Teoría de Algoritmos

(TB024) Curso Buchwald - Genender

Trabajo Práctico 1
La mafia de los algortimos greedy

10 de abril de 2025

Conde Cardó,	Colombo Farre,	Willson,
Lucas Ariel	Iván Joel	Marina
112201	111671	103532

1. Introducción

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...

1.1. Consigna

- 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.
- 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.
- 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.
- 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.
- 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.

2. Análisis del problema

2.1. Elementos presentes

Para lograr una correcta explicación del problema, decidimos introducir en primera instancia los elementos que se utilizarán a lo largo de todo este informe. Su lectura ayudará a interpretar adecuadamente lo que explicamos en cada sección posterior.

- n: Cantidad total de timestamps en cada uno de los arreglos.
- S: Arreglo que representa las operaciones realizadas por el sospechoso, de forma:

  $$[s_0, s_1, \cdots, s_n]$$

  Sus elementos, los timestamps, se encuentran ordenados de menor a mayor. Es decir,

  $$s_0 \leqslant s_1 \leqslant \cdots \leqslant s_n$$

- T: Arreglo de tuplas que contiene las aproximaciones de transacciones sospechosas. Presenta la forma:

  $$[(t_0, e_0), (t_1, e_1), \cdots, (t_n, e_n)]$$

  Donde cada t_j representa el timestamp aproximado y e_j el error asociado a dicho timestamp. A lo largo de nuestro análisis, en varias oportunidades los presentaremos como los intervalos definidos por ambas variables:

  $$T_j = [t_j - e_j; t_j + e_j]$$

- s_i compatible con (t_j, e_j): Un timestamp de S es compatible con uno de los intervalos de T cuando:

  $$t_j - e_j \leqslant s_i \leqslant t_j + e_j$$

2.2. El problema

Luego de un profundo análisis, comprendimos que el problema radica en ubicar a cada uno de los n timestamps de S dentro de los intervalos definidos por las aproximaciones y sus errores asociados en T. De esta forma, si se logra una relación 1 a 1 entre ambos arreglos, podremos concluir que el sospechoso es efectivamente la rata. En caso contrario, determinaremos que no se trata de quien estábamos buscando.

3. Análisis de posibles soluciones

Luego de identificar el objetivo del algoritmo, comenzamos a proponer diversas soluciones que, por un motivo u otro, no lograron ser óptimas en todos los escenarios. Igualmente, fueron de gran ayuda para comprender la raíz del problema y arribar a una visión mucho más enfocada del mismo.

3.1. Ordenar T según las aproximaciones t_i

Ordenar ascendentemente las aproximaciones del arreglo T fue la primera solución que propusimos. Luego iteraríamos el arreglo S para determinar para cada índice:

Si s_i es compatible con t_i, es decir, si

$$s_i \in [t_i - e_i; t_i + e_i],$$

la relación se cumple y se puede agregar a la solución. Caso contrario, se determina que no se pueden asignar todos los timestamps a los intervalos presentados.

A través de este enfoque se busca aprovechar el ordenamiento inicial de los arreglos y que ambos tienen la misma cantidad de elementos para asignar relaciones directas entre los elementos ubicados en la **misma posición** de cada arreglo.

Sin embargo, esta solución no resultó ser óptima en todos los escenarios, como en el siguiente contraejemplo:

$$T_{ordenado} = [(2,0),(5,5),(6,2)]$$

$$S = [2,5,10]$$

Los intervalos determinados por los valores de T serían $[[2,2],[0,10],[4,8]]$

Se asignarían (t_0, e_0) a s_0 y (t_1, e_1) a s_1. Como se puede ver, al llegar a $s_2 = 10$ sería imposible relacionarlo al único intervalo restante, el $[4,8]$ determinado por (t_2, e_2). Mientras que en un algoritmo óptimo la solución sería

$$2 \longrightarrow (2,2)$$
$$5 \longrightarrow (6,2)$$
$$10 \longrightarrow (5,5)$$

Consecuentemente, queda completamente descartada esta solución.

3.2. Ordenar T según el error asociado e_i

Otra solución que pensamos fue ordenar los elementos de T, nuevamente de menor a mayor, pero en función del error asociado a cada aproximación. Luego iteraríamos sobre T y emparejaríamos, en cada iteración, el timestamp compatible de S que tenga la menor diferencia con el t_i en cuestión.

