Comparativa Quicksort

Alexis Raciel Ibarra Garnica
Facultad de Informática
Universidad Autómona de Querétaro
Santiago de Querétaro, Qro, México
direccion.correo@email.com

Pablo Natera Bravo
Facultad de Informática
Universidad Autómona 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.

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 para el Quicksort[4]:

1) Si la longitud del arreglo es menor a dos, se retorna el elemento directamente (caso base).
2) Se selecciona un pivote (existen distintas técnicas para ello).
3) El arreglo se reordena colocando los elementos menores al pivote a la izquierda y los mayores a la derecha. Generando así las particiones. **Divide**
4) Los pasos anteriores se aplican recursivamente a cada partición hasta alcanzar el caso base. **Conquer**

Visto como pseudocódigo sería algo así:

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.

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 $9/10$.

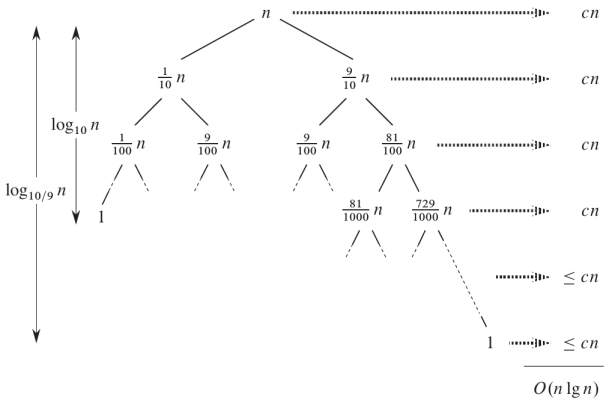

Fig. 1. Diagrama de particiones desbalanceadas. Imagen obtenida de [6].

En este ejemplo el número de niveles de recusrión llegará a ser $\log_{10/9} n$ en la rama de la derecha, y para la rama de la izquierda será $\log_{10} n$. Y 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 fácil entender porqué es necesaria una buena técina para escoger el pivote.
Una técinca 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, $\log n$ en el mejor caso y caso promedio y n en el peor caso. Usando la mediana de tres para escoger el pivote también nos aseguramos de tener una complejidad espacial logarítmica. [7]

II. Metodología

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

III. Resultados

Presente los hallazgos y datos recopilados de su metodología. Incluya figuras y tablas como se muestra a continuación.

IV. Conclusiones

El Quicksort es el mejor de los algoritmos de in-place dado que se escoga bien el pivote.

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.