

1. Análisis del problema

El enunciado plantea la siguiente situación: Sophia y Mateo juegan un juego de monedas, donde en cada turno, uno de los dos tiene que elegir una moneda, ya sea la primera o la última de la fila de n monedas. Al elegir, se remueve de la fila, y le toca luego al otro jugador, quien debe elegir otra moneda siguiendo la misma regla. Se siguen agarrando monedas hasta que no quede ninguna. Quien gane será quien tenga el mayor valor acumulado.

Nuestras consideraciones al plantear un algoritmo Greedy para resolver el problema fueron:

- Sophia siempre empieza jugando, por lo que agarra la primera moneda.
- Para que gane Sophia, el valor de las monedas acumulado siempre tiene que ser mayor al de Mateo.
- Sophia puede ver las dos monedas disponibles para agarrar y tomar una decisión basándose en el valor de la moneda.

Desarrollamos la idea de "Sophia siempre agarra la moneda más grande y Mateo la más pequeña" para que siempre gane Sophia. El algoritmo funciona de la siguiente manera:

Se tiene una fila de monedas de largo 5:

[1, 2, 4, 9, 10]

Empieza Sophia agarrando una moneda. Tiene disponibles la primera y la última, en este caso [1, 10]. La de mayor valor es la de 10. Sophia la agarra, teniendo un acumulado de 10. Mateo agarra la moneda con menor valor, acumulando 1. Ambas monedas se quitan de la fila.

Siguiente turno:

[2, 4, 9]

- Moneda que agarra Sophia: 9
- Moneda que agarra Mateo: 2
- Acumulado de Sophia: $10 + 9 = 19$
- Acumulado de Mateo: $1 + 2 = 3$

Siguiente turno:

[4]

- Moneda que agarra Sophia: 4.
- Moneda que agarra Mateo: Ninguna, ya que no hay más disponibles.
- Acumulado de Sophia: $19 + 4 = 23$.
- Acumulado de Mateo: 3.

De esta forma, se obtiene un óptimo local al elegir, de manera independiente del estado inicial, la moneda del valor más grande en cada turno. Estas decisiones nos llevan a una solución óptima global, siendo que Sophia gane.

Más adelante, se demuestra matemáticamente por qué esta solución es óptima.

2. Demostración del algoritmo

La regla Greedy tomada son "Sophia siempre agarra la moneda más grande y Mateo la más chica" para que siempre gane Sophia.

Sea (S_k, M_k) el par de monedas agarradas en el turno k . X es el número de turnos de Sophia e Y es el número de turnos de Mateo y $X + Y$ es la cantidad de monedas totales.

También sabemos que $X \geq Y$ dado que Sophia siempre empieza.

Suponemos que la cantidad de monedas es par, ya que si fuera impar, Sophia solo tendría una moneda más. Entonces para este caso suponemos $X = Y$.

Dado nuestra regla Greedy, decimos que en cada turno $S_k \geq M_k$.

$$S_1 \geq M_1 \quad S_2 \geq M_2 \quad \dots \quad S_X \geq M_Y$$

Si hacemos la sumatoria de todas las monedas de ambos, término a término $S_i \geq M_i$:

$$S_1 + S_2 \geq M_1 + M_2 \quad S_3 + S_4 \geq M_3 + M_4$$

...

$$S_{X-1} + S_X \geq M_{Y-1} + M_Y$$

Si hacemos la suma de todas las monedas:

$$S_1 + S_2 + \dots + S_X \geq M_1 + M_2 + \dots + M_Y$$

Entonces:

$$\sum_{k=1}^X S_k \geq \sum_{k=1}^Y M_k$$

Como dice el enunciado que se descarta que todas las monedas no pueden ser iguales, entonces:

$$\sum_{k=1}^X S_k > \sum_{k=1}^Y M_k$$

Probando que el algoritmo siempre llega a la solución óptima, ya que la sumatoria de los valores de las monedas de Sophia siempre va a ser mayor que la de Mateo.

