
Teoría de Algoritmos

(75.29) Curso Buchwald - Genender

# Trabajo Práctico 1
# Algoritmos Greedy

24 de enero de 2024

Emanuel Pelozo 99444

Agustina Fraccaro 103199

Agustina Schmidt 103409

Matias Gonzalez 104913

# 1. Análisis del problema

El problema consiste en que Scaloni y sus ayudantes deben ver videos de sus próximos rivales para analizar la mejor estrategia de juego.

Cada rival lleva un tiempo $s_i$ en ser analizado por Scaloni y un tiempo $a_i$ en ser analizado por cualquier ayudante.

## 1.1. Búsqueda de algoritmos de solución

A continuación detallaremos los diferentes enfoques utilizados para encontrar el algoritmo que proporcione la mejor solución.

### 1.1.1. Ordenamiento ascendente por tiempos de Scaloni

Dado que todos los videos deben ser vistos primero por Scaloni antes de pasar a algún ayudante, tomamos como primer criterio que se vean ordenados por $s_i$ *(tiempo que tarda Scaloni en verlos)* de forma ascendente, es decir, que Scaloni siempre vea el que menor tiempo le demande, sin importar los tiempos de los ayudantes.

Rápidamente, encontramos un contraejemplo donde no se haya la solución óptima utilizando este criterio de ordenamiento, el mismo se puede ver en la siguiente imágen:

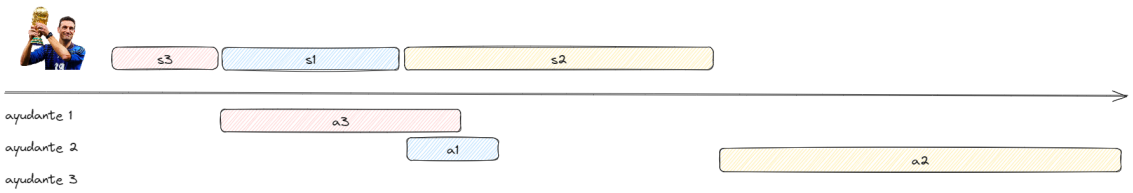



Figura 1: Contraejemplo para la solución de ordenamiento por tiempos de Scaloni.

Podemos ver que $a_2$ es el mayor tiempo para los ayudantes y se realiza último, cuando se podría realizar en paralelo a $a_1$ y $a_3$ debido a que siempre hay disponibilidad de ayudantes para revisar videos.

### 1.1.2. Ordenamiento descendiente por la diferencia entre tiempos de Scaloni y el de los ayudantes

Dado que ordenar únicamente por tiempos de Scaloni no nos dió la solución óptima, intentamos utilizar también los tiempos de los ayudantes. Para esto, elegimos relacionar los tiempos $s_i$ y $a_i$ a través de la diferencia $a_i - s_i$. En este caso pudimos utilizar el mismo caso que el anterior como contraejemplo, lo podemos ver en la siguiente imágen:

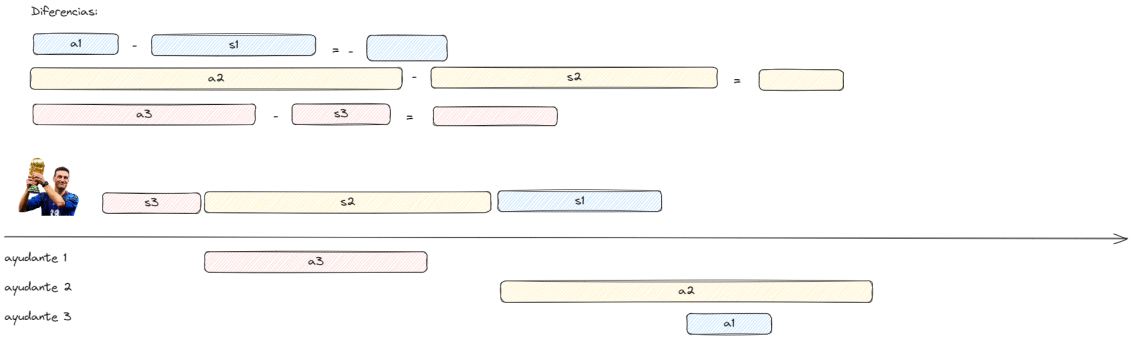



Figura 2: Contraejemplo para la solución de ordenamiento por diferencia de tiempos.

