Teoría de Algoritmos I
(75.29) Curso Buchwald - Genender

Trabajo Práctico 1
Algoritmos Greedy

---

18 de Septiembre de 2023

Martín González Prieto 105738

Santiago Langer 107912

Camila Teszkiewicz 109660

Índice

Índice **2**
Análisis del problema 3
Posibles soluciones **3**
Propuesta 1: Ordenar de menor a mayor según el tiempo de análisis de Scaloni: 3
Propuesta 2: El análisis de ayudante más corto al final: 5
Propuesta 3: Análisis del ayudante más largo al principio: 6
Solución óptima 7
Propuesta 4: Ordenar a los ayudantes de mayor a menor: 7
Implementación en Python 8
Análisis de complejidad **8**
Análisis del tiempo de ejecución 9
Comprobación con múltiples sets de datos **10**
Sets oficiales de la cátedra 10
Sets propios 12
Conclusión **13**

Análisis del problema

En este informe buscaremos resolver un problema de tipo scheduling, donde debemos encontrar un orden de análisis que garantice que se analicen a todos los equipos en el menor tiempo posible. Se conoce de antemano el tiempo de análisis de Scaloni y el tiempo de análisis de algún ayudante para cada equipo en particular. Conociendo estos tiempos, debemos encontrar un criterio greedy que encuentre siempre algún orden de análisis óptimo que minimice el tiempo en el que se terminen todos los análisis.

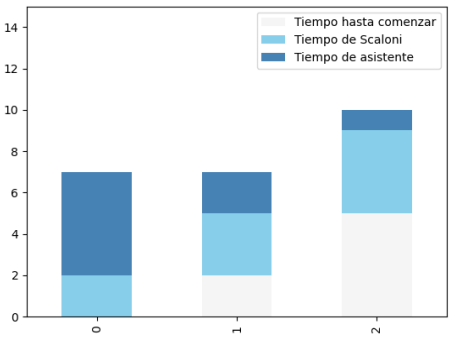
Posibles soluciones

Veamos algunas posibles soluciones con distintos criterios greedy:

Propuesta 1: Ordenar de menor a mayor según el tiempo de análisis de Scaloni:

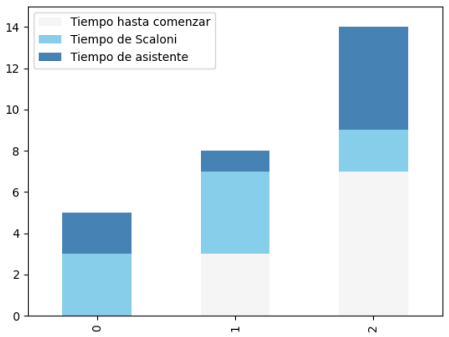
El objetivo de este criterio es que Scaloni esté libre para comenzar un nuevo análisis lo antes posible. Probemos un caso [2, 5] [3, 2] [4, 1] siendo el primer elemento de cada par el tiempo de análisis de Scaloni (S_i) y el segundo el tiempo de análisis de un ayudante (A_i):

Como los ayudantes pueden superponerse mientras analizan distintos equipos, el tiempo total que toma el análisis son 10 unidades de tiempo, según esta primera propuesta. Se puede comprobar fácilmente que es el orden óptimo comparado con todos los otros órdenes de análisis posibles:

