.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

Sabrina García	Agustina Arellano	Anthony Távara	Alexander Cruz
108751	112043	112785	109044

Índice

1. Introducción

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 tales que se logre **minimizar** la suma ponderada de los tiempos de finalización: $\sum_{i=1}^{n} b_i F_i$.

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

1.1. Consigna

- Hacer un análisis del problema, y proponer un algoritmo greedy que obtenga la **solución optima** al problema planteado: Dados los n valores de todos los t_i y b_i, determinar cuál es el orden óptimo para realizar las batallas en el cuál se minimiza $\sum_{i=1}^{n} b_i F_i$.
- Demostrar que el algoritmo planteado obtiene siempre la solución óptima.
- Escribir el algoritmo planteado. Describir y justificar la complejidad de dicho algoritmo. Analizar si (y cómo) afecta la variabilidad de los valores de t_i y b_i a los tiempos del algoritmo planteado.
- Realizar ejemplos de ejecución para encontrar soluciones y corroborar lo encontrado.
- Hacer mediciones de tiempos para corroborar la complejidad teórica indicada. Agregar los casos de prueba necesarios para dicha corroboración. Esta corroboración empírica debe realizarse confeccionando gráficos correspondientes, y utilizando la técnica de cuadrados mínimos.
- Agregar cualquier conclusión que les parezca relevante.

2. Análisis del problema

Como asesores del Señor del Fuego, se nos ha dado la misión de diseñar un algoritmo capaz de determinar el orden optimo en que su ejército debe enfrentar una serie de n batallas en distintas regiones del mundo y con el que se busca minimizar la suma ponderada de los tiempos de finalización, expresada mediante la siguiente formula:

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

Donde:

- b_i es la importancia de la batalla i.
- F_i es el tiempo de finalización de la batalla i.

Cada enfrentamiento requiere un tiempo especifico t_i para ser completado. Dado que el ejercito ataca de manera conjunta, las batallas se desarrollan de forma secuencial. Esto implica que si la batalla i se realiza en primer lugar, entonces $F_i = t_i$; en cambio, si se realiza inmediatamente despues de la batalla j, se cumple que $F_i = F_j + t_i$.

3. Algoritmo propuesto

Previo al análisis de la solución, resulta necesario presentar la idea sobre la cual se construirá esta misma: **algoritmos greedy**.

3.1. ¿Qué es un algoritmo greedy?

Un algoritmo greedy es aquel en el que, en cada paso, se aplica una regla sencilla que permite obtener una solución óptima local en el estado actual. Este procedimiento se repite de manera iterativa con el objetivo de que la sucesion de óptimos locales conduzca finalmente a una solución óptima global.

3.2. Ordenar de menor a mayor relación $\frac{t_i}{b_i}$

Nuestra idea es priorizar las batallas según la relación de tiempo de duración y su peso de importancia, es decir, elegiremos primero aquellas con menor $\frac{t_i}{b_i}$. De esta forma, en cada paso aplicamos la regla greedy: seleccionar la batalla que aporte menor costo temporal por unidad de importancia.

Por ejemplo, supongamos que tenemos las siguientes batallas:

$$(t_1 = 3, b_1 = 1) \qquad (t_2 = 3, b_2 = 10) \quad \text{y} \quad (t_3 = 1, b_3 = 2).$$

Ordenarlas por este criterio:

$$\frac{t_1}{b_1} = \frac{3}{1} = 3 \quad \frac{t_2}{b_2} = \frac{3}{10} = 0,3 \quad \frac{t_3}{b_3} = \frac{1}{2} = 0,5$$
$$(t_2 = 3, b_2 = 10) \qquad (t_3 = 1, b_3 = 2) \quad \text{y} \quad (t_1 = 3, b_1 = 1).$$

Tenemos la siguiente solución:

$$F_2 = 3 \qquad F_3 = 1 + 3 = 4 \quad \text{y} \quad F_1 = 4 + 3 = 7$$

$$\sum_{i=1}^{3} b_i F_i = 1 \cdot 7 + 10 \cdot 3 + 2 \cdot 4 = 7 + 30 + 8 = 45$$

A continuación, desarrollaremos con mayor profundidad este algoritmo y explicaremos por qué consideramos que devuelve siempre la solución optima.