Es una propuesta un poco rebuscada (y con dudosa regla greedy) que nos tomó varios ejemplos demostrar que no era óptima. En el siguiente contraejemplo se puede apreciar que no lo es:

$$T_{ordenado} = [(856,70),(892,80)]$$

$$S = [806,816]$$

Nuestro algoritmo emparejaría el $(856,70)$ de T con el 816 de S puesto que

$$856 - 816 < 856 - 806$$

y, por lo tanto, no lograría encontrar la solución óptima que resulta ser:

$$806 \longrightarrow (856, 70)$$
$$816 \longrightarrow (892, 80)$$

3.3. Emparejar con T_j que minimice la diferencia con el tiempo de inicio

El último algoritmo analizado antes de encontrar el óptimo consiste en iterar S y emparejar cada timestamp con el intervalo compatible de T cuyo inicio $t_i - e_i$ sea el mínimo disponible en cada paso. De esta forma, pensamos que se podrían *estirar* los tiempos de finalización para que los timestamps siguientes tuvieran más flexibilidad.

Sin embargo, el algoritmo resultó no ser óptimo, como se puede demostrar en el siguiente contraejemplo:

$$T = [(100, 30), (200, 200)] \rightarrow T_{intervalos} = [[70, 130], [0, 400]]$$

$$S = [125, 280]$$

Nuestra solución le asignaría al $s_0 = 125$, la transacción $(200, 200)$, que determina el intervalo $[0, 400]$. En consecuencia, cuando llegue al $s_1 = 280$, no podría relacionarlo con ningún intervalo de T. Sin embargo, en un algoritmo óptimo la solución sería la siguiente:

$$125 \longrightarrow (100, 30)$$
$$280 \longrightarrow (200, 200)$$

No obstante, estas propuestas fallidas nos ayudaron a notar por qué nuestro algoritmo no funcionaba y, luego de varios intentos, encontramos la solución que buscábamos.

4. Solución optima

4.1. Emparejar con T_j que minimice la diferencia con el tiempo de finalización

La solución final que propusimos presenta un enfoque muy similar al del último algoritmo subóptimo mencionado.

Ésta comienza iterando el arreglo S y vincula cada timestamp al intervalo compatible de T cuya finalización sea más próxima, es decir, el t_i+e_i que menor diferencia tenga con el s_i. De esta manera, logramos obtener una sucesión de soluciones óptimas locales que garantizan la compatibilidad momentánea y minimizan la probabilidad de que los próximos elementos no puedan ser enlazados apropiadamente.

Finalmente, se consigue una solución óptima global, en la que todos los timestamps del sospechoso son emparejados con los timestamps aproximados y resulta ser, en efecto, la rata. O, en caso de que no se pueda emparejar al menos uno de los timestamps, el algoritmo determina que el problema no tiene solución. De esta forma, el sospechoso no sería quien estamos buscando.

4.2. Demostración de optimalidad

Mediante un proceso inductivo, demostraremos la optimalidad de nuestro algoritmo. Para esto, es de crucial importancia tener presente, y en resumidas cuentas, qué es lo que buscamos en una solución óptima. Como ya desarrollamos apropiadamente en el apartado de análisis, esta debe cumplir la siguiente afirmación:

"*Una solución es óptima cuando a cada s_i se le asigna un elemento de T que determina un intervalo al que s_i debe pertenecer. Si se encuentra al menos un s_i al que no es posible asignarle ningún intervalo, el problema no dispone de ninguna solución admisible.* "

Dicho esto, ya estamos en condiciones de definir el caso base. Este se da cuando $n = 1$, entonces:

- Si el timestamp de S es compatible con el intervalo formado por la aproximación y el error asociado de T, la única solución optima es emparejarlos.
- Si el timestamp de S **no** es compatible con el intervalo formado por la aproximación y el error asociado de T, se puede asegurar que no existe solución alguna.

Ahora, supongamos que seguimos iterando S y encontramos soluciones óptimas para los primeros k elementos. Cuando lleguemos al timestamp de la posición $k + 1$ se podrán presentar varias situaciones:

- Si ningún intervalo de T es compatible con el timestamp en cuestión, se determina la irresolubilidad.
- Si un solo intervalo de T es compatible con el timestamp de s_{k+1}, estos son emparejados.
- Si dos o más intervalos de T son compatibles con la transacción de S en iteración, se elige la que presente un tiempo de finalización $t_j + e_j$ menor.

Este último inciso, como es el que rige la regla *Greedy* de nuestro algoritmo y verdaderamente determina la optimalidad del mismo, deberá ser demostrado y lo haremos mediante el método del *absurdo* o *contradicción*:

Supongamos que el timestamp s_v es compatible con dos intervalos de T: T_m y T_{m+1}, donde T_m presenta un tiempo de finalización menor. E, ignorando nuestras hipótesis previas, emparejamos s_v con T_{m+1}.

