Escritura del problema del ordenamiento de datos

David Enrique Palacios García[1] Karen Sofia Coral Godoy[1]

[1]Departamento de Ingeniería de Sistemas, Pontificia Universidad Javeriana
Bogotá, Colombia
{david_palacios, corallg_ksofia}@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 5 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. Experimentación

1. Se utilizó el programa *run_random_experiment.py* con los parámetros de entrada:
   - `paisajeMedio.webp` con un peso de `345KB` como archivo de entrada
   - `b=0` como tamaño inicial del arreglo
   - `e=10000` como tamaño final del arreglo
   - `s=100` como saltos entre iteraciones del algoritmo
2. El programa logró finalizar sin complicaciones, con un tiempo estimado de 2 horas.
3. Se obtuvieron 100 datos con el tiempo promedio de cada uno de los algoritmos, logrando así crear la siguiente gráfica: (ver Figura 1)

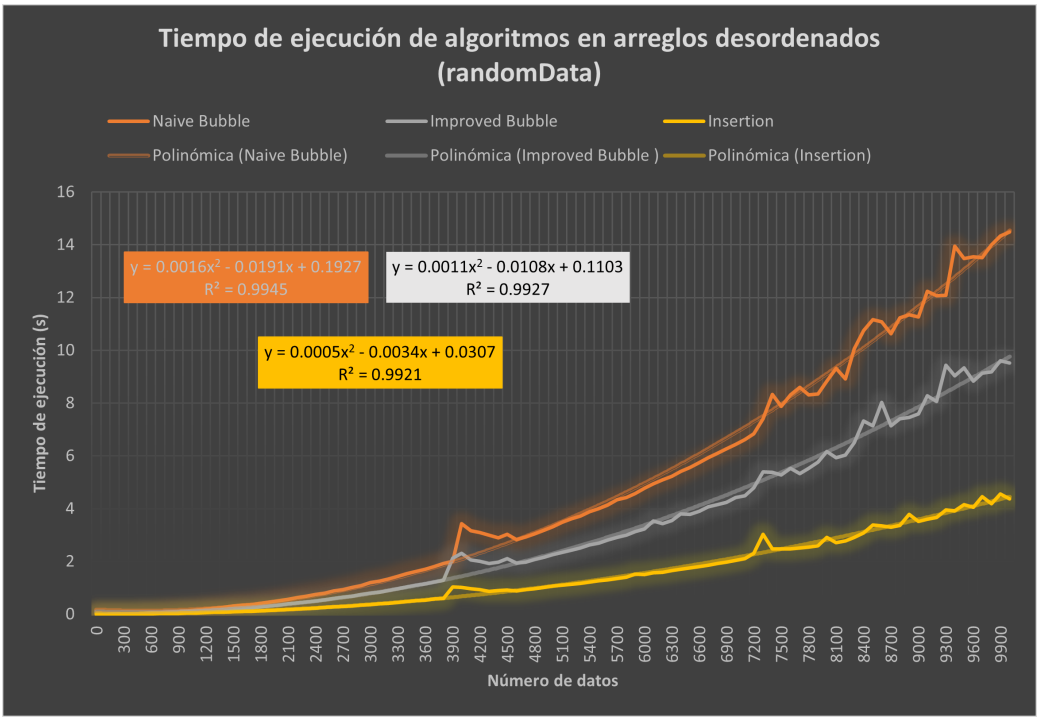

Figura 1: Tiempo de ejecución de los algoritmos con datos desordenados

4.1.3. Análisis del experimento

Como se puede notar, en este caso los 3 algoritmos tienen una complejidad $\Theta(n^2)$, sin embargo, a partir de arreglos de 1500 datos, se empiezan a notar diferencias entre ellos. La principal es que es evidente que, aunque no deja de ser cuadrática, el *InsertionSort* es mucho más rápido que el *BubbleSortInocente* y su contraparte mejorada. Eso sí, también se evidencia la mejoría del *BubbleSort* aplicando la correción estudiada anteriormente.

Así mismo, utilizando herramientas de Microsoft Excel, se pudo aplicar una regresión cuadrática, hallando las lineas de tendencia de cada algoritmo, su ecuación y valor R^2 que estaban muy cercanos a 1, lo cual indica que se aplico el modelo más preciso, y por ende se reafirma la complejidad algorítmica $O(|S|^2)$ de las soluciones presentadas.

