18

Set and String Problems

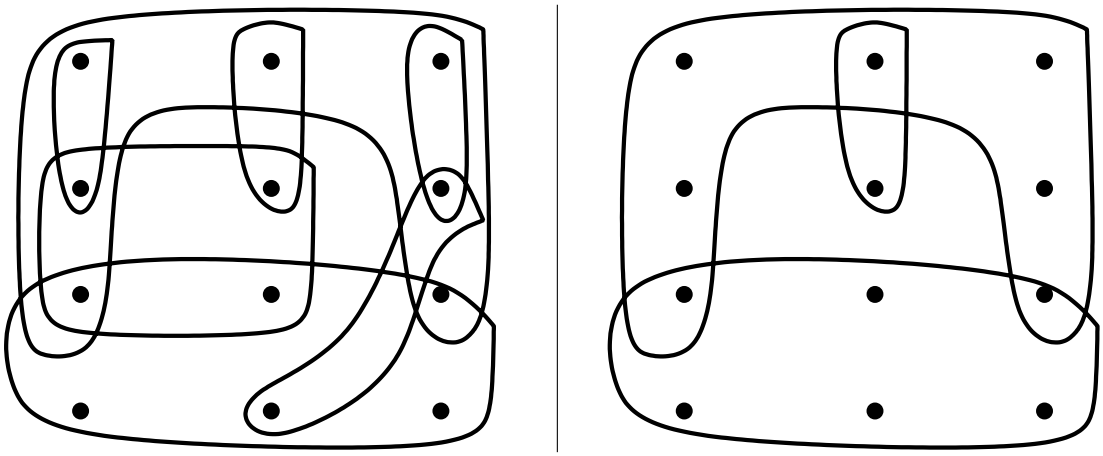
Sets and strings both represent collections of objects—the difference is whether order matters. Sets are collections of symbols whose order is assumed to carry no significance, while strings are defined by the sequence or arrangement of symbols.

The assumption of a fixed order makes it possible to solve string problems much more efficiently than set problems, through techniques such as dynamic programming and advanced data structures like suffix trees. The interest in and importance of string-processing algorithms have been increasing due to bioinformatics, Web searches, and other text-processing applications. Recent books on string algorithms include:

- *Gusfield* [Gus97] – To my taste, this remains is the best introduction to string algorithms. It contains a thorough discussion on suffix trees, with clear and innovative formulations of classical exact string-matching algorithms.
- *Crochemore, Hancart, and Lecroq* [CHL07] – A comprehensive treatment of string algorithms, written by a true leader in the field. Translated from the French, but clear and accessible.
- *Navarro and Raffinot* [NR07] – A concise but practical and implementation-oriented treatment of pattern-matching algorithms, with particularly thorough treatment of bit-parallel approaches.
- *Crochemore and Rytter* [CR03] – A survey of specialized topics in string algorithmics emphasizing theory.

Theoreticians working in string algorithmics sometimes refer to their field as *Stringology*. The annual *Combinatorial Pattern Matching* (CPM) conference is the primary venue devoted to both practical and theoretical aspects of string algorithmics and related areas.

S.S. Skiena, *The Algorithm Design Manual*, 2nd ed., DOI: 10.1007/978-1-84800-070-4_18,



INPUT OUTPUT

18.1 Set Cover

Input description: A collection of subsets $S = \{S_1, \ldots, S_m\}$ of the universal set $U = \{1, \ldots, n\}$.

Problem description: What is the smallest subset T of S whose union equals the universal set—i.e. , $\cup_{i=1}^{|T|} T_i = U$?

Discussion: Set cover arises when you try to efficiently acquire items that have been packaged in a fixed set of lots. You seek a collection of at least one of each distinct type of item, while buying as few lots as possible. Finding *a* set cover is easy, because you can always buy one of each possible lot. However, identifying a small set cover let you do the same job for less money. Set cover provided a natural formulation of the Lotto ticket optimization problem discussed in Section 1.6 (page 23). There we seek to buy the smallest number of tickets needed to cover all of a given set of combinations.

Boolean logic minimization is another interesting application of set cover. We are given a particular Boolean function of k variables, which describes whether the desired output is 0 or 1 for each of the 2^k possible input vectors. We seek the simplest circuit that exactly implements this function. One approach is to find a disjunctive normal form (DNF) formula on the variables and their complements, such as $x_1\bar{x}_2 + \bar{x}_1\bar{x}_2$. We could build one "and" term for each input vector and then "or" them all together, but we might save considerably by factoring out common subsets of variables. Given a set of feasible "and" terms, each of which covers a

subset of the vectors we need, we seek to "or" together the smallest number of terms that realize the function. This is exactly the set cover problem.

There are several variations of set cover problems to be aware of:

- *Are you allowed to cover elements more than once?* – The distinction here is between *set cover* and *set packing*, which is discussed in Section 18.2 (page 625). We should take advantage of the freedom to cover elements multiple times if we have it, as it usually results in a smaller covering.

- *Are your sets derived from the edges or vertices of a graph?* – Set cover is a very general problem, and includes several useful graph problems as special cases. Suppose instead that you seek the smallest set of edges in a graph that will cover each vertex at least once. The solution is the maximum *matching* in the graph (see Section 15.6 (page 498)), plus arbitrary edges to cover any unmatched vertices. Now suppose instead that you seek the smallest set of vertices that cover each edge at least once. This is the *vertex cover* problem, discussed in Section 16.3 (page 530).

  It is instructive to show how to model vertex cover as an instance of set cover. Let the universal set U correspond to the set of edges $\{e_1, \ldots, e_m\}$. Construct n subsets, with S_i consisting of the edges incident on vertex v_i. Although vertex cover is just an instance of set cover in disguise, you should take advantage of the superior heuristics that exist for the more restricted vertex cover problem.

- *Do your subsets contain only two elements each?* – You are in luck if all of your subsets have at most two elements each. This special case can be solved efficiently to optimality because it reduces to finding a maximum matching in a graph. Unfortunately, the problem becomes NP-complete as soon as your subsets have three elements each.

- *Do you want to cover elements with sets, or sets with elements?* – In the *hitting set* problem, we seek a small number of items that together represent each subset in a given population. Hitting set is illustrated in Figure 18.1. The input is identical to set cover, but instead we seek the smallest subset of elements $T \subset U$ such that each subset S_i contains at least one element of T. Thus, $S_i \cap T \neq \emptyset$ for all $1 \leq i \leq m$. Suppose we desire a small Congress with at least one representative for each ethnic group. If each ethnic group is defined by a subset of people, the minimum hitting set gives the smallest possible politically correct Congress.

  Hitting set is *dual* to set cover, meaning that it is exactly the same problem in disguise. Replace each element of U by a set of the names of the subsets that contain it. Now S and U have exchanged roles, for we seek a set of subsets from U to cover all the elements of S. This is exactly set cover, so we can use any set cover code to solve hitting set after performing this simple translation. See Figure 18.1 for an example.

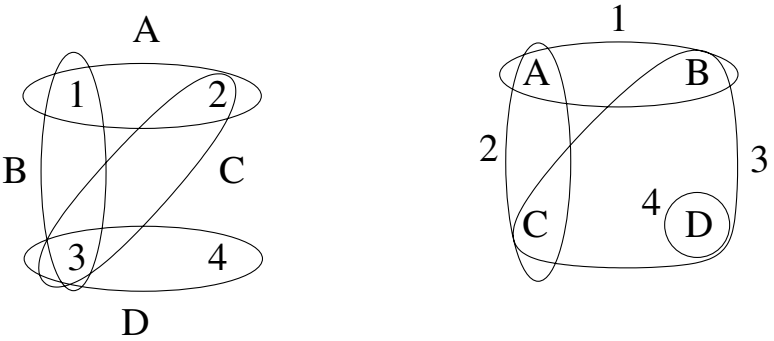


Figure 18.1: A hitting set instance optimally solved by selecting elements 1 and 3 or 2 and 3 (l). This problem converted to a dual set cover instance optimally solved by selecting subsets 1 and 3 or 2 and 4 (r).

Set cover must be at least as hard as vertex cover, so it is also NP-complete. In fact, it is somewhat harder. Approximation algorithms do no worse than twice optimal for vertex cover, but the best approximation algorithm for set cover is $\Theta(\lg n)$ times optimal.

Greedy is the most natural and effective heuristic for set cover. Begin by selecting the largest subset for the cover, and then delete all its elements from the universal set. We add the subset containing the largest number of remaining uncovered elements repeatedly until all are covered. This heuristic always gives a set cover using at most $\ln n$ times as many sets as optimal. In practice it usually does a lot better.

