TEORÍA DE ALGORITMOS
(75.29) CURSO BUCHWALD - GENENDER

Trabajo Práctico 1
La mafia de los algoritmos greedy

25 de abril de 2025

M. B. García Lapegna
111848

M. U. Sáenz Valiente
111960

T. Goncalves Rei
111405

Índice

1. Consideraciones

Este informe se distribuirá de la siguiente manera:

- Sección 2: se enunciara el problema a resolver así como también las condiciones en las que debe ser resuelto.
- Sección 3: se explicaran los detalles del algoritmo implementado para resolver el problema. Entre otras cosas, se mostrara el código de la resolución y se justificara porque este se considera un algoritmo *Greedy*.
- Sección 4: se demostrara que el algoritmo implementado es optimo (es decir, que funciona correctamente en todos los casos). Para esto, se llevara a cabo una demostración matemática acompañada de algunos gráficos explicativos.
- Sección 5: se analizara la complejidad algorítmica de la solución. Para ello, se presentaran un conjunto de pruebas y mediciones realizadas al algoritmo.
- Por ultimo, se explayaran las conclusiones del trabajo presentado.

2. Introducción

En este informe se presentará un problema, y se propondrán y compararan diferentes formas de solucionarlo utilizando algoritmos *Greedy*. En su desarrollo, se verán las ventajas y desventajas de cada solución. Dada la solución final, se demostrará que su funcionamiento es optimo, y se explicara su complejidad algorítmica haciendo uso de gráficos comparativos.

2.1. Enunciado

"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... "

2.2. Condiciones

Para resolver el ejercicio se realizara un analisis del problema, y se propondra un algoritmo Greedy que pueda obtener una solucion al siguiente problema:

"Dados 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.

3. Resolución

En esta sección se presentara el algoritmo propuesto como solución.

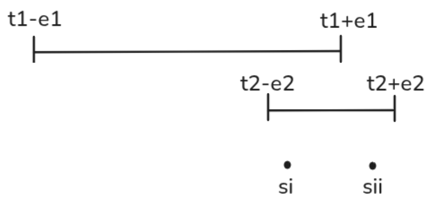
3.1. Planteo del problema a resolver

La mafia de los amigos Amarilla Pérez y el Gringo nos otorga n transacciones sospechosas en formato $[t_i - e_i; t_i + e_i]$ y n *timestamps* s_i de un posible sospechoso, quien podría ser *la rata* que buscamos (los *timestamps* se encuentran ordenados de forma ascendente). Este presunto sospechoso es *la rata* si las transacciones sospechosas coinciden con las operaciones del incriminado.

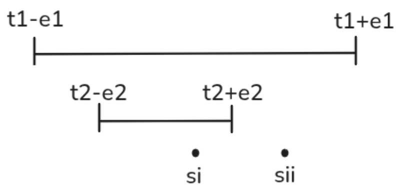
3.2. Soluciones no optimas

A continuación, se enumeraran algunas soluciones no optimas al problema, y se justificara su mal funcionamiento con un contraejemplo. En cada solución se asumira que el *timestamp* asignado a un intervalo es el primero que pertenece. Si se asigna el ultimo los contraejemplos son analogos. En adelante, llamaremos intervalo a cada par de valores $[t_i - e_i; t_i + e_i]$ correspondiente a una transacción sospechosa.

1. **Tomando el intervalo de menor duración en cada iteración**: El problema se puede intentar resolver eligiendo en cada paso el intervalo de menor duración y asignándole el primer *timestamp* perteneciente. Sin embargo, se puede demostrar con un simple contraejemplo que su funcionamiento no es optimo.



2. **Tomando el intervalo mas próximo a comenzar en cada iteración**: Otra posible solución puede ser buscar en cada paso el intervalo con tiempo de inicio mas próxima. Esto sigue sin ser optimo cuando, por ejemplo, un intervalo tiene un tiempo de inicio muy corto, y un tiempo de finalización muy largo. A continuación se muestra un ejemplo en el que esta solución no funciona.



3.3. Presentación del algoritmo optimo

Para solucionar el problema de forma optima, se propone la siguiente solución *Greedy*:

Se busca el intervalo $t_i + e_i$ con tiempo de finalización mas próximo y se le asigna (si es posible) el *timestamp* s_i mas pequeño dentro del rango $[t_i - e_i\ ;\ t_i + e_i]$ que aun no haya sido asignado a otra transacción.

Este procedimiento se realiza iterativamente hasta que todos los intervalos sean asignados (si es posible). En caso de no hallar una posible asignación, el sospechoso dado a analizar no es *la rata*.

3.4. Porque es *Greedy* el algoritmo propuesto

En esta sección se explicara porque el algoritmo presentado se considera *Greedy*.

En el libro *Algorithm Design* se encuentra la siguiente definición:

'An algorithm is greedy if it builds up a solution in small steps, choosing a decision at each step myopically to optimize some underlying criterion' ∼ Algorithm Design, p. 115

Esta definición es muy acertada, y puede desglosarse en que un algoritmo es *Greedy* si cumple con las siguientes condiciones:

1. La solucion se construye gradualmente, es decir, "paso a paso".
2. En cada paso se toma una decisión siguiendo un mismo criterio. Este criterio permite que la decisión que se toma sea siempre la mejor, es decir, la mas optima.
3. El criterio usado solo contempla el *estado local*. Definimos *estado local* como el estado en el que se encuentra nuestra solución parcial, sin tener en cuenta que decisiones se tomaron anteriormente.

   Ademas, llamaremos a una decisión *optimo local* cuando dado el *estado local*, la decision sea la mejor que se pueda tomar.
4. La iteración de optimos locales concluye en la obtención del optimo general. *

* Para facilitar la lectura de este informe, demostraremos mas adelante (sección 4.2) que el algoritmo es optimo y conduce siempre al optimo general.

En nuestro algoritmo, se observa que se realiza una asignación de cada intervalo *[ti-ei;ti+ei]* a una operación s_i con el fin de comprobar si cada transacción sospechosa puede corresponder a alguna operación del sospechoso al que se esta analizando. El criterio con el que realizamos la asignación consiste en buscar el intervalo mas próximo a finalizar (sin contar los ya asignados) y a este, asignarle el s_i mas pequeño que pertenezca a él.

