FACULTAD DE INGENIERIA
Universidad de Buenos Aires

Teoría de Algoritmos

Cátedra Buchwald - Segundo Cuatrimestre 2025

Trabajo Práctico N°1

18 de septiembre de 2025

Alumno	Número de Padrón	Email
ALDONATE, Lucas	100030	laldonate@fi.uba.ar
PETRUCCI, Maximiliano	95872	mpetrucci@fi.uba.ar
THOME, Milagros	110684	mthome@fi.uba.ar
RUIZ DIAZ, Carolina	111442	cruiz@fi.uba.ar

Índice

1. Introducción

El trabajo práctico a desarrollar se puede considerar como un problema Greedy de scheduling. En el escenario que se plantea, existe un conjunto de batallas que deben ejecutarse secuencialmente por un único ejército. Cada batalla i requiere un tiempo t_i y posee una importancia b_i. El orden en que se ejecutan afecta el impacto global.

Nuestro objetivo es encontrar un orden que minimice la suma ponderada de los tiempos de finalización, sabiendo que 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 $F_j = F_i + t_j F_j$. Por lo tanto, la suma ponderada a minimizar es:

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

En este trabajo presentamos un algoritmo Greedy óptimo que ordena por la razón $\frac{b_i}{t_i}$ de manera descendente. Este criterio favorece primero las batallas que tienen mucha importancia relativa en comparación con su tiempo de duración. Demostramos su optimalidad, analizamos su complejidad y validamos empíricamente con datos sintéticos.

2. Algoritmo Propuesto

Planteamos un ordenamiento con la relación $\frac{b_i}{t_i}$ de manera descendente. Esto surge de la idea de si una batalla es muy importante y además dura poco, conviene hacerla antes que otra que tarda más y tiene menos peso. Si solo ordenáramos por tiempo, se ignora la importancia de cada batalla por lo cual podes tardar mas en poner una batalla con mucho valor. Por otro lado, si solo ordenáramos por importancia, pondríamos primero las más "pesadas", pero podría ser que tarden mucho y retrasen a todas las demás. Entonces: b/t balancea ambos criterios:

- Valores altos de b/t: batallas muy importantes en poco tiempo → conviene hacerlas antes.
- Valores bajos de b/t: batallas menos rentables → conviene postergarlas.

Justificación Greedy del algoritmo.
Este algoritmo es Greedy debido a que en cada paso tomamos la decisión que parece la mejor localmente (en este caso: elegir la batalla con mayor b/t). No miramos todas las permutaciones posibles, sino que seguimos un criterio local simple que va construyendo el orden final. Lo que termina resultando en un óptimo global, que es minimizar la suma ponderada.

2.1. Planteo que hicimos:

Consideramos dos batallas i y j , con tiempos ti,tj y pesos bi,bj. Si en una secuencia dada la batalla i aparece antes que la batalla j, el costo parcial de ese orden es:

$$C_{i\to j} = b_i\,(F + t_i) + b_j\,(F + t_i + t_j), \tag{1}$$

mientras que si el orden se invierte (primero j, luego i), el costo parcial es:

$$C_{j\to i} = b_j\,(F + t_j) + b_i\,(F + t_j + t_i). \tag{2}$$

donde F representa el tiempo acumulado de las batallas anteriores. Al comparar las expresiones (1) y (2), calculamos la diferencia entre ambos costos:

$$C_{i\to j} - C_{j\to i} \;=\; b_i(F + t_i) + b_j(F + t_i + t_j) - \big[b_j(F + t_j) + b_i(F + t_j + t_i)\big].$$

Si esta diferencia es menor que cero, significa que $C_{i\to j} < C_{j\to i}$, por lo que resulta más conveniente ubicar la batalla i antes que la j. En cambio, si la diferencia es mayor que cero, conviene el orden inverso.

Al desarrollar y simplificar, obtenemos:

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

De esta expresión se concluye que conviene mantener el orden $i \to j$ si y solo si

$$b_i t_j - b_j t_i \le 0 \iff \frac{b_i}{t_i} \ge \frac{b_j}{t_j}.$$

2.2. Demostración de optimalidad por contradicción

Supongamos, con el fin de llegar a una contradicción, que existe un orden que minimiza

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

pero que no respeta el criterio de ordenar las batallas en forma no creciente según la razón $\frac{b_i}{t_i}$.

Esto implica que debe existir al menos un par de batallas consecutivas i, j tales que

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

pero aun así la batalla i aparece antes que la batalla j en el orden considerado.

