Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

Trabajo Práctico 1
Algoritmo Greedy

10 de Abril de 2025

Facundo Barrasso
111942

Mateo Nowenstein
112063

Juana Rehl
112185

1. Introducción

Un algoritmo Greedy tiene como objetivo dividir un problema en subproblemas, seleccionando una solución óptima local esperando a que nos acerque a la solución óptima global. Esta solución se busca aplicando iterativamente una regla sencilla que solo se enfoca en el estado actual de los elementos del problema, utilizando el conjunto de datos en el estado en que llegaron.

Este trabajo tiene como objetivo ampliar nuestros conocimientos de los algoritmos greedy. Mediante un problema planteado, verificar las posibles soluciones y determinar mediante un exhaustivo análisis, cuál es la solución que siempre detecta lo pedido y demostrar por qué lo sería.

1.1. Consigna

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 condicen, que encuentra la asignación, y si no condicen, 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

En el presente trabajo práctico se nos propone solucionar de forma óptima, mediante un algoritmo Greedy, el problema presentado: encontrar rápidamente la rata que robó dinero a la mafia de los amigos Amarilla Pérez y el Gringo Hinz, mediante las transacciones realizadas por el sospechoso y los timestamps (rango de transacciones sospechosas) aproximados.

Para esto contamos con información brindada por la mafia: n, la cantidad de transacciones sospechosas, de las cuales cada transacción tiene un timestamp aproximado como tiempos t_i con un posible error e_i, que nos determina el tiempo aproximado en el cuál fue realizada la transacción $[t_i - e_i, t_i + e_i]$. Además, nos brindan una lista de transacciones realizadas por una persona sospechosa, de las cuales hay n transacciones pero debemos verificar que estén dentro de los timestamps.

2.1 Análisis del Algoritmo Greedy Elegido

Analizado el problema propuesto, llegamos a la conclusión de que obligatoriamente se tienen que recorrer todas las transacciones del sospechoso junto con todas las transacciones sospechosas, verificando que haya un intervalo de transacción fraudulenta que corresponda a cada transacción realizada por el sospechoso.

Para lograr esto de una forma eficiente y asegurando la correcta asignación de tareas, decidimos ordenar las transferencias sospechosas por el tiempo de fin del intervalo más cercano. Además, esta decisión permite que los tiempos de transferencia menores del sospechoso se asignen a los intervalos que finalizan antes, dejando así a aquellos que ocurrieron después para aquellos intervalos que cuentan con un tiempo de finalización posterior.

Demostración de optimalidad (por método directo)

Hipótesis

Caso A (existe asignación válida): Existe una asignación inyectiva $f : S \to T$ tal que, para cada timestamp sospechoso $S_j \in S$, existe una transacción $(T_i, E_i) \in T$ con la condición:

$$T_i - E_i \leq S_j \leq T_i + E_i$$

Además, cada transacción se usa a lo sumo una vez.

Caso B (no hay asignación): Si para un timestamp $S_j \in S$, no se encuentra ninguna transacción $(T_i, E_i) \in T$ que cumpla la condición mencionada anteriormente, no hay tal asignación.

Tesis

El algoritmo Greedy propuesto se encarga de ordenar las transacciones en función de $T_i + E_i$, es decir, el tiempo de fin del intervalo. Para cada timestamp del sospechoso, selecciona la primera transacción disponible que cumpla la condición. Se cumple lo siguiente:

Si existe tal asignación válida, el algoritmo encuentra la asignación completa. Es decir, empareja cada S_j con una transacción (T_i, E_i).

Si no existe asignación válida, el algoritmo detecta la imposibilidad de asignar alguno de los timestamps con las transacciones y concluye que no se puede hacer una asignación válida, por lo que no es el sospechoso.

Demostración del Caso A

Asumamos que existe una asignación válida f. Entonces, para cada S_j hay al menos una transacción (T_i, E_i) en f que satisface la condición. El algoritmo recorre la lista de transacciones en orden creciente de $T_i + E_i$, por lo que siempre elegirá la primera transacción válida para S_j. Como está ordenado por el criterio de menor $T_i + E_i$, esto garantiza que:

- Se seleccione la transacción con el menor tiempo de finalización posible, evitando que una transacción válida se desperdicie innecesariamente.
- Se maximice la disponibilidad de transacciones restantes para los siguientes timestamps S_j.

Si en algún paso el algoritmo no encuentra transacción, esto implicaría que para ese S_j no existe ninguna transacción en T que cumpla la condición. Sin embargo, esto contradice la hipótesis de que f es una asignación válida. Por lo tanto, el algoritmo garantiza la asignación correcta cuando existe una solución.

