Escritura del problema del ordenamiento de datos

Jessica Tatiana Naizaque Guevara[1]

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

8 de agosto de 2022

Resumen

En este documento se presenta la formalización del problema de ordenamiento de datos, junto con la descripción del algoritmo de TimSort que lo soluciona. Además, se presenta un análisis experimental de la complejidad del algoritmo. **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 presenta el algoritmo de TimSort, con el objetivo de mostrar: la formalización del problema (sección 2), la escritura formal del algoritmo (sección 3) y un análisis experimental de la complejidad del mismo (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. ¿Cuántos números?
3. ¿Cómo se guardan esos números en memoria?
4. ¿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. Algoritmo de solución

3.1. TimSort

La idea de este algoritmo es: combinar los algoritmos de ordenamiento "Insertion Sort" y "Merge Sort" generando un híbrido más estable. Inicialmente, se debe verificar la cantidad de elementos que contiene la secuencia S. Si este número es menor a 64, el arreglo es ordenado únicamente mediante el algoritmo de inserción. Si no, se tiene la hipótesis de que siempre va a haber una cantidad de elementos ordenados en una secuencia, ya sea ascendente o descendentemente, esta cantidad de números en este caso se llamará "run". Continuando con la hipótesis, se habla de un "min_run", que es un número entre 32 y 64 para que el tamaño del arreglo original dividido entre min_run sea igual o un poco menor a una potencia de dos y sea posible ordenar cada subarreglo por inserción. De acuerdo con esto, la secuencia principal es dividida en $\lceil len(S)/min\_run \rceil$ subarreglos que son ordenados mediante inserción. Luego, se aplica el algoritmo de fusión para mezclar los subarreglos ordenados. En esta etapa, es importante conocer que es posible aplicar el galopeo. Esto significa que si se tienen dos subsecuencias A y B que están siendo mezcladas y, por ejemplo, en B hay 7 números menores que el actual elemento de comparación de A, entonces se busca en qué posición iría, este último elemento mencionado, en B y se toman todos los númmeros de B hasta esa posición para omitir estas comparaciones. En caso de que nunca exista la posibilidad de galopeo, se continúa con las acciones del algoritmo de fusión de forma normal.

3.1.1. Pseudocódigos

A continuación, es posible encontrar los pseudocódigos de: algoritmo de TimSort, cálculo del min_run, ordenamiento por InsertionSort y mezcla de subarreglos, respectivamente. Estos son requeridos para completar el proceso de ordenamiento por el método de TimSort, ya que como se mencionó anteriormente, se requiere de un valor min_run y de los algoritmos de inserción y fusión para formar este procedimiento más óptimo.

Algoritmo 1 Ordenamiento por TimSort.

```
1:  procedure TimSort(S)
2:      minrun ← CalculateMinRun(|S|)
3:      for b ← 1 to |S| step minrun do
4:          e ← Min(b + minrun − 1, |S| − 1)
5:          InsertionSort(S, b, e)
6:      end for
7:      size ← minrun
8:      while size < |S| do
9:          for left ← 1 to |S| step size ∗ 2 do
10:             mid ← Min(left + size − 1, |S| − 1)
11:             right ← Min(left + 2 ∗ size − 1, |S| − 1)
12:             if mid < right then
13:                 Merge(S, left, mid, right)
14:             end if
15:         end for
16:         size ← size ∗ 2
17:     end while
18: end procedure
```

Algoritmo 2 Cálculo del minrun.

```
1: procedure CalculateMinRun(n)
2:     r ← 0
3:     while run < n + 1 do
4:         r| ← n&1
5:         r ≫← 1
6:     end while
7:     return n + r
8: end procedure
```

Algoritmo 3 Ordenamiento por InsertionSort.

```
1:  procedure InsertionSort(S, b, e)
2:      for i ← b + 1 to e + 1 do
3:          j ← i
4:          while S[j] < S[j − 1] ∧ b < j do
5:              S[j] ← S[j − 1]
6:              S[j − 1] ← S[j]
7:              j ← j − 1
8:          end while
9:      end for
10: end procedure
```