Este criterio es correcto ya que en cada paso se maximiza el espacio de *timestamps* s_i disponible para el siguiente intervalo. Esto quiere decir que, en caso de que los siguientes intervalos tengan alguna asignación posible, esta no se vera comprometida por la asignación actual (se demostrara en detalle en la sección 4). Se puede notar que la decisión tomada solo contempla el estado local, ya que, la lógica utilizada para la asignación solo contempla los intervalos y *timestamps* disponibles, es decir, en ningún momento utiliza información sobre asignaciones previas. A esto se le suma que la totalidad de las asignaciónes se realiza iterando los pasos antes descriptos.

En síntesis, nuestro optimo local consiste en asignarle (de ser posible) un *timestamp* al intervalo mas próximo a finalizar (entre los aun no asignados). En caso de haber varios *timestamps* candidatos, le asigna el mas pequeño de ellos. Si no se encuentra un *timestamp* que pertenezca al intervalo, entonces el sospechoso no es *la rata*.

Por lo antes descripto, se puede ver que nuestro algoritmo cumple todas las condiciones para ser un algoritmo *Greedy*, y que por lo tanto, lo es.

3.5. Implementación

A continuación se presenta una primera implementación del algoritmo anteriormente propuesto.

```
def verificar_sospechoso(intervalos, timestamps_sospechoso):
    intervalos = sorted(intervalos, key=lambda x: x[1])
```

```
    asignaciones={}

    for intervalo in intervalos:
        encontro = False

        for i in range(len(timestamps_sospechoso)):
            timestamp = timestamps_sospechoso[i]
            if timestamp is None:
                continue
            if timestamp >= intervalo[0] and timestamp <= intervalo[1]:
                asignaciones[intervalo] = timestamp
                timestamps_sospechoso[i] = None
                encontro = True
                break
        if not encontro:
            return False, asignaciones

    return True, asignaciones
```

En esta versión inicial del código, se utiliza una lista para almacenar los *timestamps* disponibles. Cuando un *timestamp* es asignado a un intervalo se reemplaza su valor por un *None* dentro de la lista de *timestamps*.

El problema que surge a raíz de esto, es que a medida que se van asignando los *timestamps* a los distintos intervalos, la aparición de *None* en la lista se hará cada vez mas frecuente. Esto sin contar que la lógica de nuestro algoritmo *Greedy* se basa principalmente en agarrar el *timestamp* mas pequeño entre los posibles, por lo que, en una gran parte de los casos, a medida que se avance en las iteraciones, los primeros lugares de la lista serán grandes candidatos a ser *None*, lo cual ocasionaría una gran cantidad de iteraciones innecesarias.

3.5.1. Optimización propuesta

```
def verificar_sospechoso(intervalos, timestamps_sospechoso):
    dicc_timestamps = obtener_dicc_timestamps(timestamps_sospechoso)
    intervalos = sorted(intervalos, key=lambda x: x[1])
    asignaciones={}

    for intervalo in intervalos:
        encontro = False
        for timestamp in dicc_timestamps:
            if en_rango(timestamp, intervalo):
                dicc_timestamps[timestamp] -= 1

                if dicc_timestamps[timestamp]==0:
                    del dicc_timestamps[timestamp]

                encontro = True
                asignaciones[intervalo] = timestamp
                break
        if not encontro:
            return False, asignaciones

    return True, asignaciones
```

Esta es una versión mejorada del código anterior, la cual soluciona el problema anteriormente descrito.

En este caso, para almacenar los n *timestamps* se utiliza un diccionario de estructura *clave*=s_i y *valor*= 'cantidad *timestamps* de valor s_i'. Esto sirve en escenarios en los que un *timestamp* s_i tiene muchas repeticiones. En ese caso, restando uno a la cantidad de apariciones se lo descarta para próximas iteraciones. Una vez que ya no quedan mas elementos con ese valor se lo elimina del diccionario y ya no se lo itera nuevamente como se lo hacia en el primer algoritmo propuesto.

Por lo tanto y en conclusión, esta optimización mejora el algoritmo en dos aspectos:

- Evita iterar valores sin sentido.

- Evita recorrer *timestamps* de igual valor, que se encuentran por fuera del rango del intervalo.

3.5.2. Una ultima optimización

Con la optimización anterior, se solucionaba el problema de que un *timestamp* ya no disponible se siga visitando en las próximas iteraciones. Pero aun hay un problema, la búsqueda sobre los *timestamps* sigue siendo lineal. Esto puede ser un problema cuando, por ejemplo, el *timestamp* buscado se encuentra muy lejos de las primeras posiciones (por ejemplo, si el tiempo de inicio de un intervalo es muy grande). Se puede ver que este caso no sera el mas frecuente, ya que nuestro algoritmo escoge siempre el *timestamp* mas pequeño posible. Sin embargo, en algunos escenarios este problema puede tener un alto costo.

Una posible solución al problema recién planteado, es utilizar *búsqueda binaria* para determinar que *timestamp* se le asignara a cada intervalo. Se sabe que el algoritmo de *búsqueda binaria* solo puede implementarse sobre arreglos, por lo cual, el código de esta nueva implementación seria similar (en principio) al de la primera solución propuesta. Sin embargo, se puede plantear un algoritmo que *fusione* las dos versiones anteriores, de manera que, tendremos un diccionario idéntico al de la versión anterior, y ademas, tendremos un arreglo en el que guardaremos los *timestamps* sin repeticiones. Dado esto, se puede implementar búsqueda binaria sobre el arreglo de *timestamps* sin repeticiones. La implementación de este algoritmo es la siguiente:

```
def verificar_sospechoso(intervalos, timestamps_sospechoso):

dicc_timestamps , valores_timestamps = obtener_dicc_timestamps(
timestamps_sospechoso)
intervalos = sorted(intervalos, key=lambda x: x[1])
asignaciones={}

for intervalo in intervalos:
    if len(valores_timestamps)==0:
        return False , asignaciones
    elem = busqueda_binaria(valores_timestamps,intervalo[0],intervalo[1])
    if elem == -1:
        return False , asignaciones
    dicc_timestamps[elem] -= 1

    if dicc_timestamps[elem]==0: borramos del dicc para no iterar de mas
        del dicc_timestamps[elem]
        valores_timestamps.remove(elem)

    asignaciones[intervalo] = elem

return True, asignaciones
```

Se puede observar que esta ultima versión mejora drásticamente el funcionamiento del algoritmo en aquellos escenarios en los que se debe recorrer una gran cantidad de *timestamps* para realizar una asignación.

