PUCE Virtual

CLASE

3

ESTRUCTURA DE DATOS (PROGRAMACIÓN III)

Algoritmos de ordenamiento avanzados

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 desarrollo de algoritmos y estructuras de datos avanzadas constituye un componente esencial en la formación de profesionales capaces de resolver problemas de complejidad media y alta, especialmente en contextos donde la eficiencia computacional es un factor crítico. En el ámbito de la ciberseguridad, los sistemas deben procesar grandes volúmenes de información en tiempo real, lo que exige soluciones algorítmicas escalables y optimizadas. No basta con comprender métodos básicos de ordenamiento; es necesario aplicar técnicas más sofisticadas que garanticen tiempos de respuesta adecuados ante escenarios de alta demanda.

Los algoritmos de ordenamiento avanzados como Merge Sort, Quick Sort y Heap Sort representan un salto cualitativo respecto a los métodos tradicionales. Su complejidad promedio de O(n log n) permite manejar grandes conjuntos de datos con mayor eficiencia, reduciendo significativamente el tiempo de procesamiento en comparación con algoritmos cuadráticos. Estas características resultan fundamentales en centros de operaciones de seguridad (SOC), donde millones de registros deben organizarse por severidad, fecha o tipo de evento para detectar amenazas de manera oportuna.

Además del ordenamiento, la optimización de búsqueda y procesamiento desempeña un papel estratégico. La selección adecuada de estructuras auxiliares como tablas hash, árboles balanceados y colas de prioridad permite reducir la complejidad temporal y mejorar el rendimiento global del sistema. En escenarios de ataque, como intentos masivos de intrusión o eventos distribuidos, la eficiencia algorítmica puede marcar la diferencia entre una respuesta inmediata y una vulnerabilidad crítica.

En esta unidad se abordarán técnicas avanzadas de ordenamiento y estrategias de optimización orientadas a fortalecer la capacidad de análisis y respuesta en entornos de ciberseguridad modernos.

Semana 3:
Desarrollar algoritmos y estructuras de datos avanzadas para la resolución de problemas de complejidad media-alta, considerando criterios de eficiencia computacional en escenarios de ciberseguridad.

3.1. Algoritmos de ordenamiento avanzados

En sistemas de ciberseguridad modernos, el volumen de datos puede alcanzar millones de registros por hora. En estos contextos, los algoritmos básicos $O(n^2)$ resultan ineficientes y generan cuellos de botella. Cuando el número de eventos crece de forma exponencial, como ocurre durante un ataque de denegación de servicio o múltiples intentos de autenticación fallidos, el tiempo de procesamiento puede incrementarse drásticamente, afectando la capacidad de detección y respuesta en tiempo real.

Por ello, se emplean algoritmos avanzados con complejidad O(n log n), que permiten procesar grandes volúmenes de información de manera escalable. Estos métodos optimizan el rendimiento, reducen la latencia en el análisis de eventos y garantizan mayor estabilidad operativa, incluso bajo condiciones de alta carga, fortaleciendo así la resiliencia de los sistemas de monitoreo.

3.1.1. Merge Sort

Algoritmo de ordenamiento basado en la técnica Divide y Vencerás, ampliamente estudiado en la literatura clásica de algoritmos por su eficiencia y comportamiento predecible (Cormen et al., 2009). Su principio fundamental consiste en descomponer un problema grande en subproblemas más pequeños, resolverlos de manera independiente y posteriormente combinar sus soluciones de forma estructurada.

Funcionamiento:

1. Divide la lista en mitades de forma recursiva hasta que cada sublista contenga un solo elemento.
2. Ordena cada mitad (caso base: una lista de un elemento ya está ordenada).
3. Fusiona (merge) las sublistas ordenadas en una única lista ordenada, comparando sus elementos de manera secuencial.

El proceso de fusión es clave para su eficiencia, ya que permite reconstruir la lista completa manteniendo el orden global. Esta estrategia garantiza un rendimiento consistente independientemente de la distribución inicial de los datos.

Complejidad

- **Tiempo**: O(n log n) en el mejor, promedio y peor caso. Esto se debe a que el algoritmo divide el problema en log n niveles y en cada nivel realiza un procesamiento lineal de n elementos (Cormen et al., 2009).

- **Espacio**: O(n), ya que requiere memoria auxiliar para almacenar temporalmente las sublistas durante la fusión.

- **Estabilidad**: Es un algoritmo estable, lo que significa que conserva el orden relativo de elementos con claves iguales (Sedgewick & Wayne, 2011).

En la figura 1 se muestra parte de la aplicación del método de Merge Sort con los datos: 5,3,8,1,6,2

Figura 1. Ejemplo de ejecución del Merge Sort. Tomado de: https://dsa-visualizer-sigma.vercel.app/visualizer/sorting/mergesort

Aplicación en ciberseguridad

En entornos de ciberseguridad, Merge Sort resulta especialmente útil cuando se requiere ordenar grandes volúmenes de datos de manera confiable y predecible.

Por ejemplo:

- Ordenamiento de millones de logs por fecha o nivel de severidad en sistemas SIEM.
- Procesamiento masivo de evidencias en análisis forense digital.
- Sistemas distribuidos donde la estabilidad es crítica para mantener coherencia en múltiples criterios de ordenamiento.

Su comportamiento determinístico y su complejidad garantizada lo convierten en una opción adecuada para escenarios donde la escalabilidad y la consistencia son prioritarias (Weiss, 2013).

