QuickSort Algorithm Analysis

Justin Pelak

San Diego State University

CS 460-02

Professor Priya

March 2, 2025

QuickSort Algorithm Analysis

Problem Statement and Approach

Problem Statement

QuickSort is an efficient partition-based sorting algorithm developed by British computer scientist Tony Hoare in 1959. According to Hoare in an interview, the algorithm was conceived in lieu of bubble sort, as he believed there must be a faster method of sorting (Jones, 2016).

The QuickSort input is an array A that contains n elements. The result is a permutation of A, such that $A[0] \leq A[1] \leq \ldots \leq A[n-1]$.

A Quick Example

Suppose that we are given a random array of numbers *e.g.*,

$$A = [4,3,5,1,2]$$

using QuickSort on this array would result in

$$A = [1,2,3,4,5]$$

Preconditions

- A is an array containing comparable elements (*i.e.* support comparison operators, <, >, etc.)
- A is a finite size n where $n \geq 0$.

Our First Hoare Triple

- Precondition P: Given some array A with size $n \geq 0$,
- Statement S: calling `QuickSort(A,0,n-1)` on A
- Post-condition Q: results in $\texttt{low} \leq \texttt{high}$ and $A[\texttt{low..high}]$ is to be sorted

$\{n \geq 0\}$ $\texttt{QuickSort}(\texttt{A},0,\texttt{n}-1)$ $\{\texttt{low} \leq \texttt{high} \Rightarrow A[\texttt{low..high}]$ to be sorted$\}$

Algorithm Definition

QuickSort

QuickSort works by first selecting a **pivot** p. Multiple selection strategies exist, *e.g.* random pivot, median-of-three, etc. In this paper, we will take a look at middle element pivot, as it is one of the most straightforward approaches.

Once a pivot is selected, we partition all elements less than p to the left and all elements greater than p to the right. This allows for recursively calling QuickSort on both the left and right partitions until they are sorted or reach a size 0 or 1, which is inherently sorted.

Pseudocode

QuickSort

```
function QuickSort(A, LO, HI)
    if LO < HI
        p = (LO + HI) // 2 # Pivot becomes middle index
        i = Partition(A, LO, HI, p) # Splits current partition into two
        QuickSort(A, LO, i-1)
        QuickSort(A, i, HI)
```

Partition

```
function Partition(A, L, R, p)
    while L <= R
        while A[L] < p
            L++ # Shift left pointer right until element greater than p
        while A[R] > p
            R-- # Shift right pointer left until element less than p
        if L <= R
            swap(A[L], A[R])
            L++
            R--
    return L
```

Divide and Conquer Strategy

QuickSort uses the **Divide and Conquer** algorithm paradigm, which implies dividing, conquering, and combining steps. Those steps are used as follows,

1. Divide

The array A is first partitioned into two sides; one side with elements less than `p` and the other with elements greater than `p`. Say there are i elements less than `p` after the first partitioning, then the left partition will be $A[0..i-1]$, and the right partition will be $A[i..n-1]$. This division continues until there are no more unsorted elements in both partitions.

2. Conquer

The algorithm conquers the problem by iteratively sorting through all partitions using the `Partition` function and recursively creating partitions through the `QuickSort` subroutine.

3. Combine

Notice that `QuickSort` is sorted **in-place**, therefore, there is no "true" combine step like there might be for Merge Sort where all sub-arrays are recombined. Rather, QuickSort's result is based on the aggregate of the `Partition` function calls.

Proving Correctness

Correctness Proof by Induction

To prove that QuickSort sorts an array of n correctly, we will use mathematical induction. First, consider the base case.

Base Case

When $n \leq 1$, the partition is inherently sorted; thus, no further action is required. Therefore, `QuickSort` is correct for $n \leq 1$.

Inductive Hypothesis

Assume that `QuickSort` will correctly sort a partition of at most size k where $k \geq 1$.

Inductive Step

We then need to prove that it will correctly sort a partition of size $k+1$. Assume for now that `Partition` functions properly.

Then the array

$$A[\texttt{LO..HI}]$$

is partitioned into

$$A[\texttt{LO..i-1}] \text{ and } A[\texttt{i..HI}]$$

where, recalling from the algorithm, `i` is the index where the indices cross.

