FACULTAD DE INGENIERIA
Universidad de Buenos Aires

TP1: Algoritmos Greedy en la Nación del Fuego

[75.29] Teoría de Algoritmos
Primer Cuatrimestre de 2024

Integrantes

Alumno	Padrón
FUENTES, Azul	102184
GALIÁN, Tomás Ezequiel	104354
SANTANDER, Valentín	106387

Índice

1. Análisis del problema **2**

2. Solución propuesta **2**
2.1. Construcción conceptual de la solución 2
2.1.1. Ordenamiento por peso de mayor a menor 3
2.1.2. Ordenamiento por tiempo de menor a mayor 3
2.1.3. Ordenamiento por ratio teniendo en cuenta ambas variables 3
2.2. Implementación en código de la solución 3
2.3. Complejidad del Algoritmo 4

3. ¿La solución propuesta es óptima en todos los casos? **4**
3.1. Demostración por inversiones 4
3.2. Variabilidad de valores 5

4. Ejemplos **6**
4.1. Ejemplo con criterio por tiempo como peor solución 6
4.2. Ejemplo con criterio por peso como peor solución 7
4.3. Ejemplo con solución óptima usando criterio por peso 8
4.4. Ejemplo con solución óptima usando criterio por tiempo 9
4.5. Ejemplo con todos los criterios obteniendo solución óptima 9
4.6. Conclusiones de los ejemplos 10

5. Mediciones Temporales **10**
5.1. Medición con conjuntos de datos aleatorios 10
5.2. Medición con conjuntos de datos proporcionados por el curso 11
5.3. Medición de escala del algoritmo 11

6. Conclusiones **12**

1. Análisis del problema

Para el problema planteado se cuenta con una lista de distitntas batallas en un orden determinado. Una batalla a_i se define como una tupla de dos valores.

$$a_i = (t_i, b_i)$$

Donde:

- i: Es el índice de la batalla dentro de la lista ordenada de batallas. Es decir, si a_i es la i-ésima batalla en la lista, entonces i es igual a su posición dentro de esa lista. Los índices van de 1 a n
- t_i: Es el tiempo que necesita el ejército de la Nación del Fuego para ganar la i-ésima batalla.
- b_i: Es la importancia que tiene la i-ésima batalla para la población de la Nación del Fuego.

Siendo B la lista de batallas de tamaño $n \in \mathbb{N}$, $n > 0$, se cumple que:

$$\forall i \in \mathbb{N}, 1 \leq i \leq n, t_i \in \mathbb{N}, b_i \in \mathbb{N} : a_i = (t_i, b_i) \in B \implies t_i > 0 \wedge b_i > 0$$

Cabe destacar que cada batalla comienza ni bien termina la anterior. Por lo tanto el tiempo de finalización de cada batalla se calcula de la siguiente manera:

$$\begin{cases} F_1 = t_1 \\ F_i = F_{i-1} + t_i, \quad \forall i \in \mathbb{N} : 1 < i \leq n \end{cases}$$

Teniendo esto en cuenta, se busca encontrar el orden de las batallas tal que se logre minimizar la suma ponderada de los tiempos de finalización:

$$\sum_{i=1}^{n} b_i \cdot F_i$$

Siendo:

- b_i: Peso que define la importancia de la i-ésima batalla.
- F_i: Tiempo de finalización de la i-ésima batalla.

Al tener dos variables en juego, es necesario entender la implicancia de cada una. La población de la Nación del Fuego valora más que se peleen primero las batallas de mayor importancia, o bien mayor peso. En cuanto a los tiempos de finalización, se sabe que a medida que las batallas transcurren, el tiempo de finalización de las mismas aumenta, por lo tanto la suma ponderada será mayor.

Es por eso que al momento de construir la solución, vamos a querer que las batallas de mayor peso se encuentren lo más pronto posible para multiplicar al F_i de menor valor. Sin embargo, utilizar solo este criterio puede no ser suficiente, ya que dependiendo del valor t_i de cada batalla también puede variar la conveniencia del orden.

