Teoría de Algoritmos

(75.29) Curso Buchwald - Genender

Trabajo Práctico 1

18 de septiembre de 2025

Federico Codino
103533

Juan Ignacio Brugger
103583

1. Introducción

El objetivo principal de este trabajo práctico es asistir al Señor del Fuego para que pueda ganar todas las batallas de manera óptima. Para lograrlo, se deben considerar dos factores:

- t_i: el tiempo necesario para ganar la batalla i.
- b_i: la importancia de la batalla i.

Cada victoria incrementa la felicidad del pueblo de la Nación del Fuego, la cual se denota como F. Esta felicidad se calcula en función del tiempo en que se obtiene la victoria, siguiendo la relación:

$$F_j = F_i + t_j \tag{1}$$

donde F_j representa la felicidad acumulada después de ganar la batalla j y F_i la felicidad acumulada hasta la batalla i.

Dadas estas características, se busca determinar un orden de las batallas que permita **minimizar la suma ponderada de los tiempos de finalización**, expresada matemáticamente como:

$$\text{mín} \sum_{i=1}^{n} b_i \cdot F_i \tag{2}$$

donde n es el número total de batallas.

2. Análisis

2.1. Supuestos

Para el abordaje de este problema con un algoritmo que sea greedy y a la vez optimo, es necesario establecer los supuestos del problema:

- Las batallas son independientes, es decir que no hay un orden establecidos o momento para librarlas.
- Se asume que las batallas siempre terminaran en una victoria para la nación del fuego.
- El ejército no se puede dividir en múltiples frentes de batalla, solo librara una batalla a la vez.
- Se asume que los tiempos de duración t_i y la importancia de las batallas b_i son estrictamente positivos.
- Todas las batallas están disponibles desde el inicio.
- El tiempo e importancia de las batallas son valores enteros.

2.2. Regla Greedy

Dado que el objetivo del problema es minimizar la suma ponderada de los tiempos de finalización, inicialmente se consideró que sería necesario reducir el efecto de los factores t_i o b_i. Es decir, se planteó la idea de ordenar las batallas según los valores más pequeños de tiempo o de importancia.

Sin embargo, tras analizar distintas estrategias y evaluar los resultados, se llegó a la regla *greedy* basada en el cociente t_i/b_i. Según esta regla, el arreglo de batallas se ordena de manera creciente por

t_i/b_i, de forma que las batallas con menor tiempo por unidad de importancia se ejecuten primero. Esta estrategia permite priorizar aquellas batallas que generan menor "impacto por unidad de peso", asegurando que cada decisión local contribuya a minimizar la suma ponderada global y conduciendo así a una solución óptima del problema.

2.3. Demostración de optimalidad

Criterio greedy: ordenar las batallas por $\frac{t_i}{b_i}$ en orden creciente.

Nuestro objetivo es minimizar la suma ponderada de los tiempos de finalización:

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

Para comprobar que la regla greedy nos dirige a un resultado optimo, analizamos el caso donde hay dos batallas (A y B) y sus dos órdenes posibles:

- A antes que B:

$$F_A = t_a, \quad F_B = t_a + t_b$$
$$C_{AB} = b_a \cdot t_a + b_b \cdot (t_a + t_b)$$

- B antes que A:

$$F_B = t_b, \quad F_A = t_b + t_a$$
$$C_{BA} = b_b \cdot t_b + b_a \cdot (t_b + t_a)$$

Diferencia de costos:

$$\Delta = C_{BA} - C_{AB} = b_a \cdot t_b - b_b \cdot t_a$$

Este resultado nos permite decidir qué orden es mejor:

- Si $\Delta > 0 \Rightarrow$ conviene que el orden sea B $\rightarrow$ A.
- Si $\Delta < 0 \Rightarrow$ conviene que el orden sea A $\rightarrow$ B.

Si planteamos la inecuación de ambos casos:

$$\Delta = b_a \cdot t_b - b_b \cdot t_a > 0 \quad \Leftrightarrow \quad b_a \cdot t_b > b_b \cdot t_a \quad \Leftrightarrow \quad \frac{t_a}{b_a} > \frac{t_b}{b_b}$$

$$\Delta = b_a \cdot t_b - b_b \cdot t_a < 0 \quad \Leftrightarrow \quad b_a \cdot t_b > b_b \cdot t_a \quad \Leftrightarrow \quad \frac{t_a}{b_a} < \frac{t_b}{b_b}$$

Por lo tanto, este análisis confirma que el orden óptimo es aquel que pone primero la batalla con menor cociente $\frac{t}{b}$, exactamente como lo establece el algoritmo greedy.

