Comprobación matemática de la complejidad promedio del algoritmo de ordenamiento Quicksort.

Faiber Alonso Hernández Tavera @fai-aher
Diseño y Análisis de Algoritmos, Dpto de Ingeniería de Sistemas y Computación
Universidad de los Andes, Bogotá, Colombia
f.hernandezt@uniandes.edu.co, fai.ahernandez@pm.me
Fecha de presentación de la primera versión: marzo 12 de 2023

Contenidos por página

1. Contextualización

El algoritmo de ordenamiento Quicksort, que funciona por medio de la estrategia "Divide y vencerás", corresponde a un mecanismo computacional para ordenar un conjunto de datos cuyos valores pueden ser comparados por medio de igualdades o desigualdades. Este algoritmo centra su funcionamiento en la selección de un pivote y de forma recursiva ordenar los elementos que sean menores y mayores a él.

La complejidad temporal de este algoritmo en su peor caso es de $O(n^2)$ y en un caso promedio es de O(nLog(n)), todo depende de lo conveniente que resulte el pivote seleccionado. El caso promedio de este algoritmo en realidad se aproxima al siguiente valor una vez se aplican procedimientos matemáticos para realizar una determinación precisa:

$$(0) \quad 2n(ln(n)) \approx 1.39n \ log_2 n$$

Esta complejidad resulta llamativa, pues en la mayoría de los casos, este algoritmo es capaz de ordenar un conjunto de elementos de forma logarítmica. Por esta razón, es ampliamente usado en la actualidad para abordar problemas que requieren ordenamiento de datos mutuamente comparables.

2. Objetivo

Este documento tiene como finalidad comprobar de forma matemática que el algoritmo Quicksort se aproxima en complejidad temporal a la expresión (0) en su caso promedio.

3. Desarrollo

Debido a que este algoritmo utiliza la estrategia “divide y vencerás”, se compone de 3 pasos principales:

1. **Dividir:** Tomar un pivote y reorganizar los elementos de un conjunto de tal forma que todos los elementos a la izquierda del pivote al final sean de menor o igual valor a él y los que quedan a su derecha sean mayores o iguales. El pivote queda al final en su posición correcta respecto a los demás elementos. A este proceso se le llama “Partición”.
2. **Conquistar:** Ordenar los subarrays o subconjuntos que quedan a la izquierda y derecha del elemento pivote luego del primer proceso de Partición. Este ordenamiento se realiza con llamados recursivos al paso 1 (Partición).
3. **Combinar:** Al unir todos los subarrays o subconjuntos ordenados, se obtiene el conjunto inicial con todos los elementos organizados de menor a mayor.

Nota: En caso que se quiera organizar el conjunto de mayor a menor, la función del proceso de Partición debe cambiar para que los elementos a la izquierda del pivote sean mayores o iguales a él y a la derecha de menor o igual valor.

Implementación del algoritmo Quicksort

A continuación se puede apreciar una posible implementación del algoritmo Quicksort, compuesta de sus partes fundamentales.

```python
def Partition(array: list, i: int, f:int) -> int:

    #Generation of the pivot position from a random number between i and f.
    pivotPosition = randint(i, f)
    pivot = array[pivotPosition]

    #Exchange of pivot position element and last element of the array
    array[pivotPosition] = array[f]
    array[f] = pivot

    #Main cycle to exchange first element from the left that is larger than the pivot
    #and first element from the right which is smaller than the pivot.

    fromLeftIndex = i
    fromRightIndex = (f-1)

    while fromLeftIndex <= fromRightIndex:
        leftElement = array[fromLeftIndex]
        rightElement = array[fromRightIndex]

        if (leftElement > pivot) and (rightElement <= pivot):

            #swap
            temp = leftElement
            array[fromLeftIndex] = array[fromRightIndex]
            array[fromRightIndex] = temp

            #Navigator indexes advance 1 position.
            fromLeftIndex += 1
            fromRightIndex -= 1


        elif (leftElement > pivot) and (rightElement > pivot):

            #right navigator index should decrease 1.
            fromRightIndex -= 1
```

```python
        elif (leftElement <= pivot) and (rightElement <= pivot):

            #left navigator index should increase 1.
            fromLeftIndex += 1

        else:
            #if no ideal condition is achieved, navigator indices should advance 1 position.
            fromLeftIndex += 1
            fromRightIndex -= 1

    #Finally, pivot and the last element with the left index counter are swapped.
    array[f] = array[fromLeftIndex]
    array[fromLeftIndex] = pivot

    pivotPosition = fromLeftIndex

    return pivotPosition
```

```python
#Second function: Quicksort

"""
@param i: initial position (index) of the array that wants to be sorted.
@param f: final position (index) of the array that wants to be sorted.

@return array: the array that was given in parameters but sorted with the Quicksort algorithm.

"""


def Quicksort(array: list, i: int, f: int) -> list:

    #This only works if the initial index is smaller than the final index in the array it is sorting.
    if i < f:
        #Partition is made to get the pivot position after sorting the array in some range.
        pivotPosition = Partition(array,i,f)
        #Recursive call to sort elements in a range which does not include the pivot
        # (cause it is already in the final correct position).
        Quicksort(array, i, pivotPosition-1)
        Quicksort(array, pivotPosition + 1, f)

    return array
```

