problema puramente por programación dinámica, cumpliendo con el requisito que la consigna nos exigía.

2.2. Optimalidad

Para demostrar que nuestro algoritmo es óptimo, primero vamos a hablar un poco sobre la forma de nuestro problema.

Un problema tiene una subestructura óptima, si la solución óptima del problema principal, se puede construir a partir de soluciones óptimas de los subproblemas que derivan del él. En este caso, el problema de las monedas tiene subestructura óptima. Estro es asi ya que para maximizar la ganancia de Sophia al tomar monedas entre ini y fin, su elección se reduce a una subestructua óptima; una subsecuencia de monedas de menor tamaño. Esto sinifica que la solucion óptima para un intervalo de monedas depende de las soluciones óptimas de las subsecuencias que se generan luego de las decisiones de Sophia y Mateo.

Teniendo en cuenta lo anterior, vamos a probar la optimalidad del agloritmo usando inducción:

Base de la Inducción

Cuando tenemos una sola moneda en el intervalo (ini == fin), Sophia siempre tomara esa moneda, ya que es la decision que maximiza su valor acumulado de monedas. Este caso es óptimo porque no hay decisiones adicionales, ni consideraciones para optimizar. De esta manera, el algoritmo es óptimo para intervalos de longitud 1.

Hipótesis de la Inducción

Supongamos que el algoritmo es óptimo para cualquier intervalo de longitud $k <= n - 1$, (con n siendo la cantidad de monedas), es decir, que matriz[ini][fin] calcula correctamente la ganancia maxima de Sophia cuando el intervalo ini a fin tiene longitud k.

Paso Inductivo

Vamos a probar que el algoritmo es óptimo para un intervalo de longitud k + 1.

Para el intervalo ini a fin, Sophia tiene dos opciones:

- Tomar la moneda en ini, para que luego Mateo tome la moneda más grande entre $ini + 1$ y fin, dejando a Sophia con un subintervalo que es más pequeño que tiene la solución óptima almacenada en la matriz.
- Tomar la moneda en fin, para que luego Mateo tome la moneda más grande entre ini y $fin - 1$, dejando nuevamente a Sophia con un subintervalo más mequeño.

Las dos opciones se pueden expresar de acuerdo a la ecuación de recurrencia de nuestro problema, recordemos:

matriz[ini][fin] = max(monedas[ini] + min(matriz[ini + 2][fin], matriz[ini + 1][fin - 1]),
monedas[fin] + min(matriz[ini + 1][fin - 1], matriz[ini][fin - 2]))

Por la hipotesis de la inducción, sabemos que las subsoluciones matriz[ini + 1][j - 1], matriz[i + 2][fin], matriz[ini][fin - 2] son óptimas, ya que se resuelven de intervalos más pequeños(k - 1) correctamente.

Dado que Sophia elige la opción que maximiza su ganancia y que cada subsolución es óptima(hipotesis), podemos concluir que matriz[ini][fin] es la solución óptima para el intervalo ini a fin.

Ejemplo de Ejecución

Para concluir esta explicación, vamos a hacer un seguimiento de este algoritmo para un arreglo de 5 elementos. Vamos a considerar el arreglo [5, 10, 2, 6, 8].

Crear la matriz

Primero creamos una matriz de 5x5, y la llenamos de ceros.

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Caso Base

Para los subintervalos que son de longitud 1, es decir cuando $ini = fin$, los cuales corresponden a la diagonal de la matriz, simplemente agregamos el valor de la moneda de esa posicion en la matriz. En este caso, si $ini = fin = 0$, se elige la moneda de la primer posicion, el 5 en este caso, luego para $ini = fin = 1$ será el 10, luego el 2, y asi hasta el final.

$$\begin{bmatrix} 5 & 0 & 0 & 0 & 0 \\ 0 & 10 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 6 & 0 \\ 0 & 0 & 0 & 0 & 8 \end{bmatrix}$$

