

TEORÍA DE ALGORITMOS  
(75.29) CURSO BUCHWALD - GENENDER

## Trabajo Práctico 1



Fecha de entrega: 18/09/2023

- Guler, Ian Benjamin, Padrón: 109437
- Jauregui, Melina Belén, Padrón: 109524
- Tourne Passarino, Patricio, Padrón: 108725

# Índice

<b>1</b>	<b>Introducción</b>	<b>3</b>
1.1	Información de la problemática . . . . .	3
<b>2</b>	<b>Análisis de diferentes soluciones propuestas</b>	<b>3</b>
<b>3</b>	<b>Posibles algoritmos</b>	<b>4</b>
3.1	Ordenando por $a_i$ de forma creciente . . . . .	5
3.2	Ordenando por $a_i$ de forma decreciente . . . . .	5
3.3	Resultados obtenidos del Caso 1 . . . . .	5
3.4	Resultados obtenidos del Caso 2 . . . . .	6
<b>4</b>	<b>Demostración</b>	<b>6</b>
4.1	Cota de complejidad temporal . . . . .	8
<b>5</b>	<b>Mediciones</b>	<b>8</b>
<b>6</b>	<b>Conclusiones</b>	<b>10</b>

## 1. Introducción

En el presente informe, se tiene como objetivo llevar a cabo un análisis del diseño implementado para abordar la problemática planteada por Scaloni, que consiste en la optimización máxima del tiempo total empleado en la realización de un análisis exhaustivo de sus rivales. A lo largo de este, se expondrán los criterios empleados y se proporcionará su justificación.

### 1.1. Información de la problemática

El enunciado propuesto, nos da la siguiente información:

- Cada compilado lo debe analizar Scaloni y alguno de sus ayudantes.
- El análisis del rival  $i$  le toma  $s_i$  tiempo a Scaloni y luego  $a_i$  tiempo al ayudante.
- Scaloni cuenta con  $n$  ayudantes, siendo  $n$  la cantidad de rivales a analizar. Además, cada ayudante puede ver los compilados completamente en paralelo a Scaloni y a los respectivos ayudantes.
- Al momento en que Scaloni haya terminado de analizar el  $i$ ésimo compilado, comenzará inmediatamente algún ayudante a analizarlo, para no desperdiciar ninguno segundo.
- Sólo un ayudante verá el compilado, dado que no aporta mayor ganancia que dos ayudantes lo vean.

## 2. Análisis de diferentes soluciones propuestas

Vamos a proponer dos escenarios posibles para poder analizar sus soluciones. Estos dos escenarios los vamos a presentar mediante un gráfico cada uno, donde el tiempo de visualización de Scaloni es representado con el color turquesa, y el de los ayudantes es representado con el color azul.

