.UBAfiuba
FACULTAD DE INGENIERÍA

Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

Trabajo Práctico 1
Algoritmos Greedy

22 de enero de 2024

Ariel Aguirre
111111

Lucia Magdalena Berard
101213

Víctor Bravo
98.882

Maximiliano Prystupiuk
94.853

1. Algoritmo

1.1. Problema

Tenemos que ayudar a Scaloni a analizar los próximos n rivales de la selección campeona del mundo. Como técnico perfeccionista, quiere analizar videos de cada uno de los rivales. Recibió un compilado por cada rival, y necesita hacer un análisis muy detallado, lo cual le implica tomar apuntes, analizar tácticas, ver cuándo hay que hacerle un masaje a Messi, etc... Para que el análisis sea detallado, cada compilado no lo revisa únicamente él, sino también un ayudante.

El análisis del rival i le toma si de tiempo a Scaloni, y luego ai al ayudante (independientemente de cuál ayudante lo vea). Lo bueno, es que después de los grandes logros obtenidos, Scaloni cuenta con n ayudantes (es decir, la misma cantidad que rivales), que pueden ver los videos completamente en paralelo. Siempre los ayudantes podrán ver los videos después que Scaloni haya terminado de verlo y analizarlo como corresponde (esto no lo delega). Cuando llega la hora que un ayudante lo vea, puede ser cualquiera, pero sólo uno lo verá (no hay ganancia en que dos lo vean).

El DT necesita que los rivales estén todos con sus correspondientes análisis lo antes posible, y por eso te pide que lo ayudes. Dice que confía en vos. Sabe que no lo vas a dejar tirado.

1.2. Diseño

Presentamos el siguiente pseudocódigo para representar nuestro diseño de solución.

```
1 Ordernar las tareas i por a_i de mayor a menor y si dos son iguales desempatar por el
      s_i de menor duracion
2 Suponer un orden d_1, d_2, ... , d_n
3 Inicialmente f = 0
4 For i=1 hasta n
5     Asignar tarea i a Scaloni en el intervalo s(si) = f a f(si) = f + si
6     Asignar tarea i al Asistente en el intervalo s(ai) = f(si) a f(ai) = s(ai) + a_i
7     Let f = f(si)
8 Devolver el set de tareas [[s(si), f(si)], [s(ai), f(ai)]]  for i=1 ... n
```

1.3. Análisis

Demostraremos que el algoritmo es óptimo mediante la técnica An Exhange Argument.

Theorem 1. *Todos los cronogramas sin inversiones y sin tiempo libre tienen el mismo tiempo de análisis por Scaloni y sus asistentes*

Dado una serie de tiempos s_i y a_i, nuestro algoritmo podría no generar los mismos resultados, pero solo cambiaría el orden de aquellos que empataron en la comparación(los que tienen s_i iguales e a_i iguales entre si).

Todos los ordenes darían el mismo tiempo de análisis dado que se calcula como $F = max f(a_i)$, máximo tiempo de análisis entre los asistentes, y si una tarea viniera después de un bloque de empatados, la tarea empieza desde que terminó el análisis de Scaloni anterior con lo cual el tiempo de análisis sería la suma de los tiempos de Scaloni anteriores más su tiempo de análisis $s_i + a_i$, es decir, no le influye el orden de los empatados.

Theorem 2. *Existe un cronograma óptimo sin inversiones y sin tiempo libre*

Demostración. Primero haremos unas aclaraciones

1. *Si O tiene una inversión, entonces hay un par de tareas m y n tal que $d_n < d_m$ (nuestro algoritmo elegiría primero a n que a m), pero en el óptimo fue programado al revés con m*

primero que n

Si $d_n < d_m$, entonces tenemos dos opciones o $a_n > a_m$ o $a_n = a_m$ y $s_n < s_m$.

2. *Después del intercambio entre n y m tenemos un cronograma con un intercambio menos*

   Si tenemos una inversión una al lado de la otra y los intercambiamos, no aumentamos el numero de inversiones

3. *El nuevo cronograma tras el intercambio no aumenta el máximo demorado por Scaloni y sus Asistentes*

   Lo vemos en mayor detalle más abajo. Si lo llevamos al caso general, podemos notar que en el peor de los casos tendríamos que invertir todos los elementos de O, pero sabiendo que no empeora el máximo, entonces nuestra solución no es peor que la del óptimo.

