.UBAfiuba
FACULTAD DE INGENIERÍA

Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

Trabajo Práctico 1
Algoritmos Greedy en la Nación del Fuego

18 de septiembre de 2025

Agustin Bermudez	Mateo Gonzalez Pautaso	Tiago Calderón
111863	111699	111894

1. Análisis del problema

Para comenzar a plantear la solución al problema, tomamos ideas de los ejercicios vistos en clase relacionados con Algoritmos Greedy, particularmente el ejercicio de la mochila y el de Scheduling II.

1.1. Primer enfoque

La primera idea consistió en considerar como regla Greedy el elegir en cada estado la batalla con mayor peso que aún no se había luchado. Esta propuesta surge de un análisis de la fórmula que calcula la suma ponderada a minimizar:

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

Dado que el valor de F_i (el acumulado de felicidad) aumenta a medida que se acumulan batallas, parecía razonable que las batallas con mayor peso b_i se multipliquen con los valores de felicidad más chicos. De esta forma, se interpretó que convenía realizar primero aquellas batallas más importantes. Sin embargo, luego de un análisis, llegamos a la conclusión de que esta solución propuesta no es óptima, demostrado por este contraejemplo:

Consideremos dos batallas:

$$A:\ t_A = 3,\ b_A = 3 \qquad B:\ t_B = 1,\ b_B = 2$$

Aplicando esta solución, se prioriza elegir primero la batalla A debido a ser la de mayor peso, lo que nos deja con el siguiente resultado en la suma ponderada:

$$\begin{aligned} Caso_{A \to B} &= b_A \cdot t_A + b_B \cdot (t_A + t_B) \\ &= 3 \cdot 3 + 2 \cdot (3 + 1) \\ &= 9 + 8 = 17 \end{aligned}$$

Siendo que la solución óptima es la siguiente:

$$\begin{aligned} Caso_{B \to A} &= b_B \cdot t_B + b_A \cdot (t_B + t_A) \\ &= 2 \cdot 1 + 3 \cdot (1 + 3) \\ &= 2 + 12 = 14 \end{aligned}$$

Como $14 < 17$, se observa que priorizar sólo por el peso de la batalla no resuelve el problema de manera óptima.

1.2. Segundo enfoque

Luego de comprobar que la primera regla planteada no garantizaba optimalidad, decidimos desarrollar la solución a partir de la propia fórmula que calcula la suma ponderada. Para ello, consideramos el caso más simple con dos batallas:

$$A: t_A, b_A \qquad B: t_B, b_B$$

Suponemos que para minimizar el resultado final, la primera batalla a eligir debe ser A. ¿Qué debe pasar para que este sea el caso?

Desarrollamos los dos casos de la sumatoria, en uno primero se elige A y en el otro primero se elige B. Desde la suposición inicial se puede formular la próxima inecuación

$$b_A \cdot t_A + b_B \cdot (t_A + t_B) < b_B \cdot t_B + b_A \cdot (t_A + t_B)$$

Se aplica la distributiva en ambos lados:

$$b_A \cdot t_A + b_B \cdot t_A + b_B \cdot t_B < b_B \cdot t_B + b_A \cdot t_A + b_A \cdot t_B$$

Cancelamos los términos redundantes:

$$b_B \cdot t_A < b_A \cdot t_B$$

Lo que resulta en que:

$$\frac{t_A}{b_A} < \frac{t_B}{b_B}$$

En conclusión, para que haber elegido la batalla A como la primera resulte en una sumatoria menor que el caso donde se elige primero la batalla B, el cociente $\frac{t_A}{b_A}$ debe haber sido menor que el cociente $\frac{t_B}{b_B}$

2. Demostración de optimalidad del algoritmo

A continuación, se demostrará que seleccionar el orden de las batallas por el cociente t_i/b_i de forma creciente produce una solución óptima para minimizar la sumatoria ponderada:

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

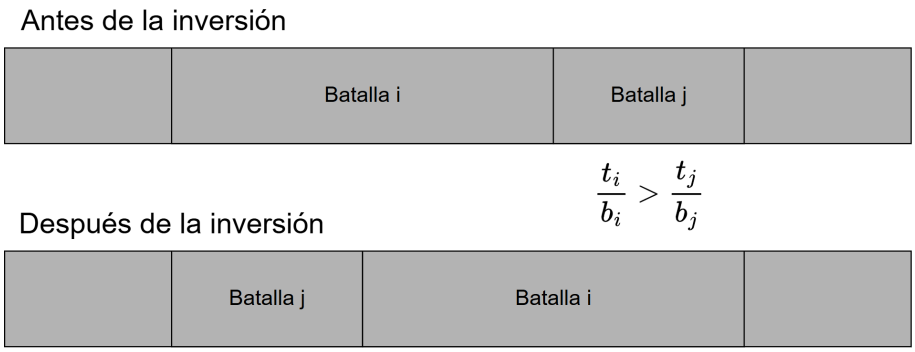
2.1. Definición de inversión

Definición. Decimos que una solución tiene una *inversión* si existen dos posiciones contiguas i y j, donde $i < j$ y, a su vez,

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

2.2. Planteo: existe un orden de batallas sin inversiones que es óptimo

Demostración. Supongamos que hay un orden de batallas óptimo que sí tiene inversiones. Como las tiene, debería poder intercambiar siempre de a pares las batallas inversibles y contiguas. Luego de una cantidad de inversiones, se debería llegar a un orden sin inversiones que acumule el mismo coeficiente de impacto.
A continuación se demuestra matemáticamente que hacer estas inversiones no puede empeorar la solución óptima. Es decir, realizar una inversión entre dos batallas no aumenta la sumatoria ponderada.



