

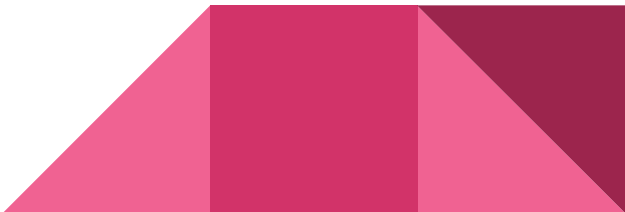
Algoritmos de ordenamiento

¿Qué son los algoritmos de ordenamiento?

Definición: Son técnicas que organizan elementos de una colección en un orden específico

Objetivo: Transformar un conjunto desordenado en uno ordenado

Tipos de ordenamiento:

- Numérico (ascendente o descendente)
 - Alfabético
 - Por múltiples criterios
- 

¿Para qué sirven los algoritmos de ordenamiento?


Búsqueda eficiente: La búsqueda binaria en datos ordenados es $O(\log n)$

Procesamiento de datos: Facilitan análisis y visualización

Organización de información: Interfaces de usuario, reportes, tablas

Base para otros algoritmos: Muchos algoritmos requieren datos ordenados

Aplicaciones prácticas:

- Clasificación de resultados de búsqueda
 - Ordenamiento de contactos/archivos
 - Análisis estadístico
 - Bases de datos
- 

Complejidad Algorítmica: $O(n^2)$

¿Qué significa $O(n^2)$?

- Tiempo de ejecución crece de forma cuadrática con el tamaño de la entrada
- Si duplicamos el tamaño, el tiempo se cuadruplica

Visualización:

- Para $n=10$: aproximadamente 100 operaciones
- Para $n=100$: aproximadamente 10,000 operaciones
- Para $n=1000$: aproximadamente 1,000,000 operaciones

Algoritmos $O(n^2)$ comunes:

- Bubble Sort
 - Selection Sort
 - Insertion Sort
- 

¿Por qué es importante aprender algoritmos $O(n^2)$?

Fundamentos conceptuales:

- Facilitan la comprensión de estructuras de datos
- Introducen conceptos de eficiencia algorítmica


Simplicidad de implementación:

- Fáciles de codificar y depurar
- Requieren pocos recursos adicionales (memoria)

Base para entender algoritmos avanzados:

- Punto de comparación para algoritmos más eficientes
- Muchos algoritmos rápidos usan conceptos similares

Útiles en situaciones específicas:

- Arrays pequeños ($n < 20$)
 - Datos parcialmente ordenados
 - Sistemas con recursos limitados
- 

Bubble Sort: Concepto Básico

Bubble Sort: Concepto Básico

- **Idea principal:**
 - Comparar pares adyacentes de elementos
 - Intercambiar si están en orden incorrecto
 - Repetir hasta que no haya intercambios
- **Analogía:**
 - Como burbujas en agua, los elementos "livianos" flotan a la superficie
 - Los valores más grandes "burbujean" hacia el final del array



Algoritmo Bubble Sort en Pseudocódigo

BUBBLE-SORT(A)

n = longitud(A)

for i = 0 to n-1

swapped = false

for j = 0 to n-i-1

if A[j] > A[j+1]

intercambiar A[j] con A[j+1]

swapped = true

if swapped == false

break



Complejidad de Bubble Sort

Tiempo (Peor caso): $O(n^2)$

- Ocurre cuando el array está en orden inverso
- Requiere el máximo número de intercambios

Tiempo (Mejor caso): $O(n)$

- Ocurre cuando el array ya está ordenado
- Con optimización de "swapped", termina en una pasada

Tiempo (Caso promedio): $O(n^2)$

Espacio: $O(1)$

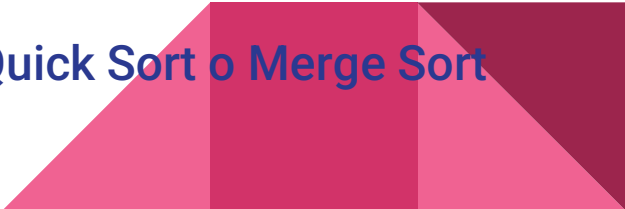
- Usa memoria constante independiente del tamaño del array

