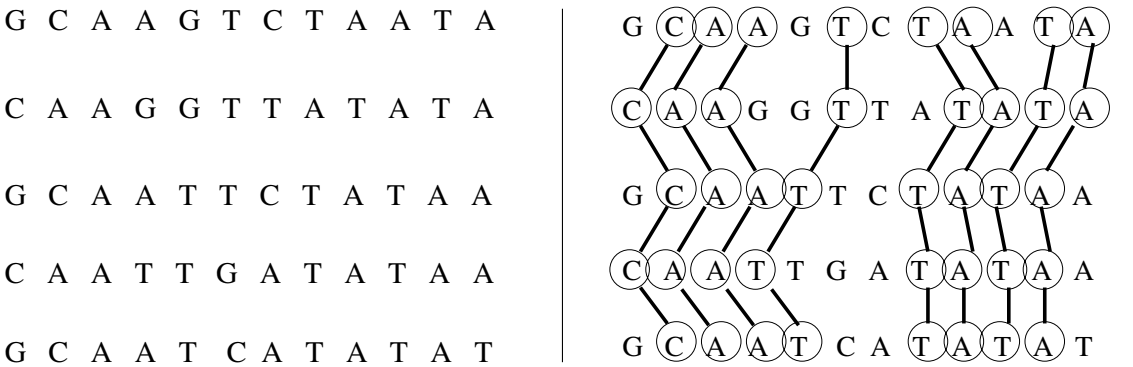

INPUT OUTPUT

18.8 Longest Common Substring/Subsequence

Input description: A set S of strings $S_1, \ldots, S_n$.

Problem description: What is the longest string S' such that all the characters of S' appear as a substring or subsequence of each S_i, $1 \leq i \leq n$?

Discussion: The problem of longest common substring/subsequence arises whenever we search for similarities across multiple texts. A particularly important application is finding a consensus among biological sequences. The genes for building proteins evolve with time, but the functional regions must remain consistent in order for them to work correctly. The longest common subsequence of the same gene in different species provides insight into what has been conserved over time.

The longest common subsequence problem for two strings is a special case of edit distance (see Section 18.4 (page 631)), when substitutions are forbidden and exact character match, insert, and delete are the only allowable edit operations. Under these conditions, the edit distance between P and T is $n + m - 2|lcs(P, T)|$, since we can delete the missing characters from P to the $lcs(P, T)$ and insert the missing characters from T to transform P to T.

Issues arising include

- *Are you looking for a common substring?* – In detecting plagiarism, we might need to find the longest phrase shared between two or more documents. Since phrases are strings of consecutive characters, here we need the longest common *substring* between the texts.

  The longest common substring of a set of strings can be identified in linear time using suffix trees, as discussed in Section 12.3 (page 377). The trick is to build a suffix tree containing all the strings, label each leaf with the input

string it represents, and then do a depth-first traversal to identify the deepest node with descendants from each input string.

- *Are you looking for a common scattered subsequence?* – For the rest of our discussion here, we restrict attention to finding common scattered subsequences. This algorithm is a special case of the dynamic program edit-distance computation. Indeed, an implementation in C is given on page 288.

  Let $M[i,j]$ denote the number of characters in the longest common substring of $S[1],\ldots,S[i]$ and $T[1],\ldots,T[j]$. When $S[i] \neq T[j]$, there is no way the last pair of characters could match, so $M[i,j] = \max(M[i,j-1], M[i-1,j])$. But if $S[i] = T[j]$, we have the option to select this character for our substring, so $M[i,j] = \max(M[i-1,j-1]+1, M[i-1,j], M[i,j-1])$.

  This recurrence computes the length of the longest common subsequence in $O(nm)$ time. We can reconstruct the actual common substring by walking backward from $M[n,m]$ and establishing which characters were matched along the way.

- *What if there are relatively few sets of matching characters?* – There is a faster algorithm for strings that do not contain too many copies of the same character. Let r be the number of pairs of positions (i,j) such that $S_i = T_j$. Thus, r can be as large as mn if both strings consist entirely of the same character, but $r = n$ if both strings are permutations of $\{1,\ldots,n\}$. This technique treats the pairs of r as defining points in the plane.

