Escritura del problema del ordenamiento de datos utilizando método TimSort

Jorge Luis Esposito Albornoz[1]

[1]Departamento de Ingeniería de Sistemas, Pontificia Universidad Javeriana
Bogotá, Colombia
{jesposito}@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 del algoritmo que lo soluciona, así como un análisis de su complejidad e invariante. **Palabras clave:** ordenamiento, algoritmo, formalización, complejidad.

Índice

1. Introducción

El algoritmo TimSort nace en el año 2002 de la mano del desarrollador Tim Peters para su uso en Python, este es un algoritmo híbrido que deriva de los algoritmos de ordenamiento InsertionSort y MergeSort. A continuación se mostrará el funcionamiento del algoritmo, así como su complejidad e invariante.

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

La idea de este algoritmo es dividir la secuencia inicial en sub arreglos, estos sub arreglos será ordenados utilizando el algoritmo de Inserción y posteriormente se generará una nueva secuencia formada a partir de aplicar un Merge Sort a los sub arreglos.

Algoritmo 1 Ordenamiento por TimSort.

Require: $S = \langle s_i \in \mathbb{T} \rangle \wedge a < b \in \mathbb{T} \times \mathbb{T}$
Ensure: S será cambiado por $S' = \langle e_i \in S \mid e_i < e_{i+1} \forall i \in [1, n) \rangle$

```
1:  procedure TimSort(S)
2:      n ← |S|
3:      minRun ← calcMinRun(n)
4:      for j ← 1 to |S| step minRun do
5:          end ← min(j + minRun − 1, n − 1)
6:          InsertionSort(S, j, end)
7:      end for
8:      size ← minRun
9:      while size < n do
10:         for left ← 1 to |S| step 2 ∗ size do
11:             mid ← min(n − 1, left + size − 1)
12:             right ← min((left + 2 ∗ size − 1), (n − 1))
13:             if mid < right then
14:                 merge(S, left, mid, right)
15:             end if
16:         end for
17:         size ← 2 ∗ size
18:     end while
19: end procedure
```

3.1.1. Análisis de complejidad

En el mejor de los casos, TimSort tiene una cota inferior Ω(—n—), esto ocurre cuando el ciclo while del InsertionSort no se deban ejecutar debido a que el sub arreglo ya se encuentra ordenado, donde así únicamente el for recorre el arreglo una vez. Por otro lado el algoritmo se comporta igual en su peor caso y en el caso promedio dándonos así una complejidad de O(—n— log —n—) y (—n— log —n—).

3.1.2. Invariante

Después de cada iteración controlada por el contador i, los i elementos más grandes quedan al final de la secuencia.

1. Inicio: Si los sub-arreglos se encuentran ordenados a la hora de aplicar inserción.
2. Iteración: Se aplica InsertionSort en los —S— ÷ MinRun sub arreglos derivados de la secuencia original, luego se genera un S' con la unión de los sub arreglos que cumpla la relación de orden parcial a ¡b.
3. Terminación: $j = |S|$, los $|S|$ primeros elementos están ordenados, entonces la secuencia está ordenada.

Algoritmo 2 Cálculo run"mínimo

```
1: procedure calcMinRun(n)
2:     r ← 0
3:     while n >= MIN_MERGE do
4:         r| = n & 1
5:         r ≫= 1
6:     end while
7:     return n + r
8: end procedure
```

Algoritmo 3 Ordenamiento por inserción.

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

Algoritmo 4 Ordenamiento por MergeSort.

```
1: procedure MergeSort(S, left, mid, right)
2:     MergeSort_Aux(S, left, mid)
3:     MergeSort_Aux(S, mid + 1, right)
4:     Merge(S, left, mid, right)
5: end procedure
```