3. Algoritmo Greedy: Sophia siempre gana

A continuación se muestra el algoritmo planteado para la solución del problema:

```
1 def moneda_inicial_es_mayor(monedas, inicio, fin):
2     return (monedas[inicio] >= monedas[fin])
3
4 def seleccionar_moneda(monedas, inicio, fin, jugador, puntos_sofia, puntos_mateo):
5
6     if jugador % 2 != 0:
7         if moneda_inicial_es_mayor(monedas, inicio.devolver_valor(), fin.
8             devolver_valor()):
9             puntos_sofia.sumar(monedas[inicio.devolver_valor()])
10            inicio.sumar()
11        else:
12            puntos_sofia.sumar(monedas[fin.devolver_valor()])
13            fin.sumar(-1)
14    else:
15        if moneda_inicial_es_mayor(monedas, inicio.devolver_valor(), fin.
16            devolver_valor()):
17            puntos_mateo.sumar(monedas[fin.devolver_valor()])
18            fin.sumar(-1)
19        else:
20            puntos_mateo.sumar(monedas[inicio.devolver_valor()])
21            inicio.sumar()
22
23 /S1/ def jugar(monedas):
24     inicio = EnteroMutable(0)
25     fin = EnteroMutable(len(monedas) - 1)
26     puntos_sofia = EnteroMutable(0)
27     puntos_mateo = EnteroMutable(0)
28
29     /S2/ for i in range(1, len(monedas) + 1):
30         seleccionar_moneda(monedas, inicio, fin, i, puntos_sofia, puntos_mateo)
31
32     /S3/ print(f"Puntos sofia: {puntos_sofia.devolver_valor()}")
33     print(f"Puntos mateo: {puntos_mateo.devolver_valor()}")
```

3.1. Funcionamiento del algoritmo

El juego tendrá tantos turnos como monedas existan. Por enunciado sabemos que Sophia siempre inicia jugando para sí misma, entonces, se considerará su turno cada vez que el contador marque un número impar, y para Mateo se considerará su turno cuando el contador marque un número par.

Teniendo en cuenta que las únicas monedas que pueden ser seleccionadas durante los turnos son los extremos, elegimos los mismos para realizar las comparaciones correspondientes:

Ejemplo:

$$\begin{array}{cccccccc} [5] & [X] & [X] & [X] & [X] & [X] & [X] & [9] \\ & \uparrow & & & & & & \uparrow \end{array}$$

Luego, llamamos a la función `seleccionar_moneda`, esta verifica si el número del turno es par o impar y procede a realizar la lógica correspondiente a cada jugador basado en lo establecido previamente.

Si es el turno de Sophia, siempre elegirá la moneda con mayor valor entre las disponibles. Después actualizará el inicio o el final según corresponda en la fila de monedas.

$$\begin{array}{ccc} [5] & < & [9] \\ & & \uparrow \end{array}$$

Seleccionamos el 9 para Sophia y se actualiza el final de la fila

Si es el turno de Mateo, siempre elegirá la moneda con menor valor entre las disponibles. Después actualizará el inicio o el final según corresponda en la fila de monedas tal cual como se hizo en el turno de Sophia y para ambos se va actualizando la cantidad de puntos que acumulan al seleccionar las monedas

Seleccionamos el 5 para Mateo y se actualiza el inicio de la fila.

Finalmente, repetimos este proceso hasta que no haya más monedas disponibles.

Observación: Para el análisis, vamos a aplicar las reglas del modelo de análisis computacional (bibliografía Méndez). Y no vamos a considerar el procesamiento de archivos, ni todo lo que esté por fuera de la función.

Podemos dividir a esta función en un bloque de tres instrucciones ejecutables: $S1$, $S2$ y $S3$, donde $S1$ empieza en la línea 21 hasta la 25 inclusive, $S2$ empieza desde 28 hasta 29 inclusive y los $S3$ conforman las dos prints. Vemos que se tiene una secuencia de instrucciones. Por ende, la regla de cálculo del tiempo de ejecución de una secuencia, se basa en aplicar el teorema de aditividad:

1. En $S1$ hay instrucciones que ejecutan 3 asignaciones de variables locales, y cada asignación cuesta $O(1)$, luego las 3 asignaciones van a tener de tiempo de ejecución $O(1)$.
2. $S3$ tiene instrucciones que ejecutan las funciones `print` que también tiene tiempo $O(1)$.
3. Solo falta determinar $O(S2)$, para ello se estudiará el peor de los casos posibles que puede ocurrir en la iteración $S2$, esto es (el caso normal, justo en este caso) cuando:

Para ello hay que determinar la cantidad de iteraciones que se hace.

