Jorge Daniel Cruz Case – A01634536

Reflexión Actividad integradora 1.3

Durante la realización de la actividad integradora se utilizaron diversos algoritmos de búsqueda y ordenamiento, así como distintos algoritmos misceláneos para otras actividades. Para esto fue importante conocer la importancia de utilizar este tipo de algoritmos.

Para este tipo de programas dónde se debe utilizar una gran cantidad de datos (16807 en este caso), es importante que el programa corra lo más rápido posible, especialmente si es de uso común o empresarial. Por lo cual que la complejidad temporal del programa o de los algoritmos sea lo más baja posible es preferible. Por esto es importante explorar diversos algoritmos que te permitirán hacer esto.

Para el programa que realice probe 3 algoritmos de ordenamiento diferentes, *Quick Sort, Merge Sort, y Bubble sort.* Para decidirme cual utilizar, investigue en que consistían cada uno.

El *Bubble sort*, fue uno de los vistos en clase, "Este consiste en iterativamente hacer una comparación entre 2 elementos o datos adyacentes entre sí" (Programiz. 2021). si se realiza la comparación y concluye que el primer dato es mayor al segundo los intercambia de lugar y prosigue al siguiente set de datos, de lo contrario el intercambio no se realiza y solamente prosigue al siguiente. Este proceso se realiza *n* veces, en este caso se utilizan i & j para el algoritmo. Por esto se puede decir que su complejidad temporal es de $O(n^2)$, siendo todos sus posibles casos iguales en complejidad.

Por otro lado, el *Merge Sort,* Consiste en particiones que se le hacen al arreglo dado, partiéndolo en mitades y dónde al final termina en pares ordenados, los cuales combina al final en un semi arreglo ordenado. En el algoritmo implementado esto se hace recursivamente. El algoritmo Merge tiene 2 partes, Merge y ordena Merge. Si resolvemos la relación de recurrencia del algoritmo terminamos con $O(n \log n)$, esto para todos sus posibles casos.

Por último, tenemos el *Quick sort, "*Este es un algoritmo secuencial, el cual consiste en la utilización de pivotes en el arreglo, dónde todo al lado derecho del pivote es mayor que este, y todo del lado izquierdo es menor". (Cidecame, s.f.). Al añadir más y más pivotes el arreglo se empieza a ordenar de esta manera. En el algoritmo implementado esto se realiza recursivamente

similar a lo hecho en el *Merge Sort.* Cuando resolvemos la relación de recurrencia obtenemos lo siguiente: $O(n \log n)$ para los casos: mejor y promedio. Para el peor caso obtenemos: $O(n^2)$.

Teniendo todo esto en cuenta terminamos con los siguientes datos:

Algoritmo	Mejor	Promedio	Peor
Bubble Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$

Si sustituimos la cantidad de datos en la *n* obtenemos lo siguiente:

n = 16807 datos

Algoritmo	Mejor	Promedio	Peor
Bubble Sort	$O(282'475{,}249)$	$O(282'475{,}249)$	$O(282'475{,}249)$
Merge Sort	$O(71{,}019)$	$O(71{,}019)$	$O(71{,}019)$
Quick Sort	$O(71{,}019)$	$O(71{,}019)$	$O(282'475{,}249)$

Teniendo en cuenta estos resultados deseche la idea de utilizar el Bubble sort completamente, esto ya que su complejidad temporal no es apta y el número de iteraciones realizadas es muy alta sin importar del estado del arreglo.

Esto me dejo con 2, *Quick & Merge Sort*, para decidirme entre estos 2 decidí utilizar las funciones de la librería *chrono* que nos pasó el profesor. Con esta pude comparar el tiempo que tomada en correr cada uno de estos algoritmos. En condiciones similares de uso de CPU y de memoria el *Quick sort* tardo en promedio 252,755 ms o 4.2 minutos en correrse. Por otro lado, el *Merge Sort* tardaba en promedio 259,036 ms. Teniendo en cuenta que los tiempos pueden variar con la potencia de la computadora y que el tiempo de llenado de los datos al arreglo es de en promedio 170,000 ms, decidí quedarme con el Quick sort.

Además de los algoritmos de ordenamiento, se implementó un algoritmo de búsqueda para encontrar la posición de los datos que meta el usuario dentro del arreglo. Para este ni tuve que elegir ya que el profesor nos pidió que utilizáramos el *Binary Search.*

Para entender este decisión solamente hay que ver como funcionan los algoritmos de búsqueda. Aquellos vistos en clase fueron, *Sequential search & Binary search*.

El *Sequential search* consiste en buscar diagonalmente el dato que necesitas. Si no encuentra el dato requerido en la diagonal, se pasa al siguiente dato y busca igual diagonalmente. Si obtenemos su complejidad temporal está nos da, $O(n)$.

Por otro lado, el *Binary search*, consiste en buscar un dato en el medio del arreglo, si el dato del medio es menor al buscado se elimina la parte izquierda. Este proceso se repite n veces. Por la naturaleza del algoritmo este solamente funciona con arreglos ordenados. Si obtenemos su complejidad temporal obtenemos, $O(log_2\ n)$.

Si juntamos sus casos en una tabla obtenemos lo siguiente:

Algoritmo	Mejor	Promedio	Peor
Sequential search	$O(1)$	$O(n)$	$O(n)$
Binary search	$O(1)$	$O(log_2\ n)$.	$O(\log n)$

Sin necesidad de sustituir nada nos podemos dar cuenta que el *Binary Search* es más eficiente para arreglos con una cantidad de datos muy alta, debido a su naturaleza y complejidad. Por lo cual es lógico que se utilizara Binary search para la búsqueda de datos en arreglos muy extensos.

Al final, entiendo la importancia de utilizar algoritmos con complejidad menor a otros, ya que es importante tener un código rápido y optimizado, especialmente si es un código mucho más extenso o se usa para actividades del día a día. Es muy diferente tener un código que corra en 60 ms a uno que corra en 60,000 ms. La gente no es muy paciente estos días, por lo cual tener un código rápido es esencial.

Referencias

Hacker Rank. (27 de septiembre de 2016). *Algorithms: Binary Search.* [Video]. YouTube. Recuperado de: https://www.youtube.com/watch?v=P3YID7liBug

InterviewBit. (17 de junio de 2019). *Merge Sort Algorithm.* [Video]. Youtube. Recuperado de: https://www.youtube.com/watch?v=uOjJPtVA24k&t=75s

Márquez M. (s.f.) *Unidad 2, Arreglos, Método Quick Sort.* Cidecame. Recuperado de: http://cidecame.uaeh.edu.mx/lcc/mapa/PROYECTO/libro9/mtodo_quick_sort.html

Programiz. (2021). *Bubble Sort (with Code).* Recuperado de: https://www.programiz.com/dsa/bubble-sort