.UBAfiuba
FACULTAD DE INGENIERÍA

Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

Trabajo práctico 1: La mafia de los algortimos greedy

10 de abril de 2025

Integrantes

Alumno	Padrón
Alan Valdevenito	107585
Ignacio Zanoni Mutti	110884
Francisco Gutierrez	103543

Índice

1. Análisis del problema

Trabajamos para la mafia liderada por Amarilla Pérez y el Gringo Hinz. Se ha detectado un robo de dinero, pero no hay claridad sobre cómo se realizó ni quién es el responsable. Tras un interrogatorio, se obtuvo el nombre de un posible sospechoso. Se encontraron registros de transacciones a su nombre, y ahora debemos verificar si estos registros coinciden con los tiempos aproximados de las transacciones sospechosas. Para definir si el sospechoso es el ladrón, deben coincidir cada uno de los tiempos con un intervalo posible.

Para esto, contamos con la siguiente información:

- Un conjunto de transacciones n sospechosas.
- Para cada transacción sospechosa, se tiene un timestamp aproximado. Es decir, un tiempo t_i y un error asociado e_i. Por lo tanto sabemos que dichas transacciones fueron realizadas en el intervalo $[t_i - e_i \ ; \ t_i + e_i \ ]$
- Un conjunto de n transacciones realizadas por el sospechoso.
- Para cada transacción del sospechoso, se tiene un tiempo exacto en que ocurrió.

1.1. Solución propuesta

Nuestra solución para el problema es un algoritmo greedy que propone iterar las transacciones del sospechoso y por cada transaccion iterar los timestamp.

Para cada transaccion, iteramos todos los timestamp y obtenemos el indice de aquel timestamp que contenga a la transaccion actual. En caso de existir multiples timestamps que contengan a la transaccion, seleccionamos aquel timestamp de menor $t_i + e_i$ llegando al óptimo local.

Esto ultimo mencionado es la regla sencilla de nuestro algoritmo Greedy.

Una vez que obtuvimos el timestamp lo borramos de la lista de timestamps para evitar que dos o mas transacciones sean asignadas al mismo timestamp.

Si no se encuentra un timestamp que contenga la transacción actual, entonces no es el sospechoso correcto.

El óptimo global se alcanzaría al encontrar todas las asignaciones o al no poder realizar al menos una asignación.

Por otro lado, como optimización decidimos ordenar los timestamps por $t_i - e_i$ de manera ascendente. Esto reduce la cantidad de comparaciones ya que si $s_j < t_i - e_i$ entonces s_j no podrá entrar en ninguno de los siguientes timestamps. Básicamente nos permite descartar de forma rápida aquellos timestamps que no pueden contener la transaccion s_j.

1.2. Otras ideas

Antes de llegar a la solución propuesta atravesamos algunas ideas que fueron descartadas pero que resultaron de utilidad para comprender el problema. En esta sección buscamos contar como fué el proceso de iterar entre distintas alternativas y el camino que nos llevó hacia la solución propuesta

1.2.1. Ordenar y elegir

Primero pensamos que ordenar los timestamps de forma ascendente por $t_i - e_i$ nos ahorraría iterar sobre los mismos. Ya que si $n_i \notin (t_0 - e_0, t_0 + e_0) \Rightarrow$ no sería el delator. El error en esta lógica greedy se dá cuando hay 'solapamientos' entre timestamps. Nuestra decisión greedy tomaría el timestamp con el $t_i - e_i$ mas bajo ignorando que el $t_i + e_i$ puede estar incluyendo otras transacciones, privandolas de un timestamp que le permita formar parte de la solución

1.2.2. Ordenar y elegir con Heap

Nuestra primera ocurrencia para solventar el error de la elección greedy de **Ordenar y elegir** fué insertar en un heap de mínimos todos los timestamps candidatos tal que $n_i \in (t_0 - e_0, t_0 + e_0)$, el criterio de ordenamiento de nuestro heap sería el $t_0 + e_0$) mas bajo. De esta manera al desencolar el heap obtendríamos el timestamp correspondiente que no quite nos dé falsos negativos. Esta variante fué rápidamente descartada ya que complejiza el algoritmo a $O(n^2 * log(n))$.

2. Demostración

- Sea $S = [s_1, \ldots, s_n]$ el conjunto de transacciones ordenadas del sospechoso.
- Sea $T = [(t_1, e_1), \ldots, (t_n, e_n)]$ el conjunto de (timestamp, error) ordenado crecientemente por $t_i - e_i$.
- El algoritmo busca encontrar una asignación para cada s_i dentro de un intervalo $(t_i - e_i; t_i + e_i)$ tal que $t_i - e_i \leq s_i \leq t_i + e_i$ y con el menor $t_i + e_i$ posible en caso de haber varios disponibles.
- ¿Por qué priorizamos el menor $t_i + e_i$? Porque si asignamos un s_i a un intervalo que termina más pronto, dejamos los intervalos más largos para las transacciones siguientes, aumentando las chances de que también se puedan asignar.