Vemos primero el resultado de las diferencias, luego vemos nuevamente que $a_2$ sigue realizándose demasiado tarde y es el que más tiempo ocupa.

### 1.1.3. Solución Óptima: Ordenamiento descendiente por tiempos de los ayudantes

A través de los anteriores análisis, notamos que el tiempo en el que Scaloni termine de analizar todos los rivales no puede optimizarse para mejorar el tiempo total, dado que ese tiempo siempre es fijo por tener que verlos de forma secuencial.

Para la demostración estableceremos que el tiempo total de análisis de todo el cuerpo técnico se representa de la siguiente manera:

$$Tiempo\ total = T_s + T_a$$

Donde $T_s$ es el tiempo que le toma a Scaloni en analizar todos los rivales y $T_a$ es el tiempo que le toma a los ayudantes.

Dado que en este problema Scaloni siempre debe ver todos los videos de forma secuencial, no importa el orden en el que los vea, en cualquier escenario va a tardar el mismo tiempo en completar todos los análisis, es decir:

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

En otras palabras, el tiempo que Scaloni tarde en analizar los rivales es **constante**. Por esto, el problema de minimizar el tiempo total de análisis consiste en minimizar $T_a$.

Por esto y teniendo en cuenta que los análisis de los ayudantes pueden superponerse podemos lograr minimzar el tiempo al comenzar los análisis de los ayudantes que tomen más tiempo primero, de manera de poder "absorber" la mayor cantidad de tiempo posible de los ayudantes que se superpongan.

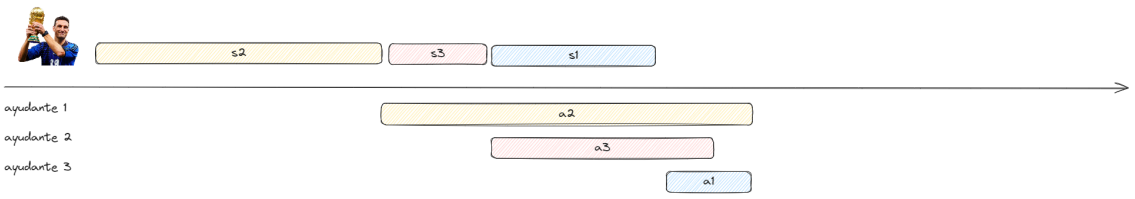



Figura 3: Ejemplo de solución ordenando por los tiempos de los ayudantes.

En el ejemplo, vemos que al ser el segundo video el primero que ve Scaloni, es el primero que se libera para ser tomado por alguno de los ayudantes. Esto hace que los próximos videos tomados se superpongan con el tiempo de $a_2$ y por lo tanto en este caso, el tiempo de ver los videos es únicamente $s_2 + a_2$, por lo cuál es lo óptimo dado que las demás revisiones se hacen en paralelo.

### 1.2. Optimalidad y variabilidad de los datos

Cómo hemos mencionado anteriormente, el tiempo que le corresponde a Scaloni dentro del análisis total ($T_s$) es constante, por esto, puede haber escenarios donde cualquier solución elegida sea la mejor.

Por ejemplo, si se presentan escenarios donde el tiempo que le tome a cada ayudante analizar sea constante, es decir $a_i = k \ \forall \ i$, ya que en este caso, cualquier orden elegido será el mejor que se pueda obtener.

A pesar de esto, el algoritmo elegido soporta cualquier tipo de variabilidad en los datos debido a su enfoque de minimizar ($T_a$).

## 2. Algoritmo para encontrar la solución óptima

A continuación se detallan las implementaciones de los algoritmos propuestos para la resolución del problema de cómo organizar los videos de rivales entre Scaloni y ayudantes.

La solución está escrita en Python, dado que no solo es más sencillo para el manejo de datos sino que también facilita la realización de gráficos.

Tenemos dos archivos principales: main.py y algoritmos.py.

### 2.1. Parseo inicial y ejecución

En esta sección organizamos los datos y la información que tenemos disponible para luego ejecutar nuestro algoritmo.