Llenar la matriz incrementando el tamaño de los subintervalos

Comenzamos con los subintervalos de longitud 2. Siempre ini toma valores hasta $n - longitud + 1$(en este caso el valor es 3, que es lo maximo que el inicio puede ir para que el subintervalo de longitud 2), y fin, por cada valor de ini toma el valor $ini + longitud - 1$. Los valores iniciales ahora entonces seran $ini = 0$, $fin = 0 + 2 - 1 = 1$. Con estos definidos, empezamos a aplicar la ecuacion de recurrencia:

$$\text{opcion\_izquierda} = monedas[0] + min(matriz[0+2][1], matriz[0+1][1-1])$$
$$\text{opcion\_izquierda} = 5 + min(0, 0) \text{ (son 0 porque } ini + 2 > fin \text{ y } ini + 1 > fin - 1)$$
$$\text{opcion\_izquierda} = 5$$

Ahora para el otro extremo:

$$\text{opcion\_derecha} = monedas[1] + min(matriz[0+1][1-1], matriz[0][1-2])$$
$$\text{opcion\_derecha} = 10 + min(0, 0)$$
$$\text{opcion\_derecha} = 10$$

Por último, guardamos el maximo valor en matriz:

$$\text{matriz[ini][fin]} = \max(\text{opcion\_izquierda, opcion\_derecha})$$
$$\text{matriz[0][1]} = \max(5,\ 10)$$
$$\text{matriz[0][1]} = 10$$

Entonces la matriz nos queda con un nuevo valor

$$\begin{bmatrix} 5 & 10 & 0 & 0 & 0 \\ 0 & 10 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 6 & 0 \\ 0 & 0 & 0 & 0 & 8 \end{bmatrix}$$

Los proximos valores que tomaran ini y fin hasta cubrir todos los subintervalos de longitud 2 serán, 1 y 2, 2 y 3, 3 y 4 respectivamente. Siguiendo el mismo procedimiento que antes, pero de acuerdo a estos valores, llegaremos a que:

$$\text{matriz[1][2]} = 10$$
$$\text{matriz[2][3]} = 6$$
$$\text{matriz[3][4]} = 8$$

Quedandonos entonces la matriz:

$$\begin{bmatrix} 5 & 10 & 0 & 0 & 0 \\ 0 & 10 & 10 & 0 & 0 \\ 0 & 0 & 2 & 6 & 0 \\ 0 & 0 & 0 & 6 & 8 \\ 0 & 0 & 0 & 0 & 8 \end{bmatrix}$$

Estas son las soluciones optimas para cada subintervalo de longitud 2, una vez terminado con estos, pasamos a los de longitud 3. Para este caso, ini va a comenzar en 0, y $fin = 0 + 3 - 1 = 2$. Entonces comenzamos utilizando la ecuación de recurrencia:

$$\text{opcion\_izquierda} = monedas[0] + min(matriz[0+2][2], matriz[0+1][2-1])$$
$$\text{opcion\_izquierda} = 5 + min(2, 10)$$
$$\text{opcion\_izquierda} = 7$$

Ahora para el otro extremo:

$$\text{opcion\_derecha} = monedas[2] + min(matriz[0+1][2-1], matriz[0][2-2])$$
$$\text{opcion\_derecha} = 2 + min(10, 5)$$
$$\text{opcion\_derecha} = 7$$

Por último, guardamos el maximo valor en matriz:

$$\text{matriz[ini][fin]} = \max(\text{opcion\_izquierda, opcion\_derecha})$$
$$\text{matriz[0][2]} = \max(7,\ 7)$$
$$\text{matriz[0][2]} = 7$$

Entonces la matriz nos queda con un nuevo valor

$$\begin{bmatrix} 5 & 10 & 7 & 0 & 0 \\ 0 & 10 & 10 & 0 & 0 \\ 0 & 0 & 2 & 6 & 0 \\ 0 & 0 & 0 & 6 & 8 \\ 0 & 0 & 0 & 0 & 8 \end{bmatrix}$$

