Escritura del problema del ordenamiento de datos

Alejandro Morales Contreras[1]

[1]Departamento de Ingeniería de Sistemas, Pontificia Universidad Javeriana
Bogotá, Colombia
a.moralesc@javeriana.edu.co

9 de agosto de 2022

Resumen

En este documento se presenta la formalización del problema de ordenamiento de datos, junto con la descripción de tres algoritmos que lo solucionan. Además, se presenta un análisis experimental de la complejidad de esos tres algoritmos. **Palabras clave:** ordenamiento, algoritmo, formalización, experimentación, complejidad.

Índice

1. Introducción

Los algoritmos de ordenamiento de datos son muy útiles en una cantidad considerable de algoritmos que requieren orden en los datos que serán procesados. En este documento se presentan tres de ellos, con el objetivo de mostrar: la formalización del problema (sección 2), la escritura formal de tres algoritmos (sección 3) y un análisis experimental de la complejidad de cada uno de ellos (sección 4).

2. Formalización del problema

Cuando se piensa en el *ordenamiento de números* la solución inmediata puede ser muy simplista: inocentemente, se piensa en ordenar números. Sin embargo, con un poco más de reflexión, hay tres preguntas que pueden surgir:

1. ¿Cuáles números?
2. ¿Cómo se guardan esos números en memoria?
3. ¿Solo se pueden ordenar números?

Recordemos que los números pueden ser naturales ($\mathbb{N}$), enteros ($\mathbb{Z}$), racionales o quebrados ($\mathbb{Q}$), irracionales ($\mathbb{I}$) y complejos ($\mathbb{C}$). En todos esos conjuntos, se puede definir la relación de *orden parcial* $a < b$.

Esto lleva a pensar: si se puede definir la relación de orden parcial $a < b$ en cualquier conjunto $\mathbb{T}$, entonces se puede resolver el problema del ordenamiento con elementos de dicho conjunto.

2.1. Definición del problema del "ordenamiento de datos"

Así, el problema del ordenamiento se define a partir de:

1. una secuencia S de elementos $a \in \mathbb{T}$ y
2. una relación de orden parcial $a < b \ \forall a, b \in \mathbb{T}$

producir una nueva secuencia S' cuyos elementos contiguos cumplan con la relación $a < b$.

- Entradas:
  - $S = \langle a_i \in \mathbb{T} \rangle \mid 1 \leq i \leq n$.
  - $a < b \in \mathbb{T} \times \mathbb{T}$, una relación de orden parcial.
- Salidas:
  - $S' = \langle e_i \in Sm \rangle \mid e_i < e_{i+1} \forall i \in [1, n)$.

3. Algoritmos de solución

3.1. TimSort

El ordenamiento por TimSort es un algoritmo híbrido, que primero divide la secuencia en $|S| \div$ MinRun sub-secuencia que son ordenados por inserción y después ordena las sub-secuencias mediante merge. Este se aprovecha de lo eficiente que es inserción para pocos datos, y del merge para unir secuencias de potencias de 2.

Algoritmo 1 Ordenamiento por TimSort.

```
1: procedure TimSort(S)
2:     MinRun ← 32                                   ▷ Es una constante, puede ser 32 o 64
3:     for b ← 1 to n step MinRun do
4:         e ← Min(b + MinRun, |S|)
5:         InsertionSort(S, b, e)
6:     end for
7:     size ← MinRun
8:     while size < |S| do
9:         for b ← 1 to n step 2 * size do
10:            q ← Min(|S|, b + size)
11:            e ← Min(b + 2 * size, |S|)
12:            if q ≤ e then
13:                Merge(S, b, q, e)
14:            end if
15:        end for
16:        size ← size * 2
17:    end while
18: end procedure
```

Algoritmo 2 Ordenamiento por inserción.

```
1: procedure InsertionSort(S, b, e)
2:     for j ← b + 2 to e + 2 do
3:         i ← j
4:         while b < i ∧ s_i < s_{i−1} do
5:             Swap(s_i, s_{i−1}) ∧ i ← i − 1
6:         end while
7:     end for
8: end procedure
```

