ANÁLISIS DE QUICKSORT – VARIANTE SECUENCIAL Y CONCURRENTE

Datos del autor: Ulises Justo Saucedo
Repositorio con el trabajo: ulises-justo-saucedo/UNLA-TP-Programacion-Concurrente (github.com)
Video explicativo: https://youtu.be/kzwCs--nqMc
Email: ulisesjustosaucedo@gmail.com

RESUMEN

En este informe se explicará el funcionamiento del algoritmo de ordenamiento QuickSort tanto en su variante secuencial como concurrente. La implementación concurrente fue hecha por mí, y es una manera simple y rápida de aplicar concurrencia a QuickSort. Por su simplicidad, no se puede obtener tanta eficiencia como podría ser con otras implementaciones, pero pese a ello con las tablas comparativas que se verán en este informe, se podrá ver como aun así la concurrencia acaba venciendo por mucho a la implementación secuencial. Aunque también se verá como esto último no ocurre siempre ya que, bajo ciertas condiciones, la implementación concurrente puede acabar perdiendo por mucho ante la secuencial.

Keywords: pivote, subarreglos, hilos, concurrencia, secuencial, eficiencia, tiempos, comparacion.

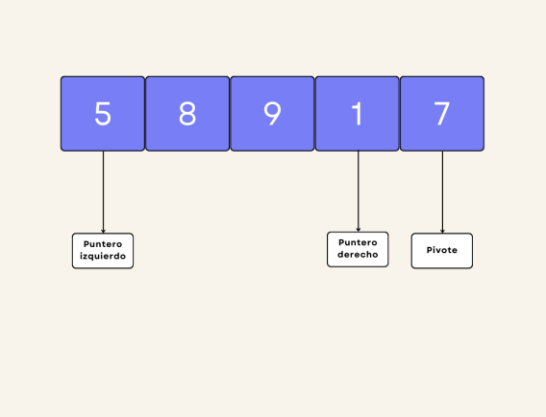
1. INTRODUCCIÓN

QuickSort es un algoritmo de ordenamiento que utiliza cuatro herramientas principales para realizar su tarea: un pivote, subarreglos, un puntero izquierdo y un puntero derecho.

Para realizar una explicación clara sobre el algoritmo, abordaré estas herramientas de la forma más concisa y clara posible.

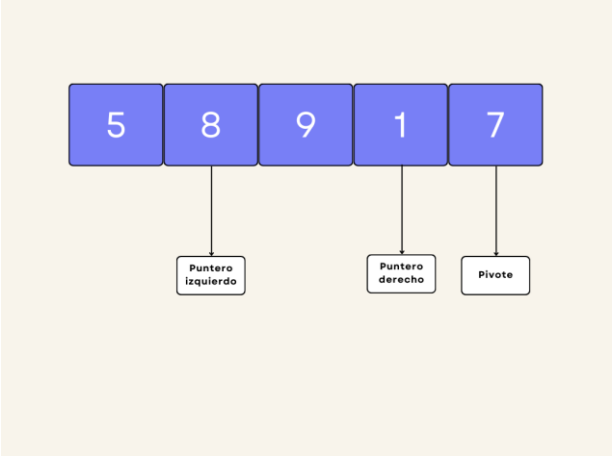
Por pivote refiere a un elemento cualquiera del arreglo en su estado inicial (desordenado), sobre el cual se realizarán las comparaciones necesarias para que, reordenando el arreglo, todos los elementos a su izquierda sean menores a él, y los que se encuentren a su derecha, mayores a él. Esto se podrá lograr gracias al uso de los dos punteros.