Si intercambiamos las posiciones de i y j, el cambio en el costo total es

$$\Delta = (b_i t_j - b_j t_i).$$

Dado que $\frac{b_i}{t_i} < \frac{b_j}{t_j}$, se cumple que $b_i t_j - b_j t_i < 0$, por lo que el intercambio **reduce el valor de la suma ponderada**.

Esto contradice la hipótesis de que el orden inicial era óptimo. Por lo tanto, ningún orden óptimo puede violar el criterio, y la secuencia que respeta el orden no creciente de $\frac{b}{t}$ es siempre óptima.

Ejemplo de Resolución

3. Implementación

En esta sección se presenta el código esencial del algoritmo propuesto. Cada batalla se representa como una tupla (b, t), donde b corresponde a la importancia (peso) de la batalla y t a su tiempo de duración. El objetivo es ordenar las batallas en forma no creciente según la razón b/t.

3.1. Función principal

Ordenar batallas. La siguiente función recibe una lista de batallas y devuelve la lista ordenada en el orden óptimo:

```
def ordenar_batallas(batallas):
    return sorted(batallas, key=lambda x: (x[0] / x[1]), reverse=True)
```

La función `sorted` utiliza un algoritmo de ordenación con complejidad $\mathcal{O}(n \log n)$.

Cálculo de la suma ponderada. A partir del orden obtenido, se calcula el tiempo de finalización acumulado de cada batalla y se evalúa la suma $\sum b_i F_i$:

```
def costo_total(batallas):
    tiempo = 0
    total = 0
    for b, t in batallas:
        tiempo += t
        total += b * tiempo
    return total
```

El cálculo de la suma ponderada es lineal, $\mathcal{O}(n)$. Por lo tanto, la complejidad total del algoritmo es $\mathcal{O}(n \log n)$, dominada por el algoritmo de ordenamiento.

3.2. Batallas de prueba

Se desean ordenar las siguientes batallas

$$(b, t) = [(100, 10),\ (1, 2),\ (5, 3),\ (4, 40)].$$

1) Orden por tiempo (SPT, menor t primero). El orden es: $[(1,2)\ ,(5,3)\ ,(4,40)\ ,(100,10)]$. Los tiempos de finalización son:

$$F_1 = 2$$
$$F_2 = 2 + 3 = 5$$
$$F_3 = 5 + 40 = 45$$
$$F_4 = 45 + 10 = 55$$

La suma ponderada resulta:

$$\sum b_i \cdot F_i = 1 \cdot 2 + 5 \cdot 5 + 100 \cdot 15 + 4 \cdot 55 = \mathbf{1762}.$$

2) Orden por prioridad (mayor b primero). El orden es: $(100,10)\ ,(5,3)\ ,(4,40)\ ,(1,2)$.

$$F_1 = 10$$
$$F_2 = 13$$
$$F_3 = 53$$
$$F_4 = 55$$
$$\sum b_i \cdot F_i = 100 \cdot 10 + 5 \cdot 13 + 4 \cdot 53 + 1 \cdot 55 = \mathbf{1332}.$$

3) Orden por b/t (greedy propuesto). En primer lugar, calculamos el cociente b_i/t_i para cada batalla en el orden dado y ordenamos de manera descendente:

$$[0{,}14,\ 1{,}6,\ 3,\ 0{,}83]$$

$$\frac{b}{t} \approx \{10,\ 0{,}5,\ 1{,}667,\ 1\} \Rightarrow \text{orden: } [(100,10)\ ,(5,3)\ ,(1,2)\ ,(4,40)].$$

A continuación, se calculan los tiempos de finalización acumulados F_i:

$$F_1 = 10$$
$$F_2 = 10 + 3 = 13$$
$$F_3 = 13 + 2 = 15$$
$$F_4 = 15 + 40 = 55$$

Finalmente la suma ponderada es:

$$\implies \sum_{i=1}^{4} b_i \cdot F_i = 100 \cdot 10 \ + \ 5 \cdot 13 \ + \ 1 \cdot 15 \ + \ 4 \cdot 55 = \mathbf{1300}.$$

Conclusión. El criterio de **orden por** b/t logra la menor suma ponderada (1300), superando tanto al orden por prioridad (1332) como al orden por tiempo (1762). Esto confirma que balancear importancia y duración mediante la razón b/t es efectivamente la mejor estrategia.

4. Mediciones

Con el objetivo de validar empíricamente el análisis teórico realizado, se llevaron a cabo mediciones de tiempos de ejecución del algoritmo de ordenamiento propuesto. El propósito de estas pruebas es comprobar que el comportamiento observado en la práctica coincide con la complejidad esperada de $O(n \log n)$ que especificamos anteriormente.

