
Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

# Trabajo Práctico 1
# La mafia de los algortimos greedy

10 de abril de 2025

Julen Leonel Gaumard
111379

Noelia Salvatierra
100116

Franco Macke
105974

# 1. Análisis del problema

El enunciado presenta dos listas de longitud $n$:

- Una lista de transacciones aproximadas, representadas como pares $(t_i, e_i)$, que son intervalos de tiempo en los que pudo haberse producido una transacción: $[t_i - e_i, t_i + e_i]$.
- Una lista de timestamps exactos $s_1, s_2, \ldots, s_n$, correspondientes a las transacciones realizadas por un sospechoso. Esta lista se encuentra ordenada de forma creciente.

El problema a resolver es comprobar si hay una asignación uno a uno entre los intervalos aproximados de los timestamps y los timestamps exactos $s_j$, donde cada $s_j$ estaría dentro del intervalo $[t_i - e_i, t_i + e_i]$. Cada intervalo puede ser utilizado solo una vez.

## 1.1. Posibles enfoques

Estas fueron alguna de las consideraciones que barajamos al resolver el problema:

- **Encontrar el timestamp más cercano:** Una opción es asignar cada timestamp exacto al intervalo más próximo con respecto a la distancia al centro del intervalo $t_i$, o sea, buscar minimizar $|s_j - t_i|$. Esta estrategia busca una correspondencia "natural" en el sentido de máxima similitud temporal.
- **Maximizar la libertad futura:** Asignar cada timestamp $s_j$ al intervalo válido cuyo extremo izquierdo sea el menor posible, es decir, priorizar intervalos que empiecen antes.
- **Minimizar el error por derecha:** La estrategia final usada fue asignar cada timestamp $s_j$ al primer intervalo posible, priorizando que $s_j$ esté lo más cerca posible del extremo derecho del intervalo. Esta estrategia busca minimizar la diferencia entre $s_j$ y el final del intervalo válido $t_i + e_i$.

## 1.2. Algo más a considerar

Dado que los timestamps ordenados están de menor a mayor, una técnica eficiente puede iterarlos secuencialmente desde menor hasta mayor y tomar decisiones locales óptimas que tiendan hacia una solución global válida.

Si en algún momento no hay ningún intervalo disponible para un timestamp, se puede concluir inmediatamente que la asignación no es posible.

# 2. Demostración

Cuando analizamos el algoritmo, barajamos varias posibilidades de cómo encarar el problema:

- **Encontrar el timestamp más cercano:** Minimizar la distancia entre $t$ y $s$. Es fácil descartar este caso con un contraejemplo simple. Supongamos que tenemos:

  $$t = [3, 4, 6], \quad s = [(3, 1), (10, 4), (11, 10)]$$

  La asignación quedaría:

  - $3 \to (3, 1)$
  - $4 \to (10, 4)$ (No es válido)
  - $6 \to (11, 10)$

  Cuando existe un óptimo que es invirtiendo los timestamps de 4 y 6.

- **Maximizar la libertad futura:** Asignamos primero por menor ventana del timestamp. Otro caso sencillo de descartar, ya que si tenemos:

  $$t = [3, 4, 6], \quad s = [(3, 1), (10, 4), (11, 10)]$$

  La asignación quedaría:

  - $3 \to (3, 1)$ con $e = 2$
  - $4 \to (10, 4)$ con $e = 4$
  - $3 \to (11, 10)$ con $e = 10$

  De nuevo, la asignación del 4 al timestamp $(10, 4)$ es errónea.

- **Minimizar el error por derecha:** Esta fue nuestra regla Greedy final. El enfoque fue el siguiente. Queremos atrasar lo más posible la decisión de ubicar una transacción dentro de un timestamp, por lo que buscamos encontrar la transacción que esté más cerca de $(s + e)$. Para ello, por cada timestamp buscamos:

  ```
  opciones = []
  para transaccion en transacciones:
      opciones.agregar((s_i + e_i - transaccion, timestamp))

  eleccion = min(opciones, key=opciones[0])
  ```

  De esta manera, nuestra regla Greedy termina siendo: *atrasar lo más posible la decisión de asociar una transacción a un timestamp para tener más disponibilidad en los otros timestamps.*

# Demostración del algoritmo

Queremos demostrar que el algoritmo propuesto determina correctamente si es posible asignar las transacciones $t = [t_1, t_2, \ldots, t_n]$ a los intervalos $(s_i, e_i)$, $1 \leq i \leq n$, o en el caso que no suceda, determine que no es posible.