3.3. Implementación del algoritmo

```python
def organizar_batallas(batallas): #Batallas es un array de (t_i,b_i)
    suma_ponderada = 0
    tiempo_fin_actual= 0

    batallas_ordenadas = merge_sort(batallas)

    for batalla in batallas_ordenadas:
        tiempo, peso = batalla
        tiempo_fin_actual += tiempo
        suma_ponderada += peso * tiempo_fin_actual

    return batallas_ordenadas, suma_ponderada
```

Donde `merge_sort` tiene la siguiente implementación:

```python
def merge_sort(arreglo):
    if len(arreglo)<=1:
        return arreglo

    medio = len(arreglo)//2

    izq = merge_sort(arreglo[:medio])
    der = merge_sort(arreglo[medio:])

    return merge(izq, der)

def merge(izq, der):
    i=j=0
    res= []

    while i < len(izq) and j < len(der):
        tiempo_izq, peso_izq = izq[i]
        tiempo_der, peso_der = der[j]
        if tiempo_izq/peso_izq <= tiempo_der/peso_der:
            res.append(izq[i])
            i+=1
        else:
            res.append(der[j])
            j+=1

    while i < len(izq):
        res.append(izq[i])
        i+=1

    while j < len(der):
        res.append(der[j])
        j+=1

    return res
```

3.4. Complejidad

1. Empezaremos analizando la complejidad de nuestro algoritmo elegido con el ordenamiento de las batallas, que se realiza con el algoritmo de `merge_sort`.

```python
    batallas_ordenadas = merge_sort(batallas)
```

```python
def merge_sort(arreglo):
    if len(arreglo)<=1:
        return arreglo

    medio = len(arreglo)//2

    izq = merge_sort(arreglo[:medio])
    der = merge_sort(arreglo[medio:])

    return merge(izq, der)
```

Este algoritmo implementa un enfoque de división y conquista, lo que significa que resuelve el problema dividiéndolo en problemas más pequeños, resolviendo cada uno de ellos de forma recursiva y luego combinando sus soluciones. Para analizar su complejidad, podemos plantear la ecuación de recurrencia correspondiente. Si tomamos a n como el tamaño del arreglo `batallas` entonces:

$$\mathcal{T}(n) = 2\mathcal{T}\left(\frac{n}{2}\right) + \mathcal{O}(n)$$

donde

- $A = 2$ es la cantidad de llamadas recursivas .
- $B = 2$ ya que en cada llamada recursiva el tamaño del subproblema se reduce a la mitad.
- $\mathcal{O}(n)$ representa el costo de juntar las soluciones de los subproblemas. En este caso, la función `merge` recorre ambos subarreglos, lo que implica un costo lineal. Por lo tanto, $C = 1$.

Aplicando el Teorema Maestro, nos encontramos en el segundo caso, donde se cumple que:

$$\log_2(2) = 1 = C \ \rightarrow \ \mathcal{T}(n) = \mathcal{O}(n^C \cdot log(n))$$

De este modo, la complejidad del ordenamiento resulta: $\mathcal{O}(n \cdot log(n))$

2. Lo siguiente a analizar en complejidad es la iteración sobre las batallas ordenadas:

```python
    for batalla in batallas_ordenadas:
        tiempo, peso = batalla
        tiempo_fin_actual += tiempo
        suma_ponderada += peso * tiempo_fin_actual
```

En este bucle se recorren todas las batallas solo una vez, y en cada iteración se realizan operaciones aritméticas $\mathcal{O}(1)$. Como el arreglo `batallas_ordenadas` tiene n elementos, esto implica una complejidad de $\mathcal{O}(n)$.