□

Theorem 3. *El nuevo cronograma tras el intercambio no aumenta el tiempo máximo de análisis por Scaloni y sus asistentes*

Primero tomaremos algunas pautas de notación para nuestros cronogramas.

Sean m y n dos tareas adyacentes cualquiera en el cronograma O.

Sea $f(a_i)$ el tiempo de finalización de la revisión de la tarea i por parte del asistente i

Cuando escribamos a_i o s_i entenderemos que se refiere a la duración de la revisión de cada tarea.

Idem anterior para $f(s_i)$, pero para los tiempo de revisión de la tarea i vista por Scaloni.

Sea $F = \text{máx}\, f(a_i)$, donde F es el tiempo de análisis del cronograma entendido como el máximo entre las duraciones finales de los asistentes.

Sea k el punto de inicio entre las tareas m y n.

Con la notación dada, luego veremos que realizar un intercambio no aumenta el máximo demorado.

Caso 1: m y n invertidos con $a_n > a_m$

Esto significa que nuestro algoritmo hubiera elegido a n antes que m porque el tiempo que le lleva a su asistente es menor, pero por inversión se eligió al revés como se ve en la siguiente imagen.

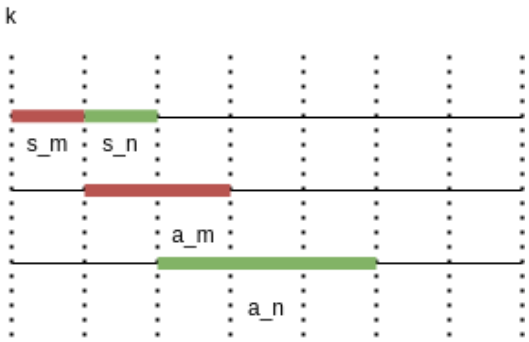


Figura 1: Caso 1 invertido

Si hacemos unas cuentas, notamos que $F = \text{máx}\{f(a_m), f(a_n)\}$, donde

$$f(a_n) = k + s_m + s_n + a_n$$
$$f(a_m) = k + s_m + a_m$$

Y dado $a_n > a_m$ y despejando tenemos que $f(a_n) > f(a_m)$ y por lo tanto $F = f(a_n)$.
Si aplicamos un intercambio, entonces nos queda

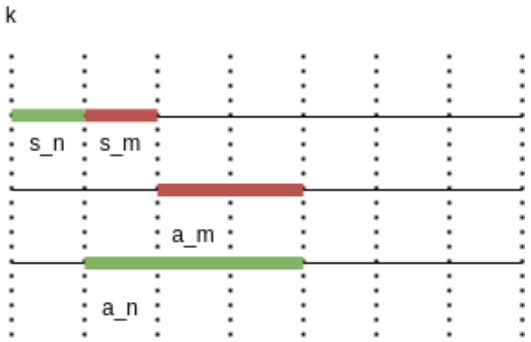


Figura 2: Caso 1 después del intercambio

Donde $\overline{F} = \text{máx}\{\overline{f(a_m)}, \overline{f(a_n)}\}$ y luego

$$\overline{f(a_n)} = k + s_n + a_n$$
$$\overline{f(a_m)} = k + s_n + s_m + a_m$$

Hasta acá no podemos saber aún cual es el máximo dado que si los comparamos

$$\overline{f(a_n)} =? \overline{f(a_m)}$$
$$k + s_n + a_n =? k + s_n + s_m + a_m$$
$$a_n =? s_m + a_m$$

Notamos que depende de la relación entre a_n y $s_m + a_m$. Para facilitarnos los cálculos, si comparamos directamente entre F y $\overline{F}$ notamos que

$$F = f(a_n) = k + s_m + s_n + a_n$$
$$\overline{F} = \text{máx}\{\overline{f(a_m)}, \overline{f(a_n)}\}$$

