.UBAfiuba
FACULTAD DE INGENIERÍA

Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

Trabajo Práctico 1
Informe

18 de septiembre de 2023

Brzoza Valeria	Cichero Tomás	Covela Maximiliano
107523	107973	102547

Índice

1. Consigna **3**
1.1. Datos . . . 3

2. Análisis inicial **3**

3. Algoritmos **3**
3.1. Algoritmo ordenado por tiempos de ayudantes, de menor a mayor . . . 3
3.1.1. Conclusiones del algoritmo . . . 4
3.2. Algoritmo ordenando por tiempos de ayudantes, de mayor a menor . . . 4
3.2.1. Casos en donde no se da el mejor caso . . . 4

4. Complejidades de funciones implementadas **5**
4.1. Función ordenar rivales . . . 5
4.2. Función calcular tiempo . . . 5
4.3. Complejidad total . . . 6

5. Gráficos **6**
5.1. Cantidad vs Tiempo . . . 6

6. Referencias **8**

1. Consigna

Para este Trabajo práctico se nos pidió ayudar a Scaloni a que él y sus ayudantes pudieran realizar análisis a sus rivales en el menor tiempo posible, partiendo de unas ciertas normas. Para realizar el algoritmo pedido, nos van a dar la cantidad de rivales (n), y los respectivos tiempos que Scaloni y algún ayudante tardarían en cada caso (S_i y A_i respectivamente).

1.1. Datos

Para realizar esto, contamos con la siguiente información:

1. Scaloni tiene que haber terminado de ver y analizar los videos antes de poder pasarselos a su ayudante para que los revise.
2. Scaloni cuenta con la misma cantidad de ayudantes que de equipos rivales pendientes de analizar.
3. Que más de un ayudante analice al rival, no genera ganancia.
4. Cada ayudante, y Scaloni, pueden ver videos diferentes en paralelo.

2. Análisis inicial

Con esta información pudimos deducir que lo ideal es que cada ayudante vea unicamente un solo rival y de esta forma cuando Scaloni termina con uno se lo pasa inmediatamente para revisar a alguno de los que estén libres. Por otro lado, sabemos que mas allá del algoritmo que usemos, siempre contamos con la base de Scaloni va a tardar siempre

$$\sum_{i=1}^{n} s_i$$

3. Algoritmos

3.1. Algoritmo ordenado por tiempos de ayudantes, de menor a mayor

Tras leer la consigna, y realizar nuestro análisis inicial, pensamos en un algoritmo que ordenara los tiempos de análisis de los ayudantes de menor a mayor, siempre dando el rival analizado por Scaloni a un ayudante diferente. A pesar de que este algoritmo haría que los primeros ayudantes terminaran rápido, sabemos que esto no nos beneficia ya que contamos con n ayudantes. Podemos demostrar que esta solución no es óptima si consideramos el siguiente ejemplo:

País	S_i	A_i
México	3	1
Colombia	2	2
Croacia	1	4

El algoritmo recién planteado devolverá que el orden es: México, Colombia, Croacia.
Sin embargo, haciendo un seguimiento, notamos que Scaloni y los primeros dos ayudantes habrían terminado, y sin embargo habría una espera de 3 horas más hasta que el ayudante que se le asignó Croacia termine.
A continuación mostramos un cuadro en donde se pueden ver los horarios de los análisis por cada persona, siendo **S** Scaloni y A_i sus respectivos ayudantes.

	09:00	10:00	11:00	12:00	13:00	14:00	15:00	16:00	17:00	18:00
S	México	México	México	Colombia	Colombia	Croacia	-	-	-	-
A_1	-	-	-	México	-	-	-	-	-	-
A_2	-	-	-	-	-	Colombia	Colombia	-	-	-
A_3	-	-	-	-	-	-	Croacia	Croacia	Croacia	Croacia

3.1.1. Conclusiones del algoritmo

Analizando nuestro algoritmo anterior nos dimos cuenta de las siguientes cosas:

1. En el mejor de los casos:
$$TiempoFinal = \sum_{i=1}^{n} s_i + a_n$$
2. Este tiempo puede ser aún mayor si al terminar el ayudante a_n su último análisis, hay ayudantes que todavía no terminaron los suyos.
3. Nuestro objetivo es minimizar la diferencia entre Tiempo Final y $\sum_{i=1}^{n} s_i$

Con esta nueva visión en mente, pensamos en el algoritmo detallado a continuación.