Tesis general: El algoritmo greedy determina correctamente siempre si los timestamps del sospechoso s_i corresponden a los intervalos sospechosos $[t_i - e_i; t_i + e_i]$. Es decir:

- Si existe una asignación válida (cada s_i cae en un intervalo distinto), el algoritmo la encuentra y devuelve las asignaciones.
- Si no existe una asignación válida, el algoritmo lo detecta y retorna "No es el sospechoso correcto".

Parte 1: Si existe una asignación válida, el algoritmo la encuentra

Hipótesis: Existe una asignación válida, es decir, una permutación π de los intervalos $[t_i - e_i, t_i + e_i]$ tal que:

$$t_{\pi(j)} - e_{\pi(j)} \leq s_j \leq t_{\pi(j)} + e_{\pi(j)}, \quad \text{para } j = 1, 2, \ldots, n,$$

donde cada s_j se asigna a un intervalo distinto $I_{\pi(j)}$.

Tesis: Bajo esta hipótesis, el algoritmo encuentra la asignación válida y devuelve las asignaciones.

Consideremos el proceso del algoritmo:

- **Paso inicial**: Comienza con n intervalos y procesa s_1.
- **Para** s_1: Existe al menos un intervalo $I_{\pi(1)}$ tal que $s_1 \in [t_{\pi(1)} - e_{\pi(1)}, t_{\pi(1)} + e_{\pi(1)}]$. La función obtener_timestamp encuentra un intervalo válido (digamos I_{a_1}) con menor $t_i + e_i$, asigna $s_1 \to I_{a_1}$, y elimina I_{a_1}. Quedan $n - 1$ intervalos.
- **Para** s_2: En la asignación válida, $s_2 \in I_{\pi(2)}$. Tras usar I_{a_1}, quedan $n-1$ intervalos, incluyendo $I_{\pi(2)}$ (o suficientes intervalos válidos). El algoritmo asigna $s_2 \to I_{a_2}$ y elimina I_{a_2}. Quedan $n - 2$ intervalos.
- **Generalización**: Para s_k (con $k = 1, 2, \ldots, n$):
  - Tras asignar $s_1, \ldots, s_{k-1}$, se han usado $k - 1$ intervalos.
  - Quedan $n - (k - 1)$ intervalos, y $s_k \in I_{\pi(k)}$, donde $I_{\pi(k)}$ no ha sido usado.
  - obtener_timestamp encuentra un intervalo que contiene a s_k, lo asigna (digamos I_{a_k}) y lo elimina.
- **Final**: Para s_n, queda 1 intervalo. Si s_n está en él, se asigna; de lo contrario, contradice la asignación válida (ver Parte 2). Como existe, el algoritmo termina con n asignaciones.

La elección greedy (menor $t_i + e_i$) no bloquea la solución, pues siempre hay un intervalo disponible para cada s_k, y priorizar el menor extremo derecho maximiza las opciones para los s_i posteriores.

$\therefore$ El algoritmo encuentra la asignación válida y queda demostrado por método directo que determina de forma correcta cuando el sospechoso es culpable.

Parte 2: Si no existe al menos una asignación válida, el algoritmo lo detecta

Hipótesis: No existe una asignación válida, es decir, no es posible asignar cada s_i a un intervalo $[t_i - e_i; t_i + e_i]$ de los timestamps.

Tesis: Bajo esta hipótesis, el algoritmo detecta la ausencia de asignación y retorna "No es el sospechoso correcto".

Analicemos la ejecución del algoritmo:

- **Paso inicial**: El algoritmo comienza asignando s_1 a un intervalo válido (si existe), digamos I_{a_1}.
- Continúa asignando $s_2, \ldots, s_{k-1}$ a intervalos $I_{a_2}, \ldots, I_{a_{k-1}}$, todos distintos.
- Llega a s_k, donde, tras usar $k-1$ intervalos, quedan $n-(k-1)$ intervalos.

Punto de fallo: Como no hay asignación válida, existe al menos un s_k tal que ninguno de los intervalos restantes lo contiene. En este caso:

- obtener_timestamp revisa los intervalos disponibles.
- Para cada intervalo I_j restante, verifica si:

$$t_j - e_j \leq s_k \leq t_j + e_j.$$

- Si ninguno satisface esta condición, retorna None, y el algoritmo termina con "No es el sospechoso correcto".

El algoritmo no puede asignar incorrectamente todos los s_i, porque:

- Solo asigna un s_i a un intervalo si s_i está dentro de $[t_i - e_i, t_i + e_i]$.
- Elimina cada intervalo tras usarlo, asegurando que las asignaciones sean a intervalos distintos.
- Si completa las n asignaciones, habría encontrado una asignación válida, lo cual contradice la suposición.

Ejemplo:

- Intervalos: $[1, 2], [3, 4]$.
- Transacciones: $s_1 = 1, s_2 = 1$.
- $s_1 = 1 \rightarrow [1, 2]$. Queda $[3, 4]$.
- $s_2 = 1$: $[3, 4]$ no contiene a 1. Falla y retorna "No es el sospechoso correcto".

$\therefore$ Si no existe al menos una asignación válida, el algoritmo lo detecta y rechaza al sospechoso, quedando demostrado por método directo que determina de forma correcta cuando el sospechoso es inocente.