Intentemos ver esto que acabo de explicar de una manera más gráfica:



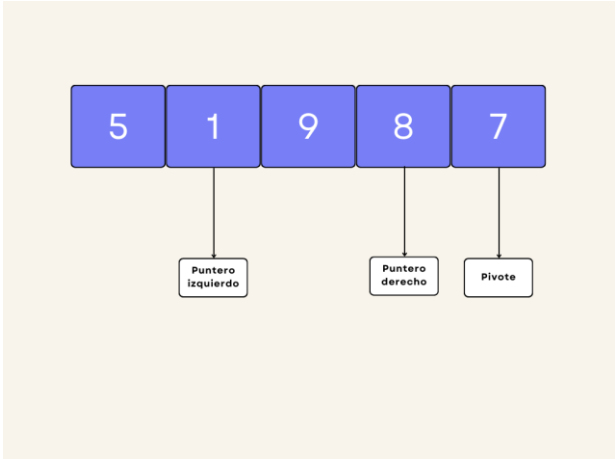
El pivote (en nuestro caso) es el último elemento del arreglo. El puntero izquierdo apunta a la primera posición del arreglo, y el puntero derecho apunta a la posición del pivote menos uno. Es decir, se ubica a la izquierda del pivote.

Para poner en funcionamiento al algoritmo, lo que se hace es que ambos punteros empiecen a "caminar" posición por posición hacia el otro; es decir, acercarlos poco a poco. Sin embargo, para hacer que caminen se deben cumplir ciertas condiciones: para empezar, únicamente se mueve al puntero izquierdo una y solo una posición por iteración. Por cada movimiento que haga, apuntará a distintos elementos del arreglo. Lo que se debe hacer es verificar si el elemento al que se encuentra apuntando es mayor que el valor de nuestro pivote; en caso de que el valor del elemento apuntado sea menor que nuestro pivote, no hacemos nada, seguimos moviendo al puntero izquierdo una posición a la derecha por iteración. Tal como se ilustra en el siguiente gráfico:

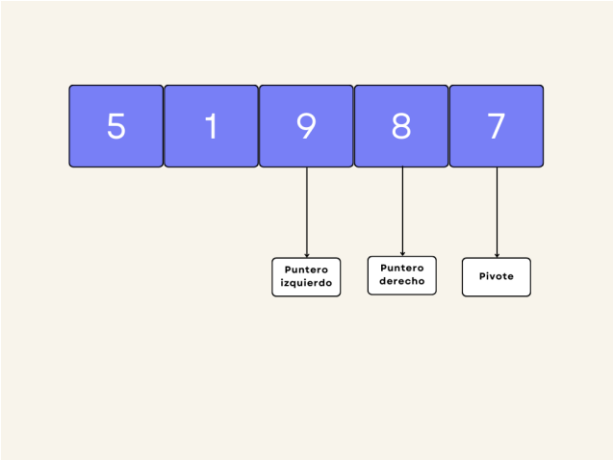


Debido a que 5 es menor que el valor de nuestro pivote (7) hacemos avanzar al puntero izquierdo. Ahora nos encontramos con que nuestro puntero izquierdo apunta a un elemento mayor a nuestro pivote. Es tal como mencionamos antes, así que ahora debemos detenernos y hacer retroceder al puntero derecho. El mismo debe retroceder hasta apuntar a un elemento menor a nuestro pivote.

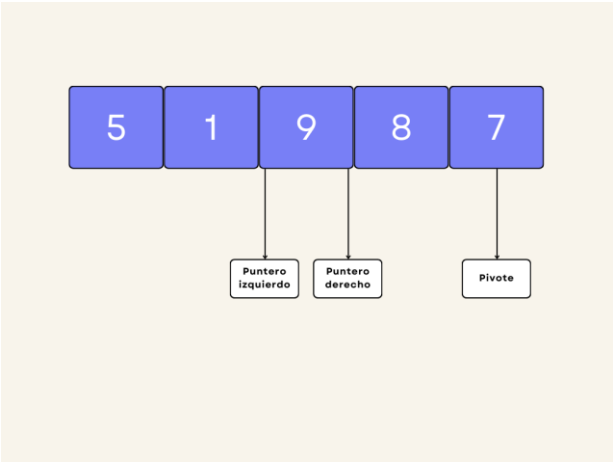
Como puede verse, nuestro puntero derecho ya está apuntando a un elemento menor a nuestro pivote. Es en este caso en que no debemos desplazar al puntero derecho hacia la izquierda, sino realizar un intercambio de valores. Dado que se cumplió la condición de que el puntero izquierdo apunte a un elemento mayor al pivote, y que el puntero derecho apunte a un elemento menor al pivote, intercambiamos los valores de ambos punteros. Es decir, el 8 pasa al lugar del 1, y el 1 al lugar del 8. Tal como se muestra en la siguiente ilustración.



