Escritura del problema del ordenamiento de datos

Juan Sebastián Herrera Guaitero[1]

[1]Departamento de Ingeniería de Sistemas, Pontificia Universidad Javeriana
Bogotá, Colombia
jsebastianherrera@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 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 **??**).

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. Burbuja "inocente"

La idea de este algoritmo es: comparar todos las parejas de elementos adyacentes e intercambiarlos si no cumplen con la relación de orden parcial $<$.

3.1.1. Análisis de complejidad

Por inspección de código: hay dos ciclos *para-todo* anidados que, en el peor de los casos, recorren todo la secuencia de datos; entonces, este algoritmo es $O(|S|^2)$.

Algoritmo 1 Ordenamiento por burbuja "inocente".

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 NaiveBubbleSort(S)
2:     for i ← 1 to |S| do
3:         for j ← 1 to |S| − 1 do
4:             if s_{j+1} < s_j then
5:                 Swap(s_j, s_{j+1})
6:             end if
7:         end for
8:     end for
9: end procedure
```

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: $i = 0$, la secuencia vacía está ordenada.
2. Iteración: $1 \leq i < |S|$, si se supone que los $i-1$ elementos más grandes ya están en su posición, entonces la nueva iteración llevará los i-ésimo elemento a su posición adecuada.
3. Terminación: $i = |S|$, los $|S|$ elementos más grandes están en su posición, entonces la secuencia está ordenada.

3.2. Burbuja "mejorado"

La idea de este algoritmo es: comparar todos las parejas de elementos adyacentes e intercambiarlos si no cumplen con la relación de orden parcial $<$, con la diferencia que las comparaciones se detienen en el momento que se alcanzan los elementos más grandes que ya están en su posición final.

Algoritmo 2 Ordenamiento por burbuja "mejorado".

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 ImprovedBubbleSort(S)
2:     for i ← 1 to |S| do
3:         for j ← 1 to |S| − i do        ▷ Mejora: parar cuando se encuentren los elementos más grandes.
4:             if s_{j+1} < s_j then
5:                 Swap(s_j, s_{j+1})
6:             end if
7:         end for
8:     end for
9: end procedure
```

3.2.1. Análisis de complejidad

Por inspección de código: hay dos ciclos *para-todo* anidados que, en el peor de los casos, recorren todo la secuencia de datos; entonces, este algoritmo es $O(|S|^2)$.

3.2.2. Invariante

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

1. Inicio: $i = 0$, la secuencia vacía está ordenada.
2. Iteración: $1 \leq i < |S|$, si se supone que los $i-1$ elementos más grandes ya están en su posición, entonces la nueva iteración llevará los i-ésimo elemento a su posición adecuada.
3. Terminación: $i = |S|$, los $|S|$ elementos más grandes están en su posición, entonces la secuencia está ordenada.

3.3. Inserción

La idea de este algoritmo es: en cada iteración, buscar la posición donde el elemento que se está iterando quede en el orden de secuencia adecuado.

Algoritmo 3 Ordenamiento por inserción.

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

```
1:  procedure InsertionSort(S)
2:      for j ← 2 to |S| do
3:          k ← s_j
4:          i ← j − 1
5:          while 0 < i ∧ k < s_i do
6:              s_{i+1} ← s_i
7:              i ← i − 1
8:          end while
9:          s_{i+1} ← k
10:     end for
11: end procedure
```

3.3.1. Análisis de complejidad

Por inspección de código: hay dos ciclos (un *mientras-que* anidado dentro de un ciclo *para-todo*) anidados que, en el peor de los casos, recorren todo la secuencia de datos; entonces, este algoritmo es $O(|S|^2)$.

El 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.

3.3.2. Invariante

Después de cada iteración j, los primeros j siguen la relación de orden parcial $a < b$.

