Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

Trabajo Práctico 1
Algoritmos Greedy

18 de septiembre de 2023

Guillermo Stancanelli	Simón Stein	Santiago Tissoni
104244	106127	103856

1. Consideraciones

En el trabajo practico se nos propone realizar un analisis al siguiente problema:

Se plantea la situación en la cual Scaloni y su cuerpo tecnico tienen que realizar analisis de ciertos rivales a enfrentar. Cada analisis, tiene que ser realizado tanto por Scaloni como por alguno de sus n ayudantes. Cada tiempo de analisis es conocido previo al mismo y denominaremos Si al tiempo que le toma a Scaloni analizar al rival i y Ai al tiempo que le toma a alguno de los ayudantes de Scaloni. Ademas, para que un ayudante pueda analizar a un rival, primero tiene que haber sido analizado por Scaloni. La cantidad de rivales a analizar es tambien n.

En este documento, plantearemos una solucion para el problema usando un algoritmo Greedy. Realizaremos un analisis tanto de optimalidad como de complejidad del mismo, mostrando mediciones y casos de prueba.

2. Algoritmo para encontrar el tiempo mínimo de análisis

Realizaremos a continuación el análisis de diferentes algoritmos que fuimos explorando para la solución al problema planteado: obtener el tiempo mínimo de análisis para los n rivales que enfrenta la selección.

2.1. Primer acercamiento Greedy: ordenar por Si ascendiente

Como primera solución, intuitivamente decidimos ordenar la lista de rivales ascendientemente por el tiempo que le toma a Scaloni analizarlos (en caso de empate, eligiendo por Ai descendente. De esta manera, podríamos asegurar que rápidamente varios ayudantes obtendrían un video a analizar, y por lo tanto minimizaríamos cuántos de ellos están inactivos. Una vez realizado este ordenamiento, simplemente deberíamos recorrer linearmente la lista de los pares (Si, Ai), quedándonos con el tiempo del ayudante que último termine, es decir $max(\sum_{n=0}^{i}(\text{Sn})+\text{Ai})\forall i\epsilon N$, como tiempo total del proceso.

Sin embargo, este algoritmo nunca llegó a implementarse en código, ya que rápidamente encontramos contraejemplos que evidenciaban tardanzas en el mismo:

S_i	A_i
1	6
1	5
1	3
2	9
2	4
2	2
4	8
4	3
5	1
6	5

Rápidamente podemos notar en esta lista de partidos ya ordenados por el algoritmo, que el último partido en ser analizado por Scaloni tiene un Ai relativamente alto. El tiempo total de análisis de partido $T = max(\sum_{n=0}^{i}(\text{Sn})+\text{Ai})\forall i\epsilon N = 33$

Es acá cuando nos percatamos que un análisis de partido con valores altos tanto en Si como Ai sería postergado al final del proceso, con malos resultados para el tiempo total. Ya que Scaloni siempre tardará lo mismo independientemente del órden en que mire los partidos, decidimos buscar un ordenamiento distinto que priorize minimizar el tiempo del último ayudante en terminar.

2.2. Solución Greedy: ordenar por Ai descendiente

Viendo los resultados del primer algoritmo propuesto, decidimos cambiar el criterio de ordenamiento de los análisis de partidos, esta vez ordenando por Ai descendiente (y por Si ascendiente en caso de empate). Una vez hecho esto recorremos linearmente la lista de los pares (Si, Ai), quedándonos con el tiempo del ayudante que último termine, es decir $max(\sum_{n=0}^{i}(\text{Sn})+\text{Ai})\forall i\epsilon N$, como tiempo total del proceso.

A continuación, mostramos la implementación de este algoritmo en lenguaje Python.

```python
class MatchReview:
    def __init__(self, si, ai):
        self.si = si  # coach analysis duration
        self.ai = ai  # helper analysis duration

    def __lt__(self, otherReview):
        return self.ai > otherReview.ai or (
            self.ai == otherReview.ai and self.si < otherReview.si)
```

```python


def full_review_time(review_list):
    sorted_reviews = sorted(review_list)

    curr_si = 0
    curr_ai = 0

    for review in sorted_reviews:
        curr_si += review.si
        curr_ai = max(curr_ai, curr_si + review.ai)

    return curr_ai
```

Aplicando este algoritmo al mismo conjunto de partidos, obtenemos el siguiente órden para los mismos:

S_i	A_i
2	9
4	8
1	6
1	5
6	5
2	4
1	3
4	3
2	2
5	1

Además obtenemos un tiempo total de análisis $T = max(\sum_{n=0}^{i}(\text{Sn})+\text{Ai})\forall i \epsilon N = 29$, que es superior a nuestro primer acercamiento.