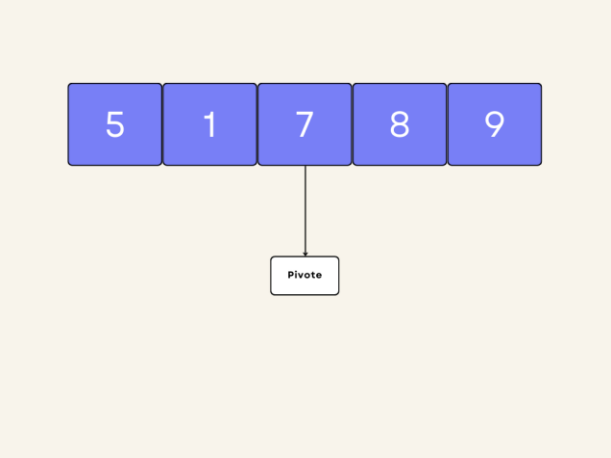
Para continuar desplazamos al puntero izquierdo una posición a la derecha, buscando nuevamente un elemento mayor a nuestro pivote.



En este caso el puntero izquierdo apunta al 9, un elemento mayor a nuestro pivote. Nos detenemos y pasamos al puntero derecho, en busca de un elemento menor al pivote. El 8 no es menor a 7, así que desplazamos el puntero derecho una posición a la izquierda.



Llegados a este punto, nuestros dos punteros se han conocido. Es decir, ambos, se encuentran apuntando a la misma posición del arreglo. Cuando esto ocurre debemos finalizar todas las comparaciones pasadas e intercambiar el valor del pivote por el de nuestros punteros. O sea, realizamos un último intercambio de valores entre nuestro pivote y puntero izquierdo (o derecho, la elección da igual en este punto).



Ahora, nuestro pivote se encuentra en su posición correcta en el arreglo. Todos los elementos a su izquierda son menores a él, y todos los que se encuentran a su derecha mayores a él. Para continuar con el algoritmo, QuickSort lo que hace es utilizar la recursión para volver a realizar todos los pasos que hicimos hasta ahora sobre los dos nuevos subarreglos que han aparecido.

{5, 1} es el primer subarreglo que aparece y {8, 9} es el segundo. Es decir, se consideran subarreglos aquellos elementos que hayan quedado a la izquierda y derecha del pivote.

Para ordenarlo completamente, simplemente se aplica QuickSort (los pasos que realizamos hasta ahora) sobre esos subarreglos, hasta llegar a un punto en que los subarreglos tengan un tamaño 1, es decir, solo quede un elemento. Es entonces cuando las llamadas recursivas finalizan y el retorno de todas ellas nos dejan nuestro arreglo inicial completamente ordenado.

2. IMPLEMENTACIÓN CONCURRENTE

Para realizar la implementación concurrente, utilicé dos hilos en la primera partición del arreglo inicial. Es decir, los dos primeros hilos se generan una vez generamos los dos primeros subarreglos con la primer llamada a QuickSort. Es en ese entonces donde, con ambos hilos instanciados, se les indica que deben aplicar QuickSort a sus correspondientes subarreglos; el primer hilo se encarga de aplicar el algoritmo sobre el subarreglo izquierdo, y el segundo hilo sobre el derecho. Tal como puede verse en el siguiente método.

```
public static void ordenar(int[] array) throws InterruptedException {
    int posicionFinalPivote = particionarArray(array, 0, array.length - 1);
    Thread hilo1 = new Thread(() -> {
        ordenar(array, 0, posicionFinalPivote - 1);
    });
    Thread hilo2 = new Thread(() -> {
        ordenar(array, posicionFinalPivote + 1, array.length - 1);
    });
    hilo1.start();
    hilo2.start();
    hilo1.join();
    hilo2.join();
}
```