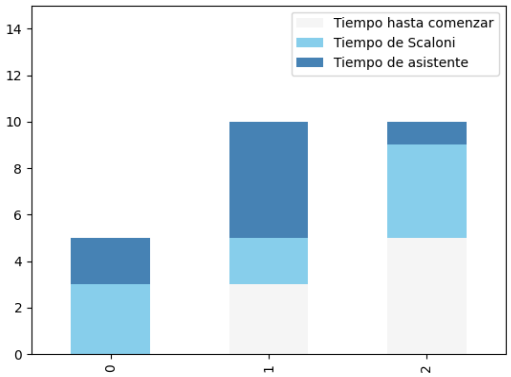


$[2, 5]\ [3, 2]\ [4, 1] = 10$

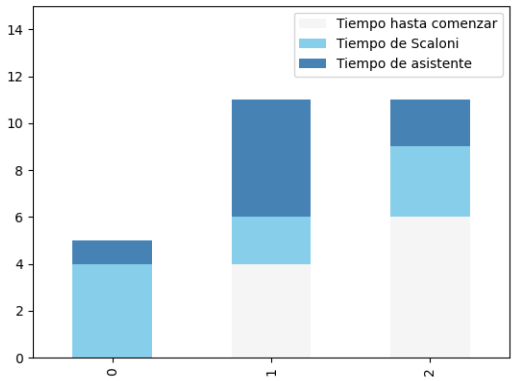
Solución óptima mediante algoritmo



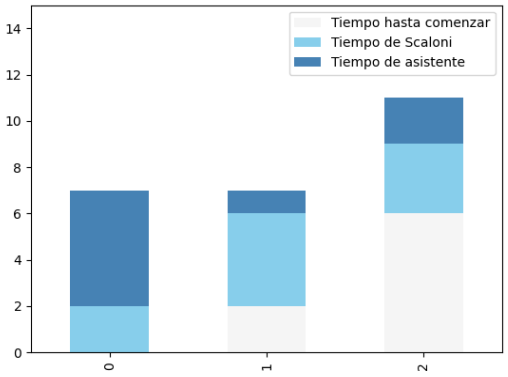
$[3, 2]\ [4, 1]\ [2, 5] = 14$



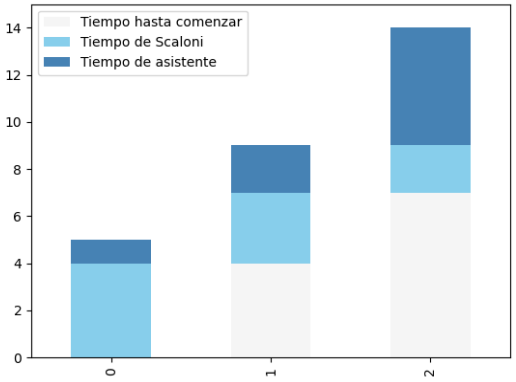
$[3, 2] [2, 5] [4, 1] = 10$
Solución también óptima



$[4, 1] [2, 5] [3, 2] = 11$



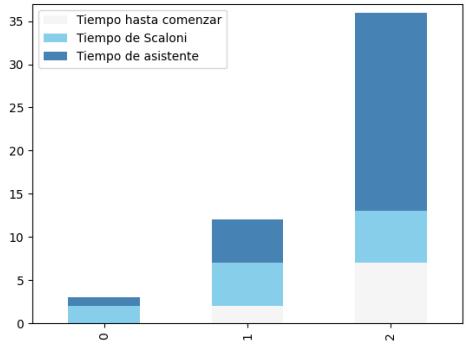
$[2, 5] [4, 1] [3, 2] = 11$



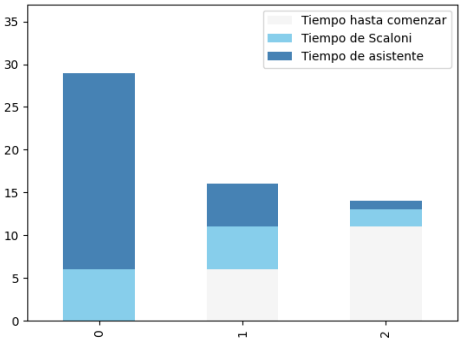
$[4, 1] [3, 2] [2, 5] = 14$

Observamos que, en este caso, el algoritmo propuesto alcanza el resultado óptimo, ya que tiene un tiempo de finalización menor o igual al resto de los ordenamientos posibles.
Sin embargo, es muy fácil encontrar un contraejemplo. ¿Qué pasa si el A_i de un equipo en particular tarda muchísimo más que el resto de los A_i de otros equipos?

Veamos un contraejemplo con [6, 23] [5, 5] [2, 1]:



[2, 1] [5, 5] [6, 23] = 36
Solución del algoritmo

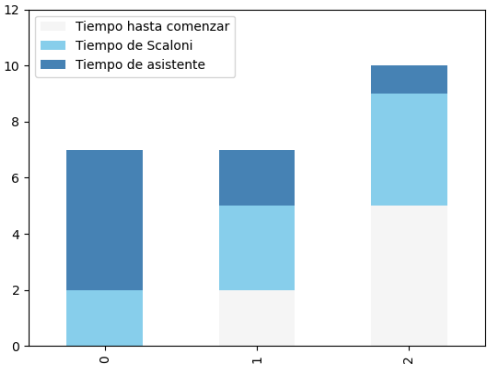


[6, 23] [5, 5] [2, 1] = 29
Solución óptima

Este contraejemplo nos indica que conviene que el ayudante más largo "colisione" con el resto de los análisis pendientes. Es decir, ***comience su análisis tempranamente y que su largo análisis se desarrolle paralelamente al resto*(1)**. Retomaremos esta idea más adelante.