3.2. Algoritmo ordenando por tiempos de ayudantes, de mayor a menor

Con las conclusiones sacadas recientemente, pensamos en ordenar los rivales según los tiempos de los ayudantes, de mayor a menor, siempre siendo cada rival asignado a un ayudante diferente. Podemos ver que esto soluciona el ejemplo dado anteriormente. Si bien el primer rival en analizarse sería Croacia, las 4 horas que usaría ese ayudante, serían horas que igualmente usaría Scaloni para analizar a Colombia y México. Para la sexta hora:

- El ayudante terminó el análisis de Croacia
- Scaloni terminó de analizar los 3 rivales
- El ayudante terminó el análisis de Colombia

Por ende, solo faltaría esperar una hora a que el ayudante termine el análisis de México. En este caso conseguimos llegar al mejo caso de tiempo final y reducir la diferencia hasta 1, ya que $TiempoFinal = 7$ y $\sum_{i=1}^{n} s_i = 6$.

3.2.1. Casos en donde no se da el mejor caso

Supongamos el siguiente caso:

País	S_i	A_i
Francia	2	2
Brasil	2	8
Marruecos	1	1

Nuestro algoritmo va a devolver: Brasil, Segundo Francia, Marruecos. En este caso, $\sum_{i=1}^{n} s_i = 5$ y $TiempoFinal = 10$. Esto se debe a que recién en la tercera hora, un ayudante arranca a analizar a Brasil, el cual le tomará 8 horas. Sin embargo, esta sigue siendo la solución óptima, ya que es el primer rival en analizarse. Mientras más tarde Scaloni analice a Brasil, mayor va a ser la diferencia entre $TiempoFinal$ y $\sum_{i=1}^{n} s_i$.
A continuación mostramos un cuadro en donde se pueden ver los horarios de los análisis por cada persona, siendo **S** Scaloni y A_i sus respectivos ayudantes.

	13:00	14:00	15:00	16:00	17:00	18:00	19:00	20:00	21:00	22:00
S	Brasil	Brasil	Francia	Francia	Marruecos	-	-	-	-	-
A_1	-	-	Brasil	Brasil	Brasil	Brasil	Brasil	Brasil	Brasil	Brasil
A_2	-	-	-	-	Francia	Francia	-	-	-	-
A_3	-	-	-	-	-	Marruecos	-	-	-	-

4. Complejidades de funciones implementadas

4.1. Función ordenar rivales

Para realizar el ordenamiento utilizamos el método sort integrado en el propio lenguaje de programación (Dart). Esta es una implementación de Dual-Pivot Quicksort, basado en el paper de Vladimir Yaroslavskiy.

La complejidad de este algoritmo es $O(nlog(n))$

A continuación se muestra la implementación en código.

```
typedef Rival = ({int scaloni, int ayudante});

List<Rival> ordenarRivales(List<Rival> rivales) {
  rivales.sort((a, b) {
    return b.ayudante.compareTo(a.ayudante);
  });

  return rivales;
}
```

4.2. Función calcular tiempo

Este algoritmo son simplemente iteraciones sobre la lista previamente ordenada en las cuales se suma el tiempo que le toma a cada ayudante al tiempo que tarda Scaloni, nos vamos quedando con las horas que nos lleva con el que termina más tarde y se calcula el tiempo total hasta terminar todos los análisis.

La complejidad es $O(n)$

A continuación se muestra la implementación en código.

```
int calcularTiempoTotal(List<Rival> rivales) {
  int tiempoScaloni = 0;
  int tiempoMaxAyudante = 0;

  for (final rival in rivales) {
    tiempoScaloni += rival.scaloni;

    final nuevoTiempoMaxAyudate = tiempoScaloni + rival.ayudante;

    if (nuevoTiempoMaxAyudate > tiempoMaxAyudante) {
      tiempoMaxAyudante = nuevoTiempoMaxAyudate;
    }
  }

  return tiempoMaxAyudante;
}
```

4.3. Complejidad total

La función completa, que ordena y calcula el tiempo, junta las dos funciones analizadas anteriormente. Por ende, la complejidad total del algoritmo es $O(nlog(n)) + O(n)$. Esto se simplifica y queda $O(nlog(n))$ como nuestra complejidad teórica.