Podemos justificar que este es un algoritmo greedy, ya que se puede pensar que el ordenamiento realizado equivale a que cada vez que se elige el siguiente partido a analizar, se toma aquel de mayor Ai para evitar que ese ayudante cause retrasos si se eligiera al final. De esta forma eligiendo iterativamente los óptimos locales, esperamos llegar a una solución óptima global.

2.3. Análisis de complejidad

Como podemos observar, el algoritmo consta de dos partes centrales. Ordenar el vector según el criterio previamente mencionado y recorrerlo para sumar los valores de tiempo y llegar al resultado.

Calcular la complejidad del mismo, resulta entonces de la suma de las complejidades de ambas partes.

$$\mathcal{T}(n) = \mathcal{O}(n) + \mathcal{O}(nlog(n))$$

Tomando el termino más pesado, resulta en:

$$\mathcal{T}(n) = \mathcal{O}(nlog(n))$$

2.4. Justificación de optimalidad

Para justificar que nuestro algoritmo greedy es óptimo, buescamos probarlo por inversión:

Supongamos que existe una solución óptima que no sigue la estrategia greedy. En esta solución, supongamos que hay dos rivales consecutivos Ri y Rj (con Ri analizado antes que Rj donde o bien $Ai < Aj$ o ($Ai = Aj$ y $Si > Sj$).

Intercambiamos para esa solución la posición de estos dos rivales. Analizamos las consecuencias de esta acción:

- Si $Ai < Aj$: Dado que Rj le toma más tiempo al ayudante, al colocarlo antes, el ayudante inicia este análisis largo antes, y por lo tanto no es posible que afecte el tiempo total. Esta inversión puede mejorar el tiempo general de análisis.
- Si $Ai = Aj$ y $Si > Sj$: Intercambiar los rivales significa que Scaloni primero analiza al rival que le lleva menos tiempo, y por lo tanto le pasa al ayudante el análisis un poco antes. De nuevo, este intercambio no puede empeorar la situación, y podría mejorarla.

Podemos repetir esta operación de inversión para cualquier par de rivales consecutivos que no sigan el orden greedy, transformando gradualmente la hipotética solución óptima en la solución greedy sin hacerla peor.

Dado que podemos transformar cualquier solución óptima en la solución greedy sin aumentar el tiempo total, la solución greedy debe ser también óptima.

2.5. Análisis de variabilidad de Ai y Si

A continuación, analizaremos cómo la variabilidad de los tiempos de Ai y Si afectan la optimalidad y el tiempo de ejecución de nuestro algoritmo:

Si los valores de Ai son muy variados, significa que hay una gran diferencia entre los tiempos que diferentes ayudantes toman para analizar diferentes rivales. En tales casos, es esencial que el algoritmo greedy priorice aquellos rivales que toman más tiempo para los ayudantes. Esto es precisamente lo que hace el algoritmo al ordenar basándose en Ai decreciente, por lo que debería ser efectivo en estos escenarios.

