

TEORÍA DE ALGORITMOS  
(75.29) CURSO BUCHWALD - GENENDER

# Trabajo Práctico 1

## Algoritmos Greedy en la Nación del Fuego

A horizontal progress bar consisting of small square icons. The first 18 squares are dark grey, indicating completed items, while the remaining 2 squares at the end are light grey.

18 de septiembre de 2025

Alan Turing  
101010

Barbara Liskov  
111111

Alan Turing  
101010

Barbara Liskov  
111111

## Índice

<b>1. Introducción</b>	<b>3</b>
1.1. Consigna . . . . .	3
<b>2. Análisis del problema</b>	<b>4</b>
<b>3. Algoritmo propuesto</b>	<b>4</b>
3.1. ¿Qué es un algoritmo greedy? . . . . .	4
3.2. Ordenar de menor a mayor relación $\frac{t_i}{b_i}$ . . . . .	4
3.3. Implementación del algoritmo . . . . .	5
3.4. Complejidad . . . . .	5
3.5. ¿Es un algoritmo greedy? . . . . .	6
3.6. ¿Obtiene siempre la solución óptima? . . . . .	6
3.6.1. Demostración de optimalidad del algoritmo . . . . .	7
<b>4. Mediciones</b>	<b>8</b>
<b>5. Conclusiones</b>	<b>9</b>
5.1. Análisis de casos extremos . . . . .	9
5.1.1. Analizando 2 batallas . . . . .	9
5.1.2. Analizando 10 batallas . . . . .	9

## 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 óptima** 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 específico  $t_i$  para ser completado. Dado que el ejército 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 después 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 sucesión 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:

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

Ordenarlas por este criterio:

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

Tenemos la siguiente solución:

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

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

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

### 3.3. Implementación del algoritmo

```
1 def organizar_batallas(batallas): #Batallas es un array de (t_i,b_i)
2     suma_ponderada = 0
3     tiempo_fin_actual = 0
4
5     batallas_ordenadas = merge_sort(batallas)
6
7     for batalla in batallas_ordenadas:
8         tiempo, peso = batalla
9         tiempo_fin_actual += tiempo
10        suma_ponderada += peso * tiempo_fin_actual
11
12    return batallas_ordenadas, suma_ponderada
```

Donde merge\_sort tiene la siguiente implementación:

```
1 def merge_sort(arreglo):
2     if len(arreglo) <= 1:
3         return arreglo
4
5     medio = len(arreglo) // 2
6
7     izq = merge_sort(arreglo[:medio])
8     der = merge_sort(arreglo[medio:])
9
10    return merge(izq, der)
11
12 def merge(izq, der):
13     i = j = 0
14     res = []
15
16     while i < len(izq) and j < len(der):
17         tiempo_izq, peso_izq = izq[i]
18         tiempo_der, peso_der = der[j]
19         if tiempo_izq/peso_izq <= tiempo_der/peso_der:
20             res.append(izq[i])
21             i += 1
22         else:
23             res.append(der[j])
24             j += 1
25
26     while i < len(izq):
27         res.append(izq[i])
28         i += 1
29
30     while j < len(der):
31         res.append(der[j])
32         j += 1
33
34    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.