En general, las asignaciones tendrán complejidad $O(n)$ solo cuando al *timestamp* encontrado le quede una única aparición (es decir, o bien ya se asignaron el resto de repetidos o siempre fue único). En los casos restantes, el mayor costo de la asignación recaerá en la búsqueda del valor del *timestamp* mediante *búsqueda binaria*, lo cual tendrá complejidad $O(log(n))$.

Mas adelante, se realizarán mediciones sobre estas dos ultimas versiones para comparar su rendimiento. Estas dejaran ver que el algoritmo anterior podía funcionar mejor en escenarios específicos. Sin embargo, esta ultima versión destaca por tener un rendimiento aceptable en cualquier escenario.

4. Demostración

Demostraremos que, en cada paso, la asignación de un *timestamp* a un intervalo (o de un intervalo a un timestamp) que hace nuestro algoritmo no obstruye la asignación de ningún *timestamp* siguiente.

4.1. Hipótesis

Suponemos que:

- $\exists$ n *timestamps* del sospechoso si, con $si \in S$ / $S = [s_1, .., s_n]$ $i \in [0, n]$, $s_i \in \mathbb{R}^+$, además se cumple que $\forall$ $s_i \wedge s_j$ con $i < j$, $s_i \leq s_j$.
- $\exists$ n *timestamps* de transacciones sospechosas t_i, con margen de error e_i, de manera que definimos a un intervalo x_i como $x_i = [t_i - e_i; t_i + e_i]$, $t_i - e_i, t_i + e_i \in \mathbb{R}^+$ / $X = [x_1, .., x_n]$ $i \in [0, n]$.
- Definimos $\forall$ x_i

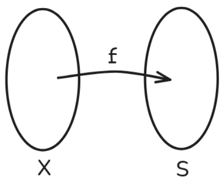
$$x_i = \begin{cases} a &= t_i - e_i \\ b &= t_i + e_i \end{cases}$$

de modo que $x_i = [a, b]$.

Además:

- $\exists$ una función de asignacion $f : X \rightarrow S$ / $f(x_i) = s_i \Leftrightarrow$ **i.** $s_i \in x_i$.
  ii. si no $\exists!$ $s_i \in x_i$, es decir,
  $\exists$ $s_k \in x_i \wedge \exists$ $s_j \in x_i$
  con $j > k \Rightarrow f(x_i) = s_k$*.
  iii. $\forall x_j = [c, d], x_i = [a, b], d < b$
  $s_i \notin x_j$

  * En palabras, si tengo un $x_i \in X$ con k *timestamp* $s_k \in S$ / $S = [s_1, .., s_k] \wedge$ todos los $s_k \in x_i$. Se cumple que $f(xi) = s_1$.



Notar que la asignación realizada por f es la misma que realiza nuestro algoritmo.

4.2. Tesis

Siempre que exista una posible asignación entre *timestamps* e intervalos, la funcion f la hallara. Si la función f no encuentra asignación, entonces no la hay.

Para ello demostraremos por metodo inductivo:

- La asignación se hace correctamente para el *timestamp* s_1 (caso base). **(Sección 4.3.1)**
- Para facilitar la demostración del paso inductivo, demostraremos que en caso de existir una posible asignación para todo el conjunto de intervalos, una asignación previa no dejara a ninguna posterior sin asignación. **(Sección 4.3.2)**
- $\forall$ $s_k > s_1$, si s_k se asigna correctamente $\Rightarrow$ s_{k+1} se asigna correctamente (paso inductivo). **(sección 4.3.3)**

4.3. Demostración

4.3.1. Caso base

En esta sección se demostrara que la hipótesis se cumple para el primer *timestamp* s_1

Sea s_1 el *timestamp* de menor valor.

- $s_1 \in x_i \Rightarrow$ por hipótesis (**i, ii, iii**) $s_1 \in Im(f)$
- $s_1 \notin Im(f) \Rightarrow$ por hipótesis (**i, ii, iii**) $\nexists\ x_i \ /\ s_1 \in x_i, \forall i \in [0, n]$
- Por hipótesis (**iii**) sabemos que s_1 se asignara al intervalo con tiempo de finalización mas pequeño al que pertenezca.

Con esto, demostramos que el primer *timestamp* s_1 encuentra asignación y ademas que esa asignación es correcta.

4.3.2. Demostración previa al paso inductivo

A continuación demostraremos que cualquier asignación que hagamos no deja sin asignación a ningun intervalo siguiente.

Para la demostración suponemos que:

(1) $\exists\ s_i \in x_j = [a, b]$.

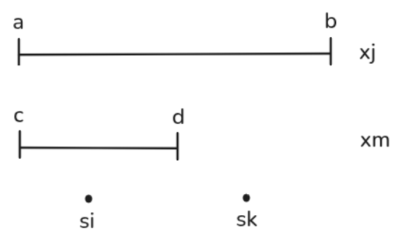
(2) $\exists\ s_k > s_i$ con $k > i$

Notar que se mostraran todas las diferentes posibilidades $\forall x_k \neq x_j$, por lo cual esta demostración aplica para cualquier cantidad de intervalos (cualquier intervalo entrara en alguno de los casos que se describirán).

Caso 1

Se tiene (1) y (2), además:

- $s_k \in x_j \ /\ x_j$ es $!x_i \in X$ al que $\in s_k$
- $s_i \in x_m = [c, d]$, $d < b$

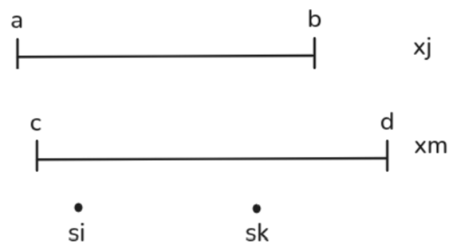


$\Longrightarrow f(x_j) = s_k \wedge\ f(x_m) = s_i$ ✓ El algoritmo encuentra asignación valida.