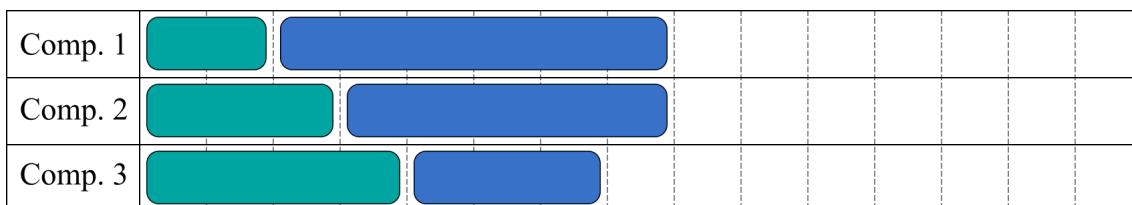
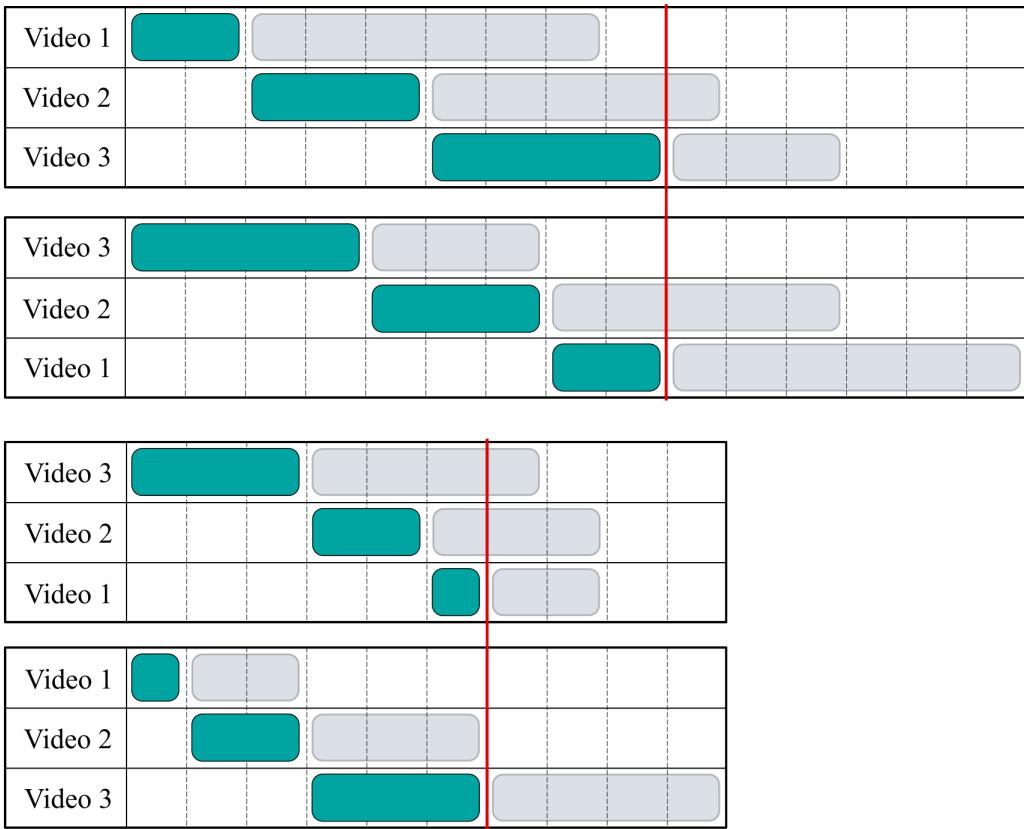


Figura 1: El tiempo que le tarda a los ayudantes ver cada compilado es inversamente proporcional a lo que le tarda a Scaloni.

Comp. 1									
Comp. 2									
Comp. 3									

Figura 2: El tiempo que le tarda a los ayudantes ver cada compilado es proporcional a lo que le tarda a Scaloni.

Lo primero que notamos fue que el orden en que Scaloni visualiza los compilados no incide en el tiempo que le toma a él en finalizar la revisión de los mismos.



Tomando este análisis en consideración, concluimos que el mecanismo de ordenamiento debería únicamente depender de  $a_i$  y no de  $s_i$ .

### 3. Posibles algoritmos

Posterior a nuestro análisis, propusimos dos algoritmos:

### 3.1. Ordenando por $a_i$ de forma creciente

Este algoritmo sigue las siguientes indicaciones:

- Ordenar los compilados de menor a mayor tiempo de análisis por parte de los ayudantes.
- Cada vez que termina Scaloni de ver un compilado, inmediatamente un ayudante es asignado a visualizarlo.

### 3.2. Ordenando por $a_i$ de forma decreciente

Este algoritmo sigue las siguientes indicaciones:

- Ordenar los compilados de mayor a menor tiempo de análisis por parte de los ayudantes.
- Cada vez que termina Scaloni de ver un compilado, inmediatamente un ayudante es asignado a visualizarlo.

