PUCE Virtual

DOCUMENTO DE PROFUNDIZACIÓN

Estructura de Datos

Laboratorio de Contenido 3: Análisis Algorítmico del Sistema

Explicación básica

El análisis algorítmico evalúa la eficiencia de los algoritmos en términos de tiempo y uso de recursos. En ciberseguridad, esto es clave para procesar grandes volúmenes de datos (logs, eventos, redes). Se enfoca en algoritmos de búsqueda, ordenamiento y estructuras como árboles y grafos, analizando su rendimiento mediante conceptos como la complejidad temporal (Big-O).

Objetivos

- Comprender la eficiencia de algoritmos de búsqueda y ordenamiento.
- Analizar el rendimiento de estructuras de datos (árboles y grafos).
- Comparar diferentes enfoques algorítmicos.
- Aplicar estos conceptos en un contexto de ciberseguridad.

Caso práctico

Una empresa necesita analizar miles de eventos de seguridad para:

- Buscar accesos sospechosos rápidamente.
- Ordenar eventos por prioridad o fecha.
- Analizar conexiones entre dispositivos (red).

Problema

El sistema actual:

- Usa búsqueda lineal (lenta con muchos datos).
- No ordena eficientemente los eventos.
- No analiza relaciones entre dispositivos.

Aplicación paso a paso

Paso 1: Búsqueda lineal (ineficiente)

```
def busqueda_lineal(logs, usuario):
    for log in logs:
        if log["usuario"] == usuario:
            return log
    return None
```

Complejidad: **O(n)**, lenta con grandes volúmenes

Paso 2: Búsqueda binaria (optimizada)

Requiere datos ordenados:

```
def busqueda_binaria(logs, usuario):
    izquierda, derecha = 0, len(logs) - 1

    while izquierda <= derecha:
        medio = (izquierda + derecha) // 2
        if logs[medio]["usuario"] == usuario:
            return logs[medio]
        elif logs[medio]["usuario"] < usuario:
            izquierda = medio + 1
        else:
            derecha = medio - 1

    return None
```

Complejidad: **O(log n)**, mucho más rápida

Paso 3: Ordenamiento (eficiencia del sistema)

```
def ordenar_logs(logs):
    return sorted(logs, key=lambda x: x["fecha"])
```

Complejidad: **O(n log n)**, permite búsquedas más eficientes

Paso 4: Uso de árboles (búsqueda eficiente)

```
class Nodo:
    def __init__(self, valor):
        self.valor = valor
        self.izquierda = None
        self.derecha = None

def insertar(raiz, valor):
    if raiz is None:
        return Nodo(valor)
    if valor < raiz.valor:
        raiz.izquierda = insertar(raiz.izquierda, valor)
    else:
        raiz.derecha = insertar(raiz.derecha, valor)
    return raíz
```

Búsqueda promedio: **O(log n)**, puede degradarse a O(n) si no está balanceado

Paso 5: Uso de grafos (análisis de red)

Representamos conexiones entre dispositivos:

```
grafo = {
   "PC1": ["Router"],
   "Router": ["PC1", "Servidor"],
   "Servidor": ["Router"]
}
```

Búsqueda en anchura (BFS):

```
from collections import deque

def bfs(grafo, inicio):
   visitados = set()
   cola = deque([inicio])

   while cola:
      nodo = cola.popleft()
      if nodo not in visitados:
         print(nodo)
         visitados.add(nodo)
         cola.extend(grafo[nodo])
```

Complejidad: **O(V + E)**

Paso 6: Comparación de eficiencia

Método	Complejidad	Uso en ciberseguridad
Búsqueda lineal	O(n)	Datos pequeños
Búsqueda binaria	O(log n)	Logs ordenados
Ordenamiento	O(n log n)	Preparar datos
Árboles	O(log n)	Índices de búsqueda
Grafos (BFS/DFS)	O(V+E)	Redes y ataques

Conclusiones

- La eficiencia algorítmica es clave en sistemas de ciberseguridad.

- Elegir el algoritmo correcto mejora el rendimiento significativamente.
- Las estructuras como árboles y grafos permiten análisis más avanzados.
- Un mal diseño puede volver lento e ineficiente todo el sistema.