Caso 2

a) Se tiene (1) y (2), además:

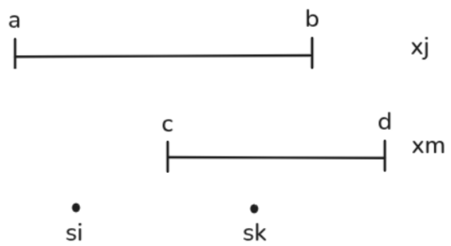
- $s_k \in x_m$=[c, d], $d > b$
- $s_i \in x_m$



$\Longrightarrow f(x_j) = s_i \land\ f(x_m) = s_k$ ✓ El algoritmo encuentra asignación valida.

b) Se tiene (1) y (2), además:

- $s_k \in x_m$=[c, d], $d > b$
- $s_i \notin x_m$

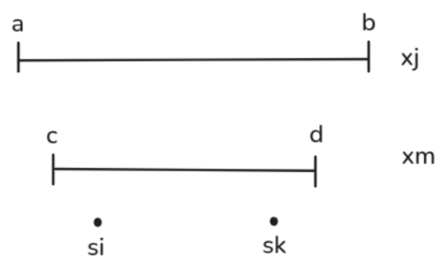


$\Longrightarrow f(x_j) = s_i \land\ f(x_m) = s_k$ ✓ El algoritmo encuentra asignación valida.

Caso 3

a) Se tiene (1) y (2), además:

- $s_k \in x_m$=[c, d], $d < b$
- $s_i \in x_m$

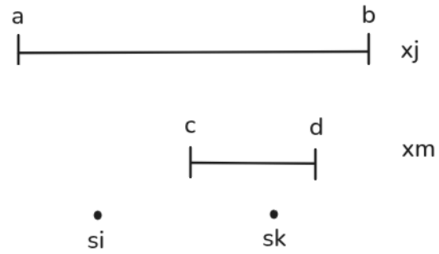


$\Longrightarrow f(x_j) = s_k \land\ f(x_m) = s_i$ ✓ El algoritmo encuentra asignación valida.

b) Se tiene (1) y (2), además:

- $s_k \in x_m$=[c, d], $d < b$

- $s_i \notin x_m$

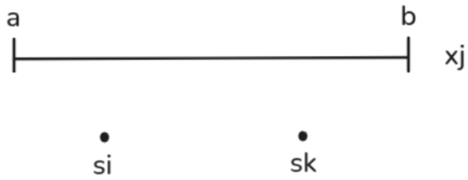


$\Longrightarrow f(x_j) = s_i \wedge \ f(x_m) = s_k$ ✓ El algoritmo encuentra asignación valida.

Caso 4

Se tiene (1) y (2), además:

- $s_k \in x_j \ / \ x_j$ es $!x_i \in X$ al que $\in s_k$
- $s_i \in x_j \ / \ x_j$ es $!x_i \in X$ al que $\in s_i$



× No hay asignación posible dado que ambos *timestamps* solo pertenecen a un único intervalo.

Demostramos así que cualquier asignación de un *timestamp* no impide la asignación de un *timestamp* posterior.

4.3.3. Paso inductivo

Sea s_h un *timestamp* y sea $x_i = [a, b]$ una transacción sospechosa tal que $f(x_i) = s_h$.

Teniendo s_{h+1}:

- Sabemos por lo anteriormente demostrado (**Sección 4.3.2**) que la asignación $f(x_i) = s_h$ no dejara sin asignación posible a s_{h+1}. Por lo tanto, de haber asignación posible sabemos que existirá $x_k \ / \ s_{h+1} \in x_k \Rightarrow f(x_k) = s_{h+1}$ (o algún otro, pero existirá al menos un intervalo al que se pueda asignar)
- De haber múltiples asignaciones posibles para el *timestamp* s_h, sabemos por hipótesis (**ii**) que s_h se asignara al intervalo de menor tiempo de finalización al que pertenezca.

Demostramos así que dada una asignación $f(x_k) = s_h$, el *timestamp* s_{h+1} se asigna también correctamente.

Concluimos entonces, que siempre que haya una posible asignación nuestra *función de asignación* lo encontrara, y por lo tanto también nuestro algoritmo.

5. Mediciones

Se realizaron una serie de mediciones para así poder determinar el tiempo real de ejecución del algoritmo con distintos tamaños de entrada. Con esto, mas adelante se analizó su complejidad temporal.

Estas mediciones ayudan a visualizar como se comporta el algoritmo observando su desempeño en distintos escenarios y comparando su complejidad algorítmica.

5.1. Complejidad

Para poder analizar la complejidad del algoritmo, desglosaremos el código en distintas secciones.

La función **verificar sospechoso** comienza de la siguiente forma:

```
def verificar_sospechoso(intervalos, timestamps_sospechoso):
    dicc_timestamps , valores_timestamps = obtener_dicc_timestamps(
    timestamps_sospechoso)
```

En esta parte del código, se observa la llamada a la función **obtener_dicc_timestamps**, la cual recorre los n *timestamps* del sospechoso:

```
def obtener_dicc_timestamps(timestamps):
    dicc_timestamps = {}
    valores = []

    for timestamp in timestamps:
        if timestamp not in dicc_timestamps:
            dicc_timestamps[timestamp] = 0
            valores.append(timestamp)

        dicc_timestamps[timestamp] += 1

    return dicc_timestamps , valores
```

Esta función tiene una complejidad lineal, es decir, su complejidad es $O(n)$. Notar que todas las operaciones realizadas dentro de cada iteración son constantes.

Luego, se realiza un ordenamiento de los n intervalos usando como criterio su tiempo de finalización. Se hace uso de la función *sorted* de *Python*, la cual tiene complejidad $O(n \times log(n))$:

```
    intervalos = sorted(intervalos, key=lambda x: x[1])
```

Por ultimo, se recorren los n intervalos. En cada iteración, utilizando *búsqueda binaria*, se busca el *timestamp* mas pequeño dentro del intervalo. En caso de hallar asignación, resta una aparición en el diccionario de *timestamps* y se guarda la asignación. Si al restar la aparición se llega a 0, significa que ya no hay mas *timestamps* con ese valor, por lo que se lo borra tanto de la lista de valores, como del diccionario de *timestamps*. Si no halla una asignación, retorna *False* junto con un diccionario de las asignaciones hasta el momento.

```
    for intervalo in intervalos:
        if len(valores_timestamps)==0:
            return False , asignaciones
        elem = busqueda_binaria(valores_timestamps,intervalo[0],intervalo[1])
        if elem == -1:
            return False , asignaciones
        dicc_timestamps[elem] -= 1

        if dicc_timestamps[elem]==0:
            del dicc_timestamps[elem]
            valores_timestamps.remove(elem)

        asignaciones[intervalo] = elem

    return True, asignaciones
```

Por lo que, basándonos en el peor caso, de base tenemos una complejidad algorítmica de *O(n)* por recorrer todos los intervalos . Luego, suponiendo que ninguno de los *n timestamps* tiene repetidos, se hace un *remove()* a la lista por iteración, lo cual también es *O(n)*. Entonces, en sintesis esto seria, por cada intervalo recorrido hago un *remove()*, la complejidad de esta sección del código se traduce en: $O(n) \times O(n) = O(n^2)$.

