
Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

# Trabajo Práctico 1
# Los Algoritmos Greedy son juegos de niños

9 de octubre de 2024


| | |
|---|---|
| Victoria Avalos | 108434 |
| Santiago Tissoni | 103856 |
| Facundo Monpelat | 92716 |

# 1. Introducción

Los algoritmos Greedy se caracterizan por seguir un enfoque çodicioso", en el que se toman decisiones locales óptimas con la esperanza de que conduzcan a una solución global óptima. Sin embargo, dicho enfoque no nos garantiza obtener soluciones óptimas en todos los casos. Esta estrategia se utiliza usualmente para resolver problemas que requieren obtener máximos o mínimos. Algunos ejemplos conocidos de algoritmos Greedy son el algoritmo de Dijkstra, los algoritmos de Prim y Kruskal, y el problema de la mochila.

El objetivo de este trabajo práctico es aplicar y analizar el uso de un algoritmo Greedy en un escenario específico: un juego de toma de decisiones en el que se posee una fila de monedas de diferentes valores donde se debe seleccionar una de alguno de los extremos con el objetivo de acumular la mayor puntuación.

A lo largo del trabajo, se evaluará si es posible garantizar una solución óptima utilizando un enfoque Greedy, así como los factores que podrían afectar la optimalidad y la eficiencia del algoritmo propuesto.

## 1.1. Descripción del Problema

Se dispone de una fila de n monedas, cada una con un valor distinto. En cada turno, el jugador puede elegir solo entre la primera o la última moneda de la fila, removiendo la elegida y cediendo el turno al otro jugador. Los jugadores son dos hermanos, Sophia y Mateo, los cuales poseen 7 y 4 años respectivamente. Sophia, siendo más competitiva y con conocimientos de algoritmos Greedy, toma las decisiones tanto por ella misma como por Mateo, con el objetivo de garantizar que su suma total sea mayor que la de su hermano.

Respecto de los valores de las monedas, se considerará que son siempre positivos y no se admite una cantidad par de monedas con el mismo valor.

A lo largo de este trabajo, se propone un algoritmo que permite a Sophia seleccionar las monedas de forma tal que asegure la victoria, maximizando el valor acumulado para ella y minimizando el valor que obtiene Mateo.

## 1.2. Objetivos

Los objetivos del presente trabajo práctico son los siguientes:

- Desarrollar un algoritmo Greedy que obtenga la solución óptima al problema planteado, es decir indicar qué monedas debe ir eligiendo Sophia para sí misma y para Mateo, de tal forma que se asegure de ganar siempre.
- Proveer una demostración de que dicho algoritmo obtiene siempre la solución óptima.
- Describir y justificar la complejidad algorítmica de la solución planteada.
- Analizar la variabilidad de los valores de las diferentes monedas a los tiempos del algoritmo planteado y si afecta la optimalidad.
- Hacer ejemplos de ejecución y analizar lo encontrado.
- Realizar mediciones de tiempos para verificar la complejidad teórica indicada.

# Resolución

## 2. Algoritmo para que gane siempre Sophia

En este trabajo realizaremos el análisis teórico y empírico de un algoritmo greedy que resuelve el problema de determinar las monedas que escogerán dos jugadores, Sophia y Mateo, en un juego por turnos. Nuestro programa permite ejecutar dicho algoritmo con los casos de prueba que se le pasen en formato txt con el comando

```
python3 algoritmo_greedy.py <ruta_archivo_txt>
```

También posee pruebas automatizadas que se ejecutan con

```
python3 run_tests.py
```

y soporta la generación automática de nuevos casos de prueba con los comandos

```
python3 generador.py <tam_maximo_arreglo> <valor_maximo_elementos>
python3 test_auto.py
```

Para más información, leer el repositorio con el código

### 2.1. Algoritmo Greedy

Para resolver el problema, implementamos un algoritmo que comienza inicializando tres estructuras de datos: dos listas vacías, sophia y mateo, para almacenar las monedas que cada jugador selecciona, y una lista esperados que registra el orden y las decisiones tomadas durante el proceso. La fila de monedas se convierte en una cola de doble extremo (deque), permitiendo a los jugadores tomar monedas tanto del principio como del final de la fila de manera eficiente.