The simplest implementation of the greedy heuristic sweeps through the entire input instance of m subsets for each greedy step. However, by using such data structures as linked lists and a bounded-height priority queue (see Section 12.2 (page 373)), the greedy heuristic can be implemented in $O(S)$ time, where $S = \cup_{i=1}^{m} |S_i|$ is the size of the input representation.

It pays to check whether or not there exist elements that exist in only a few subsets—ideally only one. If so, we should select the biggest subsets containing these elements at the very beginning. We must take such a subset eventually, and they carry along other elements that we might have paid extra to cover if we wait until later.

Simulated annealing is likely to produce somewhat better set covers than these simple heuristics. Backtracking can be used to guarantee you an optimal solution, but it is usually not worth the computational expense.

An alternate and more powerful approach rests on the integer programming formulation of set cover. Let the integer 0-1 variable s_i denote whether subset S_i is selected for a given cover. Each universal set element x adds the constraint

$$\sum_{x \in S_i} s_i \geq 1$$

to ensure that it is covered by at least one selected subset. The minimum set cover satisfies all constraints while minimizing $\sum_i s_i$. This integer program can be easily generalized to weighted set cover (allowing nonuniform costs for different subsets. Relaxing this to a linear program (i.e. , allowing $0 \leq s_i \leq 1$ instead of constricting each variable to be either 0 or 1) allows efficient and effective heuristics using rounding techniques.

Implementations: Both the greedy heuristic and the above ILP formulation are sufficiently simple in their respective worlds that one has to implement them from scratch.

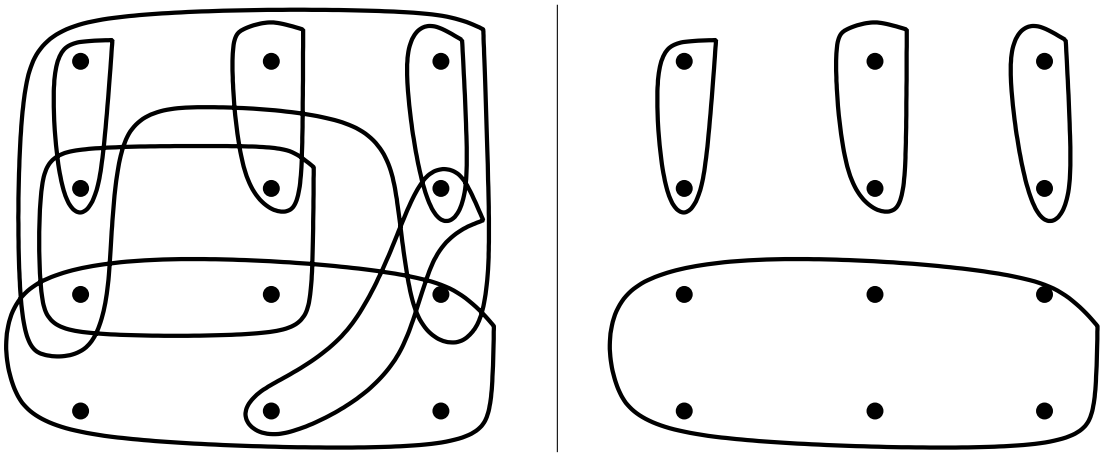
Pascal implementations of an exhaustive search algorithm for set packing, as well as heuristics for set cover, appear in [SDK83]. See Section 19.1.10 (page 662).

SYMPHONY is a mixed-integer linear programming solver that includes a set partitioning solver. It is available at *http://branchandcut.org/SPP/*

Notes: An old but classic survey article on set cover is [BP76], with more recent approximation and complexity analysis surveyed in [Pas97]. See [CFT99, CFT00] for extensive computational studies of integer programming-based set cover heuristics and exact algorithms. An excellent exposition on algorithms and reduction rules for set cover is presented in [SDK83].

Good expositions of the greedy heuristic for set cover include [CLRS01, Hoc96]. An example demonstrating that the greedy heuristic for set cover can be as bad as $\lg n$ is presented in [Joh74, PS98]. This is not a defect of the heuristic. Indeed, it is provably hard to approximate set cover to within an approximation factor better than $(1 - o(1)) \ln n$ [Fei98].

Related Problems: Matching (see page 498), vertex cover (see page 530), set packing (see page 625).



INPUT OUTPUT

18.2 Set Packing

Input description: A set of subsets $S = \{S_1, \ldots, S_m\}$ of the universal set $U = \{1, \ldots, n\}$.

Problem description: Select (an ideally small) collection of mutually disjoint subsets from S whose union is the universal set.

Discussion: Set-packing problems arise in applications where we have strong constraints on what is an allowable partition. The key feature of packing problems is that no elements are permitted to be covered by more than one selected subset.

Some flavor of this is captured by the independent set problem in graphs, discussed in Section 16.2 (page 528). There we seek a large subset of vertices from graph G such that each edge is adjacent to at most one of the selected vertices. To model this as set packing, let the universal set consist of all edges of G, and subset S_i consist of all edges incident on vertex v_i. Finally, define an additional singleton set for each edge. Any set packing defines a set of vertices with no edge in common—in other words, an independent set. The singleton sets are used to pick up any edges not covered by the selected vertices.

Scheduling airline flight crews is another application of set packing. Each airplane in the fleet needs to have a crew assigned to it, consisting of a pilot, copilot, and navigator. There are constraints on the composition of possible crews, based on their training to fly different types of aircraft, personality conflicts, and work schedules. Given all possible crew and plane combinations, each represented by a subset of items, we need an assignment such that each plane and each person is

in exactly one chosen combination. After all, the same person cannot be on two different planes simultaneously, and every plane needs a crew. We need a perfect packing given the subset constraints.

Set packing is used here to represent several problems on sets, all of which are NP-complete:

- *Must every element appear in exactly one selected subset?* – In the *exact cover* problem, we seek some collection of subsets such that each element is covered exactly once. The airplane scheduling problem above has the flavor of exact covering, since every plane and crew has to be employed.

  Unfortunately, exact cover puts us in a situation similar to that of Hamiltonian cycle in graphs. If we really *must* cover all the elements exactly once, and this existential problem is NP-complete, then all we can do is exponential search. The cost will be prohibitive unless we happen to stumble upon a solution quickly.

- *Does each element have its own singleton set?* – Things will be far better if we can be content with a partial solution, say by including each element of U as a singleton subset of S. Thus, we can expand any set packing into an exact cover by mopping up the unpacked elements of U with singleton sets. Now our problem is reduced to finding a minimum-cardinality set packing, which can be attacked via heuristics.

- *What is the penalty for covering elements twice?* – In set cover (see Section 18.1 (page 621)), there is no penalty for elements existing in many selected subsets. In exact cover, any such violation is forbidden. For many applications, the truth lies somewhere in between. Such problems can be approached by charging the greedy heuristic more to select a subset that contains previously covered elements.

The right heuristics for set packing are greedy, and similar to those of set cover (see Section 18.1 (page 621)). If we seek a packing with many (few) sets, then we repeatedly select the smallest (largest) subset, delete all subsets from S that clash with it, and repeat. As usual, augmenting this approach with some exhaustive search or randomization (in the form of simulated annealing) is likely to yield better packings at the cost of additional computation.

An alternate and more powerful approach rests on an integer programming formulation akin to that of set cover. Let the integer 0-1 variable s_i denote whether subset S_i is selected for a given cover. Each universal set element x adds the constraint

$$\sum_{x \in S_i} s_i = 1$$

to ensure that it is covered by *exactly* one selected subset. Minimizing or maximizing $\sum_i s_i$ while respecting these constraints enables us modulate the desired number of sets in the cover.

Implementations: Since set cover is a more popular and more tractable problem than set packing, it might be easier to find an appropriate implementation to solve the cover problem. Such implementations discussed in Section 18.1 (page 621) should be readily modifiable to support certain packing constraints.

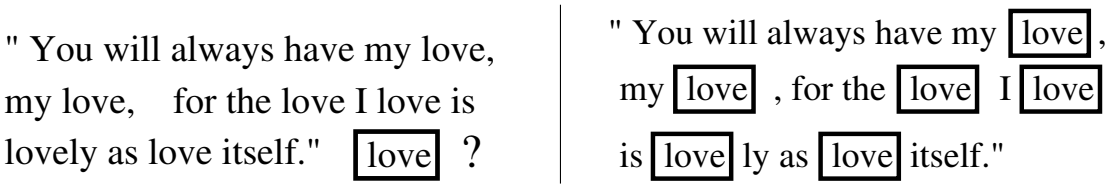
Pascal implementations of an exhaustive search algorithm for set packing, as well as heuristics for set cover, appear in [SDK83]. See Section 19.1.10 (page 662) for details on FTP-ing these codes.