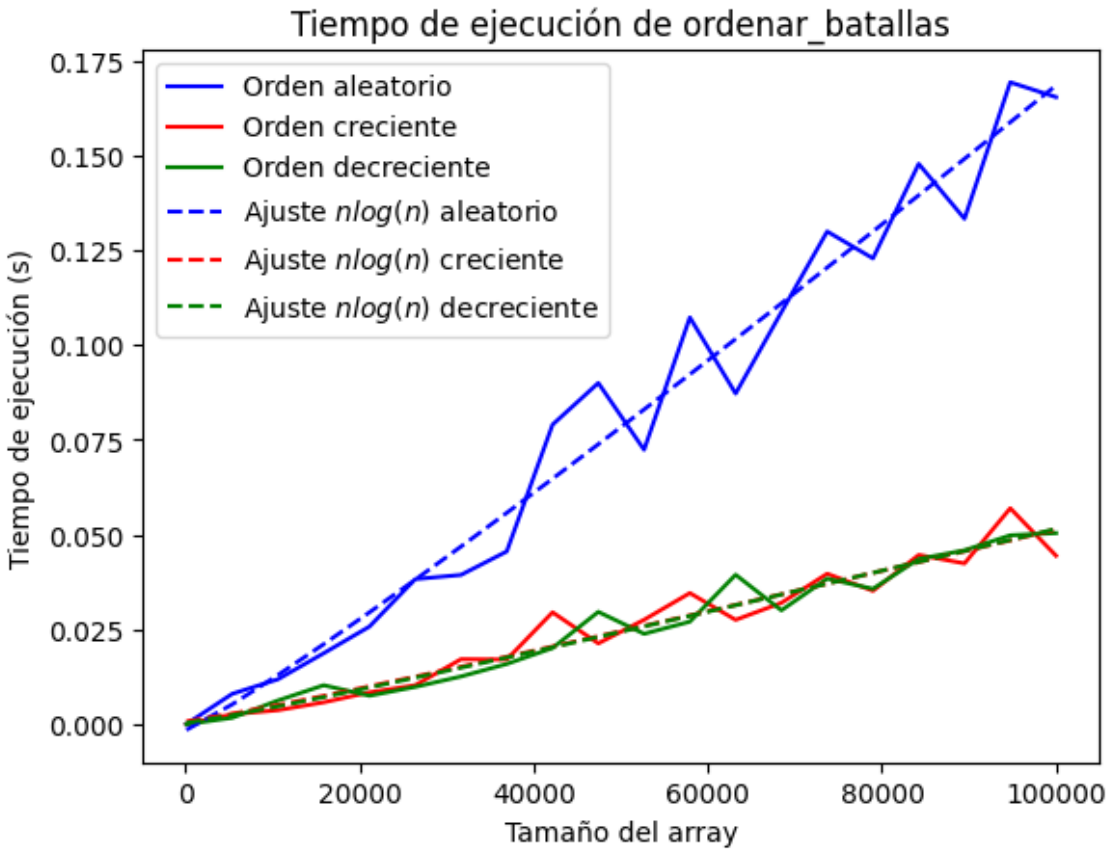
4.1. Tiempos del algoritmo

Las mediciones que hicimos del tiempo de ejecución del algoritmo fueron planteadas con batallas con distintos ordenamientos iniciales y aumentando la cantidad de batallas para evaluar su comportamiento:

- Lista de batallas en orden aleatorio
- Lista de batallas en orden b/t creciente
- Lista de batallas en orden b/t decreciente

La elección de los distintos ordenamientos se basa en buscar los peores y mejores casos del algoritmo para evaluar el comportamiento y el tiempo. Dado que el algoritmo realiza un ordenamiento decreciente se espera que el método, el cual ejecuta un TimSort internamente, responda en tiempos de ejecución más altos para una lista de batallas con orden aleatorio, que para los casos crecientes y decrecientes.