Algoritmo 4 Mezcla de subarreglos.

```
1: procedure MERGE(S, l, m, r)
2:     length1 ← m − l + 1
3:     length2 ← r − m
4:     right ← declare an empty array
5:     left ← declare an empty array
6:     for i ← 1 to length1 do
7:         left append S[l + i]
8:     end for
9:     for j ← 1 to length2 do
10:        right append S[m + 1 + j]
11:    end for
12:    i ← 0
13:    j ← 0
14:    k ← 1
15:    while i < length1 ∧ j < length2 do
16:        if left[i] <= right[j] then
17:            S[k] ← left[i]
18:            i ← i + 1
19:        else
20:            S[k] ← right[i]
21:            j ← j + 1
22:        end if
23:        k ← k + 1
24:    end while
25:    while i < length1 do
26:        S[k] ← left[i]
27:        k ← k + 1
28:        i ← i + 1
29:    end while
30:    while j < length2 do
31:        S[k] ← right[i]
32:        k ← k + 1
33:        j ← j + 1
34:    end while
35: end procedure
```

3.1.2. Análisis de complejidad

De acuerdo con la investigación realizada, se encontraron los siguientes datos:

- TimSort:
  El algoritmo de TimSort presenta la complejidad $O(nlogn)$ en su peor caso y en su caso promedio. De la misma forma, se halló que en su mejor caso, el orden de complejidad es $\omega(n)$. Son estos valores debido a que el algoritmo puede tomar ventaja de las posibles subsecciones ordenadas sin tener que revisar elemento por elemento en cada caso.

- InsertionSort:
  El algoritmo de InsertionSort presenta la complejidad $O(n^2)$ en su peor caso y, $\omega log(n)$ y $\theta log(n)$ en su mejor caso y en su caso promedio respectivamente.

- MergeSort
  El algoritmo de Merge presenta la complejidad $O(nlog(n))$ para todos sus casos.

3.1.3. Invariante

En este caso, la invariante sería el *run*, esto debido a que es aquel grupo de datos que se encuentran en todo momento ordenados, sin importar si están ascendentemente o descendentemente organizados.

1. Inicio: La secuencia S contiene al menos un subconjunto de elementos que se encuentran ordenados inicialmente.
2. Iteración entre subsecuencias: Cada una de las i-ésimas secuencia de elementos está ordenada, por lo que es posible determinar que cada subarreglo es un posible *run*.
3. Terminación: Los $|S|$ elementos de la secuencia están en su posición, entonces la secuencia está ordenada y, por lo tanto, todo el arreglo S' generado puede ser pensado como un *run*.

4. Análisis experimental

En esta sección se presentarán algunos de los experimentos para confirmar los órdenes de complejidad del algoritmo presentado 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. Cargar en memoria un archivo de, al menos, 200Kb.
2. 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, a partir del archivo de entrada, de diferentes tamaños desde b hasta e, adicionando cada vez s elementos.
3. Cada algoritmo se ejecutará 10 veces con cada secuencia y se guardará el tiempo promedio de ejecución.
4. Se generan los gráficos necesarios para comparar los algoritmos.

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 comparar los algoritmos.

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 comparar los algoritmos.

5. Resultados

A continuación, se observan los resultados obtenidos para cada experimento descrito en la sección 4. Para estos, se utilizaron los mismos datos de entrada, es decir, $b = 1$, $e = 30000$ y $s = 100$. Adicionalmente, es importante aclarar que se hizo uso de una imagen de 149KB para el primer experimento y que en las gráficas, el eje Y representa el tiempo en segundos y el eje X las particiones realizadas durante la ejecución.