SYMPHONY is a mixed-integer linear programming solver that includes a set partitioning solver. It is available at *http://branchandcut.org/SPP/*.

Notes: Survey articles on set packing include [BP76, Pas97]. Bidding strategies for combinatorial auctions typically reduce to solving set-packing problems, as described in [dVV03].

Set-packing relaxations for integer programs are presented in [BW00]. An excellent exposition on algorithms and reduction rules for set packing is presented in [SDK83], including the airplane scheduling application discussed previously.

Related Problems: Independent set (see page 528), set cover (see page 621).



18.3 String Matching

Input description: A text string t of length n. A pattern string p of length m.

Problem description: Find the first (or all) instances of pattern p in the text.

Discussion: String matching arises in almost all text-processing applications. Every text editor contains a mechanism to search the current document for arbitrary strings. Pattern-matching programming languages such as Perl and Python derive much of their power from their built-in string matching primitives, making it easy to fashion programs that filter and modify text. Spelling checkers scan an input text for words appearing in the dictionary and reject any strings that do not match.

Several issues arise in identifying the right string matching algorithm for a given application:

- *Are your search patterns and/or texts short?* – If your strings are sufficiently short and your queries sufficiently infrequent, the simple $O(mn)$-time search algorithm will suffice. For each possible starting position $1 \leq i \leq n - m + 1$, it tests whether the m characters starting from the ith position of the text are identical to the pattern. An implementation of this algorithm (in C) is given in Section 2.5.3 (page 43).

  For very short patterns (say $m \leq 5$), you can't hope to beat this simple algorithm by much, so you shouldn't try. Further, we expect much better than $O(mn)$ behavior for typical strings, because we advance the pattern the instant we observe a text/pattern mismatch, Indeed, the trivial algorithm *usually* runs in linear time. But the worst case certainly can occur, as with pattern $p = a^m$ and text $t = (a^{m-1}b)^{n/m}$.

- *What about longer texts and patterns?* – String matching can in fact be performed in worst-case linear time. Observe that we need not begin the search from scratch on finding a character mismatch, since the pattern prefix and text must exactly match up to the point of mismatch. Given a long partial

match ending at position i, we jump ahead to the first character position in the pattern/text that can provide new information about the text in position $i + 1$. The Knuth-Morris-Pratt algorithm preprocesses the search pattern to construct such a jump table efficiently. The details are tricky to get correct, but the resulting algorithm yields short, simple programs.

- *Do I expect to find the pattern or not?* – The Boyer-Moore algorithm matches the pattern against the text from right to left, and can avoid looking at large chunks of text on a mismatch. Suppose the pattern is *abracadabra*, and the eleventh character of the text is x. This pattern cannot match in any of the first eleven starting positions of the text, and so the next necessary position to test is the 22nd character. If we get very lucky, only n/m characters need ever be tested. The Boyer-Moore algorithm involves two sets of jump tables in the case of a mismatch: one based on pattern matched so far, the other on the text character seen in the mismatch.

  Although somewhat more complicated than Knuth-Morris-Pratt, it is worth it in practice for patterns of length $m > 5$, unless the pattern is expected to occur many times in the text. Its worst-case performance is $O(n + rm)$, where r is the number of occurrences of p in t.

- *Will you perform multiple queries on the same text?* – Suppose you are building a program to repeatedly search a particular text database, such as the Bible. Since the text remains fixed, it pays to build a data structure to speed up search queries. The suffix tree and suffix array data structures, discussed in Section 12.3 (page 377), are the right tools for the job.

- *Will you search many texts using the same patterns?* – Suppose you are building a program to screen out dirty words from a text stream. Here, the set of patterns remains stable, while the search texts are free to change. In such applications, we may need to find all occurrences of any of k different patterns where k can be quite large.

  Performing a linear-time scan for each pattern yields an $O(k(m + n))$ algorithm. If k is large, a better solution builds a single finite automaton that recognizes all of these patterns and returns to the appropriate start state on any character mismatch. The Aho-Corasick algorithm builds such an automaton in linear time. Space savings can be achieved by optimizing the pattern recognition automaton, as discussed in Section 18.7 (page 646). This approach was used in the original version of *fgrep*.

  Sometimes multiple patterns are specified not as a list of strings, but concisely as a regular expression. For example, the regular expression $a(a + b + c)^*a$ matches any string on (a, b, c) that begins and ends with a distinct a. The best way to test whether an input string is described by a given regular expression R constructs the finite automaton equivalent to R and then simulates

the machine on the string. Again, see Section 18.7 (page 646) for details on constructing automata from regular expressions.

When the patterns are specified by context-free grammars instead of regular expressions, the problem becomes one of parsing, discussed in Section 8.6 (page 298).

- *What if our text or pattern contains a spelling error?* – The algorithms discussed here work only for exact string matching. If you want to allow some tolerance for spelling errors, your problem becomes *approximate string matching*, which is thoroughly discussed in Section 18.4 (page 631).

Implementations: Strmat is a collection of C programs implementing exact pattern matching algorithms in association with [Gus97], including several variants of the KMP and Boyer-Moore algorithms. It is available at *http://www.cs.ucdavis.edu/~gusfield/strmat.html*.

SPARE Parts [WC04a] is a C++ string pattern recognition toolkit that provides production-quality implementations of all major variants of the classical string-matching algorithms for single patterns (both Knuth-Morris-Pratt and Boyer-Moore) and multiple patterns (both Aho-Corasick and Commentz-Walter). It is available at *http://www.fastar.org/*.

Several versions of the general regular expression pattern matcher (grep) are readily available. GNU *grep* found at *http://directory.fsf.org/project/grep/*, and supersedes variants such as *egrep* and *fgrep*. GNU *grep* uses a fast lazy-state deterministic matcher hybridized with a Boyer-Moore search for fixed strings.

The Boost string algorithms library provides C++ routines for basic operations on strings, including search. See *http://www.boost.org/doc/html/string_algo.html*.

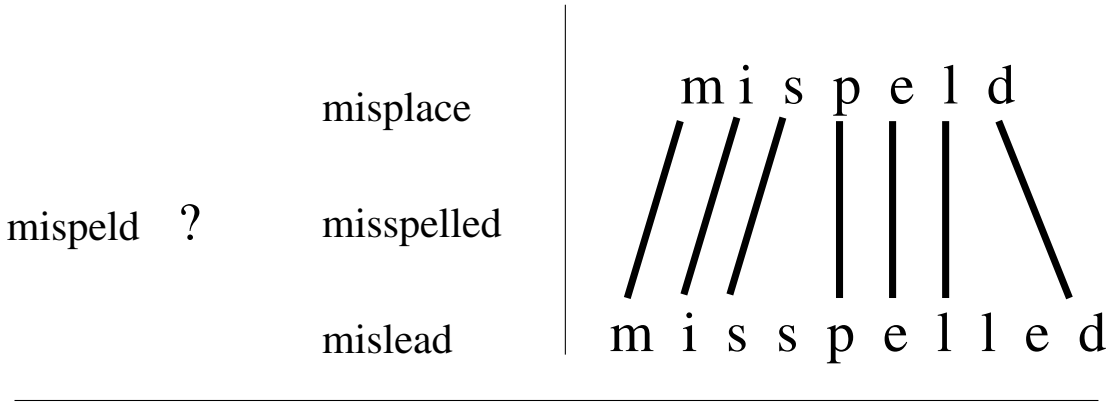
Notes: All books on string algorithms contain thorough discussions of exact string matching, including [CHL07, NR07, Gus97]. Good expositions on the Boyer-Moore [BM77] and Knuth-Morris-Pratt algorithms [KMP77] include [BvG99, CLRS01, Man89]. The history of string matching algorithms is somewhat checkered because several published proofs were incorrect or incomplete. See [Gus97] for clarification.

Aho [Aho90] provides a good survey on algorithms for pattern matching in strings, particularly where the patterns are regular expressions instead of strings. The Aho-Corasick algorithm for multiple patterns is described in [AC75].

Empirical comparisons of string matching algorithms include [DB86, Hor80, Lec95, dVS82]. Which algorithm performs best depends upon the properties of the strings and the size of the alphabet. For long patterns and texts, I recommend that you use the best implementation of Boyer-Moore that you can find.