Ejemplo en Python:

```
def merge_sort(lista):
    if len(lista) <= 1:
        return lista

    mitad = len(lista) // 2
    izquierda = merge_sort(lista[:mitad])
    derecha = merge_sort(lista[mitad:])

    return merge(izquierda, derecha)

def merge(izq, der):
    resultado = []
    i = j = 0

    while i < len(izq) and j < len(der):
```

```
        if izq[i] < der[j]:
            resultado.append(izq[i])
            i += 1
        else:
            resultado.append(der[j])
            j += 1

    resultado.extend(izq[i:])
    resultado.extend(der[j:])

    return resultado
```

3.1.2. Quick Sort

Algoritmo de ordenamiento que también se fundamenta en la estrategia Divide y Vencerás, pero a diferencia de Merge Sort, no divide la lista exactamente en mitades. En su lugar, selecciona un elemento denominado pivote y reorganiza los datos en torno a él mediante un proceso llamado partición. Durante esta fase, los elementos menores que el pivote se ubican a su izquierda y los mayores a su derecha. Posteriormente, el algoritmo aplica recursivamente el mismo procedimiento a cada sublista hasta que el conjunto completo queda ordenado.

Complejidad

- **Mejor y promedio:** O(n log n), cuando el pivote divide el arreglo de manera equilibrada.
- **Peor caso:** O(n^2), cuando el pivote genera particiones muy desbalanceadas (por ejemplo, si los datos ya están ordenados y se elige siempre el primer elemento).
- **Espacio:** O(log n), debido a la profundidad de la recursión.
- **Estabilidad:** No es estable por defecto, ya que puede alterar el orden relativo de elementos iguales.

Ventajas

- Muy rápido en la práctica, especialmente en conjuntos grandes.
- Bajo uso de memoria adicional.
- Excelente rendimiento promedio, lo que lo convierte en uno de los algoritmos más utilizados en implementaciones reales.

Aplicación en ciberseguridad

En entornos de ciberseguridad, Quick Sort es ideal para la clasificación rápida de eventos por nivel de severidad o criticidad, permitiendo priorizar incidentes de alto riesgo. Asimismo, en sistemas SIEM facilita la organización eficiente de alertas, contribuyendo a reducir la latencia en la detección y respuesta ante amenazas. En la figura 2 se muestra la forma de trabajar del método Quick Sort.

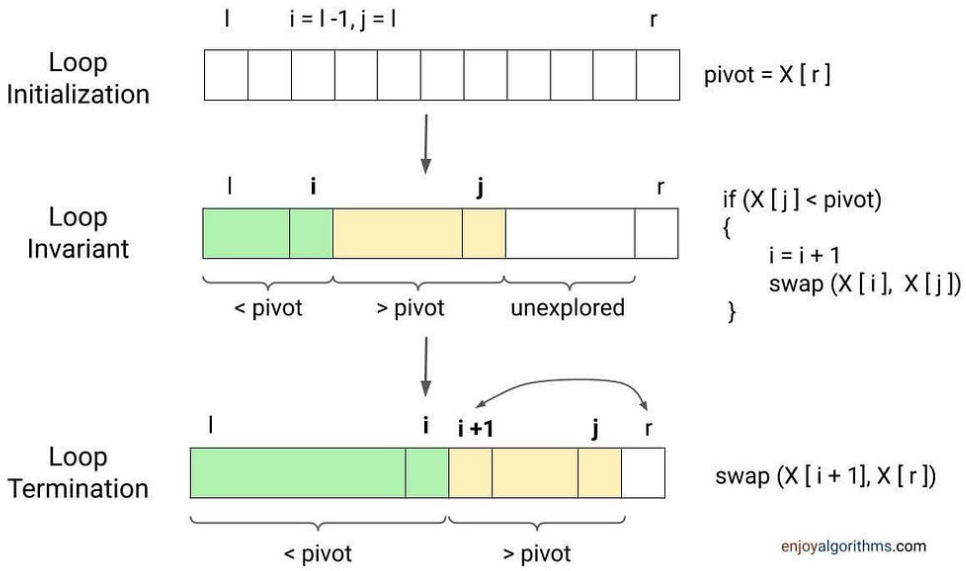


Figura 2. Algoritmo de Quick Sort. Tomado de: https://www.enjoyalgorithms.com/blog/quick-sort-algorithm?ref=tlouarn.com

La figura 3 muestra un ejemplo de la ejecución del método Quick Sort.

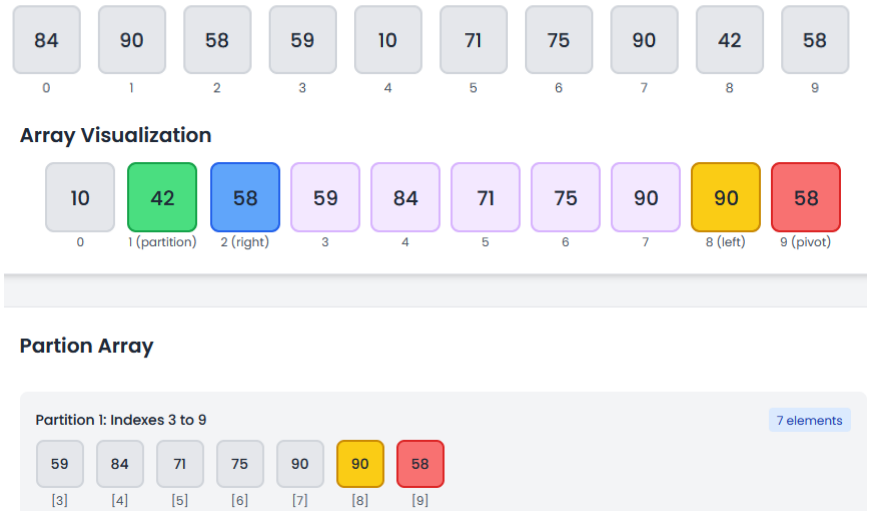


Figura 3. Ejemplo de ejecución de Quick Sort. Tomado de: https://dsa-visualizer-sigma.vercel.app/visualizer

3.1.3. Heap Sort

Algoritmo de ordenamiento que se basa en una estructura de datos denominada heap o montículo binario, la cual puede representarse como un árbol binario completo que cumple la propiedad de orden. En su versión más común para ordenamiento ascendente, se construye un **Max-Heap**, donde cada nodo padre es mayor o igual que sus hijos. Esta propiedad garantiza que el elemento de mayor valor se encuentre siempre en la raíz del árbol.