Now the sizes of these two partitions are $\texttt{i} - \texttt{LO}$ and $\texttt{HI} - \texttt{i} + 1$, respectively. It suffices to then say that $\texttt{i} - \texttt{LO} \leq k$ and $\texttt{HI} - \texttt{i} + 1 \leq k$.

According to the inductive hypothesis, `QuickSort` will correctly sort the partitions with size at most k, therefore, the correctness of the algorithm is placed on the correctness of our `Partition` function, and we revoke our previous assumption of its correctness.

The Second Hoare Triple

The loop invariant of the `Partition` function maintains that at the beginning of each iteration, the indices `LO` and `HI` are within the bounds $[\texttt{LO} \leq \texttt{HI}]$ and $\texttt{p} = \left\lfloor \frac{\texttt{LO+HI}}{2} \right\rfloor$

- **Initialization:** The subroutine `QuickSort` is defined such that we can guarantee that only when $\texttt{LO} < \texttt{HI}$ is satisfied, will the function `Partition` be called. `p` is defined such that it is always the midpoint between `LO` and `HI`, so $\texttt{p} \in A[\texttt{LO..HI}]$ must be true.

- **Maintenance:** At every iteration, the left pointer `L` moves right until an element greater than `p` is found. Then, the right pointer `R` moves left until an element less than `p` is found.

  Note: In this case, equal values may end up in either the right or left partition.

- **Termination:** Upon termination, we guarantee there is a position `i` between `LO` and `HI` such that all elements from `LO` to `i-1` are $\leq \texttt{p}$ and all elements from `p` to `HI` are $\geq \texttt{p}$.

All parts put together in a logically expressed Hoare triple gives us

$$\left\{\mathtt{LO} \leq \mathtt{HI},\ \mathtt{p} = \left\lfloor \frac{\mathtt{LO}+\mathtt{HI}}{2} \right\rfloor \right\}\ \mathtt{Partition(A,LO,HI,p)}$$

$$\left\{\exists \mathtt{i} \in [\mathtt{LO..HI}] : \left[\forall m \in \mathtt{A}[\mathtt{LO..i}-1],\ \mathtt{A}[m] \leq \mathtt{p} \wedge \forall n \in \mathtt{A}[\mathtt{i..HI}],\ \mathtt{A}[n] \geq \mathtt{p}\right]\right\}$$

and since the loop invariant holds for all three parts, the `QuickSort` function is correct. Combined with the inductive step, we conclude that the QuickSort algorithm is correct for all $n \geq 0$.

Analyzing Time Complexity

Deriving the Recurrence Relation

To derive the time complexity of QuickSort with the **recurrence tree method**, we first need to understand what makes it run efficiently and what throws a wrench into it. Suppose every time we recursively choose our pivot, we select the smallest or largest element in the array; our partitions would be maximally **unbalanced**. On the other hand, if every time we select a pivot, we roughly land at the **median** of the array, we would be nearly perfectly balanced. Figure 1 shows this best.

QuickSort Balances

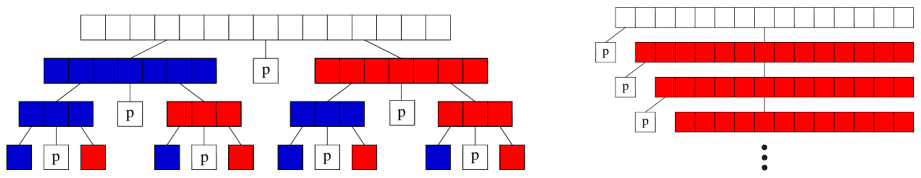

Figure 1

Note. Worst pivot selection displayed on the right, best pivot selection displayed on the left. From *The Algorithm Design Manual* (3rd ed., p.132), Skiena, 2020.

If we always select the optimal pivot, we can see that our problem is divided into subproblems of size $\frac{n}{2}$ twice. One half for the smaller elements, one half for the largest elements. If we always select the worst elements, we are only effectively "removing" 1 element from the next subproblem. In other words, our singular subproblem has size $n-1$. With this in mind, we can finally solve for the best and worst case time complexities of QuickSort.

