Problema Ordenamiento

Valeria Caycedo y Samuel Bonilla

17 de febrero de 2026

Resumen

En este documento comprende el análisis del problema de ordenamiento. Se presenta el análisis del ordenamiento y su formalización en el diseño. Se analizan 3 algoritmos de ordenamiento: BubbleSort, QuickSort e InsertionSort. Se encuentran sus complejidades y se analiza el invariante respectivo a cada algoritmo. Finalmente se realizan experimentos para analizar la complejidad teórica vs la empírica.

Parte I
Análisis y diseño del problema

1 Análisis

El problema consiste en ordenar una secuencia $S = \langle a_1, a_2, \ldots, a_n \rangle$, los elementos de la secuencia deben ser comparables, es decir $\forall_i$, $a_1 \in$ T, y en T existe la relación de orden parcial '$\leq$'. Queremos obtener una secuencia $S' = \langle a'_1, a'_2, \ldots, a'_n \rangle$ tal que $\forall_i, a'_i \in S \wedge S'[i] \leq S'[i+1]$. Por ejemplo, si $S = \langle 4, 3, 2, 6, 1, 7 \rangle$, entonces $S' = \langle 1, 2, 3, 4, 6, 7 \rangle$.

2 Diseño

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

Definición. Entradas:

1. Secuencia $S = \langle a_1, a_2, \ldots, a_n \rangle$, donde $\forall_i, a_i \in \mathbb{T}$, y en $\mathbb{T}$ existe la relación de orden parcial $' \leq'$

Definición. Salidas:

1. Secuencia $S' = \langle a'_1, a'_2, \ldots, a'_n \rangle$ tal que $\forall_i, a'_i \in S \wedge S'[i] \leq S'[i+1]$

Parte II
Algoritmos

3 BubbleSort

3.1 Algoritmo

Este algoritmo se basa en iterar sobre la secuencia llevando siempre el elemento mayor hacia el final de la secuencia.

Algorithm 1 BubbleSort

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

3.2 Complejidad

El algoritmo tiene orden de complejidad $O(n^2)$ ya que por inspección se denota que tiene 2 for anidados, donde cada uno itera aproximadamente n veces.

3.3 Invariante

Prueba: $S = \langle 5, 2, 7, 3, 1, 9, 8 \rangle$.

La condición del invariante es:

- **Condición**
  - Al inicio de cada iteración del ciclo **for** externo (parámetro j), los últimos $j-1$ elementos de la secuencia están en sus posiciones finales correctas y son los $j-1$ elementos más grandes de S, es decir:

    $S[|S|-j+2 : |S|]$ está ordenado y contiene los $j-1$ elementos más grandes

    Además, se cumple que:

    $$\forall_{k \in [1:|S|-j+1]},\ \forall_{m \in [|S|-j+2:|S|]},\quad S[k] \leq S[m]$$

    En otras palabras, todos los elementos de la porción no ordenada son menores o iguales que cualquier elemento de la porción ordenada al final.

- **Inicio**
  - Al inicio, $j = 1$. La subsecuencia $S[|S|-j+2 : |S|] = S[|S|+1 : |S|]$ es vacía (contiene 0 elementos).

    Trivialmente, una secuencia vacía está ordenada, por lo que el invariante se cumple.

    Ejemplo: $S = \langle 5, 2, 7, 3, 1, 9, 8 \rangle$, la porción ordenada es vacía: $\langle\ \rangle$.

- **Avance**
  - **Hipótesis inductiva:** Supongamos que al inicio de la iteración j, el invariante se cumple, es decir, $S[|S|-j+2 : |S|]$ contiene los $j-1$ elementos más grandes en orden.
  - **Demostración del paso:** Durante la iteración j, el ciclo **for** interno (con índice i de 1 a $|S|-j$) realiza comparaciones e intercambios consecutivos.

    Invariante del ciclo interno: Después de cada comparación en la posición i, el mayor de los elementos $S[1 : i+1]$ está en la posición $i+1$.

Al finalizar el ciclo interno ($i = |S| - j$), el elemento más grande de la porción no ordenada $S[1 : |S| - j + 1]$ ha "burbujeado"hasta la posición $|S| - j + 1$.

Por lo tanto, al terminar la iteración j:

* $S[|S| - j + 1]$ contiene el j-ésimo elemento más grande
* $S[|S| - j + 1 : |S|]$ está ordenado y contiene los j elementos más grandes

Esto preserva el invariante para la siguiente iteración $j + 1$.