Y vemos que $\overline{F}$ es menor en cualquier caso a F dado que $\overline{f(a_n)}$ y $\overline{f(a_m)}$ son menores que $f(a_n)$ visto en las siguientes ecuaciones

$$f(a_n) > \overline{f(a_n)}$$
$$k + s_m + s_n + a_n > k + s_n + a_n$$
$$s_m > 0$$

$$f(a_n) > \overline{f(a_m)}$$
$$k + s_m + s_n + a_n > k + s_m + s_n + a_m$$
$$a_n > a_m \text{ Por enunciado de inversión}$$

Por lo tanto en el caso 1 no aumenta el tiempo máximo.

Caso 2: m y n invertidos con $a_n = a_m$, pero $s_n < s_m$

Con esto nos referimos a dos tareas m y n, donde nuestro algoritmo hubiera elegido a n antes que m dado que si bien ambos tienen mismo tiempo de asistente $a_n = a_m$, el tiempo de Scaloni es menor en n ($s_n < s_m$), pero por inversión se tomó al revés.

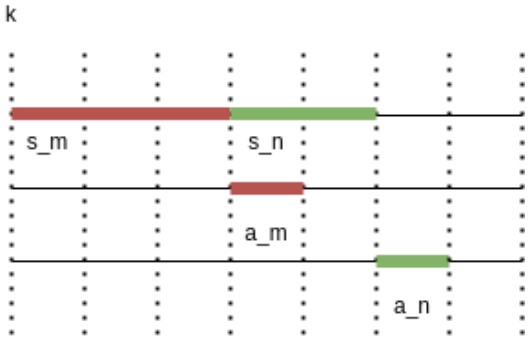


Figura 3: Caso 2 Invertido

Nuevamente notamos que el máximo es $F = f(a_n)$

$$f(a_n) = k + s_m + s_n + a_n$$
$$f(a_m) = k + s_m + a_m$$

Tras invertir tenemos

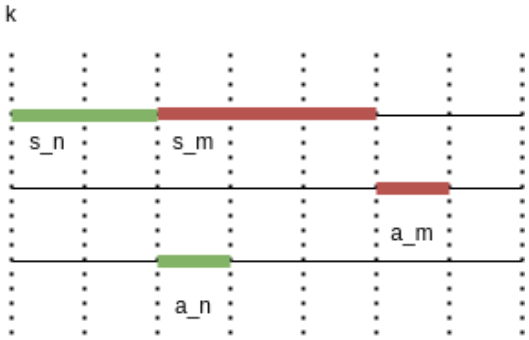


Figura 4: Caso 2 Intercambio

$$\overline{F} = \text{máx}\{\overline{f(a_n)}, \overline{f(a_m)}\}$$
$$\overline{f(a_n)} = k + s_n + a_n$$
$$\overline{f(a_m)} = k + s_n + s_m + a_m$$

Como $a_n = a_m$ entonces $\overline{F} = \overline{f(a_m)}$

Luego, si comparamos F y $\overline{F}$ tenemos que son iguales dado que $a_n = a_m$ por enunciado

$$F = \overline{F}$$
$$f(a_n) = \overline{f(a_m)}$$
$$k + s_m + s_n + a_n = k + s_m + s_n + a_m$$

Por lo tanto, no empeora el máximo.

En conclusión, en ambos casos vemos que después de hacer un intercambio no empeora el máximo entre dos tareas.

Theorem 4. *El cronograma A producido por el algoritmo greedy tiene un tiempo de análisis* $F = max f(a_i)$

Anteriormente demostramos que existe un algoritmo óptimo sin inversiones y como todos los algoritmos sin inversiones tienen el mismo tiempo de análisis, entonces el cronograma obtenido por el algoritmo greedy es óptimo

