.UBAfiuba
FACULTAD DE INGENIERÍA

Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

Trabajo Práctico 1
La mafia de los algortimos greedy

3 de abril de 2025

Pablo Gaston Choconi	Ezequiel Matias Gacitua	Abel Tomas Modesti Funes
106388	112234	111932

1. Análisis del problema

1.1. El problema

El problema el cual vamos a resolver en este trabajo es el siguiente:

> Trabajamos para la mafia de los amigos Amarilla Pérez y el Gringo Hinz. En estos momentos hay un problema: alguien les está robando dinero. No saben bien cómo, no saben exactamente cuándo, y por supuesto que no saben quién. Evidentemente quien lo está haciendo es muy hábil (probablemente haya aprendido de sus mentores).
>
> La única información con la que contamos son n transacciones sospechosas, de las que tenemos un timestamp aproximado. Es decir, tenemos n tiempos t_i, con un posible error e_i. Por lo tanto, sabemos que dichas transacciones fueron realizadas en el intervalo $[t_i - e_i; t_i + e_i]$.
>
> Por medio de métodos de los cuales es mejor no estar al tanto, un interrogado dio el nombre de alguien que podría ser la rata. El Gringo nos pidió revisar las transacciones realizadas por dicha persona... en efecto, eran n transacciones. Pero falta saber si, en efecto, condicen con los timestamps aproximados que habíamos obtenido previamente.
>
> El Gringo nos dio la orden de implementar un algoritmo que determine si, en efecto, las transacciones coinciden. Amarilla Perez nos sugirió que nos apuremos, si es que no queremos ser nosotros los siguientes sospechosos.

- Hacer un análisis del problema, y proponer un algoritmo greedy que obtenga la solución al problema planteado: Dados los n valores de los timestamps aproximados t_i y sus correspondientes errores e_i, así como los timestamps de las n operaciones s_i del sospechoso (pueden asumir que estos últimos vienen ordenados), indicar si el sospechoso es en efecto la rata y, si lo es, mostrar cuál timestamp coincide con cuál timestamp aproximado y error. Es importante notar que los intervalos de los timestamps aproximados pueden solaparse parcial o totalmente.
- Demostrar que el algoritmo planteado determina correctamente siempre si los timestamps del sospechoso corresponden a los intervalos sospechosos, o no. Es decir, si condicen, que encuentra la asignación, y si no condicen, que el algoritmo detecta esto, en todos los casos.
- Escribir el algoritmo planteado. Describir y justificar la complejidad de dicho algoritmo. Analizar si (y cómo) afecta la variabilidad de los valores de los diferentes valores a los tiempos del algoritmo planteado.
- Realizar ejemplos de ejecución para encontrar soluciones y corroborar lo encontrado. Adicionalmente, el curso proveerá con algunos casos particulares para que puedan usar de prueba.
- Hacer mediciones de tiempos para corroborar la complejidad teórica indicada. Esto, por supuesto, implica que deben generar sus sets de datos. Agregar los casos de prueba necesarios para dicha corroboración. Esta corroboración empírica debe realizarse confeccionando gráficos correspondientes, y utilizando la técnica de cuadrados mínimos.
- Agregar cualquier conclusión que les parezca relevante.

1.2. Solución propuesta

COMPLETAR...

AQUI DEBERIA ESTAR EL ANALISIS DEL PROBLEMA Y LA JUSTIFICACIÓN DE PORQUE FUNCIONA

2. Algoritmo para asignar operaciones a intervalos

2.1. Código

A continuación se muestra el código de la solución greedy del problema.

```
from timestampSort import sortIntervalsByEnd as sortIntervals

def moleFinder(intervals: list, operations: list, result_array: list = None):
    """
    Funcion Principal.\n
    Dado intervalos y operaciones, indica si cada operacion tiene un intervalo en el que encaja.\n
    Un intervalo no puede contener multiples operaciones.\n
    PARAMETER intervals:    Lista de Diccionarios, dado por fileReader().\n
    PARAMETER operations:   Lista de Integers, dado por fileReader().\n
    PARAMETER result_array: Lista vacia, Opcional. Se guarda en que intervalo encaja cada operacion.\n
    RETURNS: Boolean. Si se pudieron encajar todas las operaciones con un intervalo o no.\n
    """
    sortedIntervals = sortIntervals(intervals)

    for operation in operations:
        intervalContainingOperationThatEndsSooner = None
        for interval in sortedIntervals:
            if (operation >= interval["startTime"]) and (operation <= interval["endTime"]) and (interval["op"] is None):
                if intervalContainingOperationThatEndsSooner is None or interval["endTime"] < intervalContainingOperationThatEndsSooner["endTime"]:
                    # Se guarda el timestamp que contiene la operacion, termina antes y no tenia ninguna operacion asignada
                    intervalContainingOperationThatEndsSooner = interval
        if intervalContainingOperationThatEndsSooner is None:
            # Ningun timestamp contenia a la operacion. No es la rata
            return False
        else:
            intervalContainingOperationThatEndsSooner["op"] = operation
            if (result_array is not None):
                result_array.append(intervalContainingOperationThatEndsSooner)
    return True
```

2.2. Análisis de complejidad

La complejidad del algoritmo planteado posee dos partes identificables, el ordenamiento mediante MergeSort de los timestamp por tiempo de fin y la asignacion de las transacciones a un timestamp

Primero, hagamos un rapido analisis del ordenamiento MergeSort, este algoritmo divide el arreglo en subarreglos de la mitad de tamaño hasta llegar a un unico elemento, el cual se considera ordenado, luego combina los mismos de manera ordenada hasta llegar a nuestro arrreglo original ordenado en su totalidad. Si hablamos solo de temas de complejidad esta parte de nuestra respuesta tiene $\mathcal{O}(n \log n)$.