Demostración del Caso B

Si para algún timestamp S_j no hay una transacción (T_i, E_i) que cumpla la condición, el algoritmo detectará que no es el sospechoso, ya que al no coincidir con ningún intervalo, quedaría un timestamp sin definir y no habría coincidencia válida.

Al finalizar el recorrido para S_j sin haber realizado alguna asignación (en este caso sería el `break` del ciclo interno), el algoritmo entra directamente en el `else` terminando la iteración. Indica de forma directa que el algoritmo detecta la ausencia de una transacción válida, concluyendo correctamente que no es posible realizar la asignación completa, por ende, no es el sospechoso correcto.

Conclusión

El algoritmo Greedy propuesto garantiza que:

- Si existe una asignación válida, la encuentra en todos los casos.
- Si no existe una asignación válida, detecta esta imposibilidad correctamente.

Por lo tanto, el algoritmo es correcto y siempre determina si los timestamps del sospechoso corresponden a los intervalos sospechosos.

3. Análisis de Complejidad

- Método matchear de la clase robo

```
1 def matchear(self):
2     self.transacciones.sort(key=lambda x: x[0] + x[1])
```

Este método se encarga de encontrar las transacciones entre las transferencias y las hechas por el sospechoso. Sabemos que self.transacciones tiene un tamaño de n elementos (n transacciones). Ordenar estas nos cuesta O(n log n) ya que es la mejor complejidad posible para un algoritmo de ordenamiento comparativo.

- Asignación de transacciones

```
for transaccion_sospechoso in self.sospechoso:
    for transaccion in self.transacciones:
        if self.dic_transacciones[transaccion] > 0:
            self.dic_transacciones[transaccion] -= 1
            break
    else:
        break
```

Tenemos un bucle en el cual primeramente recorremos las transacciones del sospechoso, que tendrá una longitud de n elementos. Dentro de este, recorremos todas las transacciones. Luego accedemos y cambiamos valores del diccionario, lo cual es una operación $O(1)$. Por lo tanto, la complejidad de los ciclos `for` anidados es $O(n^2)$.

- Imprimir los matches

```
for match in self.matches.keys():
    sospechoso, transaccion = match
    while self.matches[match] > 0:
        self.matches[match] -= 1
        print(f"{sospechoso} --> {transaccion[0]} +- {transaccion[1]}")
```

En esta función, se imprimen los matches entre los timestamps y las transacciones del sospechoso, solo en el caso de que efectivamente sea el sospechoso. Como sabemos que los *matches* equivalen a la cantidad de transacciones del sospechoso, imprimir estas mismas nos cuesta $O(n)$.

- Procesamiento de datos

```
def procesar_datos(ruta):
    with open(ruta, "r") as archivo:
        # Procesamiento inicial del archivo
        for _ in range(cantidad):
            # Procesamiento de cada transaccion, O(1) por transaccion

        for _ in range(cantidad):
            # Procesamiento de cada sospechoso, O(1) por sospechoso
    return robo
```

En este caso se analizan las transacciones y las del sospechoso, y como sabemos que tienen igual tamaño (n elementos cada una), podemos concluir que la complejidad es:

$$O(n) + O(n) = O(2n) = O(n)$$

3.1. Complejidad total

Finalmente, en la función principal integramos todas las complejidades a la vez. Por lo que la complejidad total sería:

$$O(n \log n) + O(n^2) + O(n)$$

Dado que la parte cuadrática es la dominante, la complejidad final queda:

$$O(n^2)$$

3.2. Variabilidad de los valores

Los valores más importantes que pueden variar son:

- La cantidad de transacciones.
- La distribución de los momentos en los cuales el sospechoso hizo una transacción.
- Los intervalos de las transacciones.

Si los valores de los timestamps y errores tienen una gran variabilidad, el ordenamiento inicial no se ve afectado significativamente, ya que la complejidad del algoritmo de ordenamiento no depende de la distribución de los valores, sino únicamente de la cantidad de elementos.

La distribución de los timestamps del sospechoso afecta cuántas transacciones deben ser evaluadas por cada timestamp:

- Si los timestamps están muy concentrados en un rango pequeño, es más probable que coincidan en múltiples intervalos, lo que aumenta el número de comparaciones necesarias.
- Si los timestamps están distribuidos, es menos probable que coincidan con muchos intervalos, reduciendo el número de comparaciones.

Si hay transacciones repetidas (múltiples entradas con el mismo timestamp y error), el algoritmo debe verificar y reducir el contador del diccionario de transacciones para cada coincidencia:

- Si hay muchas transacciones repetidas, el algoritmo matchea a las transacciones sospechosas en menos iteraciones. Esto es más eficiente ya que se accede al diccionario con claves frecuentes.
- Si las transacciones son únicas, el tiempo de ejecucion aumenta ya que no se puede "reutilizar"las claves como en el caso anterior.