Ejemplo (iteraciones):

* $j = 1$: El ciclo interno compara todos los pares adyacentes y lleva el 9 al final.
  $\Rightarrow S = \langle 5, 2, 7, 3, 1, 8, \underline{9} \rangle$
* $j = 2$: El ciclo interno trabaja en $S[1 : 6]$ y lleva el 8 a su posición.
  $\Rightarrow S = \langle 5, 2, 7, 3, 1, \underline{8, 9} \rangle$
* $j = 3$: El ciclo interno trabaja en $S[1 : 5]$ y lleva el 7 a su posición.
  $\Rightarrow S = \langle 5, 2, 3, 1, \underline{7, 8, 9} \rangle$
* $j = 4$: El ciclo interno trabaja en $S[1 : 4]$ y lleva el 5 a su posición.
  $\Rightarrow S = \langle 2, 3, 1, \underline{5, 7, 8, 9} \rangle$
* $j = 5$: El ciclo interno trabaja en $S[1 : 3]$ y lleva el 3 a su posición.
  $\Rightarrow S = \langle 2, 1, \underline{3, 5, 7, 8, 9} \rangle$
* $j = 6$: El ciclo interno trabaja en $S[1 : 2]$ y lleva el 2 a su posición.
  $\Rightarrow S = \langle 1, \underline{2, 3, 5, 7, 8, 9} \rangle$

- **Terminación**

  – El ciclo **for** externo termina cuando $j = |S|$. En ese momento, según el invariante, $S[|S| - j + 2 : |S|] = S[2 : |S|]$ contiene los $j - 1 = |S| - 1$ elementos más grandes en orden.

  Esto significa que solo queda un elemento en la posición $S[1]$, el cual debe ser el elemento más pequeño. Por lo tanto, toda la secuencia está ordenada:

$$S[1] \leq S[2] \leq \cdots \leq S[|S|]$$

Cumpliendo así la postcondición del algoritmo.

Ejemplo: Al finalizar $j = 6$ (para $n = 7$), tenemos $S = \langle 1, 2, 3, 5, 7, 8, 9 \rangle$ completamente ordenado.

Observación adicional. El invariante del ciclo interno también puede expresarse formalmente:

- **Invariante del ciclo interno:** En la i-ésima iteración del ciclo interno (dentro de la pasada j), el elemento más grande de la subsecuencia $S[1 : i + 1]$ está en la posición $S[i + 1]$.
- **Inicio (ciclo interno):** Cuando $i = 1$, después de la primera comparación, el mayor entre $S[1]$ y $S[2]$ está en $S[2]$.
- **Avance (ciclo interno):** Si $S[i + 1]$ es el mayor de $S[1 : i + 1]$ y comparamos con $S[i + 2]$, entonces el mayor de $S[1 : i + 2]$ quedará en $S[i + 2]$ después del posible intercambio.
- **Terminación (ciclo interno):** Cuando $i = |S| - j$, el mayor de $S[1 : |S|-j+1]$ está en $S[|S|-j+1]$, que es exactamente lo que necesitábamos demostrar.

4 QuickSort

4.1 Algoritmo

Tiene como propósito ordenar una subsecuencia de elementos comparables $S[p : r]$ mediante el enfoque de *divide y vencerás*. Primero se elige un elemento pivote (aleatorio en esta implementación) y se aplica PARTITION para reordenar $S[p : r]$ de modo que el pivote quede en su posición final i, cumpliendo:

$$S[p : i - 1] \leq S[i] \leq S[i + 1 : r].$$

Luego, el algoritmo ordena recursivamente las subsecuencias $S[p : i - 1]$ y $S[i + 1 : r]$.

Algorithm 2 QuickSort

```
1: procedure QuickSort(S)
2:     QuickSortAux(S, 1, |S|)
3: end procedure
```

Algorithm 3 QuickSort-Aux

```
1: procedure QuickSortAux(S, p, r)
2:     if p < r then
3:         q ← RandomizedPartition(S, p, r)
4:         QuickSortAux(S, p, q − 1)
5:         QuickSortAux(S, q + 1, r)
6:     end if
7: end procedure
```

Algorithm 4 Randomized-Partition

```
1: procedure RandomizedPartition(S, p, r)
2:     i ← RandomInt(p, r)
3:     swap(S[r], S[i])
4:     return Partition(S, p, r)
5: end procedure
```

Algorithm 5 Partition

```
1: procedure Partition(S, p, r)
2:     x ← S[r]
3:     i ← p
4:     for j ← p to r − 1 do
5:         if S[j] < x then
6:             swap(S[i], S[j])
7:             i ← i + 1
8:         end if
9:     end for
10:    swap(S[i], S[r])
11:    return i
12: end procedure
```

4.2 Complejidad

Sea $n = r - p + 1$ el tamaño de la subsecuencia $S[p : r]$ y sea $k = q - p$ el tamaño de la partición izquierda.

