.UBAfiuba
FACULTAD DE INGENIERÍA

Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

Trabajo Práctico 1
Algoritmos Greedy

18 de septiembre de 2023

Juan Ignacio Díaz
103488

Juan Manuel Díaz
108183

Juan Martín Muñoz
106699

1. Introducción

En el presente trabajo práctico se realiza un analisis de un algoritmo *Greedy* para resolver un problema de minimización.

2. Problema

Dada una lista de entrenamientos E, siendo que cada entrenamiento e esta compuesto por dos análisis necesarios, el de Scaloni s y el de uno de sus asistentes a.

Todo entrenamiento debe primero ser completado por Scaloni (s) para que luego sea posible realizar el analisis de un asistente (a).El analisis de Scaloni s es un recurso compartido entre todos los entrenamientos (E), pero los analisis de asistentes son independientes entre si (cada entrenamiento tiene su propio asistente dedicado).

Por ende se afirma que el tiempo de finalización de cada entrenamiento (e) depende del tiempo que le haya tomado a Scaloni analizar los entrenamiento anteriores a este, sin importar lo requerido por cada asistente.

El tiempo requerido para analizar todos los entrenamientos entonces es el máximo empleado por alguno de los entrenaientos. Si consideramos a la lista de entrenamientos E como una lista de tuplas (s, a), con s y a positivos, nuestro objetivo es ordenar la lista de manera de minimizar:

$$\mathcal{M} = \max_{0 \leq k < n} (\mathcal{S}_k + a_k) \tag{1}$$

$$\text{con } \mathcal{S}_k := \sum_{i=0}^{k} s_i$$

Resolución

2.1. Algoritmo

Para minimizar $\mathcal{M}$, proponemos un algoritmo *Greedy* que siempre toma el elemento de mayor a_i. Esto es, ordenando de mayor a menor ignorando los valores de s_i.

2.1.1. Optimalidad

Sabemos que al intercambiar dos elementos, solo es necesario ver que pasa con $\mathcal{S}_i + a_i$ y $\mathcal{S}_{i+1} + a_{i+1}$, ya que $\mathcal{S}_k + a_k$ es independiente del orden en el que están los elementos anteriores, o los que le siguen. Consideremos que pasa al intercambiar dos elementos adyacentes d_i y d_{i+1}:

$$A := \mathcal{S}_i + a_i$$
$$B := \mathcal{S}_i + s_{i+1} + a_{i+1}$$

Al intercambiarlos, obtendríamos:

$$A' := \mathcal{S}_i + s_{i+1} + a_i$$
$$B' := \mathcal{S}_i + a_{i+1}$$

- $a_i < a_{i+1} \implies B > A' \wedge B > B'$. Intercambiamos, $\mathcal{M}' \leq \mathcal{M}$.
- $a_i = a_{i+1} \implies B = A' \wedge A = B'$. Intercambiemos o no, $\mathcal{M}' = \mathcal{M}$.
- $a_i > a_{i+1} \implies A < B' \wedge B < B'$. No intercambiamos. $\mathcal{M}' = \mathcal{M}$.

Con esto en mente, si partimos de una configuración óptima, podemos realizar intercambios hasta ordenar los elementos según nuestro criterio sin empeorar el valor de $\mathcal{M}$, lo que demuestra que nuestro criterio es tan bueno como el óptimo.

2.2. Implementación

```python
from collections import namedtuple
import heapq

Demora = namedtuple("Demora", ['s', 'a'])
# overload comparison operators
Demora.__eq__ = lambda self, other: self.a == other.a
Demora.__ne__ = lambda self, other: self.a != other.a
Demora.__le__ = lambda self, other: self.a >= other.a
Demora.__ge__ = lambda self, other: self.a <= other.a
Demora.__lt__ = lambda self, other: self.a > other.a
Demora.__gt__ = lambda self, other: self.a < other.a

def ordenar(datos: list[Demora]):
    """ Requiere que (x < y) <=> (x.a > y.a) """
    heapq.heapify(datos)
    return [heapq.heappop(datos) for _ in range(len(datos))]
```