Entonces la complejidad de nuestro algoritmo propuesto es:

$$\mathcal{O}(n \cdot log(n)) + \mathcal{O}(n) = \mathcal{O}(n \cdot log(n))$$

3.5. ¿Es un algoritmo greedy?

El algoritmo propuesto es greedy porque aplicamos una regla que nos permite obtener el óptimo local en cada paso hasta obtener la solución completa. En cada paso seleccionamos la batalla que, en proporción a su duración y su importancia, genere el menor 'costo' en la suma ponderada de los tiempos de finalización. De esta manera, el algoritmo avanza tomando la decisión más conveniente en ese momento.

3.6. ¿Obtiene siempre la solución optima?

El algoritmo obtiene siempre la solución óptima. Esto se da ya que la manera en que se ordenan las batallas refleja exactamente cómo cada una de ellas contribuye al costo. Es decir, si una batalla

con menor cociente $\frac{t_i}{b_i}$ se deja para después que otra con un cociente mayor, el resultado es un aumento en la suma ponderada de los tiempos de finalización. Por lo tanto, cuando se respeta el orden según $\frac{t_i}{b_i}$, nos aseguramos de que cada batalla minimice su impacto sobre el costo final, lo cual resulta en que la solución obtenida sea la mejor posible en todos los casos.

3.6.1. Demostración de optimalidad del algoritmo

Queremos demostrar que el algoritmo que ordena las batallas según el cociente $\frac{t_i}{b_i}$ en orden ascendente es óptimo, es decir, minimiza la suma ponderada de los tiempos de finalización:

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

a) Caso base: $n = 1$

Si hay una sola batalla, el orden es único. La suma ponderada es simplemente:

$$C = b_1 F_1 = b_1 t_1$$

y claramente no hay forma de mejorar el orden. Por lo tanto, el algoritmo greedy es óptimo para $n = 1$.

b) Hipótesis inductiva

Supongamos que para un conjunto de k batallas, el algoritmo greedy (ordenar por $\frac{t}{b}$ en orden ascendente) produce un orden que minimiza el costo de la sumatoria.

Es decir, asumimos que el orden:

$$\frac{t_1}{b_1} \leq \frac{t_2}{b_2} \leq \cdots \leq \frac{t_k}{b_k}$$

da la solución óptima.

c) Paso inductivo: $n = k + 1$

Ahora consideremos el caso de $k + 1$ batallas.

1. Reducción a comparación de pares

Si logramos demostrar que para cualquier par de batallas i y j, el orden correcto es poner primero la que tenga menor cociente $\frac{t}{b}$, entonces automáticamente todo el conjunto estará bien ordenado.

2. Comparación de dos batallas

Sea un orden donde la batalla i va antes que la j.

$$\frac{t_i}{b_i} \leq \frac{t_j}{b_j} \quad \Rightarrow \quad t_i b_j \leq t_j b_i$$

- Si el orden es $i \to j$, la contribución de estas dos batallas al costo total es:

$$C_{i \to j} = b_i F_i + b_j F_j = b_i t_i + b_j (t_i + t_j) = b_i t_i + b_j t_i + b_j t_j$$

- Si el orden es $j \to i$, la contribución es:

$$C_{j \to i} = b_i F_i + b_j F_j = b_i (t_j + t_i) + b_j t_j = b_i t_j + b_i t_i + b_j t_j$$

3. ¿Cuál orden conviene?

Queremos que $C_{i\to j} \leq C_{j\to i}$. Restando ambas expresiones, obtenemos:

$$C_{i\to j} - C_{j\to i} = b_j t_i - b_i t_j$$

Como se mencionó antes $b_j t_i \leq b_i t_j$, por lo que se puede concluir que la suma ponderada al realizar la batalla i antes que la j es siempre menor, o a lo sumo igual, que la suma ponderada al realizar la batalla j antes que la i.