2. Solución propuesta

2.1. Construcción conceptual de la solución

Para la implementación de la solución se propone utilizar un algorítmo Greedy. Un algorítmo Greedy es un algorítmo que busca tomar la mejor decisión (o la más óptima) local en cada paso. Es decir, el algorítmo se fija donde esta parado en ese instante y cuál es la mejor decisión que puede tomar, con el fin de que esta decisión local lleve a una solución global óptima.

Para facilitar la selección de la mejor decisión local, optamos por ordenar las batallas disponibles, buscando así ir ejecutando aquellas que más nos convenga.

Para eso vamos a ver algunos criterios sobre los cuáles podríamos ordenar estas batallas y ver cual es el óptimo

2.1.1. Ordenamiento por peso de mayor a menor

Este caso consiste en enfocarse nada más en el peso de las batallas. Buscamos ordenar las batallas de mayor peso a menor peso priorizando de esta forma las que son mas importantes primero. Este ordenamiento es crucial, ya que evita que las batallas con mayor peso se retrasen en su ejecución. De esta manera, prevenimos que la suma ponderada aumente, dado que las batallas más importantes se completan al principio.

2.1.2. Ordenamiento por tiempo de menor a mayor

Otro ordenamiento consiste en centrarse en el tiempo requerido para cada batalla. Queremos ordenar las batallas de menor a mayor tiempo, lo que nos permite ejecutar primero aquellas que requieren menos tiempo. Este ordenamiento es crucial para minimizar la suma ponderada, ya que evita que las batallas con menor tiempo se ejecuten a lo último con mucho tiempo acumulado.

2.1.3. Ordenamiento por ratio teniendo en cuenta ambas variables

Por último, el mejor algorítmo que se adecua al problema y que es el óptimo consiste en tener en cuenta ambas variables, tiempo y peso. Buscamos obtener y ordenar a partir de una relación entre ambas y de esta forma obtener la mejor solución.

Esta solución consiste en obtener un ratio que se calcula de la siguiente forma

$$\forall i \in \mathbb{N}, 1 \leq i \leq n \ : r_i = \frac{b_i}{t_i}$$

Ordenando cada batalla de forma descendente (de mayor ratio a menor ratio) se logra el mejor ordenamiento posible para la ejecución de las batallas.

Este ratio ayuda a tener en cuenta ambos criterios hablados anteriormente, donde a mayor peso y menor tiempo el ratio aumenta y donde a menor peso y mayor tiempo el ratio disminuye. De esta forma tenemos en cuenta ambas variables y llegamos a una solución óptima.

2.2. Implementación en código de la solución

Se decidió utilizar Python como lenguage de programación para implementar la solución. El algoritmo propuesto es el siguiente:

```
1  def obtener_ratio_batalla(batalla):
2      tiempo_batalla, peso_batalla = batalla
3      return peso_batalla / tiempo_batalla
4
5  def obtener_suma_ponderada(batallas):
6      regla = lambda batalla: obtener_ratio_batalla(batalla)
7      batallas_ordenadas = sorted(batallas, key=regla, reverse=True)
8
9      suma_ponderada = 0
10     suma_parcial = 0
11
12     for batalla in batallas_ordenadas:
13         tiempo_batalla,peso_batalla = batalla
14         suma_parcial += tiempo_batalla
15         suma_ponderada += suma_parcial*peso_batalla
16
17     return suma_ponderada
```

Se ordenan entonces las batallas según relación entre su valor y el tiempo que tardan $\frac{b_i}{t_i}$ de mayor a menor:

```
1  def obtener_ratio_batalla(batalla):
2      tiempo_batalla, peso_batalla = batalla
3      return peso_batalla / tiempo_batalla
4
5  def obtener_suma_ponderada(batallas):
6      regla = lambda batalla: obtener_ratio_batalla(batalla)
7      batallas_ordenadas = sorted(batallas, key=regla, reverse=True)
```

Luego, para obtener la suma ponderada de los tiempos de finalización, simplemente se recorre el arreglo ordenado y se realiza la suma correspondiente:

```
9      suma_ponderada = 0
10     suma_parcial = 0
11
12     for batalla in batallas_ordenadas:
13         tiempo_batalla,peso_batalla = batalla
14         suma_parcial += tiempo_batalla
15         suma_ponderada += suma_parcial*peso_batalla
16
17     return suma_ponderada
```

2.3. Complejidad del Algoritmo

La complejidad temporal del algoritmo resulta ser $\mathcal{O}(n \cdot \log(n))$, ya que realiza un ordenamiento que se asume en tiempo $\mathcal{O}(n \cdot \log(n))$ para la función `sorted()` de Python, y luego recorre linealmente este arreglo para encontrar la suma ponderada de los tiempos de las batallas.

