Reflexión: Actividad integradora #1

Miguel Arriaga Velasco – A01028570

Campus Santa Fe

Programación de estructuras de datos y algoritmos fundamentales

22 de marzo del 2021

1. Algoritmos de ordenamiento

Muchas veces se tienen estructuras de datos y se quieren realizar ciertas operaciones con estos datos. Sin embargo, en la mayoría de los casos es más fácil y eficiente manipular los datos y realizarles cambios si se encuentran de manera ordenada. Aquí es donde entra el rol de los algoritmos de ordenamiento.

Existen diversos algoritmos que nos permiten ordenar una estructura, algunos de los más populares son los siguientes:

- Swap Sort
- Bubble Sort
- Selection Sort
- Insertion Sort
- Merge Sort
- Quick Sort

Todavía no existe un algoritmo de ordenamiento que sea el más eficiente en todos los casos, por lo que se debe de analizar la naturaleza del problema antes de seleccionar un algoritmo. En este caso, la elección se redujo a Quick Sort vs Merge Sort. Esto se debe a que son los únicos algoritmos de esta lista que tienen una complejidad temporal promedio de O(n logn), ver figura 1.

Tabla: Complejidad de los algoritmos de ordenamiento.

Algoritmo	Mejor	Promedio	Peor	Estable	Espacio
Swap sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	Sí	$O(1)$
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	No	$O(1)$
Buble sort	$O(n)$	$O(n^2)$	$O(n^2)$	Sí	$O(1)$
Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$	Sí	$O(1)$
Merge sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	Sí	$O(n)$
Quick sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	Sí	$O(\log n)$

Figura 1. Complejidad de diversos algoritmos de ordenamiento

En este caso, se requiere ordenar una bitácora con una gran cantidad de registros (casi 17,000). Adicionalmente, el espacio de memoria no es una gran preocupación, ya que la única estructura de datos de todo el programa, es un vector de registros. Entonces… ¿Qué algoritmo usamos?

1.1. Quick Sort

Según (GeeksForGeeks, 2019), Quick Sort es un algoritmo que utiliza el enfoque “Divide y vencerás”, que ordena los datos de manera interna (utiliza únicamente memoria interna). Su fundamento es localizar un punto “pivote”, en torno al que se dividirá el arreglo de manera repetida, colocando los elementos menores a pivote de un lado y los mayores del otro, ver figura 2.

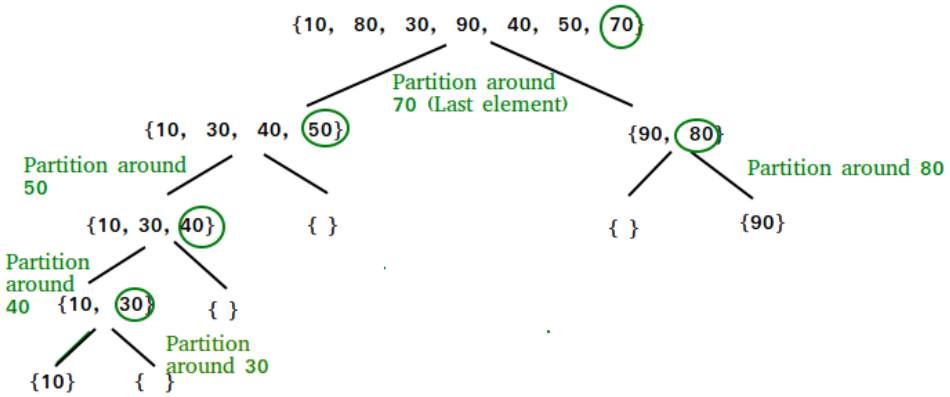

Figura 2. Representación gráfica del algoritmo Quick Sort

1.2. Merge Sort

Por el otro lado, Merge Sort es un algoritmo externo (guarda información en memoria externa), que también utiliza el enfoque "Divide y vencerás". Consiste en dividir el arreglo en sub-arreglos de tamaño n/2 hasta que solo quede un elemento. Posteriormente vuelve a juntar los arreglos (merge), ver figura 3 (GeeksForGeeks, 2019).