Es decir, que cualquier intercambio que se contraponga al orden definido por la relación $\frac{t}{b}$ no mejora la solución, y mantener el orden impuesto por la regla greedy asegura que cada batalla se posicione de manera tal que se minimice su impacto sobre la suma ponderada total.

4. Inserción de la nueva batalla Si ya tenemos k batallas ordenadas óptimamente por $\frac{t}{b}$, al agregar la batalla número $k+1$ basta con insertarla en la posición correcta según su cociente. La comparación par a par garantiza que no habrá ningún intercambio que pueda mejorar la solución.

Por el principio de inducción matemática, queda demostrado que para cualquier número n de batallas, el orden que minimiza la suma ponderada de los tiempos de finalización es el que resulta de ordenar todas las batallas en forma ascendente por $\frac{t}{b}$.

4. Mediciones

Para medir la complejidad temporal del algoritmo implementado, se realizaron 10 ejecuciones por cada tamaño de entrada para reducir el error en la medición.

Se evaluaron tamaños de entrada desde 100 hasta 500,000 batallas, tomando 20 puntos de medición distribuidos uniformemente en este rango. Para cada tamaño, se generaron batallas aleatorias con valores de tiempo y peso entre 1 y 1000.

Mediante la función `curve_fit` de scipy, se obtuvieron los coeficientes que optimizan el ajuste de dos funciones de prueba tomadas como candidatas para los tiempos:

$$\mathcal{O}(n \cdot \log(n)) : \quad \mathcal{T}(n) = c_1 \cdot n \cdot \log(n) + c_2 \tag{1}$$

$$\mathcal{O}(n^2) : \quad \mathcal{T}(n) = c_1 \cdot n^2 + c_2 \tag{2}$$

Los resultados del análisis se presentan en los siguientes gráficos.

4.1. Comparación de Ajustes

El gráfico presenta la superposición de los tiempos de ejecución medidos con ambas funciones teóricas ajustadas, revelando diferencias importantes. Mientras que la curva $\mathcal{O}(n \cdot \log(n))$ prácticamente coincide con los puntos de la medición en todo el dominio, el ajuste cuadrático $\mathcal{O}(n^2)$ muestra divergencia conforme aumenta el tamaño de entrada.