En cada iteración del ciclo, Sophia toma primero su turno. La estrategia de Sophia es siempre elegir la moneda de mayor valor disponible, comparando la primera y la última moneda de la fila. Si la primera moneda es de mayor valor, Sophia la selecciona, y en caso contrario, toma la última moneda. Esta acción es registrada en la lista esperados, indicando si tomó la primera o la última moneda.

Una vez que Sophia ha hecho su selección, se verifica si quedan monedas en la fila. Si la fila está vacía, el juego termina. De lo contrario, es el turno de Mateo. A diferencia de Sophia, Mateo utiliza una estrategia inversa, seleccionando la moneda de menor valor entre las disponibles en los extremos de la fila. Al igual que con Sophia, la selección de Mateo también se registra en esperados.

El ciclo continúa hasta que no quedan monedas por seleccionar. Al final, el algoritmo devuelve tres resultados: las monedas seleccionadas por Sophia, las seleccionadas por Mateo, y una lista de eventos que describe detalladamente las decisiones de cada jugador a lo largo del juego.

A continuación, mostramos la implementación en Python de dicho algoritmo. Lo implementamos mediante la función $problema\_monedas(fila)$ que recibe una lista de enteros fila, representando los valores de las monedas.

Como se mencionó anteriormente, la función devolverá tres listas: una con los valores de las monedas que elige Sophia, otra con los valores que elige Mateo y un una que contiene un mensaje de qué lado extrajo Sophia o Mateo las monedas según sus turnos.

```
def problema_monedas(fila):
    sofia = []
    mateo = []
    esperados = []

    fila = deque(fila)

    while len(fila) > 0:
        # Turno de Sofia
        if fila[0] > fila[-1]:
            sofia.append(fila[0])
```

```
            fila.popleft()
            esperados.append("Primera moneda para Sophia")
        else:
            sofia.append(fila[-1])
            fila.pop()
            esperados.append(" ltima  moneda para Sophia")

        if len(fila) == 0:
            break

        # Turno de Mateo
        if fila[0] >= fila[-1]:
            mateo.append(fila[-1])
            fila.pop()
            esperados.append(" ltima  moneda para Mateo")
        else:
            mateo.append(fila[0])
            fila.popleft()
            esperados.append("Primera moneda para Mateo")

    return sofia, mateo, esperados
```

## 2.2. ¿Por qué es un algoritmo Greedy?

El algoritmo presentado puede clasificarse como Greedy porque en cada turno ambos jugadores toman decisiones basadas en un criterio de ganancia local, sin evaluar el impacto de sus decisiones en las jugadas futuras.

- Estados actuales: En cada iteración del algoritmo, el estado está dado por la disposición de la fila de monedas que aún no han sido seleccionadas. Este estado se actualiza dinámicamente a medida que las monedas son retiradas de los extremos.
- Óptimos locales: El óptimo local para Sophia en cada turno consiste en elegir la moneda de mayor valor entre las dos disponibles en los extremos de la fila, maximizando así su ganancia inmediata. Para Mateo, el óptimo local consiste en tomar la moneda de menor valor disponible entre los extremos, minimizando su ganancia inmediata. Estas desiciones se toman en base al estado actual del juego, sin considerar configuraciones pasadas ni futuras.
- Óptimo global: El objetivo global para Sophia es garantizar que acumulará más puntos que Mateo. En la siguiente sección se justifica por qué el algoritmo garantiza este óptimo.
- Medida de repetición: La regla Greedy se repite en cada turno del juego: Sophia siempre compara las dos monedas en los extremos de la fila del estado actual y selecciona la de mayor valor, mientras que Mateo compara los extremos de la fila del estado actual y selecciona la de menor valor. Esta medida de repetición asegura que ambos jugadores sigan consistentemente su estrategia Greedy a lo largo de todo el proceso. En ninguna iteración se realizan otras acciones que no sean las mencionadas anteriormente.