Conclusión: Aun en el caso particular donde $t_a > t_b$ y $b_a > b_b$, la decisión correcta no depende directamente de cuál batalla es más importante o cuál es más corta, sino de cuál tiene menor cociente $\frac{t}{b}$. Esto confirma que el algoritmo greedy es correcto y produce una solución óptima también en este escenario.

3. Descripción del algoritmo

A continuación se realizaremos el análisis del algoritmo implementado en lenguaje python, describiendo su comportamiento y calculando su complejidad.

3.1. Algoritmo

A continuación se muestra el código de todo el algoritmo.

```python
import sys

# Complejidad: O(n)
def cargar_datos(ruta_archivo):
    with open(ruta_archivo, "r") as f:
        filas = f.read().strip().splitlines()[1:]  # salto el header directamente

    datos = []
    for fila in filas:
        partes = fila.split(",")
        tiempo = int(partes[0])
        peso = int(partes[1])
        datos.append([tiempo, peso])

    return datos

# Complejidad: O(n)
def calcular_impacto(registros):
    total_tiempo = 0
    impacto = 0
    for tiempo, peso in registros:
        total_tiempo += tiempo
        impacto += total_tiempo * peso
    return impacto


# Complejidad: O(n log n)
def mejor_orden_greedy(ruta_archivo):
    registros = cargar_datos(ruta_archivo)  # O(n)
    # ordenar por T_i / B_i
    registros.sort(key=lambda par: par[0] / par[1])  # O(n log n)
    impacto = calcular_impacto(registros)  # O(n)
    return registros, impacto


def main():
    ruta_archivo = sys.argv[1]
    orden, impacto = mejor_orden_greedy(ruta_archivo)

    print(f"El orden las batallas es: {orden}")
    print(f"Coeficiente de impacto: {impacto}")
    return impacto


if __name__ == "__main__":
    main()
```

A continuación, vamos a realizar un análisis por partes, para calcular la complejidad.

3.2. Función principal

A continuación se muestra el código de la función main.

```python
def main():
    ruta_archivo = sys.argv[1]
    orden, impacto = mejor_orden_greedy(ruta_archivo)

    print(f"El orden las batallas es: {orden}")
```

```
    print(f"Coeficiente de impacto: {impacto}")
    return impacto
```

La función se encarga de usar la ruta de archivo para llamar a la función `mejor_orden_greedy()`, a su vez muestra por pantalla el orden optimo de las batallas y el impacto total proporcionado por la misma. La complejidad de esta función sera igual a la de `mejor_orden_greedy()` ya que el resto son operaciones de tiempo constante.

3.3. Función mejor orden greedy

Esta función se encarga de extraer la información de las batallas de los registros usando la función `cargar_datos()` para luego ordenarlos de manera creciente según el indice que describimos en la regla greedy .

$$\frac{t}{b}$$

Luego con las batallas procede a calcular el impacto, usando la función `calcular_impacto()` , para finalmente devolver los registros, que ordenados de esa manera son el orden optimo de las batallas y el impacto total, previamente calculado.

```
def mejor_orden_greedy(ruta_archivo):
    registros = cargar_datos(ruta_archivo)
    registros.sort(key=lambda par: par[0] / par[1])
    impacto = calcular_impacto(registros)
    return registros, impacto
```

La complejidad de esta función sera la de la función con la complejidad mas alta entre las funciones llamadas:`cargar_datos()`, sort de python y `calcular_impacto()`. Esto debido a que todas son llamadas secuencialmente sin estar anidadas, se sabe que sort en python es O(n.log(n)).

3.4. Función cargar datos

La función se encarga de abrir y leer cada linea del archivo de las batallas, luego separa cada linea con comas para obtener las variables "tiempoz "peso", para agregarlas a un arreglo que sera finalmente devuelto.

```
def cargar_datos(ruta_archivo):
    with open(ruta_archivo, "r") as f:
        filas = f.read().strip().splitlines()[1:]

    datos = []
    for fila in filas:
        partes = fila.split(",")
        tiempo = int(partes[0])
        peso = int(partes[1])
        datos.append([tiempo, peso])

    return datos
```

La complejidad de esta función depende de dos partes, leer cada carácter del registro y recorrer todas las lineas (que representan las batallas) dividiéndolas en tiempo y peso para agregándolas al arreglo. Las operaciones que se realizan por cada fila son de tiempo constante, por lo que aunque están anidadas, no modificaran el resultado.

$$\mathcal{T}(n) = \mathcal{O}(m) + \mathcal{O}(n)$$

Como podemos ver la complejidad depende de m siendo la cantidad de caracteres del archivo o n siendo la cantidad de batallas , por lo que resultaría en $\mathcal{O}(n)$.

