.UBAfiuba
FACULTAD DE INGENIERÍA

Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

Trabajo Práctico 1
Algoritmo Greedy

7 de abril de 2025

Eduardo Martín Bocanegra	106028
Nicolás Martín Guerrero	112514
Facundo Maximiliano Cabral	110553

1. Introducción del Trabajo

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, coinciden 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 Pérez nos sugirió que nos apuremos, si es que no queremos ser nosotros los siguientes sospechosos...

1.1. Consigna del Problema

1. 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.

2. Demostrar que el algoritmo planteado determina correctamente siempre si los timestamps del sospechoso corresponden a los intervalos sospechosos, o no. Es decir, si conciden, que encuentra la asignación, y si no conciden, que el algoritmo detecta esto, en todos los casos.

3. 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.

4. 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.

5. 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. Para esto, proveemos una *explicación detallada*, en conjunto de ejemplos. Recomendamos tomar más de una medición de la misma muestra y quedarse con el promedio para reducir el ruido en la medición.

6. Agregar cualquier conclusión que les parezca relevante.

2. Análisis del Problema

Para tener un mejor entendimiento de cómo afrontar este problema, primero debemos definir qué es un algoritmo **Greedy** (o Voraz). Este tipo de algoritmo se basa en tomar decisiones óptimas **locales** en cada paso con la esperanza de que la solución final también sea óptima. Es decir, en cada iteración del proceso, se elige la mejor opción posible en ese momento, sin reconsiderar elecciones previas ni tener en cuenta elecciones futuras.

En nuestro caso, el problema consiste en determinar si un conjunto de transacciones de un sospechoso coincide con los intervalos sospechosos dados. Debido a la incertidumbre en los tiempos de los intervalos, debemos encontrar la mejor asignación de transacciones a intervalos respetando la condición de pertenencia a los mismos. Dado que los intervalos pueden superponerse, es posible que haya más de una opción válida para cada transacción, y es ahí donde entra en juego el enfoque Greedy.

3. Elección del Algoritmo

Cómo el objetivo es asignar los intervalos sospechosos a todas las transferencias sospechosas (ambas, de tam. n), debiamos buscar una manera de ordenar los intervalos para compararlos uno a uno. La idea es procesar los intervalos de menor a mayor tiempo de finalización (es decir, ordenándolos según $t_i + e_i$) e intentar asignar las transacciones una por una, asegurando que cada una pertenezca a algún intervalo válido. Esto quiere decir, en otras palabras:

$$Tiempo_i - Error_i \quad \leq \quad Transferencia_i \quad \leq \quad Tiempo_i + Error_i$$

En este trabajo, enfrentamos el problema de determinar si una serie de transferencias realizadas por un sospechoso coinciden con un conjunto de intervalos de tiempo considerados sospechosos. Para resolverlo, implementamos un enfoque basado en un algoritmo **greedy**, ya que nos permite encontrar una solución óptima de manera eficiente sin necesidad de explorar todas las combinaciones posibles.

Cómo lo mencionamos previamente, nuestra estrategia se fundamenta en **ordenar los intervalos sospechosos** de menor a mayor tiempo de finalización y luego asignar cada transferencia al primer intervalo disponible en el que pueda encajar. Esto nos brinda varias ventajas significativas:

- **Menor cantidad de comparaciones**: Al evaluar primero los intervalos con menor tiempo de finalización, se evita comparar cada transferencia con intervalos que de antemano sabemos que no pueden contenerla.
- **Mayor eficiencia en la búsqueda de coincidencias**: En lugar de verificar cada transferencia contra todos los intervalos disponibles, utilizamos un ordenamiento previo que nos permite reducir la cantidad de operaciones necesarias.
- **Complejidad computacional optimizada**: En lugar de explorar todas las combinaciones posibles, lo cual podría llevarnos a una complejidad exponencial, logramos una eficiencia de $\mathcal{O}(n \log n)$ gracias a que el paso más costoso del algoritmo es la ordenación de los intervalos, mientras que la asignación se realiza en tiempo lineal.