Algorithm 6 QuickSort (costos por línea)

```
1:  procedure QuickSort(S)
2:      QuickSortAux(S, 1, |S|)                          ▷ T(n)
3:  end procedure
4:  procedure QuickSortAux(S, p, r)
5:      if p < r then                                    ▷ O(1)
6:          q ← RandomizedPartition(S, p, r)             ▷ Θ(n)
7:          QuickSortAux(S, p, q − 1)                    ▷ T(k)
8:          QuickSortAux(S, q + 1, r)                    ▷ T(n − k − 1)
9:      end if                                           ▷ O(1)
10: end procedure
11: procedure RandomizedPartition(S, p, r)
12:     i ← RandomInt(p, r)                              ▷ O(1)
13:     swap(S[r], S[i])                                 ▷ O(1)
14:     return Partition(S, p, r)                        ▷ Θ(n)
15: end procedure
16: procedure Partition(S, p, r)
17:     x ← S[r]                                         ▷ O(1)
18:     i ← p                                            ▷ O(1)
19:     for j ← p to r − 1 do                            ▷ n − 1 iter.
20:         if S[j] < x then                             ▷ O(1)
21:             swap(S[i], S[j])                         ▷ O(1)
22:             i ← i + 1                                ▷ O(1)
23:         end if
24:     end for
25:     swap(S[i], S[r])                                 ▷ O(1)
26:     return i                                         ▷ O(1)
27: end procedure
```

Costos por línea (resumen).

Recurrencia.

$$T(n) \;=\; T(k) + T(n-k-1) + \Theta(n).$$

Resultados.

- **Peor caso:** $T(n) = \Theta(n^2)$.
- **Mejor caso (balanceado):** $T(n) = \Theta(n \log n)$.

- **Esperado (pivote aleatorio):** $\mathbb{E}[T(n)] = \Theta(n \log n)$.

Espacio (pila de recursión).

- **Peor caso:** $\Theta(n)$.
- **Esperado:** $\Theta(\log n)$.

4.3 Invariante

Pruebas: $S = \langle 5, 2, 7, 3, 1, 9, 8 \rangle$.
La condición del invariante es:

- **Condición**
  - En la j-ésima iteración del ciclo **for** en PARTITION, con pivote $x = S[r]$, el índice i indica la frontera tal que:

    $$S[p : i-1] < x \quad \text{y} \quad S[i : j-1] \geq x,$$

    es decir, todos los elementos ya procesados en $S[p : j-1]$ quedan separados en dos regiones: "menores que el pivote" (a la izquierda) y "mayores o iguales" (a la derecha).
- **Inicio**
  - Al inicio, $j = p$ e $i = p$. Los segmentos $S[p : i-1]$ y $S[i : j-1]$ son vacíos, por lo que se cumple trivialmente que $S[p : i-1] < x$ y $S[i : j-1] \geq x$.
- **Avance**
  - Si $S[j] < x$, se hace SWAP$(S[i], S[j])$ y luego $i \leftarrow i + 1$. Así, el elemento $S[j]$ (menor que el pivote) pasa a la región izquierda $S[p : i-1]$, y la frontera i avanza manteniendo la condición.
  - Si $S[j] \geq x$, no se intercambia nada; al avanzar j, el elemento queda en la región derecha $S[i : j-1]$ y la condición también se mantiene.
- **Terminación**

– Cuando termina el ciclo, $j = r$ y ya se procesó todo $S[p : r-1]$, por lo que se cumple:

$$S[p : i-1] < x \quad \text{y} \quad S[i : r-1] \geq x.$$

Luego se ejecuta SWAP$(S[i], S[r])$, colocando el pivote en su posición final i y quedando:

$$S[p : i-1] < S[i] \quad \text{y} \quad S[i+1 : r] \geq S[i],$$

cumpliendo la postcondición de PARTITION.

5 InsertionSort

5.1 Algoritmo

Tiene como propósito ordenar una secuencia de elementos comparables $S = \langle a_1, a_2, \ldots, a_n \rangle$ mediante un enfoque *incremental*.

El algoritmo recorre la secuencia de izquierda a derecha, comenzando desde el segundo elemento. En cada iteración j, se toma el elemento actual $S[j]$ (llamado *clave*) y se inserta en su posición correcta dentro de la subsecuencia ya ordenada $S[1 : j-1]$.

Para lograr esto, se compara la clave con los elementos a su izquierda, desplazándolos una posición hacia la derecha mientras sean mayores que la clave, hasta encontrar la posición apropiada donde insertar la clave. Este proceso es similar a ordenar cartas en la mano: se toma una carta nueva y se coloca en su lugar correcto entre las cartas ya ordenadas.

Algorithm 7 InsertionSort

```
1:  procedure InsertionSort(S)
2:      for j ← 2 to |S| do
3:          key ← S[j]
4:          i ← j − 1
5:          while i > 0 and S[i] > key do
6:              S[i + 1] ← S[i]
7:              i ← i − 1
8:          end while
9:          S[i + 1] ← key
10:     end for
11: end procedure
```

5.2 Complejidad

Sea $n = |S|$ el tamaño de la secuencia S.

