.UBAfiuba
FACULTAD DE INGENIERÍA

Trabajo Práctico 3
Diversión NP-Completa

Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

Nombre	Padrón
Denise Dall'Acqua	108645
Martín Alejo Polese	106808
Nicolás Agustín Riedel	102130

Índice

Consigna

Introducción

Los hermanos siguieron creciendo. Mateo también aprendió sobre programación dinámica, y cada uno aplicaba la lógica sabiendo que el otro también lo hacía. El juego de las monedas se tornó aburrido en cuánto notaron que siempre ganaba quien empezara, o según la suerte. Los años pasaron, llegó la adolescencia y empezaron a tener gustos diferentes. En general, jugaban a juegos individuales. En particular, Sophia estaba muy enganchada con un juego inventado en Argentina por Jaime Poniachik (uno de los fundadores de Ediciones de Mente) en 1982: La Batalla Naval Individual.

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.

Consigna

1. Hacer un análisis del problema, plantear la ecuación de recurrencia correspondiente y proponer un algoritmo por programación dinámica que obtenga la solución óptima al problema planteado: Dada la secuencia de monedas m1,m2 .... mn,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.

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

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.

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.

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.

6. Agregar cualquier conclusión que les parezca relevante.

Resolución

1. Ecuación de recurrencia

A continuación se mostrará la **ecuación de recurrencia** hallada para este problema:

$$T(monedas, inicioFila, finFila) = \begin{cases} K_1 & si \quad K_1 > K_2 \\ K_2 & si \quad K_1 < K_2 \end{cases}$$

Siendo

$$K_1 = monedas[inicioFila] + T(monedas, S(monedas, inicioFila + 1, finFila))$$
$$K_2 = monedas[finFila] + T(monedas, S(monedas, inicioFila, finFila - 1))$$

donde

$$S(monedas, inicioFila, finFila) = \begin{cases} (inicioFila + 1, finFila) & si \quad monedas[inicioFila] > monedas[finFila] \\ (inicioFila, finFila - 1) & si \quad monedas[inicioFila] < monedas[finFila] \end{cases}$$

NOTA:

$monedas$ = El vector con los valores de las monedas

$inicioFila$ = Es el valor de la primera moneda del vector monedas

$finFila$ = Es el valor de la última moneda del vector monedas

Nuestra función $T(monedas, inicioFila, finFila)$ nos otorga la ganancia que consiguió Sophia al momento de jugar con una cantidad de n monedas contra Mateo. En esta, podemos observar como nuestras variables K_1 y K_2 realizan llamados recursivos a T teniendo en cuenta como variables $inicioFila$ y $finFila$ la salida de otra función llamada $S(monedas, inicioFila, finFila)$, que son las dos posibles decisiones que puede tomar mateo al elegir una moneda (recordemos que mateo sigue las reglas del juego estríctamente).

2. Demostración del Algortimo

Para que la ecuación de recurrencia sea la solución óptima, osea que nos de el **máximo valor acumulado posible**, basta con verificar que:

1. La solución óptima de un problema grande puede obtenerse combinando soluciones óptimas de subproblemas más pequeños.
2. Esta descomposición debe hacerse de tal manera que no se omita ninguna posibilidad relevante para la solución óptima.
3. Se debe tomar una decisión sobre cuál subproblema o combinación de subproblemas proporciona el valor máximo.

- Nuestro caso base sería: Si no hay monedas, Sophia tendría ganancia cero ya que no habría monedas para seleccionar.
- Si Sophia agarra la primera moneda, podemos calcular su ganancia como el valor de esa moneda, más lo que ella recolectará en turnos posteriores. Esto teniendo en cuenta que Mateo siempre sigue las reglas del juego de manera óptima. En este punto del análisis, se ve como el problema se divide en subproblemas, osea por cada turno de Sophia.

- Si Sophia agarra la última moneda, realizamos la misma lógica anterior, nada más que para la última moneda.
- Luego, elegimos cuál de los dos escenarios logra obtener la máxima ganancia. A medida que vamos maximizando los escenarios locales, finalmente llegaremos al óptimo global de nuestro problema, osea la ganacia máxima posible que puede obtener Sophia.