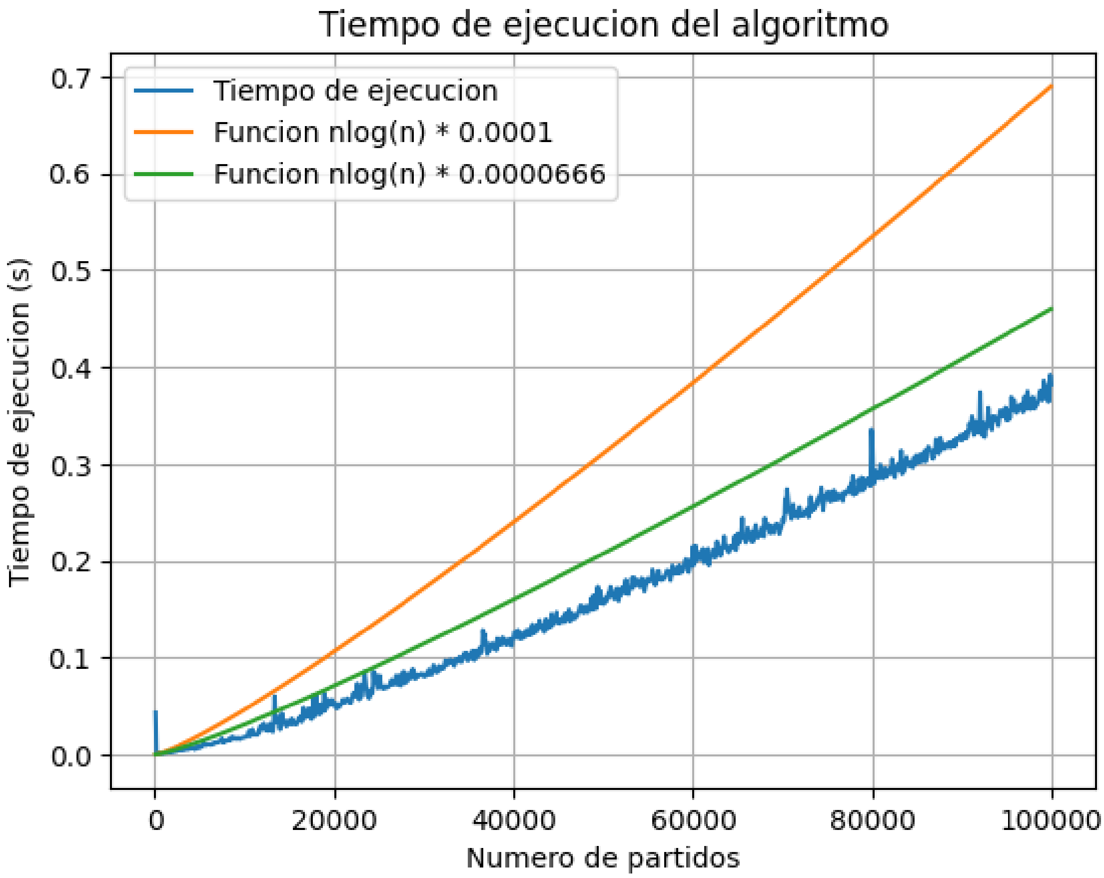
Por otro lado, si todos los Ai son aproximadamente iguales (baja variabilidad), entonces este componente del algoritmo es menos crítico. Pero aún así, no perjudica la óptima ejecución, ya que el algoritmo simplemente pasará al siguiente criterio, que es ordenar por Si en orden ascendente. En este último caso el algoritmo tomará los de Si menor, liberando antes a Scaloni permitiendo que cualquier ayudante pueda empezar antes a analizar al rival, asegurándonos entonces la optimalidad del resultado.

En cuanto al tiempo de ejecución del algoritmo, logramos reducir completamente la dependencia de la variabilidad de Si y Ai. Esto se debe al ordenamiento que se realiza previo al reccorrido de los datos. De esta manera, el tiempo de ejecución sólo será afectado por cómo están ordenados los elementos previo a la ejecución, teniendo como cota a $O(nlog(n))$.