3. Algoritmo

3.1. Codigo

```
TIMESTAMP = 0
ERROR = 1

def encontrar_rata(timestamps, transacciones):
    timestamps_ordenados = sorted(timestamps, key=lambda x: x[TIMESTAMP] - x[ERROR
    ])
    asignaciones = []

    for t in transacciones:

        if not timestamps_ordenados:
            break

        ts_index = obtener_timestamp(timestamps_ordenados, t)

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

        asignaciones.append((t, timestamps_ordenados.pop(ts_index)))

    return formatear_resultado(asignaciones)

def obtener_timestamp(timestamps, transaccion):
    min_der = ()

    for i in range(0, len(timestamps)):
        izq = timestamps[i][TIMESTAMP] - timestamps[i][ERROR]
        der = timestamps[i][TIMESTAMP] + timestamps[i][ERROR]

        if not (izq <= transaccion <= der):

            if len(min_der) == 0:
                return None

            break

        if (len(min_der) == 0) or (der < min_der[0]):
            min_der = (der, i)

    return min_der[1]
```

3.2. Complejidad

Para analizar la complejidad del algoritmo propuesto, evaluemos cada operación realizada.

1. **Ordenar los timestamps:** Ordenamos los n timestamps por $t_i - e_i$ lo cual tiene una complejidad de $O(nlogn)$.
2. **Iteración principal:** El bucle principal recorre todas las transacciones una unica vez, por lo que su complejidad es $O(t)$ con t siendo las transacciones.
3. **Timestamps:** Para cada transaccion se recorren (en el peor caso) todos los timestamps, por lo que su complejidad es $O(n)$ con n siendo los timestamps.
4. **Eliminado de timestamps asignados:** La funcion *obtener_timestamp* devuelve el indice del timestamp asignado y se elimina de la lista de timestamps ordenados mediante *pop*(). Dicho indice puede estar en cualquier posicion de la lista, por lo que su complejidad es $O(n)$.
5. **Formato del resultado:** Si el sospechoso es la rata, se llama a la funcion *formatear_resultado* que se encarga de darle el formato correcto a la salida. Para ello recorre la lista de asignaciones, por lo que su complejidad es $O(a)$ con a la cantidad de asignaciones.

Dado que la condicion para que el sospechoso sea la rata es que cada transaccion sea asignada a un unico timestamp, esto quiere decir que la cantidad de transacciones debe coincidir con la cantidad de timestamps con lo cual $n = t = a$.

Luego, la complejidad del algoritmo resulta ser $O(n^2)$.

4. Análisis de la variabilidad de valores

Para comprender como afectan los diferentes datos de entrada a los tiempos de ejecución de nuestra solución provista, es necesario ver mas de cerca como se recorren los timestamps candidatos, y que operaciones se realizan sobre los mismos para cada transacción.

1. Si la transacción pertenece al intervalo candidato y no hay mínimo local lo nombro parcialmente como mi óptimo local
2. Si la transacción pertenece al intervalo candidato, existe mínimo local y el candidato tiene menor límite a derecha $(t_i + e_i)$, nombro parcialmente a este como mi óptimo local
3. Si la transacción no pertenece al intervalo, finaliza la iteración y devuelve como óptimo local el timestamp que fué nombrado como solución parcial
4. Si al terminar la iteración no hay solución parcial, entonces no es la rata.

4.1. Si no es la rata

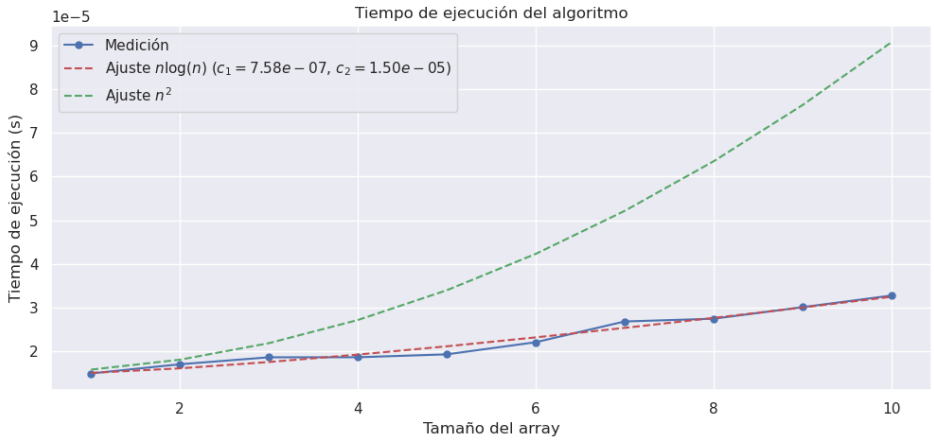
Si nuestro indagado no es la rata, la variabilidad de valores de entrada afecta únicamente en un sentido:

Estudiando nuestra iteración podemos observar que, mientras mas pequeño el valor de la transacción respecto a las demás, menor será la complejidad que tendrá nuestro algoritmo. Ya que se ahorra recorrer el resto de transacciones (todas las que sean mayores). Esto quiere decir, si la transacción que exculpa a nuestro sospechoso se encuentra es t_0 nuestro algoritmo lo resuelve en prácticamente $O(1)$. Mientras que si se encuentra en t_n, siendo n la cantidad total de transacciones, la complejidad temporal será $O(n^2)$ en el peor de los casos como veremos en el próximo estudio