Por tanto, el algoritmo toma decisiones locales y repetitivas, cumpliendo con la característica fundamental de los algoritmos Greedy: optimizar paso a paso sin considerar el estado final del problema.

## 2.3. Análisis de optimalidad

**Definiciones:**

- $r$: número de ronda. Una ronda puede consistir de 2 o 1 monedas. El último caso solamente se da si en la fila solamente queda una moneda. Siempre comienza Sophia.
- $V(\text{moneda } i)$: El valor de la moneda $i$.

- $S(r)$: La suma de los valores de las monedas elegidas por Sophia hasta la ronda $r$.
- $M(r)$: La suma de los valores de las monedas elegidas por Mateo hasta la ronda $r$.
- $n$: cantidad de monedas en la fila. $n \in \mathbb{N}$.
- $Valor(r, \text{Jugador})$ : El valor de la moneda elegida por el Jugador dado en la ronda r.

Estrategia a seguir por cada jugador en un determinado turno:

$$Sophia = max(V(0), V(n))$$
$$Mateo = min(V(0), V(n))$$

Aclaración: el valor de n disminuye dinámicamente luego de cada turno en 1 valor: n, n-1, ..., 0.

**Demostración por Inducción según el número de rondas ($r$):**

**Caso Base ($r = 1$):**

$$S(1) = \text{máx}(V(1), V(n))$$
$$M(1) = \text{mín}(V(1), V(n))$$

- Si $n = 1$: $S(1) = V(1)$, $M(1) = 0 \Rightarrow S(1) > M(1)\ \forall V(i) > 0$.
- Si $n = 2$: por definición, $\text{máx}(V(1), V(2)) > \text{mín}(V(1), V(2)) \Rightarrow S(1) > M(1)$.

Por lo tanto, $S(1) > M(1)$.

**Hipótesis Inductiva:** Supongamos que para cualquier ronda $k \leq r$, se cumple que:

$$S(k) > M(k) \tag{1}$$

**Paso Inductivo:** Consideremos la ronda $r + 1$:

**Caso 1:** $n \geq 2$. Sophia elige la moneda de mayor valor de los dos extremos disponibles. Mateo elige la moneda de menor valor de los dos extremos disponibles.

$$S(r+1) = S(r) + \text{Valor}(\text{ronda } r+1, \text{Sophia})$$

$$M(r+1) = M(r) + \text{Valor}(\text{ronda } r+1, \text{Mateo})$$

Para analizar qué sucede con los valores de las monedas elegidas en esta ronda, formalizamos la fila con las monedas restantes como $\{A, B, ..., C\}$, donde $A > C$. Según el algoritmo, Sophia elige $A$. Quedan las monedas $B$ y $C$.

Si $B > C$: Entonces $Mateo = min(B, C) = C$. Recordemos que $A > C$, por lo que $C > A$ sería un absurdo.

Si $C > B$: Entonces $Mateo = min(B, C) = B$. Como $A > C$, por transitividad resutla que $A > B$. Igual que en el caso anterior, $B > A$ es un absurdo.

En ambos casos, Mateo se queda con una moneda de menor valor que la elegida por Sophia.

De esta forma

$$\text{Valor}(\text{ronda } r+1, \text{Sophia}) \geq \text{Valor}(\text{ronda } r+1, \text{Mateo}) \tag{2}$$

Si la fila resultante se hubiera establecido como $\{A, ..., B, C\}$ con $C > A$, el análisis es análogo al anterior llegando a la misma conclusión.

Sumando (1) con (2), obtenemos:

$$S(r+1) \geq M(r+1)$$

**Caso 2:** $n = 1$. Sophia elige la única moneda restante, por lo que:

$$S(r+1) = S(r) + V(n)$$

$$M(r+1) = M(r)$$

Considerando que $V(n) \geq 0$ y la inecuación (1) resulta que

$$S(r+1) \geq M(r+1)$$

**Conclusión:** En ambos casos, se cumple que $S(r+1) \geq M(r+1)$, lo que concluye el paso inductivo y completa la demostración por inducción.

