Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

Trabajo Práctico 1
Algoritmos Greedy en la Nación del Fuego

18 de septiembre de 2025

Tomas Goncalves Rei
111405

Valentín Abaca
112088

Lucia De Sequera Nebreda
INT01

Alexis Maximiliano Torrez Vargas
111449

Índice

1. Introducción

1.1. Enunciado

Es el año 10 AG, y somos asesores del Señor del Fuego (líder supremo de la Nación del Fuego). El Señor del Fuego cuenta con un ejército de Maestros Fuego, muy temidos en el mundo. Tiene varias batallas con las cuales lidiar: una contra el Templo Aire del Este, otra en la Tribu del Agua del Norte, otra en la Isla de Kyoshi, una muy importante en Ba Sing Se (capital del Reino de la Tierra), y muchas otras más. Sabemos cuánto tiempo necesita el ejército para ganar cada una de las batallas (t_i). El ejército ataca todo junto, no puede ni conviene que se separen en grupos. Es decir, no participan de más de una batalla en simultáneo.

La felicidad que produce saber que se logró una victoria depende del momento en el que ésta se obtenga (es decir, que la batalla termine). Es por esto que podemos definir a F_i como el momento en el que se termina la batalla i. Si la primera batalla es la j, entonces $F_j = t_j$; en cambio, si la batalla j se realiza justo después de la batalla i, entonces $F_j = F_i + t_j$.

Además del tiempo que consume cada batalla, sabemos que al Señor del Fuego no le da lo mismo el orden en el que se realizan, porque comunicar la victoria a su nación en diferentes batallas genera menos impacto si pasa mucho tiempo. Además, cada batalla tiene una importancia diferente. Vamos a definir que tenemos un peso b_i que nos define cuán importante es una batalla.

Dadas estas características, se quiere buscar tener el orden de las batallas tal que se logre minimizar la suma ponderada de los tiempos de finalización:

$$\text{mín} \sum_{i=1}^{n} b_i \, F_i.$$

El Señor del Fuego nos pide diseñar un algoritmo que determine aquel orden de las batallas que logre minimizar dicha suma ponderada.

1.2. Análisis

Tenemos n batallas, y sabemos el tiempo t_i que se necesita para ganar cada batalla, junto a su peso b_i que indica cuán importante es la batalla. Además, se define F_i como el momento en que finaliza la batalla i. Si la primera batalla es la j, entonces $F_j = t_j$; en cambio, si la batalla j es la segunda, siendo primera la batalla i, entonces $F_j = F_i + t_j$.

Nos piden minimizar la siguiente suma ponderada de los tiempos de finalización:

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

2. Desarrollo

2.1. Solucion no optima

Para minimizar dicha suma, los valores b_i y F_i deberían ser lo más chicos posible. Sabemos que los valores b_i no varían (son fijos) y que el valor F_i depende del orden de las batallas. Por lo tanto, se podría pensar que, para minimizar la suma, el orden de las batallas debería definirse por la duración de estas, eligiendo en cada oportunidad la de menor duración entre las que quedan por elegir. Esta idea, si bien es una solución *greedy*, no es óptima.

Contraejemplo. Sea (b_i, t_i) una batalla, donde b_i es el indicador de importancia de la batalla i y t_i el tiempo necesario para ganarla. Dadas las siguientes batallas $(1, 10)$ y $(15, 12)$, utilizando el algoritmo greedy recién mencionado junto a la fórmula inicial, el resultado sería:

$$1 \cdot 10 \ + \ 15 \cdot (10 + 12) \ = \ 1 \cdot 10 \ + \ 15 \cdot 22 \ = \ 340.$$

En este caso se elige $(1, 10)$ como primera batalla y $(15, 12)$ como segunda. Sin embargo, si se elige como primera a $(15, 12)$ y como segunda a $(1, 10)$, se obtiene:

$$15 \cdot 12 \ + \ 1 \cdot (12 + 10) \ = \ 180 \ + \ 22 \ = \ 202,$$

lo cual logra el objetivo de minimizar la suma ponderada de los tiempos de finalización, mostrando que el enfoque greedy por duración no es óptimo.