Propuesta 2: El análisis de ayudante más corto al final:

Observando el problema podemos notar que el último análisis siempre terminará en la suma de todos los S_i más el análisis del propio ayudante (el último análisis no puede comenzar hasta que Scaloni haya visto todos los anteriores). Por ejemplo en el ordenamiento [2, 5] [3, 2] [4, 1] el último análisis termina en 10, es decir $2 + 3 + 4 + 1$. *Esto nos permite pensar que el orden de los S_i no importa*, pues el tiempo total de análisis lo definen los análisis de ayudante.



***De ahora en adelante, no tendremos en cuenta el largo del análisis de Scaloni.* (2).**

Sabiendo esto, y tomando como suposición que el análisis siempre termina cuando termina el último, podríamos tener como único criterio que el análisis de ayudante más corto vaya al final. Por supuesto, esto tiene un contraejemplo sencillo: el orden de la figura A termina en 11 cuando debería terminar en 10 (figura B).

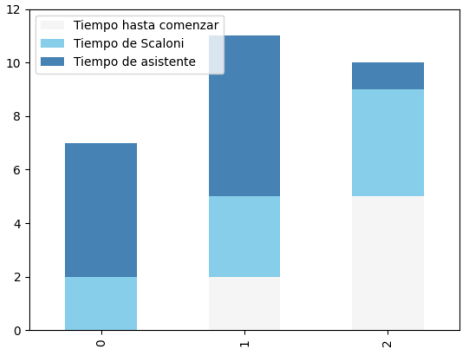


Figura A

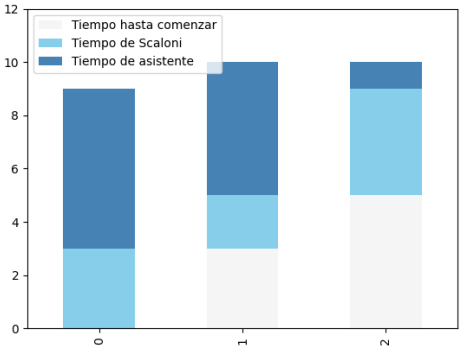
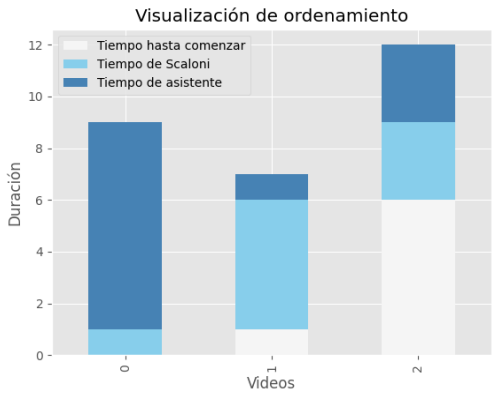


Figura B

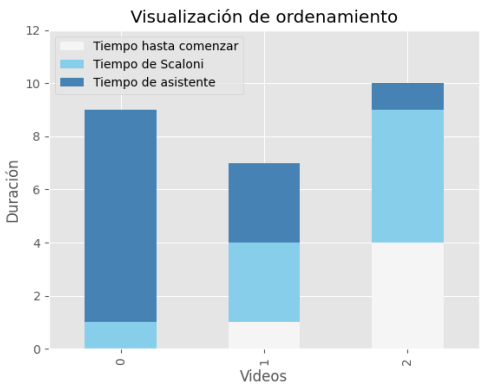
En este caso habríamos alcanzado la solución óptima si el segundo elemento estuviese primero, es decir el penúltimo A_i más corto estuviese penúltimo. Entenderemos en cuenta este **orden decreciente (del largo de A_i)** más adelante (3).

Propuesta 3: Análisis del ayudante más largo al principio:

Previamente hemos hablado de colisiones, y la importancia que tienen en nuestro problema. Queremos garantizar que el último análisis de ayudante termine lo más cercano posible a la suma de tiempos de Scaloni. Lo óptimo sería que el análisis que más tiempo tome a los ayudantes se haga cuanto antes, de esta manera un análisis largo colisionará con la mayor cantidad de análisis posibles. Tenemos n ayudantes para realizar n análisis, podemos usarlos libremente y **maximizar las colisiones (1).**



Solución según propuesta 3 (12)