4.2. Si es la rata

Si es la rata, el caso mas optimista para nuestro algoritmo es el que mejor aprovecha el ítem 3 de nuestra iteración. Esto sucede cuando el solapamiento entre los timestamps candidatos para cada transacción es mínimo. Logrando en el mejor de los casos una complejidad de $O(n * log(n))$ debido al ordenamiento realizado al inicio. El peor de los casos es cuando sucede todo lo contrario a lo dicho. Cuando el solapamiento entre los timestamps es máximo vemos que para cada transacción va a tener que estudiarse cada timestamp para poder garantizar que el seleccionado sea el que cuente con el menor límite a derecha $(t_i + e_i)$

En el siguiente gráfico podemos ver 10 mediciones de casos óptimos junto al ajuste cuadrático y logarítmico. Resulta evidente que la curva que mejor ajusta para este tipo de datos de entrada es la de la ecuación $n * log(n)$



5. Ejemplos de ejecucion

5.1. Generador

Para realizar ejemplos de ejecucion y realizar las mediciones de tiempo, generamos datos para nuestro algoritmo Greedy. Para ello utilizamos las sigueintes funciones encargadas de generar datos aleatorios y que además permiten configurar la cantidad de datos y el rango de valores.

```
def generar_timestamps(n, ts_min, ts_max):
    timestamps = [random.randint(ts_min, ts_max) for _  in range(n)]
    return timestamps

def generar_errores_por_timestamp(timestamps):
    errores = [random.randint(0, ts) for ts in timestamps]
    return errores

def generar_transacciones_ordenadas(n, ts_min, ts_max):
    transacciones = [random.randint(ts_min, ts_max) for _  in range(n)]
    transacciones.sort()
    return transacciones

def generar_entrada_aleatoria(n, rango_min = 0, rango_max = 1000):
    timestamps = generar_timestamps(n, rango_min, rango_max)
    errores = generar_errores_por_timestamp(timestamps)
    timestamps_aproximados = list(zip(timestamps, errores))

    transacciones = generar_transacciones_ordenadas(n, rango_min, rango_max)

    return timestamps_aproximados, transacciones
```

¿Como se generan los datos?. Analicemos cada una de las funciones utilizadas para ello.

1. **Generar timestamps:** Genera t timestamps aleatorios, distribuidos entre ts_{min} y ts_{max}.
2. **Generar errores:** Genera e errores aleatorios, distribuidos entre 0 y el valor del timestamp correspondiente. El rango maximo es este ultimo ya que de otra forma podriamos generar un timestamp aproximado del estilo $(100, 334)$ donde el intervalo resultaria $[-234, 434]$.
3. **Generar transacciones:** Genera n transacciones aleatorias, distribuidas entre ts_{min} y ts_{max}. Además de esto, las ordena de menor a mayor.
4. **Generar entrada aleatoria:** Invoca a las funciones encargadas de generar los datos y devuelve dos listas. Una primer lista de tuplas (t_i, e_i) que contiene cada timestamp junto con su error y una segunda lista de transaciones.

¿Como generar datos?. Ejecutar el archivo *generador.py* indicando como parametro la cantidad de elementos, el rango minimo y el rango maximo. Esto generara datos en la ruta *tests/gen* con el mismo formato que tienen los ejemplos proporcionados por la catedra.

5.2. Verificador

Para poder validar nuestros datos generados y de esta forma corroborar lo encontrado, utilizamos el siguiente verificador. Mencionamos que nuestro verificador recibe una lista de timestamps, una lista de transacciones ordenadas y un resultado (string) de la forma:

```
213 --> 229 +- 45
607 --> 599 +- 12
711 --> 727 +- 49
806 --> 856 +- 70
816 --> 892 +- 82
```

```
def validar_asignaciones(timestamps, transacciones, resultado):
    transacciones_resultado, timestamps_resultado = parsear_asignaciones(resultado)
     # Cada transaccion i se corresponde con el intervalo i

    contador = {t: timestamps.count(t) for t in timestamps}

    for i, transaccion in enumerate(transacciones_resultado):

        ts, error = timestamps_resultado[i]
        izq, der = ts - error, ts + error

        # La transaccion no esta dentro del intervalo
        if not (izq <= transaccion <= der):
            return False

        # El intervalo ya fue usado por otra transaccion
        # Esto no puede suceder ya que debe haber una transaccion por cada
    intervalo (a menos que el intervalo este repetido)
        if contador[timestamps_resultado[i]] <= 0:
            return False
        contador[timestamps_resultado[i]] -= 1

    return True
```

¿Que condiciones deben cumplirse para que un sospechoso sea la rata?.

1. **C**ada transaccion debe asignarse a un unico intervalo. Es decir, debe haber una relacion 1 a 1 entre las transacciones y los intervalos.
2. **C**ada transaccion asignada debe estar en efecto incluida en el intervalo.