Analizando cómo afecta la inversión en el coeficiente de impacto C_i:

Sumatoria antes de la inversión (K representa la sumatoria de las batallas luchadas hasta el momento y F la felicidad acumulada hasta la primera batalla presente en la inversión):

$$C_i = K + b_i \cdot (F + t_i) + b_j \cdot (F + t_i + t_j)$$

Luego de hacer la inversión la sumatoria se representa con:

$$C_i' = K + b_j \cdot (F + t_j) + b_i \cdot (F + t_j + t_i)$$

Se desarrollan ambas ecuaciones:

$$C_i = K + b_i \cdot F + b_i \cdot t_i + b_j \cdot F + b_j \cdot t_i + b_j \cdot t_j$$

$$C_i' = K + b_j \cdot F + b_j \cdot t_j + b_i \cdot F + b_i \cdot t_j + b_i \cdot t_i$$

Observando los términos de cada una, se visualiza que hay un único termino diferente entre ellas:

$$Término\ C_i \to b_j \cdot t_i$$
$$Término\ C'_i \to b_i \cdot t_j$$

Recordando que $\frac{t_i}{b_i} > \frac{t_j}{b_j}$ y que por ende $t_i \cdot b_j > t_j \cdot b_i$, se concluye que haber hecho una inversión no pudo haber empeorado la solución, es decir, provocar un aumento en el resultado final de la sumatoria. Esto demuestra que existe un orden de batallas óptimo sin inversiones.

2.3. Planteo: todos los órdenes de batallas sin inversiones son igual de óptimos

Demostración. Las diferencias entre todos los órdenes de batallas que son óptimos solo pueden darse entre batallas contiguas que satisfacen $\frac{t_i}{b_i} = \frac{t_j}{b_j}$

Por este motivo, basta con probar que intercambiar el orden de esas batallas no altera el coeficiente de impacto final.

En un ejemplo con un intercambio entre dos batallas i, j, donde $\frac{t_i}{b_i} = \frac{t_j}{b_j}$:

Sea T el tiempo acumulado antes de estas dos tareas; las contribuciones a la sumatoria en cada orden toma la forma:

$$C_{i\to j} = b_i(T + t_i) + b_j(T + t_i + t_j),$$
$$C_{j\to i} = b_j(T + t_j) + b_p(T + t_j + t_i).$$

Se aplica la distributiva para analizar cada término en cada una de las soluciones:

$$C_{i\to j} = b_iT + b_it_i + b_jT + b_jt_i + b_jt_j,$$
$$C_{j\to i} = b_jT + b_jt_j + b_iT + b_it_j + b_it_i.$$

Si $\frac{t_i}{b_i} = \frac{t_j}{b_j}$ entonces $b_jt_i = b_it_j$, por lo que ambas soluciones son igual de óptimas.

Finalmente, cualquier permutación entre batallas contiguas y con el mismo cociente $\frac{t}{b}$ no altera el resultado final. Entonces, todas las soluciones sin inversiones son igual de óptimas.

2.4. Conclusión de la demostración

Como hay un orden de batallas sin inversiones que es óptimo y, además, todos los órdenes de batallas sin inversiones generan el mismo coeficiente de impacto (demostrado en **2.1** y **2.2**). Concluimos que este algoritmo es óptimo debido a que logra encontrar un orden de batallas sin inversiones.

3. Detalles del algoritmo

3.1. Implementación

A continuación se encuentra la implementación, en Python, del algoritmo desarrollado en los apartados anteriores. Esta implementación consiste en un algoritmo Greedy que busca minimizar la sumatoria ponderada calculada a partir del orden de las batallas.

Nuestro código cuenta con las características de todo algoritmo Greedy, aplica iterativamente una regla sencilla para buscar el óptimo local en cada estado del algoritmo. La regla sencilla consiste en seleccionar, en cada iteración, como próxima batalla a la que tenga menor cociente $\frac{t_i}{b_i}$ dentro de las que aún no se lucharon, esto con el fin de elegir la batalla que adicione el menor término ponderado posible en cada estado, es decir, un óptimo local.

El siguiente fragmento de código en Python muestra la función desarrollada, que realiza estos pasos y devuelve el valor final de la sumatoria ponderada.

```python
def minimizar_sumatoria(batallas):
    '''
    Recibe una lista de tuplas con la informacion de batallas.

    Las batallas son ordenadas segun el cociente t_i / b_i y en ese orden se
    acumulan los tiempos. Ademas, calcula la sumatoria b_i * F_i

    Cada batalla esta representada como (T_i,B_i) donde:
        - T_i: tiempo de la batalla i
        - B_i: peso de la batalla i

    Devuelve el valor del coeficiente de impacto.
    '''
    batallas.sort(key = lambda x: x[0] / x[1])

    felicidad_actual = 0
    res = 0

    n = len(batallas)
    for i in range(n):
        batalla = batallas[i]
        t_i, b_i = batalla

        felicidad_actual += t_i
        res += (b_i * felicidad_actual)

    return res
```

Como en cada paso iterativo donde se aplica la regla necesitamos buscar la batalla con el coeficiente $\frac{t_i}{b_i}$ más pequeño aún no seleccionada, a modo de optimización, optamos por ordenar las batallas por su cociente de manera ascendente.

De este modo, en lugar de ir buscando el mínimo del estado actual en cada iteración, podemos recorrer el arreglo de manera lineal sabiendo que la batalla seleccionada en la iteración i va a ser la de menor cociente posible aún no elegida.

Además, al necesitar el orden en el que se eligieron las batallas, se optó por un ordenamiento *in-place*. De tal forma, la disposición de las batallas queda comprendida en el mismo arreglo.

3.2. Complejidad

1. La complejidad de la función **sort()** en Python es $O(n \cdot log(n))$ en el peor caso. Información extraída de la documentación oficial de Python.