2.2. Presentacion de algoritmo optimo

La estrategia propuesta consiste en que, cada vez que debemos decidir qué batalla realizar antes que las demás, elegimos aquella con menor cociente $\frac{t_i}{b_i}$, es decir ordenar los trabajos en orden creciente del cociente $\frac{t_i}{b_i}$ (equivalente a

$$t_i w_i^{-1} \quad \text{si } w_i = b_i,$$

) minimiza Φ. Así, el algoritmo es *greedy*, produce una solución óptima y tiene complejidad $O(n \log n)$. Continuamos con la demostración de optimalidad (intercambio por pares)

Sea una permutación (secuencia) arbitraria π de las n batallas. Consideremos dos trabajos (batallas) consecutivos en la secuencia, ubicados en posiciones k y $k + 1$. Denotemos:

$$i = \pi_k, \qquad j = \pi_{k+1}.$$

Sea

$$T \ = \ \sum_{m=1}^{k-1} t_{\pi_m},$$

el tiempo acumulado antes de ejecutar i y j (es decir, T es el instante en que comienza la batalla i en la secuencia actual).

Supondremos en todo momento que

$$t_i > 0, \quad t_j > 0, \quad b_i > 0, \quad b_j > 0.$$

2.3. Casos a comparar

Vamos a comparar dos posibles órdenes locales de estos trabajos, manteniendo fija la parte izquierda (los tiempos hasta T) y la parte derecha (trabajos posteriores):

1. **Orden i luego j (configuración actual):**

$$F_i = T + t_i,$$
$$F_j = T + t_i + t_j,$$

con contribución local al coste total:

$$C_{ij} \;=\; b_i F_i + b_j F_j \;=\; b_i(T + t_i) + b_j(T + t_i + t_j).$$

2. **Orden j luego i (configuración intercambiada):**

$$F'_j = T + t_j,$$
$$F'_i = T + t_j + t_i,$$

con contribución local al coste total:

$$C_{ji} \;=\; b_j(T + t_j) + b_i(T + t_j + t_i).$$

Obsérvese que el resto del horario (trabajos antes de T y después de estos dos) queda idéntico en ambas configuraciones; por eso basta comparar C_{ij} y C_{ji}.

2.4. Cálculo de la diferencia de costes

Calculamos entonces la diferencia:

$$\begin{aligned} C_{ij} - C_{ji} &= \big[b_i(T + t_i) + b_j(T + t_i + t_j)\big] - \big[b_j(T + t_j) + b_i(T + t_j + t_i)\big] \\ &= b_iT + b_it_i + b_jT + b_jt_i + b_jt_j - b_jT - b_jt_j - b_iT - b_it_j - b_it_i \\ &= (b_jt_i - b_it_j). \end{aligned}$$

Por tanto:

$$C_{ij} - C_{ji} \;=\; b_jt_i - b_it_j.$$

2.5. Condición de intercambio favorable

Queremos saber cuándo poner i antes que j (es decir, mantener el orden actual) no empeora el coste comparado con el orden alternativo. Buscamos entonces la condición:

$$C_{ij} \leq C_{ji}.$$

Usando el resultado anterior, esto es equivalente a:

$$b_jt_i - b_it_j \;\leq\; 0 \quad\Longleftrightarrow\quad b_jt_i \;\leq\; b_it_j.$$

Dado que $b_i, b_j > 0$, podemos dividir y obtener la forma más intuitiva:

$$\frac{t_i}{b_i} \;\leq\; \frac{t_j}{b_j}.$$

Interpretación. Colocar i antes de j no aumenta el coste (y si la desigualdad es estricta lo reduce) si y sólo si el ratio $r_i = t_i/b_i$ es menor o igual que $r_j = t_j/b_j$. Esto es exactamente la *regla local de Smith*:

- Si $r_i < r_j$, entonces $C_{ij} < C_{ji}$: el intercambio mejora estrictamente el coste, por lo que la posición correcta es i antes que j.
- Si $r_i = r_j$, entonces $C_{ij} = C_{ji}$: ambos órdenes son equivalentes en cuanto al coste Φ.
- Si $r_i > r_j$, entonces $C_{ij} > C_{ji}$: conviene intercambiar i y j.

