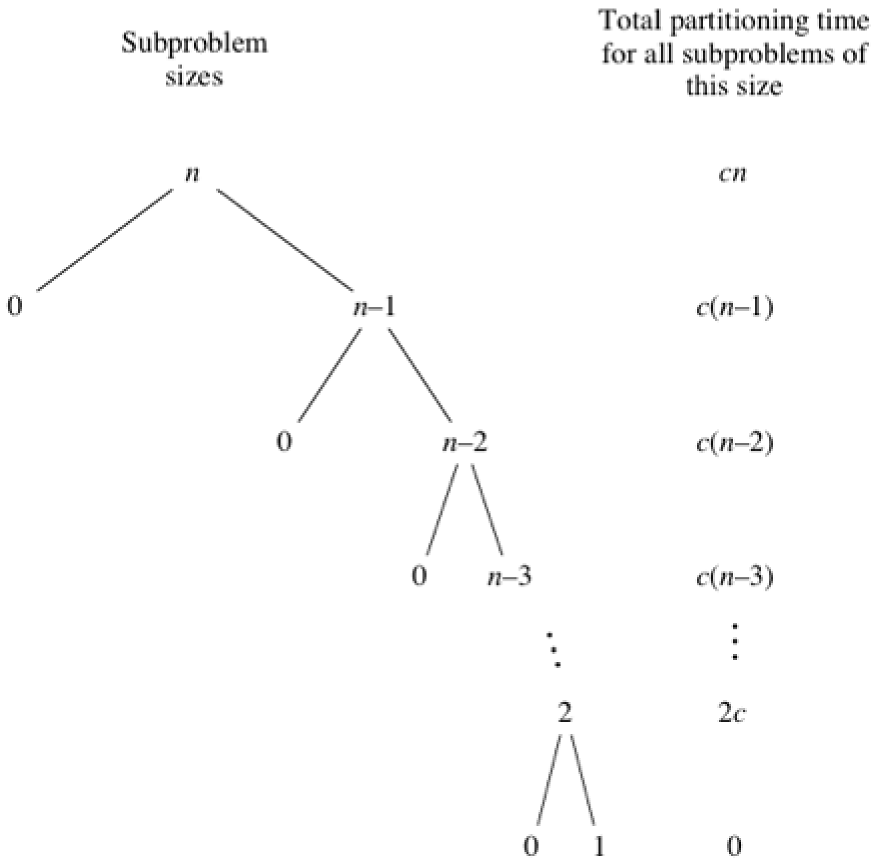
Analysis of Quicksort

How is it that quicksort's worst-case and average-case running times differ? Let's start by looking at the worst-case running time. Suppose that we're really unlucky and the partition sizes are really unbalanced. In particular, suppose that the pivot chosen by the partition function is always either the smallest or the largest element in the n-element subarray. Then one of the partitions will contain no elements and the other partition will contain $n-1$ elements—all but the pivot. So the recursive calls will be on subarrays of sizes 0 and $n-1$.

As in merge sort, the time for a given recursive call on an n-element subarray is $\Theta(n)$. In merge sort, that was the time for merging, but in quicksort it's the time for partitioning.

Worst-case running time

When quicksort always has the most unbalanced partitions possible, then the original call takes cn time for some constant cc, the recursive call on $n-1$ elements takes $c(n-1)$ time, the recursive call on $n-2$ elements takes $c(n-2)$ time, and so on. Here's a tree of the subproblem sizes with their partitioning times:



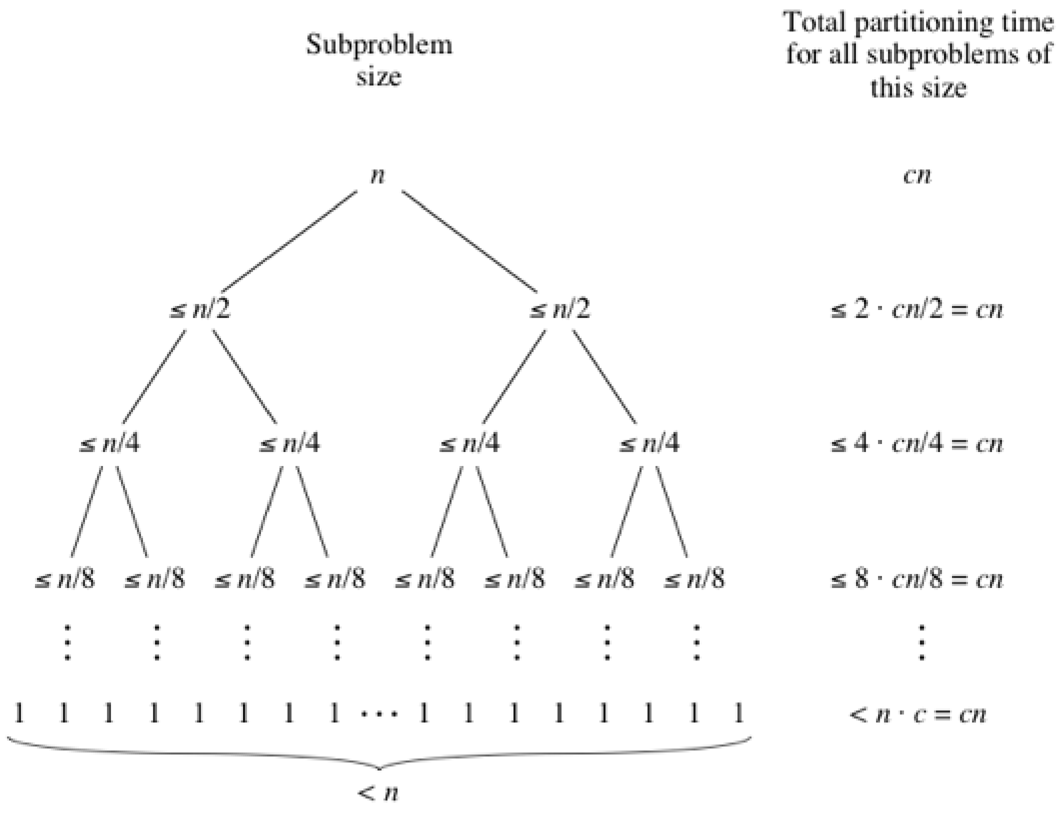
When we total up the partitioning times for each level, we get

$$cn + c(n-1) + c(n-2) + \cdots + 2c = c(n+(n-1) + (n-2) + \cdots + 2) = c((n+1)(n/2)-1)$$

The last line is because $1+2+3+\cdots+n$ is the arithmetic series, as we saw when we analyzed selection sort. (We subtract 1 because for quicksort, the summation starts at 2, not 1.) We have some low-order terms and constant coefficients, but when we use big-Θ notation, we ignore them. In big-Θ notation, quicksort's worst-case running time is $\mathbf{\Theta(n^2)}$.