The Karp-Rabin algorithm [KR87] uses a hash function to perform string matching in linear expected time. Its worst-case time remains quadratic, and its performance in practice appears somewhat worse than the character comparison methods described above. This algorithm is presented in Section 3.7.2 (page 91).

Related Problems: Suffix trees (see page 377), approximate string matching (see page 631).



18.4 Approximate String Matching

Input description: A text string t and a pattern string p.

Problem description: What is the minimum-cost way to transform t to p using insertions, deletions, and substitutions?

Discussion: Approximate string matching is a fundamental problem because we live in an error-prone world. Spelling correction programs must be able to identify the closest match for any text string not found in a dictionary. By supporting efficient sequence similarity (homology) searches on large databases of DNA sequences, the computer program BLAST has revolutionized the study of molecular biology. Suppose you were interested in a particular gene in man, and discovered that it is similar to the hemoglobin gene in rats. Likely this new gene also produces hemoglobin, and any differences are the result of genetic mutations during evolution.

I once encountered approximate string matching when evaluating the performance of an optical character-recognition system. We needed to compare the answers produced by our system on a test document with the correct results. To improve our system, we needed to identify (1) which letters were getting misidentified and (2) gibberish when the program found letters that didn't exist. The solution was to do an alignment between the two texts. Insertions and deletions corresponded to gibberish, while substitutions signaled errors in our recognizer. This same principle is used in file difference programs, which identify the lines that have changed between two versions of a file.

When no errors are permitted, our problem reduces to exact string matching, which is presented in Section 18.3 (page 628). Here, we restrict our discussion to matching with errors.

Dynamic programming provides the basic approach to approximate string matching. Let $D[i,j]$ denote the cost of editing the first i characters of the pattern string p into the first j characters of the text t. The recurrence follows because we must have done *something* with the tail characters p_i and t_j. Our only options are matching / substituting one for the other, deleting p_i, or inserting a match for t_j. Thus, $D[i,j]$ is the minimum of the costs of these possibilities:

1. If $p_i = t_j$ then $D[i-1,j-1]$ else $D[i-1,j-1]$ + substitution cost.
2. $D[i-1,j]$ + deletion cost of p_i.
3. $D[i,j-1]$ + deletion cost of t_j.

A general implementation in C and more complete discussion appears in Section 8.2 (page 280). Several issues remain before we can make full use of this recurrence:

- *Do I match the pattern against the full text, or against a substring?* – The boundary conditions of this recurrence distinguishes between algorithms for string matching and substring matching. Suppose we want to align the full pattern against the full text. Then the cost of $D[i,0]$ must be that of deleting the first i characters of the pattern, so $D[i,0] = i$. Similarly, $D[0,j] = j$.

  Now suppose that the pattern can occur anywhere within the text. The proper cost of $D[0,j]$ is now 0, since there should be no penalty for starting the alignment in the jth position of the text. The cost of $D[i,0]$ remains i, because the only way to match the first i pattern characters with nothing is to delete all of them. The cost of the best substring pattern match against the text will be given by $\min_{k=1}^{n} D[m,k]$.

- *How should I select the substitution and insertion/deletion costs?* – The basic algorithm can be easily modified to use different costs for insertion, deletion, and the substitutions of specific pairs of characters. Which costs you should use depend on what you are planning to do with the alignment.

  The most common cost assignment charges the same for insertion, deletion, or substitution. Charging a substitution cost of more than insertion + deletion ensures that substitutions never get performed, since it will always be cheaper to edit both characters out of the string. If we just have insertion and deletion to work with, the problem reduces to *longest common subsequence*, discussed in Section 18.8 (page 650). It often pays to tweak the edit distance costs and study the resulting alignments until you find the best parameters for the job.

- *How do I find the actual alignment of the strings?* – As thus far described, the recurrence only gives the cost of the optimal string/pattern alignment, not the sequence of editing operations to achieve it. To obtain such a transcript, we can work backwards from the complete cost matrix D. We had to come from one of $D[m-1,n]$ (pattern deletion/text insertion), $D[m,n-1]$

(text deletion/pattern insertion), or $D[m-1, n-1]$ (substitution/match) to get to cell $D[m, n]$. The option which was chosen can be reconstructed from these costs and the given characters p_m and t_n. By continuing to work backwards to the previous cell, we can reconstruct the entire alignment. Again, an implementation in C appears in Section 8.2 (page 280).

- *What if the two strings are very similar to each other?* – The dynamic programming algorithm finds a shortest path across an $m \times n$ grid, where the cost of each edge depends upon which operation it represents. To seek an alignment involving a combination of at most d insertions, deletions, and substitutions, we need only traverse the band of $O(dn)$ cells within a distance d of the central diagonal. If no low-cost alignment exists within this band, then no low-cost alignment can exist in the full cost matrix.

  Another idea we can use is *filtration*, quickly eliminating the parts of the string where there is no hope of finding the pattern. Carve the m-length pattern into $d+1$ pieces. If there is a match with at most d differences, then at least one of these pieces must be an exact match in the optimal alignment. Thus, we can identify all possible approximate match points by conducting an exact multi-pattern search on the pieces, and then evaluate only the possible candidates more carefully.

- *Is your pattern short or long?* – A recent approach to string-matching exploits the fact that modern computers can do operations on (say) 64-bit words in a single gulp. This is long enough to hold eight 8-bit ASCII characters, providing motivation to design *bit-parallel algorithms*, which do more than one comparison with each operation.

  The basic idea is quite clever. Construct a bit-mask B_α for each letter α of the alphabet, such that ith-bit $B_\alpha[i] = 1$ iff the ith character of the pattern is α. Now suppose you have a match bit-vector M_j for position j in the text string, such that $M_j[i] = 1$ iff the first i bits of the pattern exactly match the $(j-i+1)$st through jth character of the text. We can find *all* the bits of M_{j+1} using just two operations by (1) shifting M_j one bit to the right, and then (2) doing a bitwise AND with B_α, where α is the character in position $j+1$ of the text.

  The *agrep* program, discussed below, uses such a bit-parallel algorithm generalized to approximate matching. Such algorithms are easy to program and many times faster than dynamic programming.

- *How can I minimize the required storage?* – The quadratic space used to store the dynamic programming table is usually a more serious problem than its running time. Fortunately, only $O(\min(m, n))$ space is needed to compute $D[m, n]$. We need only maintain two active rows (or columns) of the matrix to compute the final value. The entire matrix will be required only if we need to reconstruct the actual sequence alignment.

We can use Hirschberg's clever recursive algorithm to efficiently recover the optimal alignment in linear space. During one pass of the linear-space algorithm above to compute $D[m, n]$, we identify which middle-element cell $D[m/2, x]$ was used to optimize $D[m, n]$. This reduces our problem to finding the best paths from $D[1, 1]$ to $D[m/2, x]$ and from $D[m/2, x]$ to $D[m/2, n]$, both of which can be solved recursively. Each time we remove half of the matrix elements from consideration, so the total time remains $O(mn)$. This linear-space algorithm proves to be a big win in practice on long strings, although it is somewhat more difficult to program.

- *Should I score long runs of indels differently?* – Many string matching applications look more kindly on alignments where insertions/deletions are bunched in a small number of runs or gaps. Deleting a word from a text should presumably cost less than a similar number of scattered single-character deletions, because the word represents a single (albeit substantial) edit operation.

  String matching with *gap penalties* provides a way to properly account for such operations. Typically, we assign a cost of $A + Bt$ for each indel of t consecutive characters, where A is the cost of starting the gap and B is the per-character deletion cost. If A is large relative to B, the alignment has incentive to create relatively few runs of deletions.

  String matching under such *affine* gap penalties can be done in the same quadratic time as regular edit distance. We will use separate insertion and deletion recurrences E and F to encode the cost of being in gap mode, meaning we have already paid the cost of initiating the gap:

$$V(i, j) = \max(E(i, j), F(i, j), G(i, j))$$

$$G(i, j) = V(i - 1, j - 1) + match(i, j)$$

$$E(i, j) = \max(E(i, j - 1), V(i, j - 1) - A) - B$$

$$F(i, j) = \max(F(i - 1, j), V(i - 1, j) - A) - B$$

  With a constant amount of work per cell, this algorithm takes $O(mn)$ time, same as without gap costs.

- *Does similarity mean strings that sound alike?* – Other models of approximate pattern matching become more appropriate for certain applications. Particularly interesting is *Soundex*, a hashing scheme that attempts to pair up English words that sound alike. This can be useful in testing whether two names that have been spelled differently are likely to be the same. For example, my last name is often spelled "Skina", "Skinnia", "Schiena", and occasionally "Skiena." All of these hash to the same Soundex code, *S25*.

