Análisis de Complejidad Temporal de Quicksort frente a Algoritmos In-Place Clásicos

Alexis Raciel Ibarra Garnica
Facultad de Informática
Universidad Autónoma de Querétaro
Santiago de Querétaro, Qro, México

Pablo Natera Bravo
Facultad de Informática
Universidad Autónoma de Querétaro
Santiago de Querétaro, Qro, México

***Resumen*—En este trabajo se implementó el algoritmo Quicksort en C sharp en su versión original y una versión modificada junto con otros algoritmos clásicos de ordenamiento in-place BubbleSort, Flag BubbleSort, SelectionSort e InsertionSort. El objetivo principal fue comparar su rendimiento manteniendo una complejidad espacial de $O(1)$. Los resultados muestran que el uso de la técnica de la mediana de tres en la versión optimizada del Quicksort mejora notablemente su estabilidad en arreglos previamente ordenados, evitando el peor caso de $O(n^2)$ y mejora ligeramente su rendimiento en arreglos desordenados. Además, entre los algoritmos in-place, Quicksort obtiene el mejor desempeño siempre que se utilice una estrategia adecuada para la selección del pivote.**

***Index Terms*—Quicksort, Divide and Conquer, Big O.**

I. Introducción

El algoritmo Quicksort es uno de los algoritmos más eficientes y la primera elección en muchas aplicaciones debido a su flexibilidad de los distintos tipos de datos que acepta como entrada y por usar menos recursos de memoria.

Motivación y Objetivo: El objetivo de este mini-paper es analizar y validar su eficiencia asintótica ($O(n \cdot \log n)$ promedio) mediante la implementación y el estudio de sus principios fundamentales.

Estructura del Paper: A continuación, se detallan los conceptos teóricos clave, la descripción del algoritmo, el análisis de complejidad y las conclusiones.

II. Marco Teórico

II-A. Antecedentes Históricos

El algoritmo *Quicksort*, fue desarrollado por Tony Hoare en 1959 mientras era estudiante de ciencias de la computación en la Universidad Estatal de Moscú, surgió de la necesidad de ordenar listas de palabras para traducirlas del ruso al inglés. Su propuesta superó en eficiencia al método de *Insertion Sort*, haciendo uso del nuevo concepto de particiones e implementándose inicialmente en Mercury Autocode. Posteriormente, al regresar a Inglaterra, Hoare adaptó su idea para mejorar el algoritmo de *Shellsort*, lo que lo llevó a publicar Quicksort en 1961.[1]

Desde su primera publicación han habido diversas modificaciones con la intención de hacer aún más eficiente el algoritmo, siendo la primer de estas en 1962 por el mismo Hoare. Otras modificaciones se basan en mejores métodos para elegir en pivote y así reducir la complejidad en el peor caso, la más popular es usando el algoritmo de Singleton.[2]

II-B. Divide and Conquer

El Quicksort es un algoritmo que usa el paradigma de **Divide and Conquer**. Este tipo de solución implica dividir el problema en dos subproblemas independientes usando un pivote, cada uno siendo aproximadamente la mitad del problema original, y resolver estos subproblemas respectivamente para finalmente unir ambos en la solución final.

En muchas ocasiones estos subproblemas aplican de manera recursiva hasta llegar un caso base, en donde es trivial la respuesta y de ahí se van uniendo las subsoluciones hasta llegar a la solución final. Cuando la división es balanceada, este enfoque logra reducir el número de niveles de recursión a $\log n$ niveles de recursión. Otros algoritmos que ocupan este paradigma son la *Búsqueda Binaria* y la *Transformada Rápida de Fourier*.[3]

II-C. Algoritmo

Aplicando el paradigma de *Divide and Conquer* y el concepto de particiones se obtiene la siguiente lógica en forma de pseudocódigo para el Quicksort[4]:

Quicksort ($Array, low, high$)
if $low < high$ **then**
  $pivotID \leftarrow$ PARTITION($Array, low, high$)
  QUICKSORT($Array, low, pivotID - 1$)
  QUICKSORT($Array, pivotID + 1, high$)
end if

---

para el Quicksort. Para las particiones luciría así:

Partition $(Array, low, high)$
$x \leftarrow Array[high]$
$i \leftarrow low - 1$
for $j = low$ **to** $high - 1$ **do**
  if $Array[j] \leq x$ **then**
    $i \leftarrow i + 1$
    swap $Array[i], Array[j]$
  end if
end for
swap $Array[i + 1], Array[high]$
return $i + 1$

En la primera llamada **low** sería 0 y **high** sería la longitud del arreglo menos 1. Como se puede ver en esta implementación el pivote simplemente es el valor extremo de la derecha.

II-D. Complejidad y pivote

A diferencia de otros algoritmos donde la complejidad depende de la posición de los elementos en el arreglo, en Quicksort la complejidad depende casi enteramente de la elección de pivote.[5]

1. Mejor caso: se escoge un pivote que sea exactamente la mediana y cree dos particiones de la misma longitud. Cuando esto sucede es que tenemos exactamente $\log n$ niveles de recursividad y cada nivel requiere n operaciones, por lo que la complejidad resulta ser: $\Omega(n \cdot \log n)$.
2. Peor caso: si el pivote escogido genera particiones que dividen el arreglo de manera desigual ($n-1$ y 0 elementos) en cada iteración, teniendo n niveles de recursividad en donde cada nivel necesita de $\approx n$ operaciones, resultando en una complejidad de $O(n^2)$.
3. Paso promedio: las particiones no son perfectamente balanceadas pero ambas son siempre de longitudes mayores a cero, en este caso la complejidad también converge a $\Theta(n \cdot \log n)$.

Para entender mejor porque el caso promedio converge a $n \cdot \log n$ es útil visualizar las particiones cuando están desbalanceadas pero no de longitud cero. En este ejemplo tenemos un arreglo en donde de manera recursiva las particiones van a tener una proporción de $1/10$ y otra de $9/10$.

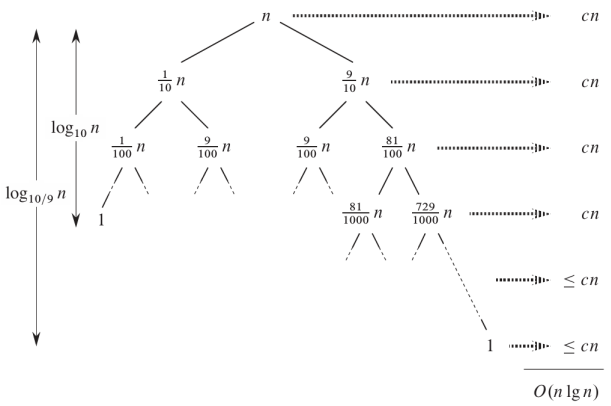

Figura 1. Diagrama de particiones desbalanceadas. Imagen obtenida de[6].

Como se puede ver el número de niveles de recursión llegará a ser $\log_{10/9} n$ en la rama de la derecha, y en la rama de la izquierda será $\log_{10} n$.

Podemos simplificar estos logaritmos haciendo uso de la fórmula de cambio de base:

$$\log a = \log b \cdot \log_b a \tag{1}$$

Y posteriormente, por nomenclatura de notación Big O, podemos ignorar la constante que aparece.

Para cada nivel de recursión se tendrán que hacer $\approx n$ operaciones, resultando en la complejidad de: $\Theta(n \cdot \log n)$.

Sabiendo esto es simple entender porqué es necesaria una buena técnica para escoger el pivote.

