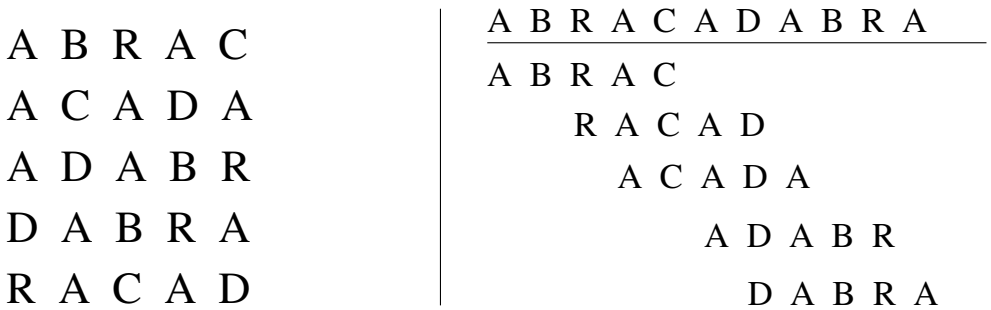

INPUT OUTPUT

18.9 Shortest Common Superstring

Input description: A set of strings $S = \{S_1, \ldots, S_m\}$.

Problem description: Find the shortest string S' that contains each string S_i as a substring of S'.

Discussion: Shortest common superstring arises in a variety of applications. A casino gambling addict once asked me how to reconstruct the pattern of symbols on the wheels of a slot machine. On every spin, each wheel turns to a random position, displaying the selected symbol as well as the symbols immediately before/after it. Given enough observations of the slot machine, the symbol order for each wheel can be determined as the shortest common (circular) superstring of the observed symbol triples.

Another application of shortest common superstring is data/matrix compression. Suppose we are given a sparse $n \times m$ matrix M, meaning that most elements are zero. We can partition each row into m/k runs of k elements, and construct the shortest common superstring S' of all these runs. We can now represent the matrix by this superstring plus an $n \times m/k$ array of pointers denoting where each of these runs starts in S'. Any particular element $M[i, j]$ can still be accessed in constant time, but there will be substantial space savings when $|S| << mn$.

Perhaps the most compelling application is in DNA sequence assembly. Machines readily sequence fragments of about 500 base pairs or characters of DNA, but the real interest is in sequencing large molecules. Large-scale "shotgun" sequencing clones many copies of the target molecule, breaks them randomly into fragments, sequences the fragments, and then proposes the shortest superstring of the fragments as the correct sequence.

Finding a superstring of a set of strings is not difficult, since we can simply concatenate them together. Finding the *shortest* such string is what's problematic.

Indeed, shortest common superstring is NP-complete for all reasonable classes of strings.

Finding the shortest common superstring can easily be reduced to the traveling salesman problem (see Section 16.4 (page 533)). Create an overlap graph G where vertex v_i represents string S_i. Assign edge (v_i, v_j) weight equal to the length of S_i minus the overlap of S_j with S_i. Thus, $w(v_i, v_j) = 1$ for $S_i = abc$ and $S_j = bcd$. The minimum weight path visiting all the vertices defines the shortest common superstring. These edge weights are not symmetric; note that $w(v_j, v_i) = 3$ for the example above. Unfortunately, asymmetric TSP problems are much harder to solve in practice than symmetric instances.

The greedy heuristic provides the standard approach to approximating shortest common superstring. Identify which pair of strings have the maximum overlap. Replace them by the merged string, and repeat until only one string remains. This heuristic can actually be implemented in linear time. The seemingly most time-consuming part is in building the overlap graph. The brute-force approach to finding the maximum overlap of two length-l strings takes $O(l^2)$ for each of $O(n^2)$ string pairs. However, faster times are possible by using suffix trees (see Section 12.3 (page 377)). Build a tree containing all suffixes of all strings of S. String S_i overlaps with S_j iff a suffix of S_i matches the prefix of S_j—an event defined by a vertex of the suffix tree. Traversing these vertices in order of distance from the root defines the appropriate merging order.

How well does the greedy heuristic perform? It can certainly be fooled into creating a superstring that is twice as long as optimal. The optimal merging order for strings $c(ab)^k$, $(ba)^k$, and $(ab)^k c$ is left to right. But greedy starts by merging the first and third string, leaving the middle one no overlap possibility. The greedy superstring can never be worse than 3.5 times optimal, and usually will be a lot better in practice.

Building superstrings becomes more difficult when given both positive and negative strings, where each of the negative strings are forbidden to be a substring of the final result. The problem of deciding whether *any* such consistent substring exists is NP-complete, unless you are allowed to add an extra character to the alphabet to use as a spacer.

Implementations: Several high-performance programs for DNA sequence assembly are available. Such programs correct for sequencing errors, so the final result is not necessarily a superstring of the input reads. At the very least, they will serve as excellent models if you really need a short proper superstring.

CAP3 (Contig Assembly Program) [HM99] and *PCAP* [HWA+03] are the latest in a series of assemblers by Xiaoqiu Huang and his collaborators, which are available from *http://seq.cs.iastate.edu/*. They have been used on mammalian scale assembly projects involving hundreds of millions of bases.

The Celera assembler that originally sequenced the human genome is now available as open source. See *http://sourceforge.net/projects/wgs-assembler/*.

Notes: The shortest common superstring (SCS) problem and its application to DNA shotgun assembly are ably surveyed in [MKT07, Mye99a]. Kececioglu and Myers [KM95] report on an algorithm for this more general version of shortest common superstring, where the strings are assumed to have character substitution errors. Their paper is recommended reading to anyone interested in fragment assembly.

Blum et al. [BJL+94] gave the first constant-factor approximation algorithms for shortest common superstring, using a variation of the greedy heuristic. More recent research has beaten this constant down to 2.5 [Swe99], progress towards the expected factor-two result. The best approximation ratio so far proven for the standard greedy heuristic is 3.5 [KS05a]. Fast implementations of such heuristics are described in [Gus94].

Experiments on shortest common superstring heuristics are reported in [RBT04], which suggest that greedy heuristics typically produce solutions within 1.4% of optimal for a reasonable class of inputs. Experiments with genetic algorithm approaches are reported in [ZS04]. Analytical results [YZ99] demonstrate very little compression on the SCS of random sequences largely because the expected overlap length of any two random strings is small.

Related Problems: Suffix trees (see page 377), text compression (see page 637).