Si el siguiente timestamp $s_{v+1} > s_v$ sólo es compatible con T_{m+1} (y no con T_m), entonces, al haber asignado previamente a T_{m+1}, no podremos encontrar una solución óptima, puesto que ningún elemento restante de T es compatible. En cambio, si a s_v le hubiésemos asignado T_m, entonces T_{m+1} seguiría disponible para s_{v+1} y podríamos asignarle ese intervalo.

De esta forma, concluimos que eligiendo siempre el intervalo compatible que tenga menor diferencia con el tiempo de finalización para cada timestamp dejamos opciones más laxas para los siguientes timestamps y, como resultado, maximizamos la compatibilidad futura. Por consiguiente, solo será posible que no se encuentre una solución óptima cuando el problema sea imposible de resolver. Es decir, cuando no exista solución con este algoritmo ni con ningún otro.

4.3. Implementación del algoritmo

Pusimos en funcionamiento nuestro algoritmo mediante el desarrollo de la siguiente función:

```
1  def verificar_coincidencia_timestamps(t, s):
2      solucion = []
3      seleccionados = set()
4
5      for timestamp in s:
6          diferencia = float('inf')
7          seleccion = None
8
9          for i, (aproximacion, error) in enumerate(t):
10             if i in seleccionados:
11                 continue
12             inicio = aproximacion - error
13             fin = aproximacion + error
14
15             if fin - timestamp < diferencia and inicio <= timestamp <= fin:
16                 seleccion = i
17                 diferencia = fin - timestamp
18
19         if seleccion is None:
20             return None
21         solucion.append((timestamp, t[seleccion][0], t[seleccion][1]))
22         seleccionados.add(seleccion)
23
24     return solucion
```

Como se puede observar, esta función recibe los arreglos s y t, que poseen las características descritas para los arreglos S y T, respectivamente. Luego, se busca una solución iterando ambos arreglos como fue explicado con anterioridad. Cabe destacar que se guardan los índices de los intervalos ya utilizados en un set `seleccionados`, para que no vuelvan a ser asignados a próximos timestamps. Es relevante señalar que el algoritmo finaliza si para algún timestamp no se encuentra un óptimo local (líneas 19 y 20). De esta forma, se evita el surgimiento de soluciones erróneas y más comparaciones innecesarias.

4.4. Optimizaciones

Más tarde, identificamos algunas optimizaciones que nos permitieron reducir considerablemente los tiempos de ejecución del algoritmo, especialmente en pruebas con grandes volúmenes de datos. La primera mejora consistió en ordenar la lista t de manera ascendente según el tiempo de finalización de cada t_i, es decir, utilizando $t_i + e_i$ como criterio. De esta manera, logramos que la primera aproximación (no asignada previamente) compatible al iterar sobre t sea la elección optima para esa instancia.

Posteriormente, convertimos t en una lista doblemente enlazada. Esto nos permitió eliminar los intervalos ya asignados, reduciendo progresivamente la cantidad de opciones disponibles para cada s_i y evitando así comparaciones innecesarias.

En los aparatados que siguen profundizaremos más sobre este asunto en particular. A continuación, el código modificado:

```python
def verificar_coincidencia_timestamps(t,s):
    solucion = []
    t_ord = ListaDoblementeEnlazada(sorted(t, key = lambda x: x[0] + x[1]))

    for timestamp in s:
        flag = False

        for nodo in t_ord:
            aproximacion, error = nodo.get_valor()
            inicio = aproximacion - error
            fin = aproximacion + error

            if inicio <= timestamp <= fin:
                flag = True
                t_ord.borrar_elemento(nodo)
                solucion.append((timestamp, aproximacion, error))
                break

        if flag == False: # El problema no tiene solucion.
            return None

    return solucion
```

Como se mencionó anteriormente —y ahora será explicado con mayor detalle—, para cada s_i se selecciona el primer elemento de la lista doblemente enlazada que sea compatible. En la implementación anterior buscábamos explícitamente que la diferencia $(t_i + e_i) - s_i$ fuera la menor posible, en la nueva esta condición se da naturalmente por el orden del arreglo T.

Así, lo único que hay que verificar es que el timestamp esté contenido en el intervalo actual. En caso de que así sea, se lo elimina para evitar considerarlo en vano más adelante.

En el Cuadro 1 se puede observar la gran diferencia en los tiempos de ejecución usando la primera versión vs la segunda y definitiva.

