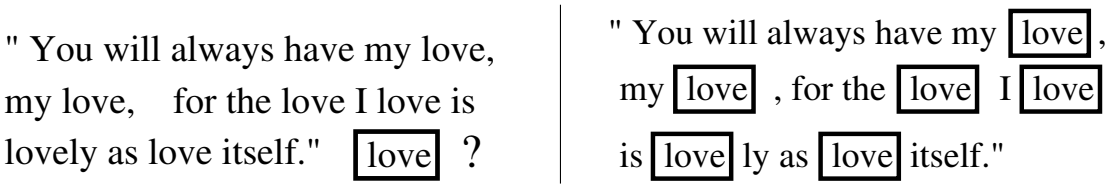

18.3 String Matching

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).