Quicksort: How Efficient Algorithms can Utilise Randomness

A case-study for randomised algorithms

Augustus Jonikas
Y12 CSSOC member

This article will feature a deep-dive of a randomised sorting algorithm, including a detailed description of how it and its subroutines work, a comprehensive runtime analysis, and a description of its space complexity. It utilises a 'divide and conquer' approach to algorithmic design, allowing it to run in $O(n \log n)$ time.

Many people do not associate randomness with efficient algorithms, and for good reason. An algorithm's precisely-defined and consistent nature, which is tangential to the concept of randomness. However, many efficient algorithms use randomness in order to be efficient. The rest of this text will explain how one such algorithm, randomised quicksort, does so.

THE ALGORITHM

As the name suggests, randomised quicksort is a sorting algorithm: given an input of a list of data, it returns the list in sorted order. There are many other sorting algorithms you are likely aware of, but quicksort's blazingly fast $O(n \log n)$ running time puts many of these to shame. With the best possible running time for a comparison-based sorting algorithm, how can such an algorithm achieve such feats by embracing randomness?

Defining the partition subroutine

The quicksort algorithm works using a subroutine that takes in an array and *partition* it around a given pivot, which will have a random position in the sorted array. A partitioned array is an array where all elements less than the pivot are to the left of the pivot, and all elements greater than the pivot are to the right of the pivot.

More formally, a partitioned array is an array where for any element e and pivot p, the following set of rules hold:

- if $e < p$ then $i_e < i_p$
- if $e > p$ then $i_e > i_p$

where e_i and p_i are the indexes of i_e and i_p in the array respectively.

We are currently ignoring the possibility of duplicates in the array. Adapting the subroutine to account for duplicates is left as an exercise for the reader. Note that this method of choosing the pivot has a random nature - there is no way of determining or guess where the pivot will go after the partition.

```
function Partition (A);
Input  : Array A
Output : Partitioned array
i_p  ← |A| − 1;
i_p' ← 0;
p    ← A[i_p];
for i ← 0 to |A| − 1 do
    e_i ← A[i];
    if e_i < p then
        A[i] ↔ A[p_i'];
        p_i' ← p_i' + 1
    end
end
A[p_i'] ↔ A[p_i]
```

where $i_p{}'$ is the index of the pivot in the partitioned array, and operation $a \leftrightarrow b$ reads 'swap a and b'.

The function selects a pivot p to be the last element with index $|A| - 1$, then separates the values that are less than the pivot from those that are greater than the pivot. Afterwards it inserts the pivot in between the boundary of the two groups.

To further explain this, let's define a sample input.

2	7	3	1	8	5	6	4

This array contains the numbers 1-8 in a completely unsorted order, and will be used as an example input for the explanation of this algorithm. Here is the result generated after the first call of the partition subroutine.

2	3	1	4	8	5	6	7

As you can see, the array satisfies the rules for a partitioned array stated above. With '4' being selected as the pivot, all elements less than 4 are to the left of 4, and all the elements that are greater than 4 are to the right of 4.

Note that the pivot is now in its correct position, and the left and right sub-arrays are not yet sorted. We can sort these sub-arrays by making two recursive calls.

Defining the randomised quicksort algorithm

function Quicksort (A);
Input : Unsorted array A
Output : Sorted array
if $|A| \leq 1$ **then**
  return;
else
  Partition (A);
  Quicksort (left sub-array);
  Quicksort (right sub-array);
end

We will dry-run the quicksort algorithm on the array below. Note that we will not go into the recursive calls, since that is not needed to show their roles in the algorithm.

Starting array:

2	7	3	1	8	5	6	4

Algorithm Q (randomised quicksort)

Q1. Check if the array has size of 1 or 0. The array has a size of 8, so continue on with the function.

Q2. Partition the array around the last element of the array.

2	3	1	4	8	5	6	7

Q3. Sort the left sub-array using a recursive call.

1	2	3	4	8	5	6	7

Q4. Sort the right sub-array using a recursive call.

1	2	3	4	5	6	7	8

The array has now been sorted.