The algorithm drops vowels and silent letters, removes doubled letters, and then assigns the remaining letters numbers from the following classes: *BFPV* gets a 1, *CGJKQSXZ* gets a 2, *DT* gets a 3, *L* gets a 4, *MN* gets a 5, and *R* gets a 6. The code starts with the first letter and contains at most three digits. Although this sounds very hokey, experience shows that it works reasonably well. Experience indeed: Soundex has been used since the 1920's.

Implementations: Several excellent software tools are available for approximate pattern matching. Manber and Wu's *agrep* [WM92a, WM92b] (approximate general regular expression pattern matcher) is a tool supporting text search with spelling errors. A recent version is available from *http://www.tgries.de/agrep/*. Navarro's *nrgrep* [Nav01b] combines bit-parallelism and filtration, resulting in running times that are more constant than *agrep*, although not always faster. It is available at *http://www.dcc.uchile.cl/~gnavarro/software/*.

TRE is a general regular-expression matching library for exact and approximate matching, which is more general than *agrep*. The worst-case complexity is $O(nm^2)$, where m is the list of the regular expressions involved. *TRE* is available at *http://laurikari.net/tre/*.

Wikipedia gives programs for computing edit (Levenshtein) distance in a dizzying array of languages (including Ada, C++, Emacs Lisp, Io, JavaScript, Java, PHP, Python, Ruby VB, and C#) Check it out at:

http://en.wikibooks.org/wiki/Algorithm_implementation/Strings/Levenshtein_distance

Notes: There have been many recent advances in approximate string matching, particularly in bit-parallel algorithms. Navarro and Raffinot [NR07] is the best reference on these recent techniques, which are also treated in other recent books on string algorithmics [CHL07, Gus97]. String matching with gap penalties is particularly well treated in [Gus97].

The basic dynamic programming alignment algorithm is attributed to [WF74], although it is apparently folklore. The wide range of applications for approximate string matching was made apparent in Sankoff and Kruskal's book [SK99], which remains a useful historical reference for the problem. Surveys on approximate pattern matching include [HD80, Nav01a]. The edit distance between two strings is sometimes referred to as the *Levenshtein distance*. Expositions of Hirschberg's linear-space algorithm [Hir75] include [CR03, Gus97].

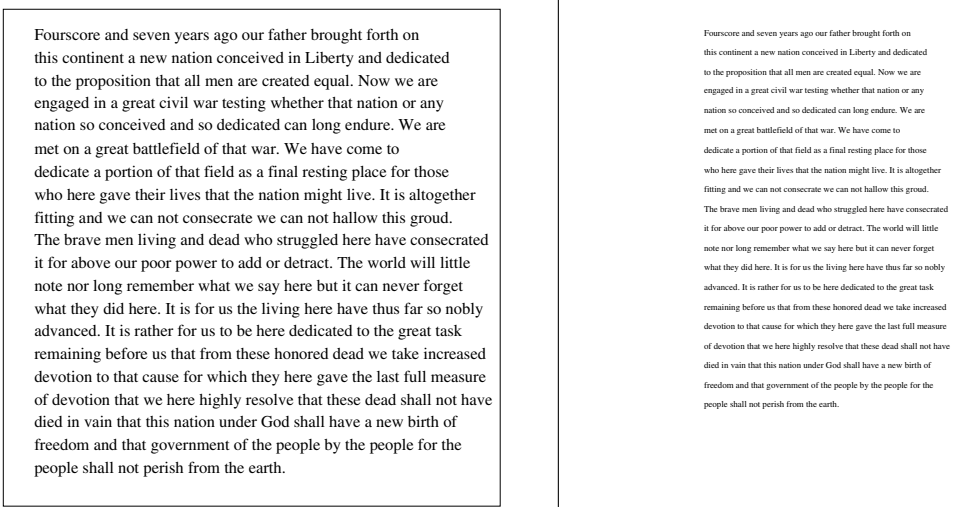
Masek and Paterson [MP80] compute the edit distance between m- and n-length strings in time $O(mn/\log(\min\{m,n\}))$ for constant-sized alphabets, using ideas from the four Russians algorithm for Boolean matrix multiplication [ADKF70].

The shortest path formulation leads to a variety of algorithms that are good when the edit distance is small, including an $O(n \lg n + d^2)$ algorithm due to Myers [Mye86] and an $O(dn)$ algorithm due to Landau and Vishkin [LV88]. Longest increasing subsequence can be done in $O(n \lg n)$ time [HS77], as presented in [Man89].

Bit-parallel algorithms for approximate matching include Myers's [Mye99b] algorithm for approximate matching in $O(mn/w)$ time, where w is the number of bits in the computer word. Experimental studies of bit-parallel algorithms include [FN04, HFN05, NR00].

Soundex was invented and patented by M. K. Odell and R. C. Russell. Expositions on Soundex include [BR95, Knu98]. Metaphone is a recent attempt to improve on Soundex [BR95, Par90]. See [LMS06] for an application of such phonetic hashing techniques to the problem entity name unification.

Related Problems: String matching (see page 628), longest common substring (see page 650).



INPUT OUTPUT

18.5 Text Compression

Input description: A text string S.

Problem description: Create a shorter text string S' such that S can be correctly reconstructed from S'.

Discussion: Secondary storage devices fill up quickly on every computer system, even though their capacity continues to double every year. Decreasing storage prices only seem to have increased interest in data compression, probably because there is more data to compress than ever before. *Data compression* is the algorithmic problem of finding space-efficient encodings for a given data file. The rise of computer networks provided a new mission for data compression, that of increasing the effective network bandwidth by reducing the number of bits before transmission.

People seem to *like* inventing ad hoc data-compression methods for their particular application. Sometimes these outperform general methods, but often they don't. Several issues arise in selecting the right compression algorithm:

- *Must we recover the exact input text after compression? – Lossy* versus *lossless* encoding is the primary issue in data compression. Document storage applications typically demand lossless encodings, as users become disturbed

whenever their data files are altered. Fidelity is not such an issue in image or video compression, because the presence of small artifacts are imperceptible to the viewer. Significantly greater compression ratios can be obtained using lossy compression, which is why most image/video/audio compression algorithms exploit this freedom.

- *Can I simplify my data before I compress it?* – The most effective way to free space on a disk is to delete files you don't need. Likewise, any preprocessing you can do to reduce the information content of a file pays off later in better compression. Can we eliminate redundant white space from the file? Might the document be converted entirely to uppercase characters, or have formatting information removed?

  A particularly interesting simplification results from applying the *Burrows-Wheeler transform* to the input string. This transform sorts all n cyclic shifts of the n character input, and then reports the last character of each shift. As an example, the cyclic shifts of $ABAB$ are $ABAB$, $BABA$, $ABAB$, and $BABA$. After sorting, these become $ABAB$, $ABAB$, $BABA$, and $BABA$. Reading the last character of each of these strings yields the transform result: $BBAA$.

  Provided the last character of the input string is unique (e.g., end-of-string), this transform is perfectly reversible to the original input! The Burrows-Wheeler string is typically 10-15% more compressible than the original text, because repeated words turn into blocks of repeated characters. Further, this transform can be computed in linear time.

- *Does it matter whether the algorithm is patented?* – Certain data compression algorithms have been patented—most notoriously the LZW variation of the Lempel-Ziv algorithm discussed below. Mercifully, this patent has now expired, although legal battles are still being fought over JPEG. Typically there are unrestricted variations of any compression algorithm that perform about as well as the patented variant.

- *How do I compress image data* – The simplest lossless compression algorithm for image data is *run-length coding*. Here we replace runs of identical pixel values with a single instance of the pixel and an integer giving the length of the run. This works well on binary images with large contiguous regions of similar pixels, like scanned text. It performs badly on images with many quantization levels and random noise. Correctly selecting (1) the number of bits to allocate to the count field, and (2) the right traversal order to reduce a two-dimensional image into a stream of pixels, has a surprisingly important impact on compression.

  For serious audio/image/video compression applications, I recommend that you use a popular lossy coding method and not fool around with implementing it yourself. JPEG is the standard high-performance image compression

method, while MPEG is designed to exploit the frame-to-frame coherence of video.

- *Must compression run in real time?* – Fast decompression is often more important than fast compression. A YouTube video is compressed only once, but decompressed every time someone plays it. In contrast, an operating system that increases effective disk capacity by automatically compressing files will need a symmetric algorithm with fast compression times, as well.