Por otro lado tenemos la parte principal del algoritmo, lo que busca es asignar cada transacción a un timestamp diferente. Consigue esto tomando y verificando dentro de que timestamps se encuentra y asociándolo al que termina antes, y esto se realiza por cada transacción. Por lo tanto podemos deducir que esta parte del algoritmo es de $\mathcal{O}(n^2)$.

Si vemos el algoritmo en su totalidad tiene $\mathcal{O}(n \log n) + \mathcal{O}(n^2)$, pero cuando nos vamos a n grandes, que son los casos que nos interesan para el análisis de la complejidad observamos que el algoritmo es de $\mathcal{O}(n^2)$

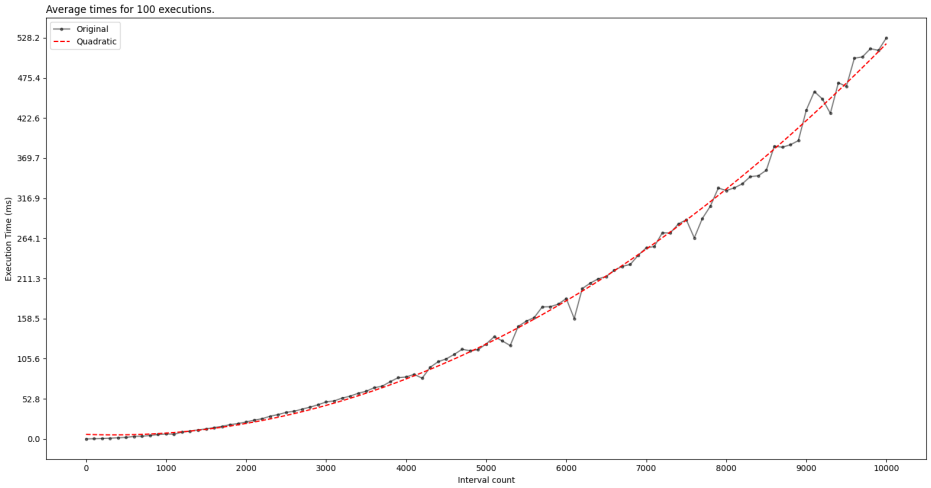
Como se va a ver en las mediciones, esto no es exacto sino que es un aproximado en los peores casos

(Lo dejo por si agregamos una ecuacion, asi esta el ejemplo)

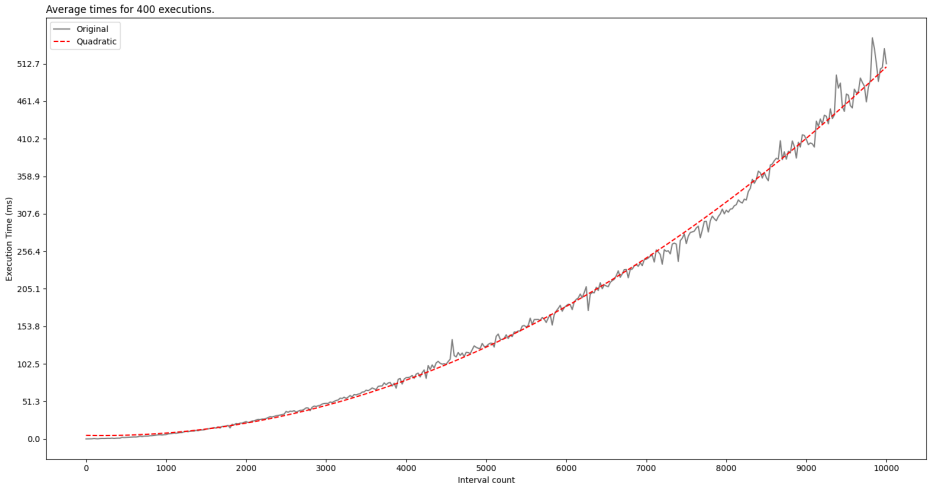
$$\mathcal{T}(n) = 2\mathcal{T}\left(\frac{n}{2}\right) + \mathcal{O}(n)$$

3. Mediciones

Se realizaron medicions 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`).



Este gráfico muestra el tiempo de ejecución promedio de 100 ejecuciones del programa (en milisegundos) dada una cantidad de intervalos con un acomodo cuadrático



Este gráfico en cambio muestra el tiempo de ejecución pero del promedio de 400 ejecuciones del programa (en milisegundos)

Como se puede apreciar, el gráfico muestra una tendencia cuadrática como lo explicado en el análisis de complejidad

4. Conclusiones

Acá irían las conclusiones de todo nuestro trabajo :)