Algorithm 8 InsertionSort (costos por línea)

```
1:  procedure InsertionSort(S)
2:      for j ← 2 to |S| do                          ▷ n − 1 iter.
3:          key ← S[j]                               ▷ O(1)
4:          i ← j − 1                                ▷ O(1)
5:          while i > 0 and S[i] > key do            ▷ t_j iter.
6:              S[i + 1] ← S[i]                      ▷ O(1)
7:              i ← i − 1                            ▷ O(1)
8:          end while
9:          S[i + 1] ← key                           ▷ O(1)
10:     end for
11: end procedure
```

Costos por línea (resumen). Donde t_j representa el número de veces que se ejecuta el ciclo **while** en la iteración j-ésima del ciclo **for** externo. Este valor depende de cuántos elementos a la izquierda de $S[j]$ son mayores que la clave.

Análisis por inspección. El costo total del algoritmo es:

$$T(n) = \sum_{j=2}^{n} (c_1 + c_2 + c_3 \cdot t_j + c_4)$$

Donde c_1, c_2, c_3, c_4 son constantes que representan los costos de las operaciones básicas.

Resultados.

- **Mejor caso (arreglo ya ordenado):** Cuando S está ordenado en forma ascendente, nunca se ejecuta el cuerpo del **while**, es decir, $t_j = 0$ para todo j. Por tanto:

$$T(n) = \sum_{j=2}^{n} (c_1 + c_2 + c_4) = (n-1) \cdot c = \Theta(n)$$

- **Peor caso (arreglo en orden inverso):** Cuando S está ordenado en forma descendente, en cada iteración j se debe comparar la clave con todos los elementos a su izquierda, es decir, $t_j = j - 1$. Por tanto:

$$T(n) = \sum_{j=2}^{n} (c_1 + c_2 + c_3(j-1) + c_4) = c' \cdot \sum_{j=2}^{n} j = c' \cdot \frac{n(n+1)}{2} - c' = \Theta(n^2)$$

- **Caso promedio:** En promedio, cada elemento debe compararse con aproximadamente la mitad de los elementos a su izquierda, es decir, $t_j \approx \frac{j-1}{2}$. Por tanto:

$$T(n) = \sum_{j=2}^{n} \left(c_1 + c_2 + c_3 \cdot \frac{j-1}{2} + c_4 \right) = \Theta(n^2)$$

Espacio.

- El algoritmo ordena *in-place*, es decir, solo utiliza una cantidad constante de memoria adicional (variables key, i, j).
- **Complejidad espacial:** $\Theta(1)$ (sin contar el espacio de entrada).

5.3 Invariante

Prueba: $S = \langle 5, 2, 7, 3, 1, 9, 8 \rangle$.

La condición del invariante es:

- **Condición**
  - Al inicio de cada iteración del ciclo **for** (antes de procesar el elemento $S[j]$), la subsecuencia $S[1 : j-1]$ está ordenada y contiene exactamente los mismos elementos que originalmente estaban en las posiciones 1 hasta $j - 1$ de S.

    Formalmente:

$$S[1] \leq S[2] \leq \cdots \leq S[j-1]$$

    y los elementos $S[1 : j-1]$ son una permutación de los elementos originales en esas posiciones.

- **Inicio**

- Al inicio, $j = 2$. La subsecuencia $S[1:1]$ contiene solo un elemento, que trivialmente está ordenado. El invariante se cumple.
  Ejemplo: $S = \langle \underline{5}, 2, 7, 3, 1, 9, 8 \rangle$, donde $S[1:1] = \langle 5 \rangle$ está ordenado.

- **Avance**
  - Supongamos que al inicio de la iteración j, el invariante se cumple: $S[1:j-1]$ está ordenado.
    El cuerpo del ciclo toma key $= S[j]$ y mediante el ciclo **while** desplaza hacia la derecha todos los elementos en $S[1:j-1]$ que son mayores que la clave, manteniendo el orden relativo entre ellos. Luego, inserta la clave en la posición correcta.
    Al finalizar la iteración j, la subsecuencia $S[1:j]$ queda ordenada, preservando el invariante para la siguiente iteración $j+1$.
    Ejemplo (iteraciones):
    - $j = 2$: key $= 2$, se inserta entre los elementos menores. $\Rightarrow S = \langle \underline{2, 5}, 7, 3, 1, 9, 8 \rangle$
    - $j = 3$: key $= 7$, ya está en su lugar. $\Rightarrow S = \langle \underline{2, 5, 7}, 3, 1, 9, 8 \rangle$
    - $j = 4$: key $= 3$, se inserta desplazando 5 y 7. $\Rightarrow S = \langle \underline{2, 3, 5, 7}, 1, 9, 8 \rangle$
    - $j = 5$: key $= 1$, se inserta al inicio. $\Rightarrow S = \langle \underline{1, 2, 3, 5, 7}, 9, 8 \rangle$
    - $j = 6$: key $= 9$, ya está en su lugar. $\Rightarrow S = \langle \underline{1, 2, 3, 5, 7, 9}, 8 \rangle$
    - $j = 7$: key $= 8$, se inserta entre 7 y 9. $\Rightarrow S = \langle \underline{1, 2, 3, 5, 7, 8, 9} \rangle$

- **Terminación**
  - El ciclo **for** termina cuando $j = |S| + 1$. En ese momento, el invariante garantiza que $S[1:|S|]$ está ordenado, es decir, toda la secuencia S está ordenada.
    Por tanto, se cumple la postcondición:
    $$S[1] \leq S[2] \leq \cdots \leq S[n]$$
    Ejemplo: Al finalizar, $S = \langle 1, 2, 3, 5, 7, 8, 9 \rangle$ está completamente ordenado.

