PUCE Virtual

CLASE

2

ESTRUCTURA DE DATOS
(PROGRAMACIÓN III)

Ordenamiento, búsquedas y ordenamiento

QS WORLD UNIVERSITY RANKINGS

Educación de calidad

INTRODUCCIÓN DE LA CLASE

GRADO / POSGRADO / TECNOLOGÍAS

Estudia a tu tiempo
son interrupciones

EDUCACIÓN EN LÍNEA DE CALIDAD

El estudio de los algoritmos de ordenamiento y búsqueda constituye un componente esencial en la formación en ciencias de la computación, ya que permite comprender cómo se organizan y localizan datos de manera eficiente. Los métodos básicos de ordenamiento, como Bubble Sort, Selection Sort e Insertion Sort, aunque simples en su estructura, proporcionan una base sólida para analizar conceptos como complejidad temporal, complejidad espacial, estabilidad y costo computacional. Estos fundamentos permiten evaluar cómo crece el número de operaciones a medida que aumenta el volumen de información y facilitan la comparación entre distintas estrategias algorítmicas.

En el ámbito de la ciberseguridad, estos conceptos adquieren especial relevancia debido a la necesidad de procesar grandes volúmenes de registros, eventos de red y transacciones en tiempo real. La correcta selección de algoritmos impacta directamente en la velocidad de detección de amenazas, la correlación de incidentes y la capacidad de respuesta ante ataques. Por ello, comprender los criterios técnicos de selección algorítmica y su aplicación práctica en sistemas de monitoreo resulta fundamental para diseñar soluciones escalables, eficientes y seguras.

Clase 2:
Comprender los conceptos de algoritmos y estructuras de datos, analizando su funcionamiento y eficiencia computacional en la resolución de problemas de ciberseguridad.

2. Ordenamiento, búsquedas y ordenamiento

El ordenamiento y la búsqueda son operaciones esenciales en ciencias de la computación. Ordenar consiste en organizar datos según un criterio determinado, mientras que buscar implica localizar un elemento dentro de una colección. Ambos procesos están estrechamente relacionados, ya que ciertos algoritmos de búsqueda eficientes, como la búsqueda binaria, requieren que los datos estén previamente ordenados para funcionar correctamente (Cormen et al., 2009). Esta relación demuestra que la organización de la información no solo mejora la claridad, sino también el rendimiento computacional.

En ciberseguridad, donde se procesan grandes volúmenes de datos en tiempo real, el ordenamiento adquiere una relevancia estratégica. Plataformas SIEM como Splunk Enterprise o IBM QRadar generan millones de registros provenientes de distintos dispositivos, aplicaciones y servicios de red. Sin una adecuada organización, estos datos pueden volverse difíciles de analizar y las amenazas podrían pasar desapercibidas.

Ordenar los registros por fecha, dirección IP, usuario o tipo de evento permite identificar patrones sospechosos, como múltiples intentos fallidos de autenticación en un periodo corto. Además, facilita la correlación de eventos, ya que muchos ataques no se detectan por un único registro, sino por la relación entre varios sucesos aparentemente aislados.

En el análisis forense digital, el orden cronológico es fundamental para reconstruir la línea de tiempo de un incidente, determinar su origen y evaluar su impacto. Una correcta organización de la información mejora la detección de amenazas, reduce el tiempo de respuesta ante incidentes y fortalece la toma de decisiones, convirtiéndose en un elemento clave para la seguridad informática moderna.

2.1. Algoritmos de ordenamiento básicos

Los algoritmos de ordenamiento básicos son métodos sencillos diseñados para organizar datos según un criterio específico, como orden ascendente, descendente o por prioridad. Se consideran básicos debido a su lógica simple y fácil implementación, lo que los convierte en herramientas ideales para introducir conceptos fundamentales del análisis y diseño de algoritmos. Su estructura clara permite comprender paso a paso cómo se comparan e intercambian los elementos dentro de una lista.

Entre los más conocidos se encuentran Bubble Sort, Selection Sort e Insertion Sort. Aunque no son los más eficientes cuando se trabaja con grandes volúmenes de datos, resultan esenciales para entender conceptos como la complejidad temporal y espacial, la cantidad de comparaciones e intercambios, y la estabilidad del algoritmo. Su análisis permite estudiar los casos mejor, peor y promedio, y observar cómo aumenta el costo computacional a medida que crece el tamaño de la entrada.

En conjuntos de datos pequeños o casi ordenados, estos algoritmos pueden ser prácticos y suficientes. Además, constituyen una base conceptual sólida para comprender técnicas de ordenamiento más avanzadas y eficientes, facilitando la transición hacia métodos con mejor rendimiento en aplicaciones reales.

2.1.1. Bubble Sort

Bubble Sort (Ordenamiento de Burbuja) compara elementos adyacentes e intercambia sus posiciones si están en el orden incorrecto. Este proceso se repite hasta que no se requieran más intercambios (Sedgewick & Wayne, 2011).

Utilizado principalmente con fines educativos y en conjuntos de datos pequeños debido a su simplicidad y facilidad de implementación. Permite comprender claramente el funcionamiento de comparaciones e intercambios entre elementos adyacentes, así como conceptos básicos como iteraciones y complejidad temporal. Además, es un algoritmo estable y trabaja “in place”, sin requerir memoria adicional significativa.

No obstante, su complejidad $O(n^2)$ en el peor y promedio de los casos lo hace ineficiente para grandes volúmenes de datos, por lo que no se emplea en sistemas de gran escala.