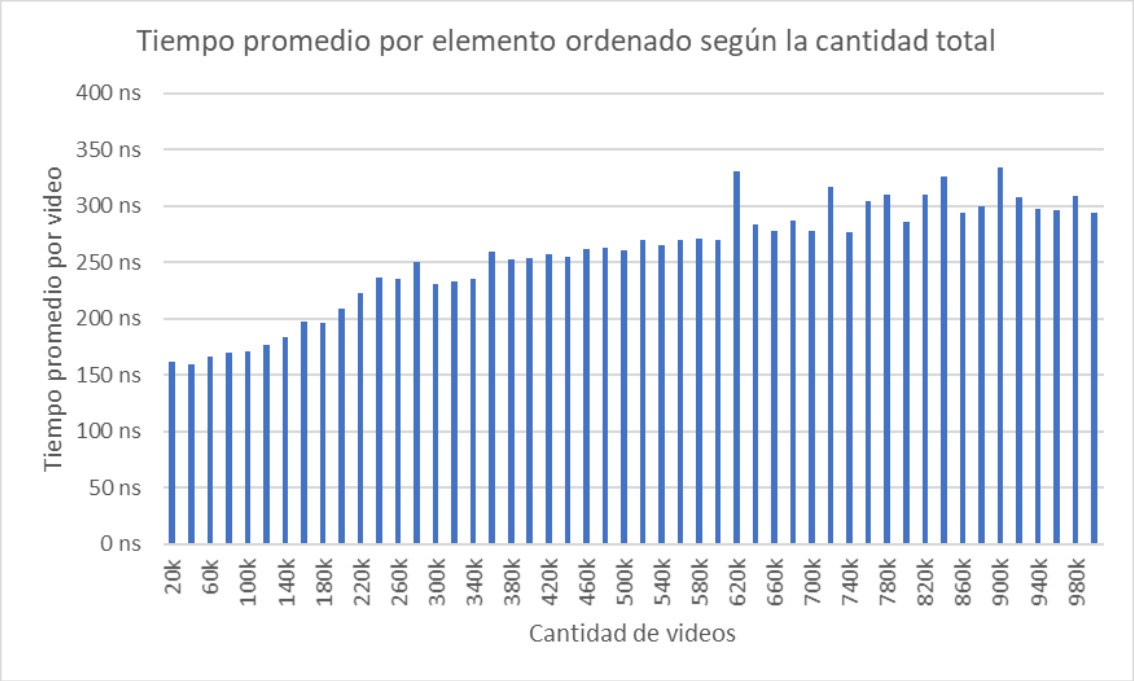
5. Gráficos

5.1. Cantidad vs Tiempo

En el siguiente gráfico podemos observar el tiempo que demora el programa en ordenar todos los análisis en relación a la cantidad de análisis totales. El análisis para el gráfico se hizo haciendo 20 pruebas por cada punto y tomando un promedio de cada una.

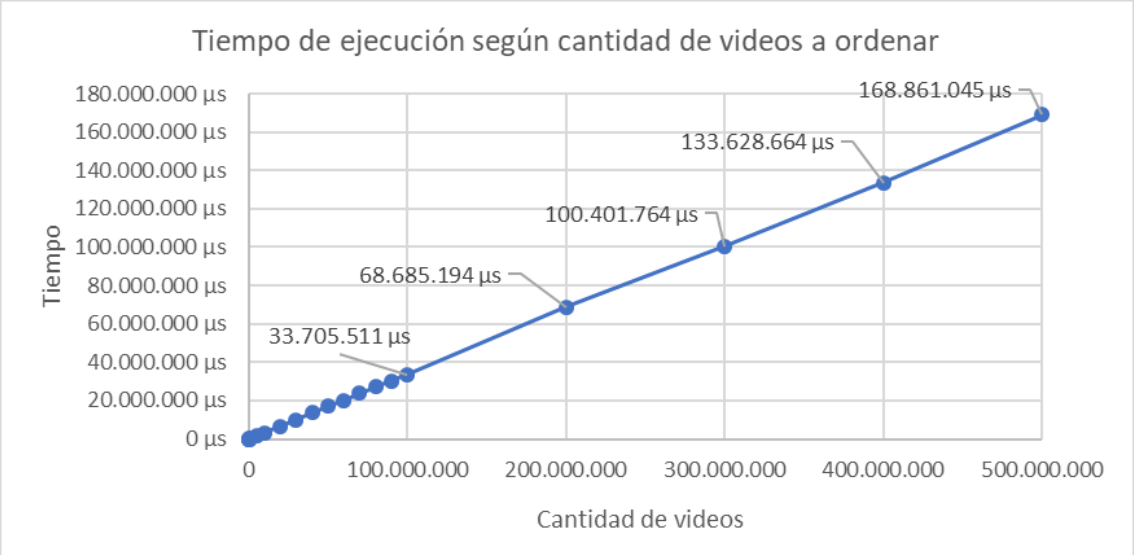
A simple vista el análisis del gráfico sugiere que a mayor cantidad de elementos la complejidad real se acerca más a $O(n)$ que a la complejidad teórica calculada $O(nlog(n))$