Best-case running time

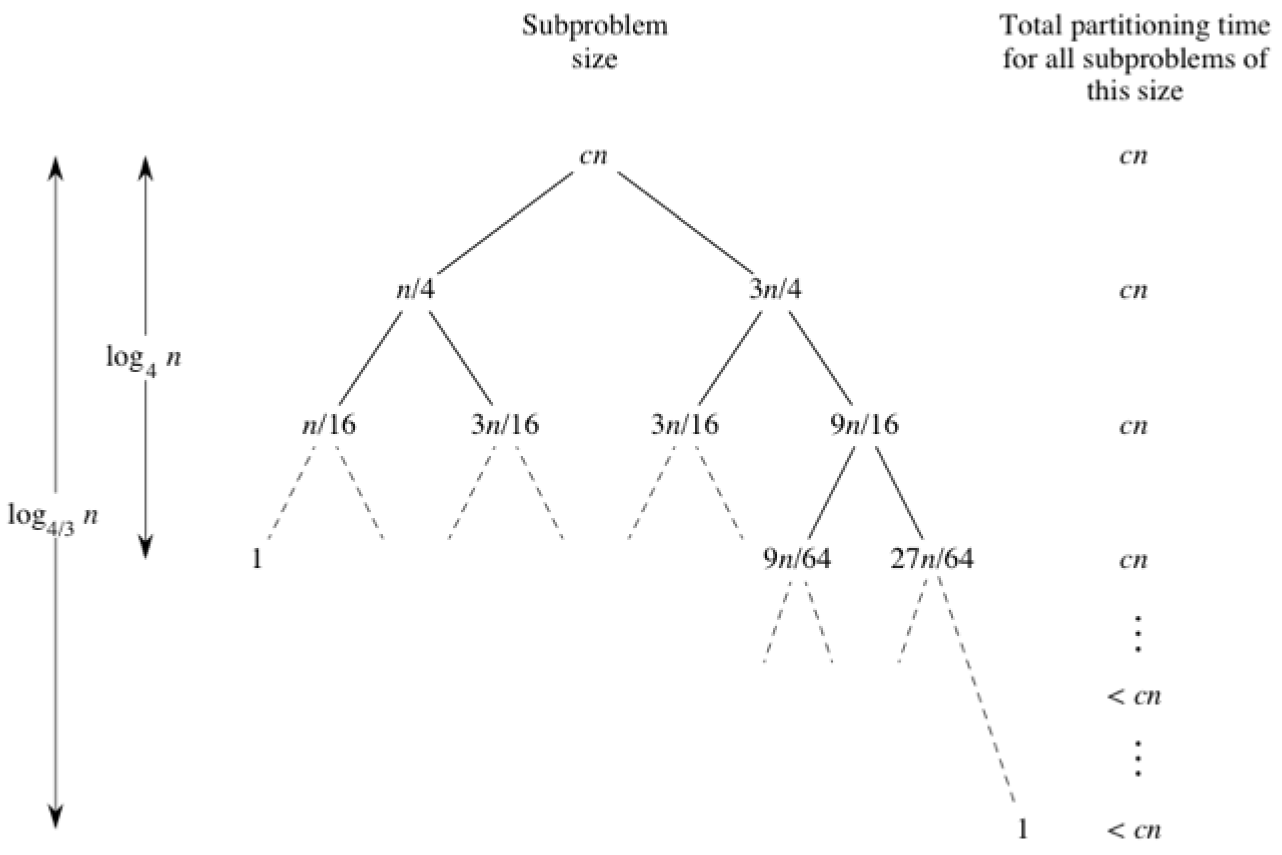
Quicksort's best case occurs when the partitions are as evenly balanced as possible: their sizes either are equal or are within 1 of each other. The former case occurs if the subarray has an odd number of elements and the pivot is right in the middle after partitioning, and each partition has (n-1)/2 elements. The latter case occurs if the subarray has an even number nn of elements and one partition has n/2 elements with the other having n/2-1. In either of these cases, each partition has at most n/2 elements, and the tree of subproblem sizes looks a lot like the tree of subproblem sizes for merge sort, with the partitioning times looking like the merging times:



Using big-Θ notation, we get the same result as for merge sort: Θ(nlgn).

Average-case running time

Showing that the average-case running time is also **Θ(nlgn)** takes some pretty involved mathematics, and so we won't go there. But we can gain some intuition by looking at a couple of other cases to understand why it might be O(nlgn). (Once we have O(nlgn), the Θ(nlgn) bound follows because the average-case running time cannot be better than the best-case running time.) First, let's imagine that we don't always get evenly balanced partitions, but that we always get at worst a 3-to-1 split. That is, imagine that each time we partition, one side gets 3n/4 elements and the other side gets n/4. (To keep the math clean, let's not worry about the pivot.) Then the tree of subproblem sizes and partitioning times would look like this:



The left child of each node represents a subproblem size 1/4 as large, and the right child represents a subproblem size 3/4 as large. Since the smaller subproblems are on the left, by following a path of left children, we get from the root down to a subproblem size of 1 faster than along any other path. As the figure shows, after **$\log_4 n$** levels, we get down to a subproblem size of 1. Why **$\log_4 n$** levels? It might be easiest to think in terms of starting with a subproblem size of 1 and multiplying it by 4 until we reach n. In other words, we're asking for what value of x is **4^x =n**? The answer is $\log_4 n$. How about going down a path of right children? The figure shows that it takes **$\log_{4/3} n$** levels to get down to a subproblem of size 1. Why **$\log_{4/3} n$** levels? Since each right child is 3/4 of the size of the node above it (its parent node), each parent is 4/3 times the size of its right child. Let's again think of starting with a subproblem of size 1 and multiplying the size by 4/3 until we reach n. For what value of x is **$(4/3)^x$ = n**? The answer is **$\log_{4/3} n$**.

In each of the first **$\log_4 n$** levels, there are n nodes (again, including pivots that in reality are no longer being partitioned), and so the total partitioning time for each of these levels is cncn. But what about the rest of the levels? Each has fewer than n nodes, and so the partitioning time for every level is at most cn. Altogether, there are **$\log_{4/3} n$** levels, and so the total partitioning time is O(n

\log$_{4/3}$n). Now, there's a mathematical fact that

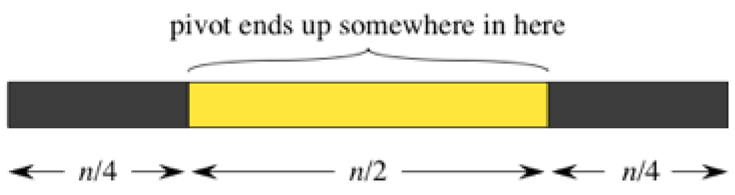
$$log_a n = \frac{log_b n}{log_b a}$$

for all positive numbers a, b, and n. Letting a = 4/3 and b = 2, we get that

$$log_{4/3} n = \frac{lgn}{lg(4/3)}$$

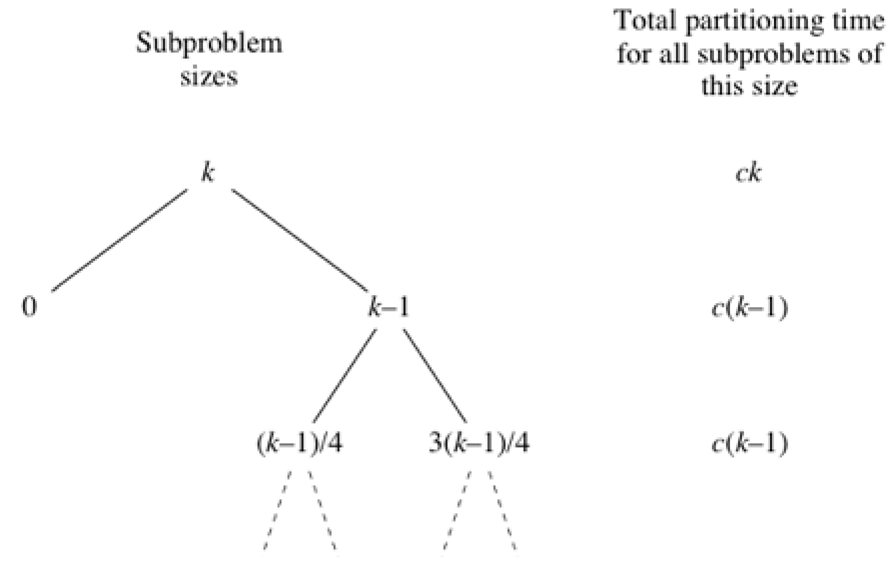
and so log$_{4/3}$n and lgn differ by only a factor of lg(4/3), which is a constant. Since constant factors don't matter when we use big-O notation, we can say that if all the splits are 3-to-1, then quicksort's running time is O(nlgn), albeit with a larger hidden constant factor than the best-case running time.