En ciberseguridad, este tipo de enfoque puede aplicarse para ordenar un pequeño grupo de alertas por nivel de riesgo o clasificar un conjunto limitado de direcciones IP según la cantidad de intentos fallidos de acceso, especialmente cuando el volumen de datos no es elevado y el análisis debe realizarse de forma rápida y comprensible. Por ejemplo, en entornos educativos, laboratorios de práctica o sistemas de monitoreo simples, resulta útil priorizar eventos sospechosos sin recurrir a algoritmos complejos.

En este tipo de escenarios limitados, donde la claridad del proceso y la facilidad de implementación son más relevantes que la eficiencia a gran escala, Bubble Sort puede cumplir adecuadamente su propósito. Su funcionamiento paso a paso facilita la comprensión del ordenamiento aplicado y permite verificar visualmente cómo se reorganizan los datos, lo que resulta valioso para tareas básicas de análisis y aprendizaje.

Complejidad:

- **Tiempo**: $O(n^2)$: Significa que el tiempo de ejecución crece de forma cuadrática en función del número de elementos (n). A medida que aumenta el tamaño del conjunto de datos, el número de comparaciones e intercambios se incrementa aproximadamente en proporción a $n \times n$. Por ello, cuando la cantidad de datos es grande, el algoritmo se vuelve poco eficiente.

- **Espacio**: O(1): Indica que el algoritmo utiliza una cantidad constante de memoria adicional, independientemente del tamaño de la entrada. Es decir, no necesita estructuras auxiliares significativas para funcionar, ya que realiza el ordenamiento directamente sobre la misma lista de datos.

- **Es estable**: Un algoritmo de ordenamiento estable conserva el orden relativo de los elementos que tienen el mismo valor. Esto significa que, si dos elementos son iguales según el criterio de comparación, mantendrán la misma posición relativa que tenían antes de ser ordenados, lo cual es importante cuando los datos contienen múltiples atributos.

En la figura 1 se muestra un ejemplo de la ejecución del método Bubble sort.

```
Arreglo inicial: 5 8 3 7 6
8 < 3 Se intercambian  -- 3 - 8 Intercambiados
5 3 8 7 6
8 < 7 Se intercambian  -- 7 - 8 Intercambiados
5 3 7 8 6
8 < 6 Se intercambian  -- 6 - 8 Intercambiados
5 3 7 6 8
5 < 3 Se intercambian  -- 3 - 5 Intercambiados
3 5 7 6 8
7 < 6 Se intercambian  -- 6 - 7 Intercambiados
3 5 6 7 8
Arreglo ordenado: 3 5 6 7 8
```

Figura 1. Ejemplo de ejecución de Bubble Sort. Creación de autor Patricio Coba

En cada pasada, el valor más grande “burbujea” hacia el final de la lista. En la Tabla 1 se muestra la explicación de la aplicación del método Bubble Sort con los datos iniciales 5,8,3,7,6.

Tabla 1. Explicación de la aplicación del método Bubble Sort. Creación de autor Patricio Coba

Pasada	Comparaciones realizadas	Intercambios	Resultado al finalizar la pasada
1	(5–8 ✓), (8–3 ✗), (8–7 ✗), (8–6 ✗)	8↔3, 8↔7, 8↔6	5, 3, 7, 6, 8
2	(5–3 ✗), (5–7 ✓), (7–6 ✗)	5↔3, 7↔6	3, 5, 6, 7, 8
3	(3–5 ✓), (5–6 ✓)	Ninguno	3, 5, 6, 7, 8

✓ = orden correcto

✗ = requiere intercambio

2.1.2. Selection Sort

Selection Sort (Ordenamiento por selección) es un algoritmo que recorre repetidamente la lista para identificar el elemento mínimo y colocarlo en su posición correcta en cada iteración (Cormen et al., 2009). En cada paso, divide la lista en una parte ordenada y otra desordenada, reduciendo progresivamente esta última.

Es especialmente útil cuando se desea minimizar el número de intercambios, ya que realiza como máximo un intercambio por cada posición.

En ciberseguridad, puede aplicarse para clasificar incidentes según su nivel de criticidad, priorizando primero los eventos más graves para su análisis y respuesta inmediata.

Complejidad

- **Tiempo**: O(n^2) El tiempo de ejecución crece de manera cuadrática respecto al número de elementos (n). Esto significa que, para cada posición del arreglo, se realiza un recorrido adicional para encontrar el elemento mínimo, generando aproximadamente n^2 comparaciones en el peor y promedio de los casos. Por ello, no es eficiente para grandes volúmenes de datos.

- **Espacio**: O(1) Utiliza una cantidad constante de memoria adicional. El ordenamiento se realiza sobre la misma estructura de datos, sin requerir arreglos auxiliares significativos.

- **No es estable por defecto**. En su forma clásica, puede alterar el orden relativo de elementos iguales debido a los intercambios realizados al colocar el mínimo en su posición correspondiente.

En la figura 2 se muestra un ejemplo de la ejecución del método Selection sort.

```
Arreglo original:
5 8 3 7 6
Datos despues de colocar el numero menor en posicion...
3 8 5 7 6
Datos despues de colocar el numero menor en posicion...
3 5 8 7 6
Datos despues de colocar el numero menor en posicion...
3 5 6 7 8
Datos despues de colocar el numero menor en posicion...
3 5 6 7 8
Arreglo ordenado:
3 5 6 7 8
```

Figura 2. Ejemplo práctico Selection Sort. Creación de autor Patricio Coba

Se divide la lista en dos partes: una parte ordenada (al inicio) y otra desordenada (el resto). En cada iteración, busca el elemento más pequeño de la parte desordenada y lo intercambia con el primer elemento de esa sección.

En la Tabla 2 se muestra la explicación de la aplicación del método Selection Sort.