Posteriormente, ambos hilos inician sus respectivas tareas: aplicar QuickSort a la parte del subarreglo que le tocó a cada uno.

Tal como mencioné antes, es una implementación que únicamente utiliza dos hilos en toda su ejecución para ordenar. Su escritura y lógica es sencilla, ya que en todo el algoritmo la concurrencia únicamente aparece en su primera llamada. En todas las llamadas recursivas posteriores cada hilo opera QuickSort como si fuera secuencial. Con la clara diferencia de la implementación secuencial que ahora tenemos dos hilos ordenando sus respectivos subarreglos a la vez.

Para mejor entendimiento sobre cómo cada hilo aplica QuickSort a sus subarreglos correspondientes, a continuación muestro los métodos que se encargan de realizar dicho algoritmo.

Método principal donde se generan las llamadas recursivas. Se hace uso de particionarArray() para que coloque a nuestro pivote en su posición correcta, colocando todos los elementos menores a él a su izquierda y los mayores a su derecha.

Posteriormente se utiliza la recursión para aplicar QuickSort a los dos subarreglos obtenidos.

```
private static void ordenar(int[] array, int indiceInferior, int indiceSuperior) {
    if (indiceInferior >= indiceSuperior) return;
    int posicionFinalPivote = particionarArray(array, indiceInferior, indiceSuperior);
    ordenar(array, indiceInferior, posicionFinalPivote - 1);
    ordenar(array, posicionFinalPivote + 1, indiceSuperior);
}
```

Método encargado de la partición del array (uso de puntero izquierdo, derecho y pivote). El pivote se selecciona como el último elemento del arreglo:

```
private static int particionarArray(int[] array, int indiceInferior, int indiceSuperior)
    int pivote = array[indiceSuperior];
    int punteroIzquierdo = indiceInferior;
    int punteroDerecho = indiceSuperior;
    while (punteroIzquierdo < punteroDerecho) {
        while (array[punteroIzquierdo] <= pivote && punteroIzquierdo < punteroDerecho) {
            punteroIzquierdo++;
        }
        while (array[punteroDerecho] >= pivote && punteroDerecho > punteroIzquierdo) {
            punteroDerecho--;
        }
        intercambiarValores(array, punteroIzquierdo, punteroDerecho);
    }
    intercambiarValores(array, indiceSuperior, punteroIzquierdo);
    return punteroIzquierdo;
};
```

Método encargado de intercambiar los valores del arreglo en caso de que se cumplan las condiciones de los punteros:

```
private static void intercambiarValores(int[] array, int primerPosicion, int segundaPosicion) {
    int auxiliar = array[primerPosicion];
    array[primerPosicion] = array[segundaPosicion];
    array[segundaPosicion] = auxiliar;
}
```

2. COMPARATIVA Y DESEMPEÑO

Para comparar la eficiencia de la implementación concurrente versus la secuencial, creé cinco escenarios de prueba que demuestran no solo las fortalezas de la concurrencia, sino también sus grandes debilidades.

Para hacer esta prueba se utilizó el siguiente volumen de elementos: 1.000.000 (un millón), 500.000 (quinientos mil), 100.000 (cien mil), 1.000 (mil) y 10.

En términos de hardware, se utilizó un CPU Ryzen 3 3200g de 4 núcleos.

Para los datos de prueba, se generaron números aleatorios del -1000 al 1000.