El siguiente fragmento de código muestra cómo ordenamos los intervalos de menor a mayor tiempo de finalización antes de comenzar la asignación de transferencias:

```
1 intervalos_sospechosos.sort(key=lambda x: x[TIEMPO] + x[ERROR]) # O(N*log(N))
```

3.1. Manejo del Caso Límite: Intervalos Solapados

Uno de los desafíos más importantes al diseñar el algoritmo fue el manejo de **intervalos solapados**, que en un primer momento llevó a decisiones incorrectas y nos obligó a reconsiderar la estrategia. Este caso límite se presenta cuando **dos o más intervalos sospechosos se superponen en el tiempo, pero no necesariamente contienen la transferencia en cuestión**.

Si simplemente recorriéramos la lista de intervalos en orden, podríamos encontrarnos con una situación en la que una transferencia no pertenece al primer intervalo evaluado, aunque sí a un intervalo posterior. En este escenario, el algoritmo original descartaría la transferencia como no perteneciente a ningún intervalo, lo cual es incorrecto.

Para solucionar este problema, modificamos el algoritmo para que, cuando una transferencia no se encuentre en el primer intervalo evaluado, **se continúe la búsqueda en los siguientes intervalos de la lista**. Si encontramos un intervalo que efectivamente contiene la transferencia, realizamos un **intercambio de posiciones** entre este intervalo y el primero evaluado.

Este intercambio es clave porque permite que las futuras transferencias aprovechen mejor los intervalos disponibles. Además, es casi seguro (por no decir seguro), que la próxima transacción utilize el intervalo intercambiado recientemente. Sin este cambio, el algoritmo simplemente descartaria el intervalo, y además, como resultado de esta negligencia, catalogaria al sospechoso como inocente.

El siguiente fragmento de código muestra esta estrategia de intercambio y búsqueda:

```
# Buscar si la transferencia coincide con algun intervalo posterior.

for j in range(i, len(intervalos_sospechosos)):
    if esta_dentro_del_intervalo(intervalos_sospechosos[j],
    transferencias_del_sospechoso[i]):

        # Intercambio los intervalos de lugar.
        intervalos_sospechosos[i], intervalos_sospechosos[j] =
    intervalos_sospechosos[j], intervalos_sospechosos[i]

        # Agrega el intervalo con la transferencia a la lista de intersecciones.
        intersecciones.append((intervalos_sospechosos[i],
    transferencias_del_sospechoso[i]))
        break
```

3.2. La Eficiencia en Términos Prácticos

A simple vista, podría parecer poco eficiente recorrer **todos** los intervalos disponibles para cada transferencia restante. Esta crítica es válida si se interpretara que el algoritmo realiza una búsqueda exhaustiva sin ninguna estrategia, lo cual iría en contra del enfoque *Greedy*. Sin embargo, nuestro planteo evita este problema desde el diseño mismo.

Comencemos analizando el caso desde un punto de vista probabilístico. ¿Qué probabilidad hay de que el siguiente intervalo (recordemos que están ordenados por tiempo de finalización ascendente) contenga a la transferencia actual? A medida que avanzamos al siguiente intervalo, la probabilidad de que *éste* sirva es cada vez mayor (o, dicho de otra forma, la probabilidad de que el intervalo no sirva disminuye).

Además, solo existen **dos casos extremos** que podrían forzar al algoritmo a recorrer múltiples intervalos, afectando su eficiencia. Sin embargo, ambos son *casos límite poco frecuentes*, y por tanto no afectan significativamente al comportamiento promedio del algoritmo.

- **Caso 1: La transferencia no pertenece a ningún intervalo.** En este caso, el algoritmo recorrerá todos los intervalos una única vez, detectará que no hay correspondencia, y detendrá la ejecución. Esto indica que el sospechoso no puede ser culpable, por lo que no tiene sentido continuar.
- **Caso 2: La transferencia pertenece únicamente al último intervalo.** Este es el peor

