Informe de Teoría de Algoritmos

110043 Santiago Alabes[1], 109574 Thomas Santillan[2], and 107235 Ricardo Wang[3]

[1]Facultad de Ingeniería, UBA
[2]Facultad de Ingeniería, UBA
[3]Facultad de Ingeniería, UBA

02 Diciembre 2024

1. Primera parte

1.1. Análisis del problema

1-Hacer un análisis del problema, y proponer un algoritmo greedy que obtenga la solución óptima al problema planteado: Dados los n valores de todas las monedas, indicar qué monedas debe ir eligiendo Sophia para si misma y para Mateo, de tal forma que se asegure de ganar siempre. Considerar que Sophia siempre comienza (para sí misma).

Respuesta: El algoritmo greedy que nosotros proponemos consiste básicamente en que Sophia, en su turno, elija la solución más optima, maximizando su ganancia de puntos. Esto lo haría eligiendo la moneda de mayor valor. Luego en el turno de Mateo, Sophia va a elegir la moneda de menor valor. Esto hace que sea la solución local mas optima ya que minimiza la ganancia de Mateo, produciendo la maximización de las ganancias de Sophia. Esto le aseguraría a Sophia la victoria, ya que ella esta maximizando sus ganancias tanto en su turno como en el de Mateo mientras minimiza las de Mateo.

1.2. Demostración

2-Demostrar que el algoritmo planteado obtiene siempre la solución óptima (desestimando el caso de una cantidad par de monedas de mismo valor, en cuyo caso siempre sería empate más allá de la estrategia de Sophia).

Respuesta: Para demostrar esto vamos a realizarlo mediante inversiones, esto consiste en demostrar que las puntuaciones de Sophia sin inversiones tienen la misma puntuación máxima, y por lo tano son igual de óptimos. También debemos demostrar que existe una solución sin inversiones que es optima. Para demostrar lo primero, decimos que una solución sin inversiones es aquella en la que sophia, en cada turno, elije la moneda de mayor valor. En cada turno sophia elije la moneda de mayor valor, Mateo que sigue siempre la misma estrategia elije la moneda de menor valor . Las monedas que puede elegir Sophia en cada turno depende de los valores que hayan disponibles en ese momento en los extremos, por lo tanto, el orden en el que se seleccionan las monedas no afecta la ganancia acumulada de sophia, ya que las decisiones están determinadas por los extremos nada mas. Esto implica que todas las soluciones sin inversiones nos dan el mismo conjunto de monedas seleccionadas por Sophia, y a su vez la misma ganancia. Luego debemos demostrar que existe una solución sin inversiones optima, en la solución sin inversiones Sophie siempre asegura que se ganancia instantánea seala mayor posible y las decisiones de mateo no afectan el valor total de las monedas que sophia puede obtener. Entonces toda solución sin inversiones es optima. En caso de realizar una solución con inversiones, es decir que Sophia elija una moneda de menor valor en su turno, su ganancia instantánea se minimiza, y esto puede generar que mateo obtenga monedas de mayor valor y que sophia no alcance la ganancia máxima que podría haber obtenido siguiendo una solución sin inversiones. Podemos decir que nuestra solución es optima y que las soluciones con inversiones es estrictamente peor que una solución sin inversiones.

1.3. Implementación de la solución

3-Escribir el algoritmo planteado. Describir y justificar la complejidad de dicho algoritmo. Analizar si (y cómo) afecta la variabilidad de los valores de las diferentes monedas a los tiempos del algoritmo planteado.

```
def elegir_monedas(monedas):
    sophia_puntuacion = 0
    mateo_puntuacion = 0

    primer_moneda = 0
    ultima_moneda = len(monedas)-1

    pasos = []

    while primer_moneda <= ultima_moneda:
        #primero elije Sophia
        if monedas[primer_moneda] > monedas[ultima_moneda]:
            sophia_puntuacion += monedas[primer_moneda]
            primer_moneda += 1
            pasos.append(" Primera moneda para Sophia")
        else:
            sophia_puntuacion += monedas[ultima_moneda]
            ultima_moneda -= 1
            pasos.append("  ltima  moneda para Sophia")

        #Elije sophia para mateo en caso de que siga
            habiendo monedas
        if primer_moneda <= ultima_moneda:
            if monedas[primer_moneda] <
                monedas[ultima_moneda]:
                mateo_puntuacion += monedas[primer_moneda]
                primer_moneda += 1
                pasos.append(" Primera moneda para Mateo")
            else:
                mateo_puntuacion += monedas[ultima_moneda]
                ultima_moneda -= 1
                pasos.append("  ltima  moneda para Mateo")

    return sophia_puntuacion,pasos
```

En este caso por cada iteración quitamos 2 monedas, a menos que quede 1 sola, por lo que iteramos sobre el arreglo $\frac{n}{2}$veces, siendo n la cantidad de monedas dentro de la iteración las operaciones son $O(1)$. La complejidad es

$$O(n/2) = 1/2.O(n) = O(n)$$

Respuesta: La variabilidad en los valores de las monedas no afecta el tiempo de ejecución del algoritmo porque este depende únicamente de la cantidad nn de monedas y no de sus valores. Todas las decisiones son tomadas mediante comparaciones locales en $O(1)$, independientemente de las magnitudes absolutas de los valores.