Figura1. Posible implementación del algoritmo Quicksort. Elaboración propia

Análisis del caso promedio del algoritmo Quicksort

Si se toma la decisión de definir el pivot como un número aleatorio entre los elementos de un conjunto o arreglo, es más probable que se obtenga un valor que logre repartir en una cantidad casi equitativa los elementos a su izquierda y derecha.

Cuando los elementos a la izquierda y derecha del primer pivot luego de la partición son similares en cantidad, es posible experimentar un comportamiento logarítmico de la función de complejidad temporal para ordenar todo el conjunto.

Para la comprobación matemática de la complejidad temporal, se considera un arreglo de valores mutuamente comparables por medio de igualdades y desigualdades. Este arreglo cuenta con **n elementos**.

La complejidad temporal promedio de este algoritmo depende directamente del número total de comparaciones que se realicen a la hora de ordenar un arreglo o conjunto. En el caso de un arreglo con n elementos, el número total de comparaciones esperadas se obtienen a partir de dos componentes centrales:

1. Las comparaciones realizadas en la primera partición de los n elementos después de haber elegido el pivote de manera aleatoria.
2. Las comparaciones realizadas en las dos sublistas resultantes de la primera partición más las comparaciones esperadas en las 4 sublistas que se generen de la partición de estas 2 sublistas y así en adelante.

Paso de partición:

Teniendo en cuenta la elección aleatoria del pivote, la mayor probabilidad corresponde al caso en el que este elemento no es el último o primero del arreglo. Esto es importante porque la cantidad de comparaciones que se realizan en la primera partición es (n-1, el pivote contra todos los demás elementos) pero en el peor caso, cuando el pivote es el primer o último elemento del arreglo.

A esta cantidad $(n-1)$ de comparaciones se le suman otras dos correspondientes al caso específico de la implementación del algoritmo realizada, pues existe siempre una comparación para revisar si el índice navegador por la izquierda es igual al de la derecha (debido al While) y si son iguales, se ejecuta de nuevo el ciclo para después volver a comparar si siguen siendo iguales o ya se sobrepasaron entre sí.

Cuando el pivote se elije de forma aleatoria, la probabilidad de que cada elemento sea elegido como tal es de:

$$p = \frac{1}{n}$$

Por lo tanto, la cantidad de comparaciones que se esperan en la primera partición es de menos de $\frac{n}{2}$ y esto se puede evidenciar en el caso de la implementación hecha de forma propia, donde existen dos números que recorren los índices del arreglo; uno desde la última posición y otro desde la primera (se terminan encontrando en el índice de la posición inicial del pivote).

Entonces, según lo explicado anteriormente, la cantidad de comparaciones iniciales para el primer paso de partición en el caso del algoritmo implementado varía de menos de $n/2$ a $(n+1)$ comparaciones. Aún así, para la presente demostración se va a tomar el peor caso de $\boldsymbol{n+1}$ **comparaciones** con el objetivo de realizar un análisis realista de la aptitud que tiene usarlo.