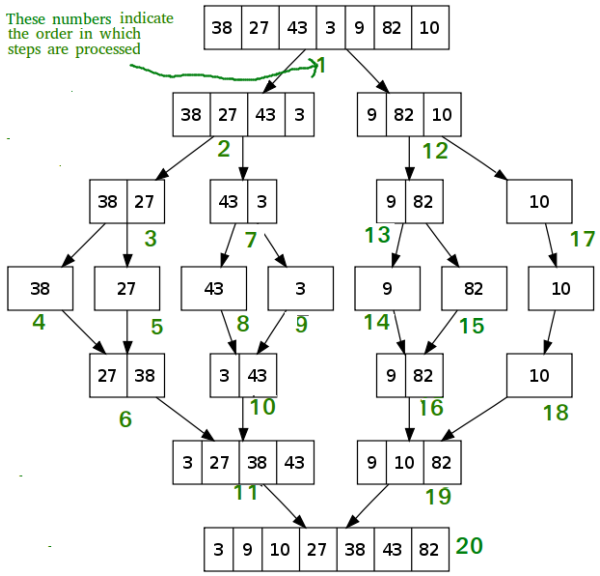

Figura 3. Representación gráfica del algoritmo Merge Sort

1.3. Quick Sort vs. Merge Sort

GeeksForGeeks (2019) indica que la peor complejidad de Quick Sort es de $O(n^2)$, mientras que la de Merge Sort es de O(n logn) en todos los casos. Por ende, Merge Sort tiende a tener mejores tiempos en estructuras con una enorme cantidad de datos, mientras que Quick Sort es el mejor algoritmo en estructuras con menor cantidad de datos. Como ya se mencionó, Merge Sort requiere de espacio de memoria adicional, mientras que Quick Sort hace el ordenamiento "in-place", por lo que no utiliza memoria adicional. GeeksForGeeks también menciona que Quick Sort es preferido para arreglos, mientras que Merge Sort se recomienda para listas ligadas.

Ya que ambos algoritmos tienen sus ventajas y desventajas, se decidió implementar ambos para el ordenamiento de la bitácora y comparar su rendimiento, tomando el tiempo promedio de cada uno después de 10 ordenamientos, ver figura 4. Para tomar el tiempo se utilizó la librería <chrono> de C++.

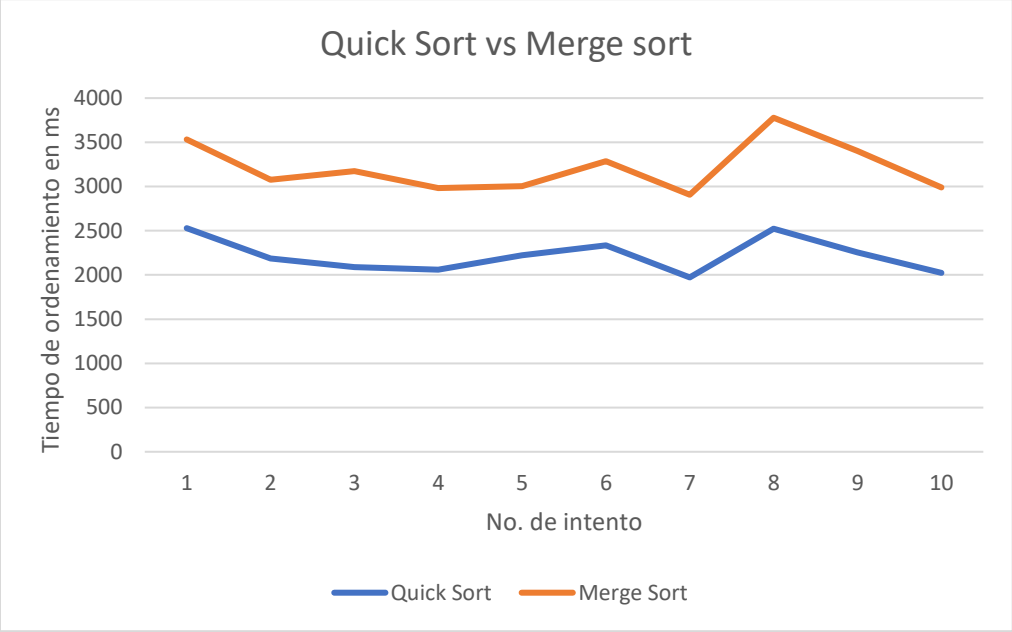

Figura 4. Tiempo de ejecución del ordenamiento de la bitácora en ms de Quick Sort y Merge Sort