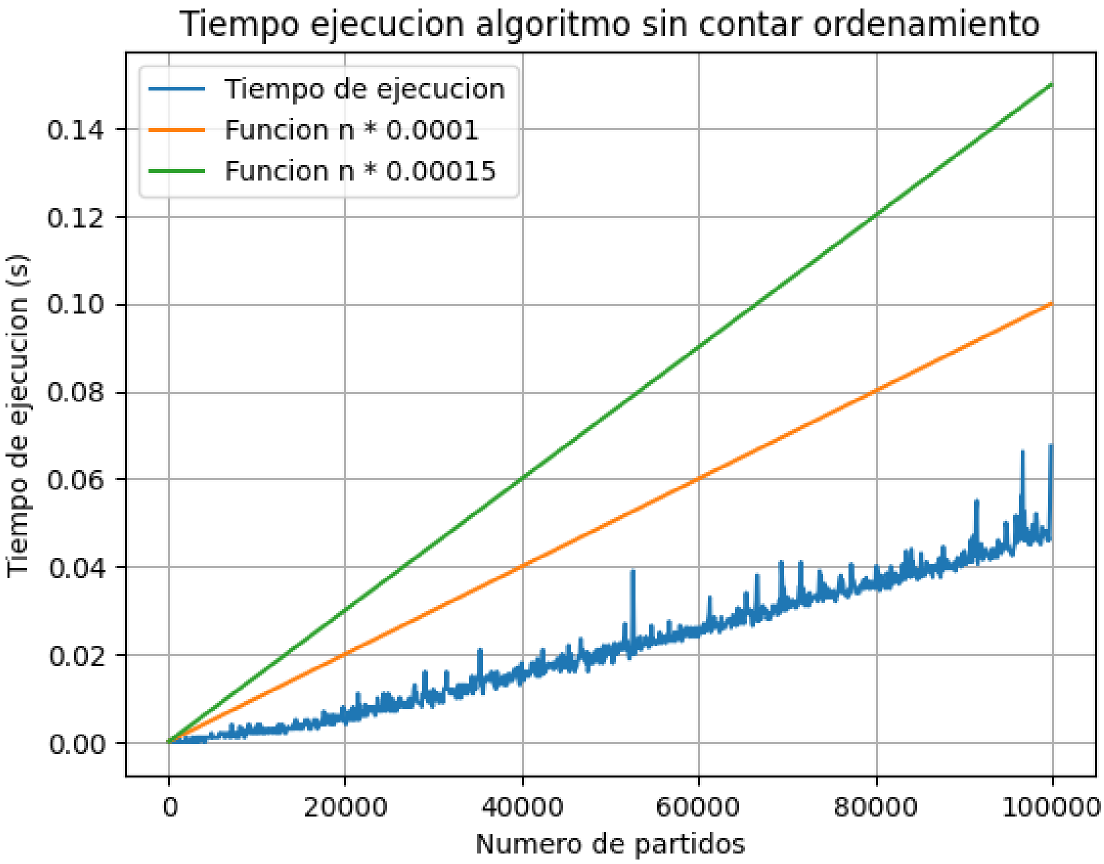
3. Mediciones

Para medir la complejidad temporal del algoritmo que propusimos para resolver el ejercicio, decidimos armar un script para la generacion de casos de pruebas de 100 en 100, es decir el primer conjunto de prueba tiene 100 reviews de rivales para analizar, el segundo 200, el tercero 300 y asi sucesivamente, aprovechando el módulo `random` de la libreria estandar de Python para generar numeros aleatorios

De esta manera, generamos 1000 conjuntos de pruebas con un crecimiento en registros de 100 en 100, asi el primer conjunto tiene 100 reviews y el ultimo tiene 100.000 registros. En el grafico podemos ver como crece de manera O(nlog(n)) nuestro algoritmo, comparandolo con una curva de crecimiento logaritmico



Ademas, para verficar que el resultado de la complejidad espacial es O(nlog(n)) debido a que el ordenamiento es O(nlog(n)) y que el resto del algoritmo es O(n), tambien corrimos las pruebas y medimos el tiempo unicamente para la parte posterior al ordenamiento, y de esta manera podemos apreciar el comportamiento lineal del resto del algoritmo



4. Conclusiones

Luego del análisis del problema y las pruebas con los algoritmos planteados, llegamos a la conclusión de que las soluciones Greedy fueron intuitivas de plantear, a pesar de no haber llegado al algoritmo óptimo en el primer intento. Debimos realizar varias iteraciones sobre el problema para finalmente hallar el algoritmo óptimo. Es decir, fuimos avanzando a prueba y error para encontrar una solución óptima. A primera vista el problema se asemejaba al Interval Scheduling visto en clase pero al adentrarnos en el tema notamos que no eran tan similares.

En nuestro caso, el concepto de ir obteniendo óptimos locales para así llegar al óptimo global, fue clave para la resolución del problema. Esta idea previamente mencionada nos llevó a darnos cuenta que ordenar los datos según algún criterio es sumamente conveniente a la hora de resolver problemas usando algoritmos Greedy. Tal es así que logramos obtener una complejidad $O(nlog(n))$ la cual creemos que es una buen resultado para nuestra solución.