2.1. Demostración de NP-completitud

Reducción desde el problema 3-Partition

Vamos a demostrar que la batalla naval es NP completo. Pero, ¿Como se demuestra que un algortimo es NP completo? Tiene que cumplir dos puntos:

- **Pertenencia a NP:** La verificación de una solución candidata es posible en tiempo polinomial.
- **NP-dificultad:** Se puede realizar una reducción polinomial desde cualquier problema NP-completo hacia este problema.

Ya en la sección anterior pudimos verificar con éxito que nuestro problema es un problema NP, ahora hay que demostrar que se puede realizar una reducción polinomial desde cualquier problema NP-Completo. Recordemos que todos los NP completos pueden ser reducidos despues cualquier problema NP completo.

En nuestro caso, utilizamos el problema de *3-Partition* para verificar que la batalla naval puede ser resuelta en tiempo logarítmico.

Definamos un poco cómo se compone el problema *3-Partition*

Definición del problema 3-Partition

Dado un conjunto de $3m$ números positivos $A = \{a_1, a_2, \ldots, a_{3m}\}$, donde cada número está representado en notación unaria y la suma total de los números es $S = m \cdot B$, queremos particionar A en m subconjuntos disjuntos $S_1, S_2, \ldots, S_m$ tales que:

- Cada subconjunto S_i tiene exactamente 3 elementos.
- La suma de los elementos en cada subconjunto es exactamente B.

Reducción de 3-Partition a La Batalla Naval

Dado una instancia del problema *3-Partition*, construiremos una instancia del problema *La Batalla Naval*.

Construcción del tablero:

- **Dimensiones del tablero:** Construimos un tablero con m filas y $3m$ columnas. Las restricciones de las filas (r_i) serán B, es decir, cada fila i debe contener exactamente B casillas ocupadas.
- **Restricciones de las columnas:** Las restricciones de las columnas (c_j) serán los elementos del conjunto $A = \{a_1, a_2, \ldots, a_{3m}\}$. Es decir, la columna j tendrá una restricción igual a a_j.
- **Barcos:** Introducimos $3m$ barcos, uno por cada elemento de A, donde el barco b_j tiene una longitud igual a a_j.

vamos con un ejemplo para poner en contexto esto que estamos diciendo:

Tenemos los siguientes datos del problema 3-Partition:

- $A = \{6, 5, 4, 5, 3, 7\}$,
- $B = 15$, porque por ejemplo si tengo los subconjuntos $S_1 = 6, 5, 4$ y $S_2 = 5, 3, 7$ la suma de los elementos de S_1 es 15 al igual que la suma de los elementos de S_2
- $m = 2$. porque $|A| = 6 = 3{,}2 => 3.m$

Ajustando el Tablero

Para que cada fila pueda contener exactamente $r_i = 15$ casillas ocupadas, necesitamos que el número de columnas del tablero sea suficiente para permitir esa suma. Ajustamos las dimensiones del tablero:

- $n = 2$ filas (porque $m = 2$, el número de subconjuntos).
- $m = 15$ columnas (ya que cada fila debe contener hasta 15 casillas ocupadas).

Por lo tanto, el tablero tiene $2 \times 15 = 30$ casillas, lo cual coincide con la suma total de $\sum A = 30$.

Nueva Configuración

1. **Restricciones de Filas y Columnas:**
   - Cada fila i debe tener exactamente $r_i = B = 15$ casillas ocupadas.
   - Cada columna j debe contener exactamente $c_j = a_j$, con $A = \{6, 5, 4, 5, 3, 7\}$.
2. **Barcos:** Introducimos un barco para cada elemento en A:
   - Barco 1: longitud 6,
   - Barco 2: longitud 5,
   - Barco 3: longitud 4,
   - Barco 4: longitud 5,
   - Barco 5: longitud 3,
   - Barco 6: longitud 7.
3. **Restricciones Adicionales:**
   - Los barcos no pueden ser adyacentes entre sí (ni horizontal, ni vertical, ni diagonalmente).
   - Todos los barcos deben estar dentro del tablero, y la suma total de las celdas ocupadas debe ser $\sum A = 30$.

Solución del Problema de Batalla Naval

Dado el tablero con $n = 2$ filas y $m = 15$ columnas, encontramos la siguiente distribución:

- **Fila 1 (S_1):** Barcos $\{6, 5, 4\}$, que suman $6 + 5 + 4 = 15$.
- **Fila 2 (S_2):** Barcos $\{5, 3, 7\}$, que suman $5 + 3 + 7 = 15$.

