Tecnológico de Monterrey

Reflexión personal

Javier Barron Vargas
A00842507

En esta actividad trabajamos con una bitácora que contiene muchos registros de accesos a una red. Al inicio puede parecer solo un archivo grande con información, pero el objetivo era procesarlo de forma eficiente para poder obtener datos útiles, como identificar cuáles direcciones IP aparecen con mayor frecuencia. Para lograr esto, no bastaba con que el programa funcionara, también era importante que el diseño y las estructuras de datos fueran adecuadas.

Lo primero que hicimos fue leer el archivo completo y almacenar todos los registros en un vector. Durante esta etapa se separó la dirección IP del puerto y se convirtió la IP a un valor numérico. Esta conversión fue importante porque permite comparar correctamente las IPs al momento de ordenarlas, evitando errores que pueden ocurrir cuando se comparan como texto. Al hacer este proceso desde la lectura, se facilitó el trabajo en las siguientes etapas del programa.

Después, ordenamos la bitácora por IP utilizando Heap Sort. Elegimos este algoritmo porque es una opción eficiente cuando se trabaja con grandes cantidades de datos. Algoritmos simples como Bubble Sort o Insertion Sort se vuelven demasiado lentos cuando el número de registros crece. También consideramos otros algoritmos como Quick Sort y Merge Sort, pero Quick Sort puede tener un mal desempeño en ciertos casos y Merge Sort requiere memoria adicional. Heap Sort, en cambio, mantiene una complejidad de n log n en todos los casos y trabaja directamente sobre el vector, lo que lo hace una opción estable para este problema.

Una vez que la bitácora quedó ordenada, el conteo de accesos fue mucho más sencillo. Como todas las entradas de una misma IP quedaron agrupadas, solo fue necesario recorrer el vector una sola vez y contar cuántas veces se repetía cada IP. Este proceso fue eficiente y nos ayudó a entender cómo un buen ordenamiento puede reducir significativamente el costo de las operaciones posteriores.

Para obtener las diez IPs con mayor número de accesos utilizamos un Binary Heap, implementado mediante una priority queue. Esta estructura es especialmente útil cuando se trabaja con prioridades, ya que permite obtener rápidamente el elemento con mayor valor. Insertar elementos y extraer el de mayor prioridad es eficiente, lo que nos permitió obtener el top 10 sin necesidad de volver a ordenar toda la información.

Durante el desarrollo también entendimos por qué un Binary Heap es más adecuado que un árbol binario de búsqueda para este problema. Aunque un árbol binario puede ser útil para búsquedas, no está optimizado para extraer repetidamente el valor máximo, y su desempeño puede empeorar si no está balanceado. El heap, en cambio, mantiene siempre clara la prioridad de los elementos y se ajusta automáticamente, lo que lo hace más conveniente en este caso.

En lo personal, esta actividad me ayudó a entender mejor que las estructuras de datos no son solo conceptos teóricos, sino herramientas que realmente influyen en

cómo se comporta un programa cuando trabaja con muchos datos. Al implementar Heap Sort y un Binary Heap pude ver claramente cómo elegir bien el algoritmo y la estructura de datos hace que el proceso sea más eficiente y más claro de entender. También me quedó más claro cómo el ordenamiento puede facilitar otras operaciones, como el conteo de accesos, y cómo una estructura jerárquica como el heap es ideal cuando se necesita obtener los elementos más importantes. En general, esta actividad me permitió reforzar lo visto en clase y aplicarlo a un problema más cercano a una situación real.