Los proximos valores que tomaran ini y fin hasta cubrir todos los subintervalos de longitud 3 serán, 1 y 3, 2 y 4 respectivamente. Siguiendo el mismo procedimiento que antes, pero de acuerdo a estos valores, llegaremos a que:

$$\text{matriz}[1][3] = 12$$
$$\text{matriz}[2][4] = 10$$

Quedandonos entonces la matriz:

$$\begin{bmatrix} 5 & 10 & 7 & 0 & 0 \\ 0 & 10 & 10 & 12 & 0 \\ 0 & 0 & 2 & 6 & 10 \\ 0 & 0 & 0 & 6 & 8 \\ 0 & 0 & 0 & 0 & 8 \end{bmatrix}$$

Ya entendimos la dinámica de este algoritmo, sabemos que ahora vamos a resolver los subproblemas de longitud 4, para finalmente, conseguir resolver el problema principal, de longitud 5, utilizando los valores utilizados anteriormente. Para los subarreglos de longitud 4, los valores de ini y fin seran, 0 y 3, 1 y 4 respectivamente. Aplicando la ecuación de recurrencia, llegamos a los siguientes resultados:

$$\text{matriz}[0][3] = 16$$
$$\text{matriz}[1][4] = 16$$

Quedandonos entonces la matriz:

$$\begin{bmatrix} 5 & 10 & 7 & 16 & 0 \\ 0 & 10 & 10 & 12 & 16 \\ 0 & 0 & 2 & 6 & 10 \\ 0 & 0 & 0 & 6 & 8 \\ 0 & 0 & 0 & 0 & 8 \end{bmatrix}$$

Finalmente, llegamos al paso más importante, tenemos todos los resultados necesarios para resolver el problema principal. Ahora longitud vale 5, y los valores de ini y fin van a ser 0 y 4, apliquemos la ecuacion de recurrencia:

$$\text{opcion\_izquierda} = monedas[0] + min(matriz[0+2][4], matriz[0+1][4-1])$$
$$\text{opcion\_izquierda} = 5 + min(10, 12)$$
$$\text{opcion\_izquierda} = 15$$

Ahora para el otro extremo:

$$\text{opcion\_derecha} = monedas[4] + min(matriz[0+1][4-1], matriz[0][4-2])$$
$$\text{opcion\_derecha} = 8 + min(12, 7)$$
$$\text{opcion\_derecha} = 15$$

Por último, guardamos el maximo valor en matriz:

$$\text{matriz[ini][fin]} = \max(\text{opcion\_izquierda, opcion\_derecha})$$
$$\text{matriz[0][4]} = \max(15, 15)$$
$$\text{matriz[0][4]} = 15$$

Quedandonos entonces la matriz:

$$\begin{bmatrix} 5 & 10 & 7 & 16 & 15 \\ 0 & 10 & 10 & 12 & 16 \\ 0 & 0 & 2 & 6 & 10 \\ 0 & 0 & 0 & 6 & 8 \\ 0 & 0 & 0 & 0 & 8 \end{bmatrix}$$

La posicion [0][4] de la matriz es el valor máximo acumulado para Sophia en este problema, ahora con la matriz completa, vamos a podes rearmar la solucion.

Para rearmar soluciones, inicializamos dos arreglos vacios, uno para Sophia y otro para Mateo, inicializamos dos variables: ini, fin = 0, cantidad_total_monedas - 1, y por último nos guardamos el valor maximo acumulado en una variable $valor\_maximo$. Para este caso $ini = 0$, $fin = 4$ y $valor\_maximo = 15$. Vamos a seguir el algoritmo.