Esta mejora permite no sólo reducir el tamaño de la lista que iterará el segundo `for` sino que la cantidad de comparaciones que se tengan que hacer allí serán potencialmente menos, ya que es más probable que el intervalo compatible esté más cerca de los primeros elementos de `t_ord`.

4.5. Análisis de la complejidad temporal

Inicialmente, nuestro algoritmo realiza una operación de tiempo constante al crear una lista `solucion`. Inmediatamente después, crea la lista doblemente enlazada que recibe como argumento la lista original ya ordenada. Para eso se utiliza `sorted` que tiene una complejidad $\mathcal{O}(n \cdot \log n)$ mientras que armar la lista doblemente enlazada es $\mathcal{O}(n)$. En ambos casos, y a lo largo de toda la explicación, n es el tamaño de la entrada del problema, o sea la cantidad de transacciones en ambos arreglos.

A continuación se inicia un ciclo `for` en el que se recorre el arreglo s. Allí se realiza una asignación simple de complejidad igual a $\mathcal{O}(1)$ seguida por un ciclo `for` anidado.

Dentro de este segundo bucle, se realizan comparaciones, asignaciones, se borra un elemento de la lista doblemente enlazada y se agrega a la lista `solucion` el s_i actual. Todas estas operaciones son de tiempo constante. El dato relevante en este punto es que el tamaño del segundo `for` se reduce en una unidad en cada iteración. Por supuesto esta última afirmación aplica en caso de que encuentre un intervalo compatible, si no se detiene la ejecución del programa.

Finalmente, el algoritmo concluye con la devolución de la lista de soluciones, lo que representa otra operación $\mathcal{O}(1)$.

Si bien la reducción del tamaño de *t_ord* en combinación con el ordenamiento que realizamos reduce el tiempo que le lleva al algoritmo encontrar la solución en la práctica, la complejidad temporal de los `for` anidados es, en el peor de los casos, cuadrática.

Entonces, la complejidad total es la suma de las operaciones constantes, el ordenamiento, la creación de la lista doblemente enlazada y los bucles anidados. Esto da como resultado una com-

n	Tiempos (s)	
	v1	v2
5	0.00002849102020263 7672	0.00006318092346191406
10	0.00002288818359375 0	0.00003361701965332031
50	0.00017410516738891602	0.00010186433792114258
100	0.00062161684036254 88	0.00008678436279296875
250	0.00353819131851196 3	0.00031113624572753906
500	0.01481926441192627	0.00089967250823974 61
750	0.03784406185150146 5	0.00298917293548584
1000	0.0662226676940918	0.00224542617797851 56
2500	0.440716028213501	0.01879429817199707
5000	1.775471568107605	0.17942780256271362
7500	3.8830437064170837	0.14208054542541504
8500	5.067214488983154	0.1766858696937561
10000	7.056303799152374	0.2588738799095154
12000	11.14504736661911	0.36099910736083984
13500	15.62834906578064	0.45282119512557983
15000	20.277705669403076	0.5644943118095398
17500	33.515971660614014	0.7412996888160706
18000	37.37140130996704	0.8770689368247986
20000	45.71362328529358	0.9628885984420776
21000	54.61054766178131	1.100819170475006
25000	88.26249289512634	1.618801772594452
30000	142.11878699064255	2.358023941516876
32500	171.35690653324127	2.8144072890281677
35000	201.94183951616287	3.2802500128746033
40000	273.26329815387726	4.368978440761566
45000	327.3479416370392	5.417539834976196
50000	368.3290709257126	6.963560163974762
55000	398.8085901737213	8.455352246761322
60000	453.8992635011673	9.789844274520874

Cuadro 1: Comparación de tiempos de ejecución de las dos implementaciones utilizando el mismo set de datos

plejidad $\mathcal{O}\left(n^2\right)$ ya que el término cuadrático abarca al resto.

4.6. Variabilidad de valores

Para la primera implementación habíamos concluido que la variabilidad de los valores no afectaba al tiempo de ejecución.

Esto tenía que ver con que en ningún momento se evitaba la evaluación completa de t para cada s_i. Por lo tanto, la superposición de los intervalos, su longitud o la distribución de los valores no impactaban en el tiempo que llevaba obtener la solución (en caso de haber una) ya que de todas formas se estudiaban todos los intervalos.

Lo que sí reducía el tiempo de ejecución era encontrar lo más temprano posible un s_i que no coincidiera con ningún intervalo disponible. Cuán rápido terminaba dependía entonces de cuán cercano al inicio del arreglo estaba el primer s_i no emparejable.