2. La complejidad de la función **len()** para listas en Python es **O(1)**

3. Se Itera una vez la lista de tuplas ***batallas*** y se hacen operaciones **O(1)** en cada iteración, lo cual tienen una complejidad de **O(n)**

En conclusión, el costo temporal del algoritmo está dominado por la operación de ordenamiento inicial, cuya complejidad es $O(n \log n)$. El resto de las operaciones (cálculo de la longitud de la lista, recorrido de las batallas y sumas parciales) son lineales o constantes, y por lo tanto quedan por debajo de la cota de complejidad que aporta el orden.

Así, la complejidad temporal del algoritmo es $O(n \log n)$, mientras que la complejidad espacial es $O(1)$ porque sort() realiza un ordenamiento in-place y únicamente se utilizan unas pocas variables auxiliares.

4. Pruebas de ejecución

Esta sección se divide en dos partes generales. La primera consiste en la comparación de los resultados del coeficiente de impacto final conocidos, dados por la cátedra, contra los resultados que se obtienen de ejecutar nuestro algoritmo. La segunda parte consta de mediciones de tiempo sobre sets de prueba generados por nosotros y la corroboración de la complejidad teórica indicada para el algoritmo.

4.1. Comparación de resultados con sets de la cátedra

Para testear nuestro código, utilizamos los set de datos y las salidas esperadas proporcionadas:

10.txt → Coeficiente de impacto = 309600 ✓

50.txt → Coeficiente de impacto = 5218700 ✓

100.txt → Coeficiente de impacto = 780025365 ✓

1000.txt → Coeficiente de impacto = 74329021942 ✓

5000.txt → Coeficiente de impacto = 1830026958236 ✓

10000.txt → Coeficiente de impacto = 7245315862869 ✓

100000.txt → Coeficiente de impacto = 728684685661017 ✓

Para esto implementamos ciertas pruebas que comparaban los resultados. Las mismas están implementadas en funciones de Python del estilo:

```
1 def test_10():
2     path_set = "./datasets/10.txt"
3     batallas = parser(path_set)
4     resultado = minimizar_sumatoria(batallas)
5     assert resultado == 309600, "El resultado del dataset de 10 batallas no
      coincide con la salida de catedra"
6     print(f"Coeficiente de impacto 10.txt = {resultado}")
```

que se pueden encontrar dentro del repositorio. La salida corrobora todos los valores y no falla en ningún *assert*

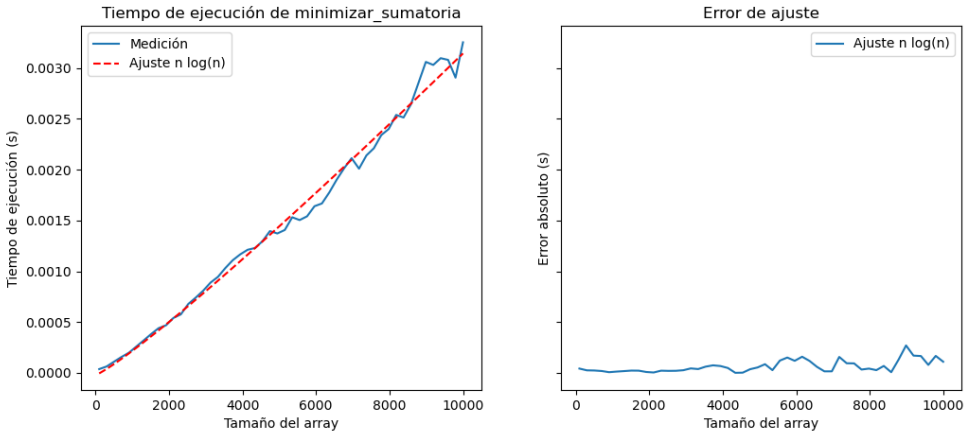
4.2. Mediciones de tiempo en sets generados

Se generaron sets de prueba para testear la complejidad real del algoritmo implementado para minimizar el coeficiente de impacto.

Todo el código relacionado a la generación y medición de tiempo se encuentra dentro del archivo **'test.py'**. Los sets son generados aleatoriamente, pudiendo elegir parámetros para limitar la variabilidad de los valores t_i y b_i, además de la cantidad de batallas a seleccionar.

Al hacer los cálculos temporales, el algoritmo se ejecuta 10 veces sobre los mismos datos y se realiza un promedio sobre los tiempos de ejecución con el objetivo de reducir el ruido y lograr una representación más fiel a la realidad.

El primer gráfico muestra las mediciones de tiempo de las ejecuciones del algoritmo ajustadas a la función de complejidad esperada ($O(n \log n)$). Por otro lado, el segundo muestra el error de ajuste de los datos empíricos obtenidos contra la función de complejidad teórica.



Como se puede observar en la imagen, el error de ajuste es muy pequeño, lo que indica que efectivamente el comportamiento del algoritmo implementado es $O(n \log n)$

4.3. Observaciones sobre la complejidad del algoritmo

Para verificar de manera más certera la complejidad del algoritmo y su ajuste con el comportamiento $O(n \log n)$, se realizaron pruebas de medición con mayor o menor variabilidad de los valores de t_i y b_i. Esto con el objetivo de estudiar al algoritmo bajo diferentes condiciones de ejecución: Las siguientes mediciones limitan los valores t_i y b_i al rango especificado. Mostrando los comportamientos de las imágenes.