Tabla 2. Explicación de la aplicación del método Selection Sort. Creación de autor Patricio Coba

Iteración	Sublista analizada	Elemento mínimo	Acción realizada	Resultado
1	5, 8, 3, 7, 6	3	Se intercambia 3 con 5	3, 8, 5, 7, 6
2	8, 5, 7, 6	5	Se intercambia 5 con 8	3, 5, 8, 7, 6
3	8, 7, 6	6	Se intercambia 6 con 8	3, 5, 6, 7, 8
4	7, 8	7	No se requiere intercambio	3, 5, 6, 7, 8

2.1.3. Insertion Sort

Insertion Sort (Ordenamiento por inserción) construye la lista ordenada de manera progresiva, tomando cada elemento e insertándolo en la posición correcta dentro de la parte previamente ordenada (Sedgewick & Wayne, 2011). Su funcionamiento es similar a la forma en que se ordenan cartas en la mano, comparando e insertando cada nuevo valor donde corresponde.

Es especialmente eficiente en listas pequeñas o casi ordenadas, ya que requiere pocos desplazamientos cuando los elementos ya están próximos a su posición final.

En ciberseguridad, puede aplicarse en la actualización incremental de registros de acceso ordenados por hora, donde los nuevos eventos se agregan continuamente a una lista ya estructurada.

Complejidad:

- **Mejor caso:** O(n), cuando la lista ya está ordenada.
- **Peor caso:** $O(n^2)$, cuando está en orden inverso.
- Espacio: O(1), utiliza memoria adicional constante.
- **Es estable**, conserva el orden relativo de elementos iguales

En la figura 3 se muestra el funcionamiento de Insertion Sort.

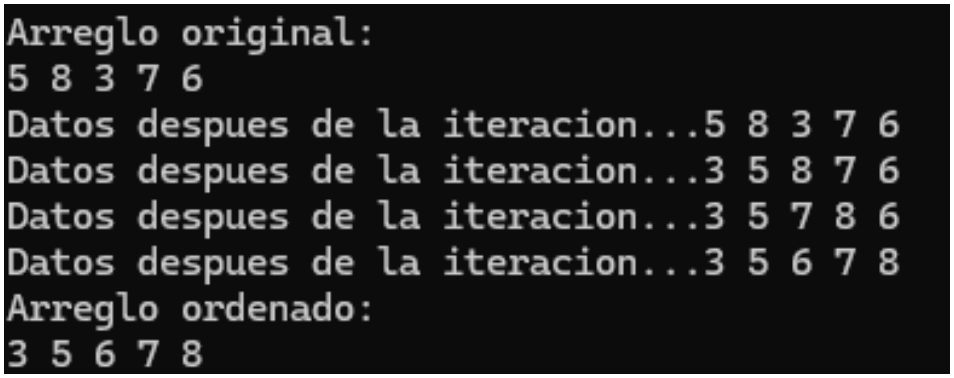


Figura 3. Ejemplo de la aplicación del método Insertion Sort. Creación de autor Patricio Coba

Insertion Sort construye la lista ordenada de izquierda a derecha. En cada iteración toma un elemento y lo inserta en la posición correcta dentro de la parte ya ordenada. En la tabla 3 se presenta la explicación del proceso de Insertion Sort.

Tabla 3. Explicación de Insertion Sort. Creación de autor Patricio Coba

Iteración	Elemento (key)	Parte ordenada antes	Proceso realizado	Resultado
1	8	[5]	8 se compara con 5. Como 8 > 5, no se mueve.	5, 8, 3, 7, 6
2	3	[5, 8]	3 se compara con 8 (se desplaza). Luego con 5 (se desplaza). Se inserta en la primera posición.	3, 5, 8, 7, 6
3	7	[3, 5, 8]	7 se compara con 8 (se desplaza). Luego con 5 (se detiene). Se inserta después de 5.	3, 5, 7, 8, 6
4	6	[3, 5, 7, 8]	6 se compara con 8 (se desplaza) y con 7 (se desplaza). Se detiene en 5. Se inserta después de 5.	

Métodos de ordenamiento. Página web donde se describe diferentes métodos de ordenamiento con código y su respectiva explicación. Enlace: https://www.swhosting.com/es/comunidad/manual/algoritmos-de-ordenacion-con-ejemplos-en-c

2.1.4. Estabilidad y costo computacional

La estabilidad de un algoritmo de ordenamiento se refiere a su capacidad para mantener el orden relativo de los elementos que tienen el mismo valor clave. Es decir, si dos registros

poseen el mismo criterio de comparación, un algoritmo estable conservará su posición original entre ellos después de ordenar. Esta característica es especialmente importante cuando se realizan ordenamientos múltiples (por ejemplo, primero por fecha y luego por prioridad) o cuando se trabaja con bases de datos y registros donde existen varios atributos asociados.

El costo computacional hace referencia a los recursos que consume un algoritmo, principalmente tiempo de ejecución (complejidad temporal) y memoria utilizada (complejidad espacial). Se expresa comúnmente mediante la notación Big-O, que describe cómo crece el número de operaciones a medida que aumenta el tamaño de los datos de entrada. Por ejemplo, algoritmos como Bubble Sort, Selection Sort e Insertion Sort tienen en general una complejidad $O(n^2)$ en el peor caso, lo que los hace poco eficientes para grandes volúmenes de información.

En conjunto, la estabilidad y el costo computacional son criterios clave para seleccionar un método de ordenamiento adecuado, especialmente en sistemas donde el rendimiento, la escalabilidad y la integridad del orden de los datos son factores críticos. (Cormen et al., 2009). La tabla 4 muestra la comparativo de los métodos revisados.