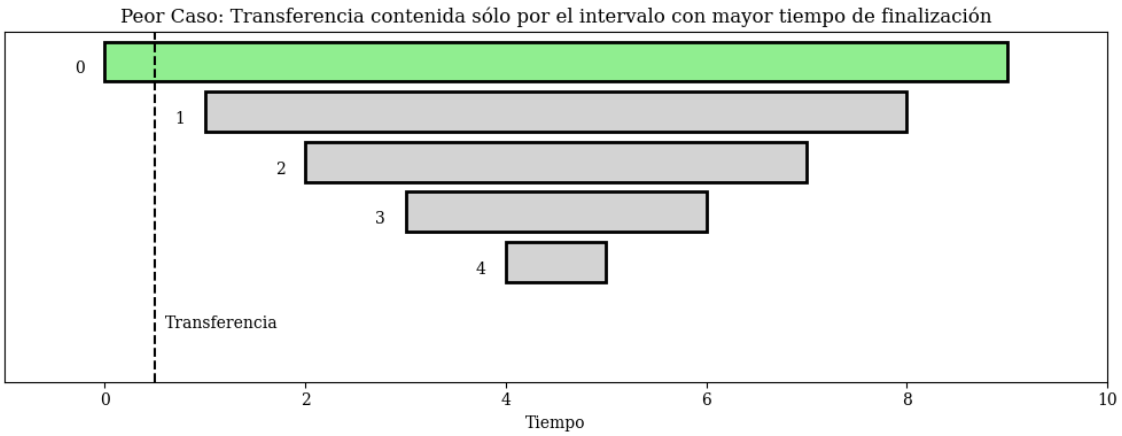


Figura 1: Ilustración del Peor y Único Caso Posible

caso posible, ya que obliga al algoritmo a descartar todos los intervalos anteriores. Sin embargo, una vez que este intervalo es usado, la probabilidad de que futuras transferencias utilicen otros intervalos se reduce drásticamente, haciendo que este patrón no se repita.

Además, el caso donde se recorren múltiples intervalos por cada transferencia no es sostenible: si una transferencia no está contenida en el primer intervalo, se pasa al siguiente y así sucesivamente, pero nunca se vuelve atrás ni se comparan múltiples intervalos con la misma transferencia sin sentido. Y si ocurre el caso contrario, en el que cada transferencia entra exactamente en el primer intervalo que le corresponde, nos encontramos ante el caso ideal.

Por lo tanto, aunque pueda parecer que el algoritmo podría caer en una complejidad cuadrática $\mathcal{O}(n^2)$, esto solo ocurre en el *peor de los casos posibles*, y de forma **única**. En la mayoría de los escenarios, especialmente en los generados aleatoriamente, el algoritmo se comporta de forma eficiente.

En la sección de *Complejidad del Algoritmo* profundizaremos sobre esta cuestión, y mostraremos que incluso en este escenario desfavorable, el rendimiento sigue siendo aceptable.

3.3. Ejemplo Visual: Reasignación de Prioridad entre Intervalos

En la Figura 2, se ilustra un caso concreto en el que el algoritmo necesita reasignar la prioridad de los intervalos para lograr una asignación válida.

