.UBAfiuba
FACULTAD DE INGENIERÍA

Teoría de Algoritmos
(75.29 - 95.06) Curso Buchwald - Genender

Trabajo Práctico 1
Algoritmos Greedy

17 de septiembre de 2023

Alexis Herrera
104639

Edgardo Francisco Saez
104896

Franco Gazzola
105103

1. Introducción

El objetivo principal de este trabajo es ayudar a Scaloni a analizar los próximos n rivales de la selección campeona del mundo.

Para lograr esto, se debe diseñar un algoritmo Greedy que determine el orden óptimo en el que Scaloni debería ver los videos de los rivales, teniendo en cuenta los tiempos s_i y a_i que Scaloni y sus ayudantes tardarán en analizar cada video.

1.1. Análisis del problema

Dado que hay n rivales y n ayudantes, y considerando que cada ayudante puede analizar un video completamente en paralelo después de que Scaloni lo haya hecho, el tiempo total de análisis (t_0) está determinado por el tiempo que Scaloni se tarda en revisar todos los videos (t_1) más el tiempo que deba emplearse para terminar todas aquellas revisiones que queden pendientes por los ayudantes (t_2).

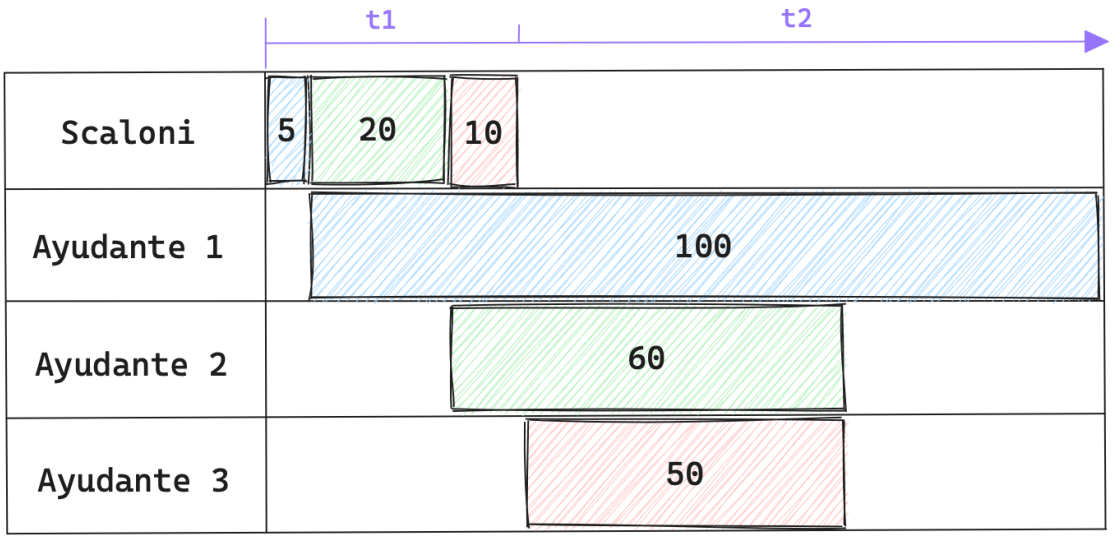


Figura 1: $t_0 = t_1 + t_2$

Podemos plantear lo siguiente en base a las características del problema:

- Siempre habrá un ayudante disponible para revisar un video después que Scaloni lo haya revisado.

  Esto es verdadero porque a lo sumo puede haber n videos por revisar de manera simultánea y como hay n ayudantes bastaría con que cada uno revise un video.

- El tiempo total que Scaloni tardará en revisar todos los videos (t_1) es una constante igual a la sumatoria de los tiempos s_i, sin importar el orden elegido.

  Si no fuera así, implicaría que Scaloni dejase tiempos entre videos que revisa, haciendo que los ayudantes deban esperar más para poder revisar, lo cual en conjunto con el item anterior, haría que el tiempo total sea mayor.

- La optimización del tiempo total de revisión (t_0) se puede lograr solo optimizando el tiempo en el que tardan los ayudantes en terminar de revisar los videos pendientes (t_2).

  Dado que siempre Scaloni va a terminar de revisar en un tiempo t_1 (sin importar el orden), entonces el óptimo del problema se puede lograr haciendo que t_2 sea mínimo.