4.2. Evaluación de tiempos

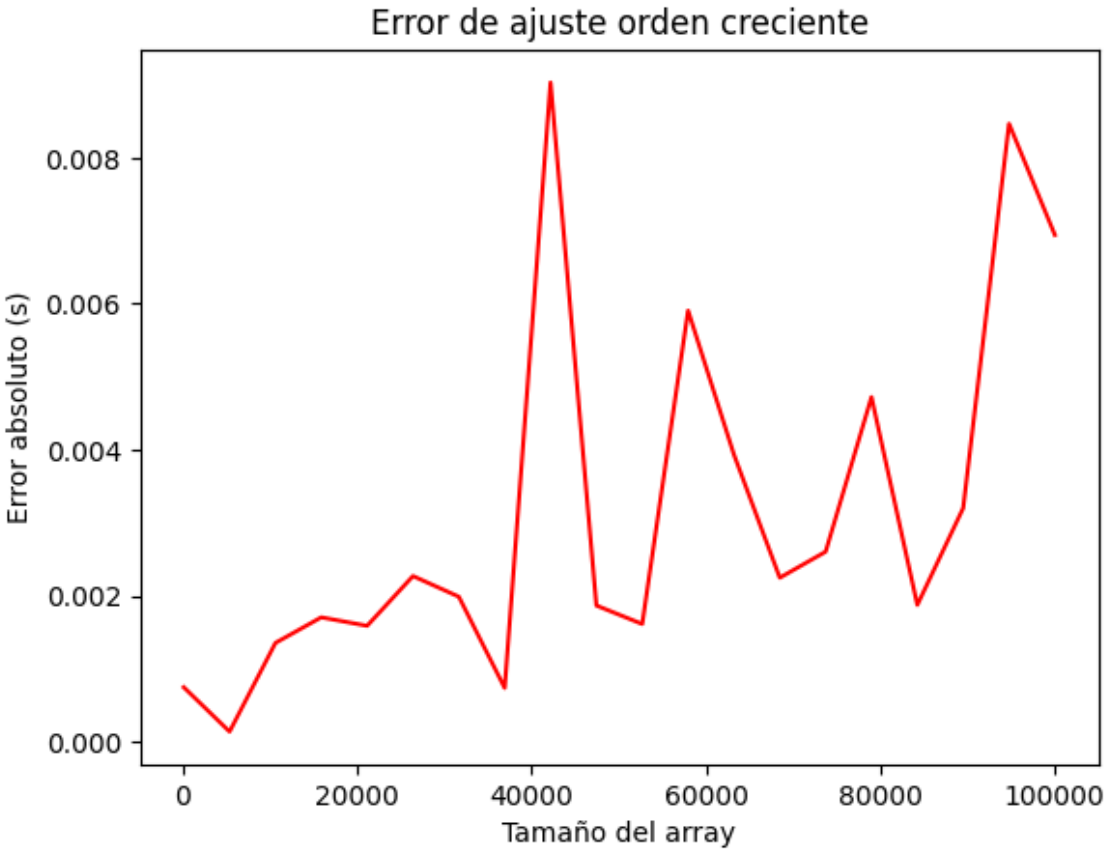


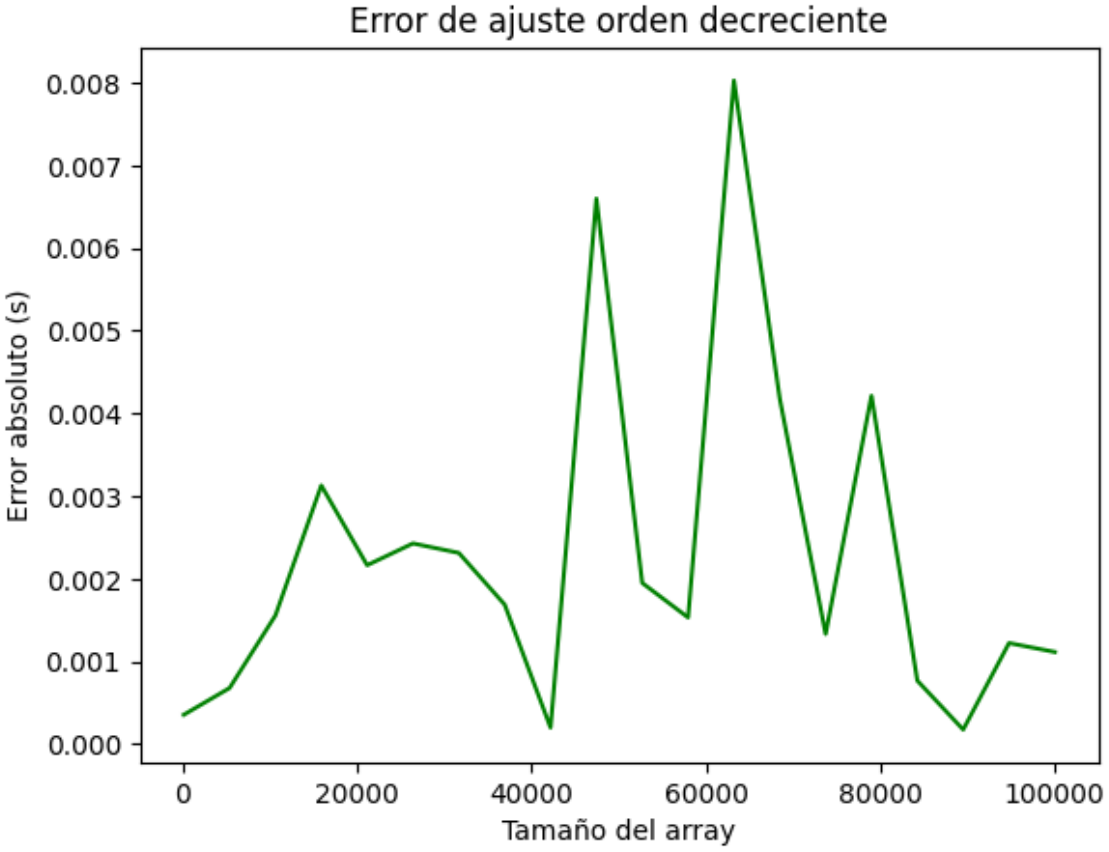
En la imagen se pueden apreciar los distintos tiempos y su ajuste a $n.log(n)$ en base al arreglo de entrada. Se destaca que el orden aleatorio, como era de esperar, tiene mayor tiempo de ejecución que los otros dos casos. Se observa que el caso aleatorio presenta tiempos de ejecución considerablemente más altos, ya que el algoritmo de ordenamiento (TimSort) no puede aprovechar ningún grado de orden previo. En contraste, los casos creciente y decreciente exhiben tiempos mucho menores y similares entre sí, dado que TimSort es capaz de detectar secuencias ya ordenadas y procesarlas eficientemente.