2.6. Lema (formal)

[Intercambio local] Sea una secuencia con dos trabajos consecutivos i y j, y tiempo acumulado T antes de ellos. Entonces, intercambiar i y j no aumenta el coste total Φ si y sólo si

$$\frac{t_i}{b_i} \leq \frac{t_j}{b_j}.$$

Si la desigualdad es estricta, el intercambio disminuye estrictamente Φ.

La prueba ya ha quedado demostrada en el cálculo anterior de la diferencia $C_{ij} - C_{ji}$.

2.7. Teorema (optimalidad global por eliminación de inversiones)

[Optimalidad de la regla de Smith] Cualquier permutación π que ordene los trabajos por ratios crecientes

$$r_1 \leq r_2 \leq \cdots \leq r_n, \qquad r_k = \frac{t_k}{b_k},$$

minimiza

$$\Phi(\pi) = \sum_{k=1}^{n} b_{\pi_k} F_{\pi_k}.$$

Es decir, el algoritmo que ordena por t_i/b_i creciente produce una solución óptima.

[Prueba (por intercambio y eliminación de inversiones)]

1. Tomemos una permutación arbitraria π. Si π ya satisface $r_{\pi_1} \leq r_{\pi_2} \leq \cdots \leq r_{\pi_n}$, entonces π es de la forma que propone Smith y ya es óptima.

2. Si π no está ordenada por r, existe al menos una inversión: un par $p < q$ con $r_{\pi_p} > r_{\pi_q}$. Consideremos el primer (más a la izquierda) par invertido. Si la inversión no es con elementos consecutivos, podemos mover el trabajo π_q hacia la izquierda intercambiándolo con su antecesor repetidamente hasta colocarlo justo después de la posición $p-1$. Cada intercambio se realiza entre un trabajo con ratio mayor y otro con ratio menor, por lo que, según el Lema, cada paso no aumenta Φ (y si la desigualdad es estricta, lo disminuye estrictamente).

3. El número de inversiones de una permutación con n elementos es un entero entre 0 y $\binom{n}{2}$. Como cada intercambio de pares adyacentes reduce este número, tras un número finito de intercambios llegamos a una permutación sin inversiones, es decir, ordenada por r. Cada una de las operaciones no aumenta Φ y, cuando las desigualdades son estrictas, lo disminuyen. Por lo tanto, la permutación final tiene Φ menor o igual que la original.

4. Por contradicción: supongamos que π^* es una permutación óptima (minimiza Φ) pero no está ordenada por r. Entonces existe una inversión adyacente (o se puede generar una por el procedimiento anterior). Intercambiar esa pareja reduciría Φ (por el Lema), contradiciendo la optimalidad de π^*. Por tanto, cualquier permutación óptima no puede contener inversiones y debe estar ordenada por r.

En conclusión, todas las permutaciones ordenadas por los ratios $r_i = t_i/b_i$ alcanzan el valor mínimo de Φ. Es decir, la *regla de Smith* es óptima.

2.8. Explicacion de porque es Greedy

Este algoritmo es de tipo *Greedy* porque, en cada paso, selecciona la opción localmente óptima (la batalla con menor relación entre su duración y su importancia), con el objetivo de llegar a una solución globalmente óptima.

Más adelante se demostrará formalmente que este criterio garantiza siempre obtener el orden óptimo de batallas.

El algoritmo implementa la regla de Smith de forma directa. El algoritmo es *greedy* porque en cada paso toma decisiones locales independientes: al ordenar las tareas únicamente se considera el cociente t_i/b_i, colocándose antes el trabajo con ratio menor.

Esta decisión local no depende de estados futuros ni obliga a revisar decisiones anteriores. Es la aplicación directa de la *propiedad de elección codiciosa*.