Esto satisface las restricciones de las filas ($r_i = 15$) y las columnas ($c_j = a_j$).

Relación entre 3-Partition y Batalla Naval

1. Resolver el problema de *La Batalla Naval* (encontrar la disposición de los barcos en el tablero) equivale a resolver el problema de 3-Partition:
   - Cada fila representa un subconjunto de A.
   - La longitud de los barcos en cada fila corresponde a los elementos del subconjunto.
   - La suma de las longitudes de los barcos en cada fila debe ser igual a $B = 15$.
2. Si no existe una solución para el tablero naval, no existe una partición válida en el problema de 3-Partition.

Equivalencia de las instancias

- Si existe una solución para el problema *3-Partition*, entonces podemos construir una configuración válida del tablero para *La Batalla Naval*.
- Si existe una configuración válida para *La Batalla Naval*, entonces podemos construir una partición válida para el problema *3-Partition*.

Conclusión

Hemos demostrado que:

- *La Batalla Naval* pertenece a NP.
- Reducimos un problema NP-completo (*3-Partition*) a *La Batalla Naval* en tiempo polinomial.

Por lo tanto, *La Batalla Naval* es NP-completo.

3. Algoritmo planteado y complejidad

El algoritmo que decidimos utilizar para resolver el problema, es el siguiente:

- Mientras hayan monedas para elegir:
  - Vemos todas las monedas disponibles y en función de eso:
    - En el turno de Sophia, se elige la moneda que maximice la ganancia total, teniendo en cuenta que Mateo siempre va a elegir la moneda de mayor valor en todos los turnos posteriores.
    - En el turno de Mateo, se elige la moneda de mayor valor.
- Devolvemos la ganancia acumulada de Sophia.

Veamos el código:

```
    def juego_monedas(arr):
        memory = {}
        return _juego_monedas(arr, 0, len(arr), memory), memory

    def _juego_monedas(arr, inicio = 0, fin = 0, memory = {}):
        if inicio >= fin:
            return 0

        key = (inicio, fin)
        if key in memory:
            return memory[key]
```

```
12
13          primer_moneda = arr[inicio]
14          ultima_moneda = arr[fin-1]
15
16          decision_mateo_1 = decision_mateo(arr, inicio + 1, fin)
17          decision_mateo_2 = decision_mateo(arr, inicio, fin-1)
18
19          ganancia = max(primer_moneda + _juego_monedas(arr,decision_mateo_1[0],
    decision_mateo_1[1], memory),
20                         ultima_moneda + _juego_monedas(arr, decision_mateo_2[0],
    decision_mateo_2[1], memory))
21
22          memory[key] = ganancia
23
24          return ganancia
25
26      def decision_mateo(arr, inicio = 0, fin = 0):
27          if arr[inicio] > arr[fin - 1]:
28              return (inicio + 1, fin)
29          return (inicio, fin - 1)
```

Lo que estamos haciendo, es de forma recursiva observar todas las decisiones que puede tomar Sophia (sabiendo como se comporta Mateo) y quedarnos con la que maximice la ganancia total. Partimos de un caso base, donde si Sophia no tiene monedas para elegir, la ganancia es 0. Luego en cada llamado recursivo, vemos que moneda debemos elegir para maximizar la ganancia de ese subproblema, y guardamos en memoria el resultado para no tener que volver a calcularlo en caso de que se repita. De esta forma, combinando la solución de los subproblemas, encontramos la solucion del problema original, que resulta en la máxima ganancia que puede obtener Sophia.

La complejidad temporal de este algoritmo en principio sería exponencial $(\mathbf{O}(2^n))$, ya que en cada llamado recursivo se hacen dos llamados recursivos más, y así sucesivamente. Sin embargo, al utilizar memoria para guardar los resultados de los subproblemas, evitamos tener que recalcularlos, reduciendo la complejidad a cuadrática $(\mathbf{O}(n^2))$. Esta complejidad surge de que la cantidad de subproblemas a resolver es n^2, ya que cada subproblema esta definido por un intervalo i,j donde i y j pueden tomar valores de 0 a $n-1$, siendo n la cantidad de monedas disponibles.

La complejidad espacial del algoritmo es $\mathbf{O}(n^2)$ ya que la memoria utilizada para guardar los resultados de los subproblemas es proporcional a la cantidad de subproblemas a resolver. Respecto al tamaño del stack es despreciable frente a este valor, ya que a lo sumo hay n llamados recursivos en simultaneo.