Características

- Construye inicialmente un Max-Heap a partir del arreglo original.
- Extrae el elemento mayor (ubicado en la raíz) y lo intercambia con el último elemento del arreglo.
- Reduce el tamaño del heap y restaura la propiedad de montículo mediante el proceso de *heapify*.
- Repite el procedimiento hasta que todos los elementos quedan ordenados.

Complejidad

- **Tiempo:** O(n log n) en mejor, promedio y peor caso, lo que lo convierte en un algoritmo con rendimiento garantizado y predecible.
- **Espacio:** O(1), ya que realiza el ordenamiento en el mismo arreglo sin requerir memoria auxiliar significativa.
- **Estabilidad:** No es estable, porque los intercambios pueden alterar el orden relativo de elementos con el mismo valor.

La Figura 4 muestra un ejemplo de el funcionamiento de Heap Sort.

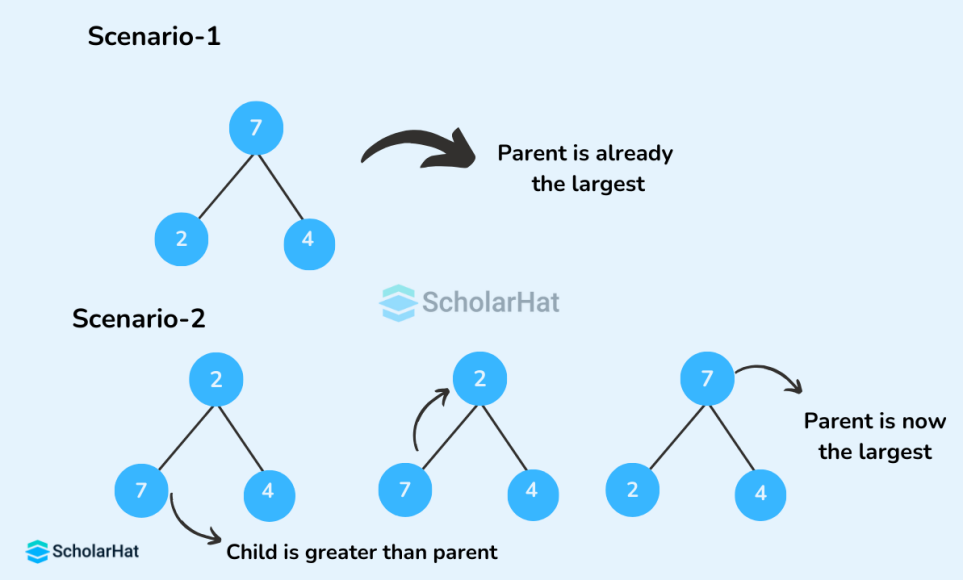


Figura 4. Ejemplo de ejecución Heap Sort. Tomado de: https://www.scholarhat.com/tutorial/datastructures/heap-sort-in-data-structures

Aplicación en ciberseguridad

Heap Sort es especialmente útil en sistemas donde se necesita garantizar un peor caso controlado y evitar degradaciones inesperadas del rendimiento. Además, su fundamento en estructuras tipo heap lo hace ideal para el procesamiento continuo de prioridades, como en colas de prioridad utilizadas para gestionar alertas críticas, asignar recursos de respuesta o clasificar eventos de seguridad según su nivel de riesgo.

3.1.4. Comparación con métodos básicos

En comparación con los métodos básicos de ordenamiento como Bubble, Selection e Insertion, los algoritmos avanzados como Merge Sort, Quick Sort y Heap Sort ofrecen una mejora significativa en eficiencia. Mientras los métodos básicos presentan complejidad $O(n^2)$, los avanzados operan en O(n log n), lo que los hace mucho más adecuados para grandes volúmenes de datos. Esta diferencia es determinante en sistemas que requieren escalabilidad y tiempos de respuesta reducidos, especialmente cuando el volumen de información crece considerablemente (Cormen et al., 2009). La tabla 1 muestra la comparación entre los métodos de ordenamiento tomando el mejor y peor caso, así como el espacio y estabilidad.

Tabla 1. Comparación de métodos de ordenamiento. Creación de autor Patricio Coba

Algoritmo	Mejor Caso	Peor Caso	Espacio	Estable
Bubble Sort	O(n)	$O(n^2)$	O(1)	Sí
Insertion Sort	O(n)	$O(n^2)$	O(1)	Sí
Merge Sort	O(n log n)	O(n log n)	O(n)	Sí
Quick Sort	O(n log n)	$O(n^2)$	O(log n)	No
Heap Sort	O(n log n)	O(n log n)	O(1)	No

Conclusión técnica:
En entornos de ciberseguridad con grandes volúmenes de datos, los algoritmos O(n log n) son prácticamente obligatorios para garantizar escalabilidad y mantener un rendimiento estable. Los centros de operaciones de seguridad (SOC) procesan continuamente millones de eventos provenientes de firewalls, servidores, aplicaciones y dispositivos de red. Si se emplearan algoritmos con complejidad $O(n^2)$, el tiempo de procesamiento crecería de forma desproporcionada a medida que aumenta el número de registros, generando retrasos críticos en la detección y respuesta ante incidentes.