Hemos demostrado que en cada ronda, independientemente de la cantidad de monedas restantes, Sophia obtiene un valor mayor o igual que Mateo, siempre y cuando ambos jugadores se limiten a elegir monedas de los extremos. Por lo tanto, al finalizar el juego, la suma total de los valores obtenidos por Sophia será mayor o igual que la suma total de los valores obtenidos por Mateo. En consecuencia, Sophia siempre ganará siguiendo la estrategia greedy propuesta.

## 2.4. Variabilidad del valor de las monedas respecto de la optimalidad

Teniendo en cuenta que no puede haber una cantidad par de monedas del mismo valor, en la demostración de optimalidad se puede apreciar que la variabilidad del valor de las monedas no afecta a la optimalidad del algoritmo. En ningún momento de la demostración tuvimos en cuenta la magnitud de los valores. Esto se puede observar en la estrategia de cada jugador:

$$Sophia = max(V(0), V(n))$$

$$Mateo = min(V(0), V(n))$$

Las únicas operaciones que se realizan respecto del valor de las monedas son comparaciones de máximos y mínimos. No influye la magnitud de los valores a comparar ya que es una operación O(1), solamente importa cuál valor es mayor.

Más adelante en este informe, en 2,8$Casosdeprueba$ se puede observar de forma empírica con casos de prueba autogenerados con diferentes valores máximos de las monedas.

## 2.5. Variabilidad del valor de las monedas para los tiempos de ejecucion

Los diferentes niveles de variabilidad no afectan los tiempos de ejecucion de algoritmos. Para un array de tamaño $n$ de variabidad baja, el tiempo de ejecucion del algoritmo es similiar a un array de tamaño $m$ con $m = n$ de variabilidad alta, debido a que cada iteracion dentro del algoritmo, los valores de las monedas son unicamente comparados con complejidad $O(1)$ constante, y comparar numeros de alta variabilidad tiene la misma complejidad y tiempo de ejecucion que comparar numeros "mas cercanos"

Veamos una ejecucion para comprobarlo:

```
[24] ✓ 0.0s
```
```python
from utils.utils import _time_run
from problema import problema_monedas
```

```
[31] ✓ 5.9s
```
```python
partida_baja_variabilidad = (
    open("./tests_cases/variabilidad_tiempos/partida_baja_variabilidad.txt", "r")
    .readlines()[0]
    .split(";")
)
partida_baja_variabilidad = [int(i) for i in partida_baja_variabilidad]

partida_alta_variabilidad = (
    open("./tests_cases/variabilidad_tiempos/partida_alta_variabilidad.txt", "r")
    .readlines()[0]
    .split(";")
)
partida_alta_variabilidad = [int(i) for i in partida_alta_variabilidad]
```

```
[34] ✓ 6.2s
```
```python
result = _time_run(problema_monedas, partida_alta_variabilidad)
print(result)

result = _time_run(problema_monedas, partida_baja_variabilidad)
print(result)
```

```
... 3.163830518722534
3.1313531398773193
```

Para *partida_baja_variabilidad* utilizamos valores entre 1 y 1M y para *partida_alta_variabilidad* utilizamos valores entre 1 y 10M, y para ambos usamos 1M de monedas, y ejecutamos cada algoritmo una única vez. Dichos ejemplos se generaron automáticamente utilizando $generador_largo_n.py$. Se puede observar que para ambos casos los tiempos de ejecución no varían significativamente.

## 2.6. Complejidad

Nuestro algoritmo recorre una vez la lista de monedas de forma iterativa, realizando en cada iteración operaciones de comparación y eliminación del primer o último elemento de la lista. Para que dichas operaciones se realicen en una complejidad O(1) utilizamos una librería llamada *deque*, la cual asegura que quitar el primer o último elemento se realice en tiempo constante.

En consecuencia, el algoritmo resulta ser de complejidad **O(n)**.

## 2.7. Ejemplos de ejecución

A continuación mostraremos algunos de los resultados obtenidos de ejemplos de ejecución para diferentes configuraciones de filas de monedas. Los mismos corresponden a tests con diversos valores de monedas.