- El tiempo óptimo de t_2 se puede lograr revisando primero aquellos videos que los ayudantes le tomen más tiempo

  Como el problema quedó reducido a optimizar t_2, entonces podemos lograr el objetivo haciendo que la mayor cantidad de tiempo de las tareas realizadas por los ayudantes transcurra mientras Scaloni aún está revisando las suyas.

  Es decir, que realizando primero las revisiones más largas, se consume la mayor cantidad de tiempo posible antes del momento t_1, haciendo que para el momento t_1 las tareas más grandes estén acortadas lo máximo posible, si es que no finalizaron aún.

  En el caso de que hayan finalizado también se puede ver que la última revisión es la más corta y por lo tanto también t_2 es mínima.

En conclusión, el algoritmo greedy para este problema se podría enunciar de la siguiente manera:

1. Ordenar las revisiones de los rivales según el tiempo que tardan los ayudantes en revisar a cada rival, de mayor a menor.
2. El orden que minimiza el tiempo se logra realizando las revisiones en el orden mencionado anteriormente.

Además este algoritmo es óptimo, como mencionamos anteriormente a través de deducciones lógicas, quedó reducido a solo optimizar t_2. Entonces acortando las tareas pendientes mediante el aprovechamiento del tiempo que Scaloni tarda en resolver las suyas, es la manera óptima de reducir t_2 y por lo tanto el tiempo total t_0.

2. Algoritmo y Complejidad

En esta sección se expondrá el código del algoritmo diseñado analizando y justificando su complejidad. Además se hará un análisis de cómo afecta la variabilidad de los valores a_i y s_i a los tiempos y optimalidad del algoritmo planteado.

2.1. Codigo

A continuación se muestra el código de solución del problema.

```
def obtener_orden_minimo(s, a):
    """
    :param s: Lista de tiempos de lo que tarda Scaloni en revisar cada rival
    :param a: Lista de tiempos de lo que tarda cada ayudante en revisar cada rival
    :return: lista con el orden tal que el tiempo total sea el minimo
    """
    return sorted(range(len(a)), key=lambda k: a[k], reverse=True) # O(n log n)

def obtener_tiempo_minimo(s, a):
    """
    :param s: lista de lo que tarda Scaloni en revisar cada rival
    :param a: lista de lo que tarda cada ayudante en revisar cada rival
    :return: El tiempo minimo que se tarda en revisar a todos los rivales
    """
    orden_minimo = obtener_orden_minimo(s, a)
    tiempo_total = 0
    tiempo_total_scaloni = 0
    for i in orden_minimo: # O(n)
        tiempo_total_scaloni += s[i]
        tiempo_total = max(tiempo_total, tiempo_total_scaloni + a[i])
    return tiempo_total
```

2.2. Complejidad

La complejidad temporal del algoritmo propuesto es $\mathcal{O}(n \cdot log(n))$, debido a que lo primero que hacemos es ordenar n elementos ejecutando la función "obtener orden minimo" lo que nos cuesta $\mathcal{O}(n \cdot log(n))$ por ser esta la mejor complejidad que se puede obtener para un algoritmo de ordenamiento. Luego iteramos por los elementos ya ordenados lo que conlleva una complejidad de $\mathcal{O}(n)$ y finalmente dentro de esta iteración hacemos sumas y comparaciones, que terminan dando una complejidad de $\mathcal{O}(1)$.

La complejidad total del algoritmo se calcula como:

$$\mathcal{O}(f(n)) = \mathcal{O}(n \cdot \log(n)) + \mathcal{O}(n) + \mathcal{O}(1) = \mathcal{O}(n \cdot \log(n)).$$

Por otro lado, la complejidad espacial es $\mathcal{O}(n)$ debido a que utilizamos el algoritmo de ordenamiento que ofrece Python, llamado Timsort, el cual tiene una complejidad espacial de $\mathcal{O}(n)$.

