# A. Yet Another Problem with Strings

Editorial

**NOTE:** In this problem, the purpose of using `LAST_YES` indices to generate queries, is to force participants to implement an online solution. From now, we assume that all queries represent input queries already transformed using `LAST_YES` variable as described in the statement.

You are given an array $S[1..N]$ of $N$ strings consisting only of lowercase English letters. Your task is to perform $Q$ queries on $S$. There are two type of queries to handle:

- $contains(t)$ := returns `YES` if at least one string from $S$ is a prefix of string $t$. Otherwise returns `NO`.
- $append(i, c)$ := appends character $c$ at the end of $S_i$.

In the below explanation, we denote a hash of a string as its rolling hash, sometimes also called a polynomial hash. It is defined as follows. For a string $t[1..k]$ its hash value $h(t) := t_1 \cdot X^{k-1} + t_2 \cdot X^{k-2} + \ldots + t_k$, where $t_i$ is the integer representing the $i^{th}$ letter of $t$. The hash is defined for some integer $X$ and computed modulo some prime number.

In order to solve the problem, we are going use two data structures:

- augmented trie
- set of hashes of strings currently in $S$.

### Augmented trie

The idea is to represent $S$ as a trie with counters. In more details, initially, we insert all strings from $S$ to the trie. Then, for each node of the trie, we want to know how many strings from $S$ are prefixes of the string corresponding to this node. For the initial $S$, it is very easy to do: first, we assign 1 to counters of nodes corresponding to strings from $S$, and then we propagate all these ones down to the trie using for example a `DFS`.

The trie augmented in such a way, will be used to answer $contains(t)$ queries. The idea is that for a given suffix $t_j$ of $t$, we are going to find its longest prefix which is in the trie and its counter is positive. Notice that a positive counter means that there is a prefix of the string corresponding to the node which is currently in $S$, so this prefix is a prefix of $t_j$, thus its a substring of $t$.

Sounds good, but do not forget that we have to handle *append* queries also, and they have an impact on the trie and its counters. The good thing is that for any $append(i, c)$ query, we can use a lazy propagation in order to update counters in the trie. In more details, first, we find the node $v$ corresponding to $S_i$. Then we decrement its counter, since it now represents one string less than it used to be. Next, if there is no node corresponding to $S_i$ with appended $c$, we create it and set up its counter. The last thing is to decrement counters for the subtrees of the other children of $v$. We have to do this, because each such node has now one less prefix which is in $S$ than before the update.

Notice that it is not a good idea to update counters in each subtree every time, because it may lead to updating a very large amount of nodes for a single query. The crucial observation is that in order to handle *contains* query, we only need the information if a counter for a particular node is positive or not, we do not need it exact value. Based on that fact, we can use lazy propagation concept and update immediately only counters which become zero - notice that counters at any path from the root forms a non-decreasing sequence. For any other node, it is sufficient to store the information that counters in all nodes in its subtree has to be decreased by some amount in the future, and as soon as this amount becomes zero, we update it and propagate the update down the subtree. This technique allows us to achieve an amortized linear time for all *append* queries. Doing this, we have a trie with up to date information if a given node has a prefix which is currently in $S$.

### Set of hashes

In addition to the trie, we will maintain a set of hashes of all strings currenctly in $S$. It is a good idea have it implemented as a map in order to store not only the information if a given string is in $S$, but also a pointer to its node. Notice that this map can be easily updated while performing *append* queries. We will use the map to check if a particular string is in $S$ and the pointer to check out if its node has a positive counter or not.

### Handling queries

Handling *append* queries is explained in the trie section above. Now, we will focus on *contains*$(t)$ queries. For a given $t$, the first thing we do is to compute hashes for all its prefixes. We will use them in order to be able to compute a hash for any substring of $t$ in a constant time.

Next, we iterate over all suffixes of $t$. For a particular suffix $t_j = t[j, \ldots, |t|]$, we are going to find its longest prefix which is in $S$ and has a positive counter. In order to do that, we can use a binary search. In more details, we are going to find the greatest index $i$, for which $t[j, \ldots i]$ is in $S$ and its counter is positive. To do so, we can use hashes computed for $t$ to compute hash of every prefix of $t_j$, and check fast using our map of hashes, if this prefix is in $S$. If it is, we can easily get its counter using a pointer. Notice that if and only if this counter is positive, then there is at least one prefix of $t$ which is currently in $S$. The reason that we are looking for the longest prefix of $t$ is that counters on any path from the root of the trie form a non decreasing sequence, so its sufficient to find the longest prefix with a positive counter. Since we are doing it for all suffixes of $t$, we are checking all possible starting points of substrings of $t$, so all cases are covered.

The time complexity of a single *contains*$(t)$ query is $O(|t| \cdot \log |t| \cdot f)$, where $f$ is the cost of checking if a given string is in $S$ using the map. This is true, because there are $|t|$ suffixes of $t$, and for each one, we are finding its longest prefix which is in $S$ using binary search and our map implementation. You can use an expected constant time access map implementation, but is sufficient to use an implementation of the map as a balanced binary tree, to achieve $(O|t| \cdot log|t| \cdot \log N)$ per single *contains*$(t)$ query.