Vale analizar la instrucción que llama a `seleccionar_moneda(...)`. `seleccionar_moneda(...)` tiene instrucciones *if*, *else*, se llama a funciones de una clase (`sumar()`, `devolver_valor()`), se usa *return*, y todas estas instrucciones tienen tiempos de ejecución son $O(1)$. Luego ejecutar la función completa va a tardar $O(1)$.

Haciendo una tabla de iteraciones:

Vemos que la cantidad de iteraciones para $i = n$ (peor caso) es n , por ende el tiempo de ejecución de $S3$ es $O(n)$.

Finalmente, se obtiene: $O(S1, S2, S3) = \max(O(1), O(n), O(1))$ y por todo lo concluido, el orden de complejidad o tiempo de ejecución de $jugar(n)$ es $O(n)$.

Un caso no menos importante a observar es que para estudiar la optimalidad y los tiempos del algoritmo, jamás se tuvo en cuenta la variabilidad de los valores de las diferentes monedas, y podemos concluir que esto no afecta.

i	iteraciones
1	1
2	2
3	3
...n	...n

4. Ejemplos de ejecución

Agregaremos 2 casos particulares más, aparte de los que ya nos proporcionaron en la cátedra y mostraremos las soluciones arrojadas, comprobando que obtendremos la solución óptima.

4.1. Ejemplo 1

Le pasamos al programa la siguiente lista:

[10, 30, 40, 4, 2, 9, 10]

y realizamos el seguimiento de la ejecución.

Primera iteración

$$\begin{array}{ccccccc} [10] & [30] & [40] & [4] & [2] & [9] & [10] \\ \uparrow & & & & & & \uparrow \end{array}$$

Como ambos valores son iguales, la primera va a Sophia y la segunda a Mateo.

Acumulado: Sophia: 10 Mateo: 10

Segunda iteración

$$\begin{array}{ccccc} [30] & [40] & [4] & [2] & [9] \\ \uparrow & & & & \uparrow \end{array}$$

Acumulado: Sophia: 40 Mateo: 19

Tercera iteración

$$\begin{array}{ccc} [40] & [4] & [2] \\ \uparrow & & \uparrow \end{array}$$

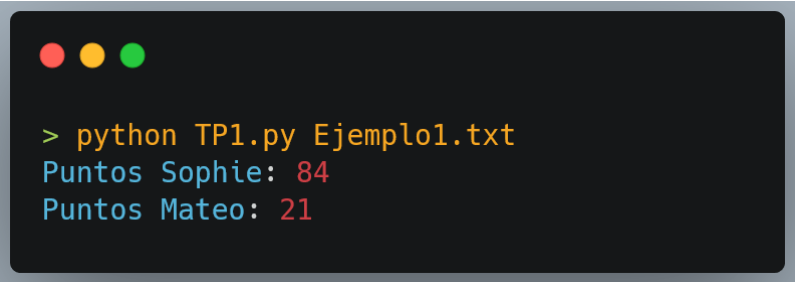
Acumulado: Sophia: 80 Mateo: 21

Cuarta iteración

$$\begin{array}{c} [4] \\ \uparrow \end{array}$$

Acumulado: Sophia: 84 Mateo: 21

Coincidiendo con la salida del programa:



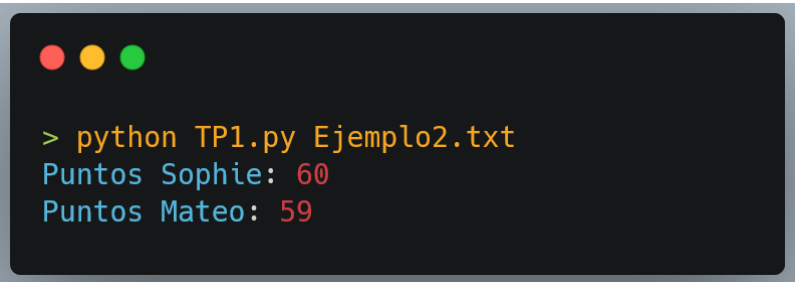
```
> python TP1.py Ejemplo1.txt
Puntos Sophie: 84
Puntos Mateo: 21
```