Paso de ordenamiento recursivo:

En cada partición que se realiza, la posición inicial del pivot seleccionado corresponde también a su posición final en el arreglo, por esta razón, se llama la función de Partición sobre substrings que no incluyan la posición del pivote.

Un caso base en el ordenamiento con Quicksort corresponde a llamar la función de partición sobre un arreglo de tamaño 0 o 1. Pues no requiere ni una comparación para "ordenarlo".

Más allá de ese caso base, el costo recursivo de ordenar los elementos menores o iguales al pivot y los mayores a este depende de la posición del pivot mismo. Para analizar de

forma matemática este costo con base en el número esperado de comparaciones, se asume **k** como una variable que toma el valor de los índices del arreglo:

$$k \in \{0,1,2,3 \dots (n-1)\}$$

En el caso del pivot seleccionado en *k = 0*, el costo recursivo de ordenamiento será $C_0 + C_{n-1}$ (peor caso). Donde C_n expresa el número de comparaciones necesarias para ordenar los elementos de un array o subarray. En el peor caso, ordenar los elementos mayores o menores al pivot en la primera o última posición del arreglo toma 0 comparaciones para los elementos que están en su extremo vacío (sin más elementos) y $\boldsymbol{n-1}$ **comparaciones** para los demás elementos (todos excepto él), por tanto, tendría que compararse con todos los demás valores.

En general, si el pivot se localiza en la posición k del arreglo, el costo recursivo medido con comparaciones será de:

$$C_k + C_{n-k}$$

el valor que tome k entre las n posibilidades tiene la misma posibilidad ($\frac{1}{n}$), entonces el costo recursivo promedio del algoritmo se puede expresar así:

$$C_n = \frac{(C_0 + C_{n-1}) + (C_1 + C_{n-2}) + \cdots + (C_{n-2} + C_1) + (C_{n-1} + C_0)}{n}$$

Nótese que cada expresión en la suma aparece repetida desde $k = 0$ hasta $k = n - 1$

y por tanto, es posible simplificar la expresión recursiva así:

$$C_n = 2 * \frac{(C_0) + (C_1) + (C_2) + \cdots + (C_{n-1})}{n}$$

Combinación del paso de partición y el paso recursivo

Al combinar el costo temporal de ejecutar los dos pasos principales que componen al algoritmo Quicksort resulta la siguiente expresión:

$$C_n = (n+1) + 2\frac{(C_0) + (C_1) + (C_2) + \cdots + (C_{n-1})}{n}$$

y la ecuación recursiva se puede denotar de la siguiente forma:

$$C_n = C(n) = \begin{cases} 0, & n = 0 \vee n = 1 \\ (n+1) + 2\dfrac{(C_0) + (C_1) + (C_2) + \cdots + (C_{n-1})}{n}, & n \geq 2 \end{cases}$$

donde $n = 0 \ \vee n = 1$ son los casos base de este algoritmo recursivo.

Análisis matemático de la complejidad temporal promedio del algoritmo

A continuación se va a intentar simplificar la expresión obtenida para el caso recursivo del algoritmo con el objetivo de observar de foma más visual el comportamiento de su complejidad temporal en función del número n de elementos que tenga el array o conjunto que quiera ordenar.

Para simplificar la variable n que se encuentra dividiendo la suma principal, se va a multiplicar a cada lado de la igualdad por n:

$$n * C_n = n * (n+1) + 2\big((C_0) + (C_1) + (C_2) + \cdots + (C_{n-1})\big) \qquad (\mathbf{1})$$

Para simplificar una gran cantidad de términos en esta última ecuación, se puede analizar la expresión para (C_{n-1}):

$$(n-1) * C_{n-1} = (n-1) * \big((n-1)+1\big) + 2\big((C_0) + (C_1) + (C_2) + \cdots + (C_{n-2})\big)$$
$$= (n-1) * n + 2\big((C_0) + (C_1) + (C_2) + \cdots + (C_{n-2})\big) \qquad (\mathbf{2})$$

