O(log log n) terms can show up in a variety of different places, but there are typically two main routes that will arrive at this runtime.

Shrinking by a Square Root

As mentioned in the answer to the linked question, a common way for an algorithm to have time complexity O(log n) is for that algorithm to work by repeatedly cut the size of the input down by some constant factor on each iteration. If this is the case, the algorithm must terminate after O(log n) iterations, because after doing O(log n) divisions by a constant, the algorithm must shrink the problem size down to 0 or 1. This is why, for example, binary search has complexity O(log n).

Interestingly, there is a similar way of shrinking down the size of a problem that yields runtimes of the form O(log log n). Instead of dividing the input in half at each layer, what happens if we *take the square root of the size* at each layer?

For example, let's take the number 65,536. How many times do we have to divide this by 2 until we get down to 1? If we do this, we get

down vote
accepted

- 65,536 / 2 = 32,768
- 32,768 / 2 = 16,384
- 16,384 / 2 = 8,192
- 8,192 / 2 = 4,096
- 4,096 / 2 = 2,048
- 2,048 / 2 = 1,024
- 1,024 / 2 = 512
- 512 / 2 = 256
- 256 / 2 = 128
- 128 / 2 = 64
- 64 / 2 = 32
- 32 / 2 = 16
- 16 / 2 = 8
- 8 / 2 = 4
- 4 / 2 = 2
- 2 / 2 = 1

This process takes 16 steps, and it's also the case that $65{,}536 = 2^{16}$.

But, if we take the square root at each level, we get

- $\sqrt{65{,}536} = 256$
- $\sqrt{256} = 16$
- $\sqrt{16} = 4$
- $\sqrt{4} = 2$

Notice that it only takes four steps to get all the way down to 2. Why is this? Well, let's rewrite this sequence in terms of powers of two:

- $\sqrt{65,536} = \sqrt{2^{16}} = (2^{16})^{1/2} = 2^8 = 256$
- $\sqrt{256} = \sqrt{2^8} = (2^8)^{1/2} = 2^4 = 16$
- $\sqrt{16} = \sqrt{2^4} = (2^4)^{1/2} = 2^2 = 4$
- $\sqrt{4} = \sqrt{2^2} = (2^2)^{1/2} = 2^1 = 2$

Notice that we followed the sequence $2^{16} \rightarrow 2^8 \rightarrow 2^4 \rightarrow 2^2 \rightarrow 2^1$. On each iteration, we cut the exponent of the power of two in half. That's interesting, because this connects back to what we already know - you can only divide the number k in half O(log k) times before it drops to zero.

So take any number n and write it as $n = 2^k$. Each time you take the square root of n, you halve the exponent in this equation. Therefore, there can be only O(log k) square roots applied before k drops to 1 or lower (in which case n drops to 2 or lower). Since $n = 2^k$, this means that $k = \log_2 n$, and therefore the number of square roots taken is O(log k) = O(log log n). Therefore, if there is algorithm that works by repeatedly reducing the problem to a subproblem of size that is the square root of the original problem size, that algorithm will terminate after O(log log n) steps.

One real-world example of this is the van Emde Boas tree (vEB-tree) data structure. A vEB-tree is a specialized data structure for storing integers in the range 0 ... N - 1. It works as follows: the root node of the tree has $\sqrt{N}$ pointers in it, splitting the range 0 ... N - 1 into $\sqrt{N}$ buckets each holding a range of roughly $\sqrt{N}$ integers. These buckets are then each internally subdivided into $\sqrt{(\sqrt{N})}$ buckets, each of which holds roughly $\sqrt{(\sqrt{N})}$ elements. To traverse the tree, you start at the root, determine which bucket you belong to, then recursively continue in the appropriate subtree. Due to the way the vEB-tree is structured, you can determine in O(1) time which subtree to descend into, and so after O(log log N) steps you will reach the bottom of the tree. Accordingly, lookups in a vEB-tree take time only O(log log N).

Another example is the Hopcroft-Fortune closest pair of points algorithm. This algorithm attempts to find the two closest points in a collection of 2D points. It works by creating a grid of buckets and distributing the points into those buckets. If at any point in the algorithm a bucket is found that has more than $\sqrt{N}$ points in it, the algorithm recursively processes that bucket. The maximum depth of the recursion is therefore O(log log n), and using an analysis of the recursion tree it can be shown that each layer in the tree does O(n) work. Therefore, the total runtime of the algorithm is O(n log log n).

O(log n) Algorithms on Small Inputs

There are some other algorithms that achieve O(log log n) runtimes by using algorithms like binary search on objects of size O(log n). For example, the x-fast trie data structure performs a binary search over the layers of at tree of height O(log U),

so the runtime for some of its operations are O(log log U). The related y-fast trie gets some of its O(log log U) runtimes by maintaining balanced BSTs of O(log U) nodes each, allowing searches in those trees to run in time O(log log U). The tango tree and related multisplay tree data structures end up with an O(log log n) term in their analyses because they maintain trees that contain O(log n) items each.

Other Examples

Other algorithms achieve runtime O(log log n) in other ways. Interpolation search has expected runtime O(log log n) to find a number in a sorted array, but the analysis is fairly involved. Ultimately, the analysis works by showing that the number of iterations is equal to the number k such that $n^{2-k} \leq 2$, for which log log n is the correct solution. Some algorithms, like the Cheriton-Tarjan MST algorithm, arrive at a runtime involving O(log log n) by solving a complex constrained optimization problem.

Hope this helps!