Literally dozens of text compression algorithms are available, but they can be classified into two distinct approaches. *Static algorithms*, such as Huffman codes, build a single coding table by analyzing the entire document. *Adaptive algorithms*, such as Lempel-Ziv, build a coding table on the fly that adapts to the local character distribution of the document. Adaptive algorithms usually prove to be the correct answer:

- *Huffman codes* – Huffman codes replace each alphabet symbol by a variable-length code string. Using eight bits-per-symbol to encode English text is wasteful, since certain characters (such as "e") occur far more frequently than others (such as "q"). Huffman codes assign "e" a short code word, and "q" a longer one to compress text.

  Huffman codes can be constructed using a greedy algorithm. Sort the symbols in increasing order by frequency. We merge the two least-frequently used symbols x and y into a new symbol xy, whose frequency is the sum of the frequencies of its two child symbols. Replacing x and y by xy leaves a smaller set of symbols. We now repeat this operation $n - 1$ times until all symbols have been merged. These merging operations define a rooted binary tree, with the original alphabet symbols as leaves. The left or right choices on the root-to-leaf path define the bits of the binary code word for each symbol. Priority queues can efficiently maintain the symbols by frequency during construction, yielding Huffman codes in $O(n \lg n)$ time.

  Huffman codes are popular but have three disadvantages. Two passes must be made over the document on encoding, first to build the coding table, and then to actually encode the document. The coding table must be explicitly stored with the document to decode it, which eats into any space savings on short documents. Finally, Huffman codes only exploit nonuniform symbol distributions, while adaptive algorithms can recognize the higher-order redundancies such as in *0101010101...*.

- *Lempel-Ziv algorithms* – Lempel-Ziv algorithms (including the popular LZW variant) compress text by building a coding table on the fly as we read the document. The coding table changes at every position in the text. A clever protocol ensures that the encoder and decoder are both always working with the exact same code table, so no information is lost.

> Lempel-Ziv algorithms build coding tables of frequent substrings, which can get arbitrarily long. Thus they can exploit often-used syllables, words, and phrases to build better encodings. It adapts to local changes in the text distribution, which is important because many documents exhibit significant locality of reference.
>
> The truly amazing thing about the Lempel-Ziv algorithm is how robust it is on different types of data. It is quite difficult to beat Lempel-Ziv by using an application-specific algorithm. My recommendation is not to try. If you can eliminate application-specific redundancies with a simple preprocessing step, go ahead and do it. But don't waste much time fooling around. You are unlikely to get significantly better text compression than with *gzip* or some other popular program, and you might well do worse.

Implementations: Perhaps the most popular text compression program is *gzip*, which implements a public domain variation of the Lempel-Ziv algorithm. It is distributed under the GNU software license and can be obtained from *http://www.gzip.org*.

There is a natural tradeoff between compression ratio and compression time. Another choice is *bzip2*, which uses the Burrows-Wheeler transform. It produces tighter encodings than *gzip* at somewhat greater cost in running time. Going to the extreme, other compression algorithms devote enormous run times to squeeze every bit out of a file. Representative programs of this genre are collected at *http://www.cs.fit.edu/~mmahoney/compression/*.

Reasonably authoritative comparisons of compression programs are presented at *http://www.maximumcompression.com/*, including links to all available software.

Notes: A large number of books on data compression are available. Recent and comprehensive books include Sayood [Say05] and Salomon [Sal06]. Also recommended is the older book by Bell, Cleary, and Witten [BCW90]. Surveys on text compression algorithms include [CL98].

Good expositions on Huffman codes [Huf52] include [AHU83, CLRS01, Man89]. The Lempel-Ziv algorithm and variants are described in [Wel84, ZL78]. The Burrows-Wheeler transform was introduced in [BW94].

The annual IEEE Data Compression Conference (*http://www.cs.brandeis.edu/~dcc/*) is the primary research venue in this field. This is a mature technical area where most current work is shooting for fairly marginal improvements, particularly in the case of text compression. More encouragingly, we note that the conference is held annually at a world-class ski resort in Utah.

Related Problems: Shortest common superstring (see page 654), cryptography (see page 641).

The magic words are Squeamish Ossifrage.	I5&AE<&UA9VEC'=0 <F1s"F%R92!3<75E96UI<V V@*3W-S:69R86=E+@K_

INPUT OUTPUT

18.6 Cryptography

Input description: A plaintext message T or encrypted text E, and a key k.

Problem description: Encode T (decode E) using k giving E (T).

Discussion: Cryptography has grown substantially in importance as computer networks make confidential documents more vulnerable to prying eyes. Cryptography increases security by making messages difficult to read even if they fall into the wrong hands. Although the discipline of cryptography is at least two thousand years old, its algorithmic and mathematical foundations have only recently solidified to the point where provably secure cryptosystems can be envisioned.

Cryptographic ideas and applications go beyond the commonly known concepts of "encryption" and "authentication." The field now includes such important mathematical constructs such as cryptographic hashes, digital signatures, and useful primitive protocols that provide associated security assurances.

There are three classes of cryptosystems everyone should be aware of:

- *Caesar shifts* – The oldest ciphers involve mapping each character of the alphabet to a different letter. The weakest such ciphers rotate the alphabet by some fixed number of characters (often 13), and thus have only 26 possible keys. Better is to use an arbitrary permutation of the letters, giving 26! possible keys. Even so, such systems can be easily attacked by counting the frequency of each symbol and exploiting the fact that "e" occurs more often than "z". While there are variants that will make this more difficult to break, none will be as secure as AES or RSA.

- *Block Shuffle Ciphers* – This class of algorithms repeatedly shuffle the bits of your text as governed by the key. The classic example of such a cipher is the *Data Encryption Standard* (DES). Although approved as a Federal Information Processing Standard in 1976, its 56-bit key length is now considered too short for applications requiring substantial levels of security. Indeed, a special purpose machine named "Deep Crack" demonstrated that it is possible to decrypt messages without a key in less than a day. As of May 19,

2005, *DES* has been officially withdrawn as a federal standard, replaced by the stronger *Advanced Encryption Standard* (AES).

However, a simple variant called *triple DES* permits an effective key length of 112 bits by using three rounds of DES with two 56-bit keys. In particular, first encrypt with key1, then *decrypt* with key2, before finally encrypting with key1. There is a mathematical reason for using three rounds instead of two; the encrypt-decrypt-encrypt pattern is used so that the scheme is equivalent to single DES when $key1 = key2$. This is enough to keep "Deep Crack" at bay. Indeed, *triple DES* has recently been approved by the National Institute of Standards and Technology (NIST) for sensitive government information through the year 2030.

- *Public Key Cryptography* – If you fear bad guys reading your messages, you should be afraid to tell anyone else the key needed to decrypt them. Public-key systems use different keys to encode and decode messages. Since the encoding key is of no help in decoding, it can be made public at no risk to security. This solution to the key distribution problem is literally its key to success.

  RSA is the classic example of a public key cryptosystem, named after its inventors Rivest, Shamir, and Adelman. The security of RSA is based on the relative computational complexities of factoring and primality testing (see Section 13.8 (page 420)). Encoding is (relatively) fast because it relies on primality testing to construct the key, while the hardness of decryption follows from that of factoring. Still, RSA is slow relative to other cryptosystems—roughly 100 to 1,000 times slower than DES.

The critical issue in selecting a cryptosystem is identifying your paranoia level—i.e. , deciding how much security you need. Who are you trying to stop from reading your stuff: your grandmother, local thieves, the Mafia, or the NSA? If you can use an accepted implementation of AES or RSA, you should feel pretty safe against anybody, at least for now. Increasing computer power often lays waste to cryptosystems surprisingly quickly; recall that DES lived less than 30 years as a strong system. Be sure to use the longest possible keys and keep abreast of algorithmic development if you are a planning long-term storage of criminal material.

That said, I will confess that I use DES to encrypt my final exam each semester. It proved more than sufficient the time an ambitious student broke into my office looking for it. The story would have been different had the NSA had been breaking in, but it is important to understand that *the most serious security holes are human, not algorithmic*. Ensuring that your password is long enough, hard to guess, and not written down is far more important than obsessing about the encryption algorithm.

Most symmetric key encryption mechanisms are harder to crack than public key ones for the same key size. This means one can get away with much shorter key lengths for symmetric key than for public key encryption. NIST and RSA Labs

both provide schedules of recommended key sizes for secure encryption, and as of this writing they recommend 80-bit symmetric keys as equivalent to 1024-bit asymetric keys. This difference helps explain why symmetric key algorithms are typically orders of magnitude faster than public key algorithms.

Simple ciphers like the Caesar shift are fun and easy to program. For this reason, it is healthy to use them for applications needing only a casual level of security (such as hiding the punchlines of jokes). Since they are easy to break, they should never be used for serious security applications.