3.5. Función calcular impacto

Se encarga de recorrer todas las batallas y calcular el impacto total en base a la formula que analizamos previamente: $\sum_{i=1}^{n} b_i \cdot F_i$

```
def calcular_impacto(registros):
    total_tiempo = 0
    impacto = 0
    for tiempo, peso in registros:
        total_tiempo += tiempo
        impacto += total_tiempo * peso
    return impacto
```

La complejidad de esta función es $\mathcal{O}(n)$, ya que por cada batalla hace un calculo de tiempo constante

3.6. Análisis de complejidad

Como analizamos anteriormente, la complejidad de este algoritmo esta definida por su función main, que a su vez esta definida por `mejor_orden_greedy()`, la cual depende de `cargar_datos()`, sort de python y `calcular_impacto()`. Con la información que tenmos podemos hacer el calculo:

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

Resultando en que la complejidad de este algoritmo es:

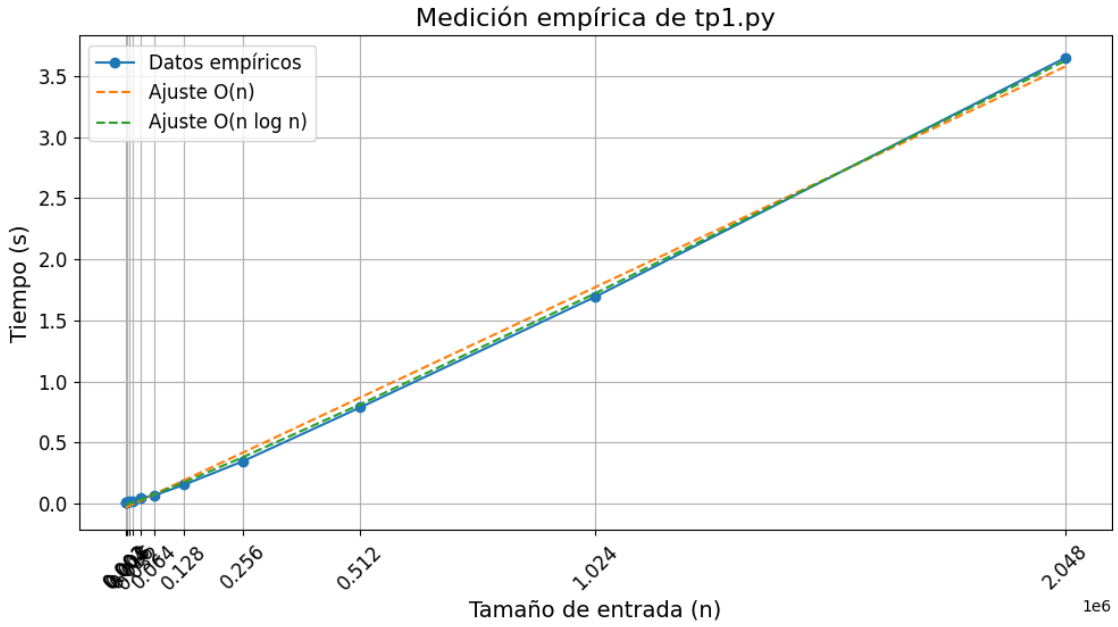
$$\mathcal{O}(n.log(n))$$

Como podemos ver lo que determina la complejidad es el ordenamiento de las batallas.

4. Mediciones

Se realizaron mediciones en base a crear datos de diferentes tamaños. Los tamaños de los datos fueron: 1000, 2000, 4000, 8000, 16000, 32000, 64000, 128000, 256000, 512000, 1024000 y 2048000 de elementos. Donde los elementos en cada caso fueron generados por los valores pseudoaleatorios del lenguaje (el módulo `random`) con la seed = 42.

Además, se agrego un ajuste por cuadrados mínimos a la curva de la complejidad O(n) y O(n log n)



Como se puede apreciar, el algoritmo se ajusta mejor a la curva O(n log n), por lo tanto podemos decir que la complejidad obtenida en la practica coincide con la complejidad teórica calculada en la sección anterior. Esto implica que la complejidad de al algoritmo es proporcional en (n log n) al tamaño de entrada de los datos.

5. Conclusiones

En este trabajo pudimos brindar un algoritmo, del tipo greedy, capaz de cumplir con la consigna de ordenar las batallas para el señor del fuego, de tal manera que el tiempo para librarlas se reduzca al mínimo. Pudimos demostrar formalmente que el algoritmo expuesto es optimo y de una complejidad temporal de O(n.log(n)), la cual coincide con los datos obtenidos en las mediciones.

Este algoritmo nos deja ver que la búsqueda de un algoritmo greedy, de óptimos locales, puede resultar en un optimo global, como bien es descrito en su definición.