1.4. Análisis de variabilidad

4-Analizar si (y cómo) afecta la variabilidad de los valores de las diferentes monedas a la optimalidad del algoritmo planteado. Respuesta: La variabilidad no afecta a la optimalidad del algoritmo, lo que si puede variar es la ventaja que consiga Sophia respecto de Mateo. Esto quiere decir que la variabilidad de los valores si afecta al resultado del juego, pero sigue siendo la opción mas optima. Si hay una alta variabilidad, es decir las diferencia de valores entre monedas es muy grande, Sophia se vería fuertemente beneficiada ya que la diferencia de puntuación final será muy elevada, en caso de no tener una alta variabilidad, la diferencia de puntuación será mucho mas baja. Esto sucede por ejemplo si tenemos monedas de este estilo: $[1, 100, 2, 100]$, en este caso Sophia sumara 200 puntos y mateo 3, pero si la variación fuera menor: $[1, 5, 2, 3]$ sophia sumaria 8 puntos y mateo 3, aquí se puede apreciar que no hay tanta diferencia de puntaje mientras que en el anterior si.

1.5. Ejemplos

5.Realizar ejemplos de ejecución para encontrar soluciones y corroborar lo encontrado. Adicionalmente, el curso proveerá con algunos casos particulares que deben cumplirse su optimalidad también. Las pruebas realizadas están en el archivo testsParte1.py y se realizan pruebas para corroborar distintos casos. El primer caso es el de alta variabilidad en el valor de las monedas, luego un caso en el que las monedas estén ordenadas, un caso en el que las monedas sean todas iguales (en caso de ser pares, hay empate, sino gana Sophia), caso con variabilidad moderada y un caso con monedas de mayor valor. Luego están las pruebas de catedra que es en donde se realizan los testeos de pruebas de Volumen.

1.5.1. Pruebas específicas

Pruebas específicas:

```
def prueba1():
    print("Caso Alta Variabilidad")
    print("Monedas = [1, 100, 2, 99]")
    print("Esperado: ")
    print("Sophia: 199, Mateo: 3")
    print("Obtenido: ")
    monedas = [1, 100, 2, 99]
    sophia,_ = elegir_monedas(monedas)
    print(f"Sophia: {sophia}")
    print()
```

```
def prueba2():
    print("Caso Ordenado")
    print("Monedas = [10, 11, 12, 13]")
    print("Esperado: ")
    print("Sophia: 25, Mateo: 21")
    print("Obtenido: ")
    monedas = [10, 11, 12, 13]
    sophia,_ = elegir_monedas(monedas)
```

```
    print(f"Sophia: {sophia}")
    print()
```

```
def prueba3():
    print("Caso con Monedas Iguales")
    print("Monedas = [5, 5, 5, 5]")
    print("Esperado: ")
    print("Sophia: 10, Mateo: 10")
    print("Obtenido: ")
    monedas = [5, 5, 5, 5]
    sophia,_ = elegir_monedas(monedas)
    print(f"Sophia: {sophia}")
    print()
```

```
def prueba4():
    print("Caso con Variabilidad Moderada")
    print("Monedas = [7, 1, 8, 10, 4, 3]")
    print("Esperado: ")
    print("Sophia: 25, Mateo: 8")
    print("Obtenido: ")
    monedas = [7, 1, 8, 10, 4, 3]
    sophia,_ = elegir_monedas(monedas)
    print(f"Sophia: {sophia}")
    print()
```

```

def prueba5():
    print("Caso con Monedas Grande")
    print("Monedas = [20, 50, 30, 40, 60, 10]")
    print("Esperado: ")
    print("Sophia: 130, Mateo: 80")
    print("Obtenido: ")
    monedas = [20, 50, 30, 40, 60, 10]
    sophia,_ = elegir_monedas(monedas)
    print(f"Sophia: {sophia}")
    print()
```