Como se puede observar, Quick Sort realizó un ordenamiento de la bitácora mucho más rápido que Merge Sort en mi computadora personal (Laptop Dell Latitude 740, Procesador Intel i5, Windows). En repl.it se obtuvieron tiempos mayores pero la tendencia continuó favoreciendo a Quick Sort (ver figura 5). Estos resultados se podrían deber a que la cantidad de datos no es lo suficientemente grande para tener un impacto negativo en su rendimiento. "Quicksort exhibe una buena ubicación de caché y esto hace que el Quicksort sea más rápido que Mergesort" (GeeksForGeeks, 2019). Por lo anterior mencionado, se seleccionó a Quick Sort como el algoritmo de ordenamiento indicado para la bitácora. Sin embargo, también se incluyó el método de Merge Sort para que se puedan realizar estas pruebas.

```
QUICK SORT:
Tiempo de ejecución en ms: 9205

MERGE SORT:
Tiempo de ejecución en ms: 12797
```

Figura 5. Tiempos de una corrida en repl.it

1.4. *Análisis de complejidad de Quick Sort*

Para analizar la complejidad temporal de Quick Sort, primero debemos de obtener la complejidad temporal del algoritmo "partition", que es el que divide la estructura de datos en dos. El código para este algoritmo es el siguiente (figura 6).

```
int Bitacora::partition(int low, int high){
    Registro pivot=registros[high];
    int i=low-1;
    for (int j = low; j <= high-1; j++)
    {
        //quickComparaciones++;
        if (registros[j].getFecha()<pivot.getFecha()){
            i++;
            intercambiar(i,j);
        }
    }
    intercambiar(i+1,high);
    return (i+1);
}
```

Figura 6. Algoritmo “partition”

Como se puede observar en el código, la operación más básica es un intercambio, que corre en tiempo constante. Sin embargo, esa operación se debe de realizar un número de veces determinado por un loop for. Este loop corre desde j(low) hasta high-1, en dónde low es el primer número del arreglo con el que se está trabajando (ya sea el arreglo original o una parte de este) y high es el mayor número de dicho arreglo. También se puede observar que el parámetro de control se incrementa en un valor constante de 1. Por esto, y siguiendo las estrategias para determinar la complejidad de un algoritmo iterativo de manera empírica, podemos determinar que la complejidad del algoritmo “partition” es de O(n).

Ahora debemos de analizar el algoritmo Quick Sort (figura 7), aquí es donde la cosa se pone algo complicada.

```
void Bitacora::quickSort(int low, int high){
    if (low<high){
        int pi=partition(low,high);
        quickSort(low,pi-1);
        quickSort(pi+1,high);
    }
}
```

Figura 7. Algoritmo “Quick Sort”

Primero obtenemos la fórmula recursiva del algoritmo, la cual es:

$$T(n) = T(k) + T(n-k-1) + O(n)$$

En donde los primeros dos términos se deben a las dos llamadas recursivas (para la primera y segunda mitades del arreglo), en las que k es el número elementos menores que pivote (pi). El tercer término es la complejidad del método “partition” que ya analizamos (es llamado una vez por cada llamada de la función Quick Sort).

Peor caso:

El peor caso ocurre cuando siempre se selecciona al pivote como el mayor o el menor elemento de arreglo. De esta forma no se logra dividir el arreglo en dos y en la fórmula recursiva, k equivale a cero, ésta se reduce a:

$$T(n) = T(0) + T(n-1) + O(n)$$

$$T(n) = T(n-1) + O(n)$$

En dónde la constante -1 se desprecia, dándonos una complejidad de: $O(n^2)$

Mejor caso:

Ocurre cuando pivote siempre es el punto medio del arreglo, esto nos deja con la ecuación:

$$T(n) = 2T(n/2) + O(n)$$

Que se puede resolver utilizando el caso dos del Teorema Maestro, como se muestra a continuación:

$$T(n) = aT(n/b) + f(n)$$

En nuestro caso, a=2, b=2 y f(n)=O(n).

