Escritura del problema del ordenamiento de datos

Leonardo Flórez Valencia[1] Jorge Luis Esposito Albornoz[1]
Juan Sebastián Herrera Guaitero[1]

[1]Departamento de Ingeniería de Sistemas, Pontificia Universidad Javeriana
Bogotá, Colombia
{florez-l, pperez}@javeriana.edu.co

{jesposito}@javeriana.edu.co

{jsebastianherrera}@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.
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. Procedimiento

1. Se ejecutó el módulo random_exp.py, para utilizar una secuencia aleatoria.
2. Se cargó un archivo de tipo jpg con un tamaño de 203kb.
3. Se utilizaron como argumentos para ejecutar el algoritmo:
   - Archivo a convertir en la secuencia.
   - tamaño inicial = 0
   - tamaño final = 30000
   - salto = 500
4. Se obtuvieron 60 datos con los tiempos promedios de ejecución de cada algoritmo.
5. Finalmente se elaboro un gráfico con la comparación entre el tiempo de ejecución y el tamaño de la secuencia para cada algoritmo: (Figura 1)

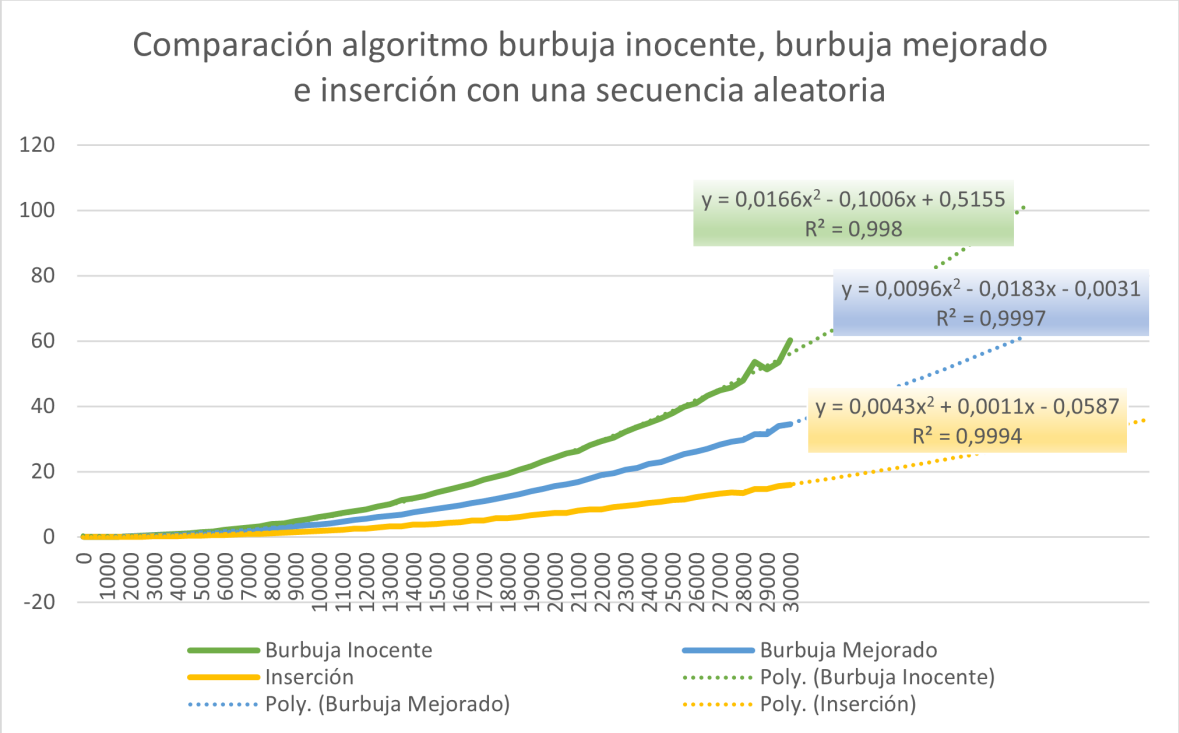

Figura 1: Tiempo de ejecución con secuencia aleatoria

4.1.3. Análisis

Con base a los resultados obtenidos en la gráfica, se puede concluir que para el caso de una secuencia aleatoria, el algoritmo de inserción es el que presenta menores tiempos de ejecución en promedio, seguido por el burbuja mejorado y estando en último lugar, el burbuja inocente. Este suceso podría explicarse debido a que el ordenamiento burbuja en cada iteración se recorre todo el arreglo, mientras que el de inserción solo recorre elementos analizados y ordenados en iteraciones previas.