3. ¿La solución propuesta es óptima en todos los casos?

En la siguiente sección se intentará demostrar por qué la solución planteada es óptima, a través de la demostración por inversiones.

3.1. Demostración por inversiones

En primer lugar, se utilizará la definición de ratio r_i descripta anteriormente para cada batalla de una lista B de tamaño n

$$\forall i \in \mathbb{N}, 1 \leq i \leq n : r_i = \frac{b_i}{t_i}$$

Se considera que si existen eventos i, j tal que $r_i = r_j$ entonces el orden en el que se encuentren, siempre y cuando sea de manera consecutiva, no afectará a la solución final.

Recordamos que la solución planteada considera que para que para minimizar la suma ponderada de los tiempos de finalización es necesario que se cumpla la siguiente relación:

$$r_1 \leq \cdots \leq r_n$$

Decimos que una solución posee una inversión si r_i se encuentra posicionada antes en la sumatoria que r_j, y $r_j > r_i$. Las soluciones que no poseen inversiones (ni tiempo muerto) tendrán la misma suma ponderada de los tiempos de finalización.

Comenzaremos por considerar a la solución óptima γ, modificando la misma (conservando su condición de óptima) hasta llegar a una alternativa que no posea inversiones.

Si γ tiene inversiones, entonces existen i, j tal que j está directamente después que i en la sumatoria, y $r_j > r_i$. Suponiendo esa como la única inversión posible, haremos que γ no tenga

inversiones intercambiando el orden de r_j con r_i. Ahora lo que resta probar es que la suma ponderada de los tiempos de finalización con una inversión menos, es menor que γ.

Teniendo que:

$$\frac{b_j}{t_j} = r_j > r_i = \frac{b_i}{t_i} \tag{1}$$

Se prueba que:

$$b_i \cdot t_i + b_j \cdot (t_i + t_j) > b_j \cdot t_j + b_i \cdot (t_j + t_i) \tag{2}$$

Por el absurdo:

$$b_i \cdot t_i + b_j \cdot (t_i + t_j) < b_j \cdot t_j + b_i \cdot (t_j + t_i) \tag{3}$$

$$b_i \cdot t_i + b_j \cdot t_i + b_j \cdot t_j < b_j \cdot t_j + b_i \cdot t_j + b_i \cdot t_i)$$

$$b_j \cdot t_i < b_i \cdot t_j$$

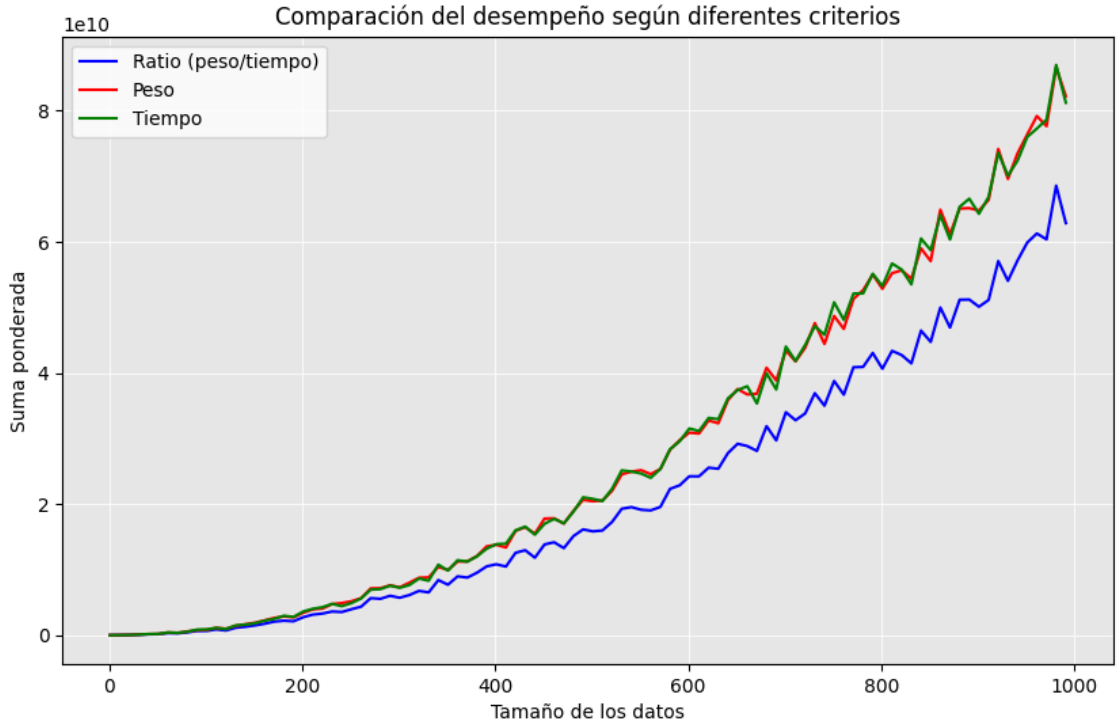
$$\frac{b_j}{t_j} < \frac{b_i}{t_i} \implies \text{ABSURDO pues contradice (1)}$$