Comparemos cómo se desempeñan nuestros dos algoritmos en cada escenario propuesto:

### 3.3. Resultados obtenidos del Caso 1

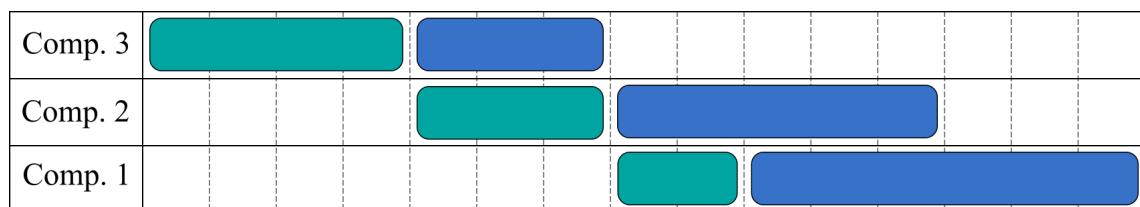


Figura 3: Caso 1 ordenado de forma creciente por  $a_i$

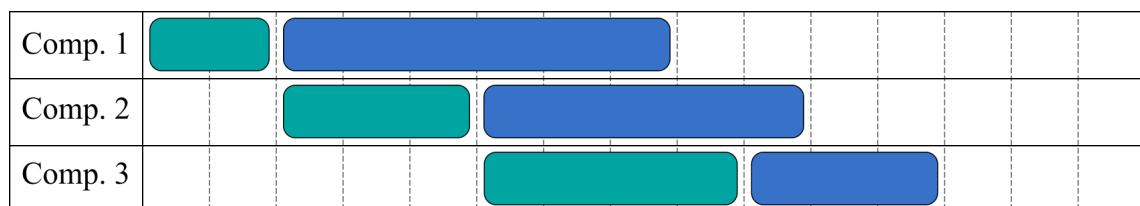


Figura 4: Caso 1 ordenado de forma decreciente por  $a_i$

### 3.4. Resultados obtenidos del Caso 2

Comp. 1							
Comp. 2							
Comp. 3							

Figura 5: Caso 2 ordenado de forma creciente por  $a_i$

Comp. 3							
Comp. 2							
Comp. 1							

Figura 6: Caso 2 ordenado de forma decreciente por  $a_i$

Como se puede observar, para ambos casos el ordenamiento decreciente resulta en un tiempo total menor.

## 4. Demostración

Procederemos a demostrar que ordenar los compilados por  $a_i$  de forma decreciente minimiza el tiempo total.

Llamamos  $C$  al conjunto ordenado de  $n$  compilados,  $s_i$  el tiempo que le tarda visualizar a Scaloni el  $i$ -ésimo compilado, y  $a_i$  lo que le tarda a un ayudante. Además, al tiempo que tarda Scaloni analizar los compilados hasta el  $k$ -ésimo lo representaremos como  $S_k$ .

$$S_k = \sum_{i=1}^k s_i$$

Luego, el tiempo total que se tarda en analizar los compilados hasta el  $k$ -ésimo, en el orden en el que se los presenta, sigue la siguiente fórmula:

$$T_k = \begin{cases} s_1 + a_1 & k = 1 \\ \max(T_{k-1}, S_k + a_k) & k > 1 \end{cases}$$

Ordenar  $C$  para minimizar  $T_n$  requiere minimizar  $\max(T_{n-1}, S_n + a_n)$ . Como  $S_n$  es independiente del orden de  $C$ ,  $S_n + a_n$  se minimiza colocando último al compilado con el tiempo del ayudante más pequeño. Luego,  $T_{n-1}$  se minimiza de la misma manera, resultando en un algoritmo recursivo. Solo nos falta justificar por qué el orden que minimiza  $S_n + a_n$  también minimiza

$\max(T_{n-1}, S_n + a_n)$ . Haremos esto partiendo de la asunción de que se respeta el orden propuesto y de que modificarlo resulta en un  $T_n$  más grande.