4.2. Ejemplo 2

Le pasamos al programa la siguiente lista:

[10, 10, 10, 10, 10, 10, 10, 9, 10, 10, 10, 10]

Y la salida del programa fue:



```
> python TP1.py Ejemplo2.txt
Puntos Sophie: 60
Puntos Mateo: 59
```

4.3. Caso 25.txt

Este ejemplo es uno de los dos provistos en el enunciado. La salida del programa fue:



```
> python TP1.py 25.txt
Puntos Sophie: 9635
Puntos Mateo: 3577
```

4.4. Caso 10000.txt

Este ejemplo es el segundo de los dos provistos en el enunciado. La salida del programa fue:

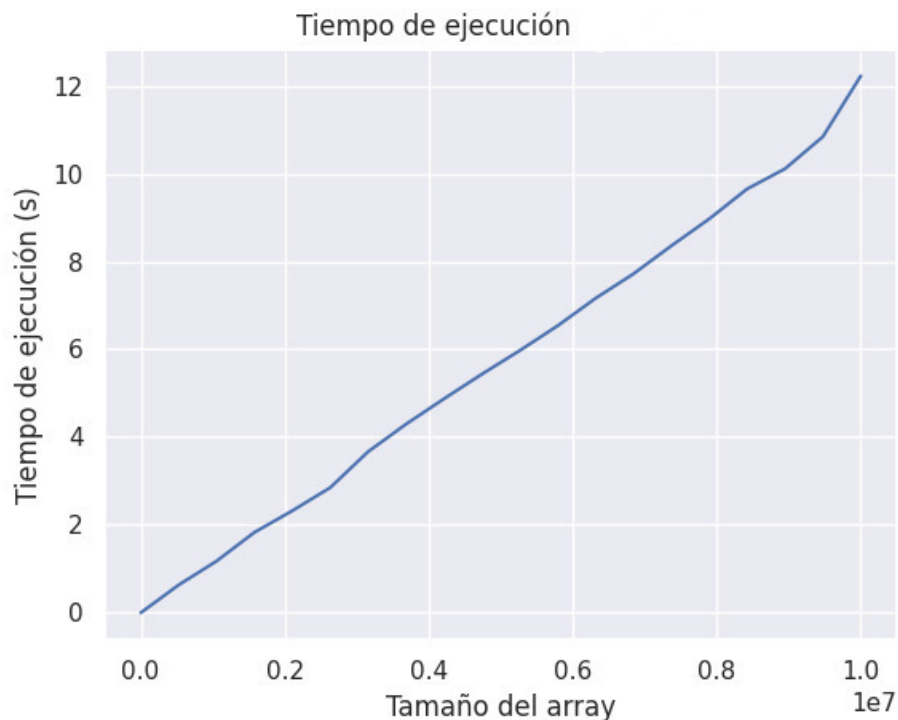
```
> python TP1.py 10000.txt
Puntos Sophie: 3550552
Puntos Mateo: 1484076
```

5. Análisis de la complejidad por cuadrados mínimos

Para este caso, vamos a corroborar la complejidad calculada de manera teórica haciendo mediciones para el algoritmo Greedy de la función jugar, para posteriormente realizar un ajuste por cuadrados mínimos.

```
1 def get_random_array(size: int):
2     return np.random.randint(0, 100.000, size)
3
4 x = np.linspace(100, 10_000_000, 20).astype(int)
5
6 results = time_algorithm(jugar, x, lambda s: [get_random_array(s)])
```