Best Case

If we consistently divide our subproblems into near-equal halves, our recurrence relation can be described as $T(n) = 2T(\frac{n}{2}) + O(n)$, where $2T(\frac{n}{2})$ describes the recursive partitioning of subproblems, and $O(n)$ describes the amount of work done at each recursive level.

The recurrence tree, then, for the size of each partition (and thus work done at each level) looks like:

- **Level 0:** n
- **Level 1:** $\frac{n}{2}$
- **Level 2:** $\frac{n}{4}$

$\vdots$

- **Level k:** $\frac{n}{2^k}$

Note: Keep in mind that while each partition *size* is split by 2^k, there are $k+1$ partitions to work on, so the *total* work done is still n.

Because we continue partitioning in half until reaching the base case, we can write this as the equation

$$\frac{n}{2^k} = 1$$

Solving for k (because k represents levels until base case),

$$\frac{n}{2^k} = 1$$

$$n = 2^k$$

$$\log_2 n = \log_2 2^k \Rightarrow \boxed{\log_2 n = k}$$

Now, as we see, there are $\log n$ levels, and as aforementioned, the amount of work done at each is $O(n)$, so the total time complexity for the best case of QuickSort is $\Omega(n \log n)$.

Worst Case

The worst case occurs when the partitions are maximally unbalanced; that is, it occurs when only one element (the pivot) is partitioned out when partitioning the array. For example, in a size n array, if we select the largest element, the left partition contains $n-1$ elements, and if this continues, we will continue down yet another tree:

- **Level 0:** $T(n)$ does n units of work
- **Level 1:** $T(n)$ does $n-1$ units of work
- **Level 2:** $T(n)$ does $n-2$ units of work

$\vdots$

- **Level n:** $T(n)$ does 1 unit of work

Expressed mathematically, this tree looks like:

$$\begin{aligned} T(n) &= T(n-1)+n \\ T(n) &= T(n-2)+(n-1)+n \\ &\vdots \\ T(n) &= n+(n-1)+(n-2)+\cdots+1 \end{aligned}$$

This is an arithmetic series taking the form $(x-1)+(x-2)+\cdots+(x-(x-1))$ which can be represented as $\frac{n(a_1+a_n)}{2}$ where n is the number of terms added (in our case, also n) and a_1, a_n represent the first and last terms, respectively. So, in our case, we have $\frac{n(n+1)}{2}$.

Simplifying this down, we get $\frac{n^2+n}{2} \Rightarrow \frac{n^2}{2}+\frac{n}{2}$. Removing all constants and only considering the term of largest order, we get the final worst case of $O(n^2)$, and our time complexity analysis is complete.

Opinion on Applicability of QuickSort

Most Applicable

QuickSort seems to be a very good sorting algorithm for large, random datasets of primitive types. In datasets that are large and random, the chance of selecting one of the two worst elements out of your entire dataset is very low. The chance of doing that *every* time you select a pivot is even lower, which is why the average case works out to $O(n \log n)$. In fact, digging into what people regularly use, you will find QuickSort is certainly a dominant figure in the sorting world. For example, C, C++, and Java (and likely more) all use some variation of QuickSort in their standard library sorting functions.

QuickSort also works incredibly well when used in conjunction with other sorting algorithms, forming hybrid algorithms. However, scenarios where QuickSort is not an ideal sorting algorithm is when sorting objects where relative order matters. In other words, QuickSort is an **unstable** sorting algorithm, and while this *typically* doesn't matter, in cases where it *does* matter, this absolutely needs to be considered.

QuickSort, as previously mentioned, also doesn't work well with non-primitive types because QuickSort banks on being able to directly access and compare arbitrary positions in $O(1)$ time. So, sorting something like a Linked List is inefficient because partitioning would be much slower, as accessing and swapping some i-th node is not done in $O(1)$ time.

References

Jones, C. (2016, October). *Tony hoare, 1980 acm turing award recipient* [Interview conducted on November 24, 2015. YouTube video uploaded by ACM]. https://www.youtube.com/watch?v=tAl6wzDTrJA

Skiena, S. S. (2020, October). *The algorithm design manual* (3rd ed.). Springer Charm. https://doi.org/https://doi.org/10.1007/978-3-030-54256-6