Problema del «ordenamiento de elementos»

Leonardo Flórez-Valencia

6 de agosto de 2020

Resumen

En este documento se presenta el problema del ordenamiento de elementos: análisis, diseño y algunos algoritmos, que son ampliamente conocidos en la literatura, que lo solucionan.

Parte I
Análisis y diseño del problema

1. Análisis

El problema, informalmente, se puede describir como: ordenar/organizar una lista/vector/arreglo de números. Genéricamente, no se puede hablar de listas, vectores o arreglos porque estas son estructuras de datos reales que pueden (o no) existir fácilmente en un lenguaje de programación (por ejemplo, piense en la forma de implementar un vector o una lista en lenguaje ensamblador); entonces, se habla de secuencias de números:

$$S = \langle s_1, s_2, \cdots, s_n \rangle = \langle s_i \in \mathbb{Z} \wedge 1 \leq i \leq n \rangle$$

donde n es la cardinalidad (i.e. cantidad de elementos) de la secuencia e i es el índice de cada elemento (note que el primer índice es 1 y no 0).

Varias preguntas pueden surgir acá:

- ¿únicamente se pueden ordenar números enteros ($\mathbb{Z}$)?
- ¿Qué «criterio» existe para decir que una secuencia está ordenada?
- ¿Se puede usar este «criterio» para ordenar una secuencia?

Resulta obvio que los números enteros no son los únicos ordenables: al menos los naturales ($\mathbb{N}$), quebrados ($\mathbb{Q}$), reales ($\mathbb{R}$) e imaginarios ($\mathbb{I}$) son intuitivamente ordenables. Pero, se puede pensar en otros elementos que pueden ser ordenables: palabras (para editar diccionarios, por ejemplo), frutas (por su tamaño),

animales (por su nombre, su género o su número de dientes, por poner algunos ejemplos). Entonces, si nos enteros no son los únicos elementos ordenables, nuestra definición de secuencia para este problema debe cambiar un poco:

$$S = \langle s_i \in \mathbb{T} \wedge 1 \leq i \leq n \rangle$$

donde $\mathbb{T}$ es un conjunto de elementos que pueden ser ordenables.

Ahora, ¿qué quiere decir que unos elementos sean ordenables? Es decir, ¿cuál es el «criterio» de ordenamiento? Intuitivamente, sabemos que dados dos números podemos saber quién va primero y quién va después, para esto existe el símbolo $\leq$; que, formalmente, expresa una relación de orden parcial entre elementos. Entonces, si en el conjunto $\mathbb{T}$ podemos definir una relación de orden parcial $\leq$, un algoritmo de ordenamiento debe generar una permutación S', a partir de S, que cumpla con la condiciones:

$$S' = \left\langle s'_i \in S \wedge s'_{i-1} \leq s'_i \forall 1 < i \wedge 1 \leq i \leq n \right\rangle$$

es decir, que la relación de orden parcial $\leq$ se cumpla entre elementos adyacentes de la secuencia permutada. Debe notarse que la única condición para que se pueda crear la permutación es que la relación de orden parcial $\leq$ este definida en $\mathbb{T}$. Recordemos que una relación de orden parcial debe cumplir con las propiedades:

- Reflexividad: $a \leq a$.
- Antisimetría: si $a \leq b$ y $b \leq a$, entonces $a = b$.
- Transitividad: si $a \leq b$ y $b \leq c$, entonces $a \leq c$.

2. Diseño

Con las observaciones presentadas en el análisis anterior, podemos escribir el diseño de un algoritmo que solucione el problema de ordenamiento. A veces este diseño se conoce como el «contrato» del algoritmmos o las «precondiciones» y «poscondiciones» del algoritmo. El diseño se compone de entradas y salidas:

Definición. Entradas:

1. Una secuencia $S = \langle s_i \in \mathbb{T} \wedge 1 \leq i \leq n \rangle$ de $n \in \mathbb{N}$ elementos que pertenecen a un conjunto $\mathbb{T}$. En este conjunto debe estar definida la relación de orden parcial $\leq$; o la relaciones de transitividad $<$ y de equivalencia $=$.

Definición. Salidas:

1. Una permutacion $S' = \left\langle s'_i \in S \wedge s'_{i-1} \leq s'_i \forall 1 < i \wedge 1 \leq i \leq n \right\rangle$ de la secuencia de entrada S.

Parte II
Algoritmos

3. Permutativo

3.1. Algoritmo

Este algoritmo se basa en la idea de calcular todas las permutaciones posibles de la secuencia; cuando una permutación ordenada es encontrada, el algoritmo para.

Algorithm 1 Ordenamiento permutativo

```
1: procedure PermutativeSort(S)
2:     S' ← S
3:     while ¬IsSorted (S') do
4:         S' ← NextPermutation (S)
5:     end while
6:     return S'
7: end procedure
```

Este algoritmo necesita de dos algoritmos auxiliares:

1. IsSorted, que verifica si una secuencia sigue la relación de orden parcial $\leq$.
2. NextPermutation, que calcula la siguiente permutación de S.

