

Escritura del problema del ordenamiento de datos

Jose Fernando Zuluaga¹

Nicolas Vargas Ortiz¹

¹Departamento de Ingeniería de Sistemas, Pontificia Universidad Javeriana
Bogotá, Colombia
{zuluaga_jose, vargasondaniel}@javeriana.edu.co

28 de julio 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	2
2. Formalización del problema	2
2.1. Definición del problema del “ordenamiento de datos”	2
3. Algoritmos de solución	2
3.1. Burbuja “inocente”	2
3.1.1. Análisis de complejidad	2
3.1.2. Invariante	3
3.2. Burbuja “mejorado”	3
3.2.1. Análisis de complejidad	3
3.2.2. Invariante	3
3.3. Inserción	4
3.3.1. Análisis de complejidad	4
3.3.2. Invariante	4
4. Análisis experimental	4
4.1. Secuencias aleatorias	5
4.1.1. Protocolo	5
4.1.2. Resultados	5
4.2. Secuencias ordenadas	5
4.2.1. Protocolo	6
4.2.2. Resultados	6
4.3. Secuencias ordenadas invertidas	7
4.3.1. Protocolo	7
4.3.2. Resultados	7

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

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  $i \leftarrow 1$  to  $|S|$  do
3:     for  $j \leftarrow 1$  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  $i \leftarrow 1$  to  $|S|$  do
3:     for  $j \leftarrow 1$  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: $S = \langle S_i \in \mathbb{T} \rangle \wedge a < b \in \mathbb{T} \times \mathbb{T}$

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 INSERTIONSORT( $S$ )
2:   for  $j \leftarrow 2$  to  $|S|$  do
3:      $k \leftarrow s_j$ 
4:      $i \leftarrow j - 1$ 
5:     while  $0 < i \wedge k < s_i$  do
6:        $s_{i+1} \leftarrow s_i$ 
7:        $i \leftarrow i - 1$ 
8:     end while
9:      $s_{i+1} \leftarrow k$ 
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 $\Omega(|S|)$, donde 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.

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. Estableciendo datos de prueba, los cuales son utilizados por los programas que implementan los algoritmos. Dichos parametros de prueba, son constante en el experimento, de tal forma podemos medir el rendimiento y complejidad de una forma mas equitativa entre las diferentes secuencias, ya que están operando con el mismo conjunto de datos en común.



Figura 1: Parametro Imagen

- **Figura 1:** Se utiliza una imagen de tamaño 2870KB, esta imagen fue realizada por nosotros, y se utiliza como imagen de parametro para el algoritmo, el nombre del archivo es "masterSword".
- **Rango:** Se establece un rango, inicio y final, este parametro indica al algoritmo cuando detener su ejecución en medida que se mueve en el rango.
 - **b:** Como tamaño inicial utilizamos el valor de cero (0).
 - **e:** El tamaño final utilizado es tres mil (3000).
 - **s:** El valor de los saltos realizados es de cien (100).

4.1. Secuencias aleatorias

Acá se presentan los experimentos cuando los algoritmos se ejecutan con secuencias de entrada de orden aleatorio. Reconocidas en el algoritmo como la opción numero 1.

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.

4.1.2. Resultados

Siguiendo el protocolo, y utilizando de forma correcta los valores, datos y archivos de prueba establecidos, y realizando un análisis, obtenemos la siguiente grafica con el tiempo utilizado de cada algortimo con respecto a los datos operados(Figura 2. En en la Figura 2 se observa la grafica correspondiente a la secuencia aletaria, opción 1. Donde la funcion de color **azul** representa el algoritmo burbuja "inocente", **naranja** representa el algoritmo burbuja "mejorado" **gris** representa el algoritmo de inserción

Como se observa en la grafica (Figura 2), y analizando los datos obtenidos por el algoritmo de su tiempo de ejecución, se puede concluir, que el algoritmo con una complejidad menor corresponde a ordenamiento por inserción. Sin embargo, en los tres algoritmos se presenta un aumento no constante en su complejidad