1.5.2. Pruebas de volumen

Pruebas de Volumen:

```
def pruebaCatedra():
    print("Prueba catedra: 1")
    archivo = 'pruebas/20.txt'
    monedas = leerMonedasDesdeArchivo(archivo)
    pasos,ganancia_sophia =
        buscarResultadosEsperados('20.txt')
    ganancia_recibida,pasos_recibidos =
        elegir_monedas(monedas)
    correcto = False
    for i in range(len(pasos)):
        if pasos[i] != pasos_recibidos[i]:
```

```
            correcto = False
        else :
            correcto = True

    if int(ganancia_sophia) != int(ganancia_recibida):
        correcto = False

    print(correcto)
```

1.6. Mediciones

6. Hacer mediciones de tiempos para corroborar la complejidad teórica indicada. Agregar los casos de prueba necesarios para dicha corroboración. Esta corroboración empírica debe realizarse confeccionando gráficos correspondientes, y utilizando la técnica de cuadrados mínimos. Para esto, proveemos una explicación detallada, en conjunto de ejemplos.

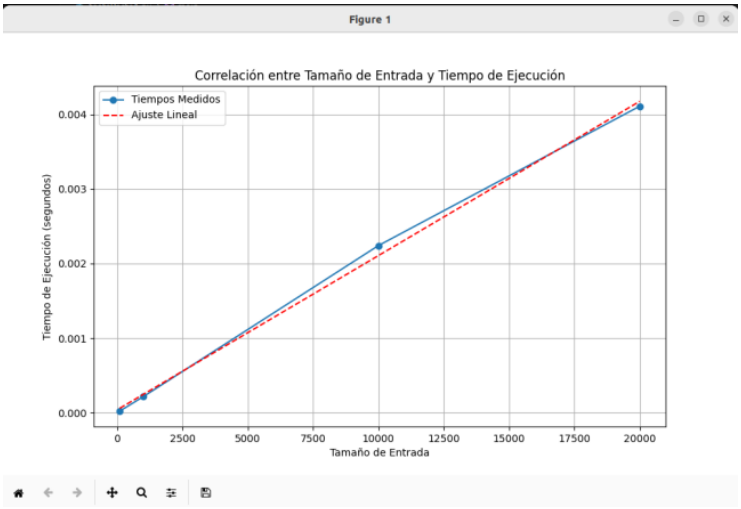

Figura 1: Mediciones de las ejecuciones

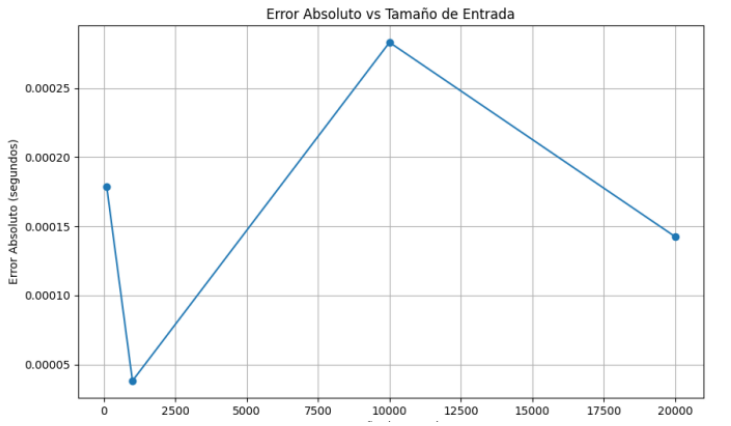

Figura 2: Error del ajuste polinomial

Las mediciones de tiempo están realizadas en el archivo pruebasParte1.py. Tras correr estas pruebas pudimos corroborar que los resultados afirman la complejidad temporal determinada O(n) ya que a medida que aumentamos el tamaño de forma lineal, el tiempo que este tarda, se aumenta de forma lineal. Al utilizar la técnica de cuadrados mínimos, pudimos ver que lo que mejor se ajusta a nuestros resultados obtenidos en una recta con pendiente positiva. Aqui se puede ver como la recta queda muy bien ajustada a los tiempos medidos. Aqui podemos apreciar los errores absolutos y podemos ver que son muy pequeños, por lo que nos lleva a pensar que efectivamente el algoritmo planteado se comporta como O(n).

1.7. Conclusión

7.Conlcusion.

En fin, pudimos concluir que nuestro algoritmo Greedy es $O(n)$, tanto las pruebas de volumen como las individuales especificas funcionan.