## Hipótesis

- Tenemos $n$ transacciones y $n$ timestamps ordenados en orden creciente por $s_i$.
- El algoritmo asigna cada transacción $t_j$ al timestamp $(s_i, e_i)$ tal que:

$$s_i - e_i \leq t_j \leq s_i + e_i$$

- La estrategia Greedy selecciona, para cada timestamp, la transacción $t_j$ válida que minimiza el valor $(s_i + e_i - t_j)$, es decir, la que esté más a la derecha posible dentro del intervalo.

## Si existe una asignación válida, el algoritmo la encuentra

Supongamos que existe una asignación válida hasta la iteración $i$. La transacción $t_j$ asignada al intervalo $(s_i, e_i)$ cumple $s_i - e_i \leq t_j \leq s_i + e_i$ y que todas las transacciones están emparejadas de forma única con los timestamps.

Demostraremos por inducción que el algoritmo Greedy puede construir esta asignación.

**Base:** Para $i = 1$, el algoritmo selecciona la transacción $t_j$ más a la derecha que todavía no fue asignada y que cumple $s_1 - e_1 \leq t_j \leq s_1 + e_1$. Como existe una asignación válida, dicha transacción existe y el algoritmo la escoge correctamente.

**Paso inductivo:** Supongamos que hasta el paso $i - 1$ el algoritmo asignó correctamente las transacciones. Al considerar el intervalo $(s_i, e_i)$, el algoritmo selecciona la transacción disponible que esté más a la derecha dentro del intervalo. Si existiera otra solución válida que asigna otra transacción distinta a $(s_i, e_i)$, eso implicaría que la transacción elegida por el algoritmo puede ser reasignada a un timestamp posterior sin romper su validez actual, lo cual contradice la minimalidad de nuestra elección por derecha.

Por lo tanto, si existe una asignación válida, el algoritmo la construye paso a paso de forma correcta.

## Si no existe una asignación válida, el algoritmo lo detecta

Supongamos ahora que no existe una asignación válida. El algoritmo, al considerar los timestamps en orden creciente, intenta asignar en cada paso una transacción válida. Si en algún paso $i$ no existe ninguna transacción $t_j$ que cumpla $s_i - e_i \leq t_j \leq s_i + e_i$, entonces no existe manera de completar una asignación válida a partir de ese punto, ya que:

- Las transacciones previas ya fueron asignadas.
- No hay ninguna transacción restante que cumpla con $(s_i, e_i)$.

Entonces, ninguna de las transacciones restantes puede satisfacer la condición.

## Conclusión

El algoritmo Greedy planteado es **correcto y completo**.

# 3. Análisis de la complejidad

Primero, vamos a analizar el código analíticamente:

```python
def algoritmo(path = None, timestamps = None, transacciones = None):
    if path is None and (timestamps is None or transacciones is None):
        raise ValueError("Se debe proporcionar un archivo o listas de timestamps y transacciones.")

    if path is not None:
        timestamps, transacciones = leer_archivo(path)

    if timestamps is None or transacciones is None:
        raise ValueError("Se deben proporcionar timestamps y transacciones.")

    resultado = []

    sorted_timestamps = sorted(timestamps, key=lambda x: x[0])
    transacciones = deque(sorted(transacciones))

    for index, transaccion in enumerate(transacciones):
        mejor_timestamp = buscar_mejor_timestamp(sorted_timestamps, transaccion)

        if mejor_timestamp is None:
            return "No es el sospechoso correcto"

        resultado.append((transaccion, mejor_timestamp[0], mejor_timestamp[1]))
        sorted_timestamps.remove(mejor_timestamp)

    return sorted(resultado, key=lambda x: x[0])
```

Listing 1: Algoritmo principal

Analizando línea por línea:

- `leer_archivo(path)`: lectura y parseo de un archivo con $n$ entradas. Complejidad: $\mathcal{O}(n)$.
- `sorted(timestamps)`: ordenamiento de la lista de timestamps. Complejidad: $\mathcal{O}(n \log n)$.
- `sorted(transacciones)`: ordenamiento de la lista de transacciones. Complejidad: $\mathcal{O}(n \log n)$.
- `deque(...)`, conversión a deque: $\mathcal{O}(n)$.
- Bucle principal:
  - Se ejecuta $n$ veces (una por cada transacción).
  - En cada iteración:
    - `buscar_mejor_timestamp(...)`: búsqueda lineal en la lista de timestamps. $\mathcal{O}(n)$.
    - `sorted_timestamps.remove(...)`: eliminación lineal en lista. $\mathcal{O}(n)$.
    - `append(...)`: tiempo constante. $\mathcal{O}(1)$.
  - Total por iteración: $\mathcal{O}(n)$.
- Complejidad total del bucle: $\mathcal{O}(n^2)$.
- Ordenamiento final del resultado: $\mathcal{O}(n \log n)$.