How often should we expect to see a split that's 3-to-1 or better? It depends on how we choose the pivot. Let's imagine that the pivot is equally likely to end up anywhere in an nn-element subarray after partitioning. Then to get a split that is 3-to-1 or better, the pivot would have to be somewhere in the "middle half":

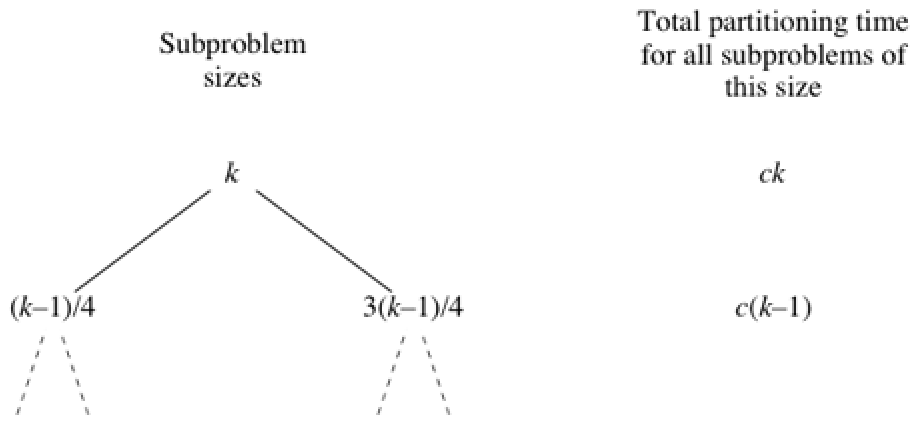


So, if the pivot is equally likely to end up anywhere in the subarray after partitioning, there's a 50% chance of getting at worst a 3-to-1 split. In other words, we expect a split of 3-to-1 or better about half the time.

The other case we'll look at to understand why quicksort's average-case running time is O(nlgn) is what would happen if the half of the time that we don't get a 3-to-1 split, we got the worst-case split. Let's suppose that the 3-to-1 and worst-case splits alternate, and think of a node in the tree with k elements in its subarray. Then we'd see a part of the tree that looks like this:



instead of like this:



Therefore, even if we got the worst-case split half the time and a split that's 3-to-1 or better half the time, the running time would be about twice the running time of getting a 3-to-1 split every time. Again, that's just a constant factor, and it gets absorbed into the big-O notation, and so in this case, where we alternate between worst-case and 3-to-1 splits, the running time is O(nlgn).

Bear in mind that this analysis is not mathematically rigorous, but it gives you an intuitive idea of why the average-case running time might be O(nlgn).

Randomized quicksort

Suppose that your worst enemy has given you an array to sort with quicksort, knowing that you always choose the rightmost element in each subarray as the pivot, and has arranged the array so that you always get the worst-case split. How can you foil your enemy?

You could not necessarily choose the rightmost element in each subarray as the pivot. Instead, you could randomly choose an element in the subarray, and use that element as the pivot. But wait—the partition function assumes that the pivot is in the rightmost position of the subarray. No problem—just swap the element that you chose as the pivot with the rightmost element, and then partition as before. Unless your enemy knows how you choose random locations in the subarray, you win!

In fact, with a little more effort, you can improve your chance of getting a split that's at worst 3-to-1. Randomly choose not one, but three elements from the subarray, and take median of the three as the pivot (swapping it with the rightmost element). By the **median**, we mean the element of the three whose value is in the middle. We won't show why, but if you choose the median of three randomly chosen elements as the pivot, you have a 68.75% chance (11/16) of getting a 3-to-1 split or better. You can go even further. If you choose five elements at random and take the median as the pivot, your chance of at worst a 3-to-1 split improves to about 79.3% (203/256). If you take the median of seven randomly chosen elements, it goes up to about 85.9% (1759/2048). Median of nine? About 90.2% (59123/65536). Median of 11? About 93.1% (488293/524288). You get the picture. Of course, it doesn't necessarily pay to choose a large number of elements at random and take their median, for the time spent doing so could counteract the benefit of getting good splits almost all the time.