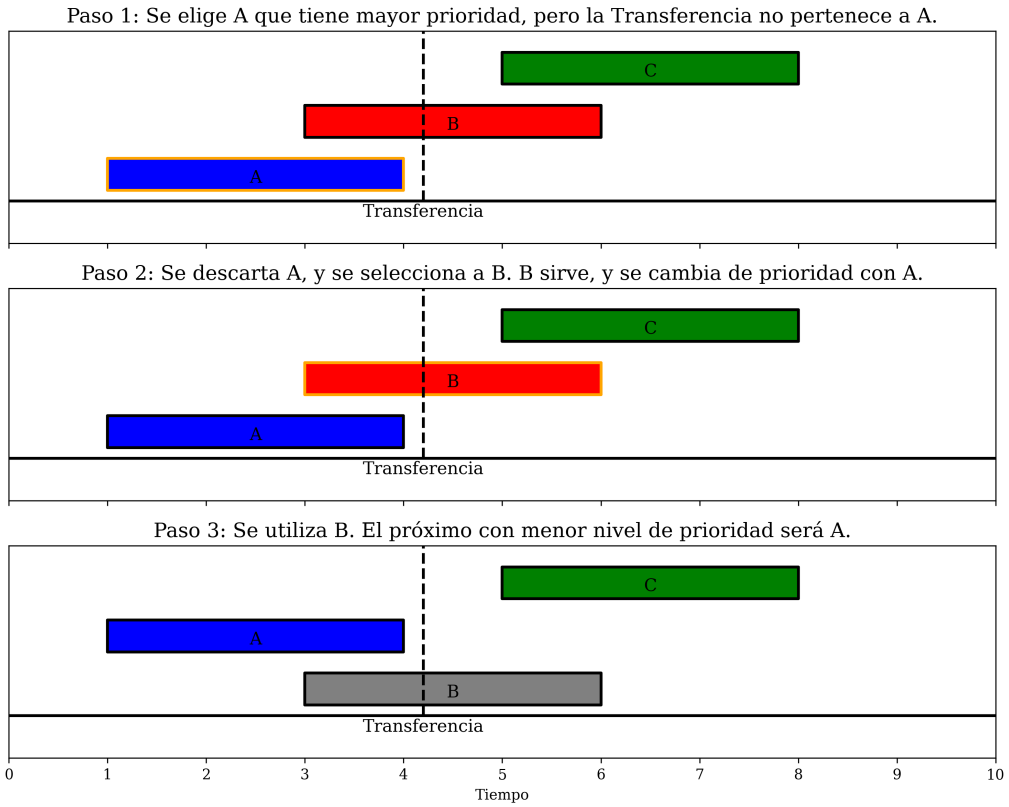


Figura 2: Ilustración del proceso de reasignación de prioridad entre intervalos.

Paso 1 – Evaluación del primer intervalo (A): El algoritmo comienza evaluando el intervalo con mayor prioridad, en este caso el intervalo **A** (color azul), como se muestra en la parte superior de la figura. Sin embargo, al verificar si la transferencia se encuentra dentro de los límites del intervalo A, se detecta que no es así. Por lo tanto, se procede a considerar los siguientes intervalos disponibles. En la práctica, siempre el próximo intervalo próximo es el mejor candidato para la transferencia actual (Al menos que sea un caso borde de 3 o más intervalos superpuestos, donde el de mayor tiempo de finalizacion es el único que contiene a la transferencia).

Paso 2 – Reasignación de prioridad con un intervalo válido (B): A continuación, se evalúa el intervalo **B** (color rojo), que sí contiene la transferencia. En ese momento, el algoritmo realiza un **intercambio de prioridad**: el intervalo B pasa a ocupar el lugar de A en la lista, asegurando que la transferencia pueda ser correctamente asignada y utilizada por las próximas transferencias de la lista.

Paso 3 – Asignación y actualización del estado: Una vez que se ha realizado el intercambio, la transferencia se asigna exitosamente al intervalo B (ahora gris, pues ya se ha usado). La siguiente iteración del algoritmo continuará desde el intervalo ahora ubicado en la siguiente posición, en este caso, A (que fue desplazado).

4. Complejidad del Algoritmo

4.1. Explicacion de la Complejidad

El algoritmo qué encuentra la transferencia sospechosa empieza ordenando todas las transferencias.

```
intervalos_sospechosos.sort(key=lambda x: x[TIEMPO] + x[ERROR]) # O(N*log(N))
```

Cómo la funcion de ordenamiento de Python *.sort()* implementa el algoritmo **TrimSort**, que combina Merge Sort e Insertion Sort, esto tiene una complejidad, en el peor caso de:

$$\mathcal{O}(n * log(n))$$

Esto se debe a que *TimSort* se basa en la técnica de división y conquista, similar a *MergeSort*, en donde se divide el problema en subproblemas de tamaño $n/2$ y luego se combinan las soluciones parciales en un costo $O(n)$, lo cual, mediante la aplicación del Teorema Maestro, resulta en una complejidad de $O(n * \log(n))$.

La siguiente parte del algoritmo, utiliza 2 for loops para verificar las intersecciones de las transferencias:

```
    for i in range(len(intervalos_sospechosos)):

        if esta_dentro_del_intervalo(intervalos_sospechosos[i],
    transferencias_del_sospechoso[i]):
            intersecciones.append((intervalos_sospechosos[i],
    transferencias_del_sospechoso[i]))



        interseccion_con_intervalo_posterior = False
        for j in range(i, len(intervalos_sospechosos)): # Solo recorremos de la
    posicion (i) actual en adelante
            if esta_dentro_del_intervalo(intervalos_sospechosos[j],
    transferencias_del_sospechoso[i]):
                # Intercambio el intervalo que me saltee (el anterior) con el usado
     (actual) -> Es 99% probable que la siguiente transferencia use ese intervalo.
                intervalos_sospechosos[i], intervalos_sospechosos[j] =
    intervalos_sospechosos[j], intervalos_sospechosos[i]
                intersecciones.append((intervalos_sospechosos[i],
    transferencias_del_sospechoso[i]))
                interseccion_con_intervalo_posterior = True
                break


        if not interseccion_con_intervalo_posterior:
            return intersecciones, False
```

En este caso, la complejidad del primer **for loop** siempre es $\mathcal{O}(n)$, ya que recorre todas las transferencias sospechosas.