Por lo tanto, la solución sin inversiónes es la óptima.

3.2. Variabilidad de valores

A partir de la demostración anterior, se puede observar que independientemente de los valores de b_i y t_i, mientras se siga el algoritmo Greedy que ordena las batallas usando el ratio r_i, la solución siempre será óptima. Por lo tanto, no es necesario realizar un análisis en profundidad de la variabilidad de los datos.

Sin embargo graficando la suma ponderada para listas de batallas de distintos tamaños con valores aleatorios de tiempo y peso se pueden inferir ciertas conclusiones. Se puede notar que los criterios por peso y por tiempo tienen resultados similares pero ninguno es mejor que el otro. Sin embargo el criterio por ratio siempre es menor que los otros dos. Esta idea se verá más en detalle en la próxima sección.



4. Ejemplos

Como se ya se ha demostrado en la sección 3, la solución que obtiene el algoritmo es óptima a pesar de utilizar una estrategia Greedy. En esta sección se mostrarán ejemplos de distintos conjuntos de batallas. Para cada uno se realizará el cálculo para cada combinación posible. De esa manera se podrá verificar que la solución propuesta por el algoritmo especificado en la sección 2.1.3 es óptima.

Además se utilizarán los algoritmos alternativos que fueron discutidos en las secciones 2.1.1 y 2.1.2 antes de llegar a la solución final. Estos resultados deberían ser iguales o mayores que la solución óptima.

4.1. Ejemplo con criterio por tiempo como peor solución

Dada la lista de batallas [q_1, q_2,q_3] donde:

- q_1 = (2,5)
- q_2 = (3,9)
- q_3 = (4,11)

Se procede a calcular la suma con los distintos criterios

Solución con criterio por peso

Los **pesos** de cada batalla son:

- $q_1 \rightarrow 5$
- $q_2 \rightarrow 9$
- $q_3 \rightarrow 11$

Por lo tanto si ordenamos por peso de mayor a menor, el orden de las batallas es $[q_3, q_2, q_1]$. En este caso se cumple que:

- $a_1 = q_2$
- $a_2 = q_3$
- $a_3 = q_1$

Calculo la suma ponderada para $[a_1, a_2, a_3]$

$$\sum_{i=1}^{n} b_i \cdot F_i = 11 \cdot 4 + 9 \cdot (4+3) + 5 \cdot (4+3+2) = 152$$

Solución con criterio por tiempo

Los **tiempos** de duración de cada batalla son:

- $q_1 \rightarrow 2$
- $q_2 \rightarrow 3$
- $q_3 \rightarrow 4$

Por lo tanto si ordenamos por tiempo de menor a mayor, el orden de las batallas es $[q_1, q_2, q_3]$. En este caso se cumple que:

- $a_1 = q_1$
- $a_2 = q_2$
- $a_3 = q_3$

Calculo la suma ponderada para $[a_1, a_2, a_3]$

$$\sum_{i=1}^{n} b_i \cdot F_i = 5 \cdot 2 + 9 \cdot (2+3) + 11 \cdot (2+3+4) = 154$$