De acuerdo a lo mencionado en los análisis de complejidad se asumió que la complejidad para los 3 algoritmos era de $O(|S|^2)$ para comprobar esto se realizó una regresión cuadrática en la herramienta Excel, donde evaluamos el valor de R^2, siendo esta una métrica para evaluar que tan bien se ajusta un conjunto de datos a una regresión, donde obtuvimos un valor en todos los casos muy cercano a 1. Por esto, se puede reafirmar que la complejidad de los algoritmos es R^2.

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.

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. Procedimiento

1. Se ejecutó el módulo sorted_exp.py, para utilizar una secuencia ordenada.
2. Se cargó un archivo de tipo jpg con un tamaño de 203kb.
3. Se utilizaron como argumentos para ejecutar el algoritmo:
   - Archivo a convertir en la secuencia.
   - tamaño inicial = 0
   - tamaño final = 30000
   - salto = 500
4. Se obtuvieron 60 datos con los tiempos promedios de ejecución de cada algoritmo.
5. Finalmente se elaboro un gráfico con la comparación entre el tiempo de ejecución y el tamaño de la secuencia para cada algoritmo: (Figura 2)

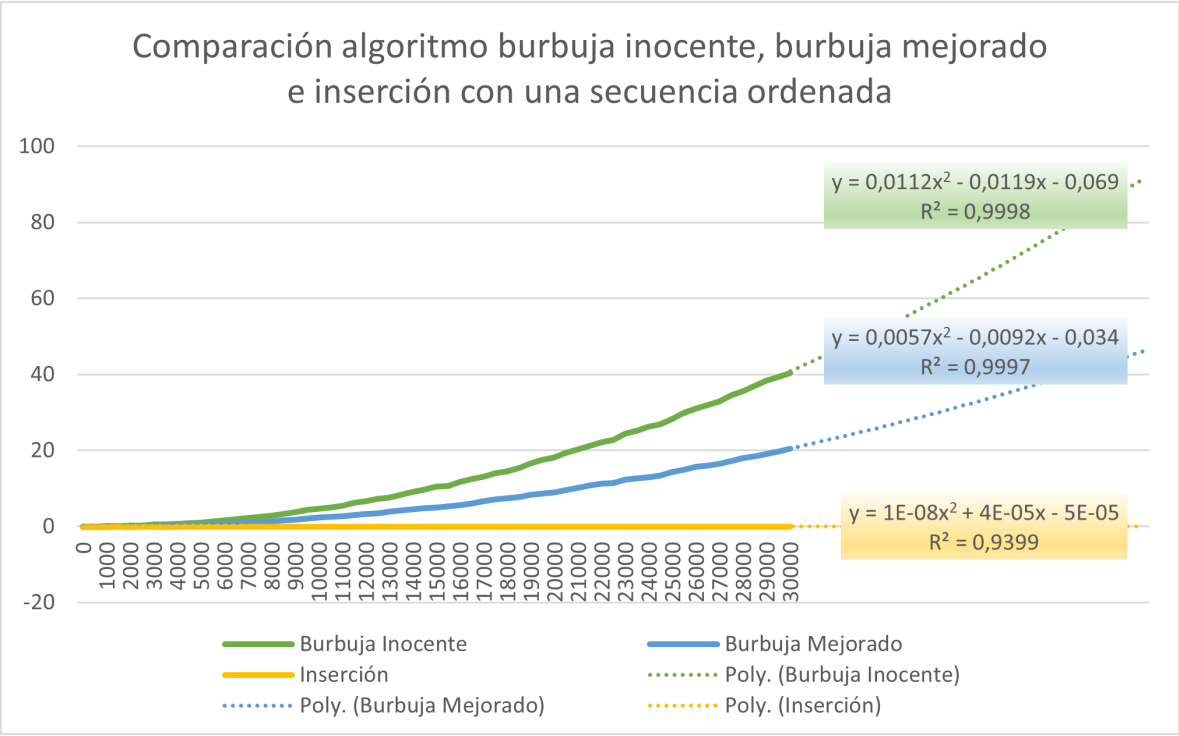

Figura 2: Tiempo de ejecución con secuencia ordenada

4.2.3. Análisis

En el caso de la secuencia ordenada se observa que nuevamente los algoritmos presentan una complejidad de $O(|S|^2)$ no obstante, cabe resaltar que el algoritmo de inserción en este caso particular cumpliría con un tiempo de ejecución $O(|S|)$ al ser su mejor caso. Esto se debe a que al estar ordenada la secuencia el tiempo que toma ordenarla es proporcional al número de elementos de esta.

Por otro lado, se observa que nuevamente el algoritmo de burbuja mejorado obtiene mejores tiempos de ejecución que su contra parte inocente, y en ambos se mantiene la tendencia cuadrática.