En resumen, la complejidad temporal y espacial del algoritmo es $\mathbf{O}(n^2)$.

NOTA: En este análisis no se considera el algoritmo para la reconstrucción de la solución.

3.1. Variabilidad

En relación a la complejidad algorítmica, la variabilidad de los valores de las monedas **no** afectan a los tiempos del algoritmo planteado, ya que los valores se utilizan para hacer comparaciones entre las monedas, y adiciones sobre el acumulado.

4. Ejemplos de ejecución

Se pueden encontrar encontrar multiples ejemplos de ejecución dentro del directorio *ejemplos* en el repositorio. Veamos algunos:

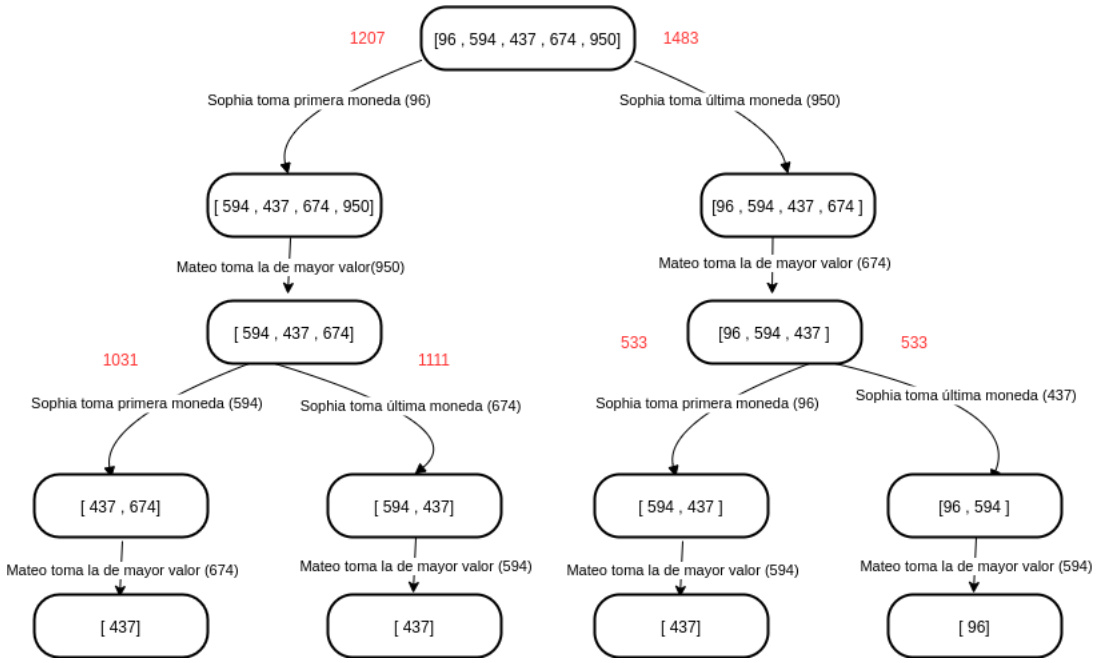
Supongamos que tenemos 5 monedas:

```
1   [96, 594, 437, 674, 950]
```

En cada turno, Sophia tiene la opcion de elegir 2 monedas, la del principio o la del final. Esto implica que cada turno puede representarse como 2 opciones posibles, donde se debe elegir la opcion que acumule el mayor valor.

Para esto se recorren las opciones de manera recursiva, y se va seleccionando el maximo de los valores agregados en cada turno.

De esta forma, como se puede observar en el siguiente grafico, Sophia puede obtener un maximo de 1.483 puntos.



Veamos lo que obtenemos al ejecutar en código el ejemplo que acabamos de ver:

```
❯ python3 main.py ./ejemplos/5.txt
Ganancia Sophia: 1483
Ganancia Mateo: 1268

Movimientos:

Sophia debe agarrar la ultima (950)
Mateo agarra la ultima (674)
Sophia debe agarrar la primera (96)
Mateo agarra la primera (594)
Sophia debe agarrar la primera (437)
```

Veamos la ejecución de otros ejemplos brindados por la cátedra:

Este ejemplo es con 10 monedas:

```
❯ python3 main.py ./ejemplos/10.txt
Ganancia Sophia: 2338
Ganancia Mateo: 1780

Movimientos:

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)
```