1. Inicio: $j \leq 1$, la secuencia vacía o unitaria está ordenada.
2. Iteración: $2 \leq j < |S|$, si se supone que los $j-1$ elementos ya están ordenados, entonces la nueva iteración llevará un nuevo elemento y los j primeros elementos estarán ordenados.
3. Terminación: $j = |S|$, los $|S|$ primeros elementos están ordenados, entonces la secuencia está ordenada.

3.4. Merge Sort

Se basa en la estrategia de dividir y conquistar. La ordenación por fusión corta continuamente una lista en múltiples sublistas hasta que cada una de ellas tiene un solo elemento, y luego combina esas sublistas en una lista ordenada.

Algoritmo 4 Merge Sort

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

```
1:  procedure MERGESORT(S, l, m, r)
2:      len1, len2 ← m − l + 1, r − m
3:      left, right ← [], []
4:      for i ← 1 to len1 do
5:          left.append(S_{l+i})
6:      end for
7:      for i ← 1 to len2 do
8:          right.append(S_{m+1+i})
9:      end for
10:     i, j, k ← 0, 0, 1
11:     while i < len1 ∧ j < len2 do
12:         if left_i ⩽ right_j then
13:             S_k ← left_i
14:             i+ ← 1
15:         else
16:             S_k ← right_j
17:             j+ ← 1
18:         end if
19:         k+ ← 1
20:     end while
21:     while i < len1 do
22:         S_k ← left_i
23:         k+ ← 1
24:         i+ ← 1
25:     end while
26:     while i < len2 do
27:         S_k ← right_j
28:         k+ ← 1
29:         j+ ← 1
30:     end while
31: end procedure
```

3.4.1. Análisis de complejidad

Para el algoritmo de MergeSort se cuenta con una complejidad en el peor de los casos $O(nlogn)$, cuando se requieren n comparaciones. En este caso la complejidad temporal en el peor de los casos es la misma para el caso medio Θ. Por último, el mejor caso se produce cuando la secuencia ya está ordenada y no se requieren intercambios, contamos con una complejidad $\Omega(n)$.

3.5. TimSort

La idea de este algorítmo es: dividir la secuencia en bloques conocidos como "Run", ordenarlos utilizando ordenamiento por inserción y luego fusionarlos utilizando ordenamiento Merge.

Algoritmo 5 Ordenamiento TimSort.

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

```
1: function calcMinRun(n)
2:     r ← 0
3:     while n ≥ MINIMUM do
4:         r| ← n&1
5:         n ≫← 1
6:     end while
7:     return n + r
8: end function

9: procedure TimSort(S)
10:    n ←  |S|
11:    minRun ← calcMinRun(n)
12:    for start ← 1 to |S| do
13:        end ← min(start + minRun − 1, n − 1)
14:        insertionSort(S, start, end)
15:    end for
16:    size ← minRun
17:    while size < n do
18:        for left ← 1 to n do
19:            mid ← min(n − 1, left + size − 1)
20:            right ← min((left + 2 ∗ size − 1), (n − 1))
21:            if mid < right then
22:                mergesort(S,left,mid,right)
23:            end if
24:        end for
25:        size ← 2 ∗ size
26:    end while
27: end procedure
```

3.5.1. Análisis de complejidad

Para el algoritmo de TimSort se cuenta con una complejidad en el peor de los casos $O(nlogn)$, cuando se requieren n comparaciones. En este caso la complejidad temporal en el peor de los casos es la misma para el caso medio Θ. Por último, el mejor caso se produce cuando la secuencia ya está ordenada y no se requieren intercambios y para este caso contamos con una complejidad $\Omega(n)$.

3.5.2. Invariante

1. Inicio: $len >= 16$, para cada merge
2. Iteración: Añadir la longitud de la nueva ejecución a la secuencia runLen mientras se cumpla $runLen[n-2] > runLen[n-1] + runLen[n]$ y $runLen[n-1] > runLen[n]$
3. Terminación: $n \leqslant size$, al fusionar todas las ejecuciones dan como resultado una secuencia ordenada.