ANALYSIS

In the analysis of the algorithm, we will cover two things:

- A proof of correctness for randomised quicksort.
- A proof for its $O(n \log n)$ running time.

Proof of correctness

To understand this proof of correctness, you must first be familiar with proof by induction. I'll cover the basics below, but please feel free to skip this part if you're already familiar with this method of proof.

Introductory proof by induction

A proof of induction proves an assertion using two parts: proof of a valid 'base case' and the proof of all above numbers, given that all values below are proved.

Example assertion:

$$\text{Given a function } f(x) = 1 + 2 + 3 + \cdots + x$$
$$f(x) = \frac{x(x+1)}{2} \quad \text{for all positive integers}$$

Algorithm I (proof by induction)

I1. Prove a base case:

$\because$ if we let $x = 1$, $f(x) = 1$, $x(x+1)/2 = 1$.
$\therefore$ $f(x) = x(x+1)/2$ when $x = 1$.

I2. Prove for any number, assuming all numbers between it and the base case have been proved.

Assume that $f(x) = x(x+1)/2$.

$$\begin{aligned} f(x+1) &= f(x) + (x+1) \\ &= \frac{x(x+1)}{2} + (x+1) \\ &= \frac{x(x+1)}{2} + \frac{2(x+1)}{2} \\ &= \frac{(x+1)(x+2)}{2} \end{aligned}$$

$\therefore$ if $f(x) = x(x+1)/2$, then $f(x+1) = (x+1)(x+2)/2$.

From this, we've proved that the statement is true for $x = 1$, and for the integer after any valid number. This means that it's true for 2 - the integer after the valid number 1, and so on.

Proof of correctness for randomised quicksort

Assertion: randomised quicksort correctly sorts every input array of length n.

I1. If the array contains only $n = 1$ element, the array must be sorted, as there are no elements the present element can be compared to. Since quicksort makes no changes to the input array when $n = 1$, it returns the sorted version of this array.

I2. Given that quicksort works for all input sizes k where $k < n$, we need to prove that it works for input size n.

After partitioning the array through a pivot p, the array will be in three parts.

- The left sub-array, where all values are less than p
- The right sub-array, where all values are greater than p
- p itself

left sub-array	p	right sub-array

Since p has been placed in its sorted placement by virtue of it being the pivot, no more sorting is done on this element. The remaining sub-arrays are unsorted. If these two sub-arrays are sorted, then the whole array will be sorted, since all elements in the left sub-array belongs to the left of p and all the elements in the right sub-array belong to the right of p.

The size of each sub-array is at most $n - 1$ when p is either the greatest or smallest element in the array, which is a valid value of k. This means that they can both be sorted, since it's been given that all arrays with a size in k can be sorted with quicksort.

That concludes the inductive proof of randomised quicksort. Note that we did not discuss how the pivot is chosen, since any method that chooses an element of the array will be correct. However, it does impact the running time. Below we will show that the best running time $O(n \log n)$ can be achieved by randomly picking a pivot.

Running-time analysis

When analysing the running time of quicksort with random pivots, we will be using some concepts from probability analysis.

- Finite sample spaces
- Random variables
- Expected values
- Linearity of expectation

Introductory probability analysis

The *expected value* of a discrete random variable is essentially its mean. For a random variable X, consider the set O as the set of observations taken of X. As the size of set O increases by taking more and more observations of X, the mean of the values in set O approaches the expected value of X. The expected value of X would be denoted $\mathrm{E}(X)$.

The *expected value* for X can be defined as $\sum xP(X = x)$. For all values that X can take, multiply the value with the probability that X takes the value in an observation, then add up all these products.

Linearity of expectation refers to the property that the sum of expected values for a group of random variables is equal to the expected value of the sum of the random variables. For example, for random variables X, Y and Z.

$$\mathrm{E}(X + Y + Z) = \mathrm{E}(X) + \mathrm{E}(Y) + \mathrm{E}(Z)$$

More generally, for a list of random variables $X_1, X_2, X_3, \ldots, X_m$

$$\mathrm{E}\left(\sum_{n=1}^{m} X_n\right) = \sum_{n=1}^{m} \mathrm{E}(X_n)$$

With the required background information covered, we can now start with the runtime analysis of the algorithm.