Este ejemplo es con 25 monedas:

```
❯ python3 main.py ./ejemplos/25.txt
Ganancia Sophia: 7491
Ganancia Mateo: 6523

Movimientos:

Sophia debe agarrar la primera (704)
Mateo agarra la primera (752)
Sophia debe agarrar la primera (956)
Mateo agarra la primera (874)
Sophia debe agarrar la primera (790)
Mateo agarra la ultima (361)
Sophia debe agarrar la ultima (539)
Mateo agarra la ultima (323)
Sophia debe agarrar la ultima (623)
Mateo agarra la ultima (935)
Sophia debe agarrar la ultima (30)
Mateo agarra la ultima (390)
Sophia debe agarrar la ultima (367)
Mateo agarra la ultima (172)
Sophia debe agarrar la ultima (983)
Mateo agarra la ultima (427)
Sophia debe agarrar la ultima (266)
Mateo agarra la ultima (345)
Sophia debe agarrar la primera (157)
Mateo agarra la ultima (903)
Sophia debe agarrar la ultima (921)
Mateo agarra la ultima (636)
Sophia debe agarrar la ultima (931)
Mateo agarra la primera (405)
Sophia debe agarrar la primera (224)
```

Como podemos observar, los resultados obtenidos coinciden con los *Resultados esperados* compartidos por la cátedra.

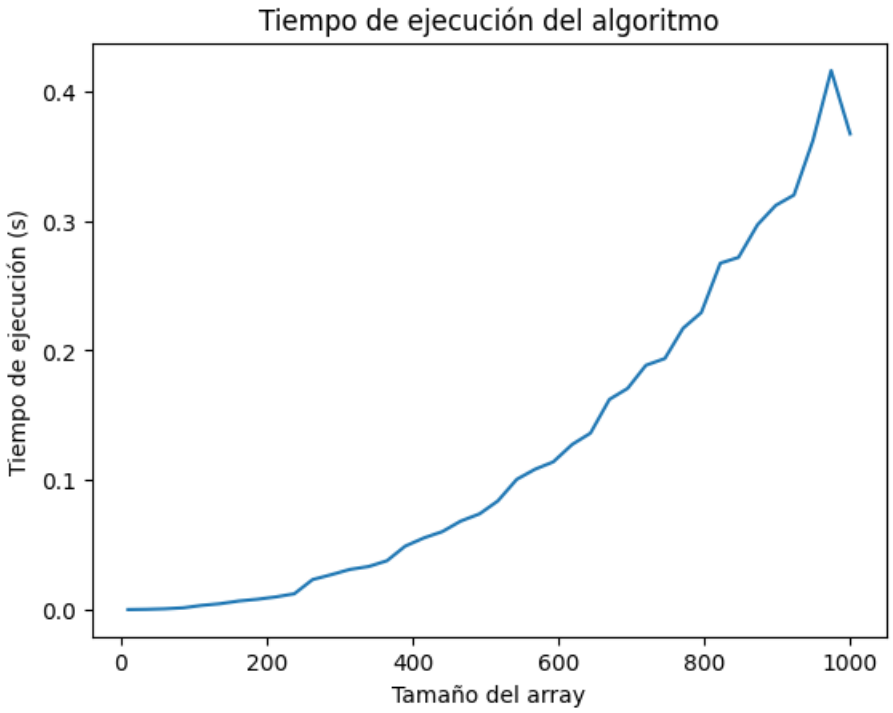
Ademas de estos ejemplos, se pueden encontrar otros en el directorio *ejemplos* del repositorio. Los mismos pueden ser ejecutados con el comando *pytest -v -s* en la terminal.

```
riedel@riedel-ThinkBook-14-G4-IAP:~/Documents/fiuba/TDA/TDA-TP2$ pytest -v -s
============================= test session starts ==============================
platform linux -- Python 3.12.3, pytest-7.4.4, pluggy-1.4.0 -- /usr/bin/python3
cachedir: .pytest_cache
rootdir: /home/riedel/Documents/fiuba/TDA/TDA-TP2
collected 11 items

test_main.py::test_4_monedas PASSED
test_main.py::test_5_monedas PASSED
test_main.py::test_10_monedas PASSED
test_main.py::test_20_monedas PASSED
test_main.py::test_25_monedas PASSED
test_main.py::test_50_monedas PASSED
test_main.py::test_100_monedas PASSED
test_main.py::test_1000_monedas PASSED
test_main.py::test_2000_monedas PASSED
test_main.py::test_5000_monedas PASSED
test_main.py::test_10000_monedas PASSED

============================================================================== 11 passed in 40.65s ====
riedel@riedel-ThinkBook-14-G4-IAP:~/Documents/fiuba/TDA/TDA-TP2$
```

