

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

Alexis Raciél Ibarra Garnica

Facultad de Informática

Universidad Autónoma de Querétaro

Santiago de Querétaro, Qro, México

direccion.correo@email.com

Pablo Natera Bravo

Facultad de Informática

Universidad Autónoma de Querétaro

Santiago de Querétaro, Qro, México

pablonatera16@gmail.com

Abstract—In this paper we implemented Quicksort in C sharp along with other in-place sorting algorithms such as BubbleSort, Flag BubbleSort, SelectionSort and InsertionSort. This with the intention of comparing performance while maintaining $O(1)$ spacial complexity. We concluded that of all the in-place sorting algorithms, Quicksort is the best option given that the pivot is chosen adequately.

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

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 primer 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]

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]

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 ← 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 ← Array[high]
i ← low - 1
for j = low to high - 1 do
    if Array[j] ≤ x then
        i ← i + 1
        swap Array[i], Array[j]
    end if
end for
swap Array[i + 1], Array[high]
return i + 1
```

D. Complejidad y pivote

- 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 $9/10$.

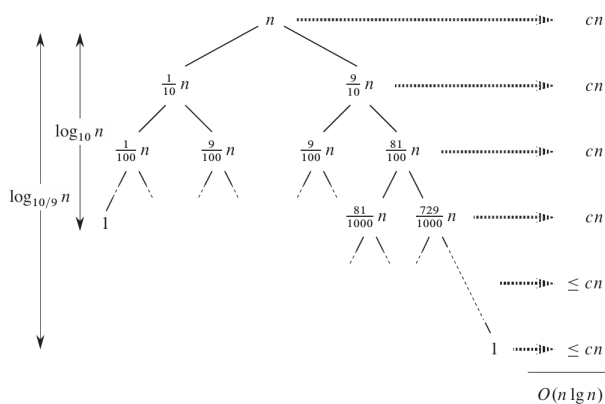


Fig. 1. Diagrama de particiones desbalanceadas. Imagen obtenida de [6].

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:

$$\log a = \log b \cdot \log_b a \quad (1)$$

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. 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.

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.

A. Implementaciones

1) *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 quicksort () {  
    ...  
}
```

Mientras que la función partición quedó así:

```
void partition() {
    ...
}
```

Añadiendo un contador de comparaciones e intercambios para cuantificar complejidad.

2) *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.

```
void median_of_3() {
    ...
}
```

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.

```
void partition_v2() {
...
}
```

3) *Bubblesort*: Para poder hacer una comparativa al quicksort se implementaron tanto el Bubblesort como el Flag Bubblesort, teniendo la implementación del Bubblesort así:

```
void bsort() {
...
}
```

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

```
void fbsort() {
...
}
```

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

4) *Insertionsort*: Para implementarlo en C# se hizo de la manera siguiente:

```
void isort() {
...
}
```

añadiendo un contador de operaciones para cuantificar la complejidad.

5) *Selectionsort*: Para implementarlo en C# se hizo de la manera siguiente:

```
void ssort() {
...
}
```

añadiendo un contador de operaciones para cuantificar la complejidad.

6) *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() {
...
}
```

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() {
...
}
```

B. Experimentos

1) *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.

2) *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 las 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

1) *Quicksort vs Quicksort optimizado*: Cuando se hace la comparación con arreglos desordenados obtenemos esta gráfica: Mientras que cuando se hace la comparación con arreglos ordenados obtenemos esta otra gráfica:

2) *Quicksort optimizado vs demás algoritmos*: Cuando se hace la comparación con arreglos desordenados obtenemos esta gráfica: Mientras que cuando se hace la comparación con arreglos ordenados obtenemos esta otra gráfica:

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?

Podemos afirmar que 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.

REFERENCES

- [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," 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.

VI. APÉNDICE

A continuación se anexan los códigos en C# de implementaciones y el de python para generar los gráficos.

A. Implementaciones C#

Código

B. Gráficos

Código