Running-time analysis for randomised quicksort

First, let's define a few variables.

- Let A be the input array of length n.
- Let Ω be a random variable with a sample space of all the sequences of pivots that could be chosen in the running of the algorithm.
- Let $\sigma \in \Omega$ be a particular sequence of pivots the algorithm may choose.
- Let $C(\sigma)$ be the number of comparisons made when the algorithm runs by choosing pivot sequence σ.

  The running time of quicksort is dominated by the total number of comparisons it makes, since that determines how many actions the algorithm performs on the array. Therefore, the running time of quicksort is $\mathrm{E}(C)$, for all possible inputs to C.

- Let z_n be the nth smallest element in A. For example, 3 is the third smallest element in the example input, meaning it is denoted as z_3. z_n should end up in the nth element of the output array.

- Let $X_{ij}(\sigma)$ be the number of times the element z_i and z_j are compared for any given pivot sequence. For example $X_{36}(\sigma)$ is the number of times z_3 and z_6 are compared in the running of the algorithm when the algorithm chooses pivot sequence σ.

  It's important to note that $X_{ij}(\sigma)$ cannot be more than 1. The only part of the algorithm in which two elements are compared is in the partition subroutine, and they are compared only if the elements at i or j are chosen as the pivot. After this comparison, the quicksort function will never again be passed with both the pivot and the other element, since no element can be selected as a pivot more than once. $X_{ij}(\sigma)$ may be 0 if a pivot other than z_i or z_j is chosen and the elements are separated into different sides of the pivot, they will not be compared at all.

We have now defined two variables C and X. C can be expressed as the sum of all values of X for all possible pairs of i and j. More formally, the following is true.

$$C(\sigma) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}(\sigma)$$

i can never be equal to n, since j in that scenario would not have a valid value, since $i < j \leq n$. Similarly, j can never be 1 since i would not have a valid value.

Now that we have expressed C in terms of a simpler random variable X, we can apply *linearity of expectation*, since the right hand side of the equation above can be thought of as a list of all possible permutations of X. Therefore, the below equation can be derived.

$$\mathrm{E}(C) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathrm{E}(X_{ij})$$

As discussed before, X can only take values 0 or 1.

$$\begin{aligned} \mathrm{E}(X_{ij}) &= 0 \times P(X_{ij} = 0) + 1 \times P(X_{ij} = 1) \\ &= P(X_{ij} = 1) \end{aligned}$$

Substitute this into the right side of the first equation.

$$\mathrm{E}(C) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} P(X_{ij} = 1)$$

By analysing the right hand side of this equation, we will be able to determine the overall running time of the algorithm, since $\mathrm{E}(C)$ dominates it.

To evaluate this, let's first consider the set $S = \{z_i, z_{i+1}, \ldots, z_{j-1}, z_j\}$, considing all elements with values between z_i and z_j inclusive, set S has size $j - i + 1$.

It should be apparent that if the chosen pivot is not in S, then the whole set will be passed into the same recursive call. However, this case cannot happen infinitely, as the minimum size of the set is 2 when the set only consist of z_i and z_j, and a pivot will always be chosen until the input array is empty or has a size of one.

Therefore, at least one element from S must be chosen as a pivot. If z_i or z_j are chosen, then they get compared. If the chosen pivot is between z_i and z_j, they get split into different sub-arrays and thus not compared.