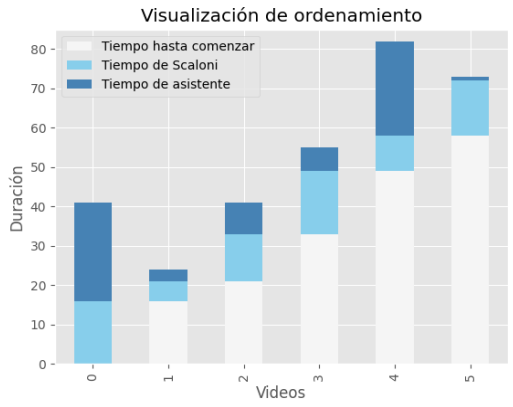
Orden óptimo (10)

Este criterio tampoco es suficiente, como se nota en este contraejemplo, pero nos sirve para construir la solución definitiva.

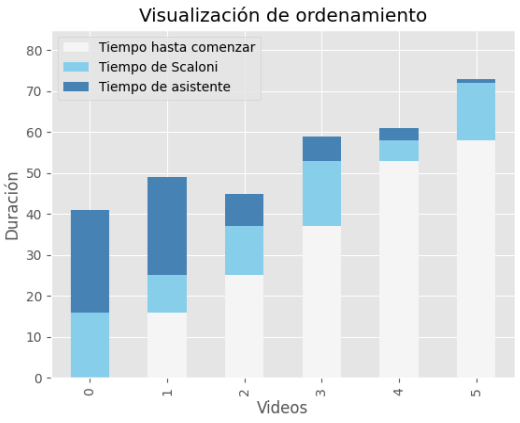
Solución óptima

Propuesta 4: Ordenar a los ayudantes de mayor a menor:

En las propuestas anteriores notamos que el A_i más largo al principio maximiza las colisiones, y que el A_i más corto al final minimizaba el tiempo final. Entonces nos preguntamos si aplicando ambas propuestas a la vez podemos alcanzar la solución óptima. Esta unión, si bien funciona para los ejemplos que venimos usando, no alcanza para sets de datos más grandes.



Solución uniendo propuestas 2 y 3 (82)



Solución óptima (73)

Al no ser suficiente simplemente unir estos criterios, podríamos combinarlos. Si el A_i más largo debe ir al principio y el más corto debe ir al final, **entonces todos los A_i intermedios deberían seguir este orden, es decir que debemos ordenar los análisis de mayor a menor,** los análisis de ayudante más largos al principio y los más cortos al final.

El criterio greedy debe ser analizar al equipo con A_i más largo (de los equipos aún no analizados), habiendo ordenado decrecientemente en función del largo de A_i antes de comenzar a elegir los equipos a analizar.
Este criterio sintetiza todas las conclusiones sacadas a lo largo de la evaluación de las diferentes propuestas: hace que el análisis de ayudante más largo comience al principio (1), no tiene en cuenta la duración u orden de los análisis de Scaloni (2) y envía el análisis de ayudante más corto al final (3). Creemos que esta es la solución óptima, ya que toma todo lo demostrado anteriormente y funciona con todos los ejemplos mencionados

hasta ahora. En próximas secciones pondremos a prueba varios sets de datos para intentar comprobar que este criterio greedy soluciona el problema que buscamos enfrentar.

Implementación en Python[1]

```
def get_optimal_analisis_order(teams_list):
    optimal_analisis_order = sorted(teams_list, key=lambda team:team[ASSISTANT_INDEX], reverse=True)
    return optimal_analisis_order

def get_analisis_duration(analisis_order):
    team_analisis_max_duration = 0
    scaloni_wait_time = 0
    for team in analisis_order:
        scaloni_wait_time += team[SCALONI_INDEX]
        if scaloni_wait_time + team[ASSISTANT_INDEX] > team_analisis_max_duration:
            team_analisis_max_duration = scaloni_wait_time + team[ASSISTANT_INDEX]
    return team_analisis_max_duration
```

La primera función ordena la lista de tiempos A_i en orden decreciente, mientras que la segunda función recorre todo el orden y se queda con el análisis de equipo que termina más tarde.
Para correr el algoritmo localmente, se adjunta en el repositorio un archivo de Python y un readme con instrucciones de cómo ejecutarlo.