Algoritmos de ordenamiento. En la página de Sort Vision se puede realizar una visualización interactiva de algoritmos de ordenamiento populares y que tiene métricas, detalles y comparaciones muy buenas para que el estudiante pueda aprender. En enlace es: https://www.sortvision.com

Los algoritmos con complejidad O(n log n), como Merge, Quick o Heap Sort, permiten manejar este crecimiento de manera mucho más eficiente, asegurando tiempos de respuesta adecuados incluso bajo condiciones de alta carga o durante ataques masivos. Esta eficiencia no solo mejora el desempeño técnico del sistema, sino que también fortalece la capacidad de análisis en tiempo real, reduce la latencia operativa y contribuye a la resiliencia general de la infraestructura de seguridad.

3.1.5. Aplicaciones en grandes volúmenes de datos de seguridad

En un SOC (Security Operations Center), la gestión eficiente de la información es un factor crítico para garantizar la seguridad institucional.

- Se procesan millones de eventos diarios, provenientes de múltiples fuentes como firewalls, sistemas de detección de intrusos, servidores, aplicaciones, endpoints y dispositivos de red. Cada uno de estos genera registros que deben almacenarse, clasificarse y analizarse de manera continua.

- Se requiere priorización inmediata, ya que no todos los eventos representan el mismo nivel de riesgo. Es fundamental identificar rápidamente aquellos incidentes de alta severidad, como intentos de acceso no autorizado o comportamientos anómalos, para asignar recursos y activar protocolos de respuesta.

- Se deben correlacionar eventos en tiempo real, porque muchas amenazas no se detectan por un solo registro aislado, sino por la relación entre múltiples eventos distribuidos en el tiempo. La capacidad de analizar patrones y vincular actividades sospechosas permite anticipar ataques y reducir el impacto de posibles incidentes de seguridad.

Ejemplos prácticos:

- Ordenar logs por severidad y timestamp.
- Identificar patrones de ataque distribuidos.
- Detectar ataques de fuerza bruta mediante correlación masiva.

Un algoritmo ineficiente puede retrasar la detección de un ataque activo.

3.2. Optimización de búsqueda y procesamiento

La optimización implica elegir un buen algoritmo de ordenamiento y además mejorar de manera integral las estrategias de búsqueda, almacenamiento y procesamiento masivo de datos. En sistemas de alto rendimiento, especialmente en ciberseguridad, es necesario analizar cómo interactúan los algoritmos con las estructuras de datos y con la arquitectura del sistema. Una solución eficiente debe considerar la reducción de la complejidad temporal, el uso adecuado de la memoria y la capacidad de escalar ante incrementos significativos en el volumen de información.

Además, la optimización puede incluir el uso de indexación, tablas hash para búsquedas rápidas, árboles balanceados para mantener datos ordenados dinámicamente y estructuras como heaps para gestionar prioridades. También es fundamental aplicar técnicas como paralelización, procesamiento por lotes o análisis en streaming, que permiten distribuir la carga y disminuir la latencia. En conjunto, estas estrategias garantizan que el sistema pueda responder en tiempo real, incluso bajo escenarios de alta demanda o durante incidentes de seguridad masivos.

3.2.1. Estrategias de optimización algorítmica

Entre las estrategias de optimización algorítmica están:

- **Uso de Divide y Vencerás:** Esta estrategia consiste en descomponer un problema complejo en subproblemas más pequeños y manejables, resolverlos de forma independiente y luego combinar sus soluciones. Permite reducir la complejidad de ciertos algoritmos y mejorar significativamente el rendimiento, como ocurre en Merge Sort y Quick Sort. Es especialmente útil cuando se procesan grandes volúmenes de datos que pueden dividirse en bloques.
- **Programación dinámica:** Se basa en almacenar resultados intermedios para evitar cálculos repetitivos. Es eficaz en problemas donde existen subestructuras óptimas y superposición de subproblemas. Su aplicación reduce el tiempo de ejecución en escenarios complejos como análisis de patrones o detección de anomalías.
- **Estructuras auxiliares eficientes:** El uso de tablas hash, árboles balanceados, heaps o colas de prioridad permite acelerar búsquedas, inserciones y clasificaciones. Elegir la estructura adecuada puede reducir significativamente la complejidad temporal.
- **Paralelización:** Consiste en dividir tareas para ejecutarlas simultáneamente en múltiples núcleos o procesadores, disminuyendo el tiempo total de procesamiento en sistemas de alta demanda.
- **Indexación de datos:** Permite acceder rápidamente a información específica sin recorrer todo el conjunto de datos, optimizando consultas frecuentes y reduciendo la latencia.

Optimización de algoritmos. Página web donde se hace un repaso de conceptos básicos sobre la optimización de algoritmos con una explicación clara y resumida. El enlace es: https://4geeks.com/es/lesson/optimizacion-de-algoritmos-y-estructuras-de-datos

3.2.2. Reducción de complejidad temporal

La reducción de la complejidad temporal es uno de los objetivos centrales en el diseño de algoritmos eficientes, ya que impacta directamente en el tiempo de respuesta del sistema. La notación Big-O permite estimar cómo crece el número de operaciones en función del tamaño de los datos, lo que resulta clave cuando se trabajan grandes volúmenes de información (Weiss, 2013).

Ejemplo comparativo:

- **Búsqueda lineal O(n):** Recorre secuencialmente todos los elementos hasta encontrar el objetivo o confirmar su ausencia.
- **Búsqueda binaria O(log n):** Reduce el espacio de búsqueda a la mitad en cada iteración, siempre que los datos estén ordenados.
- **Hashing O(1) promedio:** Permite acceso casi inmediato mediante funciones de dispersión que asignan claves a posiciones específicas en memoria.