4.2. Secuencias ordenadas

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

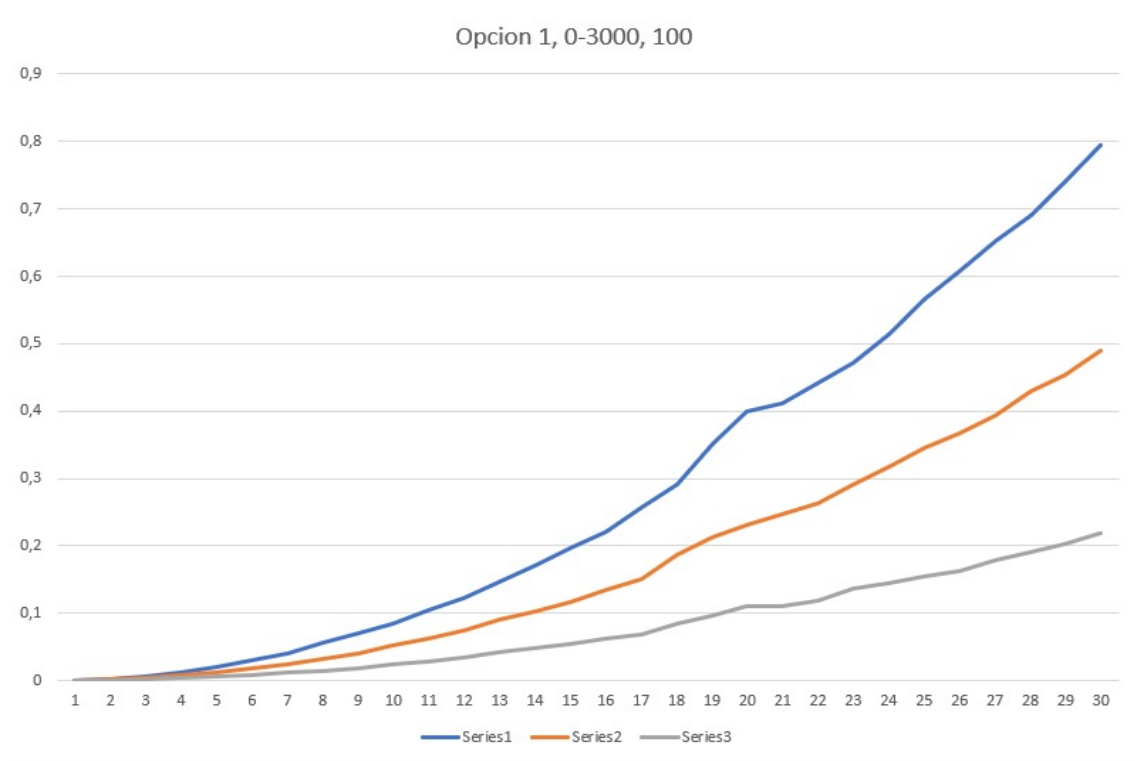


Figura 2: Grafica Secuencia Aleatoria

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.

4.2.2. Resultados

Siguiendo el protocolo, y utilizando de forma correcta los valores, datos y archivos de prueba establecidos, y realizando un análisis, obtenemos la siguiente grafica con el tiempo utilizado de cada algortimo con respecto a los datos operados(Figura 3. En en la Figura 3 se observa la grafica correspondiente a la secuencia ordenada, opción 2. Donde la funcion de color **azul** representa el algoritmo burbuja inocente”, **naranja** representa el algoritmo burbuja ”mejoradoz **gris** representa el algoritmo de inserción

Como se observa en la grafica (Figura 3), y analizando los datos obtenidos por el algoritmo de su tiempo de ejecución, se puede concluir, que los algortimos presentan un tiempo menor a la opción 1 (secuencia aleatoria), lo cual significa que mejoro su rendiemento y por ende, su complejidad disminuye. Tambien se puede observar que el algoritmo por inserción, su tiempo es cero, siendo el mas eficiente, esto debido a que el algoritmo no esta realizando ninguna operación, ya que la secuencia esta ya ordenada, es decir, no tiene nada que organizar.