Tabla 4. Comparativa de métodos de ordenamiento. Creación de autor Patricio Coba

Algoritmo	Mejor Caso	Peor Caso	Espacio	Estable	Aplicación típica
Bubble Sort	O(n)	$O(n^2)$	O(1)	Sí	Datos pequeños
Selection Sort	$O(n^2)$	$O(n^2)$	O(1)	No	Pocos intercambios
Insertion Sort	O(n)	$O(n^2)$	O(1)	Sí	Datos casi ordenados

2.1.5. Uso en ordenamiento de registros y logs de seguridad"

En sistemas SIEM (Security Information and Event Management), los registros deben ordenarse por fecha, IP o nivel de riesgo. Algoritmos simples pueden emplearse en subconjuntos pequeños antes de aplicar técnicas más avanzadas. Ordenar logs facilita la identificación de patrones de ataque y la reconstrucción cronológica de incidentes (Stallings, 2018).

2.2. Análisis algorítmico aplicado a ciberseguridad

El análisis algorítmico permite determinar la eficiencia de los sistemas de monitoreo de seguridad, donde el volumen de datos puede crecer exponencialmente debido al aumento constante de dispositivos conectados, usuarios y servicios digitales. Evaluar la complejidad temporal y espacial de los algoritmos utilizados permite anticipar cuellos de botella, optimizar tiempos de respuesta y garantizar que la detección de amenazas se realice en tiempo real. Sin este análisis, el rendimiento del sistema podría degradarse rápidamente ante cargas elevadas de información.

2.2.1. Comparación de algoritmos de búsqueda y ordenamiento

La búsqueda lineal tiene complejidad O(n), ya que en el peor de los casos debe recorrer todos los elementos hasta encontrar el objetivo o determinar que no existe. En contraste, la búsqueda binaria presenta una complejidad O(log n), pues divide el conjunto de datos en mitades sucesivas, reduciendo drásticamente el número de comparaciones necesarias, siempre que los datos estén previamente ordenados (Cormen et al., 2009). Por tanto, invertir tiempo en ordenar la información puede optimizar de manera significativa el rendimiento de consultas frecuentes, especialmente en sistemas donde se realizan múltiples búsquedas sobre grandes volúmenes de datos.

2.2.2. Medición empírica de eficiencia

Además del análisis teórico basado en modelos matemáticos y notación Big-O, es fundamental realizar pruebas empíricas que midan los tiempos reales de ejecución utilizando conjuntos de datos representativos. Estas pruebas permiten observar el comportamiento del algoritmo en entornos prácticos, considerando factores como arquitectura del hardware, uso de memoria y concurrencia de procesos. En ciberseguridad, esto implica analizar millones de eventos simulados o registros históricos para evaluar el rendimiento bajo condiciones de carga real y validar su escalabilidad. Como señalan Cormen et al. (2009), el análisis experimental complementa el estudio teórico al ofrecer evidencia concreta del desempeño en escenarios aplicados.

Comparación empírica del número de operaciones de diferentes algoritmos de ordenamiento. Documento del proyecto de Tesis que analiza los diferentes algoritmos de búsqueda y experimientos realizados. Enlace: https://ri.uaemex.mx/bitstream/handle/20.500.11799/139241/TesisDavidDaniel.pdf?sequence=1&isAllowed=y

2.2.3. Impacto del algoritmo en sistemas de monitoreo

Un algoritmo ineficiente puede retrasar la detección de ataques en tiempo real, especialmente cuando se procesan grandes volúmenes de tráfico y eventos simultáneos. En entornos críticos, incluso pequeños retrasos pueden impedir la contención oportuna de una amenaza. Los sistemas IDS (Intrusion Detection System) e IPS (Intrusion Prevention System) requieren tiempos de respuesta mínimos para analizar datos y generar alertas casi de inmediato.

Si los algoritmos de clasificación o búsqueda no son eficientes, pueden generarse cuellos de botella que aumenten la latencia, afecten la disponibilidad del servicio y reduzcan la efectividad del monitoreo continuo. Como señala Stallings (2018), la eficiencia computacional es esencial para garantizar la operación confiable de los sistemas de seguridad.

2.2.4. Criterios técnicos de selección algorítmica

Los principales criterios para seleccionar un algoritmo adecuado incluyen varios factores técnicos que influyen directamente en el rendimiento y la sostenibilidad del sistema:

- **Tamaño del conjunto de datos:** A mayor volumen de información, más relevante se vuelve la complejidad temporal del algoritmo.
- **Frecuencia de consultas:** Si las búsquedas u ordenamientos se realizan de forma constante, conviene invertir en algoritmos más eficientes que reduzcan el costo acumulado de operaciones repetidas.
- **Requerimientos de memoria:** Algunos algoritmos requieren espacio adicional significativo, lo cual puede ser una limitación en sistemas con recursos restringidos.
- **Necesidad de estabilidad:** En aplicaciones donde se manejan múltiples criterios de ordenamiento (por ejemplo, fecha y nivel de severidad), la estabilidad del algoritmo puede ser determinante.
- **Tiempo de respuesta en entornos críticos:** En sistemas que operan en tiempo real, la latencia debe mantenerse al mínimo para evitar.

En la ciberseguridad, se prioriza especialmente la eficiencia temporal y la capacidad de escalabilidad, ya que los sistemas deben adaptarse al crecimiento constante del tráfico y de los eventos generados por infraestructuras digitales cada vez más complejas. Como señalan Cormen et al. (2009), la elección de un algoritmo debe basarse en un análisis cuidadoso del contexto de aplicación y de los recursos disponibles, no únicamente en su desempeño teórico.

