# Randomized Quick sort

## Introduction to Quick sort

Quicksort, introduced by C.A.R. Hoare in 1962, stands as a foundational and efficient sorting algorithm in the realm of computer science. At its core, Quicksort employs a divide-and-conquer strategy, epitomizing elegance in algorithmic design.

The algorithm begins by selecting a pivot element from the array. Subsequently, the array is partitioned into two segments – elements smaller than the pivot and elements greater than the pivot. Recursive application of this process to the subarrays leads to a cascade effect, ultimately sorting the entire array.

Quicksort's efficiency lies in its average-case time complexity of O (n lg n), making it particularly well-suited for sorting large datasets.

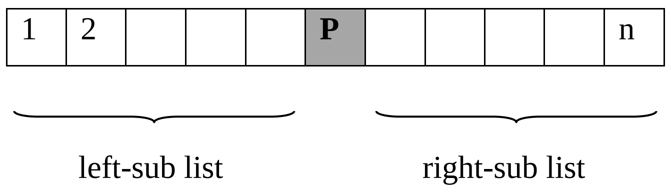


*Fig 1: Divide-and-Conquer in quick sort*

Here, the array is divided into two subarrays: one from A[low] to P-1 (the elements smaller than the pivot), and another from P+1 to some endpoint (the elements greater than the pivot).

## Algorithmic View of Quick sort

The pseudo-code for recursively defining the quick sort is very simple and is given below:

```
QuickSort (A, first, last)

1. If (first < last) then
2.         p ⟵ Partition (A, first, last)
3.         QuickSort (A, first, p-1)
4.         QuickSort (A, p+1, last)
5. Return
```

## Partitioning the array

The key to the algorithm is the PARTITION procedure, which rearranges the subarray A [first…last] in place.

```
PARTITION (A, first, last)
1. pivot ⟵ A[last]
2. i ⟵ first -1
3. for j ⟵ first to last-1
4.         do if A[j] ≤ pivot
5.                 then i ⟵ i+1
6.                     exchange A[i] ↔ A[j]
7. exchange A[i+1] ↔ A[last]
8. Return i+1
```

Note that, the partition function presented in the provided pseudocode does not align with the original partition function introduced by C.A.R. Hoare in the original version of the Quicksort algorithm. The partitioning mechanism described in the pseudocode, where the rightmost element is chosen as the pivot and elements are rearranged accordingly, is commonly referred to as the Lomuto partition scheme. It should be noted that the Hoare partition scheme, as proposed by Hoare himself, differs in its approach.

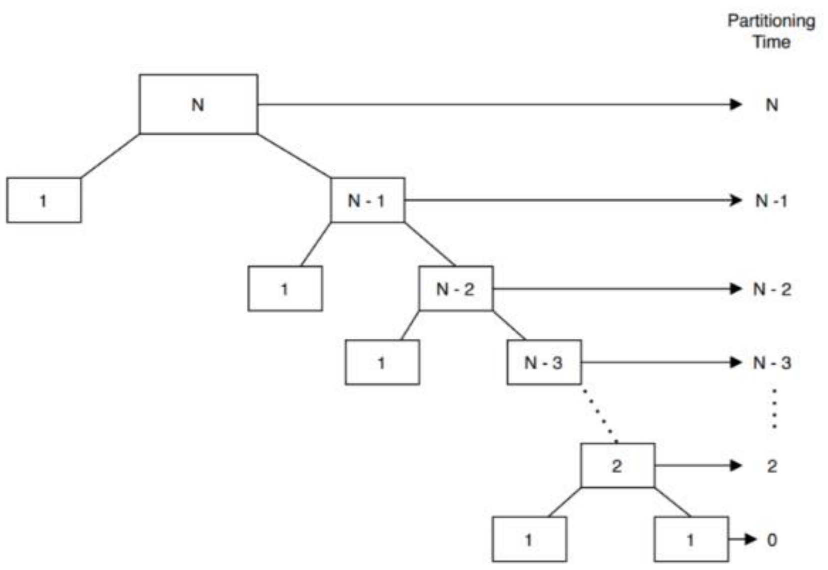
In the Hoare partition scheme, two indices traverse the array inwards from both ends, swapping elements that are out of place with respect to the pivot. This process continues until the indices meet, at which point the pivot is placed in its final sorted position.

## Need for Randomized Quick sort

Let assume an Array A,

| | | | | | | |
|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |

Quick sort of A will generate a tree structure like this:



Let’s consider an input array of size N. The first partition call takes N times to perform the partition step on the input array.

Each partition step is invoked recursively from the previous one. Given that, we can take the complexity of each partition call and sum them up to get our total complexity of the Quicksort algorithm.