Mientras que la complejidad del segundo **for loop** depende de las transferencias que coinciden. Si ninguna transferencia coincide hasta la última, el bucle interno recorrerá todas las n transferencias sospechosas desde i. Resultando en el peor caso, con una complejidad de $\mathcal{O}(n)$. Finalmente, la función utilizada dentro de los loops anteriores fue implementada de la siguiente manera:

```
def esta_dentro_del_intervalo(intervalo, transferencia):s
    inicio, fin = intervalo[TIEMPO] - intervalo[ERROR],intervalo[TIEMPO] +
    intervalo[ERROR]
    return inicio <= transferencia <= fin
```

Cabe destacar que la función `esta_dentro_del_intervalo`, utilizada dentro de los bucles, realiza un número constante de operaciones (comparaciones y sumas/restas), por lo que su complejidad es:

$$\mathcal{O}(1)$$

La complejidad total del algoritmo está dominada por los bucles anidados, ya que su complejidad $O(n^2)$ supera la complejidad del ordenamiento inicial $O(n * \log(n))$. Por lo tanto, la complejidad total en el peor caso es:

$$T(n) = O(n * \log(n)) + O(n^2) = O(n^2)$$

Es importante destacar que este análisis corresponde al peor caso posible. En la práctica, la complejidad real del algoritmo puede situarse más cerca de $O(n * \log(n))$ dependiendo de la distribución de las transferencias. *(Veáse Sección 4.2)*

Si las transferencias están bien distribuidas en el tiempo y los rangos de sospecha son pequeños, es probable que la mayoría de las coincidencias se encuentren rápidamente dentro del primer bucle, reduciendo significativamente las iteraciones del bucle interno.

Por otro lado, si las transferencias están muy solapadas o los rangos de sospecha son grandes, el algoritmo podría comportarse más cercano a su peor caso $O(n^2)$.

Por lo tanto, el comportamiento real del algoritmo se encuentra entre:

$$\mathcal{O}(n * \log(n)) \quad \text{y} \quad \mathcal{O}(n^2)$$

Dependiendo de las características de los datos de entrada.

4.2. Prueba de la Complejidad con Casos Pseudo-Aleatorios

Si las transferencias están bien distribuidas en el tiempo y los intervalos de sospecha son pequeños, es probable que cada transferencia del sospechoso tenga una coincidencia más cercana al inicio en la lista ordenada de transferencias sospechosas. En este caso, el bucle interno se ejecutará pocas veces, disminuyendo la complejidad total del algoritmo a $\mathcal{O}\left(n * log(n)\right)$

Por otro lado, si las transferencias sospechosas tienen intervalos grandes o están muy solapadas, una misma transferencia del sospechoso podría necesitar ser comparada con múltiples transferencias sospechosas antes de encontrar una coincidencia,o incluso, comparar con todas y no encontrar ninguna. En este caso, el bucle interno podría ejecutarse en el peor de los casos hasta $\mathcal{O}\left(n\right)$ veces por cada iteración del bucle externo, llevando la complejidad total a $\mathcal{O}\left(n^2\right)$

Entonces, la distribución y el solapamiento de los rangos de tiempo de las transferencias son variables con un gran impacto en la eficiencia del algoritmo. El algoritmo propuesto busca emparejar cada intervalo con su transferencia correspondiente y verificar si está contenida dentro de los márgenes de error. Para mejorar el tiempo de ejecución, los intervalos se ordenan por su tiempo de finalización. Esto permite reducir la cantidad de comparaciones necesarias en promedio, ya que se garantiza que, si un intervalo no contiene la transferencia, entonces ningún intervalo anterior en la lista ordenada lo hará.