1.4. Análisis de Complejidad

```python
def ordenamiento(x, y):
    if x[1] > y[1]:
        return -1
    elif x[1] < y[1]:
        return 1
    else:
        if x[0] < y[0]:
            return -1
        elif x[0] > y[0]:
            return 1
        else:
            return 0

def algoritmo(data):
    """data es una lista de tuplas (s, a) donde s es el tiempo de Scaloni y a es el
     tiempo de los asistentes. Cada elemento est  ordenado como se lee en el
    archivo de origen.
    Devuelve una lista donde el primer elemento representa el intervalo de Scaloni
    y el segundo el del asistente.
    """
    data = sorted(data, key=functools.cmp_to_key(ordenamiento))
    f = 0
    resultado = []
    for s, a in data:
        s_si = f
        f_si = f + s
        s_ai = f_si
        f_ai = s_ai + a
        resultado.append(((s_si, f_si), (s_ai, f_ai)))
        f = f_si
    return resultado
```

Para el análisis de complejidad podemos notar que el ordenamiento es $O(n \log n)$ dado que Python internamente usa TimSort [1]

En las siguientes lineas se recorre la lista ordenada y cada elemento se inserta en $O(1)$ en una lista de tuplas con los tiempos de inicio y termino de cada tarea de Scaloni lo cual en total es $O(n)$.

[1] https://wiki.python.org/moin/HowTo/Sorting.

Es decir, el algoritmo es $O(n \log n)$.

Por último, los valores de a_i y s_i no tienen impacto en los tiempo ni optimalidad del algoritmo, es decir, no hay una relación directa entre cambios en ellos y el tiempo de ejecución del algoritmo, salvo casos particulares, como tener los datos ya ordenados, lo que haría que que el algoritmo fuese $O(N)$

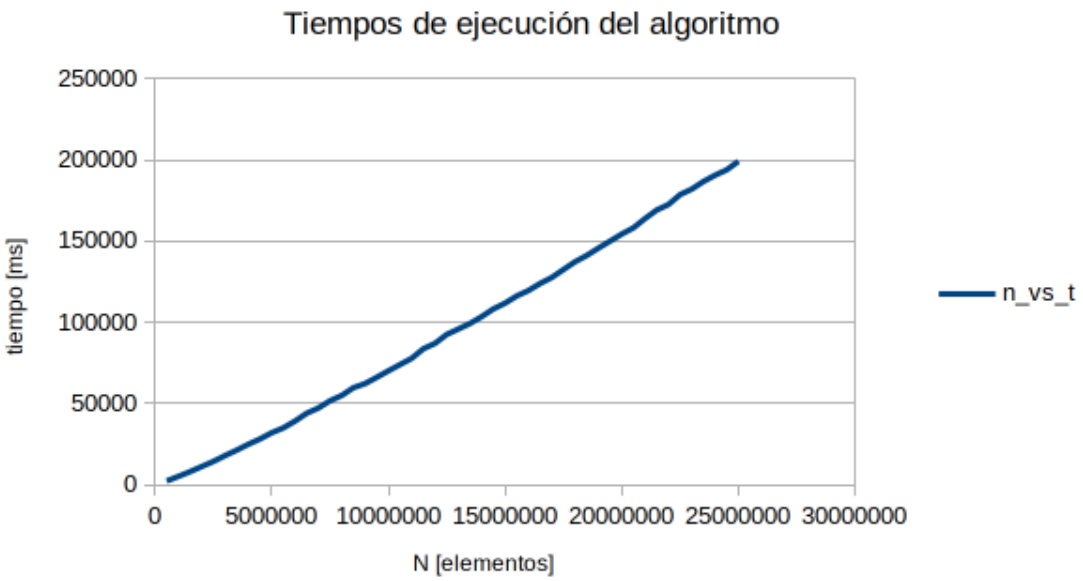
2. Mediciones

Se realizaron mediciones en base a crear arreglos de diferentes longitudes, yendo de 500.000 en 500.000 elementos, hasta un máximo de 25.000.000, donde los elementos en cada caso fueron generados por los valores pseudoaleatorios del lenguaje (el módulo `random`). El algoritmo generador de arreglos para estas mediciones se ejecutó con una semilla de 0 (elegida arbitrariamente).

