.UBAfiuba
FACULTAD DE INGENIERÍA

Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

# Trabajo Práctico 1
# La mafia de los algortimos greedy

2 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.

```
1
2
3  def merge_sort(arr):
4      # Si la lista tiene un solo elemento o est  vac a , no es necesario dividirla
5      if len(arr) <= 1:
6          return arr
7
8      # Dividir la lista en dos mitades
9      mid = len(arr) // 2
10     left_half = arr[:mid]
11     right_half = arr[mid:]
12
13     # Recursivamente ordenar ambas mitades
14     left_half = merge_sort(left_half)
15     right_half = merge_sort(right_half)
16
17     # Fusionar las dos mitades ordenadas
18     return merge(left_half, right_half)
19
20
21 def merge(left, right):
22     sorted_arr = []
23     i = j = 0
24
25     # Fusionar ambas listas mientras haya elementos en ambas
26     while i < len(left) and j < len(right):
27         if left[i]["endTime"] < right[j]["endTime"]:  # Cambiar el signo de la
       comparaci n aqu
28             sorted_arr.append(left[i])
29             i += 1
30         else:
31             sorted_arr.append(right[j])
32             j += 1
33
34     # Si quedan elementos en 'left', agregarlos
35     while i < len(left):
36         sorted_arr.append(left[i])
37         i += 1
38
39     # Si quedan elementos en 'right', agregarlos
40     while j < len(right):
41         sorted_arr.append(right[j])
42         j += 1
43
44     return sorted_arr
45
46 # Ejemplo de uso:
47
48
49 def moleFinder(timestamps, operations, result_array):
50     sorted_timestamps = merge_sort(timestamps)
51
52     for operation in operations:
53         timestampContainingOperationThatEndsSooner = None
54         for timestamp in sorted_timestamps:
55             if operation >= timestamp["startTime"] and operation <= timestamp["
       endTime"] and timestamp["operation"] is None:
56                 if timestampContainingOperationThatEndsSooner is None or timestamp[
       "endTime"] < timestampContainingOperationThatEndsSooner["endTime"]:
57                     # Se guarda el timestamp que contiene la operacion, termina
       antes y no tenia ninguna operaci n asignada
58                     timestampContainingOperationThatEndsSooner = timestamp
59         if timestampContainingOperationThatEndsSooner is None:
60             # Ningun timestamp contenia a la operacion. No es la rata
```

```
            return False
        else:
            timestampContainingOperationThatEndsSooner["operation"] = operation

    isTheRat = True
    for timestamp in sorted_timestamps:
        result_array.append(timestamp)
        if timestamp["operation"] is None:
            isTheRat = False

    return isTheRat


def printResults(timestamps, isTheRat, testName, duration, verbose):
    print("================================")
    print("Nombre de prueba: ", testName, "")
    print("---------------")
    print("N mero total de timestamps: ", len(timestamps))
    print("Tiempo total de ejecuci n: ", round(duration * 1000, 10), "
    milisegundos")
    print("---------------")
    if isTheRat:
        print("Resultado: Es la rata!!!")
    else:
        print("Resultado: NO es la rata!!!")

    if verbose and len(timestamps) > 0:
        print("---------------")
        print("Asignaciones: \n")
        for timestamp in timestamps:
            print(
                timestamp["operation"],
                " --> ",
                timestamp["time"],
                "    ",
                timestamp["error"]
            )
    print("\n")


if __name__ == "__main__":
    verbose = True      # Especifica si se quiere imprimir en la salida todas las
    asignaciones

    testNames = [
        "5-es.txt",
        "5-no-es.txt",
        "10-es.txt",
        "10-es-bis.txt",
        "10-no-es.txt",
        "10-no-es-bis.txt",
        "50-es.txt",
        "50-no-es.txt",
        "100-es.txt",
        "100-no-es.txt",
        "500-es.txt",
        "500-no-es.txt",
        "1000-es.txt",
        "1000-no-es.txt",
        "5000-es.txt",
        "5000-no-es.txt"
    ]

    for testName in testNames:
        timestamps, operations = fileReader("Tests/" + testName)

        result_array = []

        start_time = time.time()
        result = moleFinder(timestamps, operations, result_array)
```

```
129         totalTime = time.time() - start_time
130
131         printResults(result_array, result, testName, totalTime, verbose)
```

### 2.2. Análisis de complejidad

COMPLETAR... (Lo que esta abajo es todo de ejemplo)

La ecuación de recurrencia que corresponde a este algoritmo es:

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

Esto es porque tenemos 2 llamados recursivos, cada lado con la mitad del problema, y al partir hacemos una slice, lo cual en Python realiza una copia, por lo cual demora tiempo lineal en aplicarse en cada caso.