Another thing you should *never* do is try to develop your own novel cryptosystem. The security of triple DES and RSA is accepted because these systems have survived many years of public scrutiny. In this time, many other cryptosystems have been proposed, proven vulnerable to attack, and then abandoned. This is not a field for amateurs. If you are charged with implementing a cryptosystem, carefully study a respected program such as PGP to see how they handle issues such as key selection and key distribution. Any cryptosystem is as strong as its weakest link.

Certain other problems related to cryptography arise often in practice:

- *How can I validate the integrity of data against random corruption?* – There is often a need to validate that transmitted data is identical to that which has been received. One solution is for the receiver to transmit the data back to the source and have the original sender confirm that the two texts are identical. This fails when the exact inverse of an error is made in the retransmission, but a more serious problem is that your available bandwidth is cut in half with such a scheme.

  A more efficient method uses a *checksum*, a simple mathematical function that hashes a long text down to a simple number or digit. We then transmit the checksum along with the text. The checksum can be recomputed on the receiving end and bells set off if the computed checksum is not identical to what was received. The simplest checksum scheme just adds up the byte or character values and takes the sum modulo of some constant, say $2^8 = 256$. Unfortunately, an error transposing two or more characters would go undetected under such a scheme, since addition is commutative.

  Cyclic-redundancy check (CRC) provides a more powerful method for computing checksums that is used in most communications systems and internally in computers to validate disk drive transfers. These codes compute the remainder in the ratio of two polynomials, the numerator of which is a function of the input text. The design of these polynomials involves considerable mathematical sophistication, but ensures that all reasonable errors are detected. The details of efficient computation are sufficiently complicated that we recommend that you start from an existing implementation, described below.

- *How can I validate the integrity of data against deliberate corruption?* – CRC is good at detecting random errors, but not malicious changes to a document. *Cryptographic hashing functions* such as MD5 and SHA-256 are (in principle) easy to compute for a document but hard to invert. This means that given a particular hash code value x, it is difficult to construct a document d such that $H(d) = x$. The property makes them valuable for digital signatures and other applications.

- *How can I prove that a file has not been changed?* – If I send you a contract in electronic form, what is to stop you from editing the file and then claiming that your version was what we had really agreed to? I need a way to prove that any modification to a document is fraudulent. *Digital signatures* are a cryptographic way for me to stamp my document as genuine.

  Given a file, I can compute a checksum for it, and then encrypt this checksum using my own private key. I send you the file and the encrypted checksum. You can now edit the file, but to fool the judge you must also edit the encrypted checksum such that it can be decrypted to yield the correct checksum. With a suitably good checksum function, designing a file that yields the same checksum becomes an insurmountable problem. For full security, we need a trusted third party to authenticate the timestamp and associate the private key with me.

- *How can I restrict access to copyrighted material?* – An important emerging application for cryptography is digital rights management for audio and video. A key issue here is speed of decryption, as it must keep up with data transmission or retrieval in real time. Such *stream ciphers* usually involve efficiently generating a stream of pseudorandom bits, say using a shift-register generator. The exclusive-or of these bits with the data stream gives the encrypted sequence. The original data is recovered by exclusive-oring the result with the same stream of pseudorandom bits.

  High-speed cryptosystems have proven to be relatively easy to break. The state-of-the-art solution to this problem involves erecting laws like the Digital Millennium Copyright Act to make it illegal to try to break them.

Implementations: *Nettle* is a comprehensive low-level cryptographic library in C. Cryptographic hash functions include MD5 and SHA-256. Block ciphers include DES, AES, and some more recently developed codes. An implementation of RSA is also provided. *Nettle* is available at *http://www.lysator.liu.se/∼nisse/nettle*.

A comprehensive overview of cryptographic algorithms with assessments of strength is available at *http://www.cryptolounge.org/wiki/Category:Algorithm*. See *http://csrc.nist.gov/groups/ST/toolkit* for related cryptographic resources provided by NIST.

Crypto++ is a large C++ class library of cryptographic schemes, including all we have mentioned in this section. It is available at *http://www.cryptopp.com/*.

Many popular open source utilities employ serious cryptography, and serve as good models of current practice. *GnuPG*, an open source version of PGP, is available at *http://www.gnupg.org/*. *OpenSSL*, for authenticating access to computer systems, is available at *http://www.openssl.org/*.

The *Boost CRC Library* provides multiple implementations of cyclic redundancy check algorithms. It is available at *http://www.boost.org/libs/crc/*.

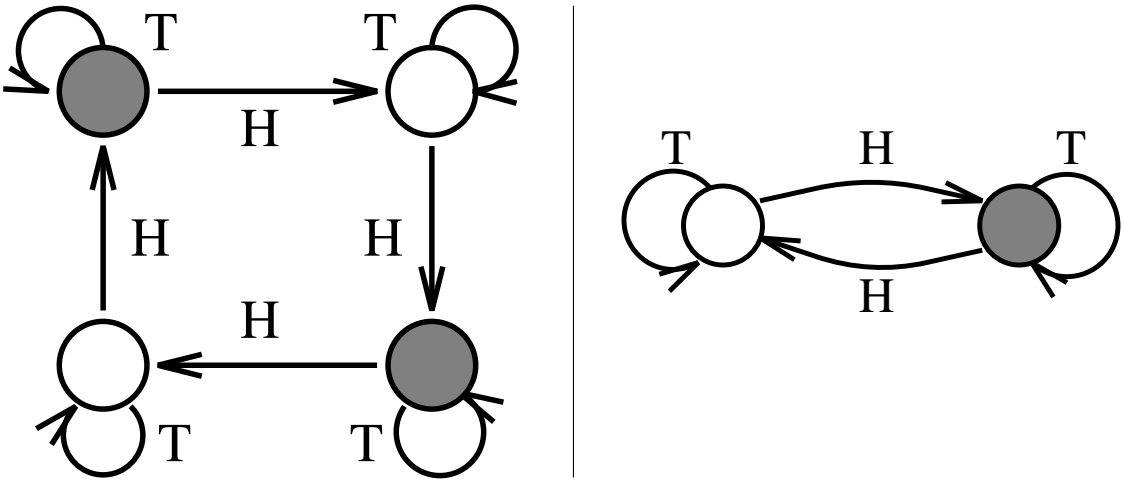
Notes: The *Handbook of Applied Cryptography* [MOV96] provides technical surveys of all aspects of cryptography, and has been generously made available online at *http://www.cacr.math.uwaterloo.ca/hac/*. Schneier [Sch96] provides a thorough overview of different cryptographic algorithms, with [FS03] as perhaps a better introduction. Kahn [Kah67] presents the fascinating history of cryptography from ancient times to 1967 and is particularly noteworthy in light of the secretive nature of the subject.

Expositions on the RSA algorithm [RSA78] include [CLRS01]. The RSA Laboratories home page *http://www.rsa.com/rsalabs/* is very informative.

Of course, the NSA (National Security Agency) is the place to go to learn the real state of the art in cryptography. The history of DES is well presented in [Sch96]. Particularly controversial was the decision by the NSA to limit key length to 56 bits.

MD5 [Riv92] is the hashing function used by PGP to compute digital signatures. Expositions include [Sch96, Sta06]. Serious problems with the security of MD5 have recently been exposed [WY05]. The SHA family of hash functions appears more secure, particularly SHA-256 and SHA-512.

Related Problems: Factoring and primality testing (see page 420), text compression (see page 637)).



INPUT OUTPUT

18.7 Finite State Machine Minimization

Input description: A deterministic finite automaton M.

Problem description: Create the smallest deterministic finite automaton M' such that M' behaves identically to M.

Discussion: Constructing and minimizing finite state machines arises repeatedly in software and hardware design applications. Finite state machines are very useful for specifying and recognizing patterns. Modern programming languages such as Java and Python provide built-in support for *regular expressions*, a particularly natural way of defining automata. Control systems and compilers often use finite state machines to encode the current state and possible associated actions/transitions. Minimizing the size of these automata reduces both the storage and execution costs of dealing with such machines.

Finite state machines are defined by directed graphs. Each vertex represents a state, and each character-labeled edge defines a transition from one state to another on receipt of the given alphabet symbol. The automata shown analyzes a sequence of coin tosses, with dark states signifying that an even number of heads have been observed. Such automata can be represented using any graph data structure (see Section 12.4 (page 381)), or by an $n \times |\Sigma|$ *transition matrix* where $|\Sigma|$ is the size of the alphabet.