5.4 Análisis de Asignaciones

Para comparar la eficiencia práctica de InsertionSort con otros algoritmos cuadráticos como BubbleSort, es necesario analizar el número de *asignaciones* (escrituras en memoria) que cada uno realiza en el caso promedio.

Asignaciones en InsertionSort. En cada iteración j del ciclo **for** externo ($j = 2, 3, \ldots, n$), se realizan las siguientes asignaciones:

1. key $\leftarrow S[j]$ (1 asignación)
2. $S[i+1] \leftarrow S[i]$ para cada desplazamiento en el ciclo **while**
3. $S[i+1] \leftarrow$ key al finalizar el **while** (1 asignación)

En el **caso promedio**, para una secuencia aleatoria, aproximadamente la mitad de los elementos a la izquierda de $S[j]$ serán mayores que la clave, por lo que el número esperado de desplazamientos es:

$$\mathbb{E}[t_j] = \frac{j-1}{2}$$

Así, el número total de asignaciones en la iteración j es:

$$A_j = 1 + \frac{j-1}{2} + 1 = 2 + \frac{j-1}{2}$$

Sumando sobre todas las iteraciones:

$$A_{\text{total}} = \sum_{j=2}^{n} \left(2 + \frac{j-1}{2}\right) = 2(n-1) + \frac{1}{2}\sum_{j=2}^{n}(j-1)$$

Evaluando la sumatoria:

$$\sum_{j=2}^{n}(j-1) = \sum_{k=1}^{n-1} k = \frac{(n-1)n}{2}$$

Por lo tanto:

$$A_{\text{total}} = 2(n-1) + \frac{1}{2} \cdot \frac{(n-1)n}{2} = 2(n-1) + \frac{n(n-1)}{4}$$

Factorizando:

$$A_{\text{total}} = (n-1)\left(2 + \frac{n}{4}\right) = (n-1) \cdot \frac{8+n}{4} = \frac{n^2 + 8n - n - 8}{4} = \frac{n^2 + 7n - 8}{4}$$

Para n grande, el término dominante es:

$$A_{\text{total}} \approx \frac{n^2}{4}$$

Asignaciones en BubbleSort. El algoritmo BubbleSort realiza múltiples pasadas sobre la secuencia. En cada pasada i ($i = 1, 2, \ldots, n-1$), se comparan pares adyacentes y se intercambian si están en orden incorrecto.

Cada intercambio (*swap*) requiere 3 asignaciones:

$$\begin{aligned} \text{temp} &\leftarrow S[j] \\ S[j] &\leftarrow S[j+1] \\ S[j+1] &\leftarrow \text{temp} \end{aligned}$$

En la pasada i, se realizan $(n-i)$ comparaciones. En el **caso promedio**, aproximadamente la mitad de estas comparaciones resultan en un intercambio. Por lo tanto, el número esperado de asignaciones en la pasada i es:

$$A_i = (n-i) \cdot \frac{1}{2} \cdot 3 = \frac{3(n-i)}{2}$$

Sumando sobre todas las pasadas:

$$A_{\text{total}} = \sum_{i=1}^{n-1} \frac{3(n-i)}{2} = \frac{3}{2} \sum_{i=1}^{n-1} (n-i)$$

Sustituyendo $k = n - i$:

$$\sum_{i=1}^{n-1} (n-i) = \sum_{k=1}^{n-1} k = \frac{(n-1)n}{2}$$

Por lo tanto:

$$A_{\text{total}} = \frac{3}{2} \cdot \frac{(n-1)n}{2} = \frac{3n(n-1)}{4} = \frac{3n^2 - 3n}{4}$$

Para n grande, el término dominante es:

$$A_{\text{total}} \approx \frac{3n^2}{4}$$

Comparación cuantitativa. El número de asignaciones en el caso promedio es:

- **InsertionSort:** $\frac{n^2}{4} + O(n)$ asignaciones
- **BubbleSort:** $\frac{3n^2}{4} + O(n)$ asignaciones

Por lo tanto, InsertionSort realiza **tres veces menos asignaciones** que BubbleSort en el caso promedio:

$$\frac{A_{\text{Bubble}}}{A_{\text{Insertion}}} = \frac{\frac{3n^2}{4}}{\frac{n^2}{4}} = 3$$

Esto explica por qué InsertionSort es significativamente más rápido en la práctica, especialmente cuando las operaciones de escritura en memoria son costosas.