- Si  $\max(T_{n-1}, S_n + a_n) = S_n + a_n$  ( $\implies S_n + a_n \geq T_{n-1}$ )

Es el caso más trivial, ya que minimizar  $S_n + a_n$  también minimiza  $\max(T_{n-1}, S_n + a_n)$ .

- Si  $\max(T_{n-1}, S_n + a_n) = T_{n-1}$  ( $\implies T_{n-1} \geq S_n + a_n$ )

Este caso implica que existe un compilado  $j$  tal que  $a_j \geq \sum_{i=j+1}^n s_i + a_n \implies a_j > a_n$ . Colocar a este compilado último causaría que  $T'_n = S_n + a_j$ , y

$$T'_n = S_n + a_j, \quad T_n = S_j + a_j$$

$$\begin{aligned} S_n &> S_j \\ S_n + a_j &> S_j + a_j \\ T'_n &> T_n \end{aligned}$$

Esto demuestra que el orden de  $C$  que minimiza  $T_n$  es con los tiempos  $a_i$  decrecientes.

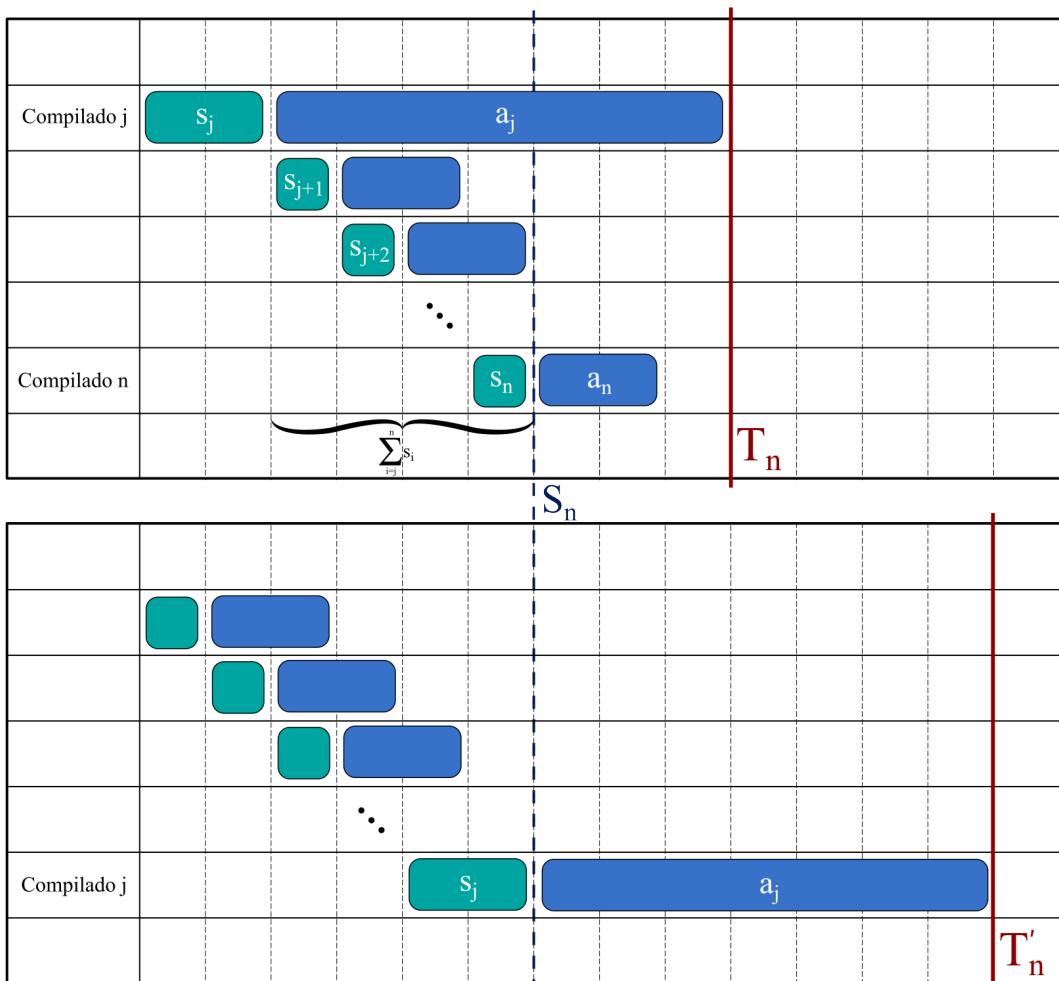


Figura 7: Ilustración de la demostración

### Para ello propusimos el siguiente algoritmo:

En primer lugar, hemos definido una clase llamada `Compilado` para modelar el compilado de cada oponente, con los atributos `tiempo_scaloni` y `tiempo_ayudante`, que almacenan el tiempo que le lleva analizarlo a Scaloni y a algún ayudante, respectivamente.

```
1 class Compilado:
2     def __init__(self, scaloni, ayudante):
3         self.tiempo_scaloni = scaloni
4         self.tiempo_ayudante = ayudante
```

De esta forma, y teniendo en cuenta los criterios previamente detallados, definimos la función `compilados_ordenados_de_forma_optima` que recibe como parámetro un arreglo con elementos de la clase `Compilado`. Esta ordena el arreglo en función del tiempo requerido por los asistentes para visualizar cada compilado, en orden descendente.

```
1 def compilados_ordenados_de_forma_optima(compilados):
2     return sorted(compilados, key=lambda compilado: compilado.tiempo_ayudante,
3                   reverse=True)
```

- Los asistentes realizan el análisis de cada uno de los compilados asignados en paralelo. Por lo tanto, el tiempo que se invierte en la revisión de un compilado específico de máxima duración, puede ser aprovechado de manera tal que este sea visto mientras Scaloni se dedica a la revisión de otros compilados. De esta forma, nos aseguramos que se minimice el tiempo que suman los ayudantes en la revisión total.