Además, puede suceder que existan dos intervalos exactamente iguales y tambien dos transacciones exactamente iguales. Esto sucede por ejemplo en el caso de prueba $5000 - es.txt$ proporcionado por la catedra donde se repite la transaccion 1780 y se repite el intervalo $(1659, 404)$. Es por esta razon que nuestro verificador utiliza un diccionario con la cantidad de apariciones de cada intervalo.

¿Que hace nuestro verificador?.

1. Parsea las asignaciones del resultado obteniendo una lista de las transacciones y una lista de los timestamps aproximados donde cada transaccion i se corresponde con el timestamp i.
2. Se calculan las apariciones de cada timestamp aproximado y se guarda esta informacion en un diccionario.
3. Se recorren las transacciones y para cada transaccion i se obtiene su respectivo timestamp aproximado i. Se valida que la transaccion este dentro del intervalo asignado y que el intervalo no haya sido utilizado por otra transaccion. Es decir, que cada transaccion este asignada a un unico intervalo.
4. Por cada transaccion asignada a un intervalo, se decrementa la cantidad de apariciones de dicho intervalo en el diccionario de tal forma que si su valor es 0 quiere decir que no deberia haber sido asignado a otra transaccion.

5.3. Generar entrada y corroborar su resultado

Una forma de realizar ejemplos de ejecución para encontrar soluciones y corroborar lo encontrado es:

1. **G**enerar entrada utilizando el generador.
2. **E**jecutar el algoritmo Greedy con el archivo generado en el paso anterior y utilizar el flag $-v$ al momento de ejecutarlo.

Ejemplo 1: El sospechoso es la rata

```
alanezequiel@Alan-VirtualBox:~/Escritorio/TDA/TDA-TPS$ python3 ./generador.py 10 0 1000
alanezequiel@Alan-VirtualBox:~/Escritorio/TDA/TDA-TPS$ cat /home/alanezequiel/Escritorio/TDA/TDA-TPS/tests/gen/gen-10
10
386,102
877,89
440,81
1000,305
315,199
363,343
960,798
821,239
920,204
483,435
198
397
445
489
530
533
813
883
950
963
```

```
alanezequiel@Alan-VirtualBox:~/Escritorio/TDA/TDA-TPS$ python3 ./tp1.py /home/alanezequiel/Escritorio/TDA/TDA-TPS/tests/gen/gen-10 -v
198 --> 315 ± 199
397 --> 386 ± 102
445 --> 440 ± 81
489 --> 363 ± 343
530 --> 483 ± 435
533 --> 960 ± 798
813 --> 877 ± 89
883 --> 821 ± 239
950 --> 920 ± 204
963 --> 1000 ± 305

El algoritmo encontro una asignacion valida: True
```

Ejemplo 2: El sospechoso no es la rata

```
alanezequiel@Alan-VirtualBox:~/Escritorio/TDA/TDA-TPS$ python3 ./generador.py 10 0 1000
alanezequiel@Alan-VirtualBox:~/Escritorio/TDA/TDA-TPS$ cat /home/alanezequiel/Escritorio/TDA/TDA-TPS/tests/gen/gen-10
10
658,585
905,831
913,231
558,540
811,681
919,230
218,55
578,331
424,78
853,583
39
79
97
128
133
183
198
469
474
493
```

```
alanezequiel@Alan-VirtualBox:~/Escritorio/TDA/TDA-TPS$ python3 ./tp1.py /home/alanezequiel/Escritorio/TDA/TDA-TPS/tests/gen/gen-10 -v
No es el sospechoso correcto
```

Para este segundo ejemplo podemos ver que el sospechoso no es correcto ya que nuestro algoritmo realiza las siguientes asignaciones:

```
39 --> 558 +- 540
79 --> 658 +- 585
97 --> 905 +- 831
```

Luego de realizar estas asignaciones, al llegar a la transaccion 128 restan los intervalos $(811, 681)$, $(218, 55)$, $(578, 331)$, $(853, 583)$, $(424, 78)$, $(913, 231)$ y $(919, 230)$. Dado que la transaccion no esta incluida en ningun intervalo, el sospechoso no es correcto.

6. Mediciones

Se realizaron mediciones de tiempo con el objetivo de verificar la complejidad teórica propuesta. Para ello, se generaron gráficos y se utilizó la técnica de cuadrados mínimos, siguiendo la explicación detallada proporcionada por la cátedra.

Para el grafico del error, se utiliza el error cuadratico total (SSE) el cual se calcula como