[Optimalidad del criterio greedy] El Lema de intercambio local establece que si dos trabajos i y j están en orden según

$$\frac{t_i}{b_i} \leq \frac{t_j}{b_j},$$

entonces intercambiarlos no disminuye el coste total Φ. En cambio, si la desigualdad se incumple, entonces sí conviene intercambiarlos, reduciendo estrictamente Φ.

Por tanto, ordenar todas las tareas de acuerdo con los cocientes t_i/b_i genera una permutación sin inversiones. El Teorema de optimalidad global muestra que toda permutación óptima debe respetar este orden. De aquí se concluye que la estrategia greedy de Smith conduce siempre a una solución óptima.

Complejidad. El tiempo de ejecución está dominado por el paso de ordenamiento, que requiere $O(n \log n)$. El recorrido posterior para calcular los tiempos acumulados y la suma ponderada tiene coste lineal $O(n)$.

En consecuencia, la complejidad total es:

$$O(n \log n).$$

2.9. Implementación

El siguiente algoritmo implementa la estrategia propuesta para minimizar $\sum b_i \cdot F_i$ (suma ponderada de los tiempos de finalización):

```
def algoritmo(guerras):
    guerras = sorted(guerras, key=lambda x: x[0] / x[1])
    F = 0
    orden = []
    suma = 0

    for tiempo, peso in guerras:
        orden.append((tiempo, peso))
        F += tiempo
        suma += F * peso

    return orden, suma
```

2.9.1. Explicación paso a paso

```
guerras = sorted(guerras, key=lambda x: x[0] / x[1])
```

Cada batalla viene dada como un par (t_i, b_i).

Se ordenan de menor a mayor $\frac{t_i}{b_i}$, que es el criterio greedy óptimo. Esto asegura que las batallas con mayor importancia relativa por unidad de tiempo se resuelvan antes.

```
F = 0
suma = 0
for tiempo, peso in guerras:
    F += tiempo
    suma += F * peso
```

Se va acumulando el tiempo total transcurrido F. Cada vez que termina una batalla, se multiplica el tiempo actual F por su peso b_i y se suma al total.

Esto computa exactamente $\sum b_i \cdot F_i$.

```
return orden, suma
```

Devuelve el orden óptimo de batallas (ya ordenado) y el valor mínimo alcanzado de la suma ponderada.

3. Mediciones

3.1. Complejidad

Para poder analizar la complejidad del algoritmo, desglosaremos el codigo en distintas secciones.

La funcion principal empieza de la siguiente forma:

```
guerras = sorted(guerras, key=lambda x: x[0] / x[1])
```

En esta seccion del codigo se calcula para cada tupla de *(tiempo, peso)* el valor *tiempo/peso*. Luego se **ordena** la lista en orden **ascendente** de esa razon.

Calcular la diferencia de tiempo sobre peso de cada elemento es **lineal**, pero ordenar los n elementos utilizando la funcion *sorted* en Python tiene una complejidad de $O(n \log n)$.

Luego tenemos la inicializacion de variables, lo cual es *constante*:

```
F = 0
orden = []
suma = 0
```

Y por ultimo, el bucle principal del codigo:

```
for tiempo, peso in guerras:
    orden.append((tiempo, peso))
    F += tiempo
    suma += F * peso
```

En esta seccion se recorre cada elemento de *guerras*, por lo que se realizan n iteraciones. En cada una de estas se realizan operaciones *constantes* por lo que la complejidad del bucle sera $O(n)$.

Para poder analizar la complejidad del algoritmo, desglosaremos el código en distintas secciones, considerando el costo de cada parte de manera individual.

1. Ordenamiento de la lista `guerras`

```
guerras = sorted(guerras, key=lambda x: x[0] / x[1])
```

En esta línea se ordenan las tuplas de la lista `guerras` en función de la razón $\frac{tiempo}{peso}$. Dicha operación es constante ($O(1)$), pero se ejecuta como parte del proceso de comparación utilizado por la función `sorted`.