5. Medición empírica

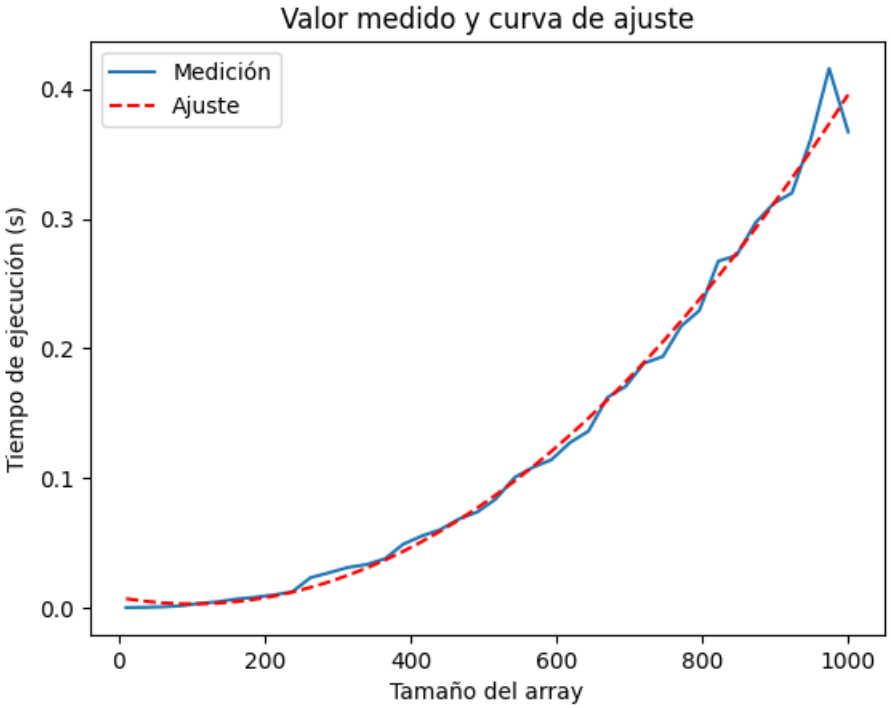
Para comprobar empíricamente la complejidad $\mathbf{O}(n^2)$ del algoritmo, se decidió ejecutar el mismo con distintos tamaños de entrada y medir el tiempo de ejecución. Se generaron muestras de tamaño n, las cuales varían desde 10 hasta 1000.

Para cada muestra se registró el tiempo de ejecución, obteniendo el siguiente gráfico:



A simple vista se puede observar un crecimiento cuadratico. Para confirmar esto, vamos a ajustar los datos a una recta mediante cuadrados mínimos. Esto lo realizamos con Python y la función *optimize.curve_fit* de la biblioteca *scipy*.

Obtenemos que el gráfico se puede ajustar a la curva $y = 4{,}86e^{-07}x^2 - 9{,}84e^{-05}x + 0{,}007$, con un error cuadrático medio de $9{,}37e^{-05}$. Por lo tanto, podemos verificar lo que ya vimos en la sección 3, que el orden es cuadratico $\mathbf{O}(n^2)$.



6. Conclusiones

Pudimos observar y verificar lo siguiente:

- Utilizando la técnica de programación dinámica con memorización, pudimos reducir la complejidad de un algortimo exponencial a uno cuadrático, al no tener que recalcular problemas anteriormente resueltos.
- Tras realizar un análisis empírico, pudimos confirmar que efectivamente la complejidad de nuestro algoritmo se vió beneficiada por la técnica de programación dinámica.
- la variabilidad de los valores de las monedas **no** afecta en el tiempo de ejecución del algortimo.

En conclusión, el presente trabajo permitió afianzar los conocimientos adquiridos en la materia de una manera práctica, donde desarrollamos un algortimo con programación dinámica para resolver el problema planteado, con una complejidad cuadrática, obteniendo de esta manera la ganancia máxima que pudo haber adquirido Sophia.