Como $ini! = fin$ y $valor\_maximo > 0$ vamos a descubrir de donde venimos. Luego de comparar, vamos a ver que habiamos elegido la moneda ini (numero 5). Entonces le agregamos la posicion de esa moneda a Sophia, le restamos el valor de la moneda a valor_maximo, le agregamos la posicion de la moneda correspondiente para Mateo (El 10 en este caso), y actualizamos los indices para la proxima iteracion, y seguimos hasta llegar alguna condicion de corte.

Asi los arreglos van a ser $Sophia = [0, 4, 2]$, $Mateo = [1, 3]$.

Como podemos ver, en este ejemplo Sophia no va a ganar el juego, pero el objetivo del algoritmo era encontrar las monedas que aseguran que Sophia consiga el mayor valor acumulado, lo cual es correcto en este caso.

2.3. Complejidad

Análisis de la complejidad

Para analizar la complejidad del algoritmo, se debe tener en cuenta el largo de la entrada inicial, el cual guarda una relación con el largo del array de monedas, por lo cual, diremos que nuestra complejidad, depende de **'n'**, siendo **'n'** la cantidad de monedas en el array; luego deberemos analizar las diferentes partes del código. Al principio del código se ven inicializaciones de variables, entre las que está incluida una matriz de n*n, esto conlleva una complejidad inicial de O(n*n). Al final del algoritmo, se reconstruye la solución, lo cual lleva a recorrer todas las monedas que escogió Sophia y las que escogió Mateo, lo que nos da una complejidad de O(n) para esta parte. Por último, tenemos un for que recorrerá desde 0 hasta n (sin incluir), rellenando la matriz inicial con los casos base, aportando una complejidad temporal de O(n); como parte central del código tenemos dos bucles for, el más externo de estos se mueve entre 2 o 3(dependiendo de si la cantidad de monedas es par o impar) y de dos en dos, hasta n(incluido), este bucle nos determinará el valor de la variable "longitud", la cual representará el largo del subarray a considerar en el bucle. El for más interno va desde 0, de uno en uno, hasta n - longitud + 1, este for le dará el valor a la variable "ini", la cual representará el índice de comienzo del subarray a considerar con respecto al array de monedas original. Mirando dentro de estos for anidados, descubrimos que solo se realizarán finitas operaciones, con lo que, dentro de los for, la complejidad es de O(1). Ahora, si consideramos que el array original tiene una longitud par, entonces veremos que para la primera pasada del bucle más externo, la longitud es de 2, por lo que el bucle más interno se moverá entre 0 y n - 2 + 1, es decir, entre 0 y n - 1, con lo que, aportando a la complejidad un O(n-1) o, mejor analizado, O(n), para la segunda pasada hay que considerar que el for se mueve de dos en dos, con lo que tendremos que la variable "longitud", valdrá 4, con lo que el bucle interno se moverá entre 0 y n - 4 + 1, es decir, entre 0 y n - 3. Siguiendo analizando, obtenemos que la cantidad de operaciones será proporcional a:

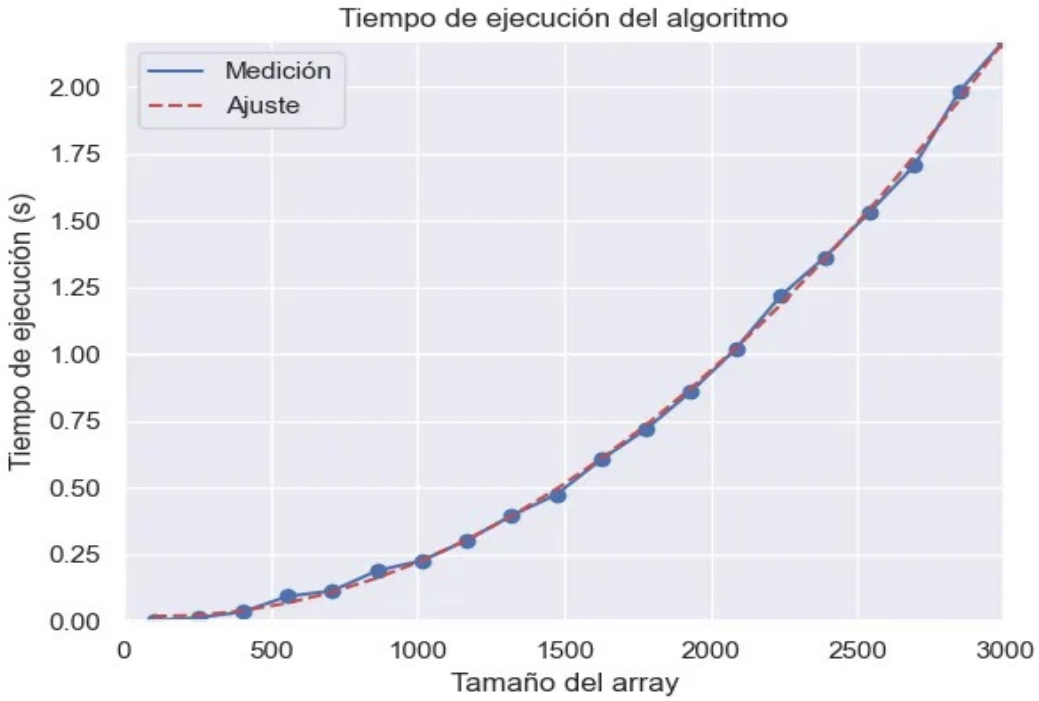
- $\sum_{i=1}^{(n-1)/2} n - (2i+1) + 1 = n * \frac{n-1}{2} - 2 * \frac{n-1}{2} * \frac{\frac{(n-1)}{2}+1}{2} + 2 * \frac{n-1}{2} = \frac{n^2}{2} - \frac{n}{2} - \frac{1}{4} * [n^2 - 2 * n + 1] - \frac{n-1}{2} + n - 1 = \frac{n^2}{4} - \frac{1}{4} - \frac{n}{2} + \frac{1}{2} + n - 1 = \frac{n^2}{4} + \frac{n}{2} - \frac{3}{4}$

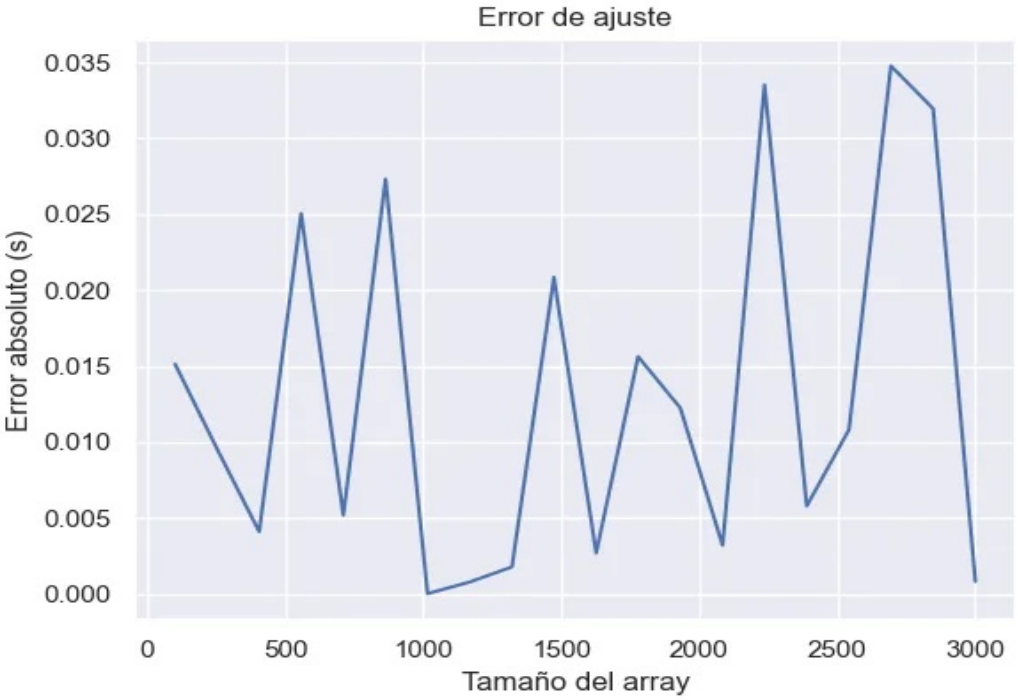
Con esto podemos deducir que la complejidad será O(n^2) en los bucles anidados. Finalmente, tenemos que el algoritmo tendrá una complejidad de O(n^2) + O(n) + O(n^2) + O(n) = O(n^2) Análogamente se puede analizar para el caso impar y se obtendrá la misma complejidad.