En todos los casos, las curvas empíricas siguen de cerca el comportamiento esperado de $O(n \log n)$, confirmando la complejidad teórica del algoritmo propuesto.

4.3. Evaluación de errores de ajuste

A continuación se realizó un ajuste mediante mínimos cuadrados sobre los tiempos medidos, y se graficó el error absoluto entre los valores experimentales y la curva de ajuste de la forma $c_0\, n \log n + c_1$. El error mostrado en las Figuras corresponde a la diferencia entre los tiempos experimentales y la curva de ajuste utilizando distintos ordenes iniciales del array de batallas.


Error de ajuste orden creciente
Error absoluto (s)
0.000
0.002
0.004
0.006
0.008
0
20000
40000
60000
80000
100000
Tamaño del array



Al observar los gráficos, se aprecia que el caso aleatorio presenta errores más altos (en algunos puntos hasta el doble de los casos crecientes y decrecientes), lo cual se explica por la mayor variabilidad en los tiempos de ejecución. Sin embargo, los errores siempre se mantienen bajos (del orden de milisegundos), independientemente del orden inicial de los datos. Esto nos ayuda a determinar que el algoritmo de ordenamiento de batallas efectivamente se comporta como $O(n.log(n))$.

5. Conclusiones

El problema analizado corresponde a un ejercicio de *greedy scheduling*, para el cual se propone un algoritmo Greedy óptimo que ordena las tareas en forma descendente según la razón b_i/t_i.

A partir del análisis, se observa que ordenar únicamente por tiempo ignora la importancia de las batallas, mientras que ordenar sólo por importancia puede generar retrasos temporales en las posteriores. Para equilibrar ambos escenarios, se prioriza la razón importancia/tiempo (b/t), de modo que las batallas con valores altos se realicen antes y las de valores bajos se posterguen.

De esta forma, el criterio seguido es efectivamente Greedy, ya que cada decisión local contribuye a una solución global óptima al minimizar la suma ponderada. Aparte se llega a que ,mediante el seguimiento de ecuaciones, se concluye que conviene mantener un orden decreciente según b_i/t_i.

Las mediciones realizadas confirman que el algoritmo se comporta según la complejidad teórica O(n log n), ya que los tiempos observados crecen de manera consistente con este ajuste. Aunque el caso aleatorio tarda un poco más que los ordenados, las diferencias son bajas y el rendimiento se mantiene eficiente en todos los escenarios.

En resumen, el algoritmo muestra un desempeño estable y eficiente independientemente del orden inicial de los datos. No solo cumple con ser un algoritmo Greedy si no que también con ser óptimo. Más allá del problema particular, el ejercicio resulta valioso para comprender cómo

un enfoque Greedy puede ser fundamentado rigurosamente, mostrando que decisiones locales bien diseñadas pueden conducir a soluciones globales óptimas.