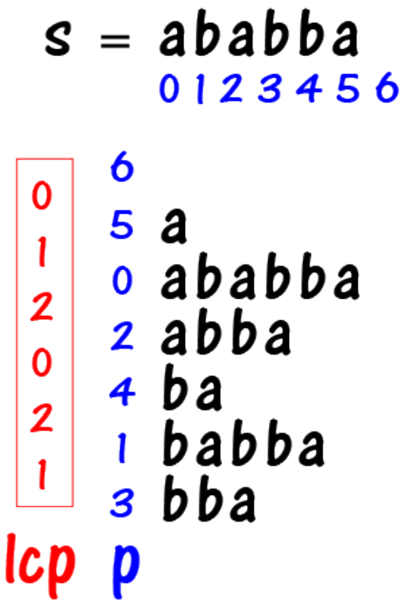
Finding LCP

Finding LCP of two suffixes

To solve more complex problems, we need a little more information, namely, we need to learn how to quickly calculate the value of $LCP(i, j)$, the length of the largest common prefix for suffixes starting at i and j.

To do this, for each pair of adjacent suffixes in the suffix array, find the length of their largest common prefix and put them in the array $lcp[i] = LCP(p[i], p[i+1])$. For example, for the string that we used in the first step, we get such an array.



Now, in order to find the LCP of any two suffixes, we just need to find their positions in the suffix array and calculate the minimum in the lcp array on the segment between them. $LCP(i, j) = \min(lcp[pos[i]..pos[j] - 1])$, where $pos[i]$ is the position of the suffix i in the suffix array.

Why is this true? Let's look at some two suffixes i and j. Let $k = LCP(i, j)$, that is, the suffixes i and j have k common characters in the beginning. Since the suffixes are sorted, in all suffixes between them the first k characters must also be the same. So all lcp on this segment is not less than k, and therefore the minimum on this segment is not less than k. On the other hand, it cannot be greater than k, since this means that each pair of suffixes has more than k common characters, which means that i and j must have more than k common characters.

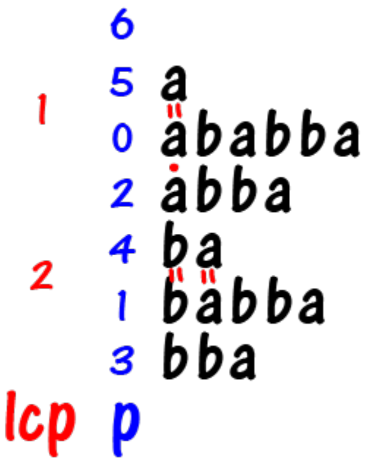
How do we quickly calculate the minimum on a segment of the lcp array? To do this, we can use one of two data structures: a segment tree or a sparse table. A segment tree is built in $O(n)$ and answers a query in $O(\log n)$, a sparse table is built in $O(n \log n)$ and answers a query in $O(1)$. Sparse tables are usually used, but a segment tree can also be useful in some specific task.

Thus, if we have built an array lcp, then the task of finding LCP for two suffixes reduces to find a minimum on a segment in array, it remains to learn how to build an array lcp.

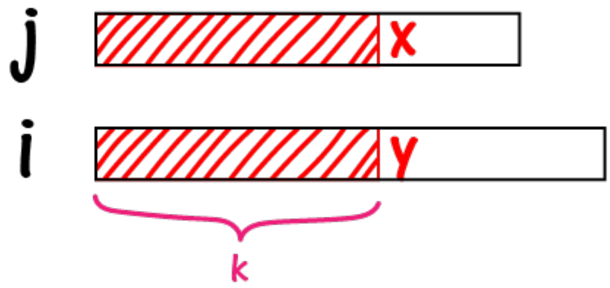
Building array lcp

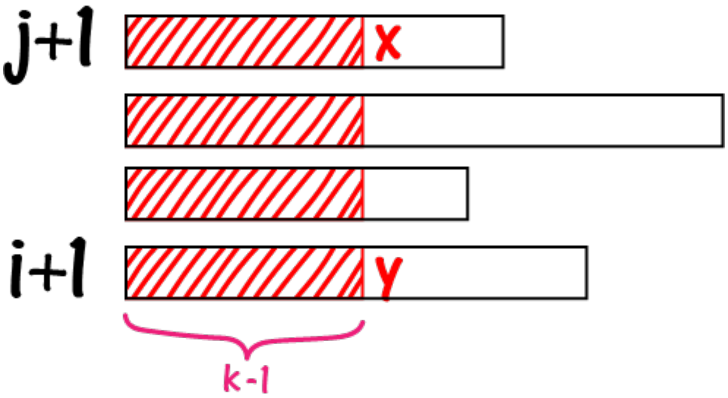
To build the lcp array, we will use the algorithm of Kasai, Arimura, Arikawa, Lee and Park. The algorithm works as follows. We will iterate over the suffixes from longest to shortest, and for each find LCP with the previous one in the suffix array.

For example, in our line, we first calculate the LCP for the suffixes 0 and 5 (it is 1), then for the suffixes 1 and 4 (it is 2). And here we will do the trick: we know that the suffixes 1 and 4 have two common characters, which means that the suffixes 2 and 5 have one common character. For all suffixes between 2 and 5, the first character must be the same, in particular, the suffix 2 with the previous one has at least one common character. Therefore, the first character of the suffixes 2 and 0 may be skipped, and we start the comparison immediately with the second character.



In the general case, let the suffix j be in the suffix array before the suffix i, and they have $k > 0$ common characters. Then the suffixes $i + 1$ and $j + 1$ have $k - 1$ common characters, so the suffix $i + 1$ has at least $k - 1$ common characters with the previous suffix. We can skip them and start the comparison with the k-th character.





What is the time complexity of this algorithm? Let's see how many times we compare characters. If we compared the characters, and they were not equal, then we found the value of lcp and move on to the next suffix. This happens $O(n)$ times. If the characters are equal, then we increase the number of common characters k. This number decreases by 1 when moving to the next suffix and cannot become more than n, which means it increases no more than $2n$ times. Thus, the total time of the algorithm is $O(n)$.