Para empezar, teniendo en cuenta los parámetros recibidos, se elige el archivo del cual leer los casos de prueba y también se define qué algoritmo se ejecutará para hacer el análisis, variando entre uno u otro la forma en que se ordenan los tiempos, como bien fue mencionado en la sección de Análisis del problema 1.

Luego, se realiza la lectura del archivo del cual obtenemos como resultado una lista de tuplas con los tiempos de Scaloni y ayudantes ($S_i, A_i$):

```
def main():
    path, algoritmo = parsear_argumentos()

    tiempos = cargar_tiempos(path)
    print("Archivo {} cargado correctamente con n={}".format(path, len(tiempos)))

    print("Utilizando algoritmo {}".format(algoritmo.__name__))
    t_i = time.time()
    tiempos_ordenados = algoritmo(tiempos)
    t_f = time.time()
    print("Tiempo de ejecucion: {} segundos".format(t_f - t_i))

    tiempo_analisis = calcular_tiempo_analisis(tiempos_ordenados)
    print("Tiempo final del analisis: {}".format(tiempo_analisis))
```

### 2.2. Algoritmos

Como se mencionó en la sección anterior, la diferencia entre los algoritmos es el criterio de ordenamiento. El código encargado de esto se puede ver a continuación:

```python
ALGORITMOS = {"ayudante": greedy_scaloni_por_ayudante,
              "diferencia": greedy_scaloni_por_diferencia,
              "scaloni": greedy_scaloni_por_scaloni}

def greedy_scaloni_por_ayudante(tiempos: List[Tuple[int, int]]) -> List[Tuple[int, int]]:
    return sorted(tiempos, key=lambda t: t[1], reverse=True)


def greedy_scaloni_por_diferencia(tiempos: List[Tuple[int, int]]) -> List[Tuple[int, int]]:
    return sorted(tiempos, key=lambda t: t[1] - t[0], reverse=True)


def greedy_scaloni_por_scaloni(tiempos: List[Tuple[int, int]]) -> List[Tuple[int, int]]:
    return sorted(tiempos, key=lambda t: t[0])
```

Donde:

- Ayudante: Devuelve lista ordenada por $A_i$ en orden descendiente.
- Diferencia: Devuelve lista ordenada por $A_i - S_i$ en orden descendiente.
- Scaloni: Devuelve lista ordenada por $S_i$ en orden ascendente.

Con los tiempos ya parseados y el algoritmo elegido, calculamos el tiempo que tarda en ejecutarse el ordenamiento para luego poder comparar:

```python
    t_i = time.time()
    tiempos_ordenados = algoritmo(tiempos)
    t_f = time.time()
    print("Tiempo de ejecucion: {} segundos".format(t_f - t_i))
```

Una vez realizado el ordenamiento, se calcula el tiempo final que lleva ver los videos de los rivales según el caso que se haya seleccionado.

```python
def calcular_tiempo_analisis(tiempos: List[Tuple[int, int]]) -> int:
    t_scaloni = 0
    t_final = 0
    for tiempo in tiempos:
        t_scaloni += tiempo[0]
        t_final = max(t_final, t_scaloni + tiempo[1])
    return t_final
```

De esta forma, podemos probar distintas maneras para luego realizar comparaciones y obtener conclusiones de cada caso.