Este $O(n^2)$ actuara de cota superior en este fragmento del código. Ya en un mejor escenario, en el cual tendriamos una mayor cantidad *timestamps* repetidos, cada iteración (por intervalo) seria logarítmica, es decir , en un buen caso por la utilización de *búsqueda binaria*, la iteración nos costaría $O(log\ (n))$.

5.2. Complejidad Final del Algoritmo

- Construcción del diccionario de *timestamps*: $O(n)$
- Ordenamiento de intervalos: $O(n \times log(n))$
- Recorrer intervalos: $O(n)$
- Busqueda Binaria: $O(log(n))$
- Operaciones en diccionario: $O(1)$
- *remove()* en lista: $O(n)$

El algoritmo tiene complejidad teórica final $O(n^2)$ en el peor caso, debido a lo explicado previamente.

5.3. Justificación de complejidad con gráficos

Se realizan gráficos comparativos entre el algoritmo planteado y las funciones $O(n \times log(n))$ y $O(n^2)$. Los datos utilizados para la gráfica son pruebas que se realizaron con distintos sets de datos.

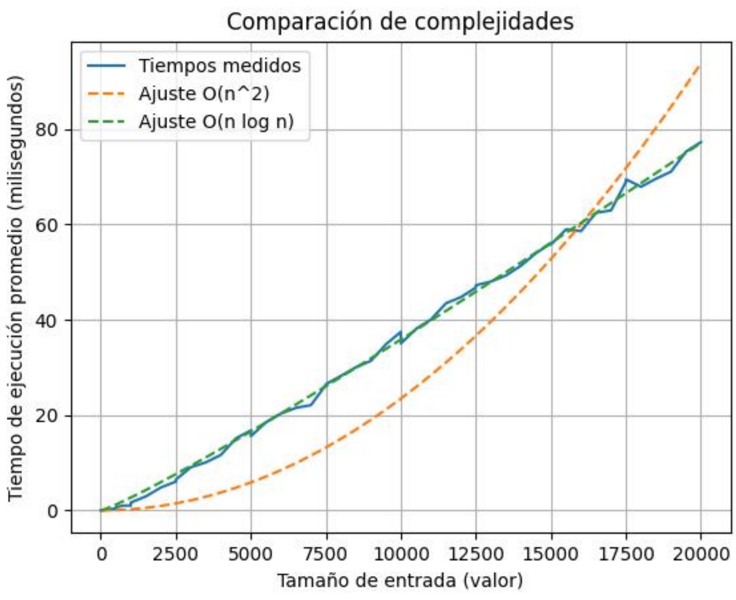


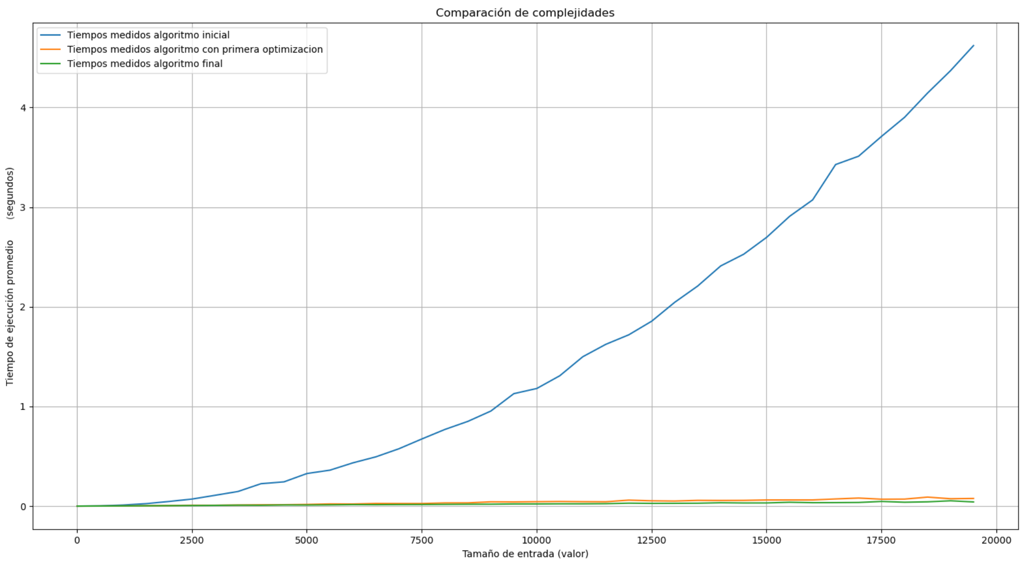
Figura 1: Comparación de Complejidades

En la imagen podemos ver como nuestro algoritmo se ajusta mejor a la recta de $O(n \times log(n))$. Esto nos da indicios de que al poner el código en practica, se comporta como un algoritmo de mejor complejidad que $O(n^2)$ aunque en la teoría se haya demostrado que lo es.

5.4. Análisis según escenario

Como se menciono anteriormente, se eligió a la versión que usaba *Busqueda Binaria*, con el fin de que el algoritmo sea lo mas optimo posible.

A continuación se hará un breve análisis de cual de nuestros algoritmos funciona mejor en cada escenario.