En nuestra implementación definitiva la situación es completamente diferente.

Para probarlo armamos dos generadores de valores diferentes. El primero, crea a T de forma más controlada, siguiendo este patrón

```
    for i in range(volumen):
        t = t_inicial + i * INCREMENTO_T
        e = e_inicial + i * INCREMENTO_E
```

```
4         s = random.randint(t - e, t + e)
```

donde `t_inicial` es igual a 100 y `e_inicial` igual a 10. Los incrementos son 10 y 5 respectivamente. Los valores de s_i se eligen aleatoriamente dentro del intervalo creado.

El segundo, genera intervalos aleatorios.

En la siguiente tabla se puede observar la diferencia de tiempos ejecutando el algoritmo definitivo con sets de datos de igual tamaño pero creados con los generadores mencionados.

n	Tiempos (s)	
	controlado	random
5	0.0000642538070678711	0.00006318092346191406
10	0.00002300739288330078	0.00003361701965332031
50	0.00006777048110961914	0.00010186433792114258
100	0.0000908970832824707	0.00008678436279296875
250	0.00032573938369750977	0.00031113624572753906
500	0.0004563331604003906	0.0008996725082397461
750	0.002078115940093994	0.00298917293548584
1000	0.000686943531036377	0.0022454261779785156
2500	0.0020709633827209473	0.01879429817199707
5000	0.003560304641723633	0.17942780256271362
7500	0.00539475679397583	0.14208054542541504
8500	0.03204190731048584	0.1766858696937561
10000	0.058083415031433105	0.2588738799095154
12000	0.008304357528686523	0.36099910736083984
13500	0.009084343910217285	0.45282119512557983
15000	0.05107593536376953	0.5644943118095398
17500	0.03207904100418091	0.7412996888160706
18000	0.031321704387664795	0.8770689368247986
20000	0.05261009931564331	0.9628885984420776
21000	0.01337897777557373	1.100819170475006
25000	0.07914060354232788	1.618801772594452
30000	0.04005599021911621	2.358023941516876
32500	0.06251156330108643	2.8144072890281677
35000	0.08647316694259644	3.2802500128746033
40000	0.053781330585479736	4.368978440761566
45000	0.09572511911392212	5.417539834976196
50000	0.09666013717651367	6.963560163974762
55000	0.10086327791213989	8.455352246761322
60000	0.12426817417144775	9.789844274520874

Cuadro 2: Comparación de tiempos de ejecución usando dos generadores diferentes

Se puede observar con claridad la diferencia que hay entre los sets de datos, sobre todo cuando n presenta bastante volumen. Evidentemente, para tamaños menores a 1000 el comportamiento es más parecido a constante y a partir de allí comienzan a diferenciarse cada vez más.

Por la naturaleza de os ejemplos creados con el generador controlado, la asignación es $s_i \to (t_i, e_i)$ para todo i. En cada iteración del `for` interno se toma el primer elemento de T y esto hace que los `for` anidados tengan un comportamiento más parecido a $\mathcal{O}(n)$ y por lo tanto que la complejidad total tienda a $\mathcal{O}(n \cdot \log n)$ a causa del ordenamiento inicial como se puede observar en la Figura 1.