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 5 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. Experimentación

1. Se utilizó el programa *run_sorted_experiment.py* con los parámetros de entrada:
   - `b=0` como tamaño inicial del arreglo
   - `e=10000` como tamaño final del arreglo
   - `s=100` como saltos entre iteraciones del algoritmo
2. Se creó un arreglo de tamaño `e` con números aleatorios entre -1000000 y 1000000.
3. Utilizando la función `sort(S)` de la librería básica de Pyhton, se ordena ascendentemente el arreglo y se empiezan a ejecutar los algoritmos.
4. El programa logró finalizar sin complicaciones, con un tiempo estimado de hora y media.
5. Se obtuvieron 100 datos con el tiempo promedio de cada uno de los algoritmos, logrando así crear la siguiente gráfica: (ver Figura 2)

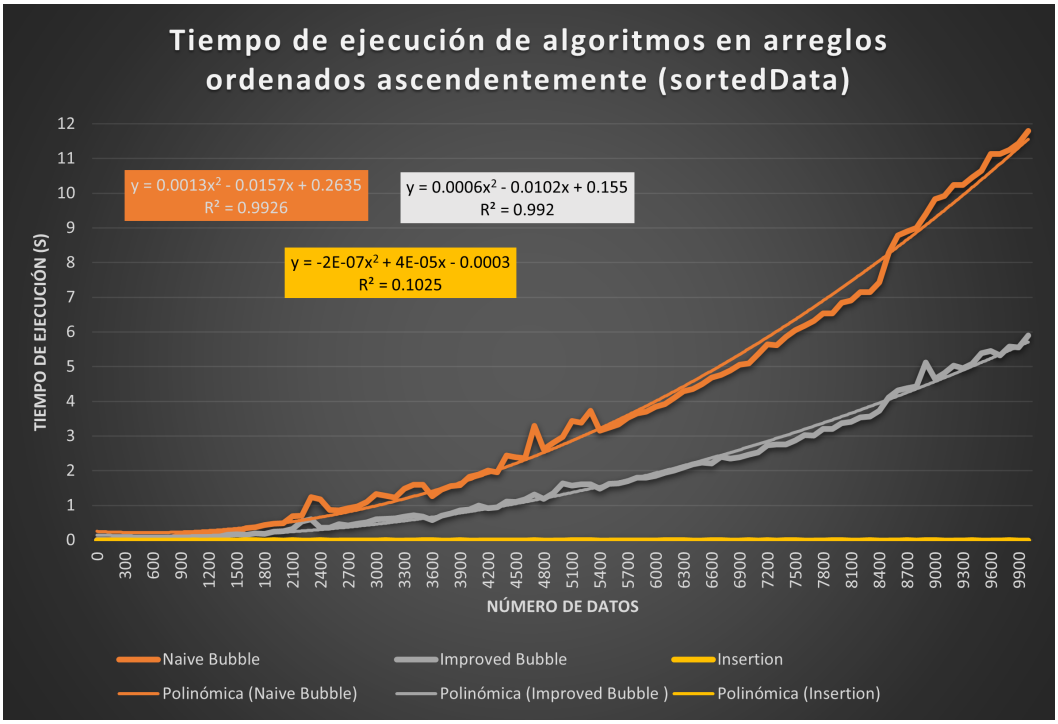

Figura 2: Tiempo de ejecución de los algoritmos con datos ordenados ascendentemente

4.2.3. Análisis del experimento

Una vez más, se evidencia la complejidad $\Theta(n^2)$ para *BubbleSort* en sus dos variantes; no obstante, mejora bastante el tiempo de ejecucicón, ya que pasa de un máximo de 14 segundos a 12 segundos aproximádamente.

Por otra parte, es clave notar el mejor caso para *InsertionSort*. El hecho de que sea casi tangente al eje x demuestra que su cota de complejidad inferior se cumple, siendo $\Omega(n)$.

Finalmente, haciendo un análisis de la regresión cuadrática aplicada a las curvas, se evidencia que el *InsertionSort* no deja de ser cuadrático, sin embargo, la constante que acompaña al x^2 es -2^{-7} que tiende a 0, por tanto, ocasiona que se convierta en una función lineal, muy cercana al eje horizontal de la gráfica.

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)` con el parámetro `reverse=True`, disponible en la librería básica de python, para ordenar dicha secuencia de manera descendente.

3. Cada algoritmo se ejecutará 5 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. Experimentación

1. Se utilizó el programa *run_reverse_sorted_experiment.py* con los parámetros de entrada:
   - `b=0` como tamaño inicial del arreglo
   - `e=10000` como tamaño final del arreglo
   - `s=100` como saltos entre iteraciones del algoritmo
2. Se creó un arreglo de tamaño `e` con números aleatorios entre -1000000 y 1000000.
3. Utilizando la función `sort(S)` de la librería básica de Pyhton, junto con el parámetro `reverse=True`, se ordena descendentemente el arreglo y se empiezan a ejecutar los algoritmos.
4. El programa logró finalizar sin complicaciones, con un tiempo estimado de 2 horas y media.
5. Se obtuvieron 100 datos con el tiempo promedio de cada uno de los algoritmos, logrando así crear la siguiente gráfica: (ver Figura 3)