El tamaño de los intervalos afecta cuántos timestamps del sospechoso pueden coincidir con una transacción específica:

- Si los valores de error son pequeños, los intervalos serán más acotados, lo que reduce la probabilidad de coincidencias y, por lo tanto, el número de comparaciones.
- Si los valores de error son más grandes, los intervalos serán más amplios, aumentando la probabilidad de coincidencias y el número de comparaciones necesarias.

Finalmente, se puede decir que una alta variabilidad en los valores tiende a reducir el número de comparaciones necesarias, ya que los intervalos serán más dispersos y menos timestamps coincidirían con múltiples intervalos. En cambio, una baja variabilidad aumentaría las superposiciones y, por consecuencia, las comparaciones necesarias.

3.3. Análisis de la variabilidad en los tiempos de ejecución según el tipo de entrada

En el siguiente gráfico se muestra el tiempo promedio de ejecución para los diferentes tipos de test enfocado en la variabilidad de los valores. En general, se observa que los tiempos se mantienen en un rango acotado, lo que sugiere una buena estabilidad en el rendimiento.

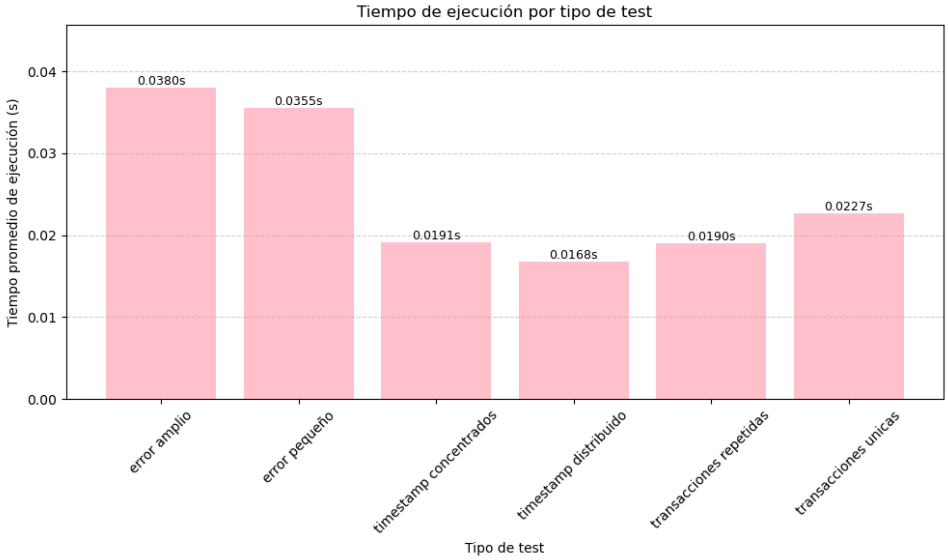

Figura 1: Tiempo promedio de ejecución según la variabilidad de los valores

Este análisis de rendimiento frente a distintos tipos de entrada nos indica que el algoritmo responde de manera eficiente y estable ante la mayoría de los escenarios. Sin embargo, se observan pequeñas optimizaciones en algunos casos como transacciones repetidas o timestamps concentrados. Así como también se observan ligeras penalizaciones en escenarios con errores amplios o datos concentrados.

4. Mediciones de tiempos para corroborar la complejidad teórica

4.1. Ajuste

Una vez comprobado que la complejidad es $\mathcal{O}(n^2)$, nos dispusimos a realizar los diferentes tests para verificar que, efectivamente, en la práctica se aproxima a una función cuadrática de la forma:

$$f(x) = 6{,}46 \cdot 10^{-13}x^2 + 1{,}1728 \cdot 10^{-6}x + 3{,}0384 \cdot 10^{-2}$$

Además, para realizar dichos tests creamos nuestros propios sets de datos para mayor precisión en el análisis. En ellos, los sets de datos varían de un rango de 10 hasta 1.000.000 transferencias, asegurándonos que el tiempo que tarda crece cuadráticamente en relación a la cantidad de transferencias en cada set.

Finalmente, para obtener la mayor precisión posible en cuanto al tiempo estimado de ejecución de cada set, corrimos cada test 20 veces con los mismos sets de datos, permitiendo la mayor equidad posible con respecto a la variabilidad de los valores entre sets y ejecuciones.

Para lograr esto, decidimos crear cada set de datos de manera aleatoria, es decir, eligiendo los intervalos y las transferencias dentro de un rango determinado según la cantidad de las mismas. Luego, como se mencionó anteriormente, se ejecutaron estas pruebas en un archivo auxiliar llamado `pruebas_mediciones.py`, en el que se ejecutaba `tp1.py` junto con el nombre del archivo enviado como parámetro.