En monitoreo de seguridad, esta diferencia es crítica. Reducir una operación de O(n) a O(log n) o incluso a O(1) puede significar detectar patrones de ataque en milisegundos en lugar de segundos. Cuando se procesan millones de eventos en tiempo real, esa optimización puede

marcar la diferencia entre contener una amenaza oportunamente o permitir que escale y comprometa la infraestructura. La eficiencia temporal, por tanto, no es solo un criterio técnico, sino un factor estratégico en la defensa digital.

En la figura 5 se muestra una comparativa de la eficiencia temporal de los algoritmos de búsqueda lineal, búsqueda binaria y Hashing.

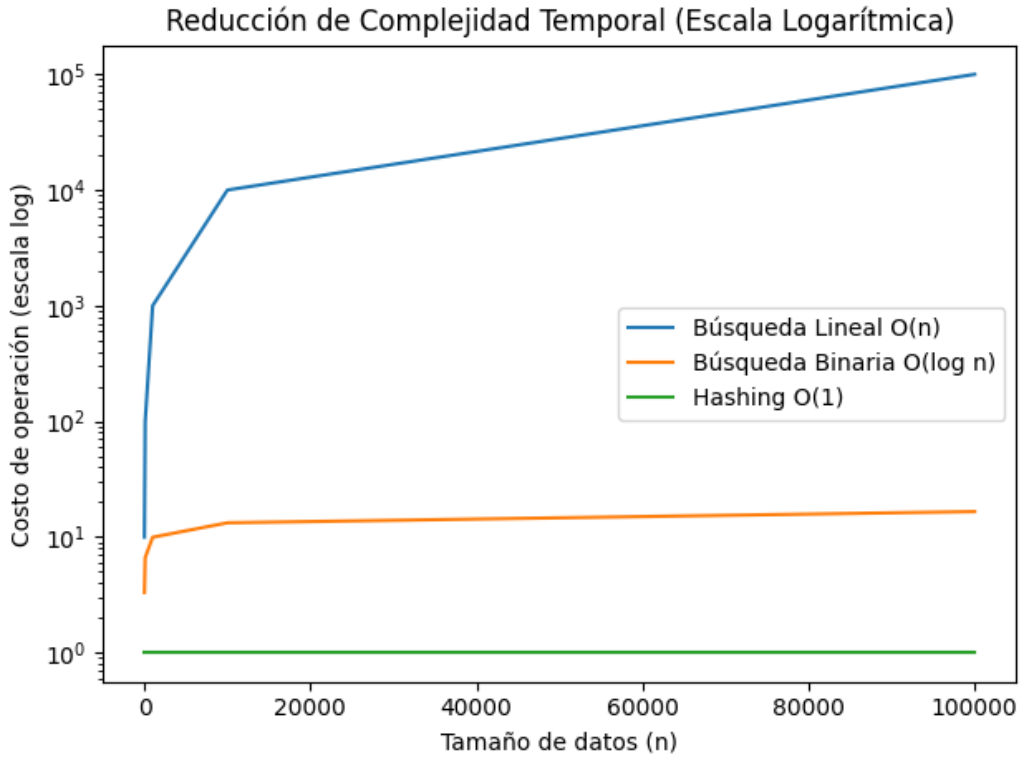


Figura 5. Comparativa de eficiencia temporal. Creación de autor Patricio Coba

3.2.3. Procesamiento eficiente de eventos masivos

En entornos de ciberseguridad, los sistemas deben manejar flujos continuos de información provenientes de múltiples fuentes. El procesamiento eficiente de eventos masivos implica aplicar técnicas que permitan analizar grandes volúmenes de datos sin afectar la disponibilidad del sistema.

Técnicas aplicadas en ciberseguridad:

- **Procesamiento por lotes:** Consiste en agrupar grandes cantidades de datos y procesarlos en intervalos definidos. Es útil para análisis históricos, generación de reportes y detección de patrones complejos que no requieren respuesta inmediata.
- **Streaming en tiempo real:** Permite analizar eventos a medida que se generan, reduciendo la latencia y facilitando la detección temprana de amenazas activas. Es fundamental en sistemas de monitoreo continuo y detección de intrusiones.
- **Uso de estructuras eficientes:**
  - **Árboles balanceados:** Mantienen los datos ordenados dinámicamente, permitiendo búsquedas, inserciones y eliminaciones en O(log n).

- **Tablas hash:** Ofrecen acceso casi inmediato a información específica.
- **Heaps:** Permiten gestionar prioridades de forma eficiente.
- **Colas de prioridad:** Facilitan la atención inmediata de eventos críticos.

3.2.4. Análisis de rendimiento en escenarios de ataque

En ataques como DDoS o intentos masivos de fuerza bruta, el volumen de eventos puede crecer de forma exponencial en cuestión de segundos.

- El sistema debe procesar miles o millones de solicitudes simultáneamente.
- Se requiere análisis en tiempo real para bloquear amenazas activas.
- Algoritmos $O(n^2)$ pueden generar cuellos de botella y colapsar el sistema.

El análisis de rendimiento debe considerar:

- **Tiempo de respuesta**, para garantizar detección inmediata.
- **Uso de memoria**, evitando saturación de recursos.
- **Escalabilidad**, permitiendo soportar incrementos abruptos de carga.
- **Capacidad de procesamiento concurrente**, aprovechando arquitecturas multinúcleo o distribuidas.

La figura 6 presenta estadísticas sobre ataques de DDOS entre 2019 y 2024 por región.