- También en esta solución tiene en consideración el análisis 2 descripto previamente, ya que en el mejor de los casos, la  $\sum_{k=1}^n s_k + a_n$  termina siendo la mínima ya que, por la forma del ordenamiento del algoritmo,  $a_n$  es la duración del compilado más corto.

### 4.1. Cota de complejidad temporal

El algoritmo presentado tiene una complejidad temporal de  $O(n \log n)$ . Esto se debe a que seguimos los siguientes pasos: Ordenamos los compilados según el tiempo de análisis de los ayudantes.

Para ello utilizamos `sorted` de la librería de python, que usa por detrás el algoritmo de [Timsort](#), teniendo este una complejidad de  $O(n \log n)$ .

## 5. Mediciones

Para las mediciones, creamos listas de compilados de ejemplo de forma aleatoria. Decidimos que el tiempo de análisis de un compilado fuera un valor aleatorio entre 1 y 99'999, basándonos en los datos de ejemplo provistos por la cátedra.

Medimos el tiempo de ejecución de nuestro algoritmo para ciertas cantidades de compilados.

Notamos que los procesos de fondo del sistema nos generaba distorsiones en los tiempos de ejecución de una misma muestra. Atacamos este problema realizando varias mediciones para el mismo escenario de compilados y tomando el promedio de los tiempos obtenidos.

Además, como definimos cada escenario arbitrariamente, en una cantidad de compilados  $x$ , este escenario puede ser poco favorable (puede estar muy desordenado), mientras que en una cantidad de compilados  $x + 1$  puede ser muy favorable. Por lo cual la diferencia de tiempos va a ser muy grande a pesar de que la cantidad de compilados solamente difiere en una unidad (TimSort es una combinación de inserción y mergeSort, por lo cual el desorden inicial toma gran relevancia en el rendimiento final del algoritmo). Para solucionar esto, decidimos hacer varios escenarios distintos para la misma cantidad de compilados.