En lo que respecta a la aproximación por cuadrados mínimos, esta la realizamos con ayuda de las bibliotecas de Python `Matplotlib` y `Numpy`, las cuales proveen una gran cantidad de herramientas para trabajar con matrices y gráficos. Al principio realizamos tests con menos sets de datos, de los

cuales no había una gran variabilidad de cantidad de elementos, por lo que la función cuadrática esperada no se llegaba a apreciar de forma correcta. Por eso, decidimos aumentar la cantidad de elementos en cada test para así lograr una mayor "curvatura" en la función, logrando comprobar que efectivamente la complejidad del algoritmo es de $\mathcal{O}(n^2)$.

En el gráfico generado, se puede observar:

- **Puntos rojos**: tiempos promedio de los distintos sets de datos junto con la cantidad de elementos del mismo.
- **Línea azul**: curva ajustada por mínimos cuadrados, que representa una función cuadrática.

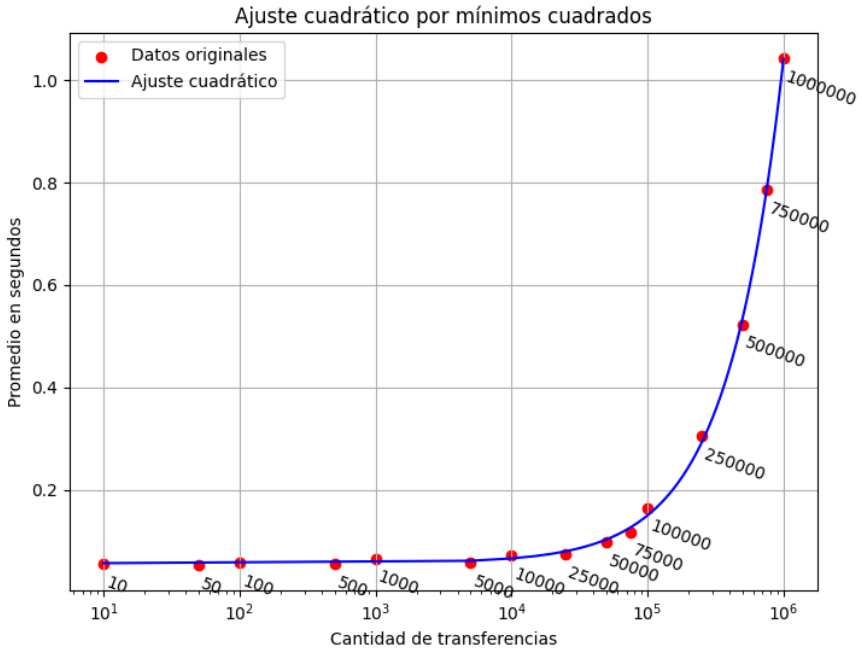

Figura 2: Curva ajustada por mínimos cuadrados vs. tiempos de ejecución

4.1.1. Comportamiento

Estos resultados no solo confirman la complejidad teórica del algoritmo, sino que también validan su comportamiento empírico. Esto refuerza la importancia de combinar análisis teóricos con experimentación para evaluar correctamente el rendimiento de un programa.

4.2. Error de ajuste

El error de ajuste representa la diferencia absoluta entre los valores reales observados y los valores predichos por el modelo cuadrático ajustado. En este caso, el modelo cuadrático se ajusta correctamente a los datos, como se observa en la figura 3 con valores de error mínimos.

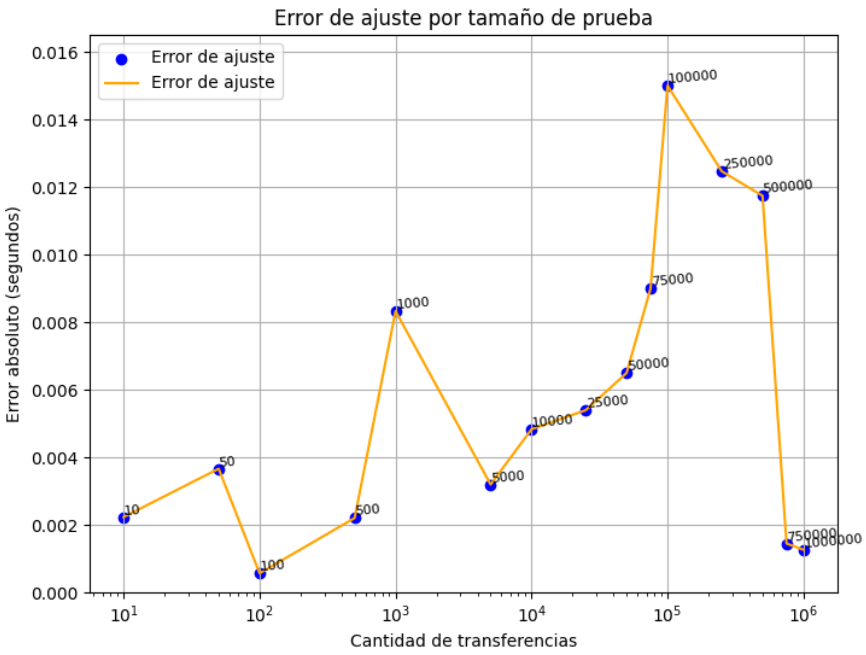

Figura 3: Tendencia del error de ajuste cuadrático

4.2.1. Interpretación del Gráfico

En el gráfico, el eje X muestra la cantidad de transferencias por prueba realizada (en escala logarítmica), mientras que el eje Y representa el error absoluto en segundos.