2. Segunda Parte

2.1. Análisis y desarrollo

A priori, sabemos que tenemos una secuencia de monedas $m_i \quad \forall i \in \{1 \leq i \leq n\}$ donde el turno inicial corresponde a Sophia, quien elegir la primera moneda. Posteriormente, Mateo elige de la subsecuencia resultante siempre la moneda lateral de mayor valor aplicando un enfoque basado en el algoritmo Greedy. El objetivo es obtener la subsecuencia de monedas con el máximo valor acumulado posible considerando que existen casos excepcionales en los que Mateo puede ganar independientemente de cómo elija Sophia.

A continuación, analizamos el problema para deducir la ecuación de recurrencia necesarias para la implementación posterior, basada en programación dinámica.

2.1.1. Análisis de casos bases

Comenzando con los casos bases, observamos que: -Para una sola moneda, Sophia elige la única disponible y gana. -Para dos monedas, Sophia elige la de mayor valor y también gana. -Para tres monedas, elige la moneda lateral más grande. Dependiendo de la elección de Mateo, ella podría ganar si la moneda del medio no supera la suma de las monedas laterales.

Para el caso de 4 monedas, representadas como $[m_1, m_2, m_3, m_4]$, Sophia tiene dos alternativas iniciales: m_1 y m_4. Si elige m_1, teniedo en cuenta la estrategía de Mateo, quedarian las subsecuencias resultantes serán $[m_3, m_4]$ o $[m_2, m_3]$ dependiendo del resultado de la comparacion máx(m_2, m_4). De manera análoga, si elige la otra alternativa, se considerán las subsecuencias $[m_2, m_3]$ o $[m_1, m_2]$. En este caso, Sophia elegirá el máximo entre los valores acumulados de estas secuencias.

2.1.2. Planteo de la ecuación de recurrencia

En los casos bases, las relaciones son las siguientes:

$$M(i,i) = m_i \quad \text{caso de una moneda, } i = j$$

$$M(i,j) = \text{máx}(m_i, m_j) \quad \text{caso de 2 moneda, } j = i + 1$$

Generalizando los casos y extendiéndolos a la secuencia original $[m_1, \quad m_2, \quad ..., \quad m_i, \quad ..., \quad m_n]$, cuando Sophia elige la i-ésima moneda m_i, Mateo seleccioná la moneda más grandes entre (m_{i+1}, m_j). En el proximo turno, dependiendo de la elección de Mateo, las monedas disponibles serán $[m_{i+2}, \quad m_j]$ o $[m_{i+1}, \quad m_{j-1}]$. Este proceso continúa alternándosehasta que no queden moneda, momento en el que se determina el ganador de juego.

Entonces, del caso generico deducimos que la ecuación de recurrencia será:

$$M(i,j) = \text{máx}\left(m_i + \text{mín}(M(m_{i+2}, m_j), M(m_{i+1}, m_{j-1})),\ m_j + \text{mín}(M(m_{i+1}, m_{j-1}), M(m_i, m_{j-2}))\right.$$

donde $M(i,j)$ representa el máximo valor acumulado dentro del intervalo $[i,j]$.

2.1.3. Justificación de la aplicación de Programación Dinámica

Mediante el análisis de los casos bases y el caso inductivo, observamos que se cumplen las dos propiedades fundamentales de la programación dinámica: la subestructura óptima y el solapamiento entre los subproblemas.

-Subestructura óptima: Cuando Sopphia toma una decisión sobre la subestructura óptima, en particular, la subsecuencia $[m_i, ..., m_j]$ ($i \leq j$, $i \geq 0$, $j \leq n$ depende de la elección óptima de los problemas más pequeños, como las subsecuencias $[m_{i-2}, m_j]$, $[m_{i+1}, m_{j-1}]$, $[m_{i+1}, m_{j-1}]$ y $[m_i, m_{j-2}]$.

-Solapamiento entre los subproblemas: Aunque en la implementacion iterativa (top-down) el solapamiento no es inmedianto, se hace evidente al examinar la solucion recursiva (bottom-up) o la ecuación planteada ya que esta calcula repetidamente problemas más pequenos.

Finalmente, estimamos la cantidad de subproblema que se derivan. Bajo la condicion planteada, sabemos que $1 \leq i \leq j \leq n$, lo que implica i itera en el intervalo $[1, n]$ y, en cada iteración, genera un subproblema con j igual a $n-i+1$. Entonces, el número total del subproblemas es:

$$\sum_{i=1}^{n} i = \frac{n \times (n+1)}{2} = \frac{n^2}{2} + \frac{n}{2}$$

.

En conclusión, también se cumple el requisito de que el número de subproblemas sea polinómico, n^2.

Con la ecuacion de recurrencia y los casos bases definidos, estamos en condición de implementar el algoritmo de programacion dinamica iterativo que resuelve el problema planteado.