Python implementa el algoritmo *Timsort*, cuyas propiedades son:

$$\text{Peor caso: } O(n \log n), \quad \text{Mejor caso: } O(n), \quad \text{Caso promedio: } O(n \log n)$$

Dado que en análisis de algoritmos se considera habitualmente el peor caso, el costo de esta sección se establece como:

$$O(n \log n)$$

2. Inicialización de variables

```
F = 0
orden = []
suma = 0
```

Se inicializan tres variables: un acumulador de tiempo, una lista vacía y una suma parcial. Todas estas operaciones son de tiempo constante, por lo que su complejidad es:

$$O(1)$$

3. Bucle principal

```
for tiempo, peso in guerras:
    orden.append((tiempo, peso))
    F += tiempo
    suma += F * peso
```

El bucle recorre n elementos, realizando en cada iteración:

- Una inserción en la lista (`append`): $O(1)$ amortizado.
- Una suma aritmética: $O(1)$.
- Una multiplicación y suma acumulativa: $O(1)$.

Dado que todas las operaciones son constantes, y el bucle se repite n veces, su complejidad es:

$$O(n)$$

3.2. Complejidad Final del Algoritmo

- Ordenamiento de guerras: $O(n \log n)$.
- Inicializacion de variables: $O(1)$.
- Recorrido de las guerras: $O(n)$.

El algoritmo tiene complejidad final $O(n \log n)$ en el peor caso, debido a lo explicado previamente.

3.3. Demostración empírica

Para demostrar, empíricamente, la complejidad planteada teóricamente tomamos mediciones de tiempo de ejecución del algoritmo con distintos tamaños de muestra.

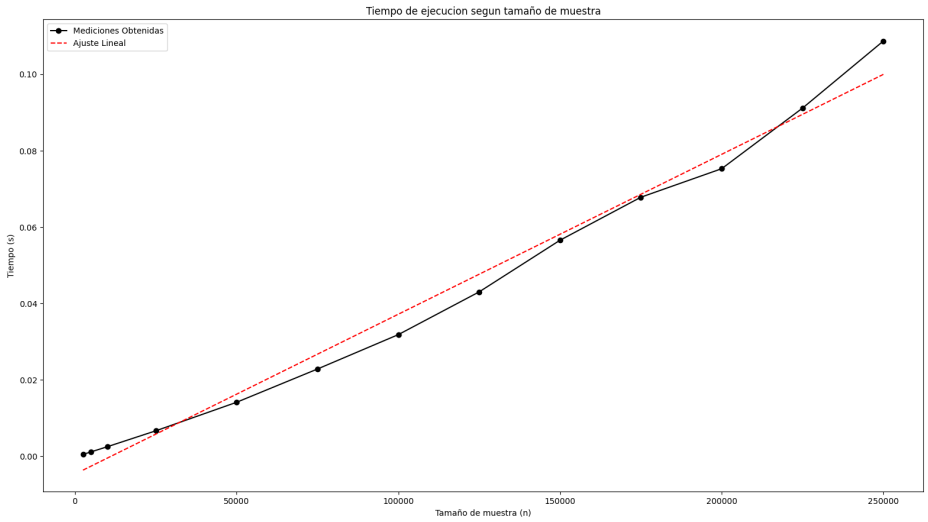

Figura 1: Gráfico del tiempo de ejecución según tamaño de muestra.