El caso dos del Teorema nos dice que:

$$2.\ Si\ f(n) = O(n^{\log_b a}),\ entonces\ T(n) = \Theta(n^{\log_b a} \log_2 n)$$

Entonces sustituimos y comprobamos:

$$O(n) = O\left(n^{log_2 2}\right)$$

$$O(n) = O(n^1)$$

Obteniendo una complejidad de:

$$O\left(n^{log_2 2} log_2 n\right)$$

$$O(nlog_2 n)$$

Caso promedio:

Las matemáticas para evaluar el caso promedio son complicadas entonces no nos meteremos en eso (ver CSSE 230 "Quick Sort algorithm", https://www.rose-hulman.edu/class/cs/csse230/201710/Slides/23-Quicksort.pdf), pero se concluye que es estadísticamente muy improbable que se ocurra el peor caso, por lo que el caso promedio es:

$$O(nlog_2 n)$$

2. Algoritmos de búsqueda

Los algoritmos de búsqueda están diseñados para localizar un elemento con ciertas propiedades dentro de una estructura de datos.

Algunos ejemplos:

- Ubicar el perfil de Facebook correspondiente a cierta persona en la base de datos de la aplicación.

- Localizar un libro de ADA dentro de la biblioteca digital del Tec.

- Encontrar un producto en Amazon.

Los algoritmos de búsqueda reciben una llave (key) para realizar la búsqueda de ésta dentro de una estructura de datos. Los más conocidos son la búsqueda secuencial y la búsqueda binaria.

La búsqueda secuencial va recorriendo la estructura de datos elemento por elemento hasta encontrar la llave. Por el otro lado, la búsqueda binaria divide el problema en dos subsecuentemente hasta encontrar el elemento deseado. La búsqueda secuencial tiene una complejidad de O(n), mientras que la búsqueda binaria tiene una complejidad de O(logn). Por lo tanto, la búsqueda binaria es mucho más rápida que la secuencial. La única ventaja que tiene la búsqueda secuencial es que la estructura de datos no debe de estar ordenada para que funcione, mientras que en la binaria sí. Sin embargo, en nuestro caso también estamos interesados en ordenar la bitácora de registros, por lo que la búsqueda binaria es la opción clara.

2.1. *Análisis de complejidad de la búsqueda binaria*

Primer analicemos el algoritmo de búsqueda binaria (figura 8).

```
int Bitacora::binSearch(Fecha clave){
    //solo sirve para arreglos ordenados
    int n=registros.size();
    int left=0;
    int right=n-1;
    //int comparaciones=0;
    while(left<=right){
        int mid=int((left+right)/2);
        //comparaciones++;
        if (registros[mid].getFecha()>clave){
            right=mid-1;
        }
        else if(registros[mid].getFecha()<clave){
            left=mid+1;
        }
        else{
            return (mid);
        }
    }
    return -1;
}
```

Figura 8. Búsqueda binaria

Se observa que la operación más costosa que puede concluir de la decisión es una suma, lo cual cprre en tiempo constante. Se puede observar que el loop corre desde el inicio al final del array. Sin embargo, la variable de control se divide entre dos por cada iteración del loop, ya que este es un valor constante, se determina que el orden del algoritmo es de logaritmo base dos (constante entre la cual se divide la variable control).

$$O(log_2 n)$$

Ahora debemos de considerar que lo que se quiere es que el usuario introduzca dos fechas distintas y se impriman los registros entre ese rango de fechas. Por lo tanto, debemos de considerar que la búsqueda binaria se ejecuta dos veces y se requiere de un tiempo lineal para imprimir los registros dentro del rango. Esto se lleva acabo en el método rangoFechas (figura 9).

```
void Bitacora::rangoFechas(Fecha inicio, Fecha fin){
    int start=binSearch(inicio);
    int finish=binSearch(fin);
    if (start ==-1 ||finish==-1){
        std::cout<<"No se encontraron esas fechas"<<std::endl;
    }
    else{
        std::cout<<"Las fechas en ese rango son:"<<std::endl;
        for (start; start <= finish; start++)
        {
            std::cout<<registros[start].getDato()<<std::endl;
        }
    }
}
```