**Complejidad total del algoritmo:**

$$\mathcal{O}(n) + \mathcal{O}(n \log n) + \mathcal{O}(n \log n) + \mathcal{O}(n^2) + \mathcal{O}(n \log n) = \boxed{\mathcal{O}(n^2)}$$

Entonces, analíticamente podemos decir que el algoritmo es $O(n^2)$.

# 4. Ejemplos de ejecución

Se creó un módulo para generar casos para validar nuestro algoritmo. Es un generador randomizado que dados los parámetros de $n$ y $error$ podemos obtener casos tanto sospechosos como no sospechosos:

```
@staticmethod
def generar_caso_es_sospechoso(n, error, timestamp_interval=(1, 100)):
    """
    Genera un caso donde es sospechoso.
    """
    timestamps = [(randint(timestamp_interval[0], timestamp_interval[1]), randint
    (1, error)) for _ in range(n)]
    timestamps.sort(key=lambda x: x[0])

    transacciones = [timestamps[i][0] + timestamps[i][1] - 1 for i in range(n)]

    return (timestamps, transacciones)

@staticmethod
def generar_caso_no_es_sospechoso(n, error, timestamp_interval=(1, 100)):
    """
    Genera un caso donde no es sospechoso.
    """
    timestamps = [(randint(timestamp_interval[0], timestamp_interval[1]), randint
    (1, error)) for _ in range(n)]
    timestamps.sort(key=lambda x: x[0])

    randint_index = randint(0, n - 1)

    transacciones = [timestamps[i][0] + timestamps[i][1] for i in range(n)]

    transacciones[randint_index] = transacciones[randint_index] + 1

    return (timestamps, transacciones)
```

Listing 2: Generación de casos con timestamp ajustado

En la funcion *generar_caso_es_sospechoso*, al calcular las transacciones le restamos 1 para asegurarnos que haya un timestamp que incluya la transacción $i$ dentro del intervalo.

Por el contrario, la función *generar_caso_no_es_sospechoso* se genera un índice random entero y se le sumará 1 a $transacciones[indice\_random]$, asegurando que esa transacción esté fuera del intervalo del timestamp.

## 4.1. Ejemplo 1

Este ejemplo se generó un caso donde es sospechoso.

### Timestamps

`(timestamp, error): (80, 78), (93, 80), (38, 37), (21, 67), (50, 56)`

### Transacciones

`74, 87, 105, 157, 172`

# Paso a paso del algoritmo

**Transacción #1: 74**
Evalúa todas las transacciones dentro del timestamp y se queda con la que está más cerca del

límite derecho del timestamp.

- Candidato válido: timestamp=21, error=67, offset=14
- Candidato válido: timestamp=38, error=37, offset=1
- Candidato válido: timestamp=50, error=56, offset=32
- Candidato válido: timestamp=80, error=78, offset=84
- Candidato válido: timestamp=93, error=80, offset=99
- ✓ Seleccionado: timestamp=38, error=37 (offset mínimo: 1)

**Transacción #2: 87**

- Candidato válido: timestamp=21, error=67, offset=1
- Candidato válido: timestamp=50, error=56, offset=19
- Candidato válido: timestamp=80, error=78, offset=71
- Candidato válido: timestamp=93, error=80, offset=86
- ✓ Seleccionado: timestamp=21, error=67 (offset mínimo: 1)

**Transacción #3: 105**

- Candidato válido: timestamp=50, error=56, offset=1
- Candidato válido: timestamp=80, error=78, offset=53
- Candidato válido: timestamp=93, error=80, offset=68
- ✓ Seleccionado: timestamp=50, error=56 (offset mínimo: 1)

**Transacción #4: 157**

- Candidato válido: timestamp=80, error=78, offset=1
- Candidato válido: timestamp=93, error=80, offset=16
- ✓ Seleccionado: timestamp=80, error=78 (offset mínimo: 1)

**Transacción #5: 172**

- Candidato válido: timestamp=93, error=80, offset=1
- ✓ Seleccionado: timestamp=93, error=80 (offset mínimo: 1)

## Resultado final

- Transacción 74 → Timestamp 38 ±37
- Transacción 87 → Timestamp 21 ±67
- Transacción 105 → Timestamp 50 ±56
- Transacción 157 → Timestamp 80 ±78
- Transacción 172 → Timestamp 93 ±80

## 4.2. Ejemplo 2

Este caso de ejecución es un ejemplo de no sospechoso.