2.3. Variabilidad de los valores

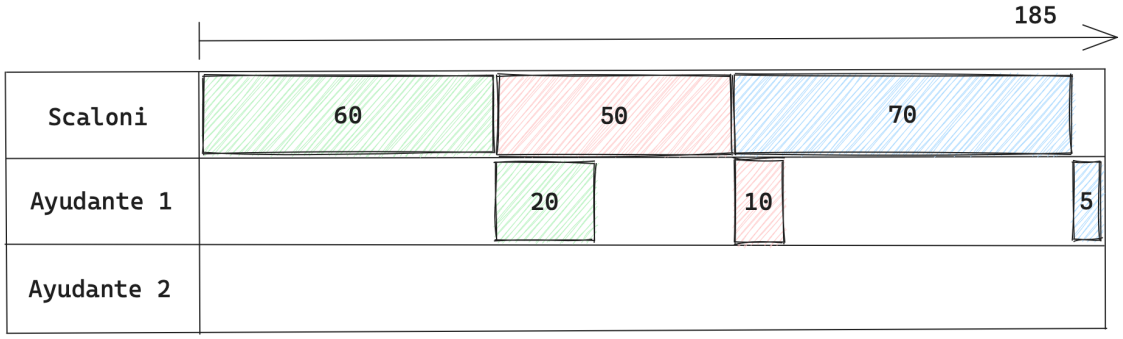
A continuación, se hará un análisis de cómo afecta la variabilidad de los valores a_i s_i a los tiempos y optimalidad del algoritmo. Presentaremos un ejemplo en donde tendremos en cuenta dos casos. En el primero las tareas que realiza Scaloni toman mucho más tiempo en hacerse que las de los ayudantes mientras que en el segundo es lo contrario.

Definimos una tupla como t = (si, ai)
Donde:

si = el tiempo que tarda Scaloni.
ai = el tiempo que tardan sus ayudantes.

En primer lugar, analizamos qué sucede cuando las tareas de Scaloni tardan mucho más tiempo en hacerse que las de los ayudantes.

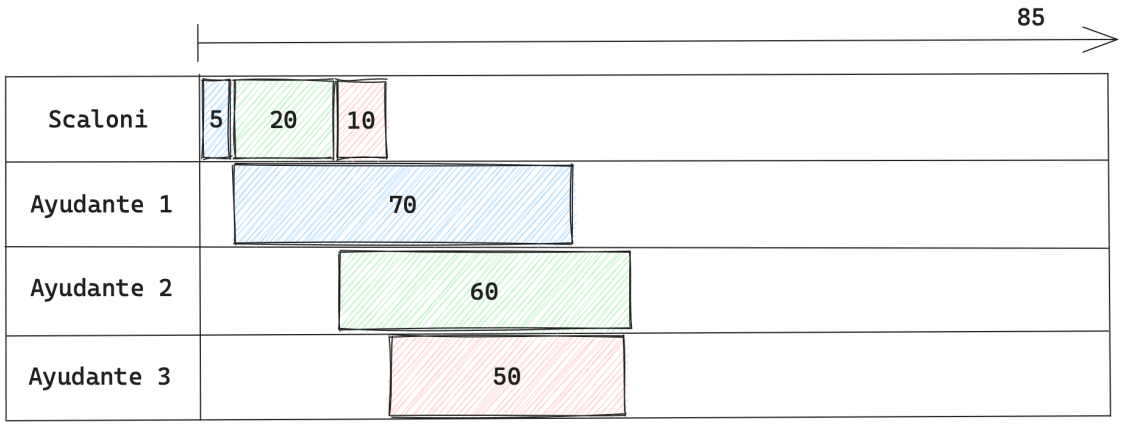
Figura 2: $a_i \ll s_i$, $S = \{(60, 20), (50, 10), (70, 5)\}$



Como se ilustra en la Figura 2, cuando las tareas de los ayudantes son muy cortas en relación a las de Scaloni, lo que termina sucediendo es que un único ayudante (de los N disponibles) termina revisando las tareas a medida que Scaloni las va realizando. En este caso se estarían desperdiciando N-1 ayudantes o también podemos decir que el problema podría resolverse con un único ayudante. El tiempo total que se tardó en resolver el problema es 185.

Ahora analizamos que sucede cuando los tiempos que tardan los ayudantes son mucho mayores a los de Scaloni.

Figura 3: $a_i \gg s_i$, $S = \{(20, 60), (10, 50), (5, 70)\}$



Como se ilustra en la Figura 3, cuando las tareas de los ayudantes son más grandes con respecto a las de Scaloni se tiende a hacer mayor uso de los ayudantes. Esto se debe a que al ser las tareas de Scaloni más cortas, el mismo termina antes de hacerlas y los ayudantes pueden comenzar a trabajar con ellas más pronto. El tiempo total que se tardó en resolver el problema es de 85.

