Computing the LCP array in linear time, given $S$ and the suffix array $POS$.

Given a string $S$, define $Suff_k$ as the suffix of string $S$ starting at position $k$. Define $lcp(S_1, S_2)$ as the length of the longest common prefix of strings $S_1$ and $S_2$. If POS is the suffix array of a string $S$, and $k$ is an entry at a position, say $i$, of POS, then define $Pred(k)$ as the entry in position $i - 1$ of POS. That is $Pred(k)$ is the entry in POS just to the left of where $k$ is in array POS. We want to compute, for each $k$ from 1 to $n$, $lcp(Suff_k, Suff_{Pred(k)})$, which is defined to be the length of the longest common prefix of $Suff_k$ and $Suff_{Pred(k)}$; this is also called $depth(k)$.

We will compute these in order of $k$ from 1 to $n$. Of course, for each $k$, we could compute $lcp(Suff_k, Suff_{Pred(k)})$ by doing a direct comparison from the start of $Suff_k$ and $Suff_{Pred(k)}$ for as long as they match. We call that the “direct approach”. But the total time for the direct approach would be $O(n^2)$, not $O(n)$. We will use one simple speedup of the direct approach to obtain an $O(n)$ time algorithm.

Suppose $j = Pred(k)$ and $lcp(Suff_k, Suff_j) = h > 0$.

The first claim is: $lcp(Suff_{k+1}, Suff_{j+1}) = h - 1$. This follows immediately from the fact that $lcp(Suff_k, Suff_j) = h > 0$. Draw a picture of the string and positions $k, k + 1, j, j + 1$.

The second claim is that if $h > 0$, then $lcp(Suff_{k+1}, Suff_{Pred(k+1)}) \geq lcp(Suff_{k+1}, Suff_{j+1})$, and hence $lcp(Suff_{k+1}, Suff_{Pred(k+1)}) \geq h - 1$.

This follows from looking at the locations of the leaves $k + 1$, $j + 1$ and $Pred(k+1)$ are in the suffix tree. By definition and construction of POS, the LCA of leaves $k + 1$ and $Pred(k+1)$ is at or below the LCA of leaves $k + 1$ and $j + 1$ (draw a picture). In more detail, the paths to leaf $k + 1$ and to leaf $j + 1$ agree for exactly $h - 1$ characters, and then they diverge at some node, say $v$. Now $Pred(k+1)$ is the leaf visited in the lexicographic DFS (which is conceptually one way to obtain or define the suffix array) just before leaf $k + 1$ is visited, and if the path to leaf $Pred(k+1)$ does not extend below $v$, that would be impossible. Hence $lcp(Suff_{k+1}, Suff_{Pred(k+1)}) \geq h - 1$.

The consequence of the second claim is that when we want to compute $lcp(Suff_{k+1}, Suff_{Pred(k+1)})$ in the direct approach, we don’t have to start character comparisons at positions $k+1$ and $Pred(k+1)$ in $S$, but rather can skip ahead by $h - 1$ positions and start comparing at positions $k+1 + h - 1 = k + h$ and $Pred(k+1) + h - 1$. This is because we already know that if we did start comparing at positions $k + 1$ and $Pred(k+1)$ then those comparisons
would match for $h - 1$ positions, if $h > 0$.

We claim that with the above little speedup, compared to the $O(n^2)$ direct approach, the number of comparisons in $O(n)$. To see this, consider how Depth($k$) changes as $k$ increases from 1 to $n$. At the start of each iteration, the known depth either decreases by one (if Depth($k - 1$) $> 0$), or it remains the same (if Depth($k - 1$) $= 0$). After the start of any iteration, the Depth increases by exactly the number of matches made. Since the total decrease of Depth is at most $n$ (the number of iterations), and Depth can never be larger than $n$, there can be at most $2n$ matches over the execution of the algorithm. Each iteration ends as soon as there is a mismatch, so there can be at most $n$ mismatches. So, the total number of comparisons is bounded by $3n$. All other work done in the algorithm is proportional to the number of compares.