Definición de los términos citados en la Semana 2

Algoritmos de ordenamiento: Los algoritmos de ordenamiento son procedimientos que organizan un conjunto de datos según un criterio específico, como orden ascendente, descendente o por prioridad. Su objetivo es facilitar el acceso, análisis y procesamiento eficiente de la información dentro de un sistema computacional.

En ciberseguridad, se aplican para organizar logs, eventos y alertas por fecha, severidad o dirección IP. Esto permite detectar patrones de ataque, correlacionar incidentes y mejorar la rapidez en la respuesta ante amenazas.

Costo computacional: El costo computacional de un algoritmo es una medida de los recursos que se requieren para llevar a cabo una tarea específica en un sistema computacional. Estos recursos pueden incluir tiempo de procesamiento, cantidad de memoria utilizada y consumo energético. Su eficiencia depende del costo computacional, es decir, del tiempo y memoria que requieren para ejecutarse según el tamaño del conjunto de datos.

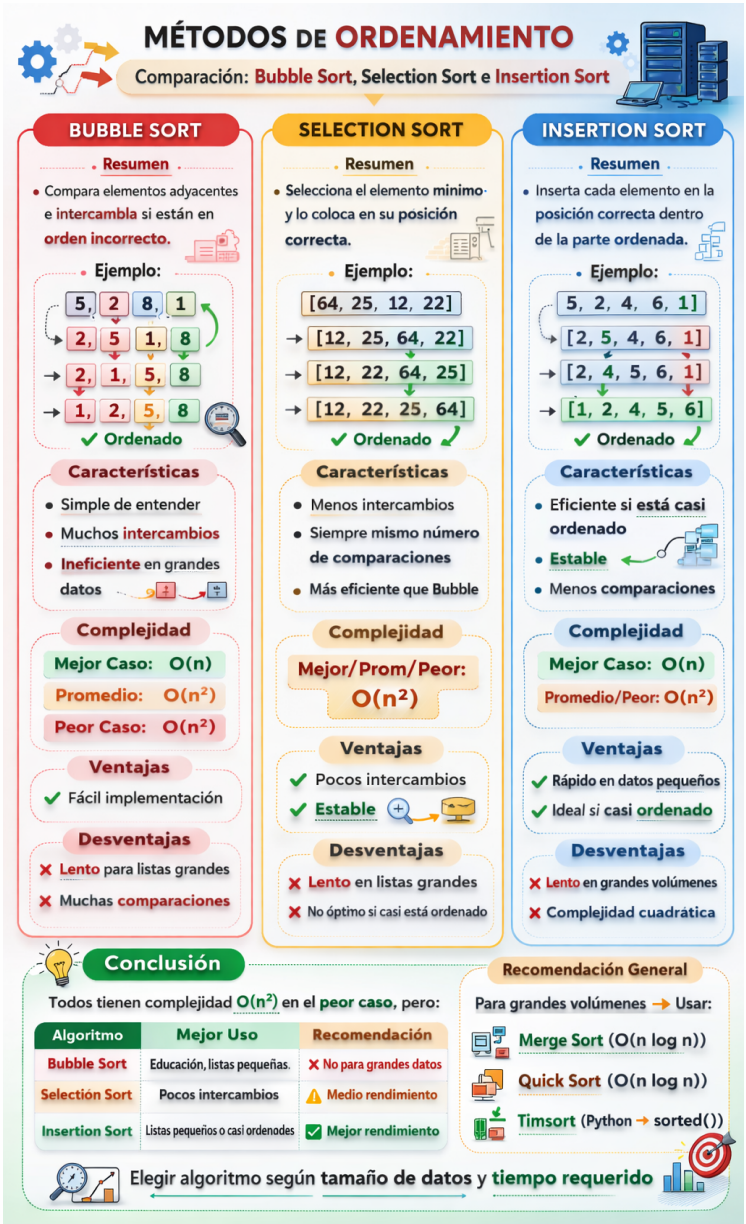
En ciberseguridad, permiten ordenar logs y alertas por fecha o severidad, facilitando la detección de amenazas. Un bajo costo computacional es clave en sistemas que procesan grandes volúmenes de eventos en tiempo real.