Una técnica muy popular y relativamente sencilla de entender el agarrar el primer, último y elemento de el medio del arreglo o partición, calcular su mediana y usar eso como pivote. Esta técnica ayuda a que el quicksort pase de tener una complejidad del peor caso para arreglos ordenados al mejor caso, y también ayuda a siempre tener dos particiones de longitud distintas a cero.

II-E. Algoritmos de ordenamiento in-place

Quicksort es un algoritmo in-place, lo que significa que funciona haciendo intercambios sin hacer uso de estructuras de datos externas. Otros algoritmos que son in-place serían ambas versiones del Bubblesort, Insertion sort y el Selection sort. La gran ventaja de usar este paradigma es que la complejidad espacial del algoritmo es bastante baja, $\Theta(\log n)$ en el mejor caso y caso promedio y $O(n)$ en el peor caso. Sin embargo, si hacemos uso de la técnica de la mediana de tres para escoger el pivote, nos aseguramos de tener una complejidad espacial logarítmica al tener los $\log n$ niveles.[7]

III. Metodología

Aquí debe describir los métodos, enfoques o procedimientos utilizados en su investigación.

III-A. Implementaciones

III-A1. Quicksort: En esta primera implementación se recreó paso a paso el pseudocódigo mencionado anteriormente en C#, donde el pivote escogido es el elemento de la derecha del arreglo/partición. La función principal quedó de la siguiente manera:

```
void q_sort(int low, int high)
        {
            if (low < high)
            {
                int q = Partition(low, high);
                q_sort(low, q - 1);
                q_sort(q + 1, high);
            }
        }
```

Mientras que la función partición quedó así:

```
int Partition(int low, int high)
        {
```

```
        int i = low - 1;
        int pivot = Array[high];
        int temp;
        for (int j = low; j < high; j++)
        {
            NumComparisons++;
            if (Array[j] <= pivot)
            {
                i++;
                // swap
                temp = Array[j];
                Array[j] = Array[i];
                Array[i] = temp;
                NumSwaps++;
            }
        }

        // final swap
        temp = Array[i + 1];
        Array[i + 1] = Array[high];
        Array[high] = temp;
        NumSwaps++;

        return i + 1;
    }
```

Añadiendo un contador de comparaciones e intercambios para cuantificar complejidad.

III-A2. Quicksort Optimizado: Posterior a la implementación original del quicksort, se modificó para que el pivote sea escogido usando la mediana del primer, medio y último elemento de la partición. Para ello se hizo una función que tiene como parámetros un arreglo y los índices inferior y superior, y su valor de retorno es el índice de la mediana de las posiciones inferior, media y superior.

```
int Median_Of_3(int low, int high)
    {
        int middle = (low + high + 1)
        ↪  / 2;
        int x1 = Array[low];
        int x2 = Array[middle];
        int x3 = Array[high];

        if ((x2 <= x1 && x1 <= x3) ||
        ↪  (x3 <= x1 && x1 <= x2))
            return low;
        else if ((x1 <= x2 && x2 <=
        ↪  x3) || (x3 <= x2 && x2 <=
        ↪  x1))
            return middle;
        else
            return high;
    }
```

Teniendo esta función lista sólo hubo que añadir dos líneas antes del ciclo for en la función partición para escoger el pivote y mandarlo al final.

```
        int i = low - 1;
        int pivotIndex = Median_Of_3(low, high
        NumComparisons++;
        // swap pivot con high
        NumSwaps++;
        int temp = Array[pivotIndex];
        Array[pivotIndex] = Array[high];
        Array[high] = temp;

        int pivot = Array[high];
```

III-A3. Bubblesort: Para poder hacer una comparativa al quicksort se implementaron tanto el Bubblesort como el Flag Bubblesort, teniendo la implementación del Bubblesort así:

```
void B_Sort()
{
    for (int i = 0; i < n - 1; i++)
    {
        for (int j = 0; j < n - i - 1;
        ↪  j++)
        {
            NumComparisons++;
            if (Array[j + 1] < Array[j])
            {
                // Swap
                int temp = Array[j + 1];
                Array[j + 1] = Array[j];
                Array[j] = temp;

                NumSwaps++;
            }
        }
    }
}
```