- **Puntos azules**: indican los errores individuales para cada tamaño de prueba
- **Línea naranja**: conecta estos puntos para visualizar la tendencia general del error.
- **Tendencia General**: El error de ajuste es bajo en la mayoría de los casos, lo que indica que el modelo cuadrático describe adecuadamente los datos.
- **Escala Logarítmica**: La escala logarítmica en el eje X permite observar cómo el error varía en diferentes órdenes de magnitud de las transferencias. Esto es útil para identificar patrones en tamaños de prueba pequeños y grandes con facilidad.
- **Picos de Error**: Los picos más altos en el gráfico podrían deberse a variaciones en los datos originales o, como las diferencias son tan pequeñas, a variaciones aleatorias en la ejecución de las pruebas.
- **Error Cuadrático Total**: En esta ejecución el error cuadrático total (suma del cuadrado del error en cada punto) fue de 0.00010926367096483172 $segundos^2$. Aunque no es directamente interpretable en la misma escala que los datos originales, es útil para comparar ajustes o evaluar la magnitud relativa de los errores. Además, al tener un valor bajo, complementa a la afirmación de que el ajuste cuadrático es correcto.

4.2.2. Significado

El error de ajuste es una métrica clave para evaluar la calidad del modelo. Un error mínimo, tal como se percibe en la mayoría de los puntos, señala que el modelo cuadrático es apropiado para explicar la correlación entre la cantidad de transferencias y el tiempo medido. No obstante,

los picos de error evidencian la relevancia de examinar minuciosamente los datos para detectar potenciales fuentes de variabilidad o considerar modificaciones extra en el modelo si se requieren.

En conclusión, el modelo cuadrático es preciso y describe correctamente los datos generales, sin embargo, los picos de error indican que podrían existir factores adicionales que afecten los datos y que el modelo no está capturando.

5. Conclusiones

A lo largo del trabajo se abordó un problema de asignación utilizando una estrategia Greedy. Esto se llevó a cabo con el objetivo de determinar si los timestamps del sospechoso coincidían con los intervalos de las transacciones fraudulentas. Se planteó un algoritmo eficiente que mediante un ordenamiento por tiempo de finalización y una asignación inyectiva.

El algoritmo Greedy propuesto permite determinar de forma eficiente y correcta si los timestamps del sospechoso coinciden con los intervalos sospechosos. Garantiza encontrar una asignación válida si existe, y detectar su ausencia en caso contrario, cumpliendo con los objetivos del trabajo y demostrando la aplicabilidad del enfoque voraz al problema planteado.

Finalmente, las pruebas empíricas y el análisis de tiempos confirman la complejidad teórica planteada, reflejando un comportamiento coherente con el diseño del algoritmo. Este trabajo no solo fortalece el entendimiento de los algoritmos Greedy, sino que también evidencia su aplicabilidad a problemas concretos y relevantes.

6. Anexo

6.1. Demostración de Optimalidad (por método del absurdo)

Consideramos que el método directo no era el más adecuado para la justificación que proponíamos, por lo que decidimos que la mejor opción es el método del absurdo.

- Hipótesis

Caso A (existe asignación válida): Existe una asignación inyectiva

$$f : S \to T$$

tal que, para cada timestamp sospechoso $S_j \in S$, existe una transacción $(T_i, E_i) \in T$ con la condición:

$$T_i - E_i \leq S_j \leq T_i + E_i$$

Además, cada transacción se usa a lo sumo una vez.

Caso B (no hay asignación válida): Si existe al menos un timestamp $S_j \in S$ para el cual no hay ninguna transacción $(T_i, E_i) \in T$ que satisfaga la condición anterior, entonces no existe una asignación válida.

- Tesis

El algoritmo **Greedy** propuesto se encarga de ordenar las transacciones en función de $T_i + E_i$ (el tiempo de finalización del intervalo). Para cada timestamp del sospechoso, selecciona la primera transacción disponible que cumpla la condición. Queremos demostrar que:

- Si existe una asignación válida, el algoritmo encuentra dicha asignación.
- Si no existe una asignación válida, el algoritmo detecta correctamente esta imposibilidad.

