Analyzing Quick Sort as a Divide and Conquer Algorithm

Eli Mishriki

San Diego State University

CS 420: Algorithms

Manju Muraldiharan

March 2, 2025

Analyzing Quick Sort as a Divide and Conquer Algorithm

Problem Statement

Sorting lists is a very important concept in computer science. The goal of a sort is to re-order lists in ascending, descending, or other niche ordering. Different algorithms can accomplish the same task but require varying amounts of time and data. Given any array of n comparable elements,

$$P = \{p_1, p_2, \ldots, p_n\}$$

where each element p_i is a comparable element, the goal of a sort is to re-order P into a sorted list P' so:

$$p'_1 \leq p'_2 \leq \cdots \leq p'_n.$$

We would expect a sorting algorithm to be efficient and perform well on a wide range of varying inputs, including already sorted, reverse sorted, very long input size, and randomly ordered lists.

This is an important problem because it is often the backbone of much computing done on a computer, so finding a fast and efficient algorithm means a better computer. Sorting algorithms can be found in areas such as database indexing, scientific computing, and search engine optimization, where performance is highly dependent on fast and efficient sorting (Knuth, 1998).

QuickSort is a quite efficient comparison-based sorting algorithm that uses a recursive divide-and-conquer approach to achieve sorting. This paper will analyze Quicksort's viability in depth.

Input and Output

Quicksort accepts an unsorted array of n elements as its input. Quicksort will output a sorted version of the array in ascending or descending order, depending on implementation. QuickSort can sort an array of elements that can be integers, floating-point numbers, or any comparable elements. The Quicksort algorithm can sort in place or return a sorted algorithm.

- **Example Input:** Quicksort accepts an array of n elements: $[5,3,0,32,1,15,4]$.
- **Example Output:** Quicksort returns a sorted array in ascending order: $[0,1,3,4,5,15,32]$.

Algorithm Definition

QuickSort is an efficient sorting algorithm developed by Tony Hoare in 1959(Hoare, 1961). It is commonly used in general purpose sorting, database systems, graphics, and more. Quicksort is slightly faster than merge sort and heapsort for random data(Sedgewick, 1977). It is considered a divide-and-conquer algorithm, meaning it splits the problem up and employs recursive to sort the array. More specifically, QuickSort works by selecting a "pivot" element from the array, then partitioning the array into two sub-arrays based on whether they are less than or greater than the pivot. The algorithm then recursively applies the same process to the sub-arrays until the array is sorted. Efficiency depends on the choice of the pivot. Strategies include choosing the first, last, or random element.

The divide and conquer technique can be broken up into 3 steps:

1. **Divide:** First, we select a pivot and then partition the array around that pivot so that elements smaller than the pivot are on the left side and larger ones are on the right side.
2. **Conquer:** Then, we recursively apply the Divide step of Quicksort on the left and right sub-arrays.
3. **Combine:** Eventually, the sorted sub arrays combine and form a fully sorted array either ascending or descending depending on implementation.

Pseudo Code

```
QuickSort(array[], low, high):
    if low >= high: # (Base Case): the array has 0-1 elements that its sorted
        return
    pivot = Partiton(array, low, high) # partition and get pivot
    QuickSort(array, low, pivot - 1) # Recusively sort left
    QuickSort(array, pivot + 1, high) # Recusively sort right


Parition(array, low, high):
    pivot = array[high] # last element pivot
    i = low # I tracks the position to place the element less than the pivot
    for j from low to high - 1: # iterates the array
    if array[j] <= pivot: # if less than pivot
        swap array[i] and array[j] # place to left
        i = i + 1 # increment I for next smaller element
    swap array[i] and array[high] # place pivot
    return I # return pivot index
```

Proof of Correctness of QuickSort

Now, our goal is to prove that QuickSort can sort arrays of comparable methods using the mathematical technique of induction. We want to prove that it works for any input size of n.

1. If $n \leq 1$, return P. This is the base case and means P is already sorted, so we just return

2. Next, we want to choose a pivot p to begin our partitioning. This can be done by either picking the first element, the last element, a random element, or using a median-of-three method.