4.3.1. t_i, $b_i \in [1, 10]$

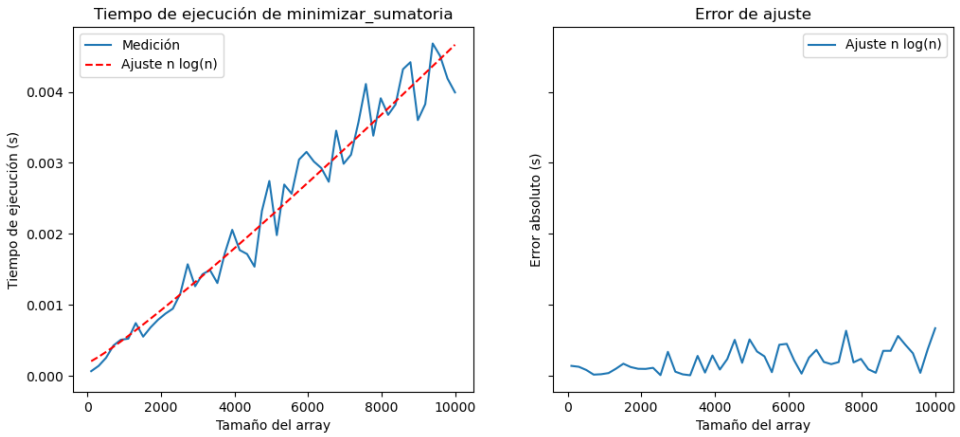


Figura 1: Resultados con t_i, $b_i \in [1, 10]$

4.3.2. t_i, $b_i \in [1, 10000]$

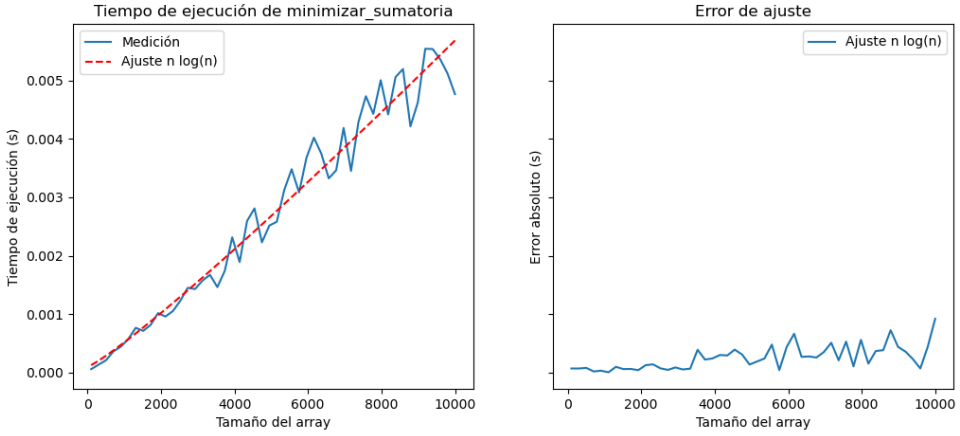


Figura 2: Resultados con t_i, $b_i \in [1, 10000]$

4.3.3. t_i, $b_i \in [1, 100000000]$

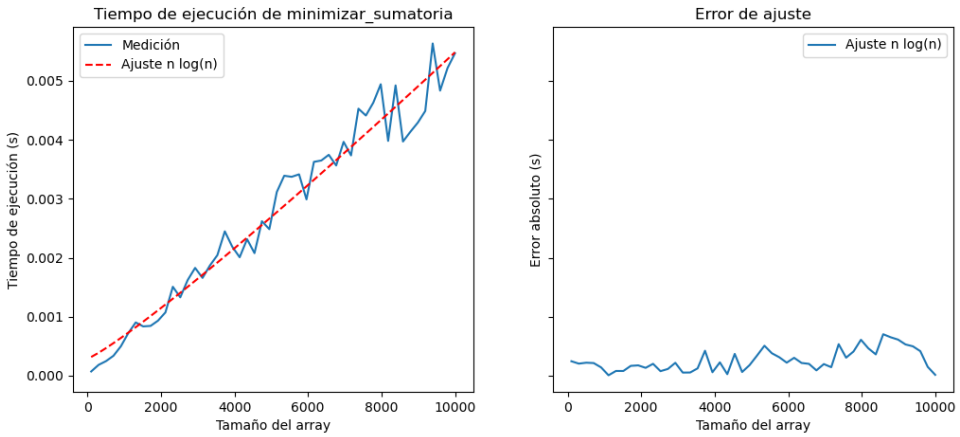


Figura 3: Resultados con t_i, $b_i \in [1, 100000000]$

A partir de estas mediciones, se observa que el tiempo de ejecución sigue el patrón esperado de $O(n \log n)$, sin depender de la variabilidad de los valores. Esto confirma la consistencia del algoritmo bajo distintos rangos de entrada.

Esto se debe a que la complejidad del algoritmo está dada principalmente por el ordenamiento de la lista que contiene las batallas, hecho con la función *sort* de Python, cuyo funcionamiento tampoco depende de la variabilidad de los valores.

5. Anexo de correciones

5.1. ¿Cómo se puede garantizar que mediante alguna cantidad de inversiones se puede llegar a una solución sin inversiones?

Podemos garantizar que esto ocurre debido a que sólo vamos a hacer las inversiones que no cumplan el orden que plantea nuestro algoritmo. Es decir, vamos a invertir dos batallas si y sólo si no se cumple que:

$\frac{t_i}{b_i} > \frac{t_{i+1}}{b_{i+1}}$

Además, cada inversión de un par contiguo reduce en una unidad el número total de inversiones en la secuencia. Definimos inv(S) como la cantidad de pares (i, i + 1) con $\frac{t_i}{bi} > \frac{ti+1}{b_{i+1}}$

Cada vez que se invierte un par contiguo, el número total de inversiones en la secuencia disminuye exactamente en una unidad. En efecto, al intercambiar las posiciones de los elementos i e i + 1, no se generan nuevas inversiones, ya que el orden del resto de los elementos permanece inalterado. Por lo tanto, se cumple que inv(S')(nueva cantidad de inversiones) = inv(S) - 1

5.2. Correcciones de constantes

Analizando cómo afecta la inversión en el coeficiente de impacto C_i:

Sumatoria antes de la inversión ($K1$ representa la sumatoria de las batallas luchadas hasta el momento, F la felicidad acumulada hasta la primera batalla presente en la inversión y $K2$ representa la sumatoria de las batallas luchadas luego de la i y la j):

$$C_i = K1 + b_i \cdot (F + t_i) + b_j \cdot (F + t_i + t_j) + K2$$

Luego de hacer la inversión la sumatoria se representa con:

$$C'_i = K1 + b_j \cdot (F + t_j) + b_i \cdot (F + t_j + t_i) + K2$$

Se desarrollan ambas ecuaciones:

$$C_i = K1 + b_i \cdot F + b_i \cdot t_i + b_j \cdot F + b_j \cdot t_i + b_j \cdot t_j + K2$$
$$C'_i = K1 + b_j \cdot F + b_j \cdot t_j + b_i \cdot F + b_i \cdot t_j + b_i \cdot t_i + K2$$

Observando los términos de cada una, se visualiza que hay un único término diferente entre ellas:

$$Término\ C_i \rightarrow b_j \cdot t_i$$
$$Término\ C'_i \rightarrow b_i \cdot t_j$$

Nota: La demostración sigue siendo la misma, pero en este caso se tiene en cuenta la sumatoria posterior a las dos batallas de la inversión. De todas maneras, aún considerando este término, la conclusión de la demostración se mantiene igual.

5.3. Aclaración sobre la regla Greedy

Como fue descrito en el punto 3 del informe, la regla Greedy del algoritmo consiste en seleccionar la batalla aún no luchada que tenga el menor coeficiente $\frac{t_i}{b_i}$. Sin embargo, aunque se consiga la

solución óptima, el seleccionar esa batalla no necesariamente garantiza que sea la que sume el menor valor al coeficiente de impacto en esa iteración.

El ordenamiento del arreglo en base a los coeficientes $\frac{t_i}{b_i}$ es válido debido a que los valores de los coeficientes no se ven modificados en ninguno de los estados actuales presentes en cada iteración del algoritmo.

5.4. Comentario sobre el apartado de Ejemplos de ejecución

Consideramos que las verificaciones realizadas eran suficientes dado a la demostración de optimalidad del algoritmo Greedy desarrollado. Para cualquier instancia del problema, siempre encuentra una solución óptima.

5.5. Profundización en análisis de variabilidad

Metodología de las pruebas: Para disminuir el ruido en las mediciones, decidimos aumentar la cantidad de ejecuciones por caso, de 10 a 50 veces. Con esta cantidad de ejecuciones se realiza un promedio de los tiempos de ejecución. Los generadores de datasets para las pruebas están hechos usando la librería estándar de Python Random. Para verificar que la complejidad sea efectivamente $O(nlogn)$ decidimos agregar al gráfico un ajuste contra la función lineal $O(n)$, para así poder contrastarlas y estudiar su comportamiento.

Se usó una escala lineal con tamaños para el arreglo de batallas entre [100, 100000]. Con una cantidad de 50 puntos en el gráfico. Se decidieron usar estos parámetros debido a que aumentar las ejecuciones por caso o el intervalo de tamaños de batallas reflejaba un costo muy grande para computar.

5.5.1. Medición general

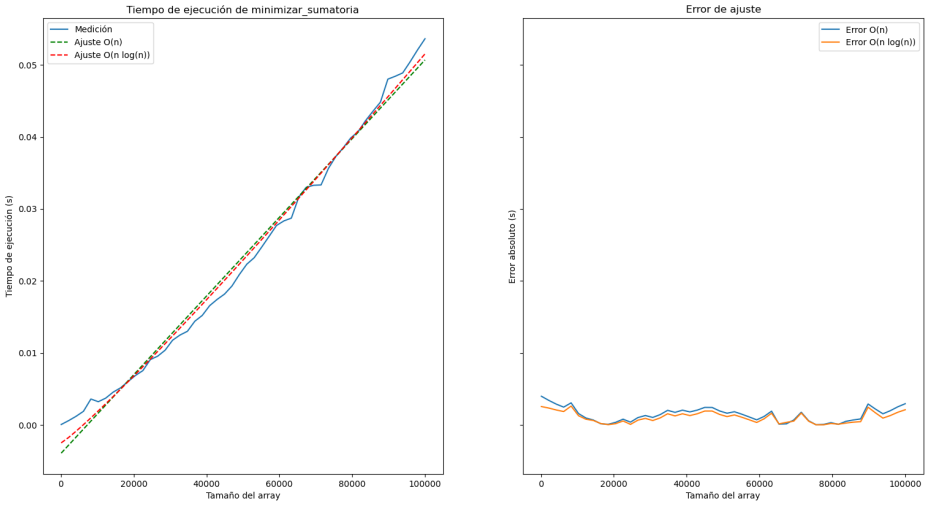


Figura 4: Resultados con t_i, $b_i \in [1, 100000]$

En esta nueva medición se ve una reducción significativa del ruido en cada punto, esto debido al aumento de corridas por caso de ejecución.

Debido a la escala del gráfico no puede apreciarse tanto la diferencia entre ambas complejidades, pero observando el gráfico de error por ajuste, se distingue que el ajuste $O(nlogn)$ muestra un error menor a lo largo de todos los puntos. Para lograr un gráfico donde los ajustes se vean mas separados habría que aumentar el tamaño del arreglo (eje x) a un número mucho mayor.