Solución con criterio por ratio

Los **ratios** de cada batalla son:

- $r_1 = \frac{5}{2} = 2.5$
- $r_2 = \frac{9}{3} = 3$
- $r_3 = \frac{11}{4} = 2.75$

Por lo tanto si ordenamos por ratio de mayor a menor, el orden de las batallas es $[q_2, q_3, q_1]$. En este caso se cumple que:

- $a_1 = q_2$
- $a_2 = q_3$
- $a_3 = q_1$

Calculo la suma ponderada para $[a_1, a_2, a_3]$

$$\sum_{i=1}^{n} b_i \cdot F_i = 9 \cdot 3 + 11 \cdot (3+4) + 5 \cdot (3+4+2) = 149$$

A partir de estos resultados podemos ver que en este caso puntual el criterio de ordenamiento por tiempos fue el peor. No obstante, aunque el criterio de ordenanamiento por peso tuvo mejor desempeño, no igualó el resultado del ordenamiento por ratio, que se sabe que es el óptimo.

4.2. Ejemplo con criterio por peso como peor solución

Dada la lista de batallas [q_1, q_2,q_3] donde:

- q_1 = (5,2)
- q_2 = (9,3)
- q_3 = (11,4)

Se procede a calcular la suma con los distintos criterios

Solución con criterio por peso

Procediendo análogamente al ejemplo anterior, calculo la suma ponderada para las batallas ordenadas por peso de forma descendente.

$$\sum_{i=1}^{n} b_i \cdot F_i = 4 \cdot 11 + 3 \cdot (11+9) + 2 \cdot (11+9+5) = 154$$

Solución con criterio por tiempo

Procediendo análogamente al ejemplo anterior, calculo la suma ponderada para las batallas ordenadas por tiempo de forma ascendente

$$\sum_{i=1}^{n} b_i \cdot F_i = 2 \cdot 5 + 3 \cdot (5 + 9) + 4 \cdot (5 + 9 + 11) = 152$$

Solución con criterio por ratio

Procediendo análogamente al ejemplo anterior, calculo la suma ponderada para las batallas ordenadas por ratio de forma descendente

$$\sum_{i=1}^{n} b_i \cdot F_i = 2 \cdot 5 + 4 \cdot (5 + 11) + 9 \cdot (5 + 11 + 9) = 149$$

A partir de estos resultados podemos ver que en este caso puntual el criterio de ordenamiento por peso fue el peor. Pero el criterio de ordenamiento por tiempo a su vez fue peor que el criterio por ratio.

4.3. Ejemplo con solución óptima usando criterio por peso

Dada la lista de batallas [q_1, q_2,q_3] donde:

- q_1 = (2,5)
- q_2 = (3,8)
- q_3 = (4,11)

Se procede a calcular la suma con los distintos criterios análogamente a los ejemplos anteriores.

Solución con criterio por peso

$$\sum_{i=1}^{n} b_i \cdot F_i = 11 \cdot 4 + 8 \cdot (4 + 3) + 5 \cdot (4 + 3 + 2) = 145$$

Solución con criterio por tiempo

$$\sum_{i=1}^{n} b_i \cdot F_i = 5 \cdot 2 + 8 \cdot (2 + 3) + 11 \cdot (2 + 3 + 4) = 149$$

Solución con criterio por ratio

$$\sum_{i=1}^{n} b_i \cdot F_i = 11 \cdot 4 + 8 \cdot (4 + 3) + 5 \cdot (4 + 3 + 2) = 145$$

A partir del ejemplo, podemos inferir que el criterio por peso puede igualar a la solución óptima teniendo un mejor desempeño que el criterio por tiempo.

4.4. Ejemplo con solución óptima usando criterio por tiempo

Dada la lista de batallas [q_1, q_2,q_3] donde:

- q_1 = (5,2)
- q_2 = (8,3)
- q_3 = (11,4)

Se procede a calcular la suma con los distintos criterios análogamente a los ejemplos anteriores.