Educación de calidad

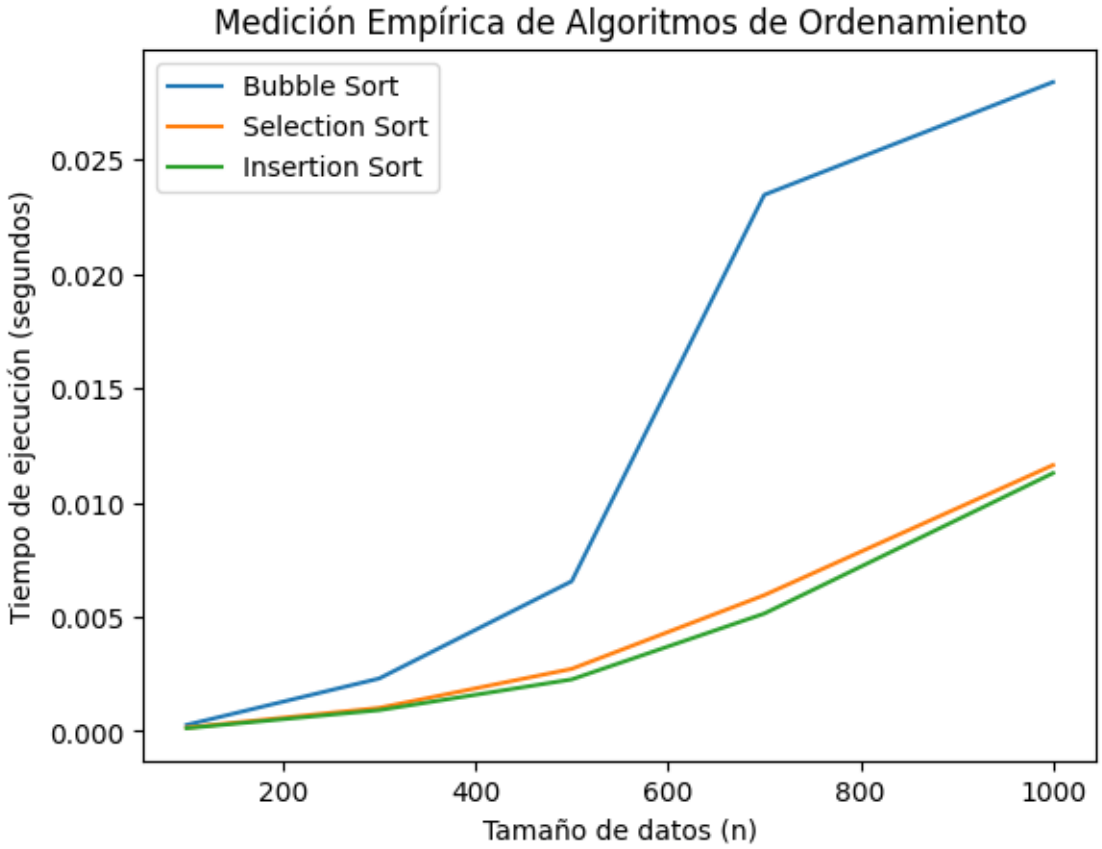
REFERENCIAS

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). MIT Press.
Sedgewick, R., & Wayne, K. (2011). Algorithms (4th ed.). Addison-Wesley.
Weiss, M. A. (2013). Data Structures and Algorithm Analysis in C++ (4th ed.). Pearson.
Stallings, W. (2018). Computer Security: Principles and Practice (4th ed.). Pearson.

Métodos de ordenamiento. Infografía sobre los métodos de ordenamiento Bubble Sort, Selection Sor y Insertion Sort que incluye un ejemplo, ventajas, desventajas, conclusiones y recomendaciones y su aplicación en Ciberseguridad.



Medición empírica de algoritmos de ordenamiento. Gráfico que muestra una medición empírica de la eficiencia de los algoritmos de ordenamiento estudiados (Bubble, Selection, Insertion).



El gráfico muestra claramente la medición empírica:

- A medida que aumenta el tamaño de datos (n), el tiempo crece de forma cuadrática.
- Bubble Sort presenta el crecimiento más acelerado.
- Selection Sort e Insertion Sort muestran comportamiento similar, pero ligeramente más eficiente que Bubble.
- Se evidencia en la práctica la complejidad $O(n^2)$ de los tres métodos.

Laboratorio de Contenido: Bubble Sort en Python

Explicación básica

Bubble Sort ordena comparando elementos adyacentes e intercambiándolos si están en orden incorrecto. En cada pasada, el valor mayor "sube" hasta su posición final. Su complejidad es $O(n^2)$.

Objetivos

- Comprender el funcionamiento de Bubble Sort.
- Implementarlo en Python.
- Analizar su número de comparaciones e intercambios.

Caso práctico

Un sistema registra tiempos de respuesta (en ms) de un servidor:
tiempos = [120, 85, 150, 95, 110]
Se requiere ordenarlos de menor a mayor para identificar los tiempos más críticos.

Problema

Implementar Bubble Sort para ordenar la lista y mostrar el estado de la lista en cada pasada.

Ejemplo paso a paso (Algoritmo básico)

Lista inicial:
120, 85, 150, 95, 110
Primera pasada:
120–85 ✗
120–150 ✓
150–95 ✗
150–110 ✗
Resultado:
85, 120, 95, 110, 150
(150 queda fijo)

Implementación en Python

```
# ----------------------------------------
# Método Bubble Sort en Python
# ----------------------------------------

        def bubble_sort(lista):
            n = len(lista)
            comparaciones = 0
            intercambios = 0
```

```
    for i in range(n):
        hubo_intercambio = False
        print(f"\nPasada {i+1}:")

        for j in range(0, n - i - 1):
            comparaciones += 1
            print(f"  Comparando {lista[j]} y {lista[j+1]}")

            if lista[j] > lista[j + 1]:
                # Intercambio
                lista[j], lista[j + 1] = lista[j + 1], lista[j]
                intercambios += 1
                hubo_intercambio = True
                print(f"  → Se intercambia: {lista}")

        # Optimización: si no hubo intercambio, la lista ya está ordenada
        if not hubo_intercambio:
            break

    print("\nLista ordenada:", lista)
    print("Total comparaciones:", comparaciones)
    print("Total intercambios:", intercambios)

    return lista

# ----------------------------------------
# Programa principal
# ----------------------------------------

if __name__ == "__main__":
    datos = [5, 8, 3, 7, 6]
    print("Lista original:", datos)
    bubble_sort(datos.copy())
```

Reflexión o Análisis

- ¿Cuántas comparaciones realiza?
- ¿Qué sucede si la lista ya está ordenada?
- ¿Por qué no es eficiente para grandes volúmenes de datos?

Laboratorio de Contenido: Selection Sort en Python

Explicación básica

Selection Sort busca el elemento mínimo y lo coloca en su posición correcta en cada iteración. Reduce la cantidad de intercambios, pero mantiene complejidad $O(n^2)$.

Objetivo

- Comprender el proceso de selección del mínimo.
- Implementar Selection Sort.
- Analizar su comportamiento.