Y para el Flag Bubblesort sólo se anadió el flag de intercambio:

```
void FB_Sort()
    {
        for (int i = 0; i < n - 1;
        ↪  i++)
        {
            bool swapped = false;
            for (int j = 0; j < n - i
            ↪  - 1; j++)
            {
                NumComparisons++;
                if (Array[j + 1] <
                ↪  Array[j])
                {
                    int temp = Array[j
                    ↪  + 1];
                    Array[j + 1] =
                    ↪  Array[j];
                    Array[j] = temp;

                    NumSwaps++;
                    swapped = true;
```

```
                }
            }
            if (!swapped)
            {
                break;
            }
        }
    }
```

Ambas implementaciones tienen un contador de operaciones e intercambios para medir complejidad.

III-A4. Insertionsort: Para implementarlo en C# se hizo de la manera siguiente:

```
void I_Sort()
    {
        for (int i = 1; i < n; i++)
        {
            var key = Array[i];
            int j = i - 1;
            while ( j >= 0 && Array[j]
            ↪  > key)
            {
                NumComparisons++;
                Array[j + 1] =
                ↪  Array[j];
                j--;
                NumSwaps++;
            }
            Array[j + 1] = key;
        }
    }
```

añadiendo un contador de operaciones para cuantificar la complejidad.

III-A5. Selectionsort: Para implementarlo en C# se hizo de la manera siguiente:

```
void S_Sort()
    {
        for (int i = 0; i < n - 1;
        ↪  i++)
        {
            int min = i;

            for (int j = i + 1; j < n;
            ↪  j++)
            {
                NumComparisons++;
                if (Array[j] <
                ↪  Array[min])
                {
                    min = j;
                }
            }
            NumSwaps++;
            int temp = Array[i];
            Array[i] = Array[min];
            Array[min] = temp;
        }
    }
```

añadiendo un contador de operaciones para cuantificar la complejidad.

III-A6. Generador de arreglos: Se crearon dos funciones para poder probar nuestras implementaciones con arreglos de longitud n.

La primera genera n elementos aleatorios de la siguiente manera:

```
int[] rand_n_array(int n)
    {
        int[] Array = new int[n];
        Random random = new Random(100);
        int Valor;
        for (int i = 0; i < n; i++)
        {
            Valor = random.Next(0, 2 * n);
            Array[i] = Valor;
        }
        return Array;


    }
```

Cabe aclarar que se usa una semilla para asegurar reproducibilidad pero no es necesaria.

Mientras que la segunda función genera n elementos ordenados de menor a mayor:

```
int[] n_array(int n)
    {
        int[] Array = new int[n];
        for(int i = 0; i < n; i++)
        {
            Array[i] = i;
        }
        return Array;
    }
```

III-B. Experimentos

III-B1. Quicksort vs Quicksort optimizado: Primeramente se compararon ambas implementaciones de Quicksort en arreglos ordenados y aleatorios de longitudes desde 10 hasta 1000 elementos.

La elección de que el límite de longitud sea 1000 es para evitar el StackOverFlowException de C# cuando la implementación no optimizada ordena arreglos ordenados.

Finalmente se guardaron los resultados en archivos csv para su posterior análisis.

III-B2. Quicksort optimizado vs demás algoritmos: En el segundo experimento ahora se compararon las implementaciones del Quicksort optimizado, Bubblesort, Flag Bubblesort, Insertionsort y Selectionsort con arreglos tanto ordenados como desordenados.

Para los arreglos ordenados las longitudes de las listas estuvieron en el rango de 100 a 50,000 elementos y para los arreglos desordenados los rangos variaron desde 100 hasta 10,000. Esta decisión se tomó para evitar saturar la

computadora debido a la complejidad $O(n^2)$ de los demás algoritmos.