Solución con criterio por peso

$$\sum_{i=1}^{n} b_i \cdot F_i = 4 \cdot 11 + 3 \cdot (11 + 8) + 2 \cdot (11 + 8 + 5) = 149$$

Solución con criterio por tiempo

$$\sum_{i=1}^{n} b_i \cdot F_i = 2 \cdot 5 + 3 \cdot (5 + 8) + 4 \cdot (5 + 8 + 11) = 145$$

Solución con criterio por ratio

$$\sum_{i=1}^{n} b_i \cdot F_i = 2 \cdot 5 + 3 \cdot (5 + 8) + 4 \cdot (5 + 8 + 11) = 145$$

A partir del ejemplo, podemos inferir que el criterio por tiempo puede igualar a la solución óptima teniendo un mejor desempeño que el criterio por peso.

4.5. Ejemplo con todos los criterios obteniendo solución óptima

Dada la lista de batallas [q_1, q_2,q_3] donde:

- q_1 = (2,3)
- q_2 = (1,4)
- q_3 = (2,2)

Se procede a calcular la suma con los distintos criterios análogamente a los ejemplos anteriores.

Solución con criterio por peso

$$\sum_{i=1}^{n} b_i \cdot F_i = 4 \cdot 1 + 3 \cdot (1 + 2) + 2 \cdot (1 + 2 + 2) = 23$$

Solución con criterio por tiempo

$$\sum_{i=1}^{n} b_i \cdot F_i = 4 \cdot 1 + 3 \cdot (1 + 2) + 2 \cdot (1 + 2 + 2) = 23$$

Solución con criterio por ratio

$$\sum_{i=1}^{n} b_i \cdot F_i = 4 \cdot 1 + 3 \cdot (1+2) + 2 \cdot (1+2+2) = 23$$

A partir de este ejemplo, podemos inferir que puede darse el caso en el que todos los criterios obtengan la solución óptima simultáneamente

4.6. Conclusiones de los ejemplos

Teniendo en cuenta los resultados de los ejemplos podemos observar que los criterios por tiempo y por peso no son comparables pues existe al menos un caso en el que uno se desempeña mejor que el otro.

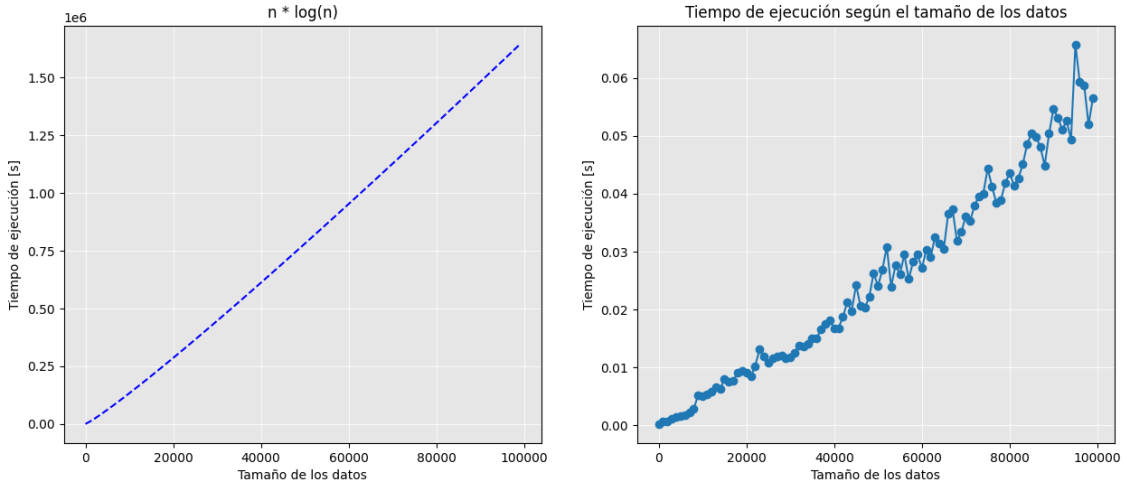
Sin embargo, como se ha demostrado anteriormente, el criterio por ratio nunca puede ser superado por los otros dos. Puede ocurrir que alguno de los criterios o ambos criterios obtengan también la solución óptima, pero nunca pueden dar un resultado menor.