- Demostración del Caso A (Si existe una asignación válida, el algoritmo la encuentra)

Asumamos que existe una asignación válida f. Entonces, para cada S_j hay al menos una transacción (T_i, E_i) en f que satisface la condición. El algoritmo recorre la lista de transacciones en orden creciente de $T_i + E_i$, por lo que siempre elegirá la primera transacción válida para S_j. Esta estrategia garantiza que:

- Se elige siempre la transacción cuyo intervalo termina más temprano, lo que evita "ocupar" transacciones que podrían ser necesarias para timestamps posteriores.
- Se maximiza la disponibilidad de transacciones restantes para los siguientes timestamps S_j.

Supongamos ahora que, a pesar de que existe una asignación válida f, el algoritmo falla en asignar una transacción a algún S_j. Esto implicaría que no hay ninguna transacción restante que cumpla la condición para dicho S_j, lo cual contradice la hipótesis de que f es una asignación válida. Por lo tanto, la suposición de que el algoritmo falla es falsa y, si existe una asignación válida, el algoritmo necesariamente la encuentra.

- Demostración del Caso B (Si no existe una asignación válida, el algoritmo lo detecta)

Supongamos que no existe una asignación válida. Es decir, hay al menos un S_j para el cual ninguna transacción cumple la condición del intervalo. Al finalizar el recorrido sin haber asignado una transacción a dicho S_j, el algoritmo detecta directamente la ausencia de una transacción válida. Esto implica que no es posible realizar una asignación completa de todos los timestamps sospechosos, por lo que el algoritmo concluye correctamente que el sospechoso no corresponde con las transacciones.

- Conclusión

El algoritmo Greedy propuesto garantiza que:

- Si existe una asignación válida, la encuentra en todos los casos.
- Si no existe una asignación válida, detecta esta imposibilidad correctamente.

Por lo tanto, el algoritmo es correcto y siempre determina si los timestamps del sospechoso corresponden a los intervalos definidos en las transacciones.

6.2. Mediciones de tiempos para corroborar la complejidad teórica

6.2.1. Ajuste

Una vez comprobado que la complejidad es $\mathcal{O}(n^2)$, nos dispusimos a realizar los diferentes tests para verificar que, efectivamente, en la práctica se aproxima a una función cuadrática de la forma:

$$f(x) = 7{,}686 \cdot 10^{-14}x^2 + 8{,}846 \cdot 10^{-7}x + 5{,}133 \cdot 10^{-2}$$

Además, para realizar dichos tests creamos nuestros propios sets de datos, asegurándonos de que fueran lo suficientemente variados para obtener un análisis preciso. En ellos, los sets de datos varían de un rango de 10 hasta 1.000.000 transferencias, asegurándonos que el tiempo que tarda crece cuadráticamente en relación a la cantidad de transferencias en cada set.

Finalmente, para obtener la mayor precisión posible en cuanto al tiempo estimado de ejecución de cada set, corrimos cada test 20 veces con los mismos sets de datos, permitiendo la mayor equidad posible con respecto a la variabilidad de los valores entre sets y ejecuciones.

Para lograr esto, decidimos crear cada set de datos de manera aleatoria, es decir, eligiendo los intervalos y las transferencias dentro de un rango determinado según la cantidad de las mismas. Luego, como se mencionó anteriormente, se ejecutaron estas pruebas en un archivo auxiliar llamado `pruebas_mediciones.py`, en el que se ejecutaba `tp1.py` junto con el nombre del archivo enviado como parámetro.

En lo que respecta a la aproximación por cuadrados mínimos, esta la realizamos con ayuda de las bibliotecas de Python `Matplotlib` y `Numpy`, las cuales proveen una gran cantidad de herramientas para trabajar con matrices y gráficos. Al principio realizamos tests con menos sets de datos, de los cuales no había una gran variabilidad de cantidad de elementos, por lo que las funciónes esperadas no se llegaban a apreciar de forma correcta. Por eso, decidimos aumentar la cantidad de elementos en cada test para así lograr una mayor "curvatura" en las funciónes, logrando comprobar que efectivamente la complejidad del algoritmo es de $\mathcal{O}(n^2)$.

En el gráfico generado, se puede observar:

- **Puntos rojos**: tiempos promedio de los distintos sets de datos junto con la cantidad de elementos del mismo.
- **Línea azul**: curva ajustada por mínimos cuadrados, que representa una función cuadrática.
- **Línea verde**: curva ajustada por mínimos cuadrados, que representa una función cúbica.
- **Comportamiento general**:
  - **ajuste cuadrático**: sigue de manera más consistente la tendencia de los puntos rojos en todo el rango de datos, desde tamaños pequeños hasta grandes.
  - **ajuste cúbico**: muestra mayores desviaciones en algunos intervalos, especialmente en los extremos, lo que indica que el modelo cúbico podría estar sobreajustando los datos.
- **Simplicidad del modelo**: La función cuadrática es más simple que la cúbica, lo que la hace más adecuada para describir una relación que ya se sabe teóricamente que es $\mathcal{O}(n^2)$. El modelo cúbico introduce un término adicional (x^3) que no aporta mejoras al ajuste.