2.2. Demostración de la ecuación de recurrencia

2-Demostrar que la ecuación de recurrencia planteada en el punto anterior en efecto nos lleva a obtener el máximo valor acumulado posible.

Para demostrar que la ecuacion de recurrencia presentada, $M(i, j)$, conduce al resultado esperado, recurrimos al principio de induccion débil. Planteamos nuestra proposición $P(k)$ donde afirmamos que la ecuacion de recurrencia $M(i, j)$ genera el maximo valor acumulado que Sophia puede obtener para un problema de tamaño k.

Proposición P(k): Para toda subsecuencia de monedas de tamaño k, definida por los índices i y j con $j - i + 1 = k$, el valor $M(i, j)$ calculado, mediante la ecuación de recurrencia, representa la suma máxima que Sophia puede obtener al jugar óptimamente, asumiendo que Mateo también juega óptimamente (usando su estrategia greedy).

Caso base:

1. Para $k = 1$:

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

En este caso, Sophia toma la única moneda y gana. Este caso es trivialmente correcto ya que no hay decisiones adicionales. Por lo tanto, P(1) es verdadero.

2. Para $k = 2$:

$$M(i, j) = \text{máx}(m_i, m_j)$$

Aquí, ella simplemente elegi la moneda con mayor valor entre m_i y m_j. Por lo tanto, P(2) también es verdadero.

Caso inductivo:

Supongamos que $P(k)$ es verdadero $\forall k \leq n$. Demostraremos que también es verdadero para el caso $k = n + 1$. Primer caso: Si Sophia toma m_i, Mateo selecciona la moneda con mayor valor entre las dos opciones de las monedas restantes,m_{i+1} y m_j. Esto deja a Sophia con una subsecuencia de tamaño reducido:

$$M(i, j) = m_i + \text{mín}(M(m_{i+2}, m_j), M(m_{i+1}, m_{j-1})).$$

Segundo caso:

$$M(i, j) = m_j + \text{mín}(M(m_{i+1}, m_{j-1}), M(m_i, m_{j-2})).$$

Conclusión: Por el principio de inducción, la proposición $P(k)$ es válida $\forall k \geq 1$. Esto demuestra que la ecuación de recurrencia $M(i, j)$ produce efectivamente el máximo valor acumulado posible para Sophia, bajo la suposición de que ambos jugadores siguen estrategias óptimas.

Adicionalmente, se verifica que:

$$\sum_{i=1}^{\alpha} m_i \geq \sum_{j=1}^{\beta} m_j \quad \text{con} \quad \alpha + \beta = k,$$

donde α y β representan los turnos en que Sophia y Mateo seleccionan monedas, respectivamente. Por lo tanto, la ecuación cumple con las propiedades requeridas y lleva al resultado correcto.

2.3. Implementación

3-Escribir el algoritmo planteado. Describir y justificar la complejidad de dicho algoritmo. Analizar si (y cómo) afecta a los tiempos del algoritmo planteado la variabilidad de los valores de las monedas.

```python
def obtener_maxima_monedas(monedas):
    n = len(monedas)
    valores_acumulados = [[0] * n for _ in range(n)]
    elecciones = [[""] * n for _ in range(n)]

    for i in range(n):
        valores_acumulados[i][i] = monedas[i]
        elecciones[i][i] = "primero"

    for intervalo in range(2, n + 1):
        for i in range(n - intervalo + 1):
            j = i + intervalo - 1
            valores_acumulados[i][j], elecciones[i][j] =
                decidir_extremo(monedas, i, j,
                valores_acumulados)

    sophia_monedas, mateo_monedas, ganancia_sophia,
        ganancia_mateo, jugadas =
        recuperar_elecciones_optimas(monedas, elecciones)
    return sophia_monedas, ganancia_sophia, mateo_monedas,
        ganancia_mateo, jugadas
```

```python
def decidir_extremo(monedas, i, j, valores_acumulados):
    siguiente = valores_acumulados[i + 2][j] if i + 2 <= j
        else 0
    ultimo = valores_acumulados[i + 1][j - 1] if i + 1 <= j
        - 1 else 0
    elegir_izquierda = monedas[i] + min(siguiente, ultimo)

    primero = valores_acumulados[i + 1][j - 1] if i + 1 <=
        j - 1 else 0
    anteultimo = valores_acumulados[i][j - 2] if i <= j - 2
        else 0
    elegir_derecha = monedas[j] + min(primero, anteultimo)
    if elegir_izquierda >= elegir_derecha:
        return elegir_izquierda, "primero"
    else:
        return elegir_derecha, "ultimo"
```

```python
def recuperar_elecciones_optimas(monedas, elecciones):
    n = len(monedas)
    i, j = 0, n - 1
    sophia_monedas = []
    mateo_monedas = []
    jugadas = []
    turno_sophia = True

    while i <= j:
        if turno_sophia:  # Turno de Sophia
            if elecciones[i][j] == "primero":
                sophia_monedas.append(monedas[i])
                jugadas.append(f"Sophia elige {monedas[i]} de la izquierda")
                i += 1
            else:
                sophia_monedas.append(monedas[j])
                jugadas.append(f"Sophia elige {monedas[j]} de la derecha")
                j -= 1
        else:  # Turno de Mateo
            if elecciones[i][j] == "primero":
                mateo_monedas.append(monedas[i])
                jugadas.append(f"Mateo elige {monedas[i]} de la izquierda")
                i += 1
            else:
                mateo_monedas.append(monedas[j])
                jugadas.append(f"Mateo elige {monedas[j]} de la derecha")
                j -= 1
        turno_sophia = not turno_sophia

    ganancia_sophia = sum(sophia_monedas)
    ganancia_mateo = sum(mateo_monedas)
    return sophia_monedas, mateo_monedas, ganancia_sophia, ganancia_mateo, jugadas
```