## 3. Complejidad algorítmica

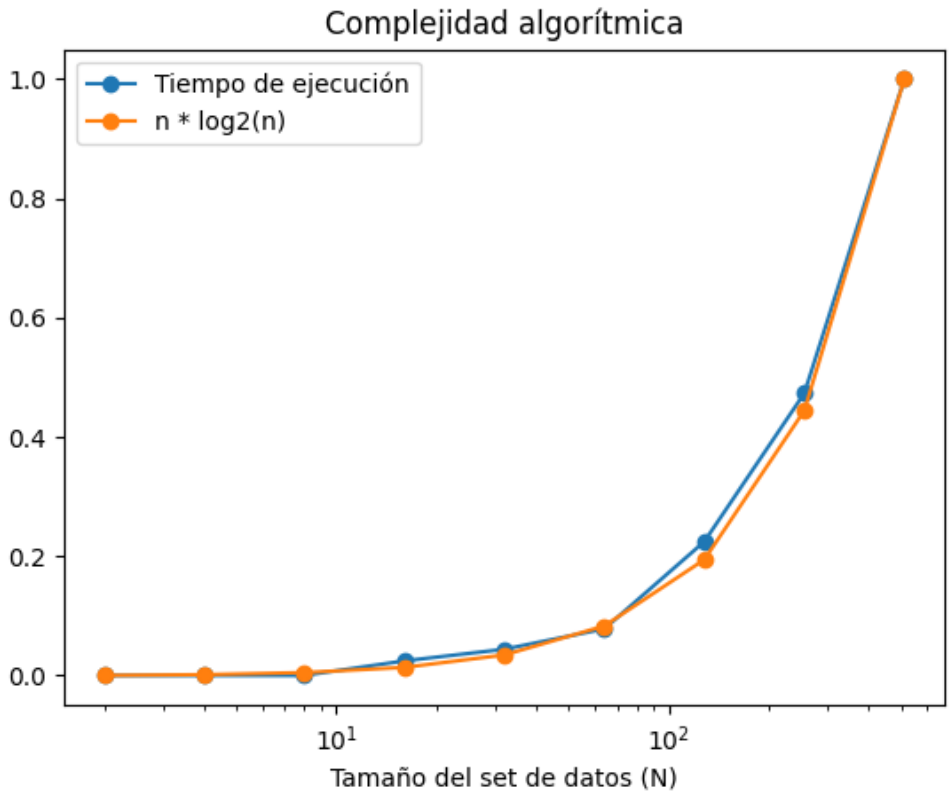
El algoritmo que devuelve el orden en el que los rivales deben ser analizados utiliza la función built-in sorted() de Python cuya complejidad algorítmica es $O(n.log(n))$ según la documentación oficial[1].

Para verificar esto, se hizo un experimento utilizando el siguiente código:

```python
t = []
nlogn = normalizar([n * math.log2(n) for n in tamanios])
for caso in casos:
    t_i = time.time()
    greedy_scaloni_por_ayudante(caso)
    t_f = time.time()
    t.append(t_f - t_i)

t = normalizar(t)
```

A través de este, se midió el tiempo de ejecución de 9 casos con tamaño de set de datos exponencial: $n_i = 2^i$. También se calculó $n.log(n)$ para cada $n_i$. Se normalizaron los valores y se graficaron superpuestos en el siguiente gráfico:




Como se puede apreciar, el algoritmo tiene una complejidad algorítmica de $O(n.log(n))$.

# 4. Comparaciones

Se realizaron distintas comparaciones sobre los tres algoritmos desarrollados, estas fueron realizadas sobre set de datos generados de manera pseudoaleatoria a través de la biblioteca random[2]. Los mismos fueron construídos asignando un valor aleatorio entre 1 y 100 para cada valor de $\mathbf{s}_i - a_i$. Estas mediciones tienen como objetivo determinar la efectividad de cada algoritmo sobre los distintos escenarios.

## 4.1. Comparación de resultados sobre muestras alteatorias

Para este primer análisis se generaron 4 conjuntos de datos de diferentes tamaños y se ejecutó sobre ellos cada uno de los algoritmos desarrollados.

A continuación se pueden observar los resultados:

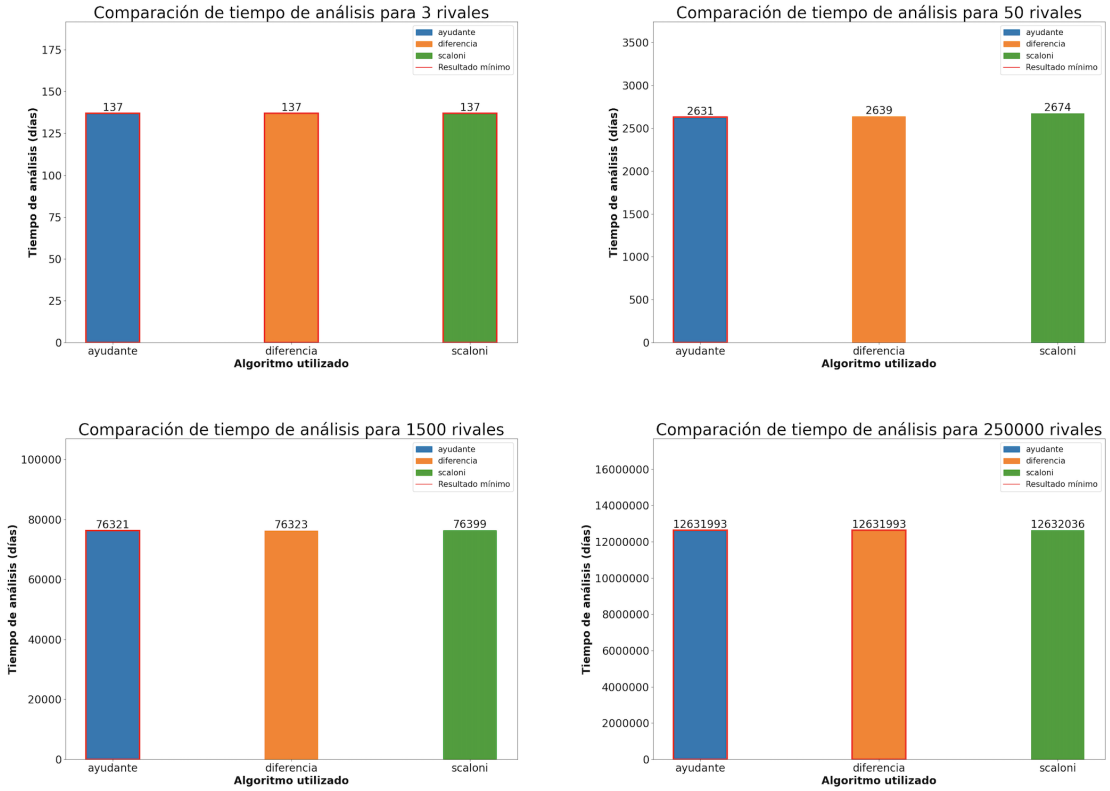



Figura 4: Comparación de resultados sobre 4 set de datos aleatorios.

A través de estas evaluaciones se puede observar cómo el algoritmo que prioriza el ordenamiento por el tiempo de los ayudantes obtuvo el mejor valor en los 4 casos evaluados, seguido por el de diferencia que compartió el mejor resultado en dos oportunidades.

## 4.2. Comparación de efectividad sobre muestras alteatorias

Para continuar con el análisis, generamos dos nuevos conjuntos de datos, el primero de ellos constituido por 20 casos en los que la cantidad de rivales se fue aumentando basándonos nuevamente en las potencias del número dos, es decir $n_i = 2^i$. Donde el último valor era el 524288. Decidimos establecer este límite como máximo porque para las siguientes potencias el algoritmo comienza a elevar bastante sus tiempos.

En el siguiente gráfico se puede ver el porcentaje de efectividad con el que cada algoritmo obtuvo el resultado mínimo de días totales en cada evaluación.