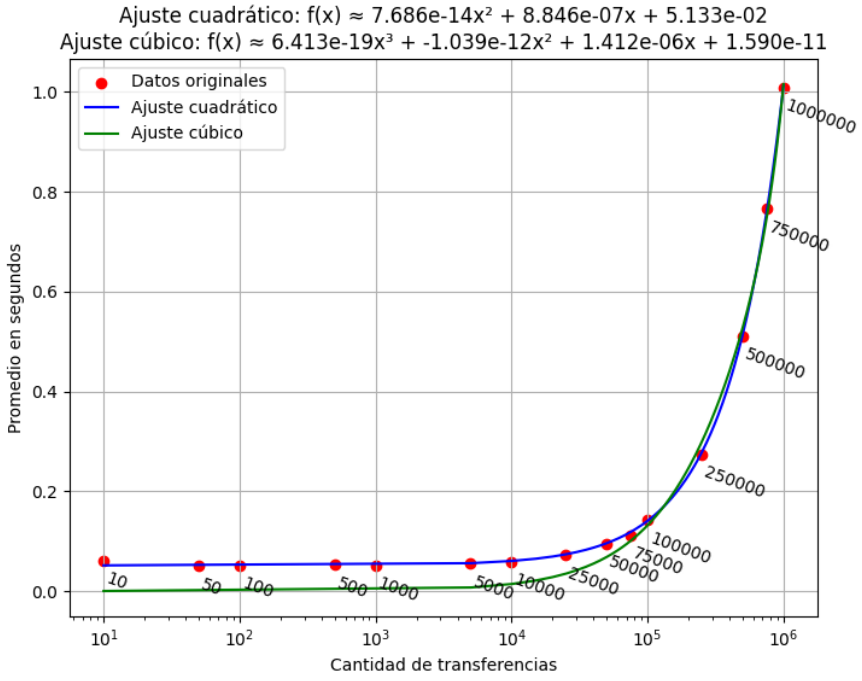

Figura 4: Curvas ajustadas por mínimos cuadrados vs. tiempos de ejecución

Comportamiento: Los resultados obtenidos no solo confirman la complejidad teórica del algoritmo, sino que también validan su comportamiento empírico. La curva cuadrática ajustada describe adecuadamente los datos, mientras que la curva cúbica, aunque más compleja, no mejora significativamente el ajuste y presenta errores mayores en algunos puntos. Esto refuerza la importancia de combinar análisis teóricos con experimentación para evaluar correctamente el rendimiento de un programa.

6.2.2. Error de ajuste

El error de ajuste mide la diferencia absoluta entre los valores reales observados y los valores predichos por los modelos ajustados. En este caso, se evaluaron dos modelos: uno cuadrático y otro cúbico. Como se observa en el gráfico **"Error de ajuste cuadrático vs. cúbico"**, el modelo cuadrático presenta un error significativamente menor en comparación con el cúbico, lo que confirma que el ajuste cuadrático es más adecuado para describir los datos.