Con el objetivo de comparar los tiempos de ejecución de nuestro algoritmo con la complejidad teórica, optamos por realizar tanto un análisis de regresión lineal como uno de regresión lineal logarítmica que se ajustara a nuestros datos. Para evaluar qué curva se ajusta mejor, usamos la

raíz del error cuadrático medio (RMSE). Realizamos este análisis en un intervalo con tamaños pequeños (hasta 1'000) y en un intervalo con tamaños grandes (hasta 10'000). Ver figuras 8 y 9

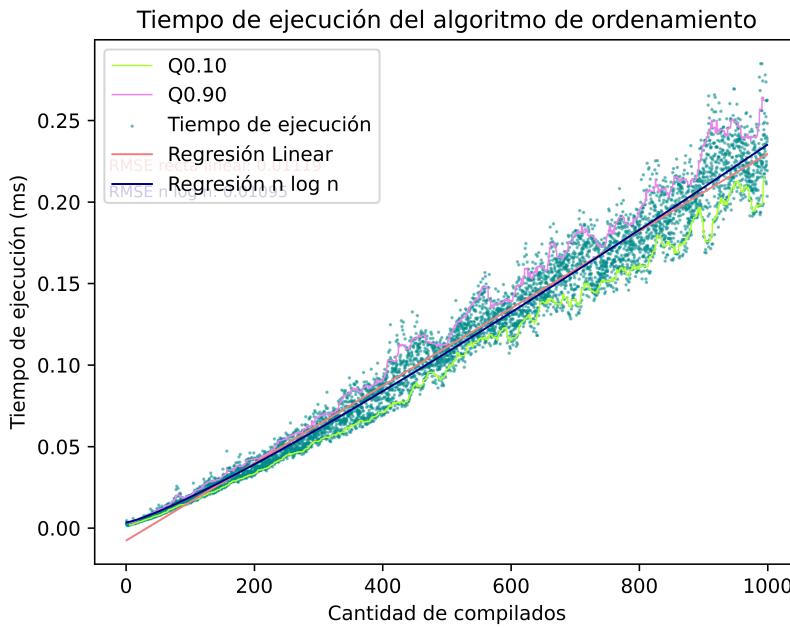


Figura 8: Tendencia de la complejidad algorítmica para  $n$  pequeños.

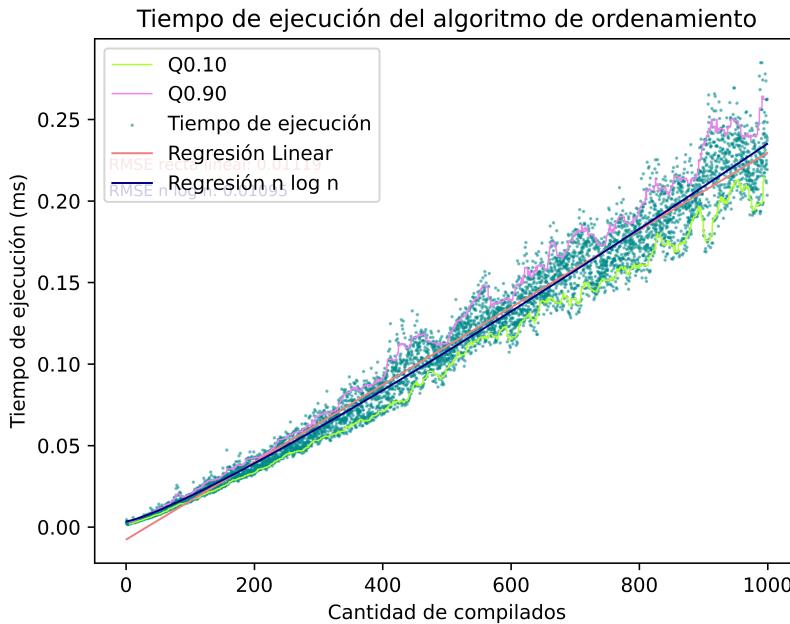


Figura 9: Tendencia de la complejidad algorítmica para  $n$  grandes.