The probability of z_i and z_j being compared is equal to

$$P(z_i \text{ or } z_j \text{ get chosen as the pivot} \mid \text{an element of } S \text{ is chosen as the pivot})$$

Since this is the exact definition for X_{ij}, we can now write the following.

$$\mathrm{E}(X_{ij}) = \frac{2}{j-i+1}$$

Furthermore, by substituting this into the equation.

$$\begin{aligned}\mathrm{E}(C) &= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n}\frac{2}{j-i+1} \\ &= 2\sum_{i=1}^{n-1}\sum_{j=i+1}^{n}\frac{1}{j-i+1}\end{aligned}$$

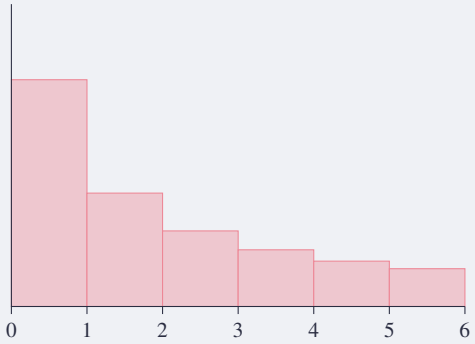
Firstly, let's focus on the inner summation, each iteration is $1/2 + 1/3 + 1/4 + \ldots$. The largest values of this summation is when $i = 1$, then the last term of the summation will be $1/n$, where each subsequent iteration will have smaller values than the ones before. We can upper cound each iteration to be equal to the value of the greatest iteration when $i = 1$.

$$\sum_{j=i+1}^{n}\frac{1}{j-i+1} \leq \sum_{k=2}^{n}\frac{1}{k}$$

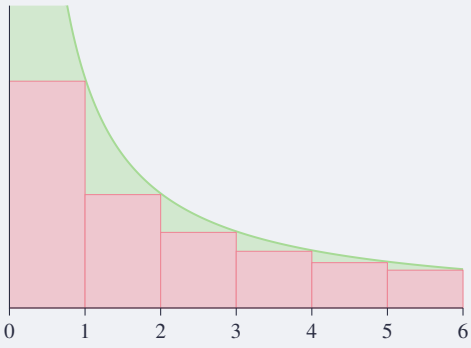
Now, let's focus on the outer summation. This simply iterates $n-1$ times, a term we can simply upper bound with n, With these upper bounds, we can write the following.

$$\mathrm{E}(C) \leq 2n\sum_{k=2}^{n}\frac{1}{k}$$

Now we need to find an upper bound for the summation above. First, let's represent the summation as an area - it can be represented as the sum of areas of a series of rectangles, where the kth square has a width of 1 and height of $1/k$.



Consider the line $y = 1/x$ and overlay it over the diagram.



As you can see, the area of the line $y = 1/x$ is always bigger than the summation, since there is always a section of non-zero area between the curve and the rectangle below. Therefore, we can say

$$\begin{aligned}\sum_{k=2}^{n} \frac{1}{k} &\leq \int_{1}^{n} \frac{1}{x} dx \\ &= \ln n \\ &= \log_e n\end{aligned}$$

The constant from integration has been ignored for the purpose of asymptotic analysis. Now, we can write $\mathrm{E}(C)$ as

$$\mathrm{E}(C) \leq 2n \log_e n$$

Given this, we can write

$$\mathrm{E}(C) = O(n \log n)$$

As discussed before, the big-O of $\mathrm{E}(C)$ is the overall running time of quicksort, since it's dominated by the total number of comparisons the program makes. Therefore, the running time of a randomised quicksort must be $O(n \log n)$.

Space complexity

The space complexity of an algorithm is the amount of extra memory it needs, scaled with input size. For example, a linear search through a list of data has an $O(1)$ space complexity, since the number of extra variables created is constant, regardless of how large the input list may be.

Every call to the quicksort subroutine creates a constant $O(1)$ number of variables. These variables come from the partition subroutine and the pointers which define the left and right sub-arrays to either side of the pivot.

In addition, the number of times the partition subroutine is called scales at a rate of $O(\log n)$. This can be derived by dividing the runtime of quicksort $O(nlogn)$ by the running time of partition $O(n)$. Therefore, the space complexity of quicksort is $O(1) \times O(\log n) = O(\log n)$.

This space complexity is the main reason why quicksort is usually chosen over merge sort, as merge sort has a space complexity of $O(n)$ (an explanation of why is outside the scope of this article, but regardless quite interesting). Since quicksort has an identical running time but better space complexity than merge sort, there are very few cases where using merge sort is more optimal than quicksort.