**Therefore, the time complexity of the Quicksort algorithm in worst case is**

$$[\mathbf{N} + (\mathbf{N}-\mathbf{1}) + (\mathbf{N}-\mathbf{2}) + (\mathbf{N}-\mathbf{3}) + ..... + \mathbf{2}] = [\frac{\mathbf{N(N+1)}}{\mathbf{2}} - \mathbf{1}] = \mathcal{O}(\mathbf{N^2})$$

There are two ways to randomize the quicksort –

- **Randomly shuffling the inputs**: Randomization is done on the input list so that the sorted input is jumbled again which reduces the time complexity. However, this is not usually performed in the randomized quick sort.
- **Randomly choosing the pivot element**: Making the pivot element a random variable is commonly used method in the randomized quick sort. Here, even if the input is sorted, the pivot is chosen randomly so the worst-case time complexity is avoided.

## Randomized Quick sort

Randomized Quicksort is an enhanced version of the classic Quicksort algorithm that incorporates an element of randomness in its pivot selection process. Originally introduced by Robert Sedgewick in 1978, this variant seeks to mitigate the potential worst-case time complexity scenarios that deterministic pivot selection strategies might encounter. The primary goal of Randomized Quicksort is to achieve improved average-case performance by distributing input data more evenly during the partitioning steps.

## Algorithmic View of Randomized Quick sort

Instead of always using a A[last] as the pivot, we will use a randomly chosen element from the subarray A [first … last]. We do so by exchanging element A[last] with the randomly chosen element. This modification, in which we randomly sample the range first, …, last, ensures that the pivot element pivot = A[last] is equally likely to be any of the r-p+1 element in the sub array. Because the pivot element is randomly chosen, we expect the split of the input array to be reasonably well balanced on average. We simply implemented the swap before the actual partitioning:

```
RANDOMIZED-Partition (A, first, last)

1. i ⟵ RANDOM (first, last)
2. exchange A[last] ↔ A[i]
3. Return Partition (A, first, last)
```

The new quicksort calls RANDOMIZED-Partition in place of Partition:

```
RANDOMIZED-QUICKSORT (A, first, last)

1. If (first < last) then
2.         p ⟵ RANDOMIZED-Partition (A, first, last)
3.         RANDOMIZED-QUICKSORT (A, first, p-1)
4.         RANDOMIZED-QUICKSORT (A, p+1, last)
5. Return
```

## “Expected” Running Time Randomized Quick sort

The expected running time of a randomized algorithm is a well-defined concept, just like the worst-case running time. If an algorithm is randomized, its running time is also random, which means we can define the expected value of its running time.

In the randomized quick sort, if, in each level of recursion, the split induced by RANDOMIZED-Partition puts any constant fraction of the elements on one side of the partition, then the recursion tree has depth Φ (lg n), and O(n) work is performed at each level. Even if we add a few new levels with the most unbalanced split possible between these levels, the total time remains O (n lg n). We can analyze the expected running time of RANDOMIZED-QUICKSORT precisely by first understanding how the partitioning procedure operates and then using this understanding to derive an O (n lg n) bound on the expected running time. This upper bound on the expected running time, combined with the Φ (n lg n) best-case bound, yields a Φ (n lg n) expected running time. We assume throughout that the values of the elements being sorted are distinct.

## Lemma

**Let X be the number of comparisons performed in PARTITION over the entire execution of QUICKSORT on an n-element array. Then the running time of QUICKSORT is O (n +X).**

proof:

Our goal, therefore, is to compute X, the total number of comparisons performed in all calls to PARTITION. We will not attempt to analyze how many comparisons are made in each call to PARTITION. Rather, we will derive an overall bound on the total number of comparisons. To do so, we must understand when the algorithm compares two elements of the array and when it does not. For ease of analysis, we rename the elements of the array A as $Z_1, Z_2, \ldots, Z_n$ with $Z_i$ being the ith smallest element. We also define the set $Z_{ij} = \{Z_i, Z_{i+1}, \ldots, Z_j\}$ to be the set of elements between $Z_i$ and $Z_j$, inclusive.

we observe that each pair of elements is compared at most once. Because elements are compared only to the pivot element and, after a particular call of PARTITION finishes, the pivot element used in that call is never again compared to any other elements.