Figura 9. Método rangoFechas

La complejidad de este método es:

$$O(2log_2 n + O(k - m))$$

En dónde el primer método indica las dos llamadas de la búsqueda binaria y el segundo método indica la impresión de los registros entre los resultados de la búsqueda: la fecha mayor (k) y la fecha menor (m). En el peor caso, k es la última fecha del arreglo y m es la primera, por lo tanto, la complejidad sería de O(n).

Ya que O(n) crece más rápido que O(2log2n), la complejidad final de este método es de:

$$O(n)$$

3. Funcionamiento del programa

El programa cuenta con cuatro archivos de código, tres de los cuales son headers con las declaraciones y definiciones de clases y un archivo main.cpp. Se decidió realizar la declaración y definición de las clases en el mismo archivo por razones de conveniencia al momento de compilar el código.

Las tres clases que se implementaron fueron la clase “fecha”, que sirve para almacenar los datos de la fecha (mes, día, hora) y comparar dos o más fechas entre ellas. Esto se realizó sobrecargando varios operadores booleanos y sirve para poder ordenar y encontrar fechas de la bitácora. La segunda clase que se implementó es “Registro”, esta guarda la información de cada registro: la dirección ip, razón de falla, puerto y casa instancia de la clase “Registro”, también tiene una “Fecha”. La última clase es la clase “Bitacora”, en esta clase se almacena un vector de Registros y además contiene los métodos necesarios para ordenarlos y buscarlos. Adicionalmente, cuenta con un constructor que toma como parámetro el nombre de un archivo de texto, del cual lee los registros. También cuenta con otros métodos como printBitacora() o outputFile(), que imprimen el vector en consola o a un archivo de texto respectivamente. Además del método pideDatos(), que se mencionó anteriormente.

En el main se inicializan dos Bitacoras a partir del archivo de texto “bitacora.txt”, luego se realiza un ordenamiento a partir de quickSort y otro a partir de mergeSort, para comparar su rendimiento. Además, se llama al método outputFile, para crear archivos de texto con los datos ya ordenados.

Finalmente se realiza una búsqueda binaria y se llama al método pideUsuario(), para preguntarle al usuario una fecha y hora de entrada y salida y obtener los registros en ese rango.

Conclusión

Es un hecho que los algoritmos de ordenamiento y búsqueda son de gran importancia en la programación. La velocidad de procesamiento de las computadoras y la memoria de estas todavía es un recurso limitado y asegurarnos de que estos recursos sean administrados de la mejor manera posible es nuestra labor como ingenieros en tecnologías computacionales.

Consecuentemente, se debe de aspirar por los algoritmos más eficientes, tanto en uso de memoria como de tiempo de ejecución. Como se mencionó previamente, acomodar una estructura de datos brinda mucha utilidad para las funciones posteriores de un programa. Conocer y entender varios tipos de algoritmos de ordenamiento y en qué situaciones uno es mejor que otro es algo vital. Se logró analizar este comportamiento y determinar que Quick Sort era el mejor algoritmo en este caso.

Además, se logró parsear una base de datos y construir un programa que se adapte a la misma, cosa que se debe de hacer en caso de que un cliente brinde alguna base de datos. La habilidad de un ingeniero de investigar y buscar información para resolver problemas fue un factor clave para resolver esta actividad.

Referencias

GeeksForGeeks (2019) “Quick Sort vs. Merge Sort”. Recuperado desde: https://www.geeksforgeeks.org/quick-sort-vs-merge-sort/

GeeksForGeeks (2021) “QuickSort”. Recuperado desde: https://www.geeksforgeeks.org/quick-sort/

Presentaciones por el Dr. Eduardo Arturo Rodríguez Tello, durante la materia de Programación de estructuras de datos y algoritmos fundamentales