Para realizar una generación y ejecución de los mismos valores usados para generar el gráfico, ejecutar:

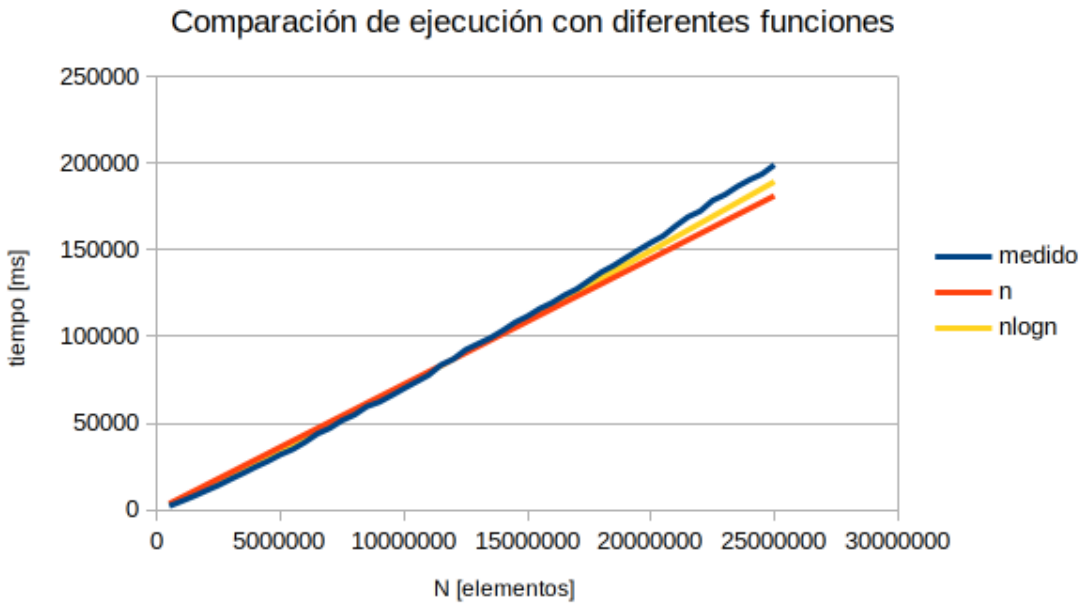
```
./graficador.sh 0
```

En nuestro caso, deshabilitamos la impresión por pantalla de algoritmo.py



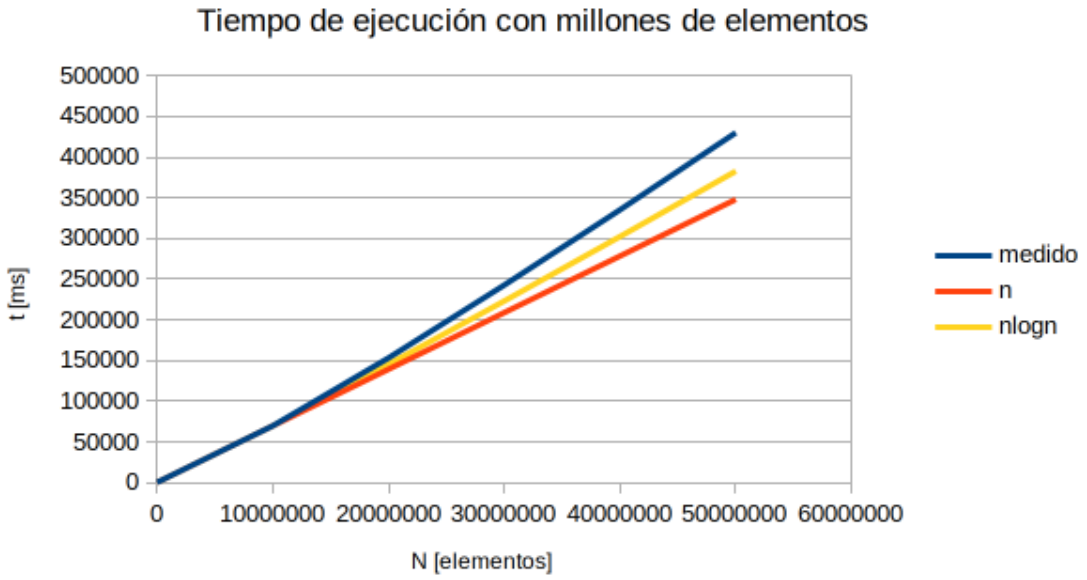
Puede apreciarse una tendencia lineal, que nosotros creemos, por el análisis previo, ajusta mejor con una función del tipo $Y(N) = K_a * N * Log(N)$