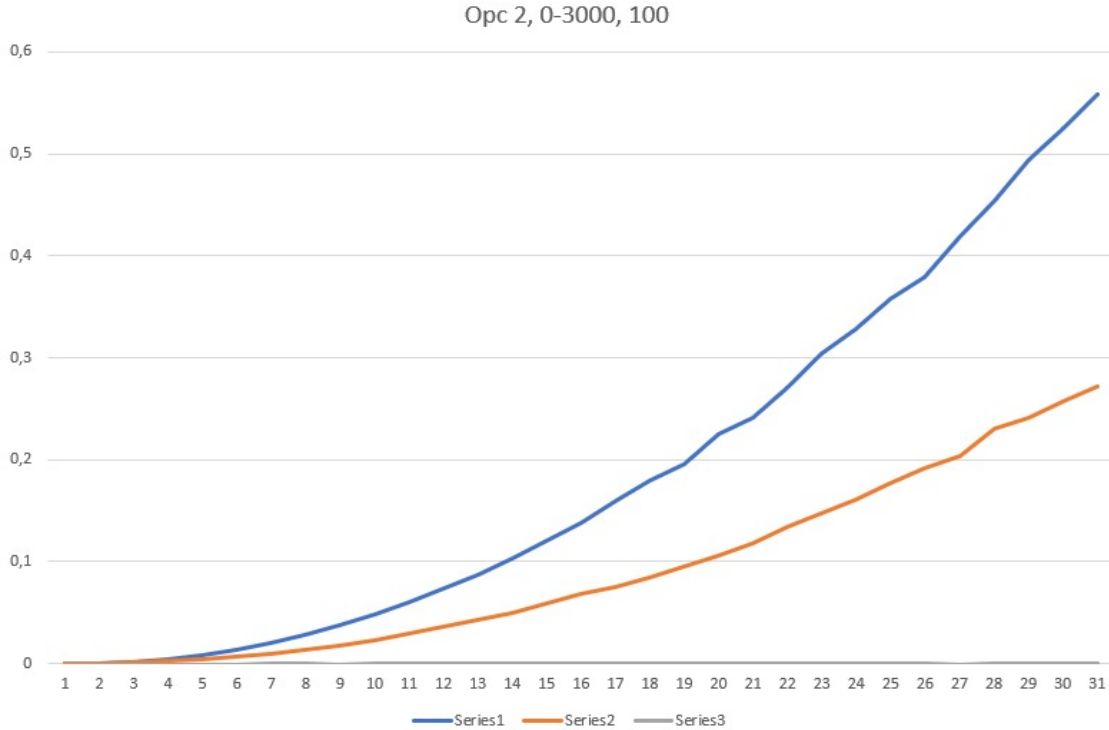


Figura 3: Grafica Secuencia Ordenada

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.

4.3.2. Resultados

Siguiendo el protocolo, y utilizando de forma correcta los valores, datos y archivos de prueba establecidos, y realizando un análisis, obtenemos la siguiente grafica con el tiempo utilizado de cada algortimo con respecto a los datos operados(Figura 4. En en la Figura 4 se observa la grafica correspondiente a la secuencia inversa, opción 3. Donde la funcion de color **azul** representa el algoritmo burbuja "inocente", **naranja** representa el algoritmo burbuja "mejorado" **gris** representa el algoritmo de inserción

Como se observa en la grafica (Figura 4), y analizando los datos obtenidos por el algoritmo de su tiempo de ejecución, se puede concluir, que los algoritmos presentan una tendencia similar a la opción 1, presenta un tiempo mayor, esto se debe a la secuencia inversa, pues tiene que organizar muchos mas elementos, como es el caso del burbuja "inocente" donde deberá flotar varias veces cada elemento en su iteración correspondiente.

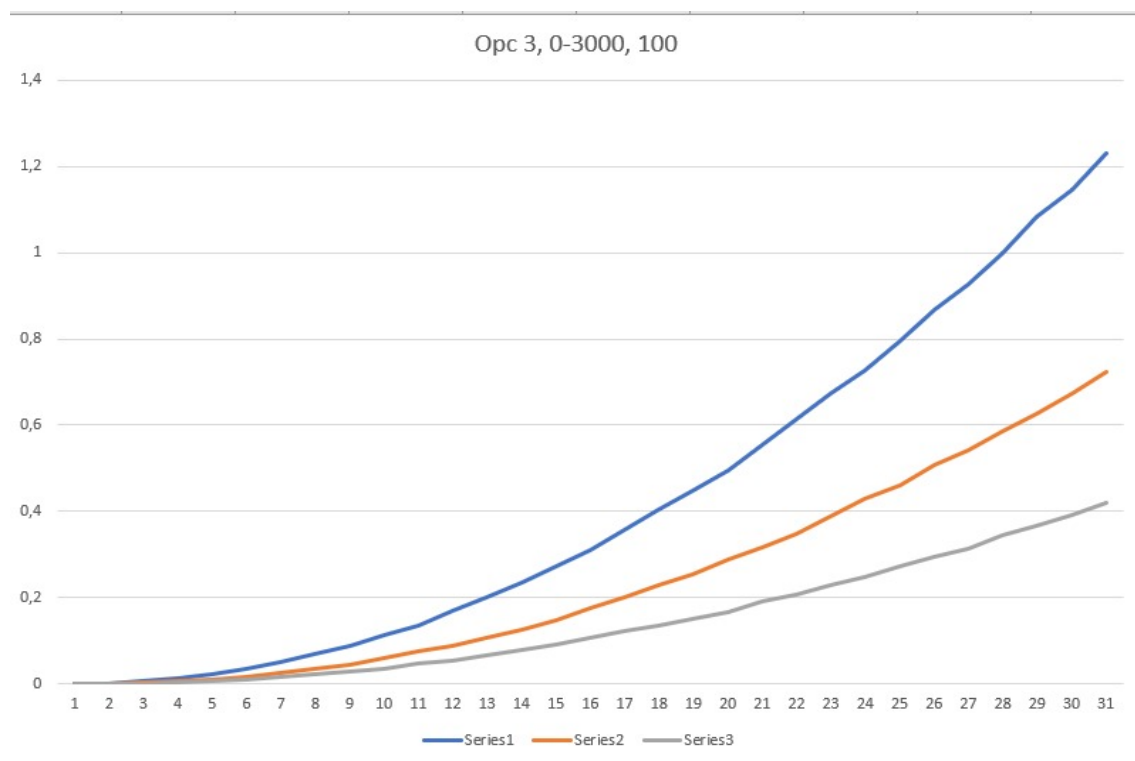


Figura 4: Grafica Secuencia Inversa