Caso práctico

Un sistema de monitoreo registra niveles de severidad de alertas:

alertas = [4, 2, 5, 1, 3]

Se necesita ordenar para priorizar la atención.

Problema

Desarrollar Selection Sort mostrando el valor mínimo encontrado en cada iteración.

Ejemplo paso a paso (Algoritmo básico)

Lista inicial:
4, 2, 5, 1, 3

Primera iteración:
Mínimo = 1
Intercambio con 4

Resultado:
1, 2, 5, 4, 3

Implementación en Python

```
# ----------------------------------------
# Método Selection Sort en Python
# ----------------------------------------
```

```
def selection_sort(lista):
    n = len(lista)
    comparaciones = 0
    intercambios = 0

    for i in range(n):
        minimo = i
        print(f"\nIteración {i+1}:")
        print(f"  Sublista a evaluar: {lista[i:]}")

        for j in range(i + 1, n):
            comparaciones += 1
            print(f"  Comparando {lista[j]} con {lista[minimo]}")

            if lista[j] < lista[minimo]:
                minimo = j
                print(f"  → Nuevo mínimo encontrado: {lista[minimo]}")

        # Solo intercambia si es necesario
        if minimo != i:
            lista[i], lista[minimo] = lista[minimo], lista[i]
            intercambios += 1
            print(f"  → Se intercambia: {lista}")
        else:
            print("  → No se requiere intercambio")

    print("\nLista ordenada:", lista)
    print("Total comparaciones:", comparaciones)
    print("Total intercambios:", intercambios)

    return lista

# ----------------------------------------
# Programa principal
# ----------------------------------------

if __name__ == "__main__":
    datos = [5, 8, 3, 7, 6]
    print("Lista original:", datos)
    selection_sort(datos.copy())
```

Reflexión o Análisis

- ¿Realiza siempre el mismo número de comparaciones?
- ¿Realiza menos intercambios que Bubble Sort?
- ¿En qué casos podría ser preferible?

Laboratorio de Contenido: Insertion Sort en Python

Explicación básica

Insertion Sort toma un elemento y lo inserta en la posición correcta dentro de la parte ya ordenada de la lista. Es eficiente para listas pequeñas o casi ordenadas. Complejidad promedio $O(n^2)$.

Objetivo

- Comprender el proceso de inserción progresiva.
- Implementar Insertion Sort.
- Analizar su eficiencia en listas casi ordenadas.

Caso práctico

Un administrador registra intentos de acceso fallidos:

intentos = [3, 1, 4, 2, 5]

Se requiere ordenarlos para análisis estadístico.

Problema

Implementar Insertion Sort mostrando el estado de la lista después de cada inserción.

Ejemplo paso a paso (Algoritmo básico)

Lista inicial:
3, 1, 4, 2, 5

Se toma 1 y se inserta antes de 3.

Resultado parcial:
1, 3, 4, 2, 5

Implementación en Python

```
# ----------------------------------------
# Método Insertion Sort en Python
# ----------------------------------------
```

```
def insertion_sort(lista):
    comparaciones = 0
    desplazamientos = 0

    for i in range(1, len(lista)):
        clave = lista[i]
        j = i - 1

        print(f"\nIteración {i}:")
        print(f"  Elemento a insertar: {clave}")
        print(f"  Parte ordenada: {lista[:i]}")

        # Mover elementos mayores que la clave
        while j >= 0 and lista[j] > clave:
            comparaciones += 1
            lista[j + 1] = lista[j]
            desplazamientos += 1
            print(f"  → Se desplaza {lista[j]} a la derecha")
            j -= 1

        # Cuenta la última comparación fallida (si aplica)
        if j >= 0:
            comparaciones += 1

        lista[j + 1] = clave
        print(f"  → Lista después de insertar: {lista}")

    print("\nLista ordenada:", lista)
    print("Total comparaciones:", comparaciones)
    print("Total desplazamientos:", desplazamientos)

    return lista

# ----------------------------------------
# Programa principal
# ----------------------------------------

if __name__ == "__main__":
    datos = [5, 8, 3, 7, 6]
    print("Lista original:", datos)
    insertion_sort(datos.copy())))
```

Reflexión o Análisis

- ¿Qué ocurre si la lista está casi ordenada?
- ¿Por qué suele ser más eficiente que Bubble Sort en ese caso?
- ¿En qué tipo de aplicaciones reales podría utilizarse?

Estudio de Caso 1: Ordenamiento de Registros y Logs de Seguridad en un Centro de Monitoreo

Contexto: Una institución educativa cuenta con un Centro de Operaciones de Seguridad (SOC) que monitorea accesos a sus servidores, intentos fallidos de autenticación y actividades sospechosas.

Diariamente se generan miles de registros (logs) con información como:

- Fecha y hora
- Dirección IP
- Tipo de evento
- Nivel de severidad

Para priorizar incidentes críticos, el equipo necesita ordenar los registros por nivel de severidad y por fecha.

Problema: Los registros llegan desordenados, por lo que el analista debe:

1. Ordenarlos por nivel de severidad (1–5).
2. En caso de empate, ordenarlos por fecha más reciente.
3. Procesar los eventos más críticos primero.