Algoritmo 3 Merge de dos sub-secuencias.

```
1: procedure Merge(A, b, q, e)
2:     n_1 ← q − b + 1 ∧ n_2 ← e − q
3:     let L[1, n_1 + 1] and R[1, n_2 + 1]
4:     for i ← 1 to n_1 do
5:         L[i] ← A[b + i − 1]
6:     end for
7:     for i ← 1 to n_2 do
8:         L[i] ← A[q + i]
9:     end for
10:    L[n_1 + 1] ← ∞ ∧ R[n_2 + 1] ← ∞ ∧ i ← 1 ∧ j ← 1
11:    for k ← b to e do
12:        if L[i] < R[j] then
13:            A[k] ← L[i] ∧ i ← i + 1
14:        else
15:            A[k] ← R[j] ∧ j ← j + 1
16:        end if
17:    end for
18: end procedure
```

3.1.1. Análisis de complejidad

Como se mencionó anteriormente, el algoritmo de TimSort en verdad es una mezcla híbrida entre el ordenamiento por inserción y merge sort. Para el peor de los casos, el algoritmo presenta una complejidad equivalente a $O(|S| \log |S|)$. Así mismo, para el caso promedio, la complejidad del algoritmo es equivalente a $\Theta(|S| \log |S|)$.

Ahora bien, para el mejor de los casos, TimSort se aprovecha del ordenamiento por inserción cuyo ciclo interior, por el hecho de ser *mientras-que*, puede que en algunas configuraciones no se ejecute (i.e. cuando la secuencia ya esté ordenada); entonces, este algoritmo tiene una cota inferior $\Omega(|S|)$, dónde solo el *para-todo* recorre la secuencia.

3.1.2. Invariante

La invariante del algoritmo es una combinación de las invariantes de inserción y merge sort. Después de la división de sub-secuencias y su posterior ordenamiento por inserción, la subsecuencia de tamaño $|S|$ está dividida en $|S| \div \textsc{MinRun}$ sub-secuencias que siguen la relación de orden parcial $a < b$ desde sus respectivos inicios hasta sus fines. Finalmente, se ejecuta el merge para unir las sub-secuencias de izquierda a derecha hasta tener una única secuencia donde todos sus elementos siguen la relación de orden parcial $a < b$.

4. Análisis experimental

En esta sección se presentarán algunos los experimentos para confirmar los órdenes de complejidad de los tres algoritmos presentados en la sección 3.

4.1. Secuencias aleatorias

Acá se presentan los experimentos cuando los algoritmos se ejecutan con secuencias de entrada de orden aleatorio.

4.1.1. Protocolo

1. Definir un rango $(b, e, s) \in \mathbb{N}^3$, donde: b es un tamaño inicial, e es un tamaño final y s es un salto. Se generarán secuencias aleatorias de diferentes tamaños desde b hasta e, adicionando cada vez s elementos.
2. Cada algoritmo se ejecutará 10 veces con cada secuencia y se guardará el tiempo promedio de ejecución.
3. Se generan los gráficos necesarios para visualizar la eficiencia del algoritmo.

4.2. Secuencias ordenadas

Acá se presentan los experimentos cuando los algoritmos se ejecutan con secuencias de entrada ordenadas de acuerdo al orden parcial $a < b$.

4.2.1. Protocolo

1. Definir un rango $(b, e, s) \in \mathbb{N}^3$, donde: b es un tamaño inicial, e es un tamaño final y s es un salto. Se generarán secuencias aleatorias de diferentes tamaños desde b hasta e, adicionando cada vez s elementos.
2. Se usará el algoritmo `sort(S)`, disponible en la librería básica de python, para ordenar dicha secuencia.
3. Cada algoritmo se ejecutará 10 veces con cada secuencia ordenada y se guardará el tiempo promedio de ejecución.
4. Se generan los gráficos necesarios para visualizar la eficiencia del algoritmo.