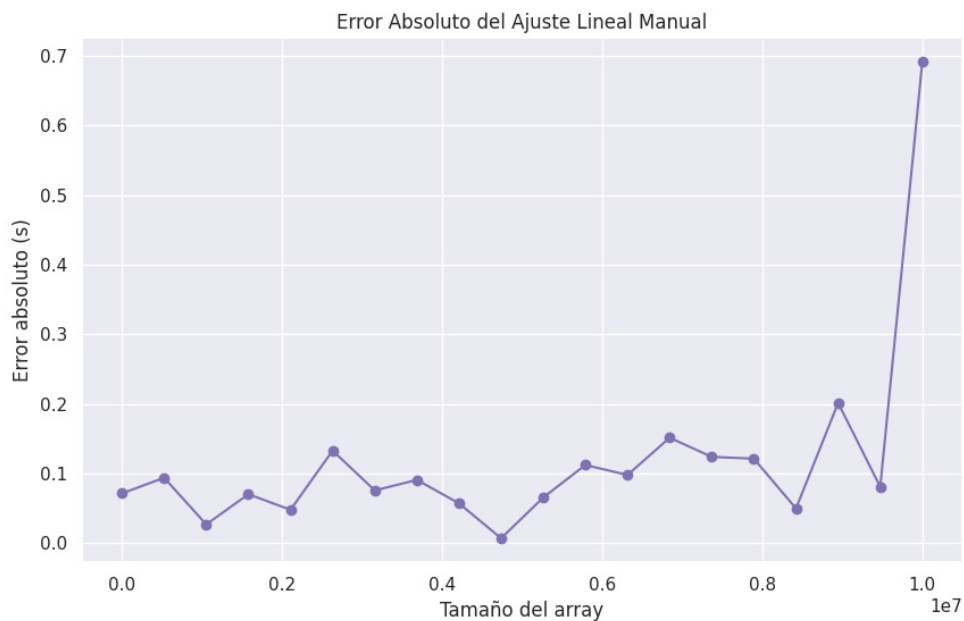
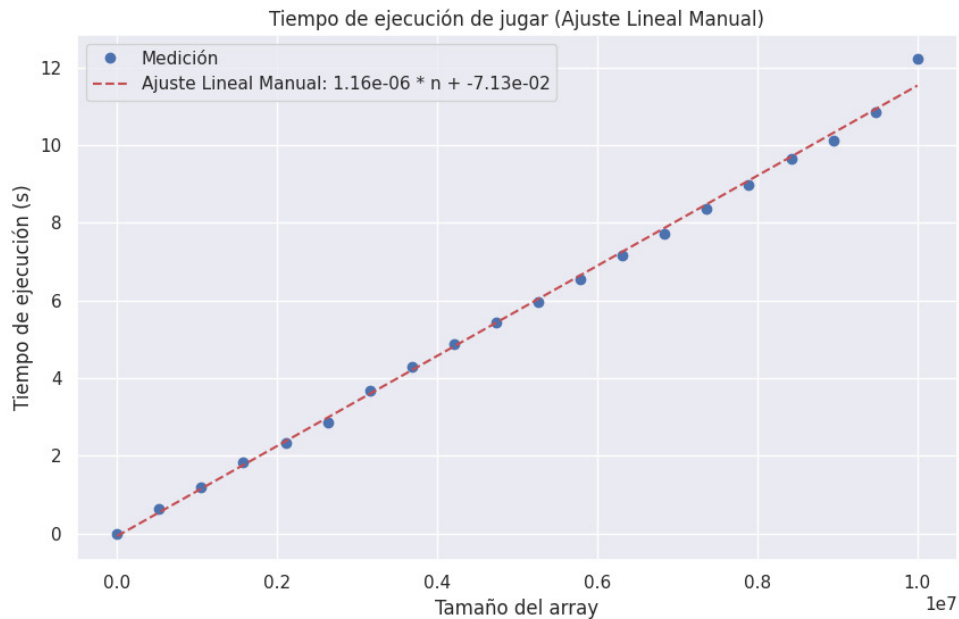
Siendo x los valores de las abscisas, y el de las ordenadas: $[results[i] \text{ for } i \text{ in } x]$, se obtuvo:



Calculando el ajuste por cuadrados mínimos (ajustamos a una recta $c_1 * n + 1$) se obtuvo el siguiente error:

Y para finalizar, graficamos el ajuste y el error para cada tamaño del array:

Método Manual: $c_1 = 1.16e-06$, $c_2 = -7.13e-02$
Método Manual: Error cuadrático total = $6.65e-01$



Se puede apreciar que para todos los tamaños del array el error de nuestro ajuste es bastante pequeño, lo cual nos lleva a concluir que nuestro algoritmo se comporta como $O(n)$.

6. Bibliografía utilizada

1. https://github.com/algoritmos-rw/tda_ejemplos/tree/main/analisis_complejidad
2. Apuntes Méndez: ejerResueltosAnálisisAlgoritmos.pdf y Análisis de Algoritmos.pdf