5. Mediciones Temporales

Para corroborar la complejidad teórica vamos a estar realizando 3 tipos de mediciones sobre distintos bancos de datos de diferentes tamaños para poder ver como se comporta el algorítmo y su tiempo de ejecución.

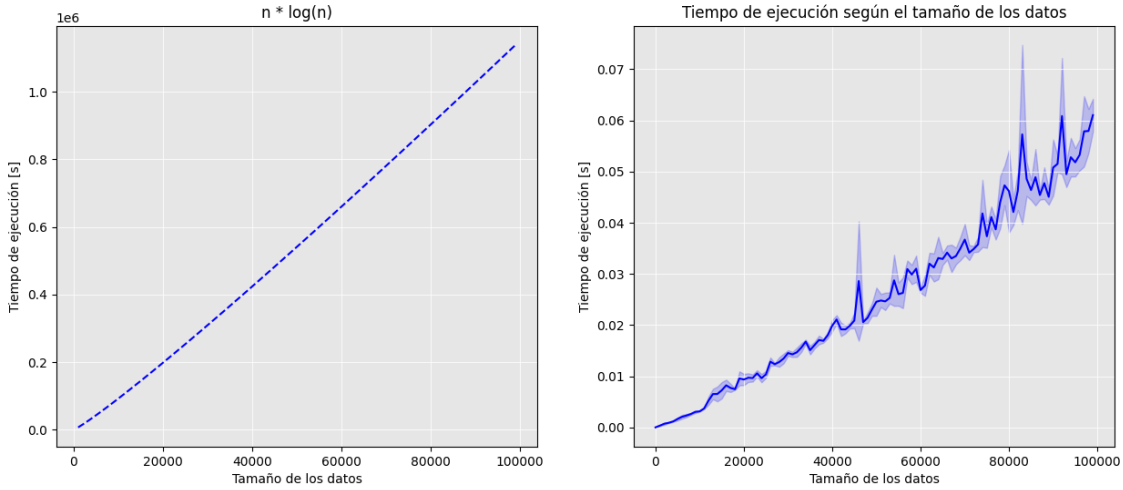
5.1. Medición con conjuntos de datos aleatorios

Para esta medición usamos tamaños de datos que van desde 0 hasta 100000 de a pasos de a 1000. Para cada tamaño de datos se crea un set de batallas, cada una con su tiempo y su peso de forma aleatoria. El algoritmo se ejecuta 5 veces para poder obtener varios tiempos de ejecución y quedarse con el promedio. Luego de eso se grafica para cada tamaño de puntos cuánto tiempo tardo y se obitene lo siguiente:



El gráfico de la izquierda ayuda a comparar ambos casos y entender como es el compartamiento del algoritmo a comparación de la cota.

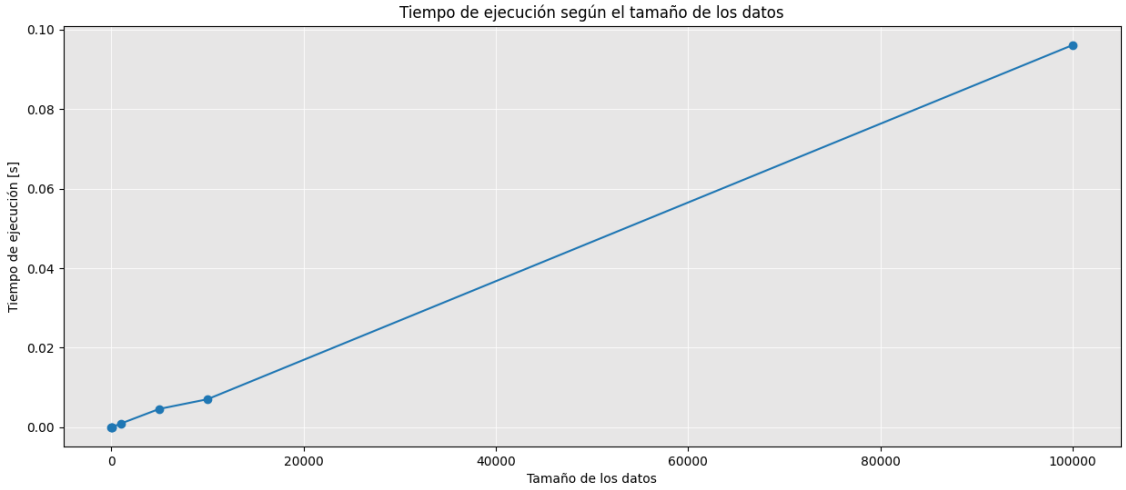
También podemos ver el mismo experimento pero observando ahora la desviación estándar que tenemos entre todas las mediciones tomadas para un mismo volumen de datos.



