Lowest Common Ancestor

Matthew Savage

March 27, 2015

1 Introduction

The Lowest Common Ancestor (LCA) problem is a well-known and greatly researched problem in computer science. In its simplest terms, the problem is simple - given a rooted tree and two nodes (let's call them p and q) from that tree, what is the lowest node on that three that is an ancestor of the given nodes?

For example, consider the tree in Figure 1. The LCA of nodes 9 and 12 is 3, because it is the lowest node on the tree that is an ancestor of both nodes (below it, the path must split to reach the two nodes). Other examples include LCA(6, 4) = 1 and LCA(11, 13) = 7.

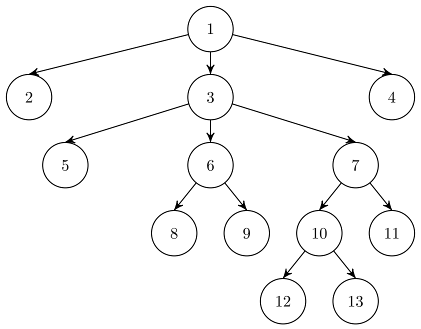

Figure 1: An example tree.

Studying this problem is useful not only in learning about the LCA problem itself but also in understanding the concepts of square root decomposition and problem reduction.

Before I start, a quick note on notation - the algorithms here will all have two parts: preprocessing and query. To denote the complexities of these algorithms efficiently, I will use ⟨preprocessing time complexity, query time complexity⟩.

2 Naive Solutions

2.1 Query by Query - $\langle O(1), O(n)\rangle$

The first (and probably easiest) solution to this problem is to do no preprocessing whatsoever and simply handle the entire problem on each query. Completing this is a relatively simple task - while p and q don't point to the same node, reassign the lower one to its parent (or p if they are at the same height). This effectively moves the two pointers up level by level until they match, taking the problem fairly literally with a line sweep-like approach. Since this will consist of at most n pointer reassignments in the worst case, this will take $O(n)$.

However, $O(n)$ query speed is fairly undesirable for anything practical, as the number of queries is usually quite large. However, this is definitely the fastest solution to implement, so if you think the bounds of a given problem will allow for it, definitely go with this solution.

2.2 Preprocessing Only - $\langle O(n^2), O(1)\rangle$

An alternative approach to this problem is to precompute the solutions to all $O(n^2)$ subproblems and simply returning the answer on each query. (Note that precomputation still only takes $O(n^2)$ and not $O(n^3)$ because some simple DP can be used to avoid computing the same subproblems multiple times.)

Of course, $O(n^2)$ is still quite slow, and isn't much of a step up over the other algorithm. (In fact, this solution performs *worse* when the number of queries is less than n.)

3 Square Root Decomposition - $\langle O(n), O(\sqrt{n})\rangle$

The square root function has a bit of magic to it, though it may seem like a bit of a silly statement at first. The all-important statement is that

$$\sqrt{n} \times \sqrt{n} = n \tag{1}$$

Though this seems inherently obvious (it *is* the square root function, after all), we can use this to our advantage: if we evenly distribute n items into $\sqrt{n}$ groups, each group will contain $\sqrt{n}$ items. This effectively allows us to process range queries on a list of numbers of length n in two steps - first by considering up to $\sqrt{n}$ buckets containing $\sqrt{n}$ elements each, and then by considering up to $2\sqrt{n}$ individual elements. This gives us a final complexity of $O(\sqrt{n})$. This method is called *square root decomposition.*

How does this help us with LCA? Consider what would happen if we split the tree (which we will say has height h) into $\sqrt{h}$ segments by height. We could then store each node's highest ancestor within each segment - this would give us steps of height $\sqrt{h}$. We could then use this in the same way as the naive linear solution to find the LCA - first by comparing at steps of $\sqrt{h}$, and then considering individual items when the larger steps begin to match.

4 Logarithmically - $\langle O(n \log n), O(\log n)\rangle$

Of course, there's a reason we don't generally use square root decomposition - doing a binary search-style structure is generally more efficient. Let's apply that here too.

For precomputation, let's make a table L where $L[i][j]$ represents the node 2^j steps above node i for $0 \leq j \leq \log n$. (For example, $L[i][0]$ is node i's parent and $L[i][2]$ is node i's great great grandparent, or the node four steps up from i.) Using this table, we can use the same method but take steps at powers of 2 instead of at $\sqrt{n}$ increments, reducing query speed to $O(\log n)$ at the cost of a slight increase to preprocessing and memory (which now require $O(n \log n)$).