Finalmente se guardaron los resultados en archivos csv para su posterior análisis.

IV. Resultados

IV-A. Quicksort vs Quicksort optimizado

Cuando se hace la comparación con arreglos desordenados obtenemos esta gráfica:

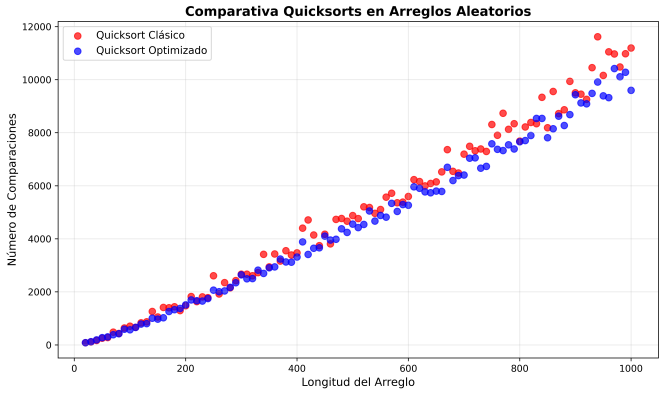

Figura 2. Comparativa de Quicksorts en escala lineal.

Como se puede ver el quicksort optimizado es ligeramente más eficiente que el tradicional.
Mientras que cuando se hace la comparación con arreglos ordenados obtenemos esta otra gráfica:

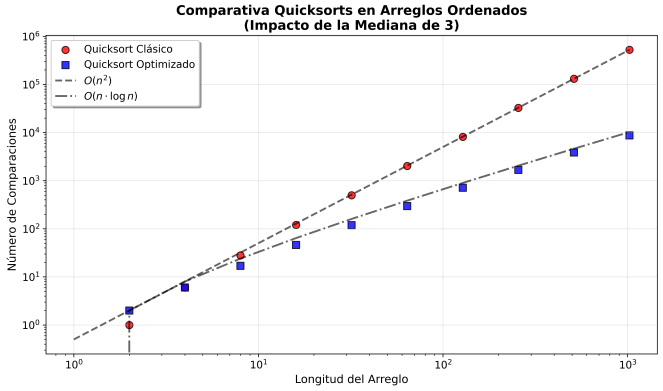

Figura 3. Comparativa de Quicksorts en escala logarítmica.

Aquí es considerablemente mayor la diferencia, mientras que el quicksort optimizado mantiene $O(n \cdot \log n)$ operaciones, el tradicional aumenta a $O(n^2)$ operaciones.

IV-B. Quicksort optimizado vs demás algoritmos

Cuando se hace la comparación con arreglos desordenados obtenemos esta gráfica:

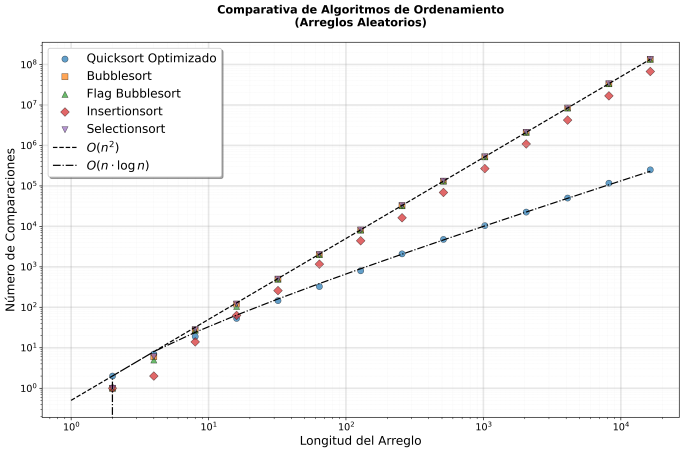

Figura 4. Quicksort vs demás en escala logarítmica.