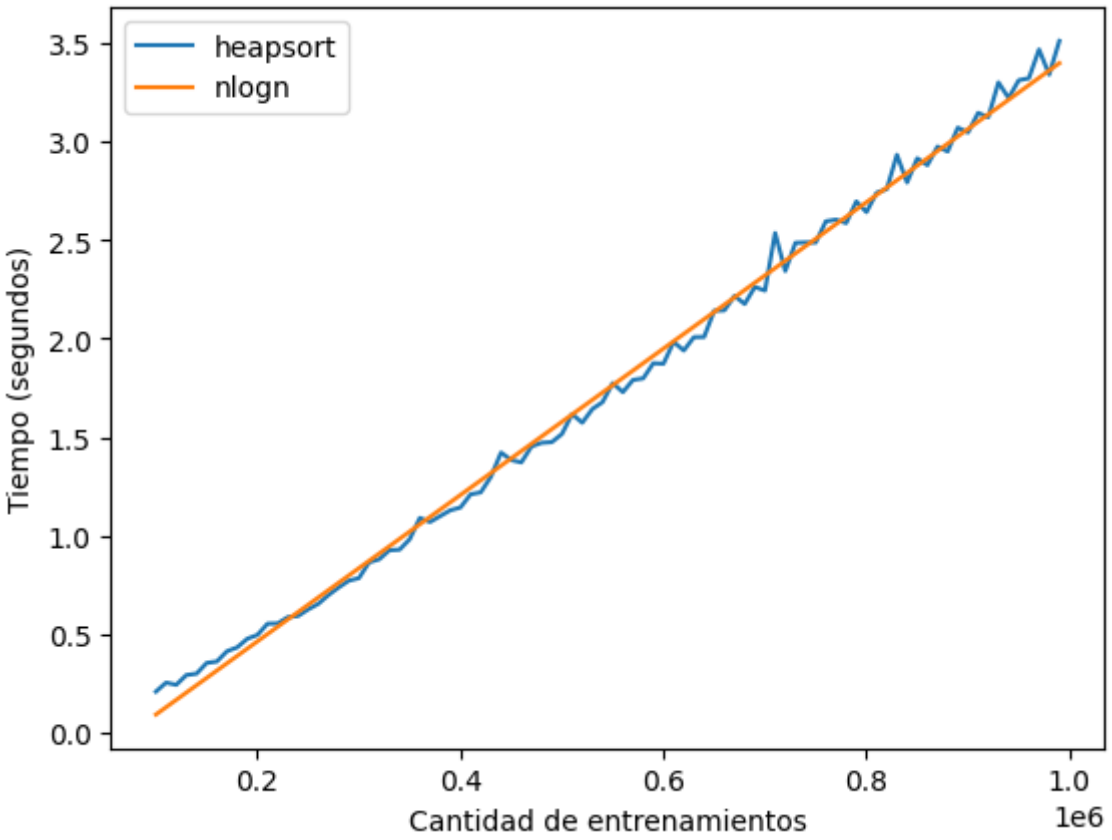
La ecuación de recurrencia que corresponde a este algoritmo es:

$$\mathcal{T}(n) = \mathcal{T}(n-1) + \mathcal{O}(\log n)$$

Esto es porque la operación `heapify` es lineal, y `heappop` es logarítmica, al iterar por todos los elementos, obtenemos un ordenamiento $\mathcal{O}(n \log n)$.

3. Mediciones

Se realizaron mediciones en base a crear arreglos de diferentes largos, yendo de 100 en 100 elementos, donde los elementos en cada caso fueron generados por los valores pseudoaleatorios del lenguaje (el módulo `random`).



Como se puede apreciar, el algoritmo tiende a $\mathcal{O}(n \log n)$.

4. Conclusiones

La solución propuesta es de simple interpretación e implementación, manteniendo una buena complejidad algorítmica. La mayor dificultad estuvo en demostrar que efectivamente resuleve el problema en cuestión de manera óptima.

Cabe aclarar que si bien en este caso encontramso la solución óptima con un algoritmo *Greedy*, esto no siempre es posible.