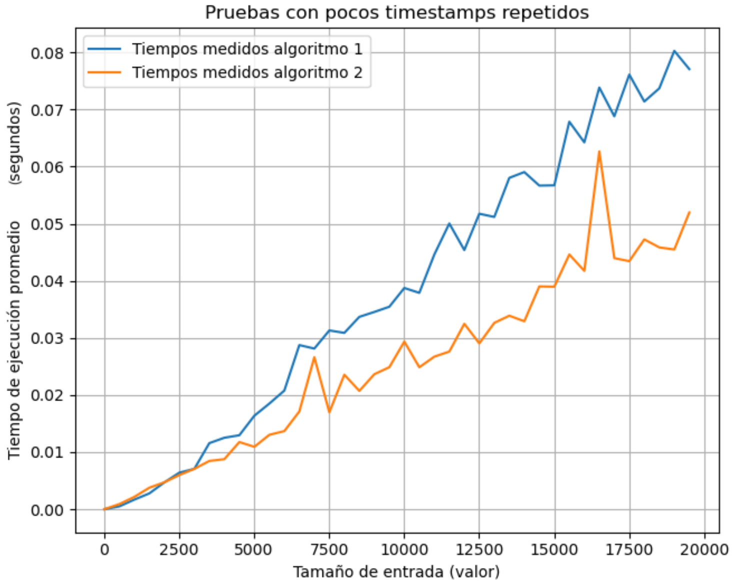
Esta primera imagen muestra como se comportan las tres versiones del algoritmo cuando hay muchos *timestamps* con valores repetidos. Como se puede observar, hay una diferencia notoria entre la versión inicial y las optimizadas. La curva del algoritmo inicial tiene una trayectoria muy similar a una curva cuadrática, y eso tiene sentido. Recordemos que en el primer algoritmo por cada intervalo se podía llegar a recorrer toda la lista de *timestamps* sin importar si su valor había o no sido utilizado.

En este gráfico a las curvas restantes, en comparación con la curva anteriormente mencionada, se las nota bastante pares. A continuación, se mostrara un análisis mas profundo entre estas dos curvas que revelara como varia su comportamiento dependiendo la situación a la que es sometido el algoritmo. En adelante, llamaremos *algoritmo 1* al planteado en la seccion 3.5.1. y *algoritmo 2* al de la sección 3.5.2..

5.4.1. Comparación entre algoritmos

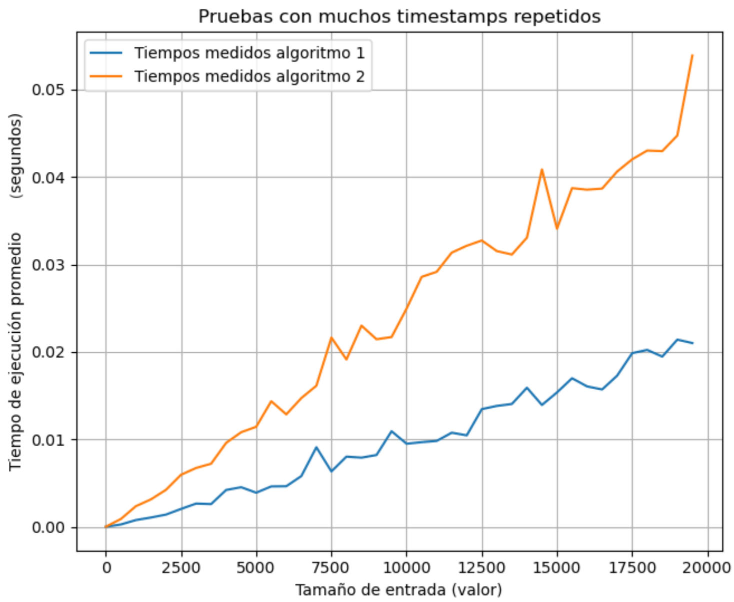
En esta sección analizaremos el comportamiento de los algoritmos mencionados previamente en dos distintos escenarios:

1. **Mayor cantidad de solapamiento entre intervalos y menor cantidad de *timestamps* con valores repetidos**



En el gráfico podemos observar como este escenario beneficia ampliamente a los tiempos de ejecución del *algoritmo 2*. Si bien la complejidad de ambos algoritmos era idéntica, la nueva versión mejora notoriamente su rendimiento frente a la anterior.

2. **Menor cantidad de solapamiento entre intervalos y mayor cantidad de *timestamps* con valores repetidos**

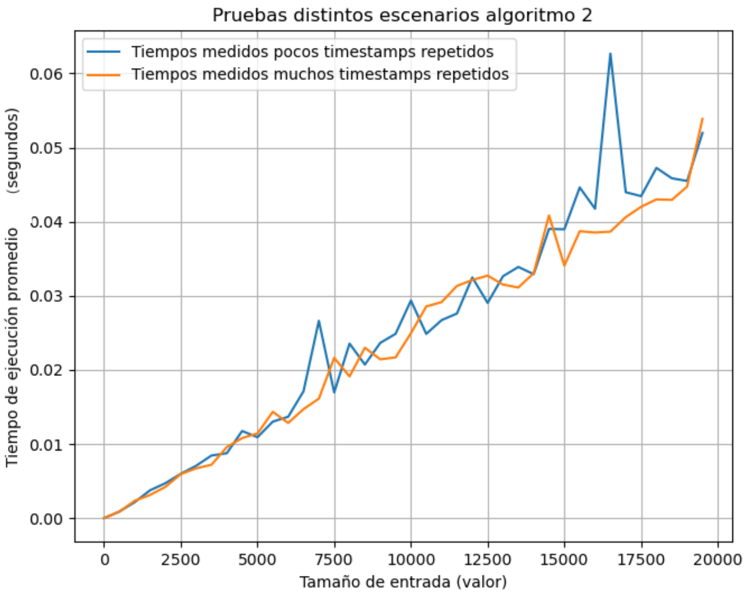


En este gráfico, se puede ver notablemente como el *algoritmo 1* se comporta de manera mas eficiente que el *algoritmo 2*. Se podria afirmar entonces, que una menor cantidad de solapamientos entre los intervalos aporta positivamente al rendimiento del *algoritmo 1*.