Complejidad teórica vs complejidad empírica

Por lo anterior, se intentará comparar los tiempos de ejecución con una función del tipo $y = a * x^2 + b * x + c$

Al hacerlo, se obtuvo lo siguiente:





En el primer gráfico se observa en forma de puntos azules los tiempos medidos del algoritmo para arrays de monedas con valores aleatorios y con la longitud especificada en el eje x. Mientras que en el segundo gráfico se observa la diferencia de tiempos que hay entre el tiempo obtenido para un determinado conjunto de monedas y el tiempo que se debería obtener según el ajuste propuesto como función que más se ajusta a los datos. Vemos que en el caso más lejano no se acerca siquiera

a una diferencia de 0,1, lo que sugiere que el modelo teórico ajusta con una exactitud muy alta a los resultados empíricos.

Relación valores/complejidad

Para analizar la relación de la complejidad con los valores de las monedas, se realizaron comparaciones con diferentes arrays de igual longitud, que poseían valores aleatorios, y comparaciones entre arrays de igual longitud, que poseían todos sus valores iguales salvo el ultimo. Mientras que en el eje y de ambos se refleja el tiempo que tardo el algoritmo en resolver el problema para los diversos arrays, en el eje x la situación es distinta para el primer caso y el segundo caso. En el caso de los valores aleatorios, se decidió que el eje x represente el promedio de los valores contenidos en los diferentes arrays. Mientras que en el segundo caso se usó, como valor del eje x, el valor que predominaba en el array. Luego de analizar los tiempos en relación a los valores en cada caso, se planteó una aproximación lineal como ajuste, de esta manera, si los valores de las monedas provocarán un cambio en el tiempo, se debería ver una clara pendiente en la recta de ajuste. De esta manera se obtuvo:

En la primera imagen se ve que se presenta una pequeña pendiente en el ajuste lineal, con una magnitud de 10^{-7}, esta pendiente se puede deber a las diferentes cargas que pudo poseer el procesador en el momento de ejecución y a la aleatoriedad de las monedas, aleatoriedad que forzó al branch predictor a una resolución cada vez más lenta. Sin embargo, se puede apreciar que es una pendiente cercana al "0", con lo cual es una aproximación cercana a un tiempo constante por cada valor promedio posible. Mientras que en el segundo gráfico se puede observar que el valor de la pendiente obtenida es incluso más bajo, acercándose a valores de incluso 10^{-8}, mostrando así una recta prácticamente paralela con el eje x, acentuándose el hecho de que el valor de las monedas no afecta al tiempo de ejecución, pero si lo puede hacer la aleatoriedad de los mismos.

3. Conclusiones

Teniendo en cuenta que el objetivo no es sólo que gane Sophia, sino también, que obtenga el máximo valor acumulado posible y que en consecuencia a eso pueda ganar o no, el algoritmo propuesto es óptimo, ya que resuelve el problema utilizando una estructura que garantiza soluciones óptimas a los subproblemas más pequeños, construyendo así la solución al problema completo. Además, la complejidad $O(n^2)$ es adecuada para resolver instancias de tamaño moderado del problema, como se evidencia en las mediciones realizadas.