Ahora, para calcular la complejidad algoritmica, usamos la siguiente expresión:

$$T(n) = \sum \text{operaciones elementales} = O(1) + 2 \times O(n^2) + n \times O(1) + O(1) + O(n)$$

$$T(n) = c_1 \times O(n^2) + c_2 \times O(n) + c_3 \times O(1)$$

$$T(n) = O(n^2) \quad \text{(siendo n el tamaño de las monedas)}$$

"La inicialización de las matrices tiene una complejidad de $O(n^2)$, ya que se recorren $n \times n$ posiciones para llenarlas. Dentro del bucle anidado, se realizan operaciones constantes $O(1) \times ((n-2) \times n$ veces, lo que también contribuye a una complejidad de $O(n^2)$. La función decidir-extremo tiene una complejidad de $O(1)$, ya que únicamente realiza operaciones elementales constantes, como indexación, aritmética y comparaciones lógicas. Por último, la función que recupera los datos tiene una complejidad de $O(n)$, dado que utiliza un bucle while

que recorre los extremos de las monedas y realiza n iteraciones, avanzando i o j en cada una de ellas.

Por la definición de Big O Notation, la complejidad se acota con $O(n^2)$.

A continuación, estimamos el impacto de la variabilidad de los números en el tiempo de algoritmo. Para caso de baja variabilidad, según lo mencionado anteriormente, no afecta el tiempo sino al resultado. Del modo contrario, la decisiones que toman se vuelve más significativos y tampoco afectará al tiempo. Por ende, la variabilidad de los valores de las monedas no afecta los tiempos del algoritmo, ya que la complejidad depende solo del tamaño de la lista de monedas.

2.4. Ejemplos

4-Realizar ejemplos de ejecución para encontrar soluciones y corroborar lo encontrado. Adicionalmente, el curso proveerá con algunos casos particulares que deben cumplirse su optimalidad también.