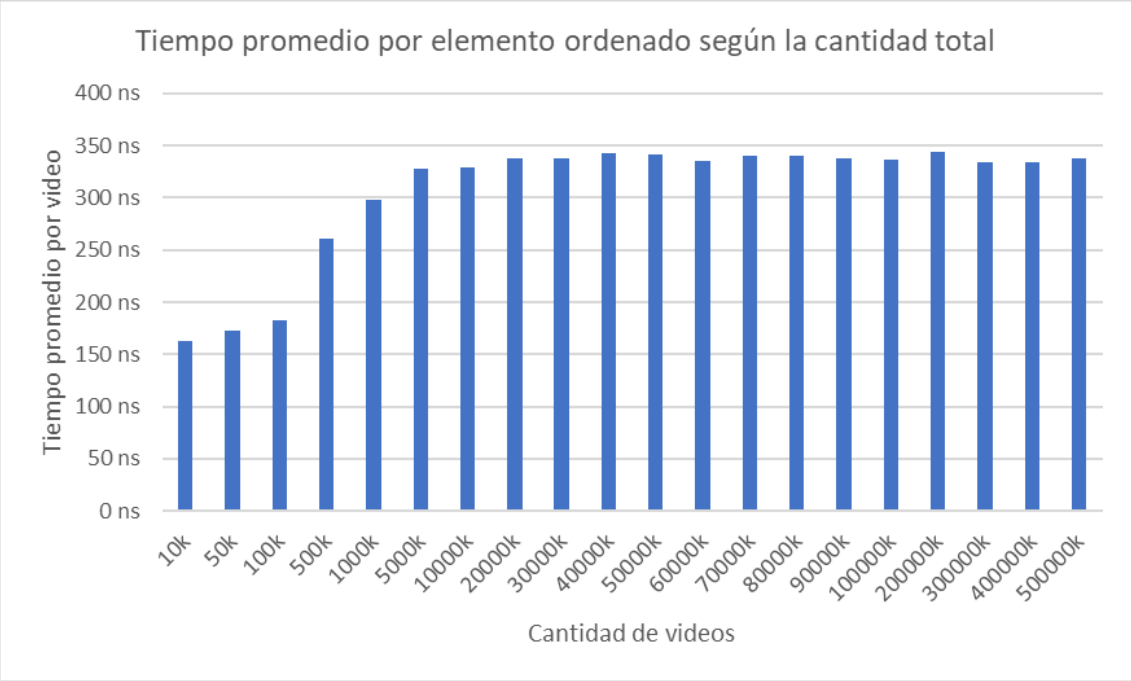
El siguiente gráfico muestra las variaciones de tiempo por cada ítem que el algoritmo debe ordenar según la cantidad de elementos totales.



Se observa que efectivamente con el aumento en la cantidad de elementos a ordenar el costo de ordenar cada elemento aumenta aunque también disminuye la diferencia.

Para seguir investigando estos resultados se realizaron las mismas pruebas pero con otras cantidades de entrada y se descubrió que efectivamente a partir de un punto la complejidad se comporta de manera lineal. A continuación se muestran los gráficos con una escalada de hasta 500 millones de elementos a ordenar.





6. Referencias

- https://codeblab.com/wp-content/uploads/2009/09/DualPivotQuicksort.pdf
- https://github.com/dart-lang/sdk/blob/a75ffc89566a1353fb1a0f0c30eb805cc2e8d34c/sdk/lib/internal/sort.dart