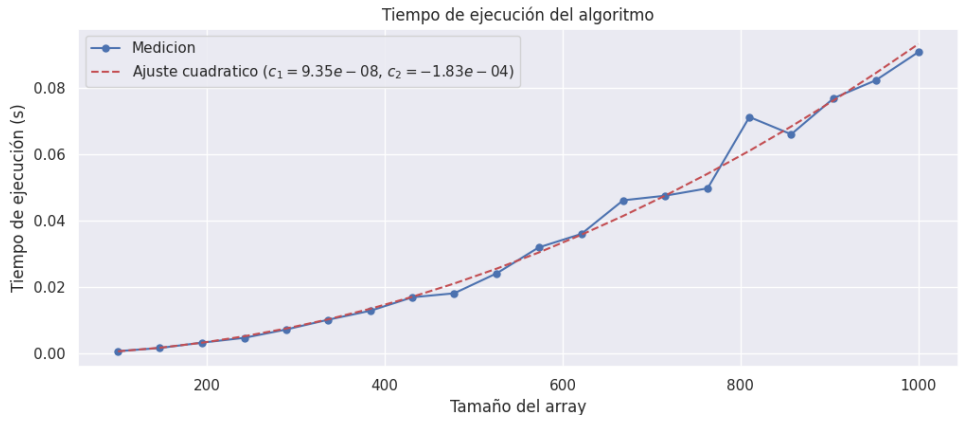
$SSE = \sum_{i=1}^{n}(y_i - \hat{y}_i)^2$

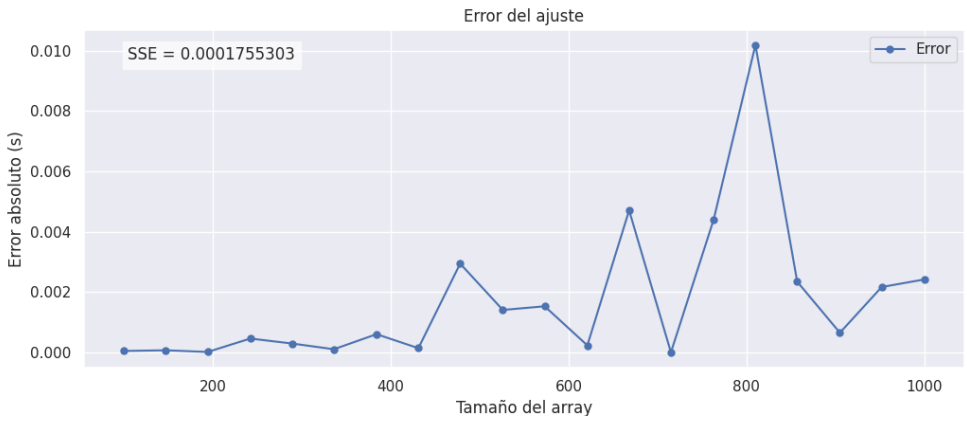
donde y_i es el valor real, $\hat{y}_i$ es el valor predicho y n la cantidad de puntos.

Mencionamos tambien que, tal y como se indica en la consigna, tomamos más de una medición de la misma muestra y nos quedamos con el promedio para reducir el ruido en la medición. Esto utilizando el archivo util.py que nos proporciona la catedra y colocando la constante `RUNS_PER_SIZE` con valor 30.

6.1. Medicion 1

Consideremos el siguiente grafico generado considerando un tamaño de hasta 1.000 timestamps:

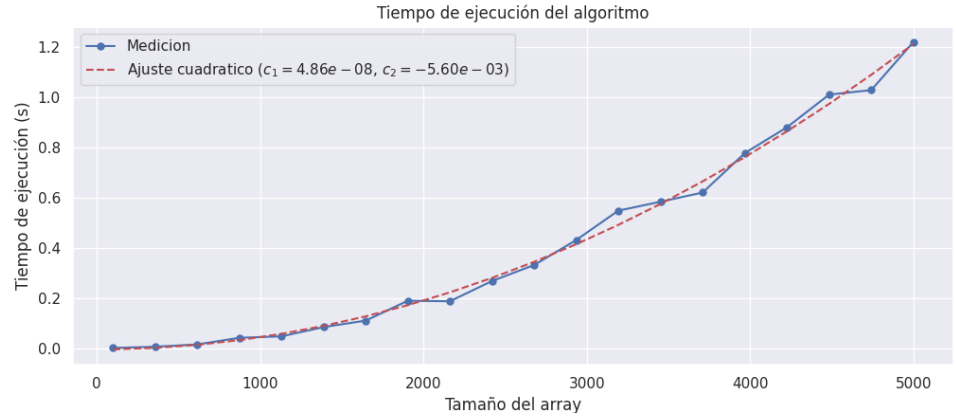


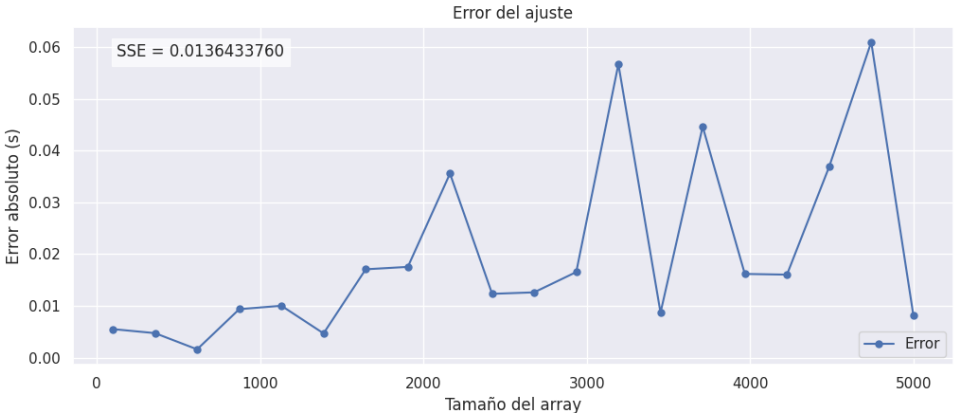


El error cuadratico total con las mediciones indicadas anteriormente para nuestra aproximacion es 0.00018 s^2.