### Timestamps

```
(timestamp, error): (15, 21), (15, 33), (35, 58), (69, 22), (97, 38)
```

### Transacciones

```
36, 48, 93, 92, 135
```

### Paso a paso del algoritmo

### Transacción #1: 36

- Candidato válido: timestamp=15, error=21, offset=0
- Candidato válido: timestamp=15, error=33, offset=12
- Candidato válido: timestamp=35, error=58, offset=57
- No entra en rango: timestamp=69, error=22 → Rango válido: [47, 91]
- No entra en rango: timestamp=97, error=38 → Rango válido: [59, 135]
- Seleccionado: timestamp=15, error=21 (offset mínimo: 0)

### Transacción #2: 48

- Candidato válido: timestamp=15, error=33, offset=0
- Candidato válido: timestamp=35, error=58, offset=45
- Candidato válido: timestamp=69, error=22, offset=43
- No entra en rango: timestamp=97, error=38 → Rango válido: [59, 135]
- Seleccionado: timestamp=15, error=33 (offset mínimo: 0)

### Transacción #3: 92

- Candidato válido: timestamp=35, error=58, offset=1
- No entra en rango: timestamp=69, error=22 → Rango válido: [47, 91]
- Candidato válido: timestamp=97, error=38, offset=43
- Seleccionado: timestamp=35, error=58 (offset mínimo: 1)

### Transacción #4: 93

- No entra en rango: timestamp=69, error=22 → Rango válido: [47, 91]
- Candidato válido: timestamp=97, error=38, offset=42
- Seleccionado: timestamp=97, error=38 (offset mínimo: 42)

### Transacción #5: 135

- No entra en rango: timestamp=69, error=22 → Rango válido: [47, 91]
- No se encontró timestamp válido. Transacción no emparejada.

**Explicación:** Esta transacción no se encuentra dentro del rango de ningún timestamp disponible. Por lo tanto, no es posible que haya sido generada por el sospechoso.

## 5. Análisis de la complejidad por cuadrados mínimos

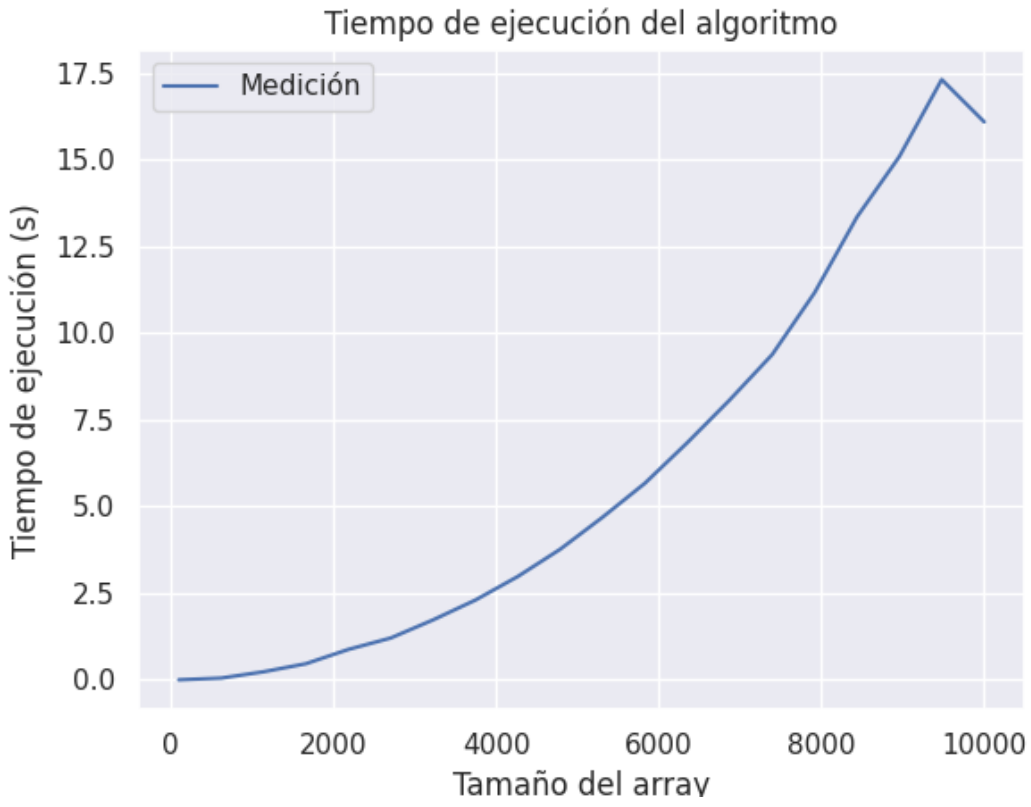
Para este caso, vamos a corroborar la complejidad calculada de manera teórica haciendo mediciones para el algoritmo Greedy de la funcion algoritmo, para posteriormente realizar un ajuste por cuadrados mınimos:

```
def get_random_timestamps_with_errors(size: int):
    return [(np.random.randint(0, 100000), np.random.randint(1, 101)) for _ in
    range(size)]

def get_random_timestamps_with_errors_and_transacciones(size: int):
    timestamps, transacciones = Generador.generar_caso_es_sospechoso(size, 100, (1,
     100))
    return (None, timestamps, transacciones)

x = np.linspace(100, 10_000, 20).astype(int)
results = time_algorithm(algoritmo, x, lambda s:
    get_random_timestamps_with_errors_and_transacciones(s))
```