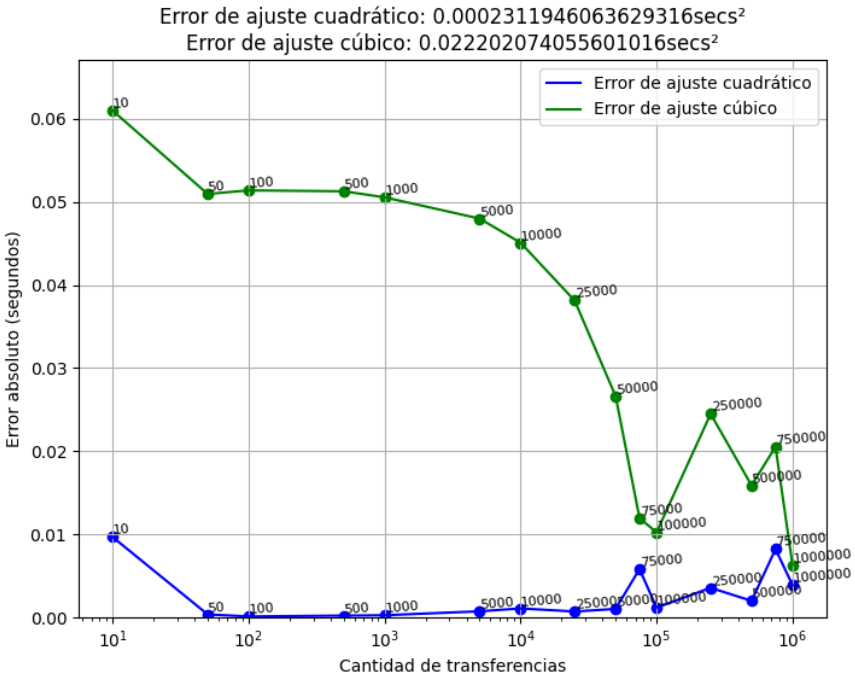

Figura 5: Error de ajuste cuadrático vs. cúbico

Interpretación del Gráfico: En el gráfico, el eje X muestra la cantidad de transferencias por prueba realizada (en escala logarítmica), mientras que el eje Y representa el error absoluto en segundos.

- **Puntos azules**: Representan los errores individuales para cada tamaño de prueba en el ajuste cuadrático.
- **Puntos verdes**: Representan los errores individuales para el ajuste cúbico.
- **Líneas conectadas**: Visualizan la tendencia general del error para ambos modelos.
- **Tendencia General**: El error de ajuste cuadrático es bajo en la mayoría de los casos, lo que indica que el modelo cuadrático describe adecuadamente los datos. Por otro lado, el ajuste cúbico presenta errores más altos y una mayor variabilidad, especialmente en tamaños de prueba grandes, lo que sugiere que no es el modelo más adecuado para estos datos.

- **Picos de Error**: En el ajuste cúbico, se observan picos de error más pronunciados, lo que podría deberse a que el modelo cúbico sobreajusta los datos o no captura correctamente la relación subyacente. En contraste, el ajuste cuadrático mantiene un error bajo y consistente, con picos menores que podrían atribuirse a variaciones aleatorias en los datos o en la ejecución de las pruebas.

Comparación del Error Cuadrático Total: El error cuadrático total, que es la suma del cuadrado de los errores individuales, aunque no es directamente interpretable en la misma escala que los datos originales, es útil para comparar ajustes o evaluar la magnitud relativa de los errores:

- **Error del ajuste cuadrático**: $0{,}0002311946063629316\,\text{secs}^2$.
- **Error del ajuste cúbico**: $0{,}02222074055601016\,\text{secs}^2$.

Estos valores refuerzan que el modelo cuadrático es significativamente más preciso que el cúbico, ya que su error cuadrático total es mucho menor. Además, al tener un valor bajo, complementa a la afirmación de que el ajuste cuadrático es correcto.

Significado: El error de ajuste es una métrica clave para evaluar la calidad del modelo. En este caso, el modelo cuadrático es el más adecuado para describir la relación entre la cantidad de transferencias y el tiempo medido, como lo demuestra su bajo error cuadrático total y su comportamiento más consistente en el gráfico. No obstante, los picos de error evidencian la relevancia de examinar minuciosamente los datos para detectar potenciales fuentes de variabilidad o considerar modificaciones extra en el modelo si se requieren.

Por otro lado, el modelo cúbico, aunque más complejo, no mejora la precisión del ajuste y presenta errores más altos, lo que indica que no es el modelo correcto para estos datos. Esto refuerza la importancia de elegir un modelo que no solo se ajuste a los datos, sino que también sea interpretable y consistente con la teoría subyacente.

En conclusión, el modelo cuadrático describe correctamente los datos generales y confirma la complejidad teórica $\mathcal{O}(n^2)$ del algoritmo. Sin embargo, los picos de error, aunque pequeños, sugieren que podrían existir factores adicionales que afecten los datos y que no están siendo capturados por el modelo.

6.3. Generación de datos para analizar variabilidad de datos

Para poder analizar y comprobar los tiempos según la variabilidad de los datos decidimos crear nuestros propios sets de datos. Para eso, decidimos que cada set tenga la misma cantidad de elementos (en nuestro caso, 200 elementos) para que los tiempos no dependen de la cantidad sino de su variabilidad.

Para generar dichos archivos, tuvimos en cuenta diferentes situaciones: el margen de error es amplio o pequeño, los timestamps están más concentrados o más distribuidos, y las transacciones son únicas o se repiten. Luego, según sea el caso, los archivos se crean con números aleatorios (gracias a la biblioteca de Python random) según las condiciones impuestas para cada test por medio de ciclos que escriben las transacciones y los timestamps directamente en el archivo.

Estos escenarios fueron seleccionados porque representan los extremos del espacio de entrada y permiten analizar el comportamiento del algoritmo tanto en situaciones optimizadas como en aquellas potencialmente más costosas.

Finalmente pero no menos importante, estos tiempos se comprobaron por medio de un gráfico creado por medio de la biblioteca Matplotlib, el cual muestra el tiempo promedio de ejecución de los test dependiendo de la condición del mismo.