```
1 batallas_ordenadas = merge_sort(batallas)
```

```
1 def merge_sort(arreglo):
2     if len(arreglo) <= 1:
3         return arreglo
4
5     medio = len(arreglo) // 2
6
7     izq = merge_sort(arreglo[:medio])
8     der = merge_sort(arreglo[medio:])
9
10    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:

```
1 for batalla in batallas_ordenadas:
2     tiempo, peso = batalla
3     tiempo_fin_actual += tiempo
4     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 óptima?

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:

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

donde  $F_i$  es el tiempo de finalización de la batalla  $i$ .

#### 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:

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

da la solución óptima.

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

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

**c.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.

**c.2) Comparación de dos batallas** Sea un orden donde la batalla  $i$  va antes que la  $j$ .

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

$$C_{i \rightarrow j} = (b_i + b_j)t_i + b_j t_j$$

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

$$C_{j \rightarrow i} = (b_i + b_j)t_j + b_i t_i$$

**c.3) ¿Cuál orden conviene?** Queremos que  $C_{i \rightarrow j} \leq C_{j \rightarrow i}$ . Restando ambas expresiones, obtenemos:

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

Entonces:

$$C_{i \rightarrow j} \leq C_{j \rightarrow i} \iff \frac{t_i}{b_i} \leq \frac{t_j}{b_j}$$

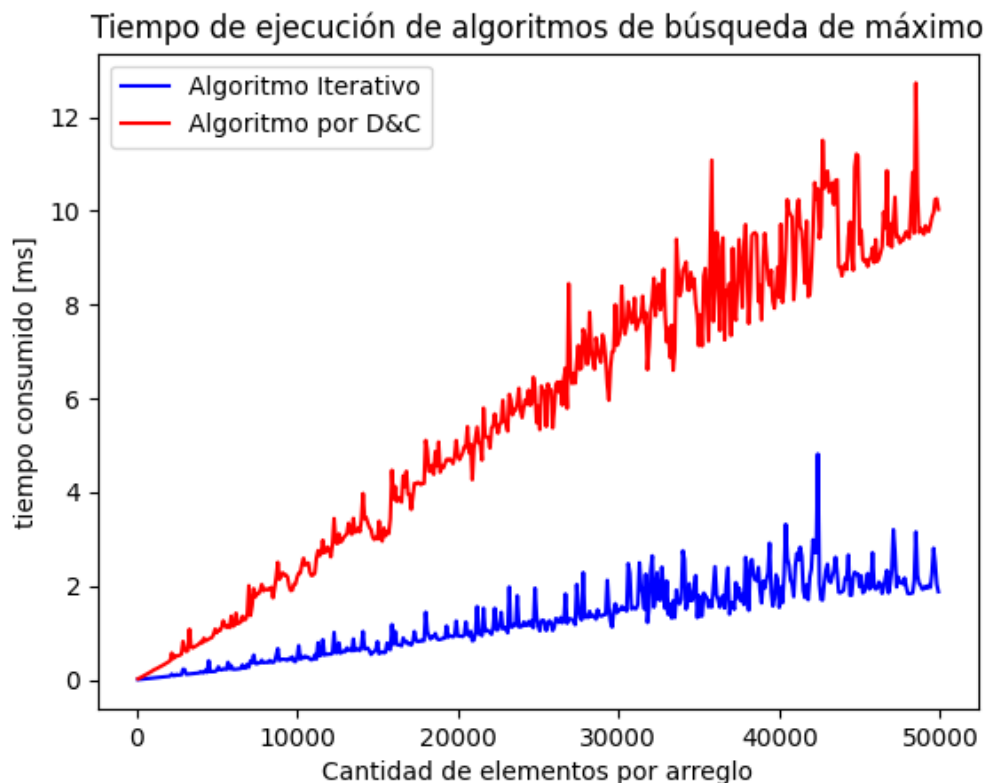
Esto significa que conviene siempre ubicar primero a la batalla con menor cociente  $\frac{t}{b}$ .

**c.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

Se realizaron mediciones en base a crear arreglos de diferentes largos, yendo de 100 en 100 elementos, donde los elementos en cada caso fueron generados por los valores pseudoaleatorios del lenguaje (el módulo `random`).



Como se puede apreciar, ambos algoritmos tienen una tendencia efectivamente lineal en función del tamaño de la entrada, si bien el algoritmo iterativo es más veloz en cuestión de constantes.



## 5. Conclusiones

### 5.1. Analisis de casos extremos

#### 5.1.1. Analizando 2 batallas

$A = (1, 1000) \rightarrow$  tiempo muy corto, peso enorme.

$B = (1000, 1) \rightarrow$  tiempo muy largo, peso chico.

El cociente  $1/1000$  da como resultado 0.001.

El cociente  $1000/1$  da como resultado 1000.

Analizando el orden  $A \rightarrow B$  la suma total queda en 2001.

Analizando el orden  $B \rightarrow A$  la suma total queda en 1,002,000.

#### 5.1.2. Analizando 10 batallas

Batallas Propuestas

- (1,1000)
- (2,800)
- (3,600)
- (4,400)
- (5,200)
- (100,1)
- (200,1)
- (300,1)
- (400,1)
- (500,1)

Analizando el costo optimo para el orden creciente de  $t/b$  obtenemos 17575.

Analizando el peor caso para el orden decreciente de  $t/b$  obtenemos 4,543,500 que es 250 veces mayor.