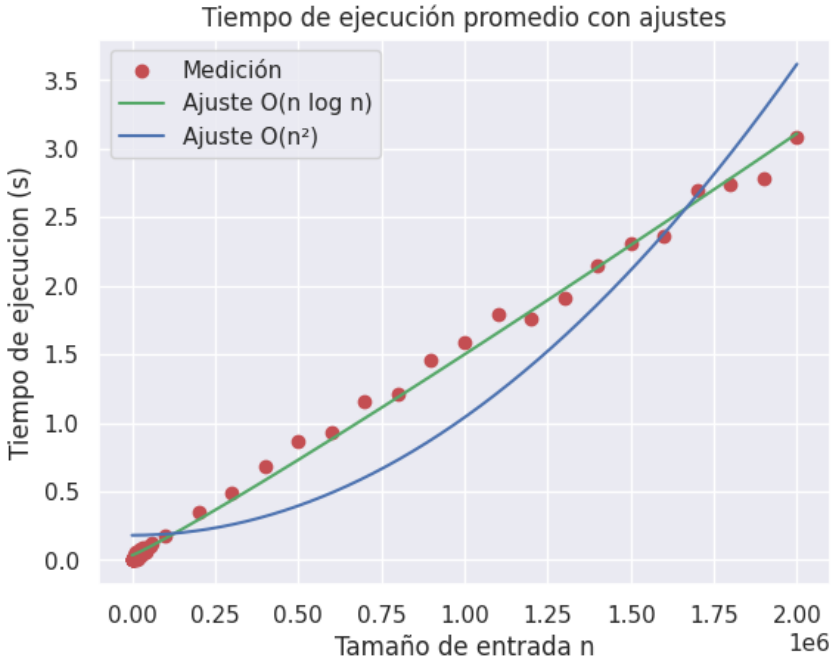

Figura 1: Tiempo de ejecución vs tamaño de entrada. Algoritmo definitivo con set de datos controlado

Si bien en el próximo apartado justificaremos gráficamente la complejidad del peor caso, queríamos mostrar acá cómo la variabilidad de los valores puede generar, en un caso controlado de valores de entrada, un mejor ajuste con lo que podríamos definir como la cota mínima del algoritmo $\mathcal{O}(n \cdot \log n)$ versus la complejidad teórica $\mathcal{O}(n^2)$.

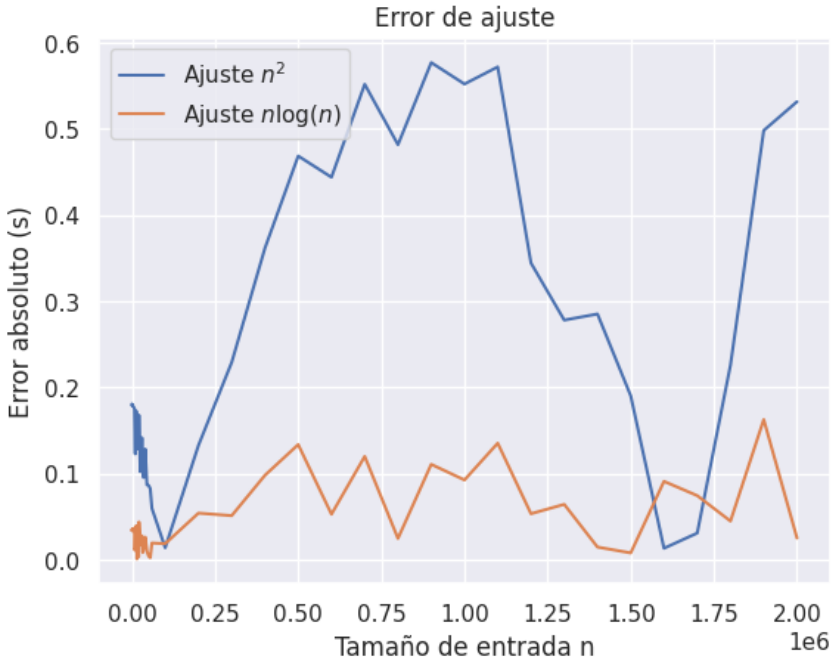

Figura 2: Error absoluto vs tamaño de entrada para el ajuste cuadrático y el lineal-logarítmico

En la Figura 2 se observa que el error absoluto para el ajuste lineal-logarítmico es mucho menor al del ajuste cuadrático.

5. Mediciones

Para corroborar que la complejidad hallada analíticamente sea correcta, realizamos mediciones de tiempo de ejecución para problemas de distintos tamaños n.

Generamos los arreglos aleatoriamente pero asegurándonos de que tuvieran solución y también usamos algunos de los ejemplos provistos por la cátedra.

Calculamos el promedio de 4 ejecuciones para cada ejemplo con el fin de minimizar el impacto de cualquier perturbación ajena a nuestro algoritmo. Esos tiempos promedio fueron los que se graficaron versus el tamaño de los arreglos. En el Cuadro 3 se muestran los valores obtenidos.

n	**Tiempo (s)**
5	6.318092346191406e-05
10	3.361701965332031e-05
50	0.00010186433792114258
100	8.678436279296875e-05
250	0.00031113624572753906
500	0.0008996725082397461
750	0.00298917293548584
1000	0.0022454261779785156
2500	0.01879429817199707
5000	0.17942780256271362
7500	0.14208054542541504
8500	0.1766858696937561
10000	0.2588738799095154
12000	0.36099910736083984
13500	0.45282119512557983
15000	0.5644943118095398
17500	0.7412996888160706
18000	0.8770689368247986
20000	0.9628885984420776
21000	1.100819170475006
25000	1.618801772594452
30000	2.358023941516876
32500	2.8144072890281677
35000	3.2802500128746033
40000	4.368978440761566
45000	5.417539834976196
50000	6.963560163974762
55000	8.455352246761322
60000	9.789844274520874

Cuadro 3: Tiempos de ejecución para distintos valores de n (tamaño de entrada)

Es claro que al aumentar el tamaño de los arreglos de entrada, los tiempos de ejecución aumentan considerablemente. En el gráfico que sigue, la Figura 3, se puede observar que la tendencia que siguen dichos puntos es aparentemente cuadrática.

