CAMPUS QUERÉTARO

TC1031. PROGRAMACIÓN DE ESTRUCTURAS DE DATOS Y ALGORITMOS FUNDAMENTALES

(GRUPO 826)

"Actividad Integradora 1. Conceptos básicos y algoritmos fundamentales "

Evidencia presentada por la estudiante:

A01706095 - Naomi Estefanía Nieto Vega

Profesor:

Dr. Eduardo Arturo Rodríguez Tello

Fecha de entrega:

Lunes 22 de marzo de 2021

Actividad Integradora. Conceptos básicos y algoritmos fundamentales

A01706095 - Naomi Estefanía Nieto Vega

***Resumen*—En esta reflexión se hablará acerca de la importancia de los algoritmos de ordenamiento y búsqueda en el área computacional, asimismo se verá el análisis de la complejidad y ventajas del algoritmo QuickSort como algoritmo seleccionado para la soluación de la situación problema.**

***Index Terms*—algorítmos de ordenamiento, quick sort, algoritmos de búsqueda, complejidad algorítmica**

I. INTRODUCCIÓN

Hoy en día los algoritmos de ordenamiento son muy importantes en el área computacional y en muchas más áreas en los que son utilizados. De acuerdo con la literatura un algoritmo es una serie o secuencia ordenada de pasos que dan solución a un determinado problema. Entonces, pueden existir distintos algoritmos que den solución a un mismo problema y es aquí donde entra la importancia de utilizar el más eficiente en cuanto a tiempo y recursos ya que estos deben ser utilizados correctamente para que de esta forma la computadora pueda darnos resultados muy buenos.

Asimismo, tener distintas opciones de algoritmos de ordenamiento nos da la posibilidad de mejorar la eficiencia de algún programa, utilizando el algoritmo que mejor solucione el problema y por consiguiente mejorar la eficiencia de nuestro programa al compilarlo, además de una adecuada utilización de los recursos ya que entre menos recursos utilice será mejor y más barata su implementación hablando económicamente en el impacto que tienen cuando desarrollamos software para una empresa. (Ver figura 1)

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	θ(n log(n))	O(n log(n))	O(n)

Figura 1. Complejidad espacial y de tiempo de algunos algoritmos de ordenamiento. [1]

Otro aspecto importante son los algoritmos de búsqueda ya que gracias a estos algoritmos es que podemos buscar datos sin tener que ir buscando uno por uno individualmente, tal vez esto parezca insignificante al utilizar una base de datos pequeña, pero cuando tenemos bases de datos con grandes volúmenes de datos esto se volvería algo tedioso, es por ello que su importancia radica en que nos facilitan la búsqueda de datos y además de una forma eficiente.

Así, al conocer la importancia y eficiencia del uso de estos algoritmos podemos desarrollar opciones viables en cuanto a tiempo y eficacia para búsquedas y ordenamientos de bases de datos con archivos muy grandes, como la de la situación problema.

II. DESARROLLO

Para los algoritmos de búsqueda tenemos dos opciones, la búsqueda secuencial que en su mejor caso cuenta con una complejidad de $C(n) = 1$ si el elemento buscado es el primero de la lista, y en su peor caso cuenta con una complejidad de $C(n) = n$ si el elemento buscado no existe o se encuentra en la última posición. La segunda opción es la búsqueda binaria, cabe aclarar que ambos métodos realizan lo mismo pero con un algoritmo distinto, en este caso utilicé la búsqueda binaria en un arreglo ya ordenado que cuenta con una complejidad de $O(\log n)$ para su peor caso y un $O(1)$ en su mejor caso.

A continuación podemos observar una tabla con algunos de los algoritmos de ordenamiento:

Algoritmo	Mejor	Promedio	Peor	Estable	Espacial
SwapSort	$O(n^2)$	$O(n^2)$	$O(n^2)$	Sí	$O(1)$
SelectionSort	$O(n^2)$	$O(n^2)$	$O(n^2)$	No	$O(1)$
BubbleSort	$O(n)$	$O(n^2)$	$O(n^2)$	Sí	$O(1)$
InsertionSort	$O(n)$	$O(n^2)$	$O(n^2)$	Sí	$O(1)$
MergeSort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	Sí	$O(n)$
QuickSort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	Sí	$O(\log n)$

Figura 2. Complejidades de los algoritmos de ordenamiento vistos a lo largo de estas cinco semanas.