Análisis de complejidad

Timsort

Class	Sorting algorithm
Data structure	Array
Worst-case performance	$O(n \log n)$[1][2]
Best-case performance	$O(n)$[3]
Average performance	$O(n \log n)$
Worst-case space complexity	$O(n)$

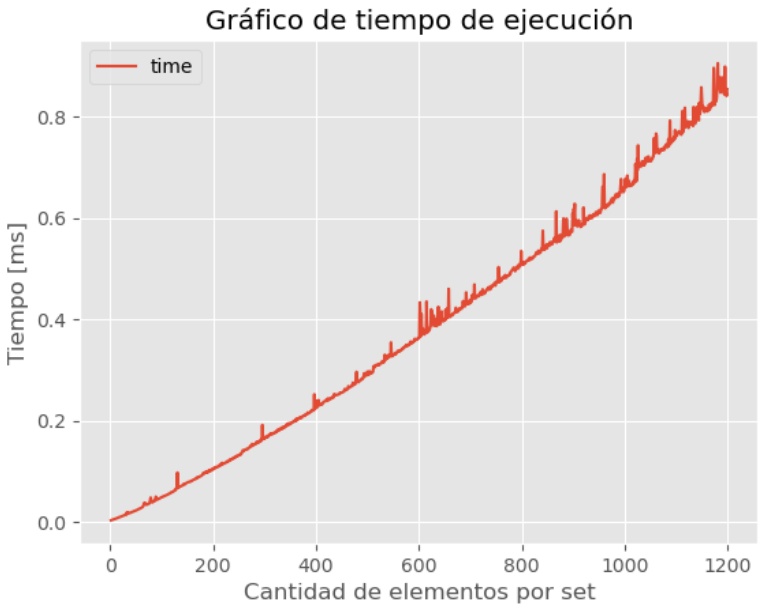
En el algoritmo diseñado es elemental el ordenamiento. Para esto, aprovechamos el método *sorted*() incluído en el lenguaje. Según la documentación de Python al respecto, la función está implementada con TimSort.[2]

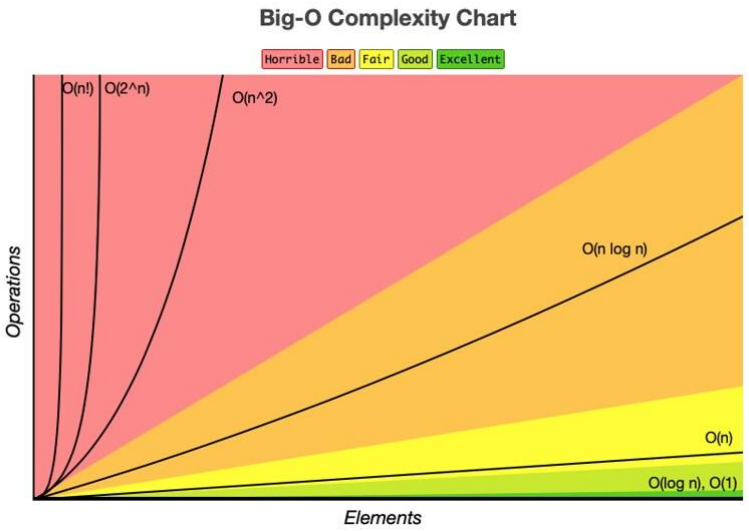
Consideramos que no es objeto del presente trabajo desarrollar el funcionamiento de timsort, pero mencionamos un par de particularidades: es un algoritmo de ordenamiento estable (si dos elementos son iguales no los permuta) derivado de merge sort e insertion sort.
En la próxima sección comprobaremos esta complejidad.

[1] Decidimos no agregar una explicación en pseudocódigo ya que Python es fácilmente legible y se asemeja al pseudocódigo.
[2] Fuente de complejidad de Timsort: Wikipedia y Tim Peters

Análisis del tiempo de ejecución

Para poder calcular el rendimiento temporal de ejecución de nuestra propuesta de solución, tomamos mediciones de su funcionamiento (cuanto milisegundos tarda en ejecutarse) para un conjunto aleatorio de sets de datos incrementales hasta 1200 elementos (véase *gráfico de tiempo de ejecución*). Estas mediciones se realizaron varias veces para cada set con la intención de sacar un promedio e independizarnos de los procesos simultáneos del procesador.