Aquí podemos ver como los demás algoritmos requieren de $O(n^2)$ operaciones mientras que el quicksort es constante.
Mientras que cuando se hace la comparación con arreglos ordenados obtenemos esta otra gráfica:

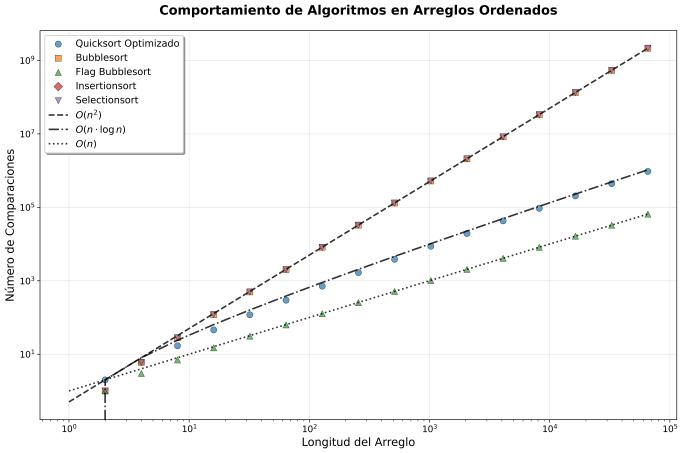

Figura 5. Quicksort vs demás en escala logarítmica.

Aquí el Insertion Sort y el Flag BubbleSort reducen su número de operaciones a $O(n)$, mientras que los demás mantienen el número de operaciones.

V. Conclusiones

Después de analizar estas distintas implementaciones hay 2 puntos importantes de mencionar:

- Escoger bien el pivote ayuda ligeramente en arreglos desordenados pero ayuda demasiado para arreglos ordenados y nos da una complejidad tanto temporal como espacial bastante más estable.
- De los algoritmos de ordenamiento in-place que se analizaron el quicksort es indiscutiblemente el ganador.

Sin embargo hay un dos preguntas que se podrían explorar a futuro:

- ¿Hay alguna implementación que supere la complejidad temporal obtenida aquí revisando si el arreglo está previamente ordenado?
- ¿qué otras técnicas para escoger el pivote existen y cuáles son sus respectivas complejidades?

El Quicksort es un algoritmo sumamente efectivo tanto en complejidad espacial como temporal, es relativamente sencillo de implementar y es bastante flexible ya que acepta distintos tipos de datos como entrada; haciéndolo una opción bastante sólida para diversas aplicaciones.

REFERENCIAS

[1] C. A. R. Hoare, "Algorithm 64: Quicksort," *Commun. ACM*, vol. 4, no. 7, p. 321, Jul. 1961. [Online]. Available: https://doi.org/10.1145/366622.366644

[2] L. Khreisat, "Quicksort a historical perspective and empirical study," *International Journal of Computer Science and Network Security*, 07 2008.

[3] G. Heineman, G. Pollice, and S. Selkow, *Algorithms in a Nutshell, 2e*. Sebastopol, CA: O'Reilly Media, Mar. 2016.

[4] A. Y. Bhargava, *Grokking Algorithms*. New York, NY: Manning Publications, May 2016.

[5] R. Sedgewick and K. Wayne, *Algorithms*. Addison-Wesley Professional, Feb. 2011.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Third Edition*, 3rd ed. The MIT Press, 2009.

[7] C. LLC, *Introduction to algorithms a comprehensive guide for Beginners*, 1st ed. Packt Publishing, 2024.

APÉNDICE A
REPOSITORIO

Para ver el repositorio completo de GitHub haga click **aquí**.

A-A. Implementaciones en C#

Para las implementaciones en C# haga click **aquí**.

A-B. Experimentos

Para los experimentos en C# haga click **aquí**.

A-C. Tablas de resultados

Para las tablas de resultados haga click **aquí**.