Ejemplo 1: $[20, 30, 2, 10, 50]$

Ejemplo 2: $[10, 15, 50, 40, 60, 1000, 20, 1500, 22]$

Ejemplo 3: $[4, 2, 9, 5, 10, 1, 3, 6, 8, 7]$

| Jugador | Monedas | Total de Puntos |
|---|---|---|
| Sophia | [50, 20, 30] | 100 |
| Mateo | [10, 2] | 12 |

| Jugador | Monedas | Total de Puntos |
|---|---|---|
| Sophia | [22, 1500, 50, 1000, 60] | 2632 |
| Mateo | [10, 15, 20, 40] | 85 |

| Jugador | Monedas | Total de Puntos |
|---|---|---|
| Sophia | [7, 8, 9, 10, 6] | 40 |
| Mateo | [4, 2, 5, 1, 3] | 15 |

Se puede apreciar que a pesar de los distintos rangos de valores utilizados para las monedas, en todos los casos gana Sophia.

## 2.8. Casos de prueba

Se implementaron dos programas: el primero, llamado 'generador.py', se encarga de generar arreglos de números aleatorios y crear archivos con casos de prueba (un test por cada longitud de array); el segundo, llamado 'test_auto.py', ejecuta cada uno de los casos de prueba y resuelve el problema greedy. Ambos programas se ejecutan de la siguiente manera:

```
python3 generador.py 8 10
python3 test_auto.py
```

El programa 'generador.py' recibe dos argumentos: el primero corresponde al tamaño máximo del arreglo (por ejemplo, 8), y el segundo al valor máximo de los elementos en el arreglo (por ejemplo, 10).

A continuación, se muestra la ejecución del programa 'test_auto.py':

```
vicky@vicky-Lenovo-ideapad-330-15IKB:~/Escritorio/TDA/TP-Greedy-TDA$ python3 generador.py 8 10
vicky@vicky-Lenovo-ideapad-330-15IKB:~/Escritorio/TDA/TP-Greedy-TDA$ python3 test_auto.py
Testing test4.txt Resultado: Sophia ganó
Testing test5.txt Resultado: Sophia ganó
Testing test8.txt Resultado: Sophia ganó
Testing test1.txt Resultado: Sophia ganó
Testing test6.txt Resultado: Sophia ganó
Testing test2.txt Resultado: Sophia ganó
Testing test3.txt Resultado: Sophia ganó
Testing test7.txt Resultado: Sophia ganó
```

Como se puede observar, Sophia ganó en todos los casos de prueba.

Ahora se prueba nuevamente con arrays de 8 elementos, pero con valores en el rango de 1 a 1000, para poder apreciar que sin importar la variabilidad de los valores, Sophia gana siempre.

```
vicky@vicky-Lenovo-ideapad-330-15IKB:~/Escritorio/TDA/TP-Greedy-TDA$ python3 generador.py 8 1000
vicky@vicky-Lenovo-ideapad-330-15IKB:~/Escritorio/TDA/TP-Greedy-TDA$ python3 test_auto.py
Testing test4.txt Resultado: Sophia ganó
Testing test5.txt Resultado: Sophia ganó
Testing test8.txt Resultado: Sophia ganó
Testing test1.txt Resultado: Sophia ganó
Testing test6.txt Resultado: Sophia ganó
Testing test2.txt Resultado: Sophia ganó
Testing test3.txt Resultado: Sophia ganó
Testing test7.txt Resultado: Sophia ganó
```

De esta forma, se puede verificar de forma empírica la nula influencia de la variabilidad de los valores de las monedas en la optimalidad del algoritmo.

A continuación, se detallan las decisiones tomadas por el algoritmo paso a paso para uno de estos casos de prueba, donde el arreglo generado es: $[4, 8, 6, 1, 7]$. Sophia debe decidir si tomar la moneda del inicio o del final. Utiliza la función `max(4, 7)`, que devuelve 7 ya que es el mayor. El óptimo local en este turno es la tupla $(\text{Ganancia Total Sophia}, \text{Ganancia Total Mateo}) = (7, 0)$.

