Coding Competitions Farewell Rounds - Round D

Analysis: Genetic Sequences

Test Set 1

The problem asks us to answer queries on two strings $\mathbf{A}$ and $\mathbf{B}$. In each query, we are given an $\mathbf{A}$-prefix and a $\mathbf{B}$-suffix. We are asked to find the longest prefix of the $\mathbf{B}$-suffix that is a substring of the $\mathbf{A}$-prefix.

A naive algorithm might try every single prefix of the $\mathbf{B}$-suffix and do a substring search in the $\mathbf{A}$-prefix. Assuming a fast string-searching algorithm, this gives us an $O(\mathbf{Q}M(N+M))$ algorithm where $\mathbf{Q}$ is the number of queries, N is the length of $\mathbf{A}$, and M is the length of $\mathbf{B}$. This is too slow to pass the first test set.

This approach can be sped up. One approach is to use the Z-function. The i-th entry of the Z-function is denoted by Z_i is the longest match between a string and the suffix of that string starting at index i. We can precompute the Z-function for the concatenation of $\mathbf{A}$ onto the end of each $\mathbf{B}$-suffix. This requires $O(NM)$ time. When answering a query, we find the Z-function table corresponding to the $\mathbf{B}$-suffix in that query. Then, we can iterate through each index corresponding to the $\mathbf{A}$-prefix to find the longest match. Overall, this gives an $O(\mathbf{Q}N+NM)$ algorithm, which is fast enough for Test Set 1. Other techniques including Knuth-Morris-Pratt or suffix trees/arrays can be used too.

Test Set 2

A faster approach is needed for the larger second test set. We can start with some observations. Consider a single query. The longest match in the $\mathbf{A}$-prefix must be fully contained in the prefix. Imagine we had an algorithm that could quickly find the longest match but that match is allowed to go outside the $\mathbf{A}$-prefix. Specifically, the match only needs to start in the $\mathbf{A}$-prefix, but might go outside of the prefix later on. Call this a *relaxed query*. This may seem arbitrary, but we will see later that relaxed queries can be answered efficiently using well-known techniques.

Assume we have a fast algorithm for answering relaxed queries. Can we now solve the problem efficiently? Assume we want to check if the longest match is at least k characters long. We know that any match must start in the $\mathbf{A}$-prefix with the last $k-1$ characters removed, so we can solve it using a relaxed query. Also, observe that if there is a match that is at least k characters, it implies there is a match of at least $k-1$ characters. This means we can binary search on k. This allows us to answer a query in $O(R\log M)$ time, where R is the time required to answer a relaxed query.

Now we need to find an efficient algorithm for anwering relaxed queries. One way to do this involves two data structures. A suffix array with its longest common prefix table, and a persistent binary search tree.

Consider the suffix array of $\mathbf{A}$ concated with $\mathbf{B}$. We will need some properties of the suffix array and the longest common prefixes between pairs of suffixes. We can find the suffix array position of any $\mathbf{B}$-suffix by keeping a lookup table into the suffix array. Let us say that that position is b. The longest common prefix between suffix b and suffixes $b+1, b+2, \ldots$ will decrease monotonically. The same is true for $b-1, b-2, \ldots$. To answer a relaxed query, we need to find the longest common prefix between b and any suffix beginning in the $\mathbf{A}$-prefix. Using the previously mentioned property, we will therefore need to find the first index less than b that corresponds to a suffix starting in the $\mathbf{A}$-prefix, as well as the first index greater than b also corresponding to a suffix starting in the $\mathbf{A}$-prefix.

We can find these two entries by sweeping over the suffixes of $\mathbf{A}$ and constructing a persistent binary search tree T. We can sweep over $\mathbf{A}$ in increasing order of indexes. The corresponding suffix array location can be found and added to T. The end result will be N different versions of T, each corresponding to the set of suffixes for each $\mathbf{A}$-prefix. When answering a relaxed query, we can look up the version of T corresponding to the $\mathbf{A}$-prefix, and do a tree traversal to find the first elements immediately before and after the $\mathbf{B}$-suffix. Note that the query value will be the suffix array location of the $\mathbf{B}$-suffix.

This leads to an algorithm whose complexity is $O((N+M)\log(N+M))$ to build the suffix array and persistent binary search trees. Then, we require $O(\mathbf{Q}\log(M)\log(N))$ to answer all of the queries. This is sufficient to pass Test Set 2.