4.3. Secuencias ordenadas invertidas

Acá se presentan los experimentos cuando los algoritmos se ejecutan con secuencias de entrada ordenadas de forma invertida de acuerdo al orden parcial $a < b$.

4.3.1. Protocolo

1. Definir un rango $(b, e, s) \in \mathbb{N}^3$, donde: b es un tamaño inicial, e es un tamaño final y s es un salto. Se generarán secuencias aleatorias de diferentes tamaños desde b hasta e, adicionando cada vez s elementos.
2. Se usará el algoritmo `sort(S)`, disponible en la librería básica de python, para ordenar dicha secuencia.
3. Cada algoritmo se ejecutará 10 veces con cada secuencia ordenada y se guardará el tiempo promedio de ejecución.
4. Se generan los gráficos necesarios para visualizar la eficiencia del algoritmo.

5. Resultados de los experimentos

Para la realización de los experimentos, se sigue el protocolo para cada tipo de secuencias que fueron definidas en la sección 4. Los resultados de los experimentos esperan probar (o refutar) el análisis de complejidad realizado para el algoritmo en la sección 3.1.1.

Para analizar estos resultados, se hace primeramente una inspección empírica de la gráfica resultante en donde se grafica el tamaño de la secuencia (número de elementos) contra el tiempo promedio que le toma al algoritmo ordenarla. Por ejemplo, para el peor de los casos que son secuencias aleatorias o ordenadas al revés, las cuales representan una complejidad de $O(n \log n)$, se esperaría que la gráfica resultante fuera una que siguera esa función.

Después de la inspección de la gráfica, es necesario confirmar más detalladamente si realmente los resultados prueban el análisis de complejidad. Una buena forma de confirmar esto es mediante una regresión de los datos. Para simular la función $f(n) = n \log n$, un procedimiento fácil y rápido sería transformar la variable independiente (n tamaño de la secuencia) así:

$$X(n) = n \log n$$

Y ahora se procede a realizar una regresión lineal simple a partir de esta nueva variable X así:

$$f(X) = aX + b$$

donde $f(X)$ representaría la variable dependiente (tiempo que toma ordenar la secuencia). Después de la regresión, se obtienen los coeficientes del polinomio a y b.

Una medida utilizada para determinar si la regresión se cumple es el coeficiente de determinación (R^2), el cual refleja que tan bien ajustado es el polinomio resultante a los datos de entrada. Esta se calcula así:

$$R^2 = \frac{\sum_{n=1}^{N} (\hat{Y_n} - \bar{Y})^2}{\sum_{n=1}^{N} (Y_n - \bar{Y})^2}$$

en donde:

- N representa la cantidad de elementos o tamaño de la secuencia
- $\bar{Y}$ representa la media de la variable dependiente de entrada
- $\hat{Y_n}$ representa $f(X_n)$ o el resultado de evaluar el polinomio de regresión en X_n
- Y_n representa el n-ésimo dato de la variable dependiente de entrada

R^2 varía en el rango $(0, 1)$ y a mayores medidas de este se espera una mayor confianza en que los resultados tienen un buen ajuste.

5.1. Secuencias aleatorias

Para el experimento con secuencias aleatorias, se sigue el protocolo definido en la sección 4.1, dando como rango (b, e, s) los datos $(100, 10000, 100)$. Obtenidos los resultados, se realiza la regresión y se genera el gráfico presentado en la figura 1.

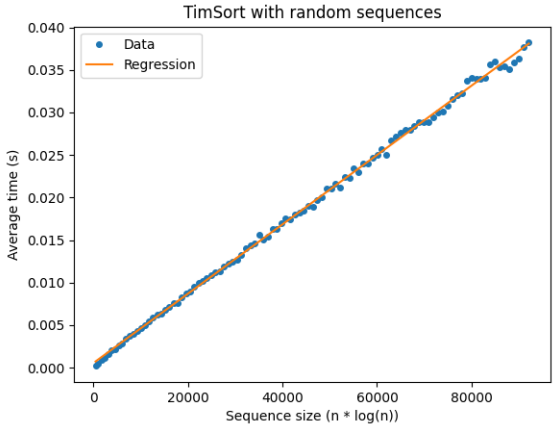

Figura 1: Gráfica con secuencias aleatorias

La inspección empírica de la gráfica parece revelar que los datos y la regresión siguen la misma tendencia. De la regresión se obtienen los siguientes resultados:

a	$4.076e-07$
b	0.0005474
R^2	0.9987612

Como se puede evidenciar, el algoritmo obtiene un $R^2 > 0.99$. Esto representa un ajuste bastante alto. Es posible confirmar que, con secuencias aleatorias, este algoritmo tiene una complejidad de $O(n \log n)$.

5.2. Secuencias ordenadas

Para el experimento con secuencias ordenadas, se sigue el protocolo definido en la sección 4.2, dando como rango (b, e, s) los datos $(100, 10000, 100)$. Obtenidos los resultados, se realiza la regresión y se genera el gráfico presentado en la figura 2.

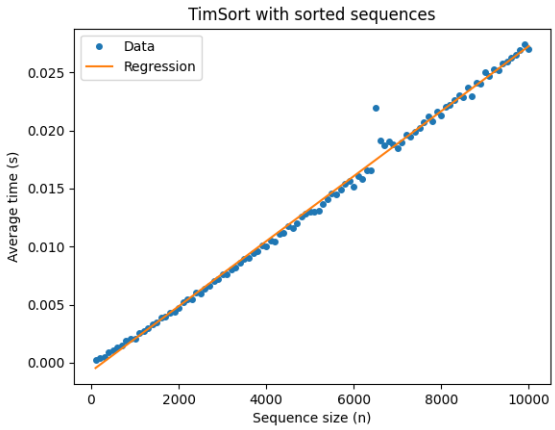

Figura 2: Gráfica con secuencias ordenadas

La inspección empírica de la gráfica parece revelar que los datos y la regresión siguen la misma tendencia. Debido a que los datos ya ordenados es el mejor caso para el algoritmo, la regresión utilizada es una lineal simple de grado 1. Se obtienen los siguientes resultados:

a	$2.802e-06$
b	-0.0007395
R^2	0.9949353

Como se puede evidenciar, el algoritmo obtiene un $R^2 > 0.99$. Esto representa un ajuste bastante alto. Es posible confirmar que, con secuencias ordenadas, este algoritmo tiene una complejidad de $\Omega(n)$.

5.3. Secuencias ordenadas invertidas

Finalmente, para el experimento con secuencias ordenadas invertidas, se sigue el protocolo definido en la sección 4.3, dando como rango (b, e, s) los datos $(100, 10000, 100)$. Obtenidos los resultados, se realiza la regresión y se genera el gráfico presentado en la figura 3.

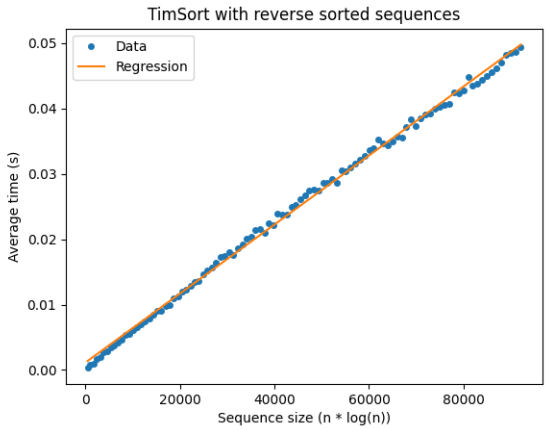

Figura 3: Gráfica con secuencias ordenadas invertidas

La inspección empírica de la gráfica parece revelar que los datos y la regresión siguen la misma tendencia. De la regresión se obtienen los siguientes resultados:

a	$5.283e-07$
b	0.001125
R^2	0.9978855

Como se puede evidenciar, el algoritmo obtiene un $R^2 > 0.99$. Esto representa un ajuste bastante alto. Es posible confirmar que, con secuencias ordenadas invertidas, este algoritmo tiene una complejidad de $O(n \log n)$.