Se puede observar que en valores pequeños, el tiempo de ejecución del algoritmo sigue una clara curva lineal logarítmica. Sin embargo, a medida que aumenta el tamaño de la muestra de compilados, su comportamiento se approxima más a una complejidad lineal.

Esto ocurre debido a la naturaleza de la función lineal logarítmica, que presenta una pendiente logarítmica. Cuando se trabajan con valores grandes, esta pendiente se vuelve casi constante, lo que la asemeja a una función lineal. Esto se debe a que el logaritmo es una función monótona creciente, lo que significa que su derivada es siempre positiva para valores mayores que cero. Sin embargo, el crecimiento de esta función se vuelve cada vez más lento a medida que los valores aumentan, indicando que su segunda derivada es siempre negativa. En consecuencia, para valores muy grandes, la función lineal logarítmica se estabiliza y, aunque no es completamente constante, su comportamiento se asemeja a una función lineal.

De esta manera, mediante el gráfico y la validación a través del error cuadrático medio, hemos comprobado que la curva lineal logarítmica se adapta de manera más precisa a nuestros datos.

De esta forma pudimos comprobar empíricamente que la complejidad tiende a  $O(n \log n)$ .

Nótese que graficamos también dos curvas que demarcan una estimación de los cuantiles verticales 0,1 y 0,9. Esta estimación se realizó calculando los cuantiles para un grupo pequeño centrado en cada punto. Estas curvas nos ayudan a dimensionar como la varianza del tiempo de ordenamiento crece con el aumento del tamaño de la información de entrada.

Adicionalmente, graficamos la densidad de los tiempos que también brinda una visualización de como aumenta la varianza con el aumento del tamaño de los datos de entrada. Ver figuras 10 y 11

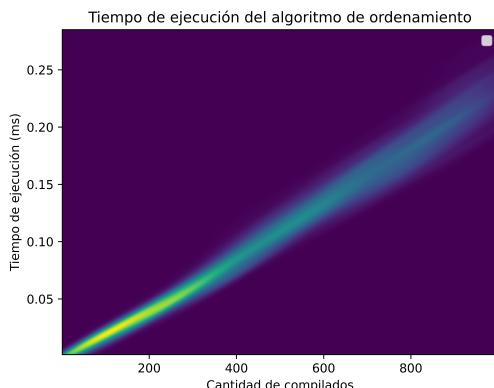


Figura 10: Mediciones para  $n$  bajos.

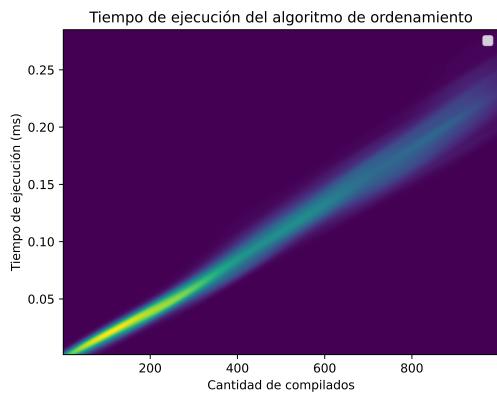


Figura 11: Mediciones para  $n$  altos.

## 6. Conclusiones

Finalmente, consideramos que la solución óptima para abordar la problemática de Scaloni sería que éste analizara los compilados en función del tiempo requerido por los asistentes para visualizar cada uno, organizándolos en orden descendente. Esta estrategia permitiría resolver el problema con una complejidad algorítmica de orden  $O(n \log n)$ . Este enfoque garantiza la máxima eficiencia en la visualización de los compilados y permite que el tiempo invertido por Scaloni y sus asistentes se administre de manera óptima.