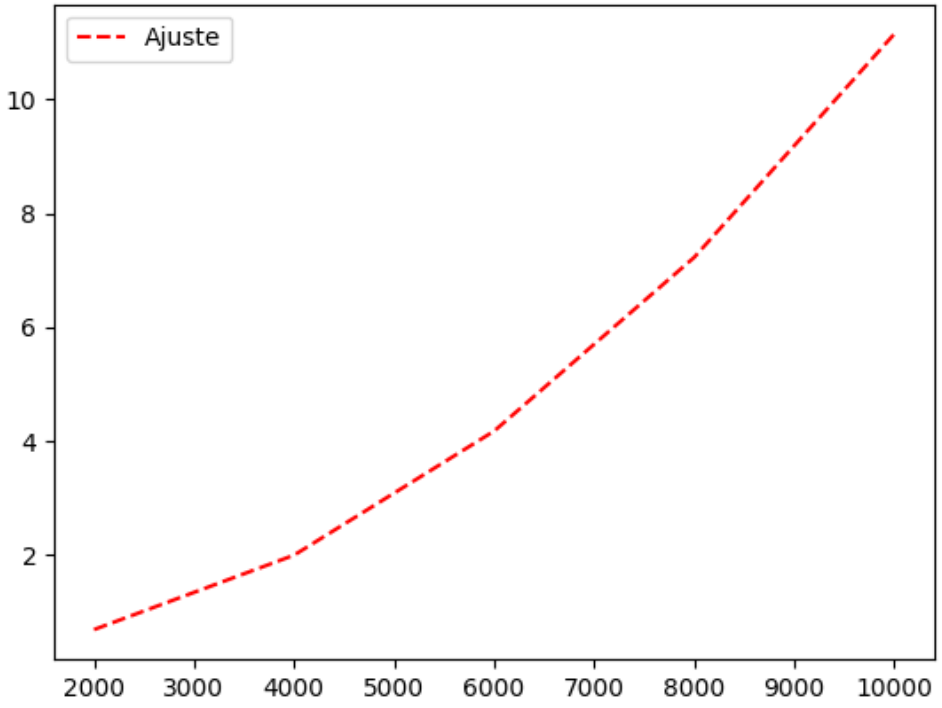
Siendo $x$ los valores de las abscisas, y el de las ordenadas: $[results[i]\ for\ i\ in\ x]$, se obtuvo:



Calculando el ajuste por cuadrados mínimos (ajustamos a una recta $c_1 * n * n + 1$) se obtuvo el siguiente error:

$$c_1 = 1{,}09 \times 10^{-6}$$
$$c_2 = 0{,}13582020927877914$$
Error cuadrático total = 0.10140855717689642

Y para finalizar, graficamos el ajuste y el error para cada tamano del array:




Se puede apreciar que para todos los tamanos del array el error de nuestro ajuste es bastante pequeño, lo cual nos lleva a concluir que nuestro algoritmo se comporta como $\mathcal{O}(n^2)$.