**5.3.1** Develop a brute-force substring search implementation Brute, using the same API as Algorithm 5.6.

**5.3.2** Give the dfa[][] array for the Knuth-Morris-Pratt algorithm for the pattern A A A A A A A A A, and draw the DFA, in the style of the figures in the text

**5.3.3** Give the dfa[][] array for the Knuth-Morris-Pratt algorithm for the pattern A B R A C A D A B R A, and draw the DFA, in the style of the figures in the text.

**5.3.4** Write an efficient method that takes a string txt and an integer M as arguments and returns the position of the first occurrence of M consecutive blanks in the string, txt.length if there is no such occurrence. Estimate the number of character compares used by your method, on typical text and in the worst case

**5.3.5** Develop a brute-force substring search implementation BruteForceRL that processes the pattern from right to left (a simplified version of Algorithm 5.7).

**5.3.6** Give the right[] array computed by the constructor in Algorithm 5.7 for the pattern A B R A C A D A B R A.

**5.3.7** Add to our brute-force implementation of substring search a count() method to count occurrences and a searchAll() method to print all occurrences.

**5.3.8** Add to KMP a count() method to count occurrences and a searchAll() method to print all occurrences

**5.3.9** Add to BoyerMoore a count() method to count occurrences and a searchAll() method to print all occurrences

**5.3.10** Add to RabinKarp a count() method to count occurrences and a searchAll() method to print all occurrences.

**5.3.11** Construct a worst-case example for the Boyer-Moore implementation in Algorithm 5.7 (which demonstrates that it is not linear-time).

**5.3.12** Add the code to check() in RabinKarp (Algorithm 5.8) that turns it into a Las Vegas algorithm (check that the pattern matches the text at the position given as argument).

**5.3.13** In the Boyer-Moore implementation in Algorithm 5.7, show that you can set right[c] to the penultimate occurrence of c when c is the last character in the pattern

**5.3.14** Develop versions of the substring search implementations in this section that use char[] instead of String to represent the pattern and the text

**5.3.15** Design a brute-force substring search algorithm that scans the pattern from right to left

**5.3.16** Show the trace of the brute-force algorithm in the style of the figures in the text for the following pattern and text strings

a. pattern: AAAAAAAB text: AAAAAAAAAAAAAAAAAAAAAAAAB

b. pattern: ABABABAB text: ABABABABAABABABABAAAAAAAA

**5.3.17\*** Draw the KMP DFA for the following pattern strings.

a. AAAAAAB

b. AACAAAB

c. ABABABAB

d. ABAABAAABAAAB

e. ABAABCABAABCB

**5.3.18** Suppose that the pattern and text are random strings over an alphabet of size R (which is at least 2). Show that the expected number of character compares for the brute-force method is

**5.3.19** Construct an example where the Boyer-Moore algorithm (with only the mismatched character heuristic) performs poorly

**5.3.20** How would you modify the Rabin-Karp algorithm to determine whether any of a subset of k patterns (say, all of the same length) is in the text? Solution : Compute the hashes of the k patterns and store the hashes in a StringSET (see Exercise 5.2.6).

**5.3.21** How would you modify the Rabin-Karp algorithm to search for a given pattern with the additional proviso that the middle character is a “wildcard” (any text character at all can match it).

**5.3.22** How would you modify the Rabin-Karp algorithm to search for an H-by-V pattern in an N-by-N text?

**5.3.23** Write a program that reads characters one at a time and reports at each instant if the current string is a palindrome. Hint : Use the Rabin-Karp hashing idea.

**5.3.24** Find all occurrences. Add a method findAll() to each of the four substring search algorithms given in the text that returns an Iterable that allows clients to iterate through all offsets of the pattern in the text.

**5.3.25** Streaming. Add a search() method to KMP that takes variable of type In as argument, and searches for the pattern in the specified input stream without using any extra instance variables. Then do the same for RabinKarp.