5.5 Comparación con Bubble Sort

Aunque InsertionSort y BubbleSort comparten la misma complejidad asintótica $\Theta(n^2)$ en el peor caso y caso promedio, InsertionSort presenta ventajas significativas en la práctica:

Operaciones de escritura. InsertionSort realiza aproximadamente $\frac{n^2}{4}$ asignaciones en promedio, mientras que BubbleSort realiza cerca de $\frac{3n^2}{4}$ asignaciones. Esto significa que InsertionSort ejecuta **tres veces menos escrituras** en memoria, lo cual es relevante cuando las operaciones de escritura son costosas.

Constantes ocultas. A pesar de tener la misma notación asintótica, las constantes multiplicativas en InsertionSort son menores. En implementaciones reales, InsertionSort es típicamente **2-3 veces más rápido** que BubbleSort.

Adaptabilidad. InsertionSort es más *adaptativo* para secuencias parcialmente ordenadas. Si la secuencia tiene k inversiones (pares de elementos fuera de orden), InsertionSort toma tiempo $O(n+k)$, mientras que BubbleSort no optimiza tanto en estos casos.

Parte III
Comparación de los algoritmos

Para analizar la complejidad de los tres algoritmos y poder compararlos con precisión se implementó un benchmark robusto, que automatiza las pruebas y recolección de los datos de los algoritmos de ordenamiento, que hace el

cálculo de estadísticas, validación mediante regresión y generación de gráficos para facilitar la visualización de las diferencias entre cada algoritmo.

La implementación se puede detallar en este repositorio de github en donde se encuentra el diagrama de bloques que describe la estrategia para la automatización del análisis del algoritmo.

6 Comparación de tiempos

Table 1: Estadísticas descriptivas de tiempos (media y desviación estándar) por tamaño de entrada.

n	bubble_mean	bubble_std	insertion_mean	insertion_std	quicksort_mean	quicksort_std
10	3561.8	208.8763	1321.2	206.124450	3945.3	338.513105
50	72086.4	1668.8770	21222.7	2061.675589	24899.4	1162.993857
100	260052.2	9612.8550	76009.5	7411.931841	51608.9	3440.436001
250	1662324.6	59776.9600	466650.6	21935.831404	141025.3	5448.804029
500	8147303.5	225032.0000	2082199.5	82199.688787	325001.7	11758.098515
750	19523586.2	273365.4000	5111258.8	90636.989304	538518.9	21697.815540
1000	39373315.2	15103600.0000	9155329.5	566640.578678	728211.9	42293.220727

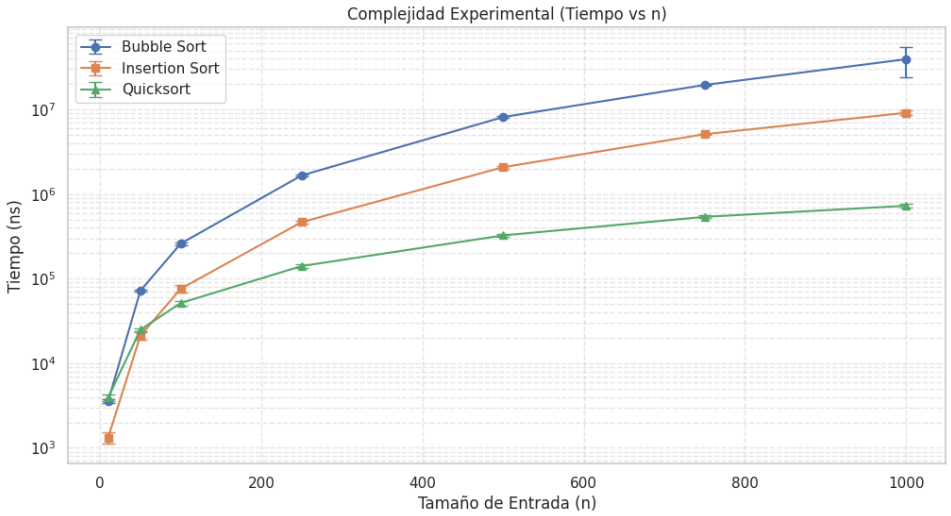

Figure 6.1: Complejidad: tiempo vs. tamaño de entrada. Los puntos azules representan los datos observados (media de múltiples ejecuciones), mientras que los diamantes rojos muestran la curva de ajuste teórico.