  The complete set of r such points can be found in $O(n+m+r)$ time using bucketing techniques. We create a bucket for each alphabet symbol c and each string (S or T), then partition the positions of each character of the string into the appropriate bucket. We then create a point (s,t) from every pair $s \in S_c$ and $t \in T_c$ in the buckets S_c and T_c.

  A common subsequence describes a monotonically nondecreasing path through these points, meaning the path only moves up and to the right. The longest such path can be found in $O((n+r)\lg n)$ time. We sort the points in order of increasing x-coordinate, breaking ties in favor of increasing y-coordinate. We insert points one by one in this order, and maintain the minimum terminal y-coordinate of any path going through exactly k points for each k, for $1 \leq k \leq n$. The new point (p_x, p_y) changes exactly one of these paths, either identifying a new longest subsequence or reducing the y-coordinate of the shortest path whose endpoint lies above p_y.

- *What if the strings are permutations?* – Permutations are strings without repeating characters. Two permutations define n pairs of matching characters, and so the above algorithm runs in $O(n \lg n)$ time. A particularly important case occurs in finding the longest *increasing* subsequence of a numerical sequence. Sorting the sequence and then replacing each number by

its rank defines a permutation p. The longest common subsequence of p and $\{1, 2, 3, \ldots, n\}$ gives the longest increasing subsequence.

- *What if we have more than two strings to align?* – The basic dynamic programming algorithm can be generalized to k strings, taking $O(2^k n^k)$ time, where n is the length of the longest string. This algorithm is exponential in the number of strings k, and so it will likely be too expensive for more than a few strings. Furthermore, the problem is NP-complete, so no better exact algorithm is destined to come along soon.

  Many heuristics have been proposed for multiple sequence alignment. They often start by computing the pairwise alignment for each of the $\binom{k}{2}$ pairs of strings. One approach then replaces the two most similar sequences with a single merged sequence, and repeats until all these alignments have been merged into one. The catch is that two strings often have many different alignments of optimal cost. The "right" alignment to pick depends upon the remaining sequences to merge, and is hence unknowable to the heuristic.

Implementations: Several programs are available for multiple sequence alignment of DNA/protein sequence data. *ClustalW* [THG94] is a popular and well-regarded program for multiple alignment of protein sequences. It is available at *http://www.ebi.ac.uk/Tools/clustalw/*. Another respectable option is the *MSA* package for multiple sequence alignment [GKS95], which is available at *http://www.ncbi.nlm.nih.gov/CBBresearch/Schaffer/msa.html*.

Any of the dynamic programming-based approximate string matching programs of Section 18.4 (page 631) can be used to find the longest common subsequence of two strings. More specialized implementations in Perl, Java, and C are available at *http://www.bioalgorithms.info/downloads/code/*.

Combinatorica [PS03] provides a Mathematica implementation of an algorithm to construct the longest increasing subsequence of a permutation, which is a special case of longest common subsequence. This algorithm is based on Young tableaux rather than dynamic programming. See Section 19.1.9 (page 661).

Notes: Surveys of algorithmic results on longest common subsequence (LCS) problems include [BHR00, GBY91]. The algorithm for the case where all the characters in each sequence are distinct or infrequent is due to Hunt and Szymanski [HS77]. Expositions of this algorithm include [Aho90, Man89]. There has been a surprising amount of recent work on this problem, including efficient bit-parallel algorithms for LCS [CIPR01]. Masek and Paterson [MP80] solve longest common subsequence in $O(mn/\log(\min\{m, n\}))$ for constant-sized alphabets, using the four Russians technique.

Construct two random n-character strings on an alphabet of size α. What is the expected length of their LCS? This problem has been extensively studied, with an excellent survey by Dancik [Dan94].

Multiple sequence alignment for computational biology is large field, with the books of Gusfield [Gus97] and Durbin [DEKM98] serving as excellent introductions. See [Not02]

for a more recent survey. The hardness of multiple sequence alignment follows from that of shortest common subsequence for large sets of strings [Mai78].

We motivated the problem of longest common substring with the application of plagiarism detection. See [SWA03] for the interesting details of how to implement a plagiarism detector for computer programs.

Related Problems: Approximate string matching (see page 631), shortest common superstring (see page 654).