Comparativa de ambos algoritmos

Algoritmo	1.000.000 elementos	500.000 elementos	100.000 elementos	1000 elementos	10 elementos
QuickSort secuencial	1.438 milisegundos	691 milisegundos	37 milisegundos	44.600 nanosegundos	500 nanosegundos
QuickSort concurrente	686 milisegundos	252 milisegundos	10 milisegundos	210.800 nanosegundos	252.200 nanosegundos

4. CONCLUSIÓN

Tras haber visto la comparativa entre QuickSort secuencial y QuickSort concurrente, queda en evidencia no solo la fortaleza de la concurrencia, siendo capaz de ordenar un millón de elementos en menos de la mitad de tiempo que su versión secuencial, sino también su gran flaqueza. La implementación concurrente decae en eficiencia a medida que la cantidad total de elementos es cada vez menor. Siendo el ejemplo más extremo 10 elementos, donde la versión concurrente perdió en tiempo contra la secuencial, tardando 300 veces más.

Estos resultados se deben a dos motivos. Uno de ellos es el hecho del tiempo de inicialización de los hilos; para pocos elementos, en lo que la implementación concurrente inicia los hilos con sus correspondientes tareas, la versión secuencial ya está ordenando todo su arreglo finalizando mucho antes.

El segundo motivo, y válido para el algoritmo sobre el cual se trata este informe, trata sobre el comportamiento de QuickSort con los datos de entrada. Para la versión concurrente, es muy

importante que los datos de entrada sean lo más variados posible. ¿Por qué? Bien, esto es así en gran medida por la concurrencia. Al utilizar solo dos hilos, es muy importante tener en cuenta la carga de trabajo. Dado que tratamos con valores de entrada aleatorios, por cada testeo que se hace, se está poniendo a prueba la eficiencia del algoritmo no solo por ser concurrente, sino también por como se distribuye la carga de trabajo entre los dos hilos. Esto último es, cuántos elementos debe ordenar cada hilo.

Por la aleatoriedad, en la primer partición del arreglo, pueden quedarnos a lo mejor 100.000 (cien mil) elementos a la izquierda del pivote, y 900.000 (novecientos mil) a su derecha. Esto es un punto de partida muy malo para nuestra versión concurrente, ya que el 2do hilo realizará nueve veces el trabajo del otro. Significa entonces que no podríamos aprovechar la concurrencia al máximo.

Para esta versión concurrente de QuickSort lo mejor que puede ocurrirle con los datos de entrada es que cada hilo ordene la mitad del arreglo. Quiero decir, que cada hilo obtenga una carga de trabajo lo más aproximada posible a la mitad del tamaño del arreglo, así puede aprovechar la concurrencia al máximo.

Solo para dar prueba de esto último, QuickSort concurrente llegó a darme resultados mucho peores que su versión secuencial a la hora de ordenar 1.000.000 (un millón) de elementos. Esto es así ya que la carga de trabajo, completamente aleatoria, no era equitativa para cada hilo; llegué a obtener tiempos de casi 2.000 (dos mil) milisegundos solo porque el segundo o primer hilo debían de realizar la gran mayoría del ordenamiento, mientras que el restante ordenaba una pequeña cantidad del total de elementos.

Podemos concluir entonces que, QuickSort en su versión concurrente, es un muy poderoso algoritmo de ordenamiento, llegando a obtener tiempos muy buenos respecto a su versión secuencial a la hora de ordenar muchísimos elementos. Sin embargo, no hay que perder de vista su flaqueza frente a la carga desequilibrada de trabajo y la poca cantidad de elementos a ordenar. Aunque esto último es algo que afecta a casi todos los algoritmos de ordenamiento concurrentes.

REFERENCIAS

John Marty. (2021). *Quicksort Sort Algorithm in Java - Full Tutorial With Source.* https://youtu.be/h8eyY7dIiN4?si=Q7SP1mNqKA3sd1Ws

Baeldung. (2024). *Quicksort Algorithm Implementation in Java.* https://www.baeldung.com/java-quicksort