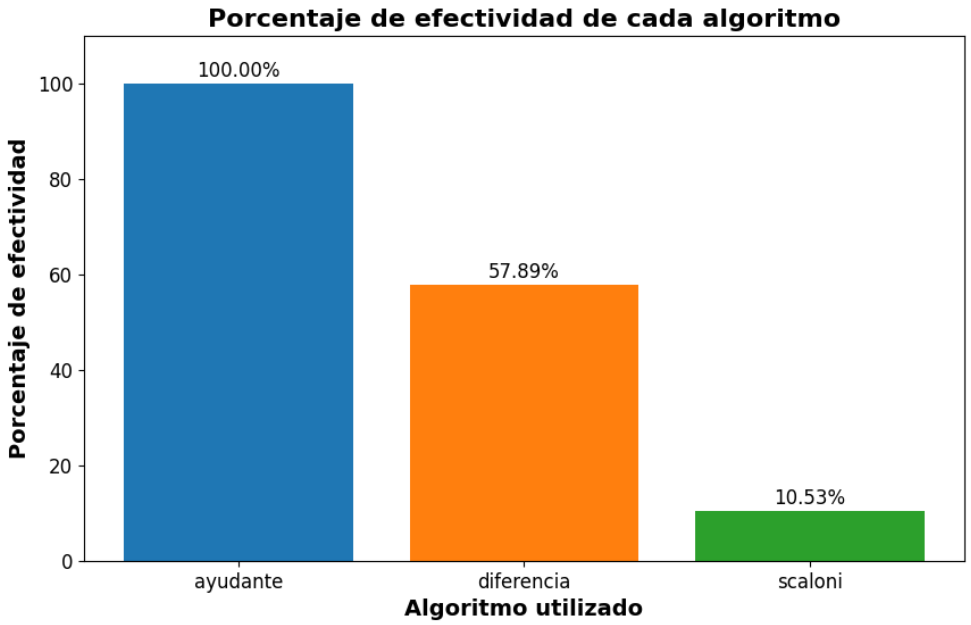



Figura 5: Comparación de efectividad de los algoritmos sobre 20 casos de tamaño $n_i = 2^i$.

En un segundo análisis decidimos mantener los tamaños de los rivales entre un número de 1 y 100, nuevamente utilizando la biblioteca random[2]. En esta prueba se generaron 1000 casos, a continuación podemos ver los resultados de efectividad:

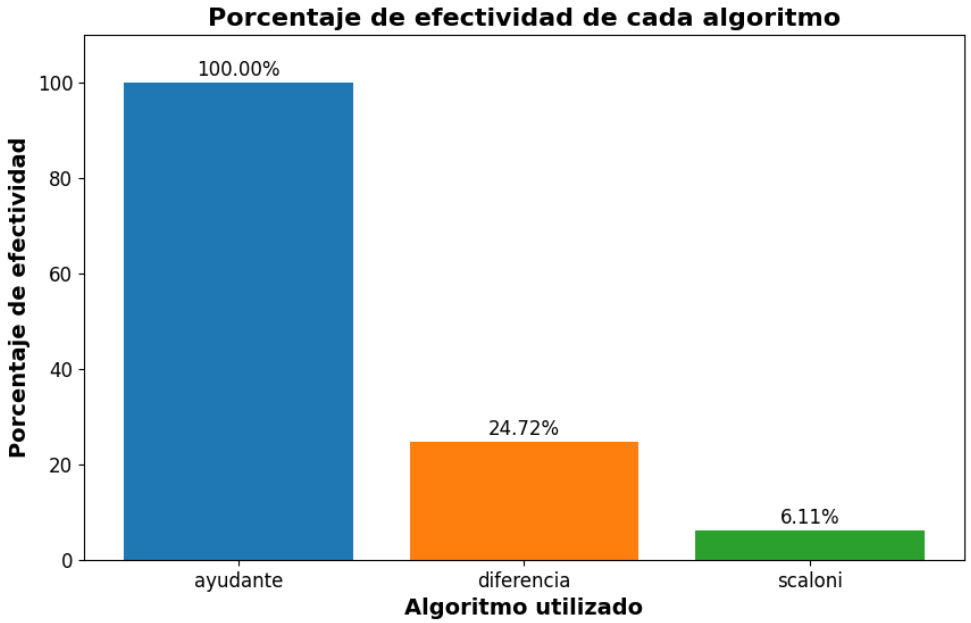



Figura 6: Comparación de efectividad de los algoritmos sobre 1000 casos con de tamaño entre 1 y 100.

# 5. Conclusiones

A través de los distintos análisis explicados en las secciones anteriores se puede evidenciar que el algoritmo óptimo para la resolución del problema es el que realiza un ordenamiento a los tiempos de los ayudantes priorizando los de mayor tamaño primero.

Este algoritmo ha probado devolver el mejor valor en todos los casos en los que ha sido evaluado, teniendo una complejidad algorítmica de $O(n.log(n))$. Cabe destacar que, en algunas ocasiones, los demás algoritmos también han encontrado la mejor solución, pero esto se debe a la particularidad de la distribución de los datos en esos casos, no permitiendo obtener el óptimo siempre.

# Referencias

[1] Python TimeComplexity. https://wiki.python.org/moin/TimeComplexity

[2] Biblioteca random de números pseudoaleatorios. https://docs.python.org/3/library/random.html