Para corroborar lo encontrado, retomamos la implementación del sección anterior para obtener el resultado del juego donde las monedas tienen alguno patrón o distribución específico tal como monedas de mismo valor y monedas de grandes pequeños intercalados (ver archivo test.py y testpropio.py). En el primer caso de las monedas iguales, trivialmente el resultado dependera de la paridad de las monedas. Del mismo modo, se realizán pruebas de volumen propuesto por la cátedra para estimar y comparar la pendiente gráfica que se utilizara en los próximos secciones.

```
1  #Prueba de volumen
2  #"10000.txt"
3  MONEDAS_10 = obtenerDatos.obtener_monedas("10000.txt")
4
5  RESULTADOS_ESPERADOS =
       obtenerDatos.obtener_resultados("Resultados
       Esperados.txt")
6  class TestMonedasElegidas(unittest.TestCase):
7      def test_monedas_elegidas(self):
8          for archivo, esperado in
               RESULTADOS_ESPERADOS.items():
9              monedas = obtenerDatos.obtener_monedas(archivo)
10             sophia_moneda, sophia_ganancia, mateo_monedas,
                   mateo_ganancia, jugadas =
                   parte2.obtener_maxima_monedas(monedas)
11
12             self.assertEqual(sophia_moneda,
                   esperado["Sophia"], f"Error en Sophia para
                   {archivo}")
13             self.assertEqual(mateo_monedas,
                   esperado["Mateo"], f"Error en Mateo para
                   {archivo}")
```

```
#Prueba de variacion
def probar_ejemplos():
    generador = GeneradorDeMonedas()

    # Casos con un patr n predeterminado y variaci n de n mero espec fico.
    ejemplos = [
        ("Monedas de mismo n mero", generador.generar_ejemplo_de_mismo_numero(10)),
        ("Monedas de dos n meros intercalados", generador.generar_ejemplo_dos_numeros_intercalados(10)),
        ("Monedas con valor grande centrado", generador.generar_ejemplo_con_un_valor_grande_centrado(10)),
        ("Monedas con valores grandes centrados", generador.generar_ejemplo_grandes_centrados(10)),
        ("Monedas con valores grandes en los laterales", generador.generar_ejemplo_grandes_en_los_laterales(10)),
        ("Moendas de Peque os y grandes intercalados", generador.generar_ejemplo_peque os_grandes_intercalados(10)),
        ("Monedas aleatorio", generador.generar_ejemplo_aleatorio(10)),
    ]

    for descripcion, monedas in ejemplos:
        sophia_monedas, ganancia_sophia, mateo_monedas, mateo_ganancia, jugadas = parte2.obtener_maxima_monedas(monedas)
        mostrar_resultado(descripcion, monedas, sophia_monedas, ganancia_sophia, mateo_monedas, mateo_ganancia, jugadas)
```

```
def mostrar_resultado(descripcion, monedas, sophia_monedas, ganancia_sophia, mateo_monedas, mateo_ganancia, jugadas):
    assert ganancia_sophia >= mateo_ganancia, (
        f"Error en '{descripcion}': Sophia no tiene m s ganancia que Mateo.\n"
        f"Ganancia Sophia: {ganancia_sophia}, Ganancia Mateo: {mateo_ganancia}\n"
        f"Monedas: {monedas}"
    )
    print(f"Ejemplo: {descripcion}")
    print(f"monedas: {monedas}")
    print(jugadas)
    print(f"Sophia_monedas: {sophia_monedas}")
    print(f"Sophia_Ganancia: {ganancia_sophia}")
    print(f"Mateo_monedas: {mateo_monedas}")
    print(f"Mateo_Ganancia: {mateo_ganancia}")
    print()
```

2.5. Mediciones

5-Hacer mediciones de tiempos para corroborar la complejidad teórica indicada. Agregar los casos de prueba necesarios para dicha corroboración (generando sus propios sets de datos). Esta corroboración empírica debe realizarse confeccionando gráficos correspondientes, y utilizando la técnica de cuadrados mínimos. Para esto, proveemos una explicación detallada, en conjunto de ejemplos.

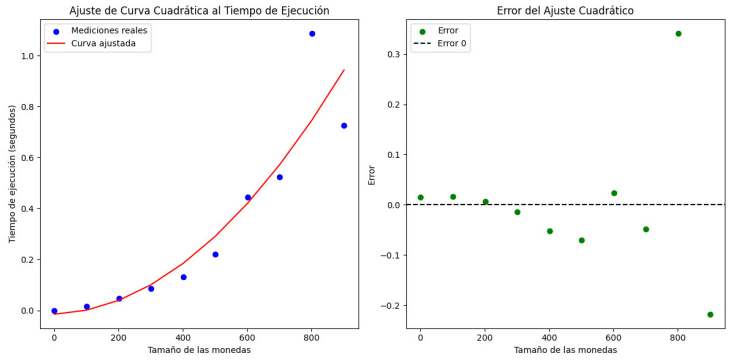

Figura 3: Gráfico de la medición y su ajuste cuadrático con su error

La implementación de esta sección se realiza en el archivo ¡mediciones.py¿. Nuestro objetivo consiste en el estudio estadistico basado en una muestra aleatorio formado por el volumen de dato y el tiempo de ejecución y el análisis con tecnica de minimo cuadrado para corroborar la complejidad calculada , examinar su tendencia de la variación y la discrepancia entre el valor real y el orden teórico.

Observando el gráfico de las mediciones, afirmamos que la curva cuadrática de ajusta respeta la complejidad calculada anteriormente, $O(n^2)$, con un error cuadrático medio de $0,13245$seg

2.6. Conclusión

6-Agregar cualquier conclusión que parezca relevante.

El análisis empírico realizado confirma que el tiempo de ejecución del algoritmo sigue la complejidad teórica calculada, $O(n^2)$. Esto se demuestra mediante el ajuste cuadrático de los tiempos medidos, que muestra una correlación significativa con la curva teórica. Luego, El error cuadrático medio obtenido (0,13245 segundos) es relativamente bajo, lo que indica que el modelo cuadrático es adecuado para describir el comportamiento del tiempo de ejecución. Esto refuerza la validez de la complejidad calculada y su ajuste a los datos experimentales. A su vez, a medida que incrementa el tamaño del parámetro, el incerteza tiende a converge a la curva de ajusta ya que el coeficiente principal de la curva tiene orden de 1×10^{-6}. Por último, destacamos quela variabilidad de los valores de las monedas no toma un peso significativo en los tiempos de ejecución.