Luego, el arreglo queda $[4, 8, 6, 1]$, y es el turno de Mateo. Mateo, tratando de minimizar su ganancia, utiliza `min(4, 1)`, y selecciona la moneda 1. El nuevo óptimo local es $(7, 1)$.

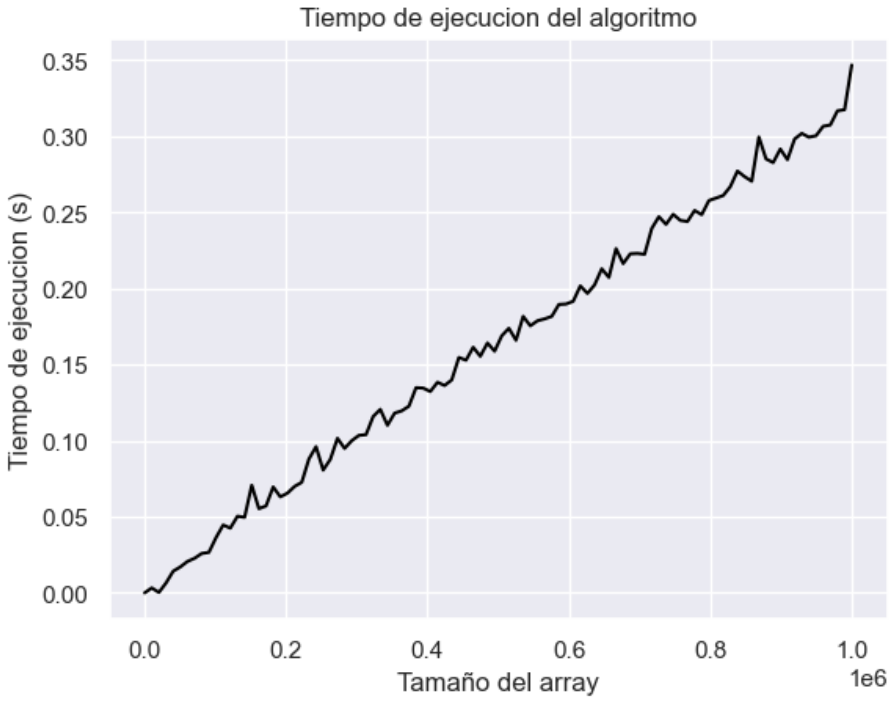
El arreglo ahora es $[4, 8, 6]$, y le toca de nuevo a Sophia, quien selecciona `max(4, 6)` $= 6$. El óptimo local se actualiza a $(13, 1)$.

Con el arreglo reducido a $[4, 8]$, Mateo selecciona la primera moneda, ya que `min(4, 8)` $= 4$. El óptimo local ahora es $(13, 5)$.

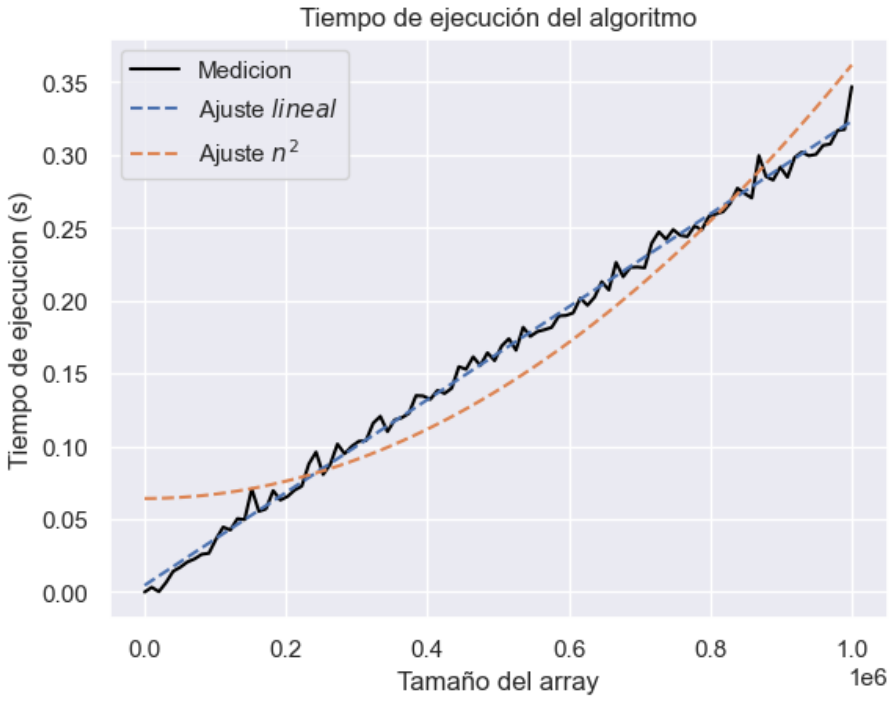
Finalmente, queda una sola moneda en el arreglo, por lo que Sophia toma el último elemento y el óptimo local se convierte en $(21, 5)$. El resultado global es $(21, 5)$. Dado que la ganancia de Sophia es 21, que es mayor que la de Mateo y Sophia gana.