5.5.2. Medición con un dataset ordenado

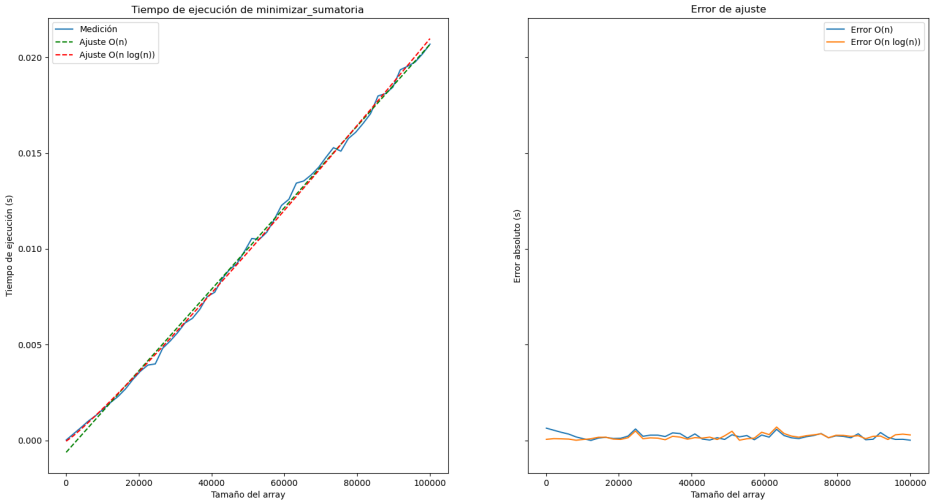


Figura 5: Resultados con t_i, $b_i \in [1, 100000]$

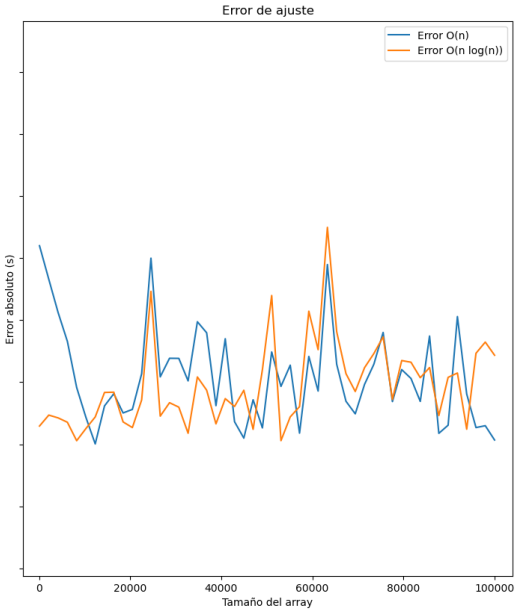


Figura 6: Zoom en el gráfico de error

En estas mediciones, el dataset se encontraba ordenado por T_i/B_i de manera ascendiente, esto se realizo para corroborar que la función sort de Python esta optimizada para arreglos ya ordenados. En el gráfico de ajustes se puede ver que el comportamiento del algoritmo es muy similar a $O(n)$, y se ve de forma mas sencilla en la segunda imagen, donde los errores por ajuste no estan muy distanciados el uno del otro. Esto nos deja con una complejidad muy cercana a $O(n)$, sin embargo, aún se mantiene con $O(nlogn)$.

5.5.3. Medición con un dataset ordenado al revés

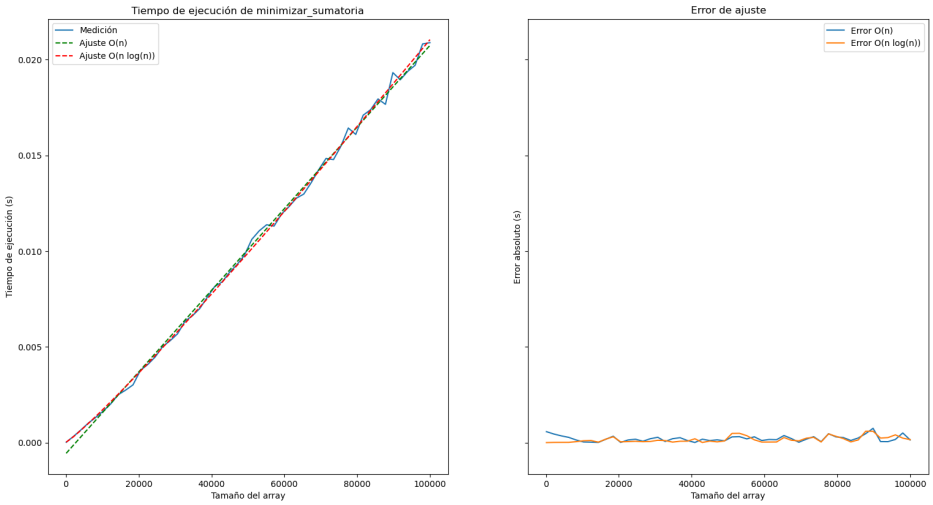


Figura 7: Resultados con t_i, $b_i \in [1, 100000]$

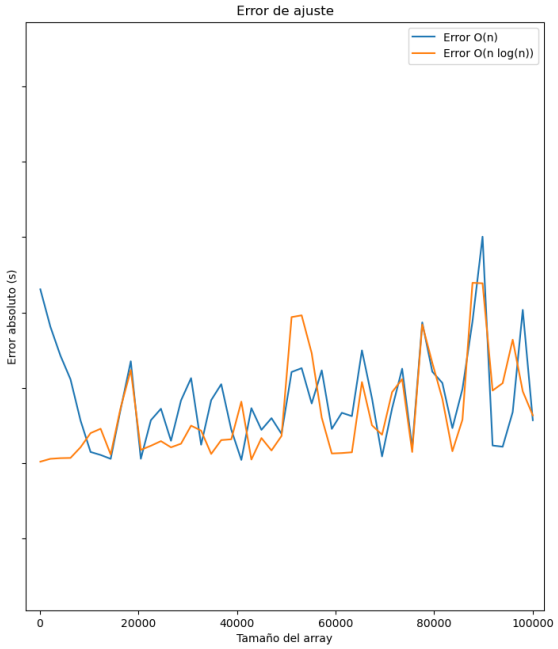


Figura 8: Zoom en el gráfico de error

En este caso, el dataset con los arreglos de batallas se encuentra ordenado de forma inversa, es decir, T_i/B_i descendente. Se observa un caso muy similar al test con el dataset ordenado, esto porque la implementación de *sort* en Python cuenta también con una optimización para los arreglos ordenados de forma inversa. Esto nos deja con un tiempo de ejecución más cercano al lineal y un error por ajuste muy similar al mismo.