6.2. Medicion 2

Consideremos el siguiente grafico generado considerando un tamaño de hasta 5.000 timestamps:

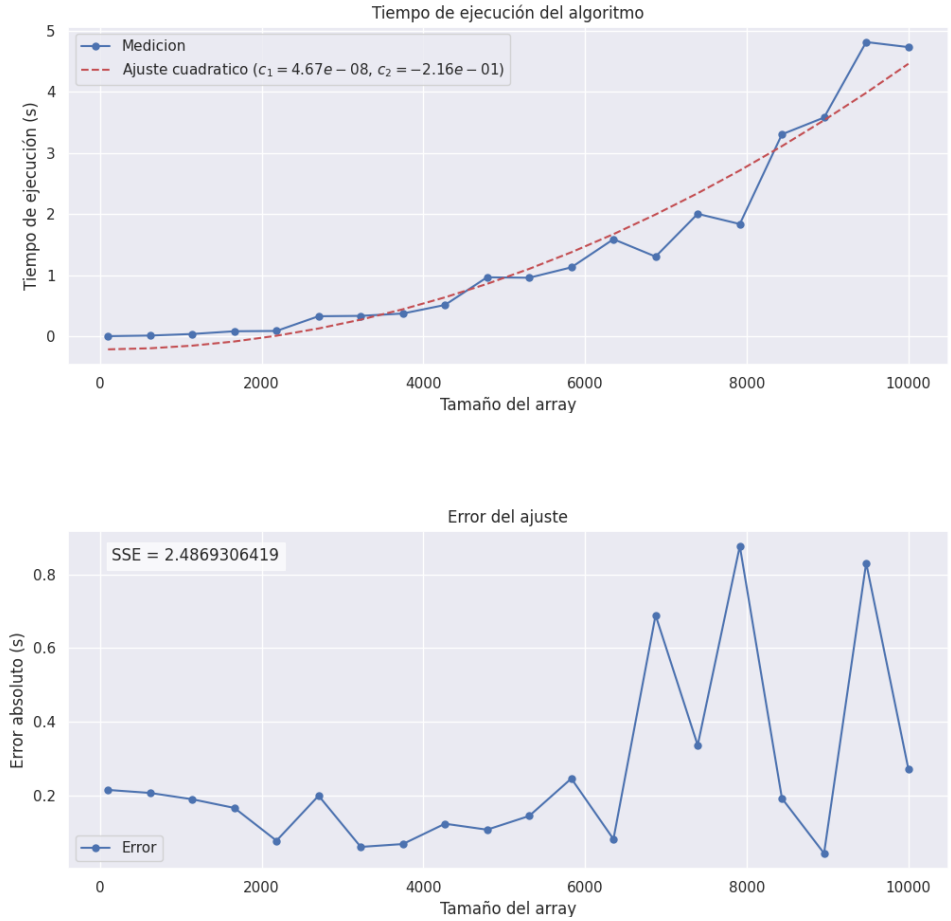




El error cuadratico total con las mediciones indicadas anteriormente para nuestra aproximacion es 0.014 s^2.

6.3. Medicion 3

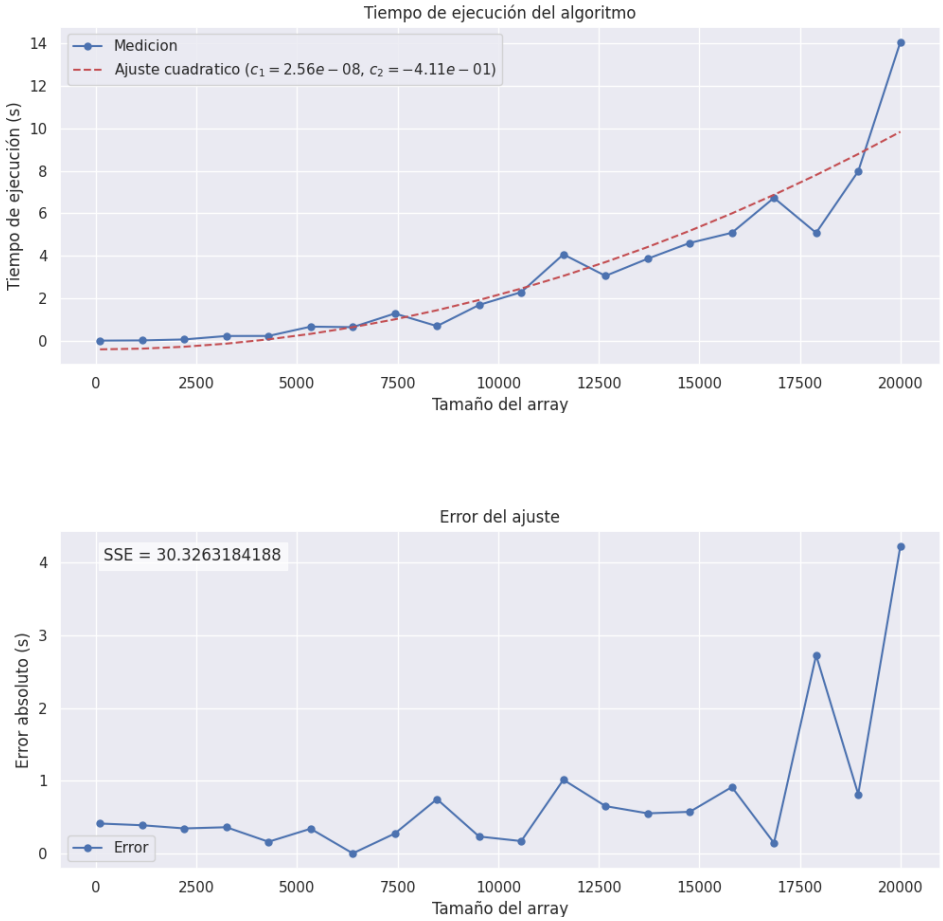
Consideremos el siguiente grafico generado considerando un tamaño de hasta 10.000 timestamps:



El error cuadratico total con las mediciones indicadas anteriormente para nuestra aproximacion es 2.49 s^2.

6.4. Medicion 4

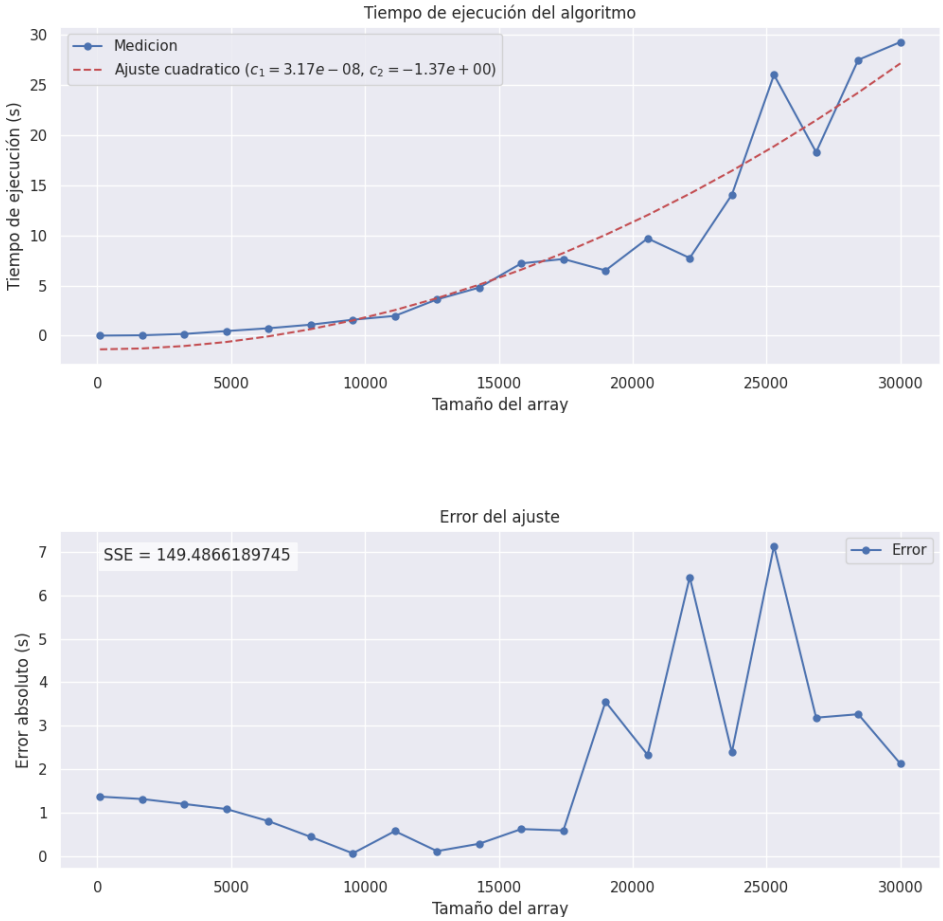
Consideremos el siguiente grafico generado considerando un tamaño de hasta 20.000 timestamps:



El error cuadratico total con las mediciones indicadas anteriormente para nuestra aproximacion es 30.3 s^2.

6.5. Medicion 5

Consideremos el siguiente grafico generado considerando un tamaño de hasta 30.000 timestamps:



El error cuadratico total con las mediciones indicadas anteriormente para nuestra aproximacion es 149.5 s^2.

7. Conclusiones

En las mediciones realizadas se puede apreciar claramente que el algoritmo sigue una tendencia cuadratica con respecto al tamaño de la entrada. Si bien el ajuste es bastante preciso y confirma que la complejidad del algoritmo es efectivamente $O(n^2)$, tal como se había teóricamente propuesto, pierde precision a medida que aumenta el tamaño de la entrada.

Esto puede observarse en cada uno de los graficos.

Particularmente en las mediciones 3, 4 y 5 en un comienzo el ajuste resulta muy preciso y se pierde esta precision conforme aumenta el tamaño de la entrada.