Finite state machines are often used to specify search patterns in the guise of regular expressions, which are patterns formed by and-ing, or-ing, and looping

over smaller regular expressions. For example, the regular expression $a(a+b+c)^*a$ matches any string on (a, b, c) that begins and ends with distinct as. The best way to test whether a string s is recognized by a given regular expression R constructs the finite automaton equivalent to R, and then simulates this machine on S. See Section 18.3 (page 628) for alternative approaches to string matching.

We consider three different problems on finite automata:

- *Minimizing deterministic finite state machines* – Transition matrices for finite automata become prohibitively large for sophisticated machines, thus fueling the need for tighter encodings. The most direct approach is to eliminate redundant states in the automaton. As the example above illustrates, automata of widely varying sizes can compute the same function.

  Algorithms for minimizing the number of states in a deterministic finite automaton (DFA) appear in any book on automata theory. The basic approach partitions the states into gross equivalence classes and then refines the partition. Initially, the states are partitioned into accepting, rejecting, and other classes. The transitions from each node now branch to a given class on a given symbol. Whenever two states s, t from the same class C branch to elements of different classes, the class C must be partitioned into two subclasses, one containing s, the other containing t.

  This algorithm makes a sweep though all the classes looking for a new partition, and repeats the process from scratch if it finds one. This yields an $O(n^2)$ algorithm, since at most $n-1$ sweeps need ever be performed. The final equivalence classes correspond to the states in the minimum automaton. In fact, a more efficient $O(n \log n)$ algorithm is known. Implementations are cited below.

- *Constructing deterministic machines from nondeterministic machines* – DFAs are simple to work with, because the machine is always in exactly one state at any given time. *Nondeterministic automata* (NFAs) can be in multiple states at a time, so their current "state" represents a subset of all possible machine states.

  In fact, any NFA can be mechanically converted to an equivalent DFA, which can then be minimized as above. However, converting an NFA to a DFA might cause an exponential blowup in the number of states, which perversely might then be eliminated when minimizing the DFA. This exponential blowup makes most NFA minimization problems PSPACE-hard, which is even worse than NP-complete.

  The proofs of equivalence between NFAs, DFAs, and regular expressions are elementary enough to be covered in undergraduate automata theory classes. However, they are surprisingly nasty to actually code. Implementations are discussed below.

- *Constructing machines from regular expressions* – There are two approaches for translating a regular expression to an equivalent finite automaton. The difference is whether the output automaton will be a nondeterministic or deterministic machine. NFAs are easier to construct but less efficient to simulate.

  The nondeterministic construction uses ϵ-moves, which are optional transitions that require no input to fire. On reaching a state with an ϵ-move, we must assume that the machine can be in either state. Using ϵ-moves, it is straightforward to construct an automaton from a depth-first traversal of the parse tree of the regular expression. This machine will have $O(m)$ states, if m is the length of the regular expression. Furthermore, simulating this machine on a string of length n takes $O(mn)$ time, since we need consider each state/prefix pair only once.

  The deterministic construction starts with the parse tree for the regular expression, observing that each leaf represents an alphabet symbol in the pattern. After recognizing a prefix of the text, we can be left in some subset of these possible positions, which would correspond to a state in the finite automaton. The *derivatives* method builds up this automaton state by state as it is needed. Even so, some regular expressions of length m require $O(2^m)$ states in any DFA implementing them, such as $(a+b)^*a(a+b)(a+b)\ldots(a+b)$. There is no way to avoid this exponential space blowup. Fortunately it takes linear time to simulate an input string on any DFA, regardless of the size of the automaton.

Implementations: *Grail+* is a C++ package for symbolic computation with finite automata and regular expressions. Grail enables one to convert between different machine representations and to minimize automata. It can handle large machines defined on large alphabets. All code and documentation are accessible from *http://www.csd.uwo.ca/Research/grail*, as well as pointers to a variety of other automaton packages. Commercial use of Grail is not allowed without approval, although it is freely available to students and educators.

The AT&T Finite State Machine Library (FSM) is a set of general-purpose UNIX software tools for building, combining, optimizing, and searching weighted finite-state acceptors and transducers. It supports automata with more than ten million states and transitions. See *http://www.research.att.com/~fsmtools/fsm/*.

JFLAP (Java Formal Languages and Automata Package) is a package of graphical tools for learning the basic concepts of automata theory. Included are functions to convert between DFAs, NFAs, and regular expressions, and minimize the resulting automata. High-level automata are also supported, including context-free languages and Turing machines. *JFLAP* is available at *http://www.jflap.org/*. A related book [RF06] is also available.

FIRE Engine provides production-quality implementations of finite automata and regular expression algorithms. Several finite automaton

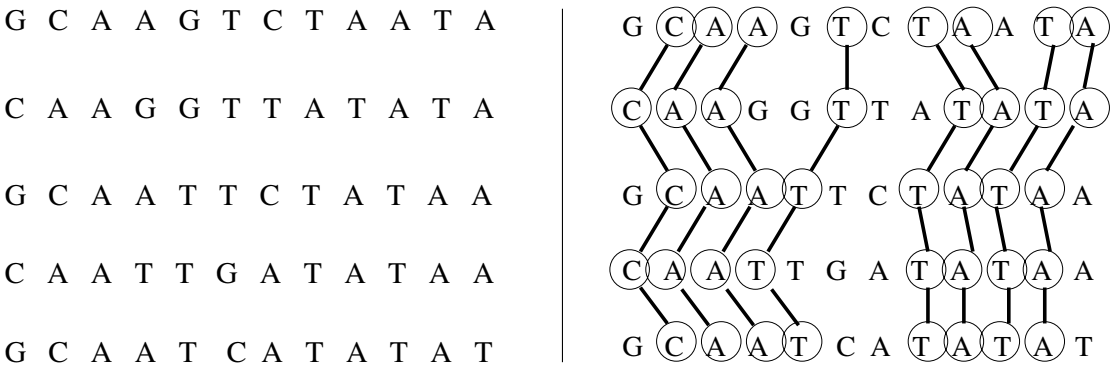
minimization algorithms have been implemented, including Hopcroft's $O(n \lg n)$ algorithm. Both deterministic and nondeterministic automata are supported. It is available at *http://www.fastar.org/* and, with certain enhancements, at *www.eti.pg.gda.pl/∼jandac/minim.html*.

Notes: Aho [Aho90] provides a good survey on algorithms for pattern matching, with a particularly clear exposition for the case where the patterns are regular expressions. The technique for regular expression pattern matching with ϵ-moves is due to Thompson [Tho68]. Other expositions on finite automaton pattern matching include [AHU74]. Expositions on finite automata and the theory of computation include [HMU06, Sip05]

The major annual meeting of interest in this field is the *Conference on Implementations and Applications of Automata* (CIAA). Pointers to current and previous meetings with associated software are available at *http://tln.li.univ-tours.fr/ciaa/*.

Hopcroft [Hop71] gave an optimal $O(n \lg n)$ algorithm for minimizing the number of states in DFAs. The derivatives method of constructing a finite state machine from a regular expression is due to Brzozowski [Brz64] and has been expanded upon in [BS86]. Expositions on the derivatives method includes Conway [Con71]. Recent work on incremental construction and optimization of automata includes [Wat03]. The problems of compressing a DFA to a minimum NFA [JR93] and testing the equivalence of two nondeterministic finite state machines [SM73] are both PSPACE-complete.

Related Problems: Satisfiability (see page 472). string matching (see page 628).



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.

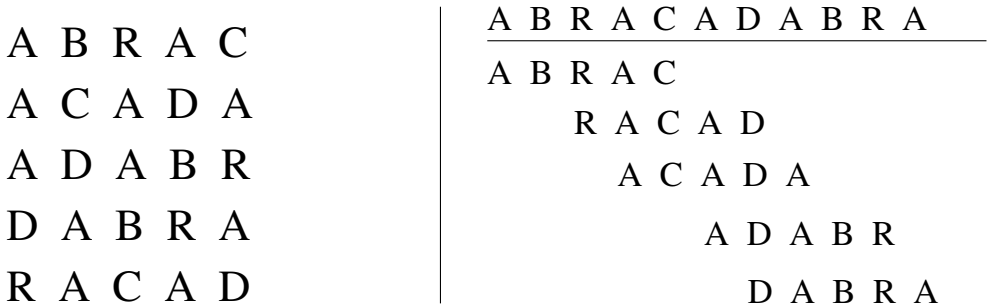
Construct two random n-character strings on an alphabet of size α. What is the expected length of their LCS? This problem has been extensively studied, with an excellent survey by Dancik [Dan94].

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).



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).