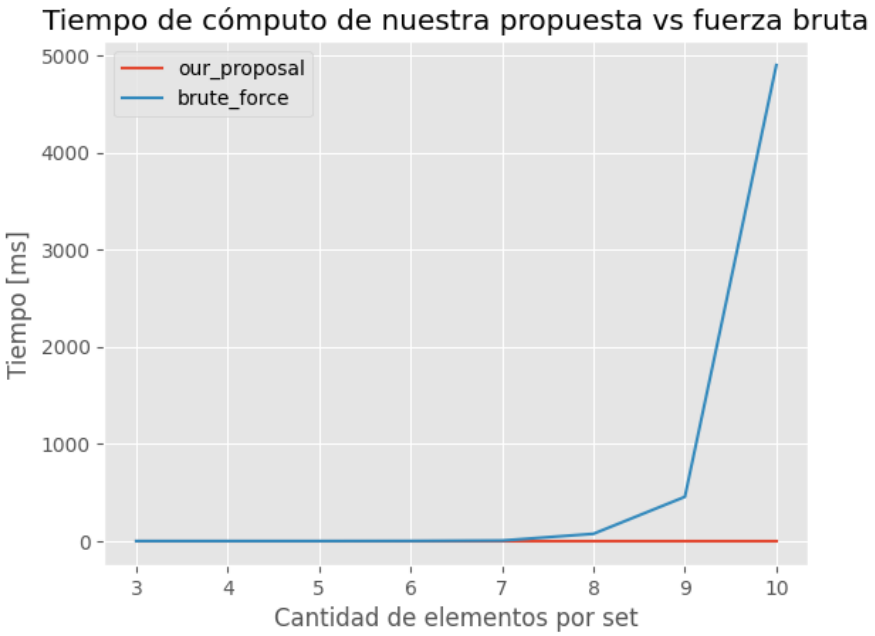




[Gráfico de referencia]

Como se puede observar, cumple con el comportamiento de un Timsort con complejidad O(n log(n)) si lo comparamos con el *gráfico de referencia.*
Así mismo, para poder tomar una dimensión del beneficio de algoritmo propuesto, también utilizamos un algoritmo de fuerza bruta. Dicho algoritmo compara todas las permutaciones que se pueden realizar con el conjunto de elementos quedándose con la que menor duración total tenga. Sin embargo, dicha alternativa es altamente costosa en complejidad ya que se trata de O(n!), la cual es de las peores complejidades que puede tener un algoritmo. Para correr dicha solución no se pudieron utilizar sets de datos mayores a 10 elementos ya que su uso de tiempo y memoria fue enorme.

En el gráfico *Tiempo de cómputo de nuestra propuesta vs fuerza bruta* se puede observar el tiempo de cómputo en relación a nuestra solución propuesta. Dada la poca cantidad de elementos es difícil notar que nuestra solución es O(n log n), sin embargo sí es notable la lentitud de fuerza bruta y la gran diferencia de un algoritmo frente al otro.



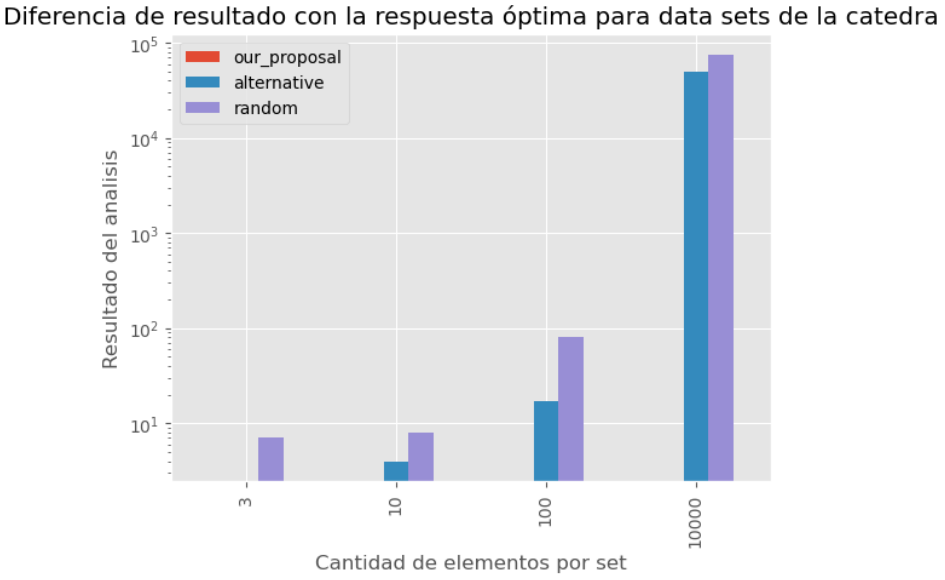
Comprobación con múltiples sets de datos

Para respaldar nuestra justificación de optimalidad, pondremos a prueba al algoritmo y su criterio con varios sets de pruebas, algunos provistos por la cátedra y otros generados por nosotros.