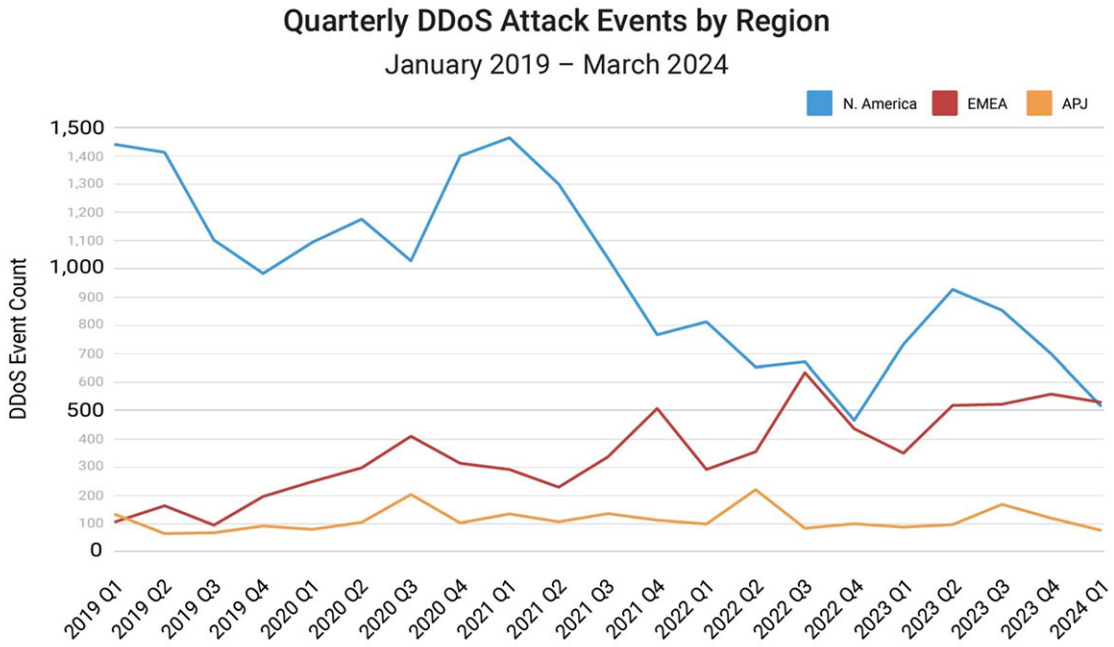


Figura 6. Estadísticas de ataques DDOS. Fuente: Informe 'Apagando fuegos: el auge de las amenazas DDoS en EMEA' de Akamai Technologies.

3.2.5. Uso estratégico de estructuras auxiliares

El rendimiento no depende únicamente del algoritmo, sino también de la estructura de datos utilizada.

Estructuras clave en ciberseguridad:

- **Heap:** Ideal para priorizar alertas según severidad o criticidad.
- **Hash Table:** Permite búsqueda instantánea de direcciones IP sospechosas o firmas conocidas.
- **Árboles balanceados (AVL o Red-Black):** Facilitan la indexación ordenada y consultas eficientes.
- **Colas de prioridad:** Gestionan incidentes críticos asignando recursos según nivel de riesgo.

La combinación de algoritmos eficientes y estructuras adecuadas permite optimizar el rendimiento global del sistema, reducir la latencia y mejorar la capacidad de respuesta ante amenazas complejas.

Definición de los términos citados en la Semana 3

Algoritmos avanzados: Los algoritmos avanzados son aquellos diseñados para resolver problemas de mayor complejidad computacional mediante técnicas sofisticadas como Divide y Vencerás, programación dinámica, estructuras de datos especializadas y optimización del uso de recursos. Se caracterizan por ofrecer mejores niveles de eficiencia temporal y espacial en comparación con métodos básicos, especialmente en escenarios donde el volumen de datos es elevado. Su análisis se fundamenta en la teoría de la complejidad algorítmica y en la evaluación formal del rendimiento mediante notación Big-O (Cormen et al., 2009).

Algoritmos con complejidad: La complejidad de un algoritmo es una medida que describe la cantidad de recursos computacionales que requiere para ejecutarse, principalmente tiempo (complejidad temporal) y memoria (complejidad espacial), en función del tamaño de los datos de entrada. Se expresa comúnmente mediante la notación Big-O, que permite analizar cómo crece el número de operaciones cuando aumenta el volumen de información. Este análisis es fundamental para comparar algoritmos y determinar su eficiencia y escalabilidad (Cormen et al., 2009).