3. Tercera Parte

Para demostrar que el problema de la Batalla Naval está en NP, necesitamos argumentar que el problema pertenece a la clase NP, lo que significa que, dado un conjunto de soluciones posibles, podemos verificar si una solución es válida en tiempo polinómico.

3.1. Verificación de una Solución

Los barcos se colocan en la orientación correcta (horizontal o vertical) y no se solapan entre sí. Se cumplen las restricciones de ocupación de casilleros por fila y por columna. (demanda) Verificación de la Ubicación de los Barcos (no se encuentre un barco pegado a otro ni feura del tablero) Los barcos no se pueden dividir (la mitad de un barco en una fila y la otra en otra)

3.2. Complejidad de la Verificación

Ahora, analicemos la complejidad de la verificación. Dada una solución propuesta (es decir, una distribución específica de los barcos en el tablero), se requiere realizar los siguientes pasos:

3.3. Verificación de la Ubicación de los Barcos

Esto implica recorrer cada barco y verificar su posición en el tablero, lo cual puede hacerse en tiempo proporcional al número de barcos y al tamaño del tablero. Este paso tiene una complejidad de $O \times (N \times M)$, dado que debemos inspeccionar los casilleros del tablero para verificar que los barcos no se solapen.

3.3.1. Verificación de las Restricciones de Fila y Columna

Esto implica recorrer las filas y columnas del tablero para contar cuántos casilleros están ocupados en cada una. Este paso también tiene una complejidad de $O \times (N \times M)$, ya que debemos verificar todas las filas y todas las columnas.

3.3.2. Verificacion de los barcos no se encuentran en dos lugares distintos del tablero

Esto implica recorrer el tablero (fila y columna) viendo que no hayan dos numeros referidos al mismo barco en dos filas y dos columnas distinas. Este paso tiene una complejidad de $O \times (N \times M)$, ya que tambien es recorrer fila y columna (chequear el valor que se encuentra es O(1)) En total, la verificación de una solución tiene una complejidad de $O \times (N \times M)$, lo cual es un tiempo polinómico en función del tamaño del tablero.

3.4. ¿Es un problema NP-completo?

Buscamos demostrar que el problema de la batalla naval es un problema NP-completo mediante la reduccion del 3-partition a este.

Reduccion

Entrada para 3-Partition: Se nos da un conjunto $A = \{a_1, a_2, \ldots, a_{3m}\}$ con 3m números y un número C tal que la suma de todos los elementos es mC.

Objetivo

Queremos colocar los barcos de tal manera que cada fila tiene exactamente C casilleros ocupados,y cada columna tiene un número de barcos correspondiente sin violar ninguna restricción.

Construcción del tablero de la Batalla Naval

Creamos un tablero con m filas y C columnas, que tendrá una estructura de m×C. Cada número ai en A se convierte en un barco de longitud ai.Colocamos estos barcos de forma que se sumen exactamente C casilleros ocupados en cada fila (cada subconjunto de la partición).

Colocación de los barcos

Cada barco debe ocupar exactamente ai casilleros consecutivos en una fila (horizontal o vertical). Las restricciones de la fila y columna deben ser tales que la distribución de los barcos no viole las reglas de ocupación de casilleros y que las sumas de casilleros en las filas y las columnas sean coherentes con la partición de los números en subconjuntos de suma C.

3.5. Problema equivalente

Si es posible colocar los barcos en el tablero de la Batalla Naval de manera que se respeten las restricciones de fila y columna, entonces es posible hacer una partición de los números A en m subconjuntos de 3 elementos cuya suma sea C. Esto es lo que define una solución válida al problema de 3-Partition.

3.6. Algoritmo de Aproximacion

El algoritmo de aproximación básicamente va colocando los barcos en las filas/columnas con mayor demanda. La aproximación puede no ser perfecta, ya que el algoritmo no contempla los casos en los cuales la suma de dos barcos con menor tamaño pueden ocupar el lugar de un barco de mayor tamaño.

Dado que el algoritmo siempre coloca los barcos en las posiciones de mayor demanda, en el mejor de los casos, la relación $A(I)/z(I) \geq r(A)$ podría acercarse a 1, pero mientras mas se agranda el problema, mas complicado se vuelve encontrar el optimo.