Ahora viendo ambos casos en conjunto obtenemos las siguientes conclusiones:

- Cuando $a_i \gg s_i$ se usan mas ayudantes que cuando $s_i \gg a_i$.
- Cuando $a_i \gg s_i$ el tiempo en resolver el problema (185) es mucho menor que cuando $s_i \gg a_i$ (85).
- Teniendo en cuenta los dos items anteriores podemos concluir que el tiempo óptimo será menor cuando los $a_i \gg s_i$ debido a que se hará más uso del los ayudantes disponibles, permitiendo que se hagan más tareas en paralelo.

3. Casos de prueba

A continuación se muestran los ejemplos mencionados anteriormente, verificando que el programa devuelve lo esperado. También se muestran los ejemplos dados por la cátedra.

En esta ocasión utilizamos la librería PyTest para ejecutar los casos de prueba, por lo que para ejecutarlos ingresamos el siguiente comando: `pytest -v`.

```python
def test_caso_1_ejemplo_tp():
    s = [60, 50, 70]
    a = [20, 10, 5]
    assert obtener_tiempo_minimo(s, a) == 185

def test_caso_2_ejemplo_tp():
    s = [20, 10, 5]
    a = [60, 50, 70]
    assert obtener_tiempo_minimo(s, a) == 85

def test_caso_1_ejemplo_catedra_3_elem():
    s, a = obtener_tiempos_de_archivo('./textos_pruebas/3_elem.txt')
    assert obtener_tiempo_minimo(s, a) == 10

def test_caso_2_ejemplo_catedra_10_elem():
    s, a = obtener_tiempos_de_archivo('./textos_pruebas/10_elem.txt')
    assert obtener_tiempo_minimo(s, a) == 29

def test_caso_3_ejemplo_catedra_100_elem():
    s, a = obtener_tiempos_de_archivo('./textos_pruebas/100_elem.txt')
    assert obtener_tiempo_minimo(s, a) == 5223

def test_caso_4_ejemplo_catedra_10000_elem():
    s, a = obtener_tiempos_de_archivo('./textos_pruebas/10000_elem.txt')
    assert obtener_tiempo_minimo(s, a) == 497886735
```