Sets oficiales de la cátedra

Comenzando con los datos provistos por la cátedra, nuestro algoritmo llega a una solución óptima. Para esto utilizamos los tiempos esperados dados en *tiempos_optimos.txt* y para obtener una mejor dimensión de las ganancias de nuestro algoritmo, lo comparamos con diferentes soluciones. Primeramente *alternative* que se refiere a la primera propuesta realizada dentro de las posibles soluciones, y luego *random*, una que ordena aleatoriamente los elementos del set.

En el siguiente gráfico podemos observar que la diferencia entre el tiempo óptimo y el tiempo obtenido por nuestra solución siempre es nulo, por lo que daría a entender que es la solución óptima, o al menos una solución equivalente a la alcanzada por la cátedra[3]. En cambio para el resto de soluciones (la alternativa y la aleatoria) vemos que, a excepción del caso del set de 3 elementos, siempre hay una diferencia notable. En el caso del set de 3 elementos coincidieron la solución propuesta con la obtenida por el algoritmo alternativo, ya que en dicho caso tanto ordenar de menor a mayor el tiempo de Scaloni como ordenar de mayor a menor el tiempo de los ayudantes resultó en la misma solución. Sin embargo, se puede ver que para todos los casos, ésta solución alternativa siempre representa una mejora respecto del ordenamiento aleatorio pese a no ser la solución óptima.



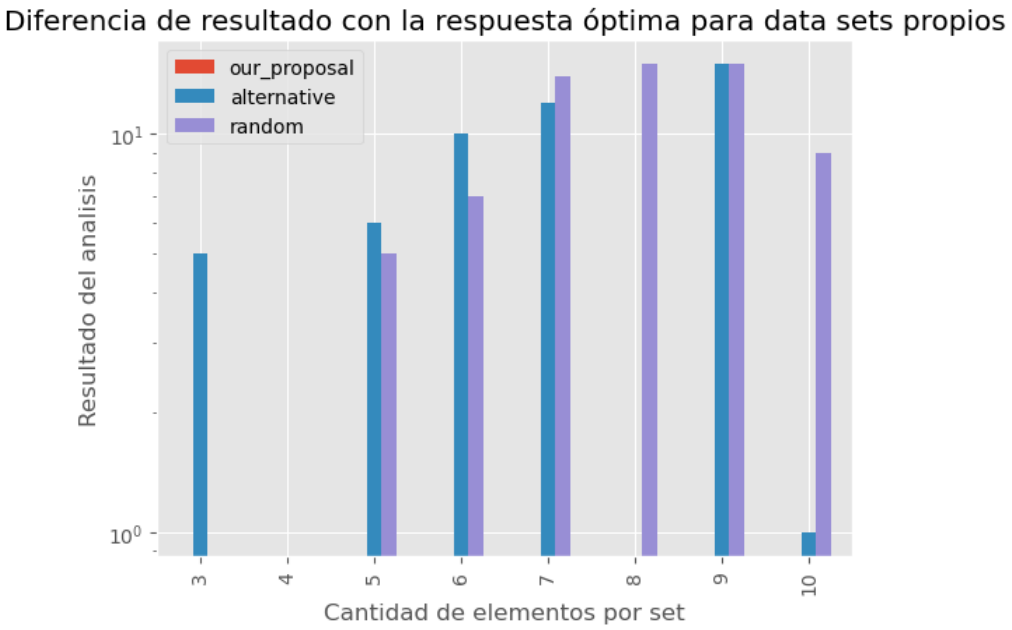
[4]

[3] En la sección dónde analizaremos los datasets propios se explicará que ese no es siempre el caso.

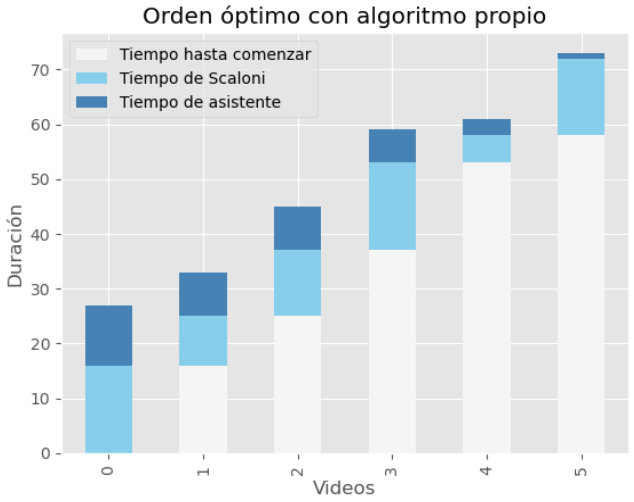
[4] Nótese la escala logarítmica para poder incluir cantidades de elementos tan distintas en un mismo gráfico. También es importante notar cómo nunca se ven barras naranjas, que son las que corresponden a nuestra propuesta. La diferencia con la respuesta óptima siempre es cero.

