Escritura del problema del ordenamiento de datos

María José Niño Rodríguez[1] David Santiago Quintana Echavarria[1]

[1]Departamento de Ingeniería de Sistemas, Pontificia Universidad Javeriana
Bogotá, Colombia
{ma.nino, quintanae-david}@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

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:
  - $S = \langle a_i \in \mathbb{T} \rangle \mid 1 \leq i \leq n$.
  - $a < b \in \mathbb{T} \times \mathbb{T}$, 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 $<$.

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
```

3.1.1. 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 $O(|S|^2)$.

3.1.2. 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.

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.

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.En esta prueba se uso un rango de 0 a 15000 en saltos de 100 como tamaño de la secuencia.

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.

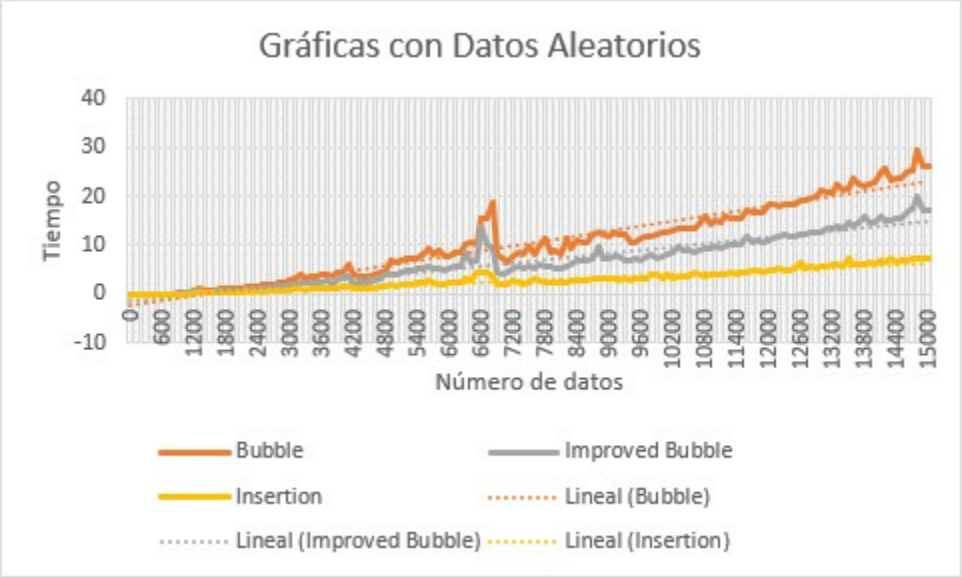

Figura 1: Gráfica con datos aleatorios

Como resultado se esperaba un comportamiento cuadrático en los tres algoritmos, ya que teóricamente la complejidad de los tres es de este orden. Como se evidencia en la Figura 1 el comportamiento de los tres es cuadrático, entre mayor es la variable dependiente, en este caso el número de datos N o el tamaño de la secuencia de entrada, mayor es el tiempo que toman los tres algoritmos, teniendo un crecimiento exponencial. Al comparar los tres algoritmos con una entrada aleatoria, a pesar de que teóricamente cuentan con la misma complejidad, el algoritmo que más tarda en el ordenamiento es el burbuja seguido del burbuja mejorado y por último o más eficaz el algoritmo de inserción. Aunque en entradas pequeñas, manejando un tamaño menor de 3000 datos estos tienden a manejar resultados muy similares a medida que aumenta la variable dependiente, mayor es la diferencia entre los tres algoritmos. En la figura se puede evidenciar que la linea de tendencia es lineal ya que la herramienta Excel no permitió una tendencia cuadrática debido al tamaño de los datos obtenidos.

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. En esta prueba se uso un rango de 0 a 15000 en saltos de 100 como tamaño de la secuencia.

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.

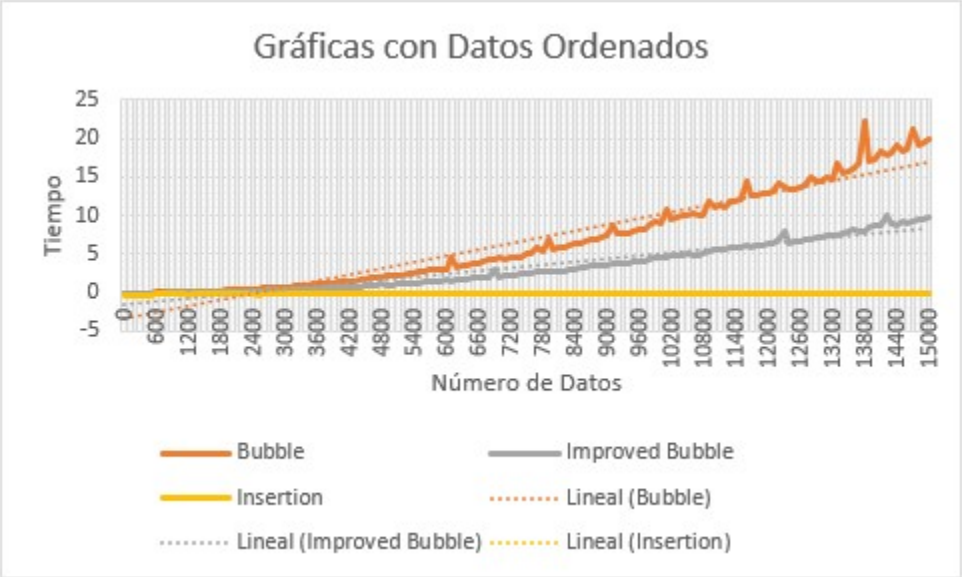

Figura 2: Gráfica con datos ordenados de acuerdo al orden parcial $a < b$

Como resultado se esperaba un comportamiento cuadrático en los tres algoritmos, ya que teóricamente la complejidad de los tres es de este orden. Como se evidencia en la Figura 2 el comportamiento de los dos algoritmos de burbuja es de tendencia cuadrática, mientras que el algoritmo de inserción es linea, entre mayor es la variable dependiente, en este caso el número de datos N o el tamaño de la secuencia de entrada, mayor es el tiempo que toman los tres algoritmos, teniendo un crecimiento cuadrático o lineal. Al comparar los tres