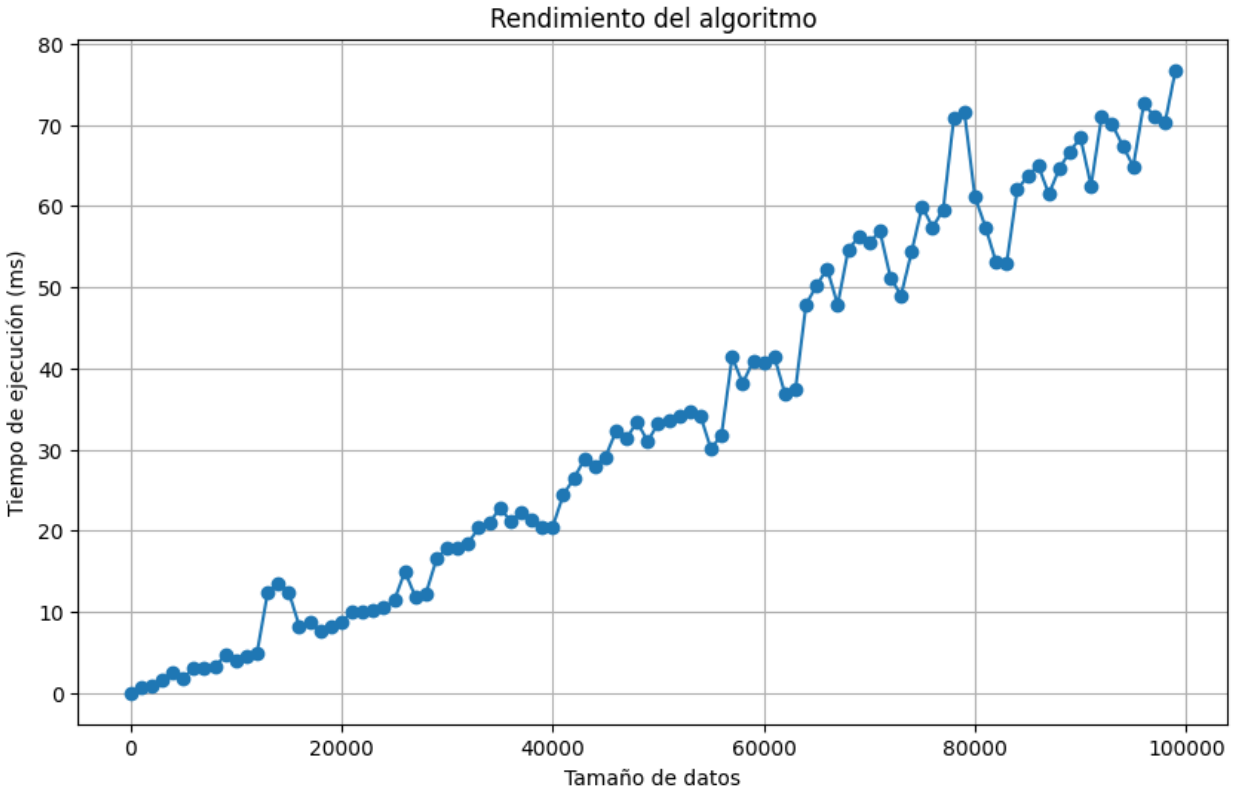
```
alexis@alexis-pc ~/Documents/fiuba/tda/TDA-TP1-Greedy main ± pytest -v
============================= test session starts ==============================
platform linux -- Python 3.10.12, pytest-7.4.1, pluggy-1.3.0 -- /usr/bin/python3
cachedir: .pytest_cache
rootdir: /home/alexis/Documents/fiuba/tda/TDA-TP1-Greedy
collected 8 items

test_ejemplos.py::test_caso_con_solo_1_partido PASSED                    [ 12%]
test_ejemplos.py::test_caso_con_2_partidos PASSED                        [ 25%]
test_ejemplos.py::test_ejemplo_1_tp PASSED                               [ 37%]
test_ejemplos.py::test_ejemplo_2_tp PASSED                               [ 50%]
test_ejemplos.py::test_ejemplo_catedra_3_elem PASSED                     [ 62%]
test_ejemplos.py::test_ejemplo_catedra_10_elem PASSED                    [ 75%]
test_ejemplos.py::test_ejemplo_catedra_100_elem PASSED                   [ 87%]
test_ejemplos.py::test_ejemplo_catedra_10000_elem PASSED                 [100%]

============================== 8 passed in 0.01s ===============================
```