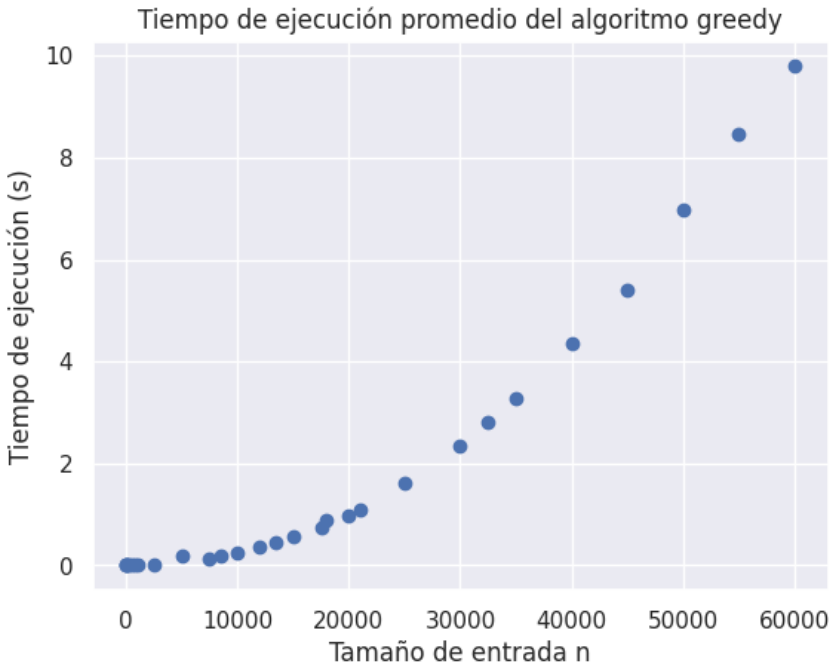

Figura 3: Tiempo de ejecución vs tamaño de entrada. Algoritmo definitivo con un set de datos aleatorio

Para corroborarlo empíricamente, fue necesario realizar un ajuste con una curva cuadrática y analizar el error asociado a dicho ajuste. Seguimos la técnica de cuadrados mínimos cuyo objetivo es hallar una función que minimice la suma de los cuadrados de las diferencias entre los valores medidos y los calculados por la función que ajusta.

Dados n puntos $p = (x_i, y_i)$, el error cuadrático queda definido como $ec_i = (y_i - f(x_i))^2$. Lo que se busca entonces es dar con la $f(x)$ que minimice la sumatoria de dichos errores, esto se conoce como el error cuadrático total.

Para hallar la función más adecuada usamos `curve_fit` de scipy que calcula los parámetros óptimos que minimizan el error cuadrático al ajustar con una función propuesta. Basándonos en el análisis teórico anteriormente mencionado, propusimos $f(x) = c_0 \cdot x^2 + c_1 \cdot x + c_2$. El resultado obtenido fue el de la Figura 4.