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. Procedimiento

1. Se ejecutó el módulo inverted_exp.py, para utilizar una secuencia ordenada de manera inversa.
2. Se cargó un archivo de tipo jpg con un tamaño de 203kb.
3. Se utilizaron como argumentos para ejecutar el algoritmo:
   - Archivo a convertir en la secuencia.
   - tamaño inicial = 0
   - tamaño final = 30000
   - salto = 500
4. Se obtuvieron 60 datos con los tiempos promedios de ejecución de cada algoritmo.
5. Finalmente se elaboro un gráfico con la comparación entre el tiempo de ejecución y el tamaño de la secuencia para cada algoritmo: (Figura 3)

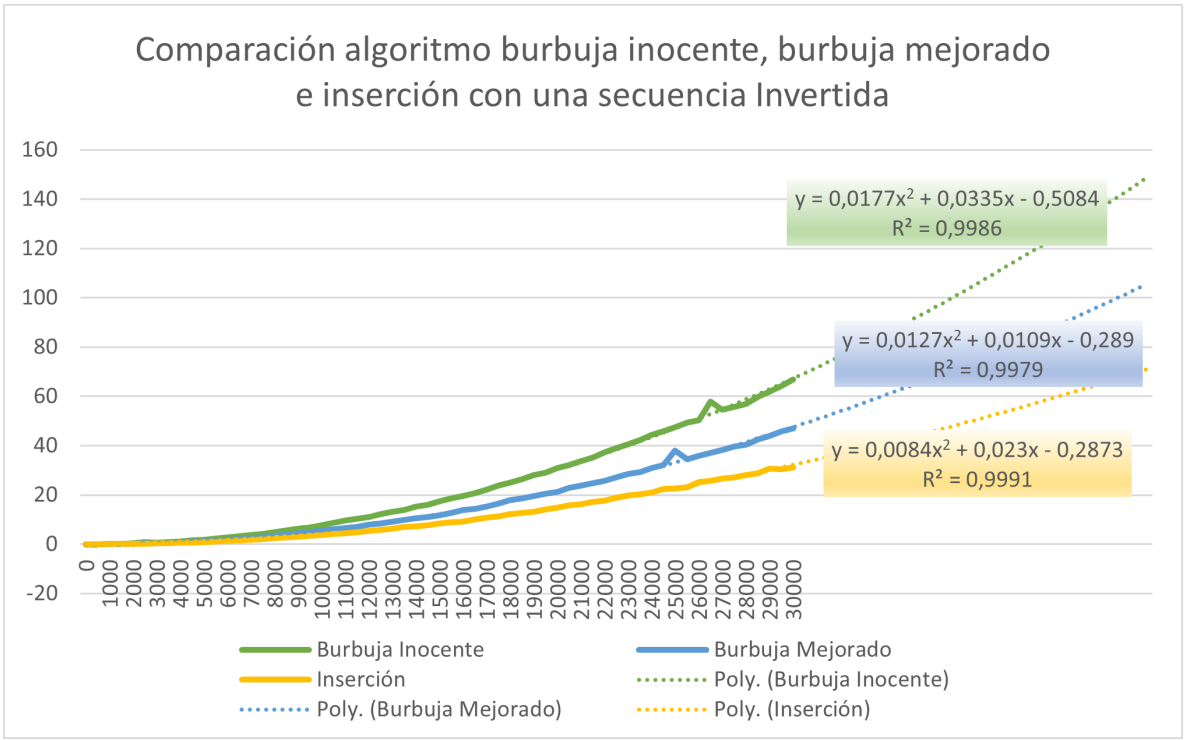

Figura 3: Tiempo de ejecución con secuencia ordenada a la inversa

4.3.3. Análisis

Por ultimo, en el caso de la secuencia ordenada a la inversa se encontró que debido a las características de cada uno de los algoritmos, esta significó el peor tiempo de ejecución para todos ellos. Para los algoritmos burbuja esta situación se debe a que al buscar comparar el elemento i con el elemento i+1, este deberá recorrer todo el arreglo en cada iteración para realizar la corrección. Por otro lado el algoritmo de inserción se ve de igual manera perjudicado en este caso, ya que para insertar el último elemento se deberán realizar n-1 comparaciones, para el penúltimo n-2 y así sucesivamente.

Aun así, se puede observar nuevamente gracias a la regresión cuadrática que la complejidad en este caso sigue siendo $O(|S|^2)$.

5. Conclusiones

De acuerdo a los anteriores análisis, podemos concluir:

1. El algoritmo de inserción fue el que mejor se desempeño en los 3 casos propuestos, en términos de tiempo de ejecución.
2. Todos los algoritmos presentaron una complejidad $O(|S|^2)$ a excepción del algoritmo de inserción que obtuvo en un caso especifico una complejidad $O(|S|)$.
3. Para todos los algoritmos el peor caso fue cuando se ordeno la secuencia a la inversa.
4. El algoritmo burbuja mejorado siempre se comportó de mejor manera que el burbuja inocente.