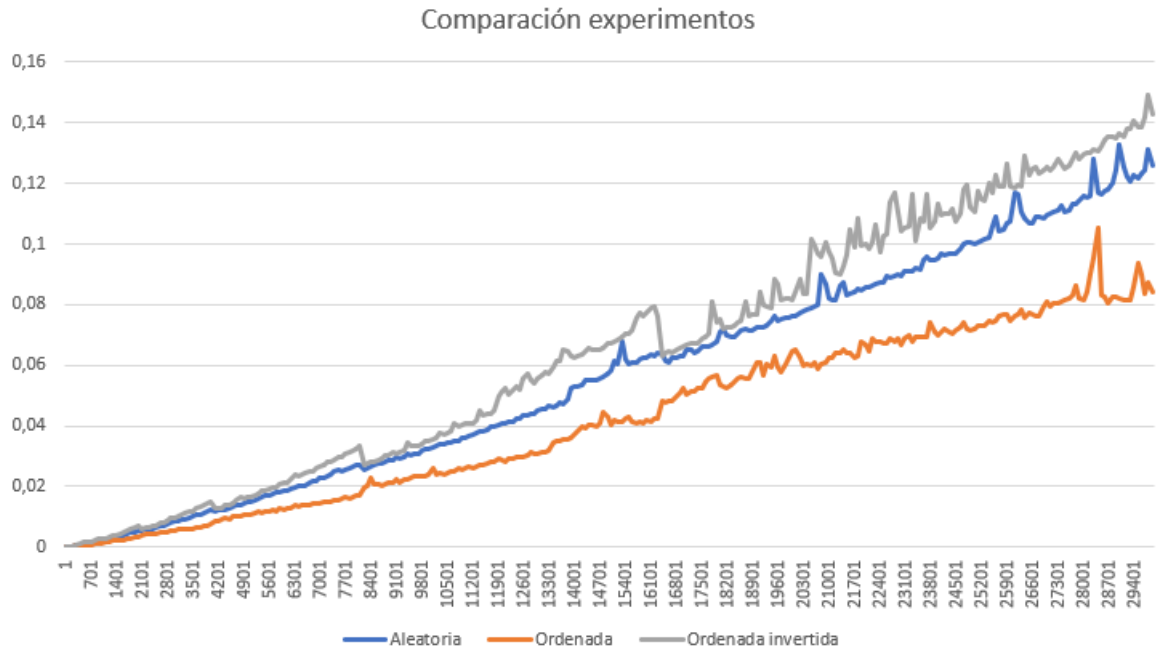

Figura 1: Ejecución experimentos con TimSort

6. Conclusiones

A partir de la sección 5, se puede observar cómo varía el tiempo de ejecución del algoritmo dependiendo de los arreglos que se usaron como entrada. Teniendo en cuenta el arreglo que está ordenado de forma invertida, se puede concluir que tuvo un tiempo de ejecución mayor que los demás. Esto puede pasar porque para el algoritmo tener una secuencia ordenada significa tener sus elementos ordenados de forma ascendente, su ordenamiento con el arreglo invertido demore más. Por otro lado, se ve cómo el arreglo aleatorio tiene un tiempo menor comparado con el arreglo ordenado de forma invertida. Dado que dentro del arreglo aleatorio pueden existir pequeños conjuntos de elementos que ya se encuentran ordenados ascendentemente, el algoritmo puede ser más eficiente en su ordenamiento.

Referencias

[1] Phyton Pool, "TimSort: Algorithm and Implementation in Python", 06-08-2021. [Online]. Available: https://www.pythonpool.com/python-timsort/.

[2] Geek for Geeks, "TimSort", 19-07-2022. [Online]. Available: https://www.geeksforgeeks.org/timsort/.

[3] Hoque, T. "TimSort", 2020. [Online]. Available: https://www.youtube.com/watch?v=_dlzWEJoU7I.

[4] Delft Stack, ."Ordenamiento Tim", 25-02-2021. [Online]. Available: https://www.delftstack.com/es/tutorial/algorithm/tim-sort/.