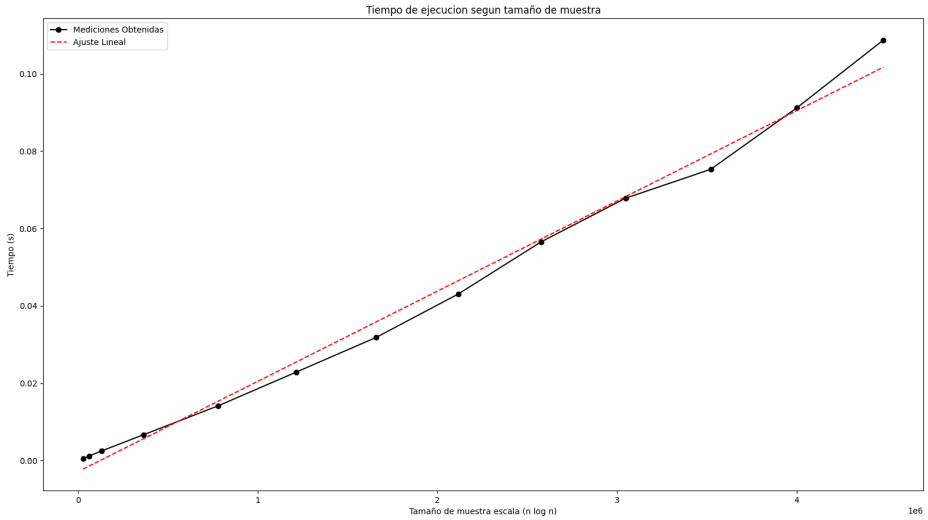

Figura 2: Gráfico del tiempo de ejecución según tamaño de muestra modificando la escala del eje X a escala logarítmica.

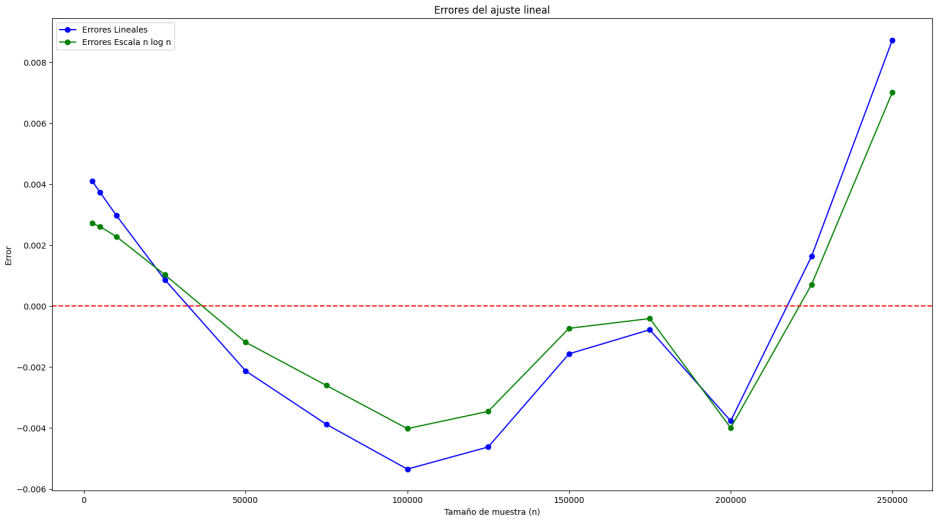

Figura 3: Comparativa de errores del ajuste lineal sobre las mediciones con escala logarítmica y escala lineal.

Para demostrar que el algoritmo planteado es efectivamente $O(n \log n)$ llamamos $T(n)$ al tiempo que demora el algoritmo para una cantidad n de elementos, como podemos ver en la figura 1 planteamos este $T(n)$ directamente contra la cantidad n de elementos, en cambio en la figura 2 cambiamos la escala del eje x y la transformamos en $n \log n$, si realmente nuestro algoritmo fuese $O(n \log n)$ los resultados obtenidos comparados con la escala transformada debería ajustar mejor ante un ajuste lineal, podemos ver que esto realmente sucede en la figura 3 donde vemos

la comparativa de errores de los ajustes lineales de las escalas previamente mencionadas, notamos que los errores obtenidos con la escala transformada son menores, justamente lo esperado.