Figura 4: Casos de Prueba

4. Mediciones

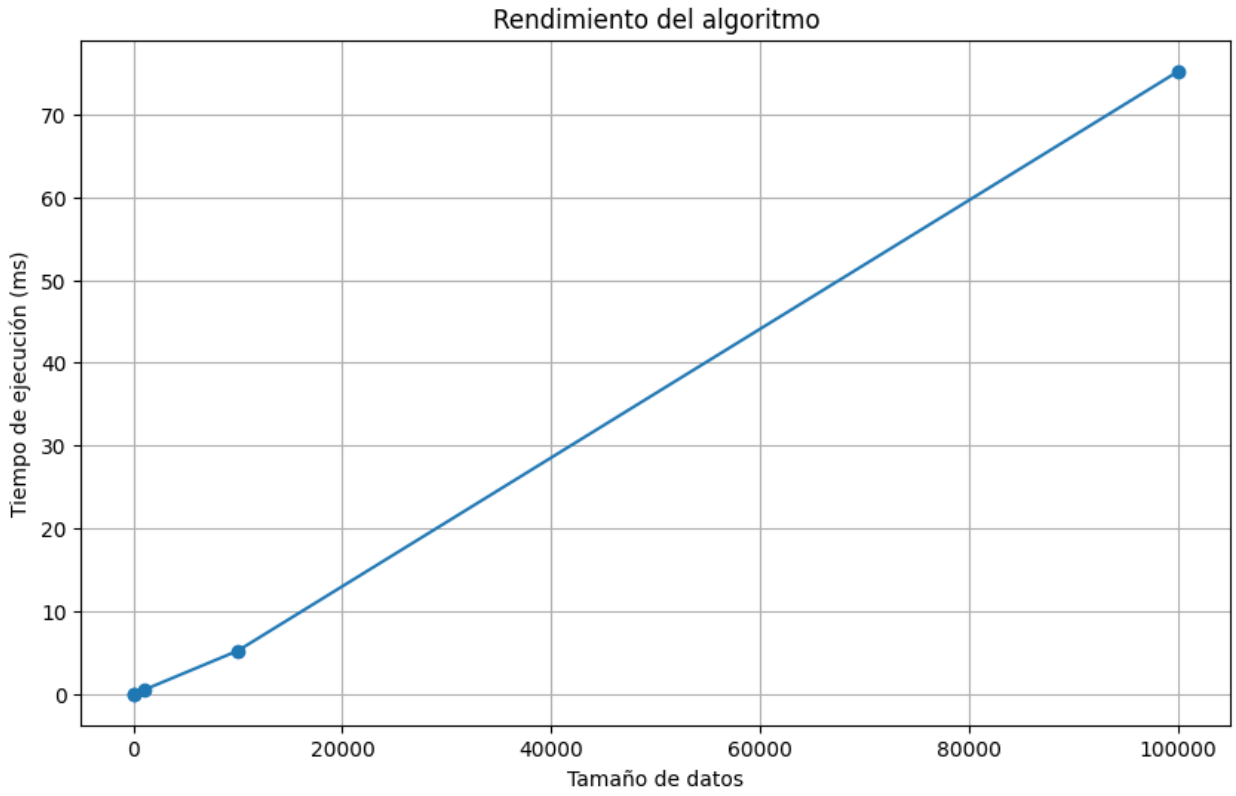
Llevamos a cabo mediciones de tiempo utilizando arreglos de diferentes longitudes. En cada uno de estos casos, repetimos el proceso en múltiples ocasiones para obtener valores de tiempo de ejecución consistentes y estables.

4.1. Medición 1



En el primer escenario, hemos creado diversas series de datos con incrementos de 1000 elementos, abarcando un rango desde 0 hasta 100,000 elementos. Al observar el gráfico resultante, notamos una densidad significativa de puntos. Inicialmente, al comienzo de la serie, se evidencia una tendencia lineal en el rendimiento del algoritmo. Sin embargo, a medida que el número de elementos aumenta, se aprecia una tendencia de crecimiento que se ajusta a una complejidad $\mathcal{O}(n \cdot log(n))$. Este patrón indica un comportamiento más complejo a medida que los datos se vuelven más voluminosos.