Educación de calidad

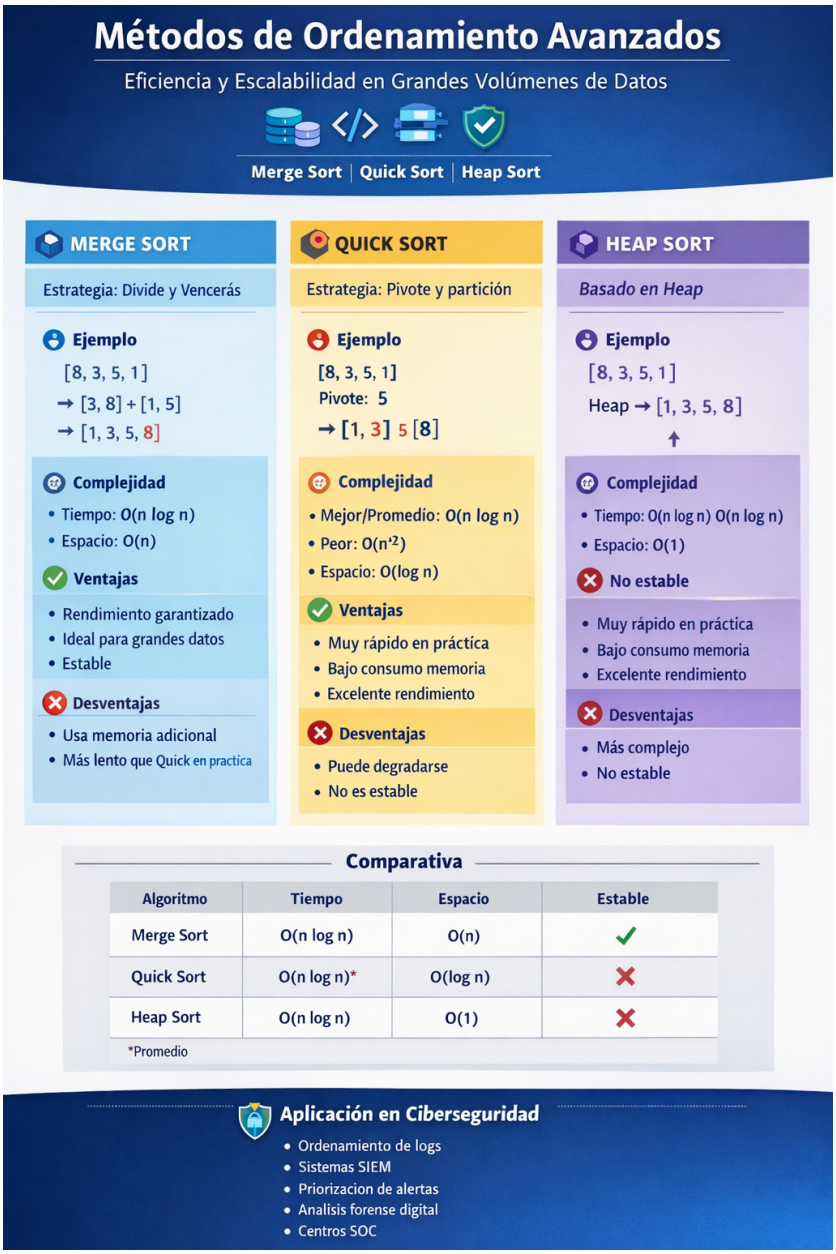
REFERENCIAS

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.

Métodos de ordenamiento avanzados. Infografía sobre los métodos de ordenamiento avanzados que incluye un ejemplo, ventajas, desventajas y su aplicación en Ciberseguridad.



Optimización de búsqueda y procesamiento. Infografía que presenta las estrategias para la eficiencia de optimización de algoritmos, reducción de complejidad y el procesamiento de eventos masivos.

Laboratorio de Contenido 1: Merge Sort en análisis de logs de seguridad

Explicación básica

Merge Sort es un algoritmo basado en la técnica *Divide y Vencerás*. Divide el arreglo en mitades recursivamente, ordena cada mitad y luego las fusiona manteniendo el orden. Tiene complejidad O(n log n) en todos los casos y es estable.

Objetivos

- Comprender el funcionamiento recursivo de Merge Sort.
- Analizar su complejidad temporal y espacial.
- Aplicarlo al ordenamiento de registros de seguridad.

Caso práctico

Un sistema SOC recibe registros con marcas de tiempo desordenadas y necesita organizarlos cronológicamente para análisis forense.

Problema

Ordenar la siguiente lista de timestamps (en segundos):
[45, 12, 78, 34, 23, 90, 11]

Ejemplo paso a paso

División:

[45,12,78] [34,23,90,11]

Subdivisión:

[45] [12,78]

[34,23] [90,11]

Ordenar y fusionar:

[12,45,78]

[11,23,34,90]

Resultado final:

[11,12,23,34,45,78,90]

Implementación en Python

```
def merge_sort(arr):
    if len(arr) <= 1:
        return arr

    mid = len(arr) // 2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])

    return merge(left, right)

def merge(left, right):
    result = []
```

```
    i = j = 0

    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1

    result.extend(left[i:])
    result.extend(right[j:])

    return result

datos = [45, 12, 78, 34, 23, 90, 11]
print(merge_sort(datos))
```

Conclusiones

Merge Sort garantiza rendimiento estable incluso en grandes volúmenes de datos. Es ideal cuando la estabilidad es crítica, como en el ordenamiento de logs con misma prioridad o misma marca temporal.

Laboratorio de Contenido 2: Quick Sort en priorización de alertas

Explicación básica

Quick Sort también usa Divide y Vencerás, pero selecciona un pivote y divide los datos en menores y mayores al pivote. Tiene rendimiento promedio O(n log n), aunque puede degradarse a $O(n^2)$.

Objetivos

- Comprender el concepto de pivote.
- Evaluar ventajas prácticas frente a otros algoritmos.
- Aplicarlo a priorización de eventos.

Caso práctico

Un SIEM necesita ordenar alertas por nivel de severidad (1–100).

Problema

Ordenar niveles de severidad:

[70, 15, 90, 45, 30, 85, 60]

Ejemplo paso a paso

Pivote: 60

Menores:

[15,45,30]

Mayores:

[70,90,85]

Resultado final:

[15,30,45,60,70,85,90]

Implementación en Python

```
def quick_sort(arr):
    if len(arr) <= 1:
        return arr

    pivot = arr[len(arr)//2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]

    return quick_sort(left) + middle + quick_sort(right)

datos = [70, 15, 90, 45, 30, 85, 60]
print(quick_sort(datos))
```

Conclusiones

Quick Sort es extremadamente rápido en la práctica y usa poca memoria. Es ideal cuando se requiere rapidez promedio en clasificación de eventos, aunque se debe cuidar la selección del pivote.