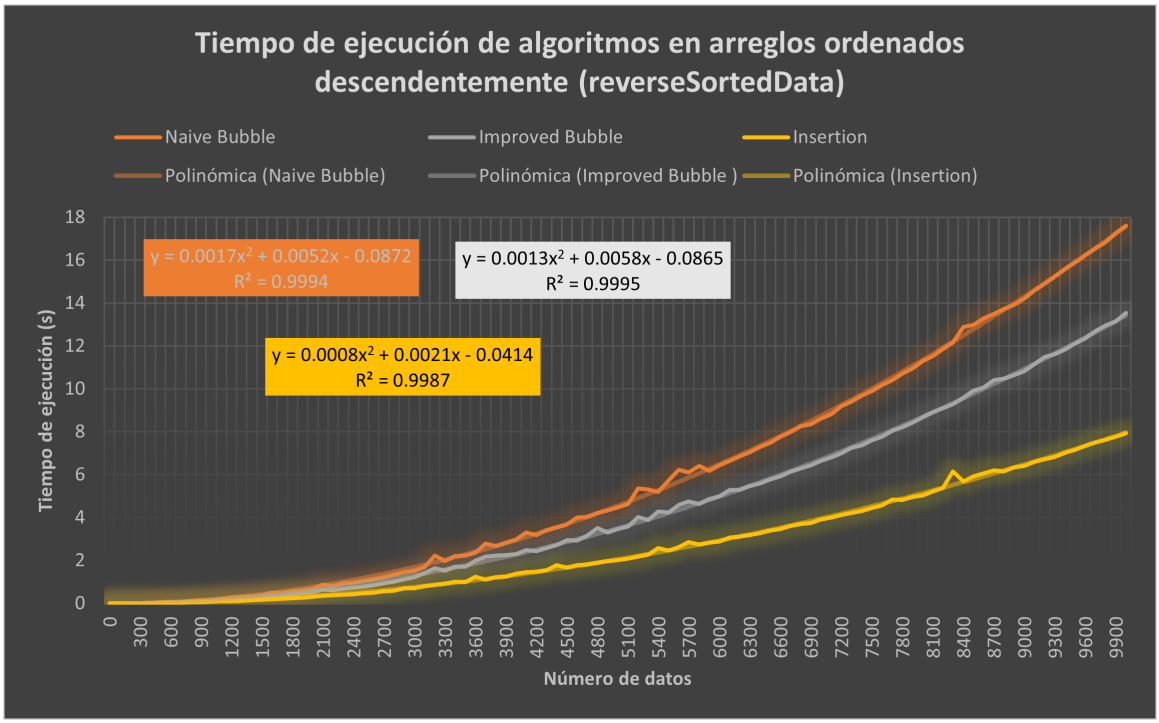

Figura 3: Tiempo de ejecución de los algoritmos con datos ordenados descendentemente

4.3.3. Análisis del experimento

Para los 3 algoritmos, este fue el peor escenario. Fue el experimento donde las iteraciones tomaron más tiempo para lograr ordenar los arreglos debido a la invariante de los algoritmos, donde, en el caso de *BubbleSort* se busca ubicar los valores más grandes hacia el final del arreglo, y en *InsertionSort*, se ajustan primero los valores menores.

Es interesante cuestionarse qué tipo de algoritmo implementa `sort(S)` de Python, pues únicamente añadiendo el parámetro `reverse=True` se efectúa el ordenamiento descendente de manera casi instantánea.

Se consultaron algunas fuentes y se encontró que, "desde la versión 2.3 de Python, éste usa el TimSort para ordenar los datos, un algoritmo que tiene una complejidad de $O(nlogn)$ siendo así uno basado en la estrategia de divide y vencerás, logrando su objetivo a velocidades prácticamente imperceptibles." [1]

5. Conclusiones

Los experimentos fueron útiles para concluir:

1. El tiempo de ejecución para el caso promedio (datos aleatorios) de estos algoritmos tiene una complejidad de $\Theta(n^2)$
2. Insertion Sort tiene una cota de complejidad de $\Omega(n)$ si el arreglo está previamente ordenado de manera ascendente.
3. Para todos los algoritmos, el peor caso es cuando el arreglo está previamente ordenado de manera descendente. Aquí, la cota de complejidad es $O(n^2)$.
4. Los algoritmos estudiados en este taller funcionan de manera educativa para aprender sobre complejidad algorítmica, pero no deberían ser usados en un proyecto real. En ese caso, se podría optar por otros como el TimSort o el HeapSort.

Referencias

[1] Educative IO, *What is the Python list sort()?*, https://www.educative.io/answers/what-is-the-python-list-sort