Ventajas y Desventajas de Bubble Sort

Ventajas:

- Fácil de entender e implementar
- Detecta si el array ya está ordenado (con bandera "swapped")
- Estable: mantiene el orden relativo de elementos iguales
- In-place: no requiere memoria adicional

Desventajas:

- Ineficiente para arrays grandes ($O(n^2)$)
 - Realiza muchos intercambios innecesarios
 - Mucho más lento que algoritmos avanzados como Quick Sort o Merge Sort
- 

Selection Sort: Concepto Básico

Selection Sort es un algoritmo de ordenamiento que divide el arreglo en dos partes:

- Una parte ordenada (al inicio vacía) que se construye de izquierda a derecha
- Una parte no ordenada (inicialmente todo el arreglo)

En cada iteración, el algoritmo **selecciona** el elemento más pequeño de la parte no ordenada y lo coloca al final de la parte ordenada. Este proceso se repite hasta que todo el arreglo queda ordenado.



Algoritmo Selection Sort en Pseudocódigo

procedimiento selectionSort(A: arreglo de elementos)

 n = longitud(A)

 para i desde 0 hasta n-2 hacer

 // Encontrar el índice del elemento mínimo en la parte no ordenada

 índice_mínimo = i

 para j desde i+1 hasta n-1 hacer

 si $A[j] < A[\text{índice_mínimo}]$ entonces

 índice_mínimo = j

 fin si

 fin para

 // Intercambiar el elemento mínimo con el primer elemento de la parte no ordenada

 si índice_mínimo \neq i entonces

 intercambiar $A[i]$ con $A[\text{índice_mínimo}]$

 fin si

 fin para


fin procedimiento



Complejidad de Selection Sort

- **Complejidad temporal:**
 - Mejor caso: $O(n^2)$
 - Caso promedio: $O(n^2)$
 - Peor caso: $O(n^2)$
- **Complejidad espacial:** $O(1)$ - ordenamiento in-situ

Selection Sort realiza siempre el mismo número de comparaciones independientemente de la distribución inicial de los datos, por lo que su complejidad es constante en todos los casos.



Ventajas y Desventajas de Selection Sort

Ventajas:

- **Simple de implementar:** Es un algoritmo muy intuitivo y fácil de codificar.
- **Número mínimo de intercambios:** Realiza a lo sumo $n-1$ intercambios, lo que puede ser beneficioso cuando el costo de intercambiar elementos es alto.
- **Rendimiento predecible:** Siempre tiene la misma complejidad $O(n^2)$, independientemente de la distribución de los datos.
- **Estabilidad en memoria:** Funciona bien en sistemas con memoria limitada porque es un algoritmo in-situ.

Desventajas:

- **Ineficiente en arreglos grandes:** Su complejidad cuadrática lo hace poco práctico para grandes conjuntos de datos.
- **No adaptativo:** No aprovecha el orden parcial que pueda existir en el arreglo.

Ventajas y Desventajas de Selection Sort

Ventajas:

- **Simple de implementar:** Es un algoritmo muy intuitivo y fácil de codificar.
- **Número mínimo de intercambios:** Realiza a lo sumo $n-1$ intercambios, lo que puede ser beneficioso cuando el costo de intercambiar elementos es alto.
- **Rendimiento predecible:** Siempre tiene la misma complejidad $O(n^2)$, independientemente de la distribución de los datos.
- **Estabilidad en memoria:** Funciona bien en sistemas con memoria limitada porque es un algoritmo in-situ.

Desventajas:

- **Ineficiente en arreglos grandes:** Su complejidad cuadrática lo hace poco práctico para grandes conjuntos de datos.
- **No adaptativo:** No aprovecha el orden parcial que pueda existir en el arreglo.