Ahora se escribe el algoritmo IsSorted (NextPermutation se deja como ejercicio):

Algorithm 2 Indica si una secuencia está ordenada.

```
1: procedure IsSorted(S)
2:     isOrdered ← true
3:     i ← 1
4:     while isOrdered ∧ i < |S| do
5:         isOrdered ← s_i ≤ s_{i+1} ∧ isOrdered
6:         i ← i + 1
7:     end while
8:     return isOrdered
9: end procedure
```

3.2. Complejidad

El algoritmo IsSorted tiene órdenes de complejidad $O(|S|)$ y $\Omega(1)$ (¿por qué?). Ambos órdenes son calculados por inspección de código.

Como este algoritmo es usado dentro del esquema permutativo del algoritmo PermutativeSort, éste tiene ordenes de complejidad $O(|S|!\,|S|)$ y $\Omega(|S|!)$, también calculados por inspección de código.

3.3. Invariante

3.3.1. IsSorted

La bandera de control `isOrdered` indica si la secuencia $S_{1\to i}$ (la secuencia desde 1 hasta i) sigue la relacion de orden parcial $\leq$.

- **Inicio**: la secuencia vacia $S_{1\to 0} = \emptyset$ está ordenada.
- **Avance**: si la secuencia $S_{1\to i}$ está ordenada, se verifica la relacion entre S_i y S_{i+1} y se modifica `isOrdered` de acuerdo.
- **Terminación**: si la secuencia $S_{i\to n}$ está ordenada, `isOrdered` nunca dejo de ser «`true`»; si alguna pareja de elementos adyacentes no sigue la relacion, `isOrdered` cambio su estado a «`false`».

3.3.2. PermutativeSort

S' contiene una permutación ordenada de S.

- **Inicio**: S puede que esté ordenada.
- **Avance**: Si $S^{(i)'}$ (la i-ésima permutación) está ordenada termine, avance a la siguiente permutación en caso contrario.
- **Terminación**: S' tiene el valor de una permutación ordenada (ejercicio de reflexión personal: ¿se puede demostrar que una secuencia siempre tendrá una permutación ordenada?).

3.4. Notas de implementación

Este algoritmo necesita de una forma de representación de secuencias que permita la implementación de un algoritmo permutativo. Hoy en día, la mayoría de lenguajes ofrecen librerías para hacer este tipo de iteraciones permutativas. La mayoría de ellas funciona sobre contenedores lineales de acceso aleatorio (arreglos).

4. Burbuja

4.1. Algoritmo

Este algoritmo se basa en la idea: en una iteración completa sobre la secuencia, se puede hacer que los elementos más grandes «floten» hasta las últimas posiciones de la secuencia.

Algorithm 3 Ordenamiento burbuja

```
1: procedure BubbleSort(S)
2:     for i ← 1 to |S| do
3:         for j ← 1 to |S| − i do
4:             if S[j + 1] < S[j] then
5:                 aux ← S[j]
6:                 S[j] ← S[j + 1]
7:                 S[j + 1] ← aux
8:             end if
9:         end for
10:    end for
11: end procedure
```

4.2. Complejidad

El algoritmo tiene orden de complejidad $O\left(|S|^2\right)$. El cálculo se hace por inspección de código.

4.3. Invariante

4.4. Notas de implementación

5. Inserción

5.1. Algoritmo

Este algoritmo se basa en la idea: en cada iteración, mantener ordenados los elementos hasta esa posición.

Algorithm 4 Ordenamiento por inserción

```
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
```

5.2. Complejidad

El algoritmo tiene orden de complejidad $O\left(|S|^2\right)$. El cálculo se hace por inspección de código.

5.3. Invariante

5.4. Notas de implementación

6. Mezclas

6.1. Algoritmo

Este algoritmo se basa en la idea: si se obtiene de alguna forma dos secuencias ordenadas, la secuencia total resulta de mezclarlas.

Algorithm 5 Ordenamiento por mezclas (1)

```
1: procedure MergeSort(S)
2:     MergeSort_Aux(S, 1, |S|)
3: end procedure
```

Algorithm 6 Ordenamiento por mezclas (2)

```
1: procedure MergeSort_Aux(S, b, e)
2:     if b < e then
3:         q ← ⌊(b + e) ÷ 2⌋
4:         MergeSort_Aux(S, b, q)
5:         MergeSort_Aux(S, q + 1, e)
6:         Merge_Aux(S, b, q, e)
7:     end if
8: end procedure
```

Algorithm 7 Ordenamiento por mezclas (3)

```
1: procedure MERGE_AUX(S, b, q, e)
2:     n_1 ← q − b + 1
3:     n_2 ← e − q
4:     let L[1, n_1 + 1] and R[1, n_2 + 1]
5:     for i ← 0 to n_1 do
6:         L[i] ← S[b + i − 1]
7:     end for
8:     for i ← 0 to n_2 docn
9:         R[i] ← S[q + i]
10:    end for
11:    L[n_1 + 1] ← ∞ ∧ R[n_2 + 1] ← ∞
12:    i ← 1 ∧ j ← 1
13:    for k ← b to e do
14:        if L[i] < R[j] then
15:            S[k] ← L[i]
16:            i ← i + 1
17:        else
18:            S[k] ← R[j]
19:            j ← j + 1
20:        end if
21:    end for
22: end procedure
```

6.2. Complejidad

De acuerdo al teorema maestro, el orden de complejidad de este algoritmo es $\Theta\left(|S| \log_2 \left(|S|\right)\right)$.

6.3. Invariante

6.4. Notas de implementación

Parte III
Comparación de los algoritmos