Como podemos observar en la figura 2, tenemos que los algoritmos de SwapSort y SelectionSort son muy similares en cuanto a su complejidad espacial y temporal, a excepción de la estabilidad, lo que significa que por ejemplo el algoritmo de SelectionSort si llegaran a existir dos objetos con la misma clave después del ordenamiento no se respetaría el mismo orden, a diferencia del SwapSort. Dado que para ambos su mejor complejidad es de $O(n^2)$ no los consideraría una opción viable para los requerimientos de esta actividad.

Posteriormente tenemos a BubbleSort e InsertionSort que son muy similares en cuanto a complejidad, pero no consideré que fueran lo suficientemente aptos para este tipo de búsqueda tratándose de archivos muy grandes. Después tenemos a MergeSort y QuickSort, que a simple vista parecería mejor MergeSort que QuickSort debido a su complejidad. Sin embargo, para el desarrollo de esta primera actividad integradora utilicé el algoritmo de ordenamiento "QuickSort" y a continuación explicaré los motivos por los cuales consideré a este algoritmo como la mejor opción de los algoritmos de ordenamiento vistos:

1. Es un algoritmo basado en la técnica de divide y vencerás por lo que funciona eligiendo un pivote para particionar el arreglo con base en esto y así ir ordenando los elementos.
2. No necesita espacio auxiliar de almacenamiento ya que es un algoritmo de ordenamiento in situ, esto quiere decir que no necesita espacio de memoria auxiliar para hacer el ordenamiento, a diferencia de "MergeSort" por ejemplo, en el que tenemos que crear dos arreglos temporales para fusionar las mitades.
3. Es un buen algoritmo de ordenamiento cuando estamos tratando con arreglos o vectores aleatorios, ya que a pesar de tener una complejidad de $O(n^2)$ en su peor caso, existe una probabilidad alta de que con el pivote correcto se vuelva muy eficiente, siendo su mejor caso un $O(n \log n)$. (Ver figura 3)
4. Es un algoritmo que es estable, esto quiere decir que si tenemos dos objetos con la misma clave aparecerán en el mismo orden en la salida arrojada.

De acuerdo con los motivos señalados anteriormente es que decidí implementar este algoritmo, además de que al cronometrar el tiempo que dicho algoritmo tarda en realizar el ordenamiento de los datos depende mucho del pivote como mencioné anteriormente, por lo que al tener un buen pivote el tiempo de ejecución se reduce significativamente. Su rango de tiempo al compilarlo osciló entre los 7 a 24 milisegundos en su peor caso (basándome en las pruebas al compilar en consola). Este tiempo es muy variable y no es definitivo. Como podemos observar en la figura 3, este algoritmo tiene una complejidad promedio de $O(n \log n)$, esto quiere decir que los datos no siguen algún patrón que aporte ventajas o desventajas, por lo que se considera la situación típica de ejecución.

Mejor	Promedio	Peor	Estable	Complejidad espacial
$O(n \log n)$	$O(n \log n)$	$O(n^2)$	Sí	$O(\log n)$

Figura 3. Complejidad temporal y espacial del algoritmo de ordenamiento QuickSort

Ahora procederemos a ver el análisis de la complejidad del algoritmo QuickSort. La complejidad de este algoritmo de ordenamiento podemos expresarla de la siguiente manera:

$$T(n) = T(k) + T(n-k-1) + O(n) \tag{1}$$

Donde:

- k es el número de elementos menores al pivote.