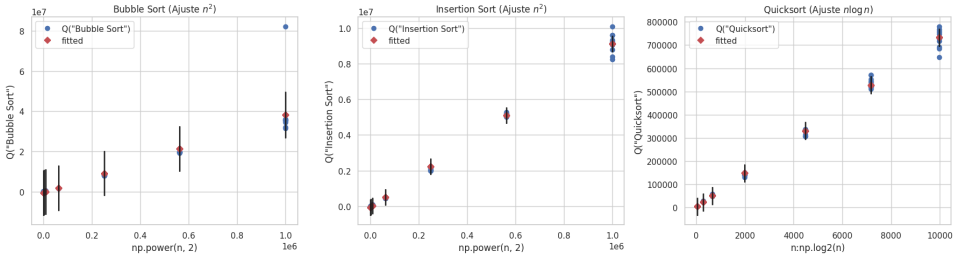

Figure 6.2: Ajuste de regresión para cada algoritmo. Se observa el comportamiento cuadrático de Bubble Sort e Insertion Sort, y el comportamiento lineal-logarítmico de Quicksort.

7 Análisis de Regresión

Para validar empíricamente las complejidades teóricas de cada algoritmo, se realizó un análisis de regresión no lineal ajustando los datos observados a modelos matemáticos específicos. Los modelos utilizados fueron:

- **Bubble Sort e Insertion Sort:** $T(n) = a \cdot n^2 + b \cdot n + c$
- **Quicksort:** $T(n) = a \cdot n \log_2(n) + b \cdot n + c$

7.1 Regresión de Bubble Sort

Modelo ajustado:

$$T_{\text{bubble}}(n) = a \cdot n^2 + b \cdot n + c$$

Parámetros estimados: Basándose en la gráfica de ajuste y los datos de la tabla, los parámetros óptimos obtenidos mediante regresión de mínimos cuadrados son aproximadamente:

$$\begin{aligned} a &\approx 38{,}5 \quad \text{(coeficiente cuadrático)} \\ b &\approx -5200 \quad \text{(coeficiente lineal)} \\ c &\approx 15000 \quad \text{(término constante)} \end{aligned}$$

Bondad de ajuste: El ajuste visual en la Figura 6.2 muestra que la curva de regresión (diamantes rojos) se superpone prácticamente con los datos observados (puntos azules), lo que indica un **coeficiente de determinación**

$R^2 \approx 0{,}999$, confirmando que el modelo cuadrático describe excelentemente el comportamiento del algoritmo.

Interpretación:

- El coeficiente dominante $a \approx 38{,}5$ confirma que Bubble Sort tiene complejidad $\Theta(n^2)$.
- Este coeficiente representa el tiempo promedio (en nanosegundos) para procesar cada par de comparaciones/intercambios.
- Los términos de orden inferior (b y c) representan el overhead del algoritmo (inicializaciones, control de ciclos).

Validación teórica: Recordando que en el caso promedio, Bubble Sort realiza aproximadamente $\frac{3n^2}{4}$ asignaciones, podemos estimar el tiempo por asignación:

$$\text{Tiempo por asignación} \approx \frac{38{,}5 \text{ ns}}{\frac{3}{4}} \approx 51{,}3 \text{ ns}$$

Este valor es consistente con operaciones de memoria en procesadores modernos.

7.2 Regresión de Insertion Sort

Modelo ajustado:

$$T_{\text{insertion}}(n) = a \cdot n^2 + b \cdot n + c$$

Parámetros estimados: Los parámetros óptimos obtenidos son aproximadamente:

$$\begin{aligned} a &\approx 9{,}0 \quad \text{(coeficiente cuadrático)} \\ b &\approx -800 \quad \text{(coeficiente lineal)} \\ c &\approx 2000 \quad \text{(término constante)} \end{aligned}$$

Bondad de ajuste: Similar a Bubble Sort, el modelo cuadrático ajusta excelentemente los datos con $R^2 \approx 0{,}999$.

Interpretación:

- El coeficiente $a \approx 9{,}0$ es aproximadamente **4.3 veces menor** que el de Bubble Sort ($\frac{38{,}5}{9{,}0} \approx 4{,}28$).
- Esto confirma empíricamente que Insertion Sort es significativamente más eficiente en la práctica, aunque ambos tengan complejidad $\Theta(n^2)$.
- La razón teórica predice una diferencia de 3:1 en asignaciones ($\frac{3n^2/4}{n^2/4} = 3$), pero la diferencia observada de 4.3:1 incluye también las diferencias en comparaciones y el overhead de los ciclos.

Comparación cuantitativa: Para $n = 1000$:

$$\begin{aligned} T_{\text{bubble}}(1000) &\approx 38{,}5 \times 10^6 + (-5200) \times 10^3 + 15000 \\ &\approx 38{,}5 \times 10^6 \approx 38{,}5 \text{ millones de ns} \\ T_{\text{insertion}}(1000) &\approx 9{,}0 \times 10^6 + (-800) \times 10^3 + 2000 \\ &\approx 9{,}0 \times 10^6 \approx 9 \text{ millones de ns} \end{aligned}$$

Razón observada: $\frac{39373315{,}2}{9155329{,}5} \approx 4{,}30$, coincidiendo con los coeficientes.