Our analysis uses indicator random variables. We define

$X_{ij}$ = 1 {Zi is compared to Zj},

where we are considering whether the comparison takes place at any time during the execution of the algorithm, not just during one iteration or one call of PARTITION. The definition of $X_{ij}$ seems to indicate that it takes a value of 1 if element $Z_i$ is ever compared to element $Z_j$ during the algorithm's execution, and 0 otherwise. Since each pair is compared at most once, we can easily characterize the total number of comparisons performed by the algorithm:

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij} .$$

Taking expectations of both sides, and then using linearity of expectation, we obtain

$$\begin{aligned} \mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right] \\ &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathrm{E}[X_{ij}] \\ &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr\{z_i \text{ is compared to } z_j\} . \end{aligned}$$

eq.1

It remains to compute Pr {Zi is compared to Zj}, Our analysis assumes that the RANDOMIZED-PARTITION procedure chooses each pivot randomly and independently.

Let us think about when two items are not compared. Consider an input to quicksort of the numbers 1 through 10 (in any order), and suppose that the first pivot element is 7. Then the first call to PARTITION separates the numbers into two sets: f1; 2; 3; 4; 5; 6g and f8; 9; 10g. In doing so, the pivot element 7 is compared to all other elements, but no number from the first set (e.g., 2) is or ever will be compared to any number from the second set (e.g., 9).

In general, once a pivot x is chosen with $Z_i < x < Z_j$, we know that $Z_i$ and $Z_j$ cannot be compared at any subsequent time. If, on the other hand, Zi is chosen as a pivot before another item in $Z_{ij}$, then $Z_i$ will be compared to each item in $Z_{ij}$, except for itself. Similarly, if $Z_j$ is chosen as a pivot before any other item in $Z_{ij}$, then $Z_j$ will be compared to each item in $Z_{ij}$, except for itself. In our example, the values 7 and 9 are compared because 7 is the first item from $Z_{7,9}$ to be chosen as a pivot. In contrast, 2 and 9 will never be compared.

We now compute the probability that this event occurs. Prior to the point at which an element from $Z_{ij}$ has been chosen as a pivot, the whole set $Z_{ij}$ is together in the same partition. Therefore, any element of $Z_{ij}$ is equally likely to be the first one chosen as a pivot. Because the set $Z_{ij}$ has j – i+1 elements, and because pivots are chosen randomly and independently, the probability that any given element is the first one chosen as a pivot is 1/ (j- i+1). Thus, we have

Pr {Zi is compared to Zj} = Pr {Zi or Zj is the first pivot chosen from Zij}

= Pr {Zi is the first pivot chosen from Zij}

+ Pr {Zj is the first pivot chosen from Zij}

= 1/ (j- i+1) + 1/ (j- i+1)

= 2/ (j- i+1) eq.2

The second line follows because the two events are mutually exclusive. Combining equations (eq.1) and (eq.2), we get that

$$\mathrm{E}[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} .$$

We can evaluate this sum using a change of variables (k = j - i) and the bound on the harmonic series in equation:

$$\begin{aligned}
\mathrm{E}[X] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \\
&= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} \\
&< \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k} \\
&= \sum_{i=1}^{n-1} O(\lg n) \\
&= O(n \lg n) .
\end{aligned}$$

eq .3

also,

We use the same reasoning for the expected number of comparisons, we just take in a different direction.

$$\begin{aligned}
\mathrm{E}[X] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \\
&= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} \qquad\qquad (k \geq 1) \\
&\geq \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{2k} \\
&\geq \sum_{i=1}^{n-1} \Omega(\lg n) \\
&= \Omega(n \lg n).
\end{aligned}$$

Using the master method, we get the solution Θ(*n*lg*n*).

## Disadvantages of Randomized Quick sort

From the above algorithm on randomized quick sort, we can say that mostly the algorithm is advantageous but there are certain drawbacks. In this algorithm there are mainly 2 disadvantages:

1. The algorithm is not based on the assumption that all elements are distinct. If there is an array in which all elements are the same, it may still result in a worst-case time complexity of O(N^2) for randomized quicksort, especially if the pivot selection strategy consistently chooses the smallest or largest element in each partition. However, the randomization in the algorithm helps mitigate this risk, making such worst-case scenarios less likely in practice.

| | | | | |
|---|---|---|---|---|
| 4 | 4 | 4 | 4 | 4 |

In this kind of array, the time complexity will be O(n^2).

2. In this algorithm, the pivot is chosen randomly between the first to last index, which is swapped with the first element. But if the randomly chosen pivot happens to be the

index value where all the elements are already sorted, then it will give the same worst time complexity as quicksort.

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 |

In this case, if the pivot is chosen in the way of the index position in the first case, then it will be swapped with itself. Consequently, the comparisons will be N-1. Similarly, in the second case, if 1 index position is chosen, the array will remain in a sorted order, and all elements will have to perform N-1 comparisons. This situation will result in a time complexity of O(N^2).