Aplicando el teorema maestro, la complejidad nos queda en $\mathcal{O}(n \log n)$. En este caso, nos quedó peor complejidad que en el caso iterativo pura y exclusivamente por abusar del lenguaje de programación sin tomar en cuenta el tiempo que consume hacer un slice. Dejando de hacer esto, podemos mostrar la siguiente versión del algoritmo:

En este caso, la ecuación de recurrencia es:

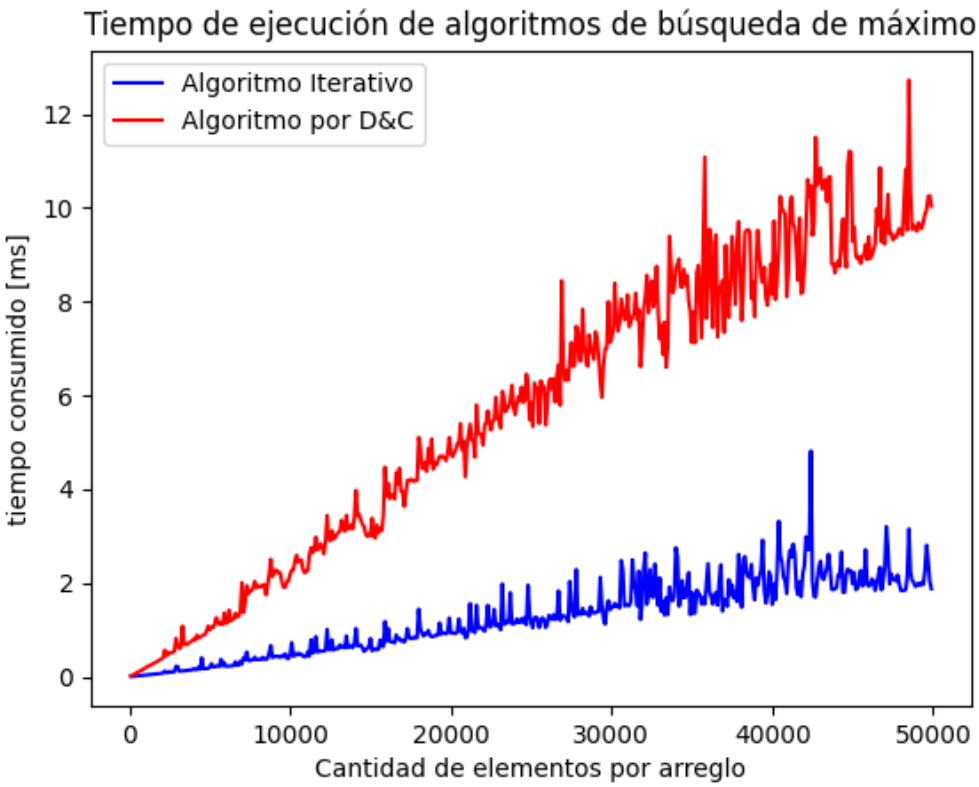
$$\mathcal{T}(n) = 2\mathcal{T}\left(\frac{n}{2}\right) + \mathcal{O}(1)$$

Aplicando el teorema maestro, nos queda que la complejidad es $\mathcal{O}(n)$.

Es importante notar que en otros lenguajes de programación esto podría no ser necesario (por ejemplo, Go). En algunos lenguajes se puede operar usando Slices que consuman $\mathcal{O}(1)$ de tiempo (a cambio de utilizar la misma memoria que el arreglo original), o bien usando aritmética de punteros como puede ser el caso de C. Independientemente del caso, es importante notar que el algoritmo de división y conquista es lógicamente igual (o extremadamente similar), pero tenemos que considerar cuestiones de implementación del lenguaje elegido a la hora de definir las complejidades.

## 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`).




Como se puede apreciar, ambos algoritmos tienen una tendencia efectivamente lineal en función del tamaño de la entrada, si bien el algoritmo iterativo es más veloz en cuestión de constantes.

## 4. Conclusiones

Acá irían las conclusiones de todo nuestro trabajo :)