4.2. Medición 2

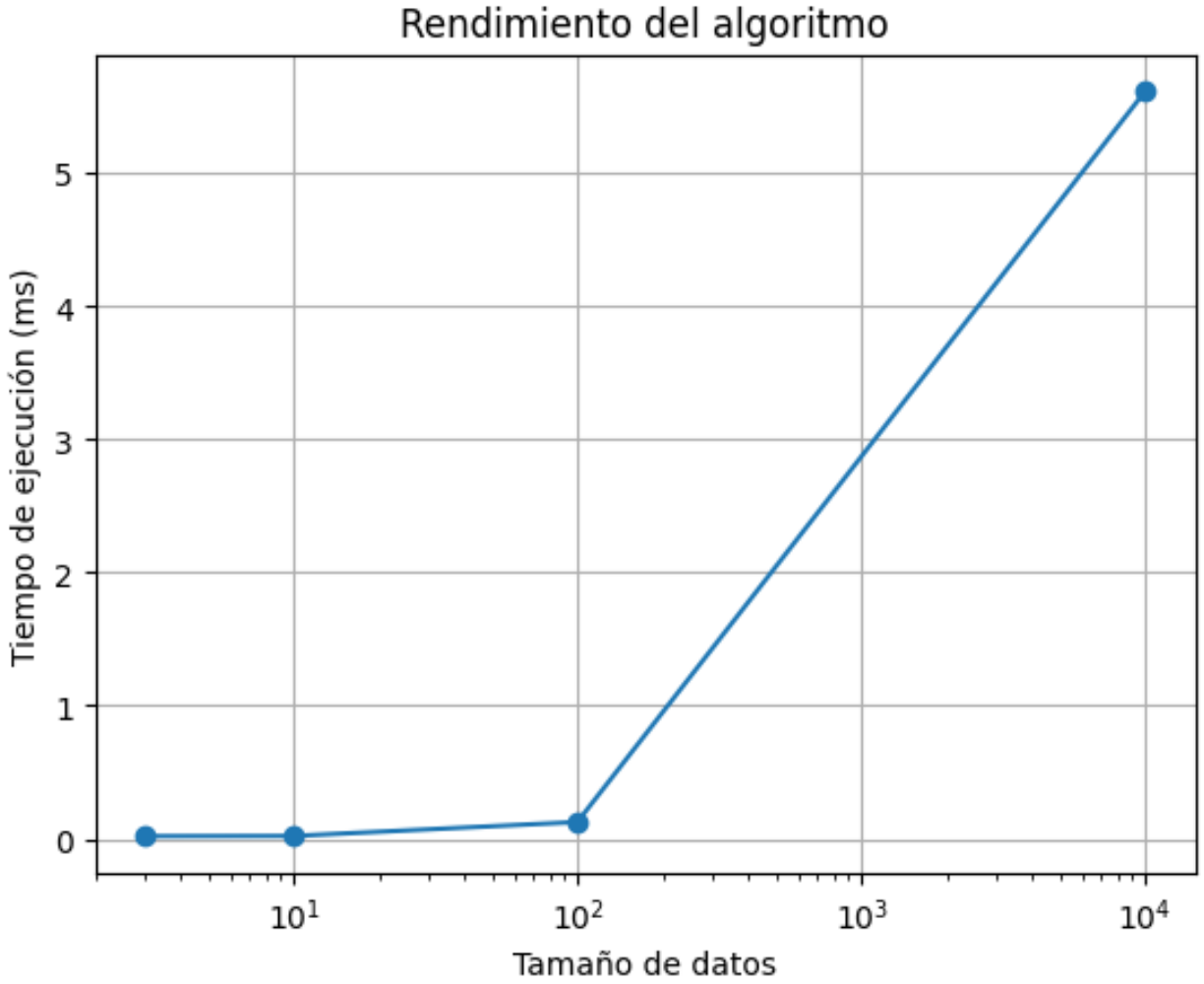


En este escenario, hemos generado series de datos de longitudes diversas, aumentando de manera exponencial desde 10 hasta 100,000 elementos.

En el gráfico, se aprecia un cambio positivo en la pendiente entre el punto donde se calculó el tiempo con 10,000 elementos y el punto con 100,000 elementos. Este fenómeno es coherente con lo planteado en el gráfico previo, donde se observa un crecimiento con complejidad $\mathcal{O}(n \cdot log(n))$.

En ambos ejemplos los elementos fueron generados por los valores pseudoaleatorios del lenguaje (el módulo `random`).

4.3. Medición 3



Para concluir, llevamos a cabo las mediciones utilizando las muestras proporcionadas por la cátedra, las cuales contenían 3, 10, 100 y 10,000 elementos para su análisis.

En el eje x, hemos implementado una escala logarítmica con el propósito de mejorar la visualización de los datos. Esto implica que los valores en el eje x no se encuentran equidistantes, sino que aumentan en potencias de 10.

Gracias a esta modificación, en este ejemplo, la evolución del gráfico se aprecia con mayor claridad.

4.4. Análisis de Complejidad

Para concluir, a partir de los casos planteados, podemos afirmar de manera concluyente que el algoritmo propuesto presenta una complejidad evidente de $\mathcal{O}(n \cdot log(n))$.

5. Conclusiones

Luego de un análisis exhaustivo del problema, llegamos a la conclusión de que la estrategia óptima para abordarlo radica en la optimización del tiempo empleado por los ayudantes al revisar

los videos. Esto se traduce en la priorización de la revisión de los videos más extensos por parte de los ayudantes, lo cual se ha identificado como la solución más eficiente.

En cuanto al análisis de la variabilidad se puede concluir, como se vio en la sección 2.3, que el problema se comporta mejor cuando Scaloni completa las tareas con mayor velocidad en relación a los ayudantes. Esto es siempre teniendo en cuenta que la solución encontrada usando el algoritmo es la óptima.

Además, al evaluar detenidamente el algoritmo propuesto, hemos determinado que su complejidad computacional es de $\mathcal{O}(n \cdot log(n))$.

6. Bibliografia

Algoritmo de ordenamiento TimSort:

https://en.wikipedia.org/wiki/Timsort