Si la complejidad temporal fuese lineal se comportaría como: $Y(N) = K_b * N$ Despejamos los valores de K para el mismo elemento (N= 1.200.000) y comparamos con los resultados obtenidos:

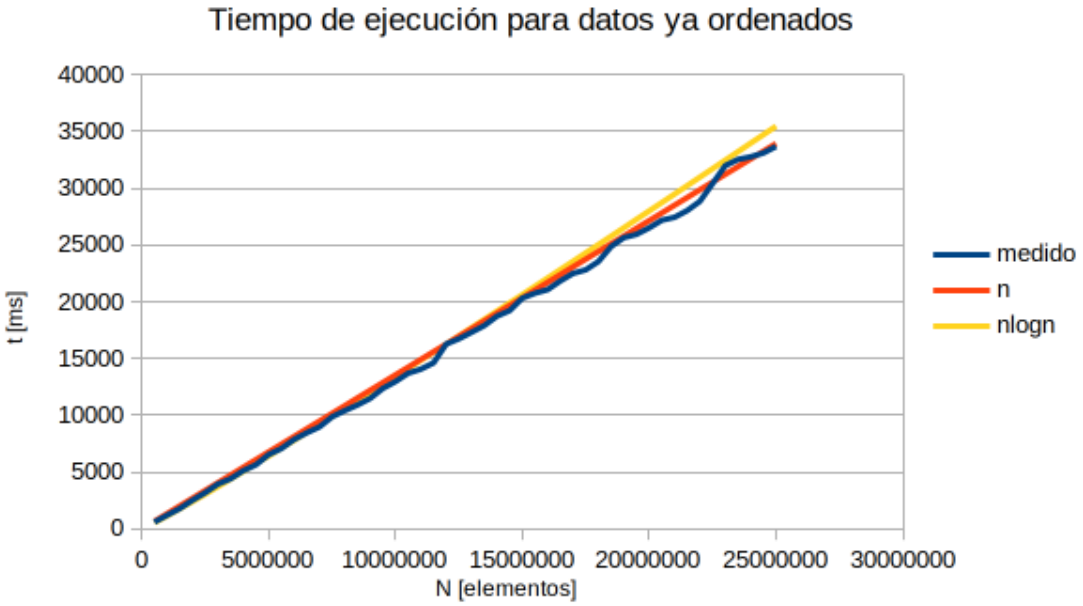


Puede notarse que las mediciones se ajustan mas a $N * Log(N)$

Realizamos también una ejecución con saltos de 10.000.000 de elementos, hasta un máximo de 100.000.000, pero a los 60.000.000 nos encontramos con limitaciones de hardware (se agotaba la memoria, y después de un tiempo el proceso algoritmo.py era terminado por el sistema operativo), por lo que la última medición confiable fue con N= 50.000.000. El punto de ajuste para los K_i en este caso se tomo en 10.000.000.



Por último, realizamos una ejecución con los datos de entrada ya ordenados, y como se predijo en el análisis de complejidad, este caso se acerca más a $Y(N) = K_d * N$, es decir, una función lineal.



Tomando él último valor calculado en ambos casos, esto se hace mas patente, habiendo tomado para un problema de 25.000.000 de elementos 199 segundos con los datos desordenados, en comparación a los 33,7 segundos del caso ordenado.

3. Conclusiones

Nuestra conclusión es que el algoritmo aquí propuesto es del tipo greedy, siempre encuentra la solución óptima, y su complejidad temporal es $O(N * log(N))$, sin depender de los valores de a_i ni s_i. Sí los datos están ordenados de manera descendente, la complejidad se vuelve $O(N)$

Referencias

[1] Time Complexity, accedido 18 Enero 2024, `https://wiki.python.org/moin/TimeComplexity`