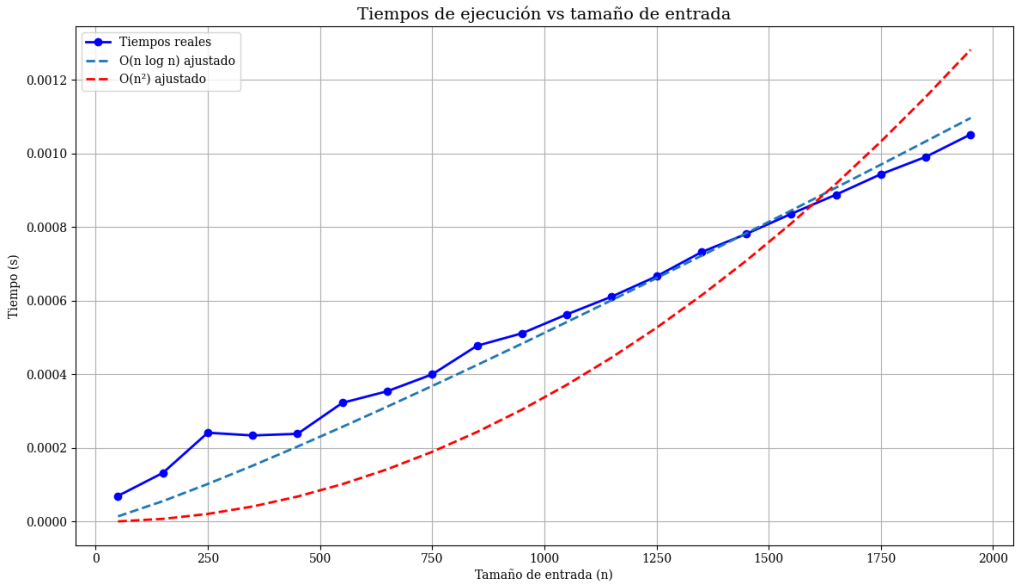


Figura 3: Gráfico de Complejidad con Casos Pseudo-Aleatorios.

Cómo se ve en el gráfico, el algoritmo tiene una tendencia de complejidad loglineal $\mathcal{O}\left(n * log(n)\right)$. Este experimento, fue hecho con semillas diferentes en cada caso, y variando el numero de las muestras para asegurar la máxima aleatoriedad. Es decir, cómo lo afirmabamos desde un comienzo (al momento de optar por este algoritmo o no), en la práctica, podemos afirmar con total seguridad que la complejidad promedia del algoritmo es loglineal, y no cuadratica.

4.3. Prueba de la Complejidad - Tipo Peor Caso

Para comprobar la complejidad del peor caso, procedimos a evaluar el tiempo de ejecución del algoritmo, dónde cada caso son del tipo **Peor Caso**(Véase Figura 1).