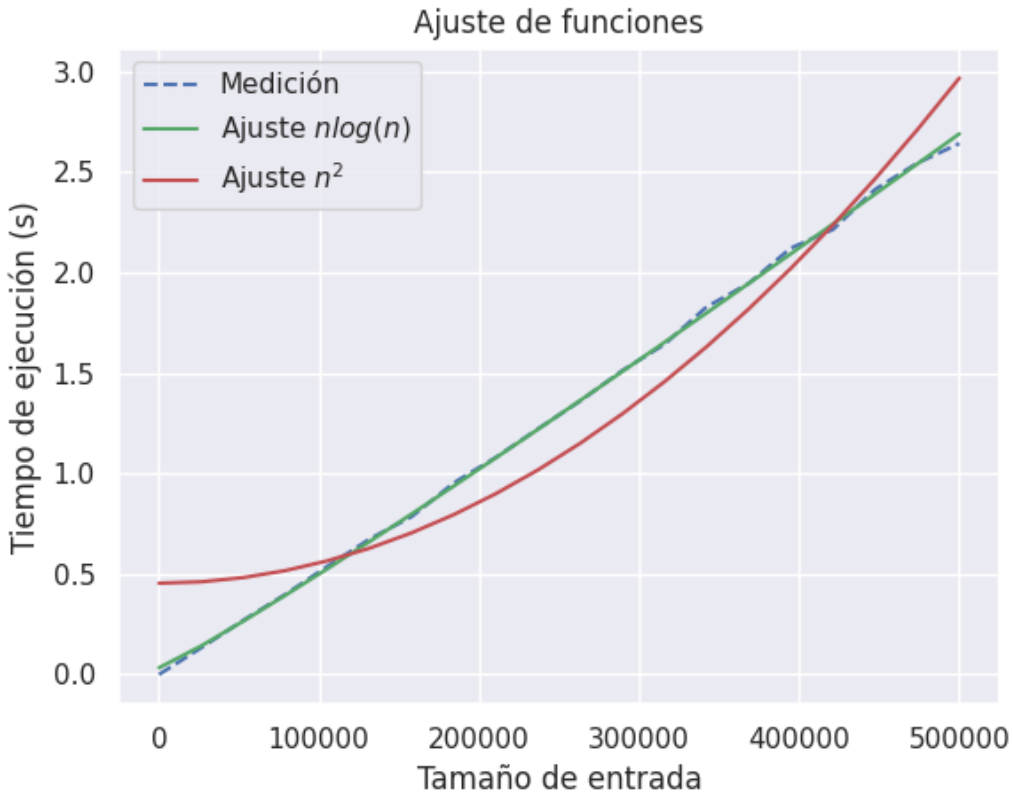

Figura 1: Comparación entre tiempos medidos en función del tamaño de entrada frente ajustes teóricos para $\mathcal{O}(n \cdot \log n)$ y $\mathcal{O}(n^2)$.

Complejidad	c_1	c_2
$\mathcal{O}(n \cdot \log(n))$	4.0485836619850475e-07	0.03341418233304693
$\mathcal{O}(n^2)$	1.0050526201161018e-11	0.45427460116155643

Cuadro 1: Coeficientes obtenidos para cada ajuste

4.2. Análisis de Error

Para determinar cuál de los dos ajustes representa mejor el comportamiento del algoritmo, se calculó el error absoluto entre cada ajuste teórico y los valores medidos.

$$\text{Error}_{n \log n}(n) = |c_1 \cdot n \cdot \log(n) + c_2 - T_{\text{medido}}(n)| \tag{3}$$

$$\text{Error}_{n^2}(n) = |c_1 \cdot n^2 + c_2 - T_{\text{medido}}(n)| \tag{4}$$

donde $T_{\text{medido}}(n)$ representa el tiempo de ejecución promedio medido para un tamaño de entrada n.