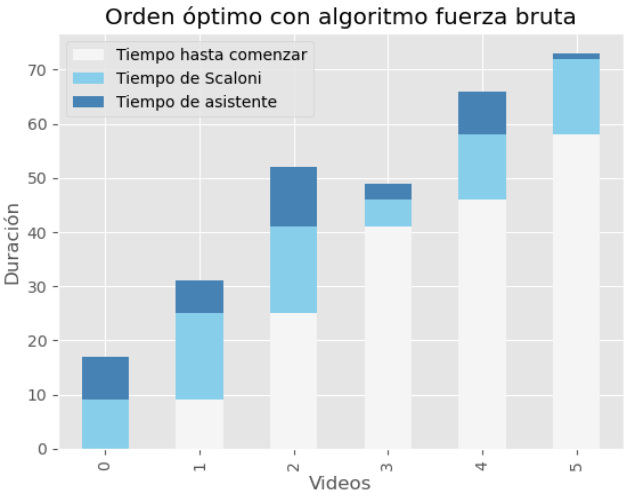
Sets propios

Para tener más muestras generamos más sets de datos y encontramos sus soluciones mediante fuerza bruta. Una vez tuvimos estas respuestas, corrimos nuestro algoritmo y comprobamos nuevamente que genera una solución óptima al compararlas con la solución por fuerza bruta. Como previamente explicamos, conseguir la solución por fuerza bruta tiene una gran limitación debido a que cuenta con una gran complejidad algorítmica y por ende generamos sets con pocos elementos. A pesar de eso, el siguiente análisis nos permitió tomar mejores conclusiones sobre lo obtenido con los datasets de la cátedra y tener aún más seguridad que nuestra solución es óptima.



Como se puede observar, la conclusión parcial anterior de que el algoritmo *alternative* tiene una mejor performance que el ordenamiento aleatorio es falsa. En varios de los sets tuvo un mejor acercamiento el algoritmo *random*, de hecho en varias ocasiones se llega a una solución óptima. Dado que se trata de un comportamiento aleatorio y que, en el caso de la solución *alternative*, depende en gran parte del conjunto de elementos que deben tratar, no se puede llegar a ninguna conclusión al respecto de estos algoritmos.





Utilizar tanto un algoritmo propio como una solución por fuerza bruta nos permitió encontrar más de un ordenamiento óptimo distinto. En este caso, los gráficos muestran claramente que el orden en el que se seleccionaron por cada algoritmo son diferentes, sin embargo en ambos casos se llega a la misma duración total de 71 unidades de tiempo.

Conclusión

Mediante el transcurso de este estudio hemos llegado a una solución que comprobamos óptima, tanto por nuestra demostración como por su puesta a prueba con sets de datos externos y propios.
Llegar a un algoritmo greedy que soluciona el problema solo fue posible mediante el estudio de todas las propiedades del ejercicio y la puesta a prueba de todas las soluciones ideadas (incluso las iniciales que luego resultaron erróneas). Habría sido imposible llegar al criterio de la solución óptima si no tomábamos nota de los detalles que aparecían al probar cada solución y su contraejemplo.
Una observación interesante en la resolución de este problema es que, si bien a primera vista pareciera tener dos variables relevantes (el tiempo que tarda Scaloni y el que tarda un ayudante), en realidad depende de una sola, ya que por la naturaleza del problema el tiempo total que demora Scaloni es invariable.

También es curioso notar que dos soluciones óptimas (que alcanzan la misma duración mínima) no tienen por qué ser iguales. En varios casos vimos cómo las mismas soluciones óptimas daban órdenes distintos.

Por último podemos notar que fuerza bruta también llega a la solución, pero de todas formas fue elemental encontrar una solución greedy para poder tratar el problema en casos de mayor volumen. Sin este algoritmo greedy no habríamos podido encontrar el orden y tiempo mínimo de un caso con decenas, centenares, o miles de equipos a analizar.

Esperamos que Scaloni y la Selección Argentina estén satisfechos con nuestro trabajo.