## 3. Mediciones

Para estimar la complejidad de nuestro algoritmo, hemos ejecutado 100 partidas que van desde 100 hasta 1.000.000 monedas. Además, cada partida es ejecutada 10 veces para obtener una estimación más precisa de los tiempos de ejecución del programa, a partir de la media de tiempos tomados por cada partida.




Como se puede apreciar, el tiempo de ejecución del algoritmo crece de forma lineal a medida que aumentamos el tamaño del arreglo de entrada del algoritmo.




Realizamos un ajuste lineal y uno cuadrático ($n^2$). Como podemos ver, el ajuste lineal es el que mejor se aproxima a nuestra curva de complejidad, tal como lo habíamos argumentado previamente en este informe. De la misma forma, se puede observar el gran error que presenta el ajuste cuadrático ($n^2$) en comparación.


De esta forma, podemos apreciar de forma empirica que el algoritmo tiene complejidad O(n), ya que el tiempo de ejecución del algoritmo crece de forma lineal con respecto al tamaño de la entrada.

## 4. Conclusiones

El análisis del problema y la implementación del algoritmo Greedy propuesto han demostrado ser efectivos para garantizar que Sophia gane siempre. A lo largo de las pruebas realizadas, se observó que la variabilidad en los valores de las monedas no afecta el resultado final del algoritmo. Esto se debe a que la estrategia aplicada siempre selecciona la moneda de mayor valor disponible, independientemente de cómo estén distribuidos los valores restantes. Al elegir de manera óptima en cada turno, Sophia asegura una ventaja constante sobre Mateo, sin importar si las monedas tienen valores muy distintos o cercanos entre sí.

Además, la optimalidad del algoritmo queda demostrada, ya que cada decisión tomada por Sophia maximiza su ganancia inmediata, lo que le asegura la victoria en todos los casos. Aunque podrían existir otras estrategias alternativas que conduzcan a resultados similares, el enfoque Greedy aplicado es suficiente para garantizar que Sophia gane en cada partida sin dejarse vencer, cumpliendo así con el objetivo propuesto.

En cuanto a la complejidad del algoritmo, las mediciones empíricas realizadas corroboraron que su tiempo de ejecución es lineal, es decir, $O(n)$. Este comportamiento eficiente se mantuvo incluso al incrementar el tamaño de la fila de monedas, y los gráficos generados mediante la técnica de mínimos cuadrados donde confirmaron el ajuste teórico propuesto. Por lo tanto, el algoritmo no solo es óptimo en términos de garantizar la victoria de Sophia, sino también en cuanto a su complejidad temporal,

permitiendo una ejecución eficiente incluso para filas grandes de monedas.

Finalizando, el algoritmo Greedy en este caso resultó en una solución óptima. Sin embargo, es importante destacar que, en general, los algoritmos Greedy no siempre garantizan una solución óptima para todos los problemas.