Ejemplo de Datos:

```
logs = [
   {"fecha": "2026-02-20 08:15", "ip": "192.168.1.10", "severidad": 3},
   {"fecha": "2026-02-20 08:10", "ip": "192.168.1.25", "severidad": 5},
   {"fecha": "2026-02-20 08:18", "ip": "192.168.1.30", "severidad": 2},
   {"fecha": "2026-02-20 08:12", "ip": "192.168.1.40", "severidad": 5},
]
```

Aplicación del Ordenamiento

Opción 1: Algoritmo básico (Insertion Sort adaptado)

```
def insertion_sort_logs(logs):
   for i in range(1, len(logs)):
      clave = logs[i]
      j = i - 1

      while j >= 0 and (
         logs[j]["severidad"] < clave["severidad"] or
         (logs[j]["severidad"] == clave["severidad"] and logs[j]["fecha"] < clave["fecha"])
```

```
        ):
            logs[j + 1] = logs[j]
            j -= 1

        logs[j + 1] = clave

    return logs

ordenados = insertion_sort_logs(logs.copy())

for log in ordenados:
    print(log)
```

- Ordena primero por severidad descendente
- Luego por fecha más reciente

Resultado Esperado

Los registros con severidad 5 aparecerán primero, ordenados por fecha más reciente. Luego severidad 3, después severidad 2, etc.

Análisis Técnico

Importancia del Ordenamiento en Ciberseguridad

- Permite priorizar incidentes críticos.
- Facilita análisis forense.
- Mejora tiempos de respuesta ante amenazas.
- Reduce riesgo de ignorar eventos graves.

Costo Computacional

- Algoritmos básicos como Bubble, Selection e Insertion → $O(n^2)$
- No son adecuados para millones de registros.
- En entornos reales se utilizan algoritmos más eficientes como:
    - Merge Sort → O(n log n)
    - Quick Sort → O(n log n)
    - Timsort (usado en Python con sorted())

Reflexión Final

En sistemas de ciberseguridad, el ordenamiento no es solo una operación técnica, sino un mecanismo crítico de priorización.
Un algoritmo ineficiente puede retrasar la detección de un ataque activo.

La elección del algoritmo debe considerar:

- Volumen de datos
- Tiempo de respuesta requerido
- Recursos del sistema

Estudio de Caso 2: Medición Empírica de Eficiencia de Algoritmos de Ordenamiento en Logs de Seguridad

Contexto: Un Centro de Operaciones de Seguridad (SOC) analiza miles de registros diarios generados por firewalls, servidores y sistemas de autenticación.

Cada registro contiene:

- Timestamp
- Dirección IP
- Tipo de evento
- Nivel de severidad

Para priorizar incidentes críticos, los registros deben ordenarse por severidad y fecha.

El desafío es determinar qué algoritmo de ordenamiento es más eficiente cuando el volumen de datos crece.

Problema

Comparar empíricamente el tiempo de ejecución de:

- **Bubble Sort**
- **Selection Sort**
- **Insertion Sort**

Aplicados a conjuntos de datos de diferentes tamaños (1 000, 5 000 y 10 000 registros simulados).

Simulación de Datos

```
import random

# Simulación de niveles de severidad
def generar_logs(n):
    return [random.randint(1, 5) for _ in range(n)]
```

Algoritmos Evaluados

Bubble Sort

```
def bubble_sort(lista):
    n = len(lista)
    for i in range(n):
```

```
        for j in range(0, n - i - 1):
            if lista[j] > lista[j + 1]:
                lista[j], lista[j + 1] = lista[j + 1], lista[j]
    return lista
```

Selection Sort

```
def selection_sort(lista):
    n = len(lista)
    for i in range(n):
        minimo = i
        for j in range(i + 1, n):
            if lista[j] < lista[minimo]:
                minimo = j
        lista[i], lista[minimo] = lista[minimo], lista[i]
    return lista
```

Insertion Sort

```
def insertion_sort(lista):
    for i in range(1, len(lista)):
        clave = lista[i]
        j = i - 1
        while j >= 0 and lista[j] > clave:
            lista[j + 1] = lista[j]
            j -= 1
        lista[j + 1] = clave
    return lista
```

Medición Empírica

```
import time

tamaños = [1000, 5000, 10000]

for tamaño in tamaños:
    datos = generar_logs(tamaño)

    for algoritmo in [bubble_sort, selection_sort, insertion_sort]:
        copia = datos.copy()
        inicio = time.time()
```

```
            algoritmo(copia)
            fin = time.time()
            print(f"{algoritmo.__name__} con {tamaño} datos: {fin - inicio:.4f} segundos")

        print("-" * 50)
```

Resultados Esperados (Ejemplo)

Tamaño	Bubble	Selection	Insertion
1 000	0.08 s	0.06 s	0.04 s
5 000	2.1 s	1.8 s	1.5 s
10 000	8.5 s	7.9 s	6.8 s

(Los valores pueden variar según el equipo.)

Análisis

- Los tres algoritmos presentan crecimiento cuadrático **$O(n^2)$**.
- La diferencia se vuelve significativa a medida que aumenta el volumen.
- Insertion Sort suele ser ligeramente más eficiente en listas parcialmente ordenadas.
- Ninguno es óptimo para grandes volúmenes reales de logs.

Aplicación en Ciberseguridad

En entornos reales:

- Los SOC pueden procesar millones de eventos por día.
- Un algoritmo ineficiente puede retrasar la priorización de incidentes críticos.
- La eficiencia impacta directamente en el tiempo de respuesta ante ataques.

Por ello, en sistemas reales se utilizan algoritmos más eficientes como:

- Merge Sort
- Quick Sort
- Timsort (implementado en sorted() de Python)

Conclusión

La medición empírica demuestra que, aunque los métodos básicos son útiles para aprendizaje y conjuntos pequeños, no son adecuados para sistemas de ciberseguridad a gran escala.

La selección del algoritmo debe basarse en:

- Volumen de datos
- Tiempo de respuesta requerido
- Recursos computacionales disponibles

En ciberseguridad, la eficiencia algorítmica es un componente estratégico en la defensa digital.

PUCE Virtual

síguenos en:

www.pucevirtual.puce.edu.ec