Podemos ver como comienza con un comportamiento lineal al principio pero que luego, a medida que el número de elementos comienza a aumentar se va ajustando a la complejidad linear-logaritmica.

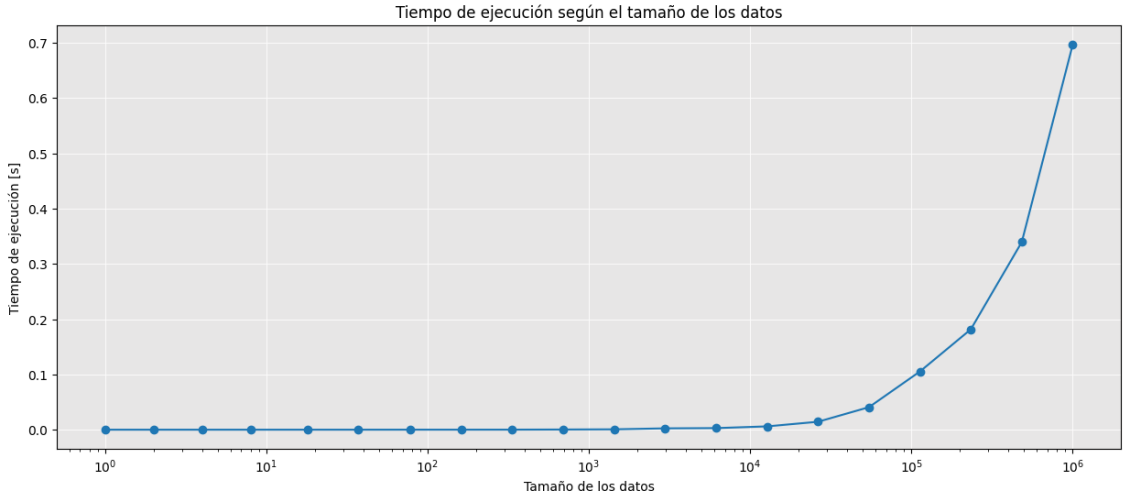
5.2. Medición con conjuntos de datos proporcionados por el curso

Para la segunda medición usamos los ejemplos que nos proporciona el curso. Nuevamente para cada uno de los casos se realiza varias veces la ejecución del algorítmo para tener un tiempo promedio. En este caso se obtiene la siguiente medición:



5.3. Medición de escala del algoritmo

En esta última medición se realiza algo pareceido a la primera. En este caso la muestra de datos consiste en una secuencia de 20 puntos que aumentan de forma exponencial comenzando en 10^0 y terminando en 10^6. Este gráfico tiene una escala logarítmica sobre su eje x que ayuda a visualizar el crecimiento del tiempo de forma $O(n.log(n))$ a medida que se tiene un aumento del volumen de datos de entrada.



Para estas mediciones se puede observar que a medida que el tamaño de los datos va aumentando el tiempo de ejecución aumenta siguiendo una curva que se parece a una función logarítmica.

En la última medición se ve de forma un poco más clara como los puntos del gráfico siguen una tendencia similar a la complejidad del algorítmo que es de $\mathcal{O}(n \cdot \log(n))$

6. Conclusiones

En conclusión, el trabajo práctico demuestra que en ciertos casos, el uso de un algoritmo Greedy puede conducir a resultados óptimos en la resolución de problemas. Además, cuando es posible demostrar que un algoritmo Greedy siempre produce la mejor solución, su implementación resulta más simple y eficiente en comparación con otras estrategias. Esto se evidencia tanto en el análisis teórico como en las mediciones temporales, lo que respalda su utilidad práctica. En resumen, el enfoque Greedy puede ser una opción poderosa y efectiva para resolver problemas con determinadas características.