**5.3.26** Cyclic rotation check. Write a program that, given two strings, determines whether one is a cyclic rotation of the other, such as example and ampleex.

**5.3.27** Tandem repeat search. A tandem repeat of a base string b in a string s is a substring of s having at least two consecutive copies b (nonoverlapping). Develop and implement a linear-time algorithm that, given two strings b and s, returns the index of the beginning of the longest tandem repeat of b in s. For example, your program should return 3 when b is abcab and s is abcabcababcababcababcab.

**5.3.28** Buffering in brute-force search. Add a search() method to your solution to Exercise 5.3.1 that takes an input stream (of type In) as argument and searches for the pattern in the given input stream. Note : You need to maintain a buffer that can keep at least the previous M characters in the input stream. Your challenge is to write efficient code to initialize, update, and clear the buffer for any input stream.

**5.3.29** Buffering in Boyer-Moore. Add a search() method to Algorithm 5.7 that takes an input stream (of type In) as argument and searches for the pattern in the given input stream.

**5.3.30** Two-dimensional search. Implement a version of the Rabin-Karp algorithm to search for patterns in two-dimensional text. Assume both pattern and text are rectangles of characters

**5.3.31** Random patterns. How many character compares are needed to do a substring search for a random pattern of length 100 in a given text

**5.3.32** Unique substrings. Solve Exercise 5.2.14 using the idea behind the Rabin-Karp method.

**5.3.33** Random primes. Implement longRandomPrime() for RabinKarp (Algorithm 5.8). Hint : A random n-digit number is prime with probability proportional to 1/n

**5.3.34** Straight-line code. The Java Virtual Machine (and your computer’s assembly language) support a goto instruction so that the search can be “wired in’’ to machine code, like the program at right (which is exactly equivalent to simulating the DFA for the pattern as in KMPdfa, but likely to be much more efficient). To avoid checking whether the end of the text has been reached each time i is incremented, we assume that the pattern itself is stored at the end of the text as a sentinel, as the last M characters of the text. The goto labels in this code correspond precisely to the dfa[] array. Write a static method that takes a pattern as input and produces as output a straight-line program like this that searches for the pattern

**5.3.35** Boyer-Moore in binary strings. The mismatched character heuristic does not help much for binary strings, because there are only two possibilities for characters that cause the mismatch (and these are both likely to be in the pattern). Develop a substring search class for binary strings that groups bits together to make “characters’’ that can be used exactly as in Algorithm 5.7. Note : If you take b bits at a time, then you need a right[] array with 2b entries. The value of b should be chosen small enough so that this table is not too large, but large enough that most b-bit sections of the text are not likely to be in the pattern—there are M-b+1 different b-bit sections in the pattern (one starting at each bit position from 1 through M-b+1), so we want M-b+1 to be significantly less than 2b. For example, if you take 2b to be about lg (4M), then the right[] array will be more than three-quarters filled with -1 entries, but do not let b become less than M/2, since otherwise you could miss the pattern entirely, if it were split between two b-bit text sections.

**5.3.36** Random text. Write a program that takes integers M and N as arguments, generates a random binary text string of length N, then counts the number of other occurrences of the last M bits elsewhere in the string. Note : Different methods may be appropriate for different values of M.

**5.3.37** KMP for random text. Write a client that takes integers M, N, and T as input and runs the following experiment T times: Generate a random pattern of length M and a random text of length N, counting the number of character compares used by KMP to search for the pattern in the text. Instrument KMP to provide the number of compares, and print the average count for the T trials.

**5.3.38** Boyer-Moore for random text. Answer the previous exercise for BoyerMoore.

**5.3.39** Timings. Write a program that times the four methods for the task of searchng for the substring it is a far far better thing that i do than i have ever done in the text of Tale of Two Cities (tale.txt). Discuss the extent to which your results validate the hypthotheses about performance that are stated in the text.