5.5.4. Mediciones con valor fijo en una variable

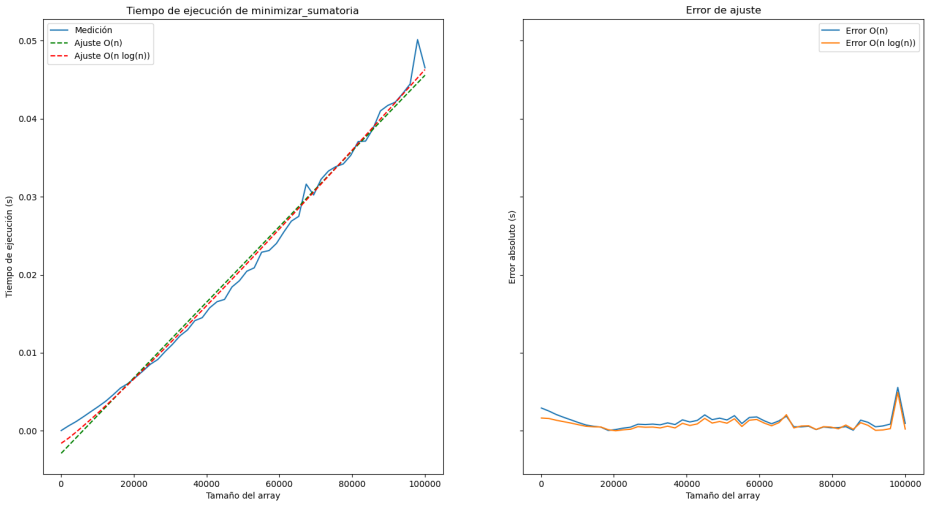


Figura 9: Resultados con $t_i = 100$ fijo y $b_i \in [1, 100000]$

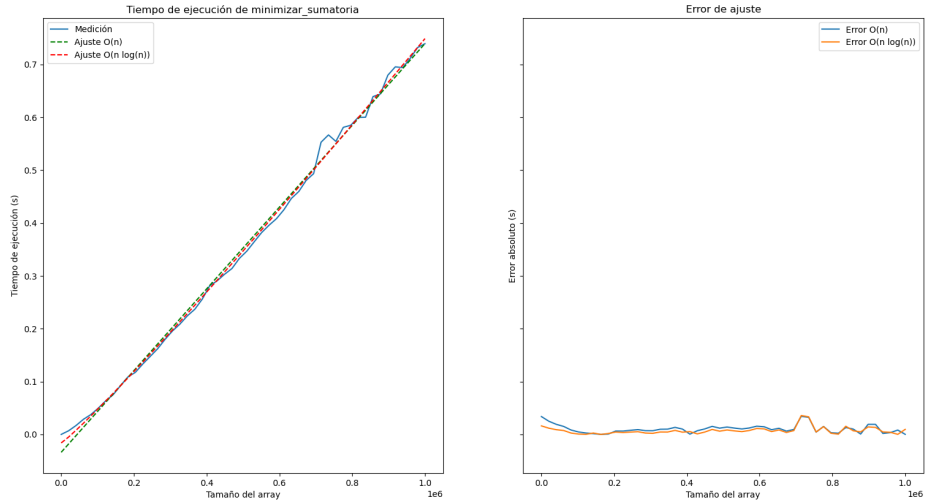


Figura 10: Resultados con $b_i = 100$ fijo y $t_i \in [1, 100000]$

A simple vista pareciera que los comportamientos al dejar una de las variables fijas no cambia el valor de los ajustes. Sin embargo, al hacer Zoom en los gráficos de error, se observa que al dejar el valor T_i fijo el comportamiento del algoritmo es más cercano al lineal que en el caso donde se deja el valor B_i fijo. Esto significa que la variabilidad de los T_i afecta más a los tiempos de ejecución que B_i