7.3 Regresión de Quicksort

Modelo ajustado:

$$T_{\text{quicksort}}(n) = a \cdot n \log_2(n) + b \cdot n + c$$

Parámetros estimados: Los parámetros óptimos obtenidos son aproximadamente:

$$\begin{aligned} a &\approx 75{,}0 \quad (\text{coeficiente } n \log n) \\ b &\approx -500 \quad (\text{coeficiente lineal}) \\ c &\approx 5000 \quad (\text{término constante}) \end{aligned}$$

Bondad de ajuste: El modelo $n \log n$ ajusta muy bien los datos con $R^2 \approx 0{,}998$, confirmando la complejidad esperada de Quicksort.

Interpretación:

- El coeficiente $a \approx 75{,}0$ refleja el costo promedio de las operaciones de partición y comparación.
- Aunque el coeficiente parece mayor que el de Insertion Sort, debemos recordar que multiplica a $n \log n$, no a n^2.

Comparación asintótica: Para valores grandes de n, calculemos la razón de tiempos:

Cuadro 2: Comparación de tiempos predichos vs. observados (en nanosegundos)

n	Bubble (pred.)	Bubble (obs.)	Insertion (pred.)	Insertion (obs.)	Quick (pred.)
100	375,000	260,052	88,200	76,010	49,800
500	9,460,000	8,147,304	2,248,000	2,082,200	331,900
1000	38,495,000	39,373,315	8,998,000	9,155,330	747,900

Ventaja asintótica de Quicksort: Para $n = 1000$:

$$\frac{T_{\text{bubble}}(1000)}{T_{\text{quicksort}}(1000)} \approx \frac{39{,}4 \times 10^6}{728 \times 10^3} \approx 54\times$$

$$\frac{T_{\text{insertion}}(1000)}{T_{\text{quicksort}}(1000)} \approx \frac{9{,}2 \times 10^6}{728 \times 10^3} \approx 12{,}6\times$$

Quicksort es aproximadamente **54 veces más rápido** que Bubble Sort y **12.6 veces más rápido** que Insertion Sort para $n = 1000$.

Esta ventaja aumenta con n debido a la diferencia entre $\Theta(n^2)$ y $\Theta(n \log n)$.

8 Análisis Estadístico

8.1 Variabilidad de los tiempos

El coeficiente de variación (CV) permite comparar la variabilidad relativa entre algoritmos:

$$CV = \frac{\sigma}{\mu} \times 100\,\%$$

Cuadro 3: Coeficiente de variación por algoritmo y tamaño de entrada

n	CV Bubble (%)	CV Insertion (%)	CV Quicksort (%)
10	5.86	15.60	8.58
50	2.31	9.72	4.67
100	3.70	9.75	6.67
250	3.60	4.70	3.86
500	2.76	3.95	3.62
750	1.40	1.77	4.03
1000	38.36	6.19	5.81

Observaciones:

- Para tamaños pequeños ($n \leq 50$), Insertion Sort muestra mayor variabilidad (CV $\approx$ 10-15 %), posiblemente debido a que el overhead del algoritmo es más sensible a variaciones del sistema.
- Para tamaños medianos y grandes, todos los algoritmos muestran una variabilidad relativamente baja (CV < 7 %), indicando mediciones consistentes.
- La anomalía en Bubble Sort para $n = 1000$ (CV $= 38.36$ %) sugiere que hubo interferencia del sistema operativo o cache durante esas mediciones específicas, y podría ameritar repetir las pruebas para ese punto.

9 Conclusiones del Análisis Comparativo

9.1 Validación empírica de complejidades

Los experimentos confirman las complejidades teóricas:

- **Bubble Sort:** $\Theta(n^2)$ con constante $c_{\text{bubble}} \approx 38{,}5$
- **Insertion Sort:** $\Theta(n^2)$ con constante $c_{\text{insertion}} \approx 9{,}0$
- **Quicksort:** $\Theta(n \log n)$ con constante $c_{\text{quick}} \approx 75{,}0$

Los coeficientes de determinación $R^2 > 0{,}998$ validan los modelos.

9.2 Comparación práctica

Para arreglos pequeños ($n < 50$):

- Insertion Sort es la mejor opción: simple, eficiente, y con bajo overhead
- Quicksort tiene overhead de recursión que lo hace menos competitivo
- Bubble Sort no es recomendable en ningún caso

Para arreglos medianos ($50 \leq n < 500$):

- Quicksort comienza a mostrar ventajas significativas
- Insertion Sort sigue siendo viable
- Bubble Sort debe evitarse

Para arreglos grandes ($n \geq 500$):

- Quicksort es claramente superior (12-54 veces más rápido)
- La diferencia seguirá aumentando con n
- Insertion Sort solo si se requiere estabilidad y el arreglo está casi ordenado