Insertion Sort: Concepto Básico

Insertion Sort es un algoritmo de ordenamiento que construye el arreglo ordenado final de uno en uno. Funciona de manera similar a como ordenamos cartas en nuestra mano:

- Se mantiene una parte del arreglo ya ordenada (inicialmente solo el primer elemento)
- Se toma un elemento de la parte no ordenada y se inserta en la posición correcta dentro de la parte ordenada
- Este proceso se repite hasta ordenar todo el arreglo

El algoritmo recorre el arreglo de izquierda a derecha, insertando cada elemento en su posición correcta entre los elementos ya ordenados a su izquierda.



Algoritmo Insertion Sort en Pseudocódigo

procedimiento insertionSort(A: arreglo de elementos)

 n = longitud(A)

 para i desde 1 hasta n-1 hacer

 // Guardar el elemento actual

 elemento_actual = A[i]

 // Mover elementos mayores que elemento_actual a una posición adelante

 j = i - 1

 mientras j >= 0 y A[j] > elemento_actual hacer

 A[j+1] = A[j]

 j = j - 1

 fin mientras

 // Insertar el elemento actual en su posición correcta

 A[j+1] = elemento_actual

 fin para

fin procedimiento



Complejidad de Insertion Sort

- **Complejidad temporal:**
 - Mejor caso: $O(n)$ - cuando el arreglo ya está ordenado
 - Caso promedio: $O(n^2)$
 - Peor caso: $O(n^2)$ - cuando el arreglo está ordenado en orden inverso
- **Complejidad espacial:** $O(1)$ - ordenamiento in-situ

La eficiencia de Insertion Sort depende significativamente de qué tan ordenado esté previamente el arreglo, lo que lo hace adaptativo.

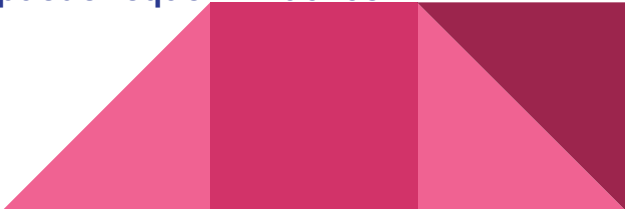


Ventajas y Desventajas de Insertion Sort

Ventajas:

- **Simple de implementar:** Fácil de entender y codificar.
- **Eficiente para arreglos pequeños:** Para n pequeño (típicamente $n < 20$), puede superar a algoritmos más complejos.
- **Adaptativo:** Eficiente para datos parcialmente ordenados, llegando a $O(n)$ en el mejor caso.
- **In-situ:** No requiere memoria adicional.

Desventajas:


- **Ineficiente para arreglos grandes:** Su complejidad cuadrática lo hace poco práctico para grandes conjuntos de datos.
 - **Requiere muchos desplazamientos:** A diferencia de Selection Sort, puede requerir muchos movimientos de elementos.
- 

Consideraciones

Aplicaciones prácticas:

- Es utilizado en la implementación de algoritmos de ordenamiento híbridos como Timsort (usado en Python y Java).
- Muy eficiente cuando se ordenan elementos a medida que llegan (ordenamiento en línea).
- Ideal para completar el ordenamiento en arreglos casi ordenados.

Comparación con Bubble y Selection:

- Es generalmente más eficiente que Bubble Sort.
 - A diferencia de Selection Sort, aprovecha el orden parcial existente.
 - Es estable, a diferencia de Selection Sort.
- 

Ejercicio Práctico

Implementar los 3 algoritmos de ordenamiento

Modificalo para contar el número de comparaciones e intercambios

Probá con diferentes arrays:

- Ordenado: [1, 2, 3, 4, 5]
- Inversamente ordenado: [5, 4, 3, 2, 1]
- Parcialmente ordenado: [1, 3, 2, 5, 4]

Compará el número de operaciones en cada caso