Laboratorio de Contenido 3: Heap Sort en sistemas de prioridad

Explicación básica

Heap Sort utiliza una estructura llamada heap (montículo binario). Construye un Max-Heap y extrae repetidamente el elemento mayor. Garantiza O(n log n) en todos los casos y usa O(1) espacio adicional.

Objetivos

- Comprender la estructura heap.
- Analizar su rendimiento garantizado.
- Aplicarlo a colas de prioridad.

Caso práctico

Un sistema gestiona incidentes críticos y debe procesar primero los de mayor prioridad.

Problema

Ordenar prioridades:

[4, 9, 1, 7, 3, 8]

Ejemplo paso a paso

Construcción del Max-Heap:

[9,7,8,4,3,1]

Extracciones sucesivas:

[1,3,4,7,8,9]

Implementación en Python

```
def heapify(arr, n, i):
  largest = i
  left = 2*i + 1
  right = 2*i + 2

  if left < n and arr[left] > arr[largest]:
    largest = left

  if right < n and arr[right] > arr[largest]:
    largest = right

  if largest != i:
    arr[i], arr[largest] = arr[largest], arr[i]
    heapify(arr, n, largest)

def heap_sort(arr):
  n = len(arr)

  for i in range(n//2 - 1, -1, -1):
    heapify(arr, n, i)
```

```
    for i in range(n-1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]
        heapify(arr, i, 0)

    return arr

datos = [4, 9, 1, 7, 3, 8]
print(heap_sort(datos))
```

Conclusiones

Heap Sort es ideal cuando se requiere rendimiento garantizado en el peor caso. Es ampliamente utilizado en sistemas de prioridad y gestión de eventos críticos en ciberseguridad.

ESTUDIO DE CASO 1: Métodos Avanzados de Ordenamiento en un Sistema SIEM

Contexto

Una organización financiera implementa un SIEM (Security Information and Event Management) que recibe más de 2 millones de eventos diarios provenientes de firewalls, IDS, servidores y endpoints.

Los eventos llegan desordenados y deben clasificarse por:

- Fecha y hora
- Nivel de severidad
- Dirección IP

El sistema inicialmente utilizaba ordenamiento básico (Bubble Sort), generando retrasos significativos.

Problema

El tiempo de procesamiento superaba los 40 segundos por lote de 100.000 registros, afectando la capacidad de respuesta ante incidentes críticos.

Solución Propuesta

Se evaluaron tres algoritmos avanzados:

Merge Sort

- Garantiza O(n log n)
- Estable (útil para mantener orden por timestamp)
- Requiere memoria adicional

Quick Sort

- Muy rápido en promedio
- Bajo consumo de memoria
- Riesgo de peor caso $O(n^2)$

Heap Sort

- Garantiza O(n log n)
- Espacio O(1)
- Ideal para priorización

Implementación y Resultados

Después de pruebas controladas:

Algoritmo	Tiempo promedio (100k eventos)
Bubble Sort	**40 s**
Merge Sort	**2.1 s**
Quick Sort	**1.8 s**
Heap Sort	**2.3 s**

Quick Sort mostró mejor rendimiento promedio. Sin embargo:

- Merge Sort fue elegido para ordenamiento por timestamp (estabilidad).
- Heap fue utilizado para priorización de alertas críticas.

Resultados

- Reducción del tiempo de procesamiento en 95%.
- Mejora en detección temprana de incidentes.
- Mayor escalabilidad ante crecimiento de datos.

Reflexión

Los métodos avanzados de ordenamiento no solo mejoran eficiencia, sino que impactan directamente la capacidad de respuesta en ciberseguridad. La elección del algoritmo depende del contexto: estabilidad, memoria disponible y comportamiento del peor caso.

ESTUDIO DE CASO 2

Aplicaciones en Grandes Volúmenes de Datos de Seguridad (Análisis DDoS)

Contexto

Un proveedor de servicios en la nube detecta un ataque DDoS que genera más de 5 millones de solicitudes por minuto.

El sistema debe:

- Identificar IPs sospechosas.
- Detectar patrones repetitivos.
- Priorizar eventos críticos.
- Generar alertas en tiempo real.

Problema

El sistema inicial utilizaba:

- Búsqueda lineal para identificar IPs repetidas.
- Ordenamiento básico para clasificar tráfico.
- Procesamiento secuencial.

Resultado: Colapso del sistema bajo alta carga.

Solución Implementada

Se aplicaron estrategias de optimización:

Reducción de Complejidad Temporal

- Hash Table → O(1) promedio para identificar IPs sospechosas.
- Búsqueda binaria → O(log n) para listas ordenadas.

Procesamiento Eficiente

- Streaming en tiempo real.
- Procesamiento paralelo.
- Indexación de logs.

Uso de Estructuras Auxiliares

- Heap → priorización de IPs más activas.
- Árboles balanceados → mantenimiento ordenado de registros.

Resultados Medidos

Métrica	Antes	Después
Tiempo de detección	8 s	120 ms
Uso de CPU	95%	60%
Capacidad de procesamiento	1M req/min	6M req/min

Impacto

- Detección casi inmediata del ataque.
- Bloqueo automatizado de IPs maliciosas.
- Escalabilidad horizontal efectiva.

Reflexión

En escenarios de grandes volúmenes de datos, la optimización algorítmica no es opcional, es obligatoria. La combinación de:

- Algoritmos O(n log n)
- Hashing O(1)
- Paralelización
- Indexación

Permite mantener la disponibilidad del sistema incluso bajo ataques masivos.

PUCE Virtual

síguenos en:

www.pucevirtual.puce.edu.ec