3. Now, we want to partition P into three separate groups:

- $L(Low) = \{x \mid x < p\}$
- $C(CurrentPivot) = \{x \mid x = p\}$
- $H(High) = \{x \mid x > p\}$

4. Then, we want to recursively sort L and H, i.e. repeating steps 1-2.

5. Finally, we want to combine L, H, and C to get a sorted array.

Inductive Proof

We will now prove by induction that any array of size n can be sorted by Quicksort

Base Case: If $n = 1$, the array P will be returned since it only has one element; it is already sorted

Inductive Hypothesis: We first must assume QuickSort correctly sorts any array of size at most k, where $k \geq 1$.

Inductive Step: Now consider the array P of size $k+1$:

- We partition P into three separate subsets titled:
  - $L = \{x \mid x < p\}$ (elements that are smaller than the pivot),
  - $C = \{x \mid x = p\}$ (elements that are equal to the pivot),
  - $H = \{x \mid x > p\}$ (elements that are greater than the pivot).

- By the inductive hypothesis, since $|L| \leq k$ and $|C| \leq k$, Quicksort has correctly sorted L and H

- Now, combing all subsets, we can produce the fully sorted array

Thus, by the principle of mathematical induction, Quicksort can correctly sort any array of size n(Knuth, 1998).

Loop Invariant for Partitioning

For the start of each iteration, the array is divided into three subarrays:

1. $P[l..i-1]$ which contains all elements less than the pivot $< p$.
2. $P[i..j-1]$, which contains all elements greater than or equal to the pivot $\geq p$.
3. $P[j..r]$ which contains elements not yet processed.

Initialization: Before the recursive call(the loop starts), all elements are considered unprocessed so the invariant holds

Maintenance: during each iteration if $A[j] < p$, swap $A[i]$ and $A[j]$, then increment we i to expand on the section of the smaller elements otherwise next element.

Termination: When $j = r$, all elements have been processed, meaning $A[l..i-1] < p$(contains all elements less than p), $A[i] = p$(is the pivot in its correct index), and

$A[i+1..r] > p$(contains elements greater than p.

This makes it so the partitioning is correctly implemented A and maintains the loop invariant.

Time Complexity Analysis

Recurrence Relation

QuickSort follows the recurrence described by the formula:

$$T(n) = T(k) + T(n-k-1) + O(n) \tag{1}$$

here k is the size of the partition and $O(n)$ accounts for partitioning in constant time.

On average(the average case), the pivot will split the array about in half, as shown in the formula(Sedgewick, 1977):

$$T(n) = 2T(n/2) + O(n) \tag{2}$$

Applying the Master Theorem ($T(n) = aT(n/b) + O(n^d)$ with $a = 2$, $b = 2$, $d = 1$):

$$O(n^{\log_2 2}) = O(n \log n) \tag{3}$$

Therefore, on average(the average-case), the time complexity is $O(n \log n)$.

Best and Worst Cases

- **Best Case:** $O(n \log n)$ when the pivot divides the array evenly.
- **Worst Case:** $O(n^2)$ when the pivot is always the smallest or largest element, leading to unbalanced recursion:

$$T(n) = T(n-1) + O(n) \Rightarrow O(n^2)(Hoare, 1961) \tag{4}$$

Opinion on Applicability

Quicksort is widely applicable due to its efficiency, adaptability, and ability to sort in place. However, certain things, such as the nature of the data and system constraints may limit its applicability.

Quicksort is most applicable in general-purpose sorting. It is the default sorting algorithm used in many of the most popular programming languages (C++, Python, Java) and is well suited for random or partially sorted data due to its O(n log n) runtime. It is applicable in large datasets due to its cache locality(Knuth, 1998). It has an edge over merge sort since it sorts in place and avoids O(n) extra space of something like merge sort. Finally, due to its recursive nature, we can employ parallel processing/multi-threading to improve its performance.

Quicksort is less applicable when a data set is already or nearly sorted; merge sort would be a better option here(Sedgewick, 1977). When data sets are small, insertion sort would be better. For linked lists or when stable sorting is required, a merge sort would be better.

References

Hoare, C. A. R. (1961). Algorithm 64: Quicksort. *Communications of the ACM, 4*(7), 321. https://doi.org/10.1145/366622.366644

Sedgewick, R. (1977). The analysis of Quicksort programs. *Acta Informatica, 7*(4), 327–355. https://sedgewick.io/wp-content/themes/sedgewick/papers/1977Analysis.pdf

Knuth, D. E. (1998). *The art of computer programming, Volume 3: Sorting and searching* (2nd ed.). Addison-Wesley.
http://lib.ysu.am/disciplines_bk/615bb516c351ea9ebfe76b40f8dbe730.pdf