5.4.2. Comparación entre escenarios en un mismo algoritmo

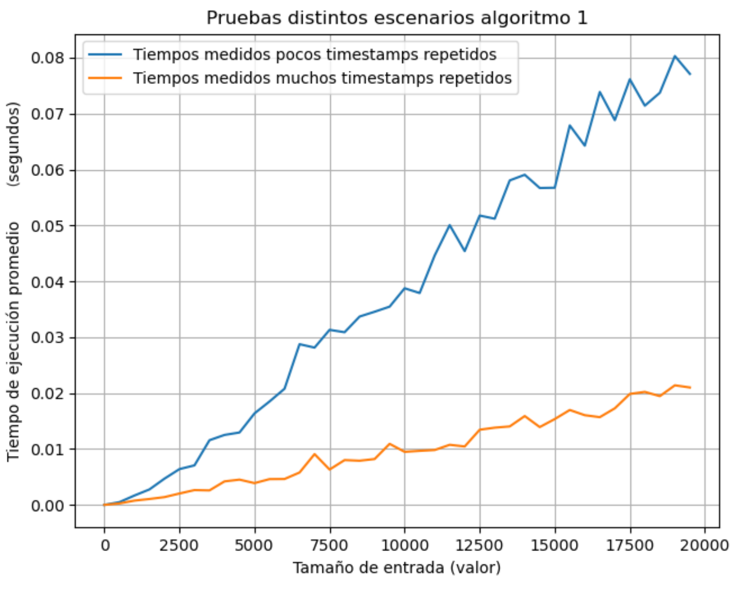
A continuación se explicara porque determinamos que, basándonos en tiempos de ejecución, la mejor opción es el *algoritmo 2* por sobre el *algoritmo 1*.

- Analizamos el comportamiento del *algoritmo 2* en los dos escenarios mencionados en la sección 5.5.1. :



Notamos que en ambos escenarios, el algoritmo parece mantenerse bastante a la par. Es decir que, sin importar de que caso se trate, los tiempos de ejecución se mantienen casi invariantes. Esta poca variación supone una gran ventaja, ya que asegura que el algoritmo funcionara de manera similar sea cual sea el escenario.

- Analizamos el del *algoritmo 1* en también ambos escenarios:



En este algoritmo si podemos notar una gran diferencia en tiempos de ejecución. Este es muy eficiente cuando hay muchos *timestamps* con repetidos (lo cual tiene sentido, en este caso el diccionario sera mas corto) y por otro lado, cuando hay muchos valores distintos de estos, notamos un peor tiempo algorítmico (esto debido a que recorrerá una mayor cantidad de *timestamps* hasta hallar el indicado).

Con este análisis podemos concluir que, aunque si es cierto que en un caso en particular el *algoritmo 1* es mas optimo (temporalmente) que el *algoritmo 2*, luego es mucho peor en otro caso. En cambio, el *algoritmo 2* mantiene un tiempo de ejecución constante para cualquier escenario posible. Por lo que, es mejor utilizar un algoritmo que no varié tanto según la situación, como lo hace el *algoritmo 1*. Nos apoyaremos en esto ultimo para justificar porque consideramos que el *algoritmo 2* es mejor para el uso general (sin información previa sobre el escenario a analizar).

5.5. Error Cuadrático Medio

El ECM(error cuadrático medio) es una medida que permite acotar el error cometido al aproximar una serie de puntos mediante una función especifica. En general, el error cuadrático se define de la siguiente manera:

Dado un conjunto de puntos del tipo (x_i, y_i):

$$ECM = \sum_{i=0}^{k} r_i, r_i = (y_i - f(x_i))^2 \tag{1}$$

En nuestro algoritmo, vamos a calcular el error medio al ajustar los puntos resultantes de nuestras mediciones (véase gráfico de la sección 5.3) a funciones correspondientes a las complejidades $O(n \times log(n))$ y $O(n^2)$. Estas funciones tienen la siguiente forma:

Función $O(n^2) \implies f(n) = a * n^2, a \in \mathbb{R}^+$

Función $O(n \times log(n)) \implies f(n) = a * n * log(n), a \in \mathbb{R}^+$

Realizando el calculo del ECM con estas dos funciones, llegamos a los siguientes resultados:

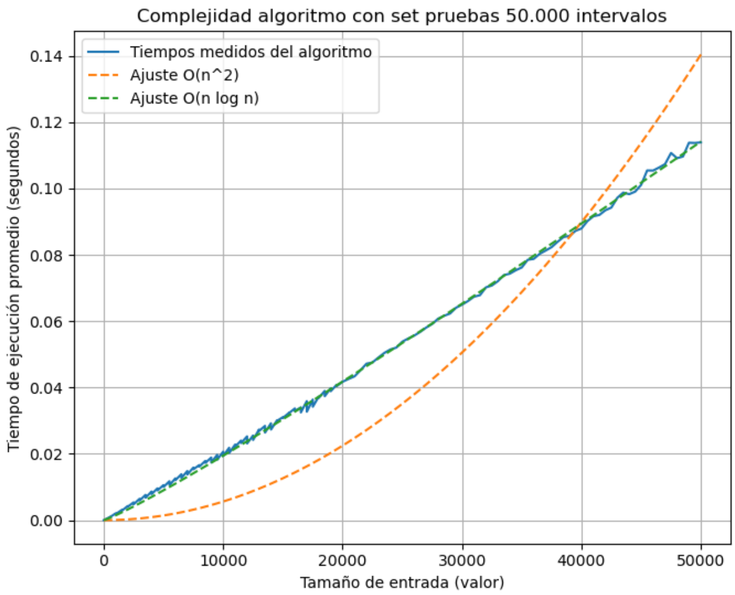
- Con función cuadratica: ECM con $O(n^2) = 1,823 \times e^-5$
- Con función lineal-logaritmica: ECM con $O(n \times log(n)) = 5,67 \times e^-7$

De esta manera se comprueba que la función que mejor ajusta al rendimiento de nuestro algoritmo es la función correspondiente a la complejidad $O(n \times log(n))$.

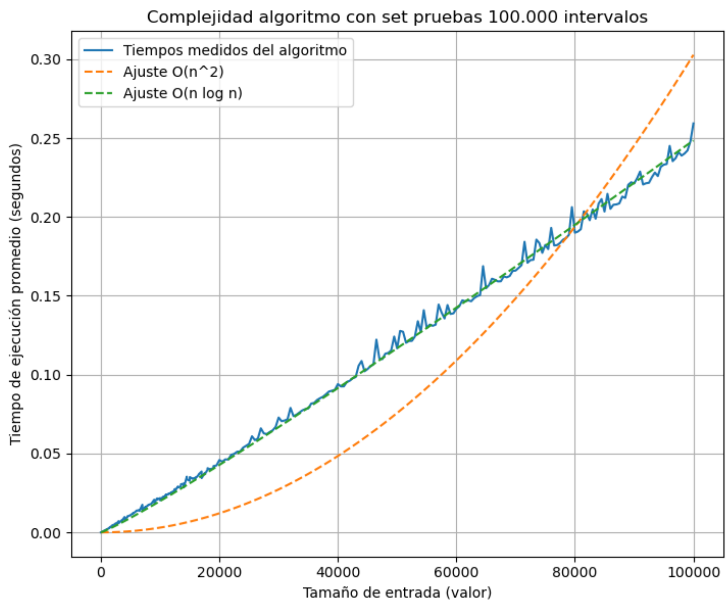
5.6. Complejidad algorítmica según volumen

En este apartado se mostraran los gráficos correspondientes a la ejecución del algoritmo con sets de mayor rango, para asi poder observar con mayor exactitud a que tiende su comportamiento.