## MEDIAN – 3 PARTITIONS

This is an improvised method of quicksort that guarantees an average time complexity of O (N log N). Let's take an example:

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 12 | 8 | 10 | 2 | 26 | 4 | 7 | 5 | 3 | 16 | 11 | 6 | 13 | 19 |
| ⬆ | | | | | | ⬆ | | | | | | | ⬆ |
| FIRST | | | | | | MIDDLE | | | | | | | LAST |

1. firstly, sort the FIRST, MIDDLE, and LAST element.

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 7 | 8 | 10 | 2 | 26 | 4 | 12 | 5 | 3 | 16 | 11 | 6 | 13 | 19 |
| ⬆ | | | | | | ⬆ | | | | | | | ⬆ |
| FIRST | | | | | | MIDDLE | | | | | | | LAST |

2. Swap the MIDDLE element with the second last element and make it pivot.

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 7 | 8 | 10 | 2 | 26 | 4 | 12 | 5 | 3 | 16 | 11 | 6 | 13 | 19 |
| | | | | | | ⬆ | | | | | | ⬆ | |

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 7 | 8 | 10 | 2 | 26 | 4 | 13 | 5 | 3 | 16 | 11 | 6 | 12 | 19 |
|  |  |  |  |  |  | ⬆ |  |  |  |  |  | ⬆ |  |
|  |  |  |  |  |  |  |  |  |  |  |  | PIVOT |  |

3. Set up the i and j counters, ignoring the FIRST and LAST element for comparison, as we know the first element is smaller than the PIVOT, and the last one is greater than the PIVOT.

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 7 | 8 | 10 | 2 | 26 | 4 | 13 | 5 | 3 | 16 | 11 | 6 | 12 | 19 |
|  | I |  |  |  |  |  |  |  |  |  |  | J |  |

Now compare each element of i with the PIVOT, and if any element is greater than the PIVOT, then stop there; otherwise, increment i. Next, compare each element of j with the PIVOT, and if any element is lesser than the PIVOT, then stop there; otherwise, increment j. When both the conditions of i and j are satisfied, swap the values of i and j.

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 7 | 8 | 10 | 2 | 26 | 4 | 13 | 5 | 3 | 16 | 11 | 6 | 12 | 19 |
|  |  |  | **i>12** |  |  |  |  |  |  | **j<12** |  |  |  |

Values 6 and 26 swapped.

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 7 | 8 | 10 | 2 | 6 | 4 | 13 | 5 | 3 | 16 | 11 | 26 | 12 | 19 |
|  |  |  |  | i |  |  |  |  |  |  |  | j |  |

Then in the next case, 13 and 11 swapped.

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 7 | 8 | 10 | 2 | 6 | 4 | 11 | 5 | 3 | 16 | 13 | 26 | 12 | 19 |
|  |  |  |  |  |  | i |  |  |  | j |  |  |  |

In the next case, i and j cross each other, then the value of i is swapped with the PIVOT.

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 7 | 8 | 10 | 2 | 6 | 4 | 11 | 5 | 3 | 16 | 13 | 26 | 12 | 19 |
|  |  |  |  |  |  |  |  | j | i |  |  |  |  |

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 7 | 8 | 10 | 2 | 6 | 4 | 11 | 5 | 3 | 12 | 13 | 26 | 16 | 19 |

Now the array is divided into two parts: from 0 to 8 index positions, which are less than 12, and the other part from 10 to 13, greater than 12. Repeat these steps for each subarray until the entire array is sorted.

This method guarantees O (N log N) time complexity as the PIVOT is always the middle element, and there are always elements on both sides of the PIVOT. However, there is a slight increase in time for the selection of the pivot due to the sorting of the first, middle, and last elements.

## REFERENCES

Cormen, T. H., Leiserson, C. E., & Rivest, R. L. (1990). Introduction to algorithms. MIT Press ; McGraw-Hill.

Samanta, D. (2008). *Classic Data Structures* (2nd ed.). PHI Learning Pvt. Ltd.

Califf, M. E. (2021, feb 12). *Improving Quicksort with Median of 3 and Cutoffs* [Video]. https://youtu.be/1Vl2TB7DoAM?si=eInHG_SkuL27uetS

GATEHUB. (2022, feb 12). *Randomized Quicksort Algorithm | Divide and Conquer | GATECSE | DAA* [Video]. https://youtu.be/nUlEfx-HgII?si=nIPZTb6uiUQcRkvP

NPTEL-NOC IITM. (2022, oct 11). *Randomized Quicksort* [Video]. https://youtu.be/hOIZpznZWMw?si=_HpmSCh_hw8izLye