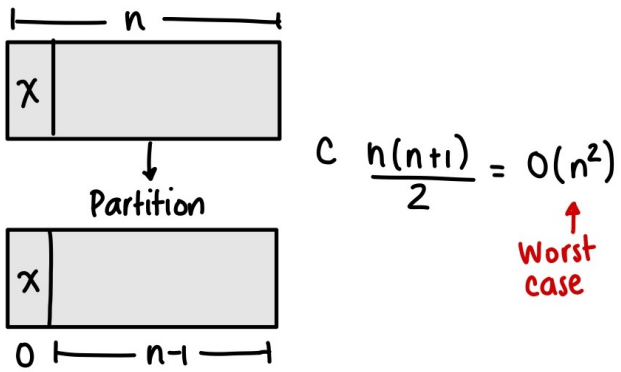

Figura 4. Ilustración del peor caso del QuickSort

Entonces para el análisis de su peor caso, sería cuando tenemos un arreglo ya ordenado, ya que nuestra x estaría hasta la izquierda como se muestra en la figura 4. Y al dividir las ramas tendríamos constantes que van como $C\,[n + (n-1) + (n-2) + ..,1]$, por lo que después esto pasaría a ser la suma de Gauss definida por $\left(\frac{n(n+1)}{2}\right)$ y la ecuación resultante de esto que nos da la complejidad quedaría como la ecuación (2).[5]

$$T(n) = O(n^2) \tag{2}$$

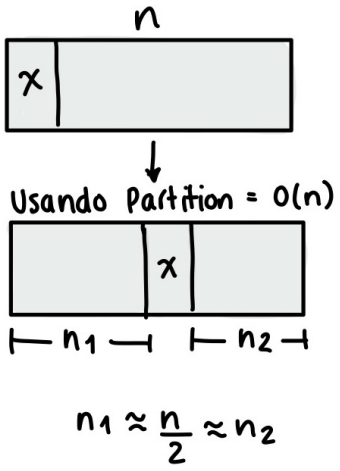

Figura 5. Ilustración del mejor caso de QuickSort

En cambio para su mejor caso tenemos un arreglo o vector de tamaño n, nuestro pivote en este caso es la x como se muestra en la figura 5, y si al momento de hacer la partición (que tiene una complejidad de $O(n)$), se divide en dos partes y entonces nuestras ecuaciones quedarían de la siguiente forma:[5]

$$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + Cn$$

$$T(n) = 2T\left(\frac{n}{2}\right) + Cn$$

$$T(n) = O(\log n) \tag{3}$$

Por lo que su complejidad espacial en el mejor caso sería igual a la ecuación (3).

III. CONCLUSIÓN

En conclusión, gracias a esta actividad pude comprender mejor las complejidades de los distintos algoritmos que existen, además de evaluarlos e implementar uno para brindar una solución específica que fuera de acuerdo con los requerimientos del reto. Asimismo pude comprender mejor el funcionamiento de estos algoritmos y su importancia en cuestión de recursos y tiempo ya que entre más eficiente sea un algoritmo dará mejores resultados. También comprendí que no existen algoritmos buenos o malos, sino que todos tienen distintas características que los hacen mejores para algunas situaciones particulares, por lo que conociendo esta variedad podemos elegir el que mejor se adapte a nuestras necesidades.

Además, al evaluar las distintas opciones de solución que existían me ayudó mucho conocer la complejidad algorítmica temporal y espacial de cada uno para implementar la que fuera mejor. Considerando también el análisis matemático que existe detrás de estos algoritmos para determinar sus complejidades y cómo esto se ve reflejado en la implementación dependiendo si es el mejor o el peor caso.

REFERENCIAS

[1] Big-O Algorithm Complexity Cheat Sheet (Know Thy Complexities!) @ericdrowell. (2020). Big O Algorithm Complexities. https://www.bigocheatsheet.com/

[2] GeeksforGeeks. (2019, 30 septiembre). Complexity Analysis of Binary Search. https://www.geeksforgeeks.org/complexity-analysis-of-binary-search/

[3] GeeksforGeeks. (2021, 27 febrero). QuickSort. https://www.geeksforgeeks.org/quick-sort/

[4] GeeksforGeeks. (2019a, septiembre 26). Stability in sorting algorithms. https://www.geeksforgeeks.org/stability-in-sorting-algorithms/

[5] Time complexity:Best and Worst cases|Quick Sort|Appliedcourse.(2019, 18 mayo).[Vídeo].YouTube. https://bit.ly/3c9VzPr

[6] GeeksforGeeks. (2021a, enero 4). Why quicksort is better than mergesort ? https://www.geeksforgeeks.org/quicksort-better-mergesort/