Esta última ecuación comparte muchos términos con la primera obtenida para (C_n),

si se sustrae la segunda (2) de la primera (1), se obtiene la siguiente expresión:

$$nC_n - \ (n-1)C_{n-1} = \big(n(n+1)\big) - (n(n-1)) + 2(C_{n-1})$$

Si se desarrolla la parte:

$$\big(n(n+1)\big) - (n(n-1))$$
$$= \ n^2 + n - n(n-1)$$
$$= \ n^2 + n - n^2 + n$$
$$= \mathbf{2n}$$

El resultado es $2n$ y la ecuación se simplifica así:

$$nC_n - \ (n-1)C_{n-1} = 2n + 2(C_{n-1})$$
$$nC_n - \ nC_{n-1} + 1\ C_{n-1} = 2n + 2(C_{n-1})$$
$$nC_n = 2n + 2(C_{n-1}) + nC_{n-1} - \ C_{n-1}$$
$$= 2n + (C_{n-1}) + nC_{n-1}$$
$$= (C_{n-1})(n+1) + 2n$$

y ahora se dividen a ambos lados de la igualdad por la expresión $n(n+1)$

$$\frac{C_n}{n+1} = \frac{(C_{n-1})}{n} + \frac{2}{n+1} \qquad (3)$$

Al comparar el término de la izquierda de la igualdad con el término de la derecha de la igualdad, se puede notar que el de la izquierda corresponde al de la derecha pero cuando n vale uno menos (n-1) y se le ha sumado la expresión $\frac{2}{n+1}$.

Esto permite expresar el término $\frac{(C_{n-1})}{n}$ y sus consecuentes valores (restando 1 a la variable n) a partir de sustituciones, teniendo en cuenta la ecuación (3).

Por tanto, las primeras expresiones que se obtienen al restarle 1 consecuentemente a n y realizar sustituciones son:

$$\frac{C_n}{n+1} = \frac{(C_{n-1})}{n} + \frac{2}{n+1}$$

$$= \left[\frac{(C_{n-2})}{n-1} + \frac{2}{n}\right] + \frac{2}{n+1}$$

$$= \left[\left[\frac{(C_{n-3})}{n-2} + \frac{2}{n-1}\right] + \frac{2}{n}\right] + \frac{2}{n+1}$$

Esta consecución de sustituciones pueden llegar hasta el caso base C_1 , cuando se realizan 0 comparaciones. Para ese momento, y teniendo en consideración que los valores de **x** (x es el valor obtenido luego de restarle 1 a n de forma consecutiva) comienzan desde **2 para el caso recursivo C_1** , se llega a la siguiente igualdad:

$$\frac{C_n}{n+1} = \frac{C_1}{2} + \frac{2}{3} + \frac{2}{4} + \frac{2}{5} + \frac{2}{6} + \frac{2}{7} + \cdots + \frac{2}{n+1}$$

Ahora, con el objetivo de simplificar el término $n+1$ y dejar despejado C_n que es la complejidad promedio que se quiere calcular, se multiplica por $n+1$ a ambos lados. Además, debido a que $C_1 = 0$, este valor se elimina de la expresión y luego se factoriza por factor común 2 la suma.

$$C_n = 2(n+1)\left(\frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \frac{1}{6} + \frac{1}{7} + \cdots + \frac{1}{n+1}\right)$$

La parte de la expresión,

$$\left(\frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \frac{1}{6} + \frac{1}{7} + \cdots + \frac{1}{n+1}\right)$$

cuenta con la particularidad de poder ser aproximada como el área de la curva $y = \frac{1}{x}$ para valores de x entre 3 y $(n + 1)$ en este caso. Este área bajo la curva puede verse como una sumatoria de Riemman de los valores de n entre 3 y $(n + 1)$ para la expresión $\frac{1}{x}$: $\sum_{x=3}^{n+1} \frac{1}{x_i} \Delta x$ y su aproximación gráfica sería la siguiente:

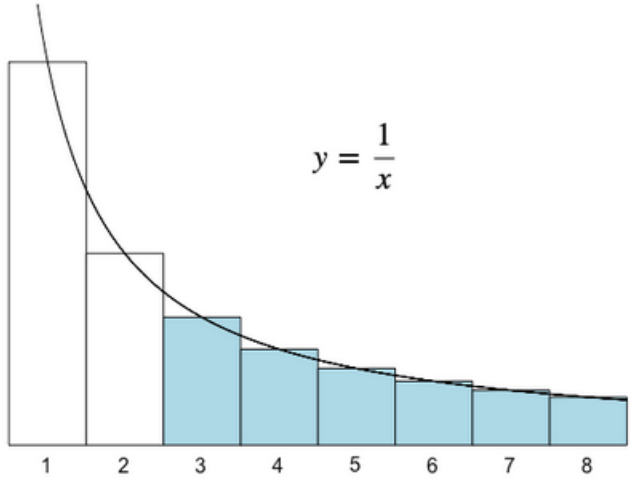

Figura 2. Gráfica de la aproximación que se podría realizar del cálculo del área bajo la curva descrita por la función $y = \frac{1}{x}$, la cual resulta ser una aproximación a la expresión de complejidad temporal obtenida para el caso promedio del algoritmo Quicksort para ordenar n elementos de un arreglo. Tomada de https://mathcenter.oxford.emory.edu/site/cs171/quickSortAnalysis/

Este área bajo la curva también puede ser determinada por medio del cálculo de la integral definida entre $x_1 = 3$ (límite inferior) y $x_2 = n + 1$ (límite superior) de la función $y = \frac{1}{x}$ y el resultado de esta operación, junto a la multiplicación por el término que acompaña esa suma: $2(n + 1)$ conformarían una acertada aproximación del caso promedio del algoritmo.

$$C_n \sim 2(n+1)\left(\frac{1}{3}+\frac{1}{4}+\frac{1}{5}+\frac{1}{6}+\frac{1}{7}+\cdots+\frac{1}{n+1}\right)$$

$$C_n \sim 2(n+1)\int_3^{n+1}\frac{1}{x}\,dx$$

Y la solución a la integral es la siguiente:

$$\int_3^{n+1}\frac{1}{x}\,dx = \ln x\ |_3^{n+1} \quad \left(Integral\ inmediata\ de\ \frac{1}{x}dx\right)$$

$$= \mathbf{ln}\ (\boldsymbol{n}+\mathbf{1}) - \mathbf{ln}\ (\mathbf{3})$$

Por tanto, la aproximación queda de la siguiente forma:

$$C_n \sim 2(n+1)(\ln\ (n+1) - \ln\ (3))$$

Sin embargo, para efectos de calcular una aproximación numérica no muy compleja con el logaritmo natural, se va a tomar el resultado de la integral sin evaluarse en sus extremos:

$$C_n \sim 2(n+1)(\ln(n))$$

Ahora, para llevar esta expresión a otra que pueda dar una idea del comportamiento temporal promedio de Quicksort se va a realizar lo siguiente:

Debido a que $\ln(n)$ es aproximadamente igual a $\log_2 n$ dividido por una constante, y específicamente por la constante $\log_2 e$, donde $\log_2 e$ tiene un valor aproximado de 1.44.

La expresión de complejidad aproximada puede escribirse ahora como:

$$C_n \sim 2(n+1)\frac{\log_2 n}{\log_2 e}$$

$$la\ sustitución\ de\ \ln(n)\ es\ equivalente\ a\ un\ cambio\ de\ base.$$

$$C_n \sim (n+1)(\log_2 n)\frac{2}{1.44}$$

Y el resultado de simplificar esta expresión es:

$$C_n \sim 1.39(n+1)(\log_2 n)$$

$$C_n \sim 1.39n(\log_2 n) + 1.39(\log_2 n)$$

donde $\frac{2}{1.44} \sim 1.39$.

Y debido a que la constante $1.39(\log_2 n)$ se vuelve despreciable al compararla con el término $1.39n(\log_2 n)$, es posible simplificar la expresión y dejarla de la siguiente manera:

$$\boldsymbol{C_n \approx 1.39n(log_2\, n)}$$

Análisis del resultado

Este resultado, correspondiente al número promedio de comparaciones para el algoritmo Quicksort en su complejidad temporal y con la implementación descrita, expresa que el algoritmo de ordenamiento tiene en promedio un 39% más de comparaciones que otro algoritmo que siempre sea tenga complejidad $\boldsymbol{n(log_2\, n)}$ como podría ser el caso de Merge sort. Sin embargo, Quicksort resulta más rapido que otros algoritmos de

ordenamiento como Merge sort debido a que se deben mover o manipular menos datos durante los pasos de ordenamiento recursivo de subarrays.