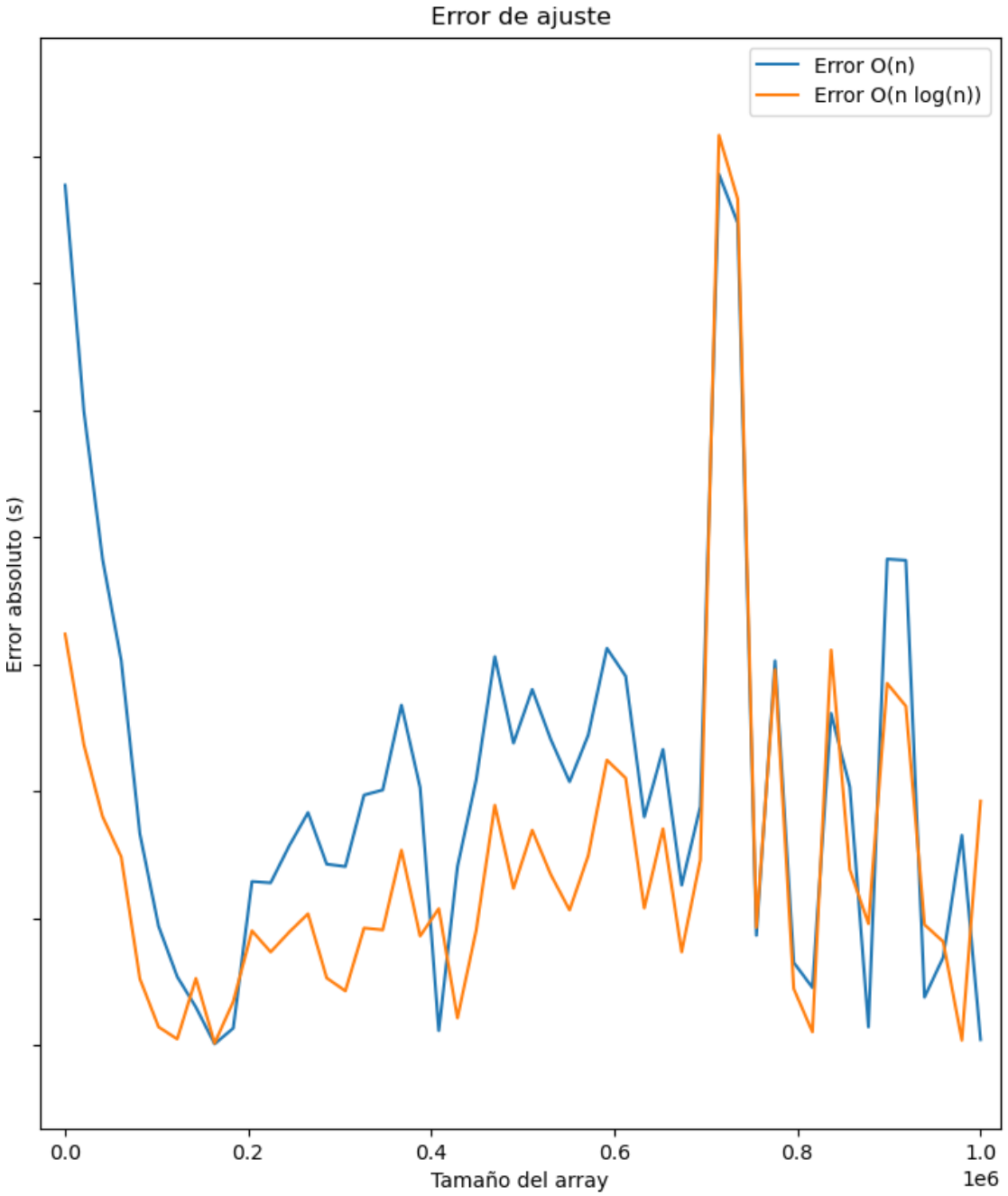


Figura 11: Error de ajuste con $b_i = 100$ fijo y $t_i \in [1, 100000]$

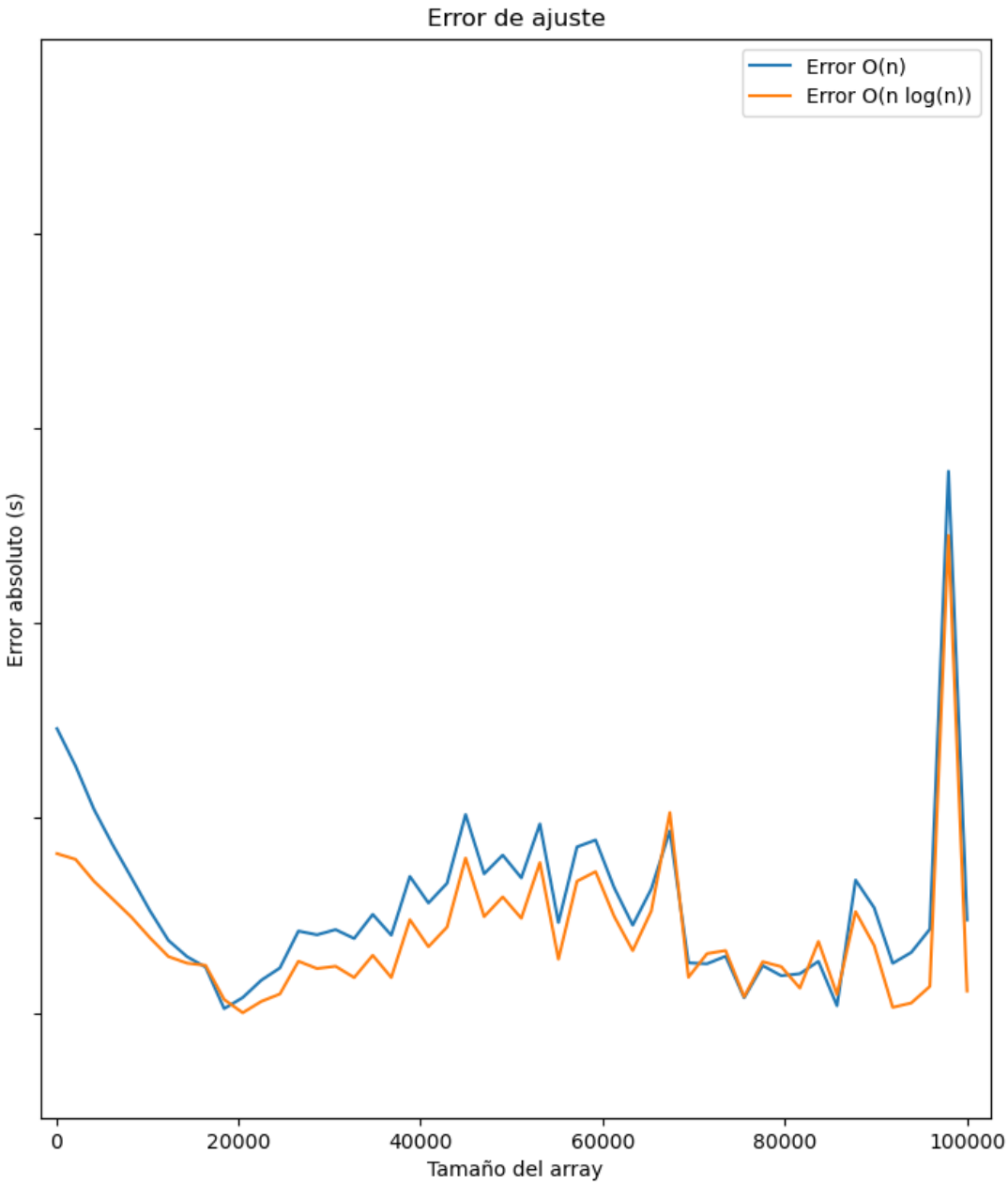


Figura 12: Error de ajuste con $t_i = 100$ fijo y $b_i \in [1, 100000]$