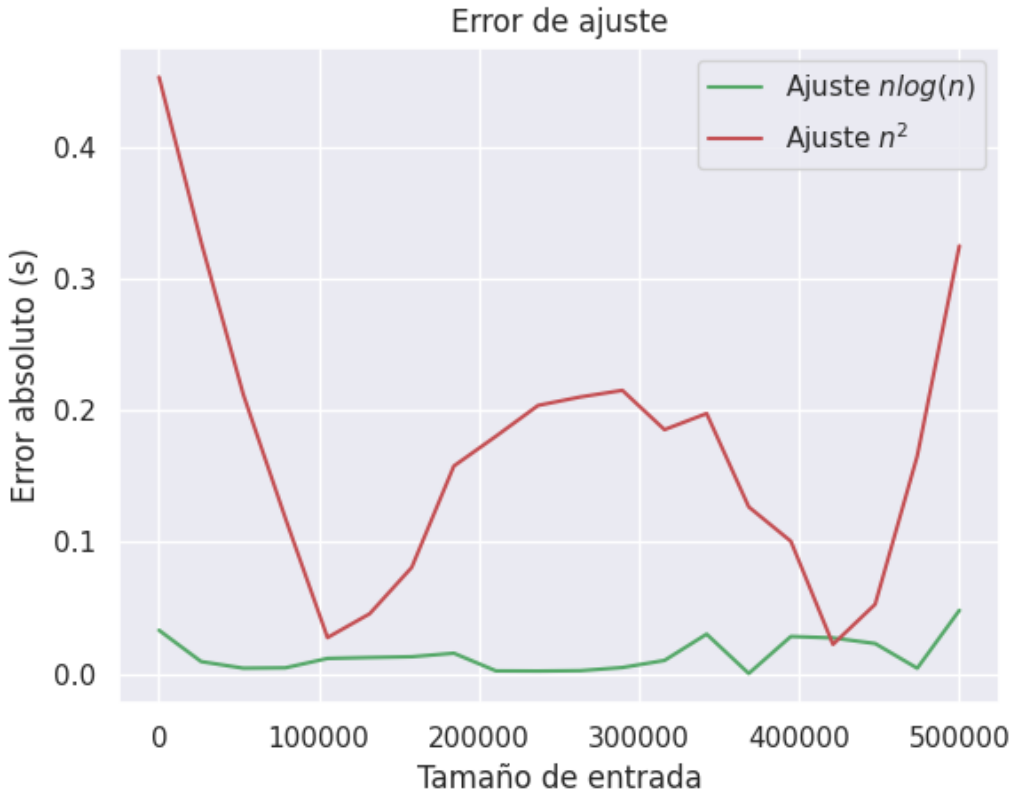

Figura 2: Error absoluto de los ajustes $\mathcal{O}(n \cdot \log(n))$ y $\mathcal{O}(n^2)$ respecto a los tamaños de entrada

Los resultados muestran que el ajuste $\mathcal{O}(n \cdot \log(n))$ presenta un error significativamente menor que el ajuste $\mathcal{O}(n^2)$ en todo el rango evaluado, lo que indica que la complejidad en tiempo del algoritmo implementado es efectivamente $\mathcal{O}(n \cdot \log(n))$.

5. Conclusiones

A lo largo de este trabajo analizamos el problema de determinar el orden óptimo de las batallas que debe enfrentar el ejercito del Señor del Fuego con el objetivo de minimizar la suma ponderada de los tiempos de finalización. Tras el análisis del problema, se propuso un algoritmo *greedy* basado en ordenar las batallas de acuerdo con la relación t/b de manera creciente. A partir de este criterio, se demostró utilizando un razonamiento por intercambio y el principio de inducción matemática, que el algoritmo encuentra siempre la solución óptima.

El análisis de complejidad mostró que el algoritmo tiene un costo temporal de $\mathcal{O}(n \cdot \log(n))$, determinado por el ordenamiento de batallas a través de *merge sort*. Lo perteneciente al calculo de la suma ponderada es lineal y no altera esta cota. Por lo tanto, el algoritmo resulta eficiente y presenta una buena escalabilidad, lo que le permite procesar sin problemas grandes cantidades de datos.

Las mediciones realizadas respaldan la eficiencia del algoritmo. Se evaluaron entradas de hasta 500.000 batallas, y los tiempos obtenidos se compararon con las funciones teóricas $\mathcal{O}(n \cdot \log(n))$ y $\mathcal{O}(n^2)$. Mientras que la curva $\mathcal{O}(n \cdot \log(n))$ coincide prácticamente con los datos medidos, la función cuadrática $\mathcal{O}(n^2)$ se aleja significativamente a medida que aumenta el tamaño de los datos. Esto confirma que la implementación se comporta según lo esperado por el análisis de complejidad.

Además, los casos extremos analizados muestran claramente la relevancia del orden de las batallas, y como un orden incorrecto puede aumentar drásticamente la suma ponderada en comparación a la resultante con el orden óptimo.

En conclusión, el algoritmo presentado garantiza la minimización de la suma ponderada de los tiempos de finalización de las batallas. Las mediciones realizadas muestran que, incluso para un numero grande de batallas, el tiempo de ejecución se ajusta al esperado teniendo una complejidad de $\mathcal{O}(n \cdot \log(n))$, confirmando la eficiencia del algoritmo. Esto demuestra que el algoritmo es preciso, consistente, y capaz de manejar entradas de distintos tamaños sin afectar su rendimiento.