Si se realiza un script en Python para ver el comportamiento de la función obtenida, esta se ve de la siguiente forma, para un número n correspondiente al tamaño del arreglo que se pasa como parámetro al algoritmo.

```python
import numpy as np
import matplotlib.pyplot as plt

n = np.linspace(1, 100000, 100000) # Input size from 1 to 100000
c = 1.39 * np.log2(n) # Complexity

plt.plot(n, c)
plt.title('Average Complexity of Quick Sort')
plt.xlabel('Input Size (n)')
plt.ylabel('Complexity')
plt.show()
```

Figura 3. Script escrito en el lenguaje de programación Python para mostrar de forma visual el comportamiento de la función de complejidad promedio de Quicksort. Elaboración propia

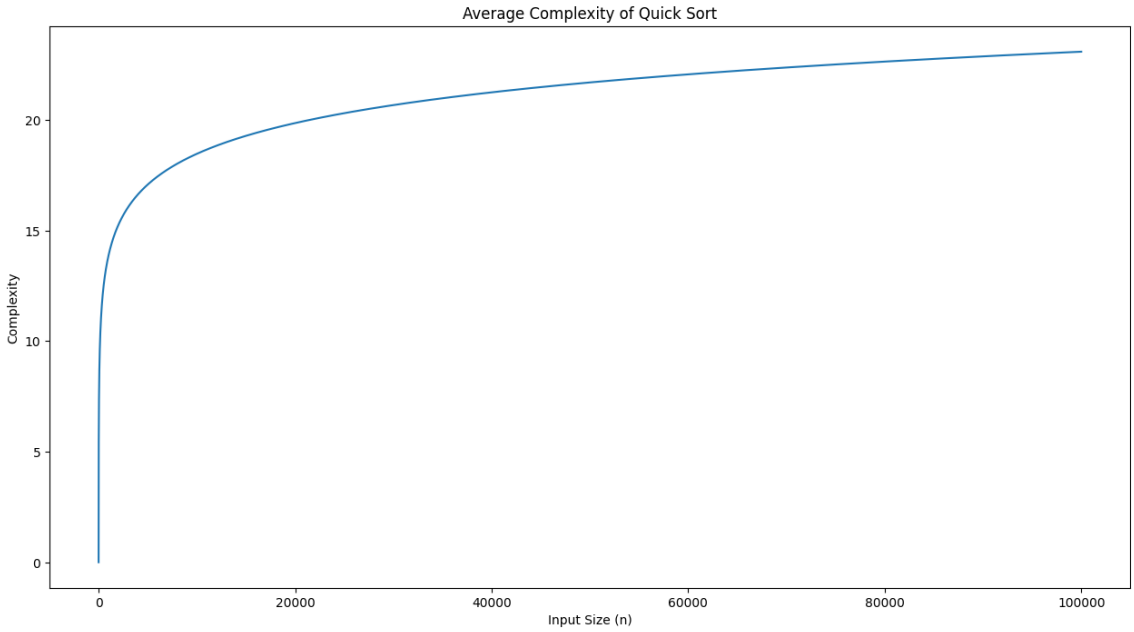

Figura 4. Gráfica que muestra el comportamiento de la función de complejidad temporal promedio para el algoritmo Quicksort. Esta función depen del tamaño n del array que se pasa como parámetro al algoritmo. Elaboración propia

Referencia consultada

Una importante parte del presente trabajo fue elaborada tomando como guía el siguiente análisis publicado en la página web de la universidad de Oxford:

Quick Sort Analysis. (s.f.). Departamento de Matemáticas y Ciencias de la Computación. Universidad de Oxford. Información consultada el 06 de marzo de 2023 de https://mathcenter.oxford.emory.edu/site/cs171/quickSortAnalysis/