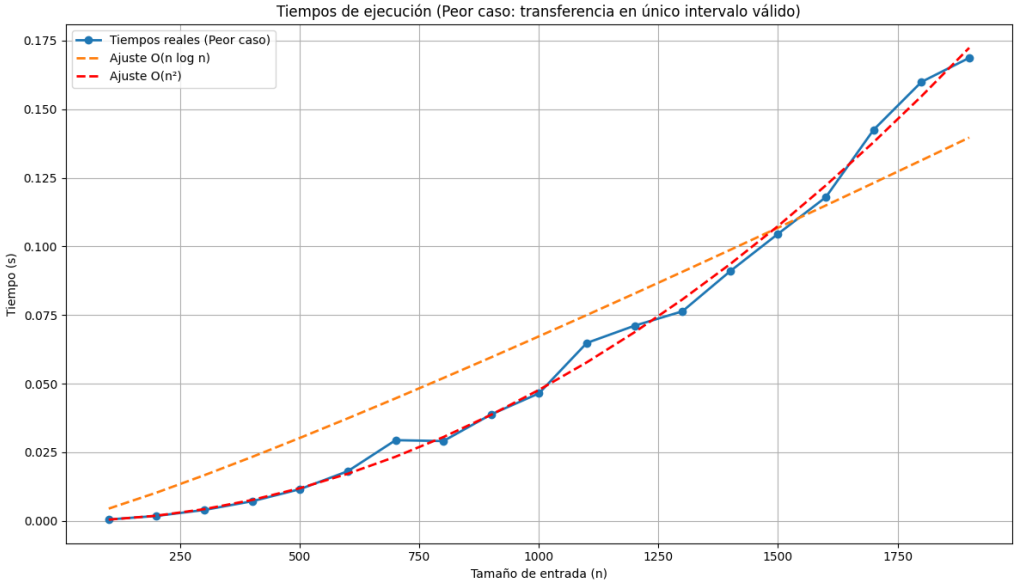
Para ello, creamos una función que dado un valor de N(enviado por la semilla), genera los N casos donde la $Transferencia[i]$ pertenece al $Intervalo[-1-i]$. En otras palabras, la **primer** transferencia solo pertenece al **último** intervalo *(Que tiene mayor tiempo de finalización)*, la **segunda** transferencia pertenece al **anteultimo** intervalo, y asi sucesivamente hasta la transferencia **N**

```python
def generar_peor_caso_ordenado(n, margen=5.0):
    intervalos = []
    transferencias = []
    for i in range(n):
        tiempo_central = 5 + i * 0.5
        error = margen
        intervalos.append((tiempo_central, error))
    intervalos.sort(key=lambda x: x[TIEMPO] + x[ERROR], reverse=True)
    for i in range(n):
        inicio = intervalos[i][TIEMPO] - intervalos[i][ERROR]
        fin = intervalos[i][TIEMPO] + intervalos[i][ERROR]
        transferencia = random.uniform(inicio + 0.01, fin - 0.01)
        # aseguramos que no est  en intervalos posteriores
        for j in range(i+1, len(intervalos)):
            inicio_otro = intervalos[j][TIEMPO] - intervalos[j][ERROR]
            fin_otro = intervalos[j][TIEMPO] + intervalos[j][ERROR]
            while inicio_otro <= transferencia <= fin_otro:
                transferencia = random.uniform(inicio + 0.01, fin - 0.01)
        transferencias.append(transferencia)
    return intervalos[::-1], transferencias  # se invierte para que queden en orden
     original.

```

```python
def generar_peor_caso_ordenado(n, margen=5.0):
    intervalos = []
    transferencias = []
    for i in range(n):
        tiempo_central = 5 + i * 0.5
        error = margen
        intervalos.append((tiempo_central, error))
    intervalos.sort(key=lambda x: x[TIEMPO] + x[ERROR], reverse=True)
    for i in range(n):
        inicio = intervalos[i][TIEMPO] - intervalos[i][ERROR]
        fin = intervalos[i][TIEMPO] + intervalos[i][ERROR]
        transferencia = random.uniform(inicio + 0.01, fin - 0.01)
        # aseguramos que no esta en intervalos posteriores
        for j in range(i+1, len(intervalos)):
            inicio_otro = intervalos[j][TIEMPO] - intervalos[j][ERROR]
            fin_otro = intervalos[j][TIEMPO] + intervalos[j][ERROR]
            while inicio_otro <= transferencia <= fin_otro:
                transferencia = random.uniform(inicio + 0.01, fin - 0.01)
        transferencias.append(transferencia)
    return intervalos[::-1], transferencias  # se invierte para que queden en orden
     original

```

Veamos, que luego entra en bucle *while* para comprobar que la transferencia no está dentro de los intervalos restantes disponibles. Cómo utilizamos `random.uniform()`, el bucle se vuelve a repetir hasta que random acierte el valor de la transferencia ideal para que solo esté únicamente dentro del último intervalo disponible, y no en otro.

Al ejecutar la prueba, podemos ver lo que esperabamos obtener: **Una complejidad cuadrática**, en dónde, sí el valor de N es pequeño, aún es mas eficiente que la complejidad loglineal. Pero sívemos detenidamente, luego de $N > 1500$, la complejidad cuadrática empieza ascender con mayor velocidad que la loglineal. En conclusión, el algoritmo tendrá mayor eficiencia en su caso promedio, aunque tambien dependerá de los datos ingresados. Lo único que podemos asegurar con certeza, es que la complejidad estará en un rango de complejidad entre $\mathcal{O}(n * \log(n))$ y $\mathcal{O}(n^2)$



Complejidad donde todos los casos son del tipo Peor Caso.