Reflexión Actividad 1.3

1. Breve descripción sobre los algoritmos

Los algoritmos representan una base fundamental para las ciencias computacionales, las matemáticas y otras disciplinas relacionadas; en pocas palabras, conciben un conjunto ordenado y finito de operaciones o reglas para resolver un problema particular. Los algoritmos reciben un estado inicial o entrada y llevan a cabo instrucciones sistemáticas para llegar a un estado final o salida.

Cada código tiene por detrás un algoritmo que describe los pasos a seguir para el programa y es la labor del ingeniero de software el diseñar estos pasos acoplándose a las necesidades del problema que quiere solucionar.

2. Breve descripción de los Algoritmos de Ordenamiento

Como su nombre lo indica, los *algoritmos de ordenamiento* acomodan un conjunto de datos, normalmente números, en algún orden específico; en este caso, en orden ascendente. Existe una amplia variedad de estos y dependiendo de los requerimientos y conocimientos del programador, se deberá seleccionar uno de ellos. A continuación, se presentan algunos de los más populares.

2.1. **MergeSort**

Algoritmo de ordenamiento de nivel 2 (complejidad $O(nlog(n))$ que divide al array ingresado en dos y recursivamente parte a la mitad los dos subarrays generados. Una vez que los subarrays llegan a una longitud de 1, comienza un proceso de *merge* donde se crea un nuevo array con los elementos en orden ascendente de los subarrays. El proceso termina cuando el array generado por el proceso de *merge* tiene la misma longitud que el array inicial.

El proceso de división tiene una complejidad de $O(log(n))$ mientras que las acciones de *merge* tienen una complejidad de $O(n)$. De esta forma, la complejidad total del algoritmo de MergeSort es $O(nlog(n))$.

2.2. **Quicksort**

Nuevamente, un algoritmo de nivel 2. Personalmente, es mi favorito y es el que **fue utilizado en nuestra actividad integradora**. Utiliza un elemento del array como un pivote y recorre los demás elementos para ordenarlos del lado izquierdo, si son menores, o del lado derecho,

si son mayores. Para el final de este proceso, el pivote se encontrará en su posición correcta y se repetirán estos pasos recursivamente para los dos *subarrays* generados hasta llegar a un subarray de longitud 1.

Al igual que MergeSort, QuickSort tiene una complejidad promedio de $O(nlog(n))$, aunque en la práctica, puede llegar a ser más rápido.

2.3. **BubbleSort**

Recibe su nombre debido a la manera en que los elementos se mandan al final de la lista. Consiste en un algoritmo que recorre el array repetidas veces y compara elementos adyacentes, si no se encuentran en su orden correcto, los intercambia.

Debido a su simplicidad, es muy popular como uno de los primeros algoritmos para aprender, sin embargo, en términos de *performance*, BubbleSort realiza, en promedio, n^2comparaciones e intercambios; por lo cual su eficacia se compromete cuando el input es muy grande.

2.4. **InsertionSort**

Recorre el array un elemento a la vez y busca, en los números anteriores a este, su posición correcta para colocarlo. Al igual que BubbleSort, realiza n^2comparaciones e intercambios; no obstante, en la práctica puede llegar a ser mejor que el algoritmo básico de BubbleSort.

3. Breve descripción de los Algoritmos de Búsqueda

Estos algoritmos nos permiten encontrar un valor específico en un conjunto de datos. Su importancia es amplia en diversas disciplinas y pueden presentarse en situaciones tan cotidianas como buscar una palabra en un diccionario. Popularmente se conocen dos algoritmos, mismos que fueron usados para esta actividad.

3.1. **BinarySearch**

Este algoritmo es el más eficaz cuando se busca un elemento entre un array previamente ordenado. Consiste en la división recursiva de un intervalo de búsqueda hasta encontrar la ubicación del elemento (intervalo de búsqueda de longitud 1). Se selecciona el elemento en la mitad del array, se compara con el elemento a buscar y si este es menor se reduce el intervalo

de búsqueda al subarray a izquierda y se repite el proceso; en caso contrario, se toma el intervalo de la derecha.

3.2. **SequentialSearch**

También conocido como *LinearSearch*, recorre todos los elementos de un array desde la izquierda hasta la derecha hasta encontrar la posición del elemento de búsqueda. Una vez que encuentra a este elemento en la lista, retorna el índice que indica su posición. Debido a que recorre todo el array en un ciclo, se considera que la complejidad promedio de *SequentialSearch* es O(n).

4. Reflexión

Los algoritmos de búsqueda y ordenamiento, conforman herramientas de gran utilidad para uno de los procesos principales de todo software: la manipulación de datos y estructuras de datos. Todo problema, todo programa comienza con una entrada que el código manipulará. En nuestro caso, la entrada fueron 16,807 líneas de errores que fueron procesadas gracias a una clase Falla que además de guardar el error, guardaba su fecha con un número entero. Es a partir de este último que pudimos implementar una adaptación de QuickSort y otra de BinarySearch para ordenar y encontrar valores, respectivamente.

Ahora bien, a título personal, considero que el análisis de este tipo de algoritmos trae varias ventajas para alumnos como nosotros en esta etapa de la carrera. Si bien es cierto que sería fácil implementar alguna función incluida en nuestro lenguaje para ordenar un array en solo una línea, el haber escrito los algoritmos para *MergeSort* y *QuickSort* me permitió conocer más sobre el lenguaje de C++ (el comportamiento de apuntadores y estructuras como array dinámicos) así como también creó en mí la noción del buen diseño de algoritmos, pues ahora entiendo que no solo es necesario crear un programa que resuelva un problema sino que también la solución tiene que ser eficaz en términos de accuracy y performance.

Más adelante, creo que este tipo de conocimiento nos ayudará a entender un poco más sobre cómo diseñar un proyecto con soluciones escalables. Incluso con esta actividad, recordé los inicios de Twitter, cuando frecuentemente se caían sus servidores debido a la demanda; ahora intuyo que esto pudo haber ocurrido porque la app no se pensó originalmente para las masas y quizás los algoritmos seleccionados no eran los más óptimos para atender a un número tan grande de usuarios. Quién sabe, quizá utilizaron un *BubbleSort* en lugar de un *QuickSort* o *MergeSort* para acomodar el timeline...