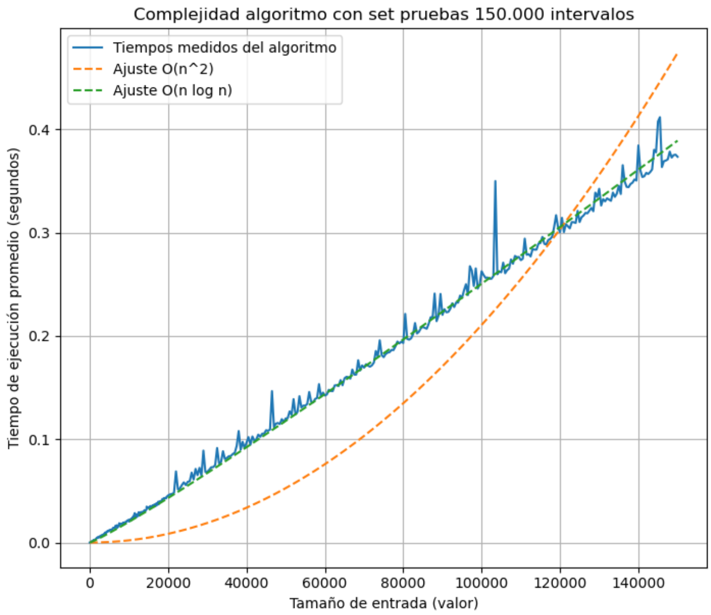
5.6.1. Comportamiento con set de 50.000 intervalos



5.6.2. Comportamiento con set de 100.000 intervalos

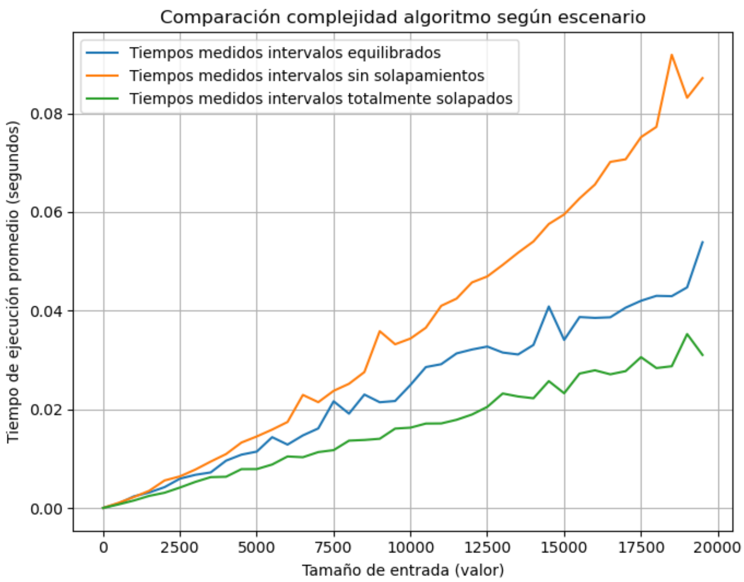


5.6.3. Comportamiento con set de 150.000 intervalos



Podemos notar que independientemente de la cantidad de intervalos que el algoritmo deba procesar, su comportamiento se comporta de manera invariante.

5.7. Comportamiento del algoritmo final ante escenarios antagónicos



Observación: con *intervalos equilibrados* se hace referencia a un set que tenga tanto intervalos con solapamientos como otros que no.

Como se puede observar en el gráfico, en el set en el que no hay intervalos solapados se puede ver un empeoramiento bastante importante respecto a los otros. Esto tiene sentido, ya que al no haber solapamientos entre intervalos tampoco hay posibles *timestamps* de igual valor (por supuesto, en el caso de que el sospechoso sea la *rata*), y por lo tanto no solo habrá mas elementos para hacer la *búsqueda binaria*, sino que cada vez que hagamos una asignación, al ser todos únicos, el *timestamp* se deberá borrar tanto del diccionario como de la lista(que usábamos para la *búsqueda binaria*), y borrar de la lista nos cuesta $O(n)$. Por lo que, cada asignación nos costara en absolutamente todos los casos $O(n)$. Es por esto que hay tanta diferencia entre este caso y los restantes.

Respecto a los casos restantes, el tiempo de ejecución es similar, aunque por lo dicho anteriormente, el algoritmo muestra ser mejor temporalmente si los intervalos se encuentran solapados en su totalidad. Esto también tiene que ver con que al estar todos los intervalos solapados, hay muchas mas probabilidades de tener *timestamps* de igual valor, lo que nos lleva a tener que hacer menos eliminaciones dentro de la lista. En este caso, cuando nuestro algoritmo realice la *búsqueda binaria* para hallar el menor *timestamp* perteneciente a este, no solo la cantidad de posibilidades sera posiblemente menor, sino que ademas siempre sera ir para el lado izquierdo del arreglo en busca del *timestamp* mas pequeño.

6. Conclusión

A lo largo de la realización del trabajo pudimos abordar distintos aspectos sobre la elaboración y optimización de algoritmos *Greedy*.

Partimos de una estrategia *Greedy* que soluciono el problema propuesto, y con esta se planteo un algoritmo inicial. Luego, se fueron realizando optimizaciones sobre ese algoritmo, hasta llegar a una versión final. Pudimos observar como con pequeños cambios, el tiempo de ejecución del algoritmo mejoró notablemente.

Dada la versión final del algoritmo, se analizo tanto su comportamiento como su rendimiento en diferentes escenarios y volúmenes respecto a la cantidad de intervalos/*timestamps*. Con este análisis, pudimos concluir que el ultimo algoritmo propuesto tenia un funcionamiento mas estable en la mayoría de escenarios, y que por lo tanto, era el mejor.