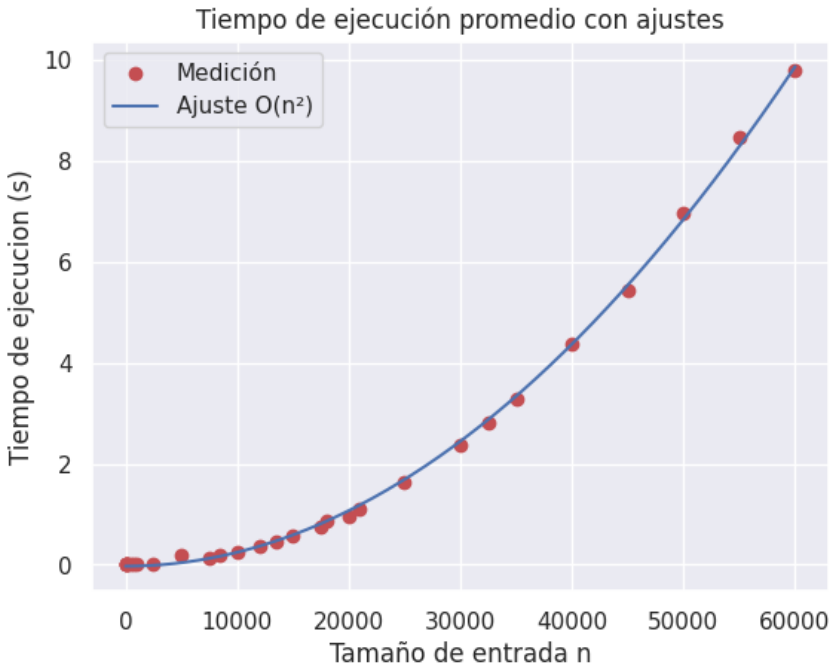

Figura 4: Tiempo de ejecución vs tamaño de entrada. Algoritmo definitivo con un set de datos aleatorio

El error cuadrático total fue igual a 0,1352904677086983, un valor bajo que nos indica que el ajuste es bueno.

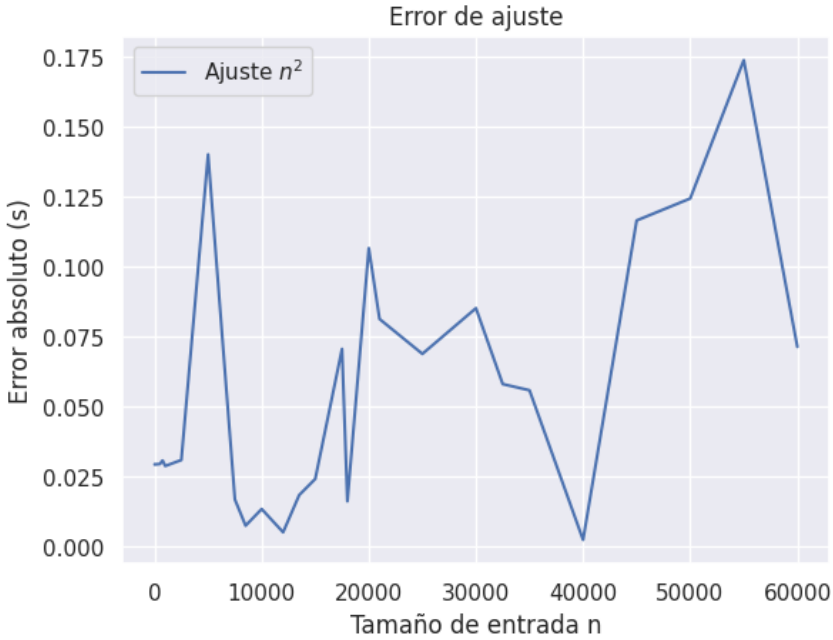

Figura 5: Error absoluto vs tamaño de entrada. Algoritmo definitivo con un set de datos aleatorio

Como se puede observar en la Figura 5, el error absoluto máximo es igual a 0.175. Teniendo en cuenta que el tiempo promedio de ejecución de todos los n fue igual a 50,90974223613739s, dicho

error representa apenas un 0,34 %, lo suficientemente bajo para reconfirmar que el ajuste es bueno.

Es por todo lo antedicho que pudimos comprobar que la complejidad temporal del algoritmo propuesto es efectivamente $\mathcal{O}\left(n^2\right)$.

6. Conclusiones

A lo largo de este trabajo hemos analizado en profundidad el problema planteado, consideramos varios algoritmos y logramos dar con uno que nos permitió resolverlo satisfactoriamente.

Concluimos que el problema puede resolverse con una estrategia Greedy y así obtener una solución óptima, la cual podría no ser única. Haciendo uso de los ejemplos provistos por la cátedra, otros que nosotros mismos creamos y realizando una demostración inductiva logramos verificar el correcto funcionamiento del algoritmo.

También observamos que la variabilidad de los valores puede impactar notablemente en el tiempo de ejecución. Por otro lado, estimamos analíticamente que la complejidad de nuestro algoritmo es $\mathcal{O}\left(n^2\right)$, y luego corroboramos esta estimación empíricamente mediante la técnica de cuadrados mínimos.

Consideramos que haber aplicado esta técnica de diseño al problema nos ayudó a lograr un mayor entendimiento de la misma. Nos encontramos con ideas que en una primera instancia parecían correctas, y ejercitamos la búsqueda de contraejemplos para derribarlas. Además, pudimos profundizar sobre la demostración formal de la optimalidad en problemas de estas características y ver con mucha claridad lo fuerte que puede ser el impacto de la variabilidad de los valores de entrada en el tiempo de ejecución.