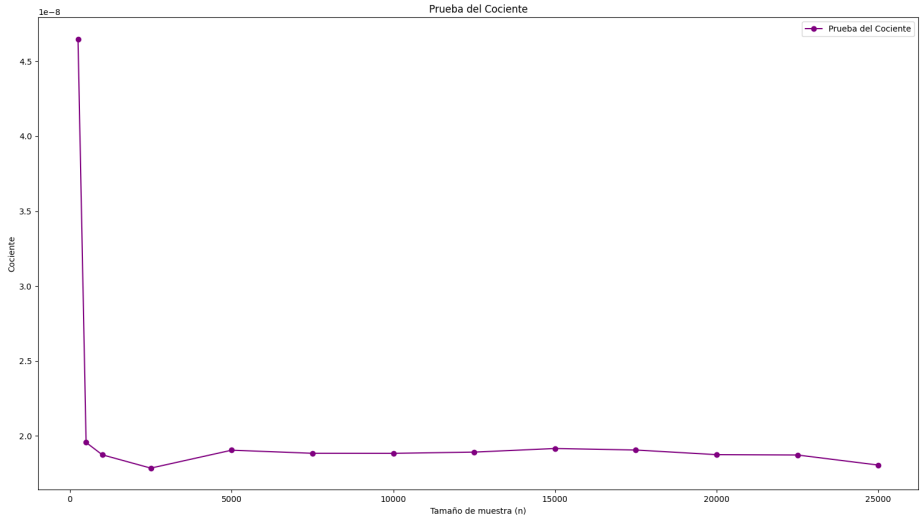

Figura 4: Gráfico de prueba del cociente.

Para reafirmar lo anterior, introducimos la prueba del cociente, sabemos que si el algoritmo es efectivamente $O(n \log n)$ podemos plantear al tiempo que demora el algoritmo como $T(n) \approx c \cdot n \log n$ de esto podemos deducir que $\frac{T(n)}{n \log n} \approx c$. Podemos esperar que para valores pequeños de n tenga mas peso el resto de operatorias del algoritmo por lo que el gráfico no quede en la constante c, debido a esto el tamaño de muestra de este gráfico es menor a los anteriores, pero a medida que la cantidad de elementos crecen el gráfico debería estabilizarse en la constante. Podemos ver esto en la figura 4 quedando reafirmada la demostración empírica de que el algoritmo es $O(n \log n)$.

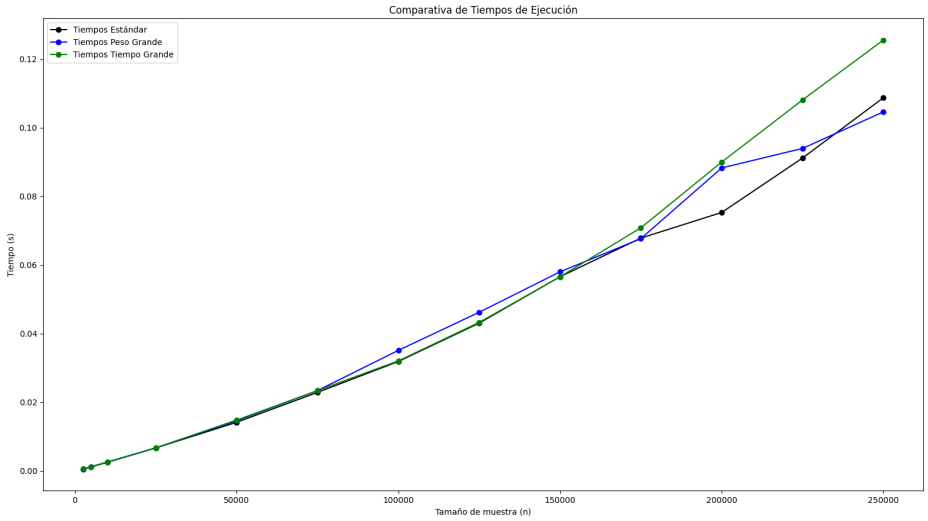

Figura 5: Gráfico de comparativa de tiempos según variabilidad de los datos.

Finalmente tomamos una variación de los datos, como se puede ver en la figura 5, corriendo el algoritmo con tiempos muy grandes o ponderaciones muy grandes y no se puede ver una variación significativa de los tiempos de ejecución del algoritmo.

4. Conclusiones

A lo largo de este trabajo se pudo abordar distintos aspectos sobre la elaboración y optimización de algoritmos Greedy.

Buscamos una estrategia Greedy la cual soluciones el problema propuesto y, en base a eso, plantear un algoritmo inicial. PSe pudieron dar optimizaciones sobre el mismo, hasta llegar a una versión final.

Teniendo una versión final del algoritmo se pudo analizar su comportamiento y su rendimientos en diferentes escenarios para realizar análisis, concluyendo en que el algoritmo final tenia un desempeño estable en la mayoría de los casos.