algoritmos con una entrada ordenada, a pesar de que teóricamente cuentan con la misma complejidad de un comportamiento cuadrático, el algoritmo de inserción llega a un resultado lineal, esto debido a que la entrada se encuentra ordenada, siendo el mejor caso de los tres algoritmos, siendo esta N o lineal. El algoritmo que más tarda en el ordenamiento es el burbuja seguido del burbuja mejorado y por último o más eficaz el algoritmo de inserción. Aunque en entradas pequeñas, manejando un tamaño menor de 4000 datos estos tienden a manejar resultados muy similares a medida que aumenta la variable dependiente, mayor es la diferencia entre los tres algoritmos. El algoritmo mas eficiente con esta prueba termina siendo el algoritmo de inserción, teniendo una pendiente mucho menor que los otros dos generando la regresión lineal de cada uno, su tiempo es pequeño ya que la entrada genera que el algoritmo solamente recorra la secuencia sin la necesidad de generar cambios en este. En una entrada de tamaño grande hay una diferencia significativa entre el burbuja y burbuja mejorado, aunque la diferencia en la implementación entre estos dos es pequeña en una entrada de gran magnitud esta tiene un tiempo menor en comparación al burbuja, llegando a una mayor eficacia y mejor rendimiento, disminuyendo la complejidad.

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. En esta prueba se uso un rango de 0 a 15000 en saltos de 100 como tamaño de la secuencia.

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.

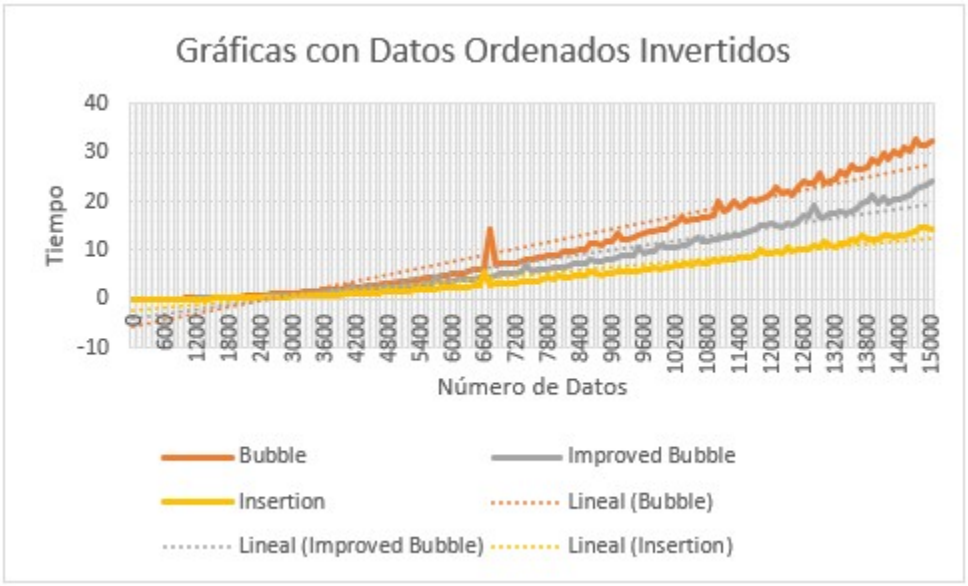

Figura 3: Gráfica con datos ordenados de forma invertida de acuerdo al orden parcial $a < b$

Como resultado se esperaba un comportamiento cuadrático en los tres algoritmos, ya que teóricamente la complejidad de los tres es de este orden. Como se evidencia en la Figura 3 el comportamiento de los tres es cuadrático, entre mayor es la variable dependiente, en este caso el número de datos N o el tamaño de la secuencia de entrada, mayor es el tiempo que toman los tres algoritmos, teniendo un crecimiento exponencial. Al comparar los tres algoritmos con una entrada aleatoria, a pesar de que teóricamente cuentan con la misma complejidad, el algoritmo que más tarda en el ordenamiento es el burbuja seguido del burbuja mejorado y por último o más eficaz el algoritmo de inserción. Aunque en entradas pequeñas, manejando un tamaño menor de 30000 datos estos tienden a manejar resultados muy similares a medida que aumenta la variable dependiente, mayor es la diferencia entre los tres algoritmos. En comparación de los resultados con las tres entradas diferentes esta última prueba fue la que mas tiempo tardo en ordenar el vector con los tres algoritmos, esto a causa del funcionamiento de los algoritmos con el vector ordenado de manera reversa. En la figura se puede evidenciar que la linea de tendencia es lineal ya que la herramienta Excel no permitió una tendencia cuadrática debido al tamaño de los datos obtenidos.