5.3 SUBSTRING SEARCH

A FUNDAMENTAL OPERATION on strings is *substring search*: given a *text* string of length N and a *pattern* string of length M, find an occurrence of the pattern within the text. Most algorithms for this problem can easily be extended to find all occurrences of the pattern in the text, to count the number of occurrences of the pattern in the text, or to provide context (substrings of the text surrounding each occurrence of the pattern).

When you search for a word while using a text editor or a web browser, you are doing substring search. Indeed, the original motivation for this problem was to support such searches. Another classic application is searching for some important pattern in an intercepted communication. A military leader might be interested in finding the pattern `ATTACK AT DAWN` somewhere in an intercepted text message; a hacker might be interested in finding the pattern `Password:` somewhere in your computer's memory. In today's world, we are often searching through the vast amount of information available on the web.

To best appreciate the algorithms, think of the pattern as being relatively short (with M equal to, say, 100 or 1,000) and the text as being relatively long (with N equal to, say, 1 million or 1 billion). In substring search, we typically preprocess the pattern in order to be able to support fast searches for that pattern in the text.

Substring search is an interesting and classic problem: several very different (and surprising) algorithms have been discovered that not only provide a spectrum of useful practical methods but also illustrate a spectrum of fundamental algorithm design techniques.

Substring search

A short history The algorithms that we examine have an interesting history; we summarize it here to help place the various methods io perspective.

There is a simple brute-force algorithm for substring search that is in widespread use. While it has a worst-case running time proportional to MN, the strings that arise in many applications lead to a running time that is (except in pathological cases) proportional to $M + N$. Furthermore, it is well-suited to standard architectural features on most computer systems, so an optimized version provides a standard benchmark that is difficult to beat, even with a clever algorithm.

In 1970, S. Cook proved a theoretical result about a particular type of abstract machine that implied the existence of an algorithm that solves the substring search problem in time proportional to $M + N$ in the worst case. D. E. Knuth and V. R. Pratt laboriously followed through the construction Cook used to prove his theorem (which was not intended to be practical) and refined it into a relatively simple and practical algorithm. This seemed a rare and satisfying example of a theoretical result with immediate (and unexpected) practical applicability. But it turned out that J. H. Morris had discovered virtually the same algorithm as a solution to an annoying problem confronting him when implementing a text editor (he wanted to avoid having to "back up" in the text string). The fact that the same algorithm arose from two such different approaches lends it credibility as a fundamental solution to the problem.

Knuth, Morris, and Pratt didn't get around to publishing their algorithm until 1976, and in the meantime R. S. Boyer and J. S. Moore (and, independently, R. W. Gosper) discovered an algorithm that is much faster in many applications, since it often examines only a fraction of the characters in the text string. Many text editors use this algorithm to achieve a noticeable decrease in response time for substring search.

Both the Knuth-Morris-Pratt (KMP) and the Boyer-Moore algorithms require some complicated preprocessing on the pattern that is difficult to understand and has limited the extent to which they are used. (In fact, the story goes that an unknown systems programmer found Morris's algorithm too difficult to understand and replaced it with a brute-force implementation.)

In 1980, M. O. Rabin and R. M. Karp used hashing to develop an algorithm almost as simple as the brute-force algorithm that runs in time proportional to $M + N$ with very high probability. Furthermore, their algorithm extends to two-dimensional patterns and text, which makes it more useful than the others for image processing.

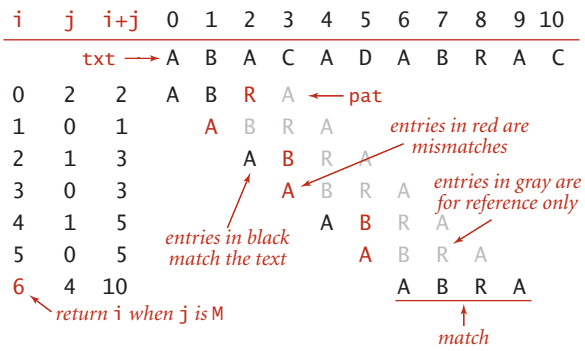
This story illustrates that the search for a better algorithm is still very often justified; indeed, one suspects that there are still more developments on the horizon even for this classic problem.

Brute-force substring search An obvious method for substring search is to check, for each possible position in the text at which the pattern could match, whether it does in fact match. The `search()` method below operates in this way to find the first occurrence of a pattern string `pat` in a text string `txt`. The program keeps one pointer (`i`) into the text and another pointer (`j`) into the pattern. For each `i`, it resets `j` to 0 and increments it until finding a mismatch or the end of the pattern (`j == M`). If we reach the end of the text (`i == N-M+1`) before the end of the pattern, then there is no match: the pattern does not occur in the text. Our convention is to return the value `N` to indicate a mismatch.

```
public static int search(String pat, String txt)
{
   int M = pat.length();
   int N = txt.length();
   for (int i = 0; i <= N - M; i++)
   {
      int j;
      for (j = 0; j < M; j++)
         if (txt.charAt(i+j) != pat.charAt(j))
            break;
      if (j == M) return i;  // found
   }
   return N;                 // not found
}
```

Brute-force substring search

In a typical text-processing application, the `j` index rarely increments so the running time is proportional to N. Nearly all of the compares find a mismatch with the first character of the pattern. For example, suppose that you search for the pattern `pattern` in the text of this paragraph. There are 191 characters up to the end of the first occurrence of the pattern, only 7 of which are the character `p` (and there are no occurrences of `pa`), so the total number of character compares is 191+7, for an average of 1.036 compares per character in the text. On the other hand, there is no guarantee that the algorithm will always be so efficient. For example, a pattern might begin with a long string of `A`s. If it does, and the text also has long strings of `A`s, then the substring search will be slow.



Brute-force substring search

Proposition M. Brute-force substring search requires ~NM character compares to search for a pattern of length M in a text of length N, in the worst case.

Proof: A worst-case input is when both pattern and text are all As followed by a B. Then for each of the $N-M+1$ possible match positions, all the characters in the pattern are checked against the text, for a total cost of $M(N-M+1)$. Normally M is very small compared to N, so the total is ~NM.

Such degenerate strings are not likely to appear in English text, but they may well occur in other applications (for example, in binary texts), so we seek better algorithms.

```
i   j  i+j    0 1 2 3 4 5 6 7 8 9
        txt→   A A A A A A A A A B
0   4   4      A A A A B ← pat
1   4   5        A A A A B
2   4   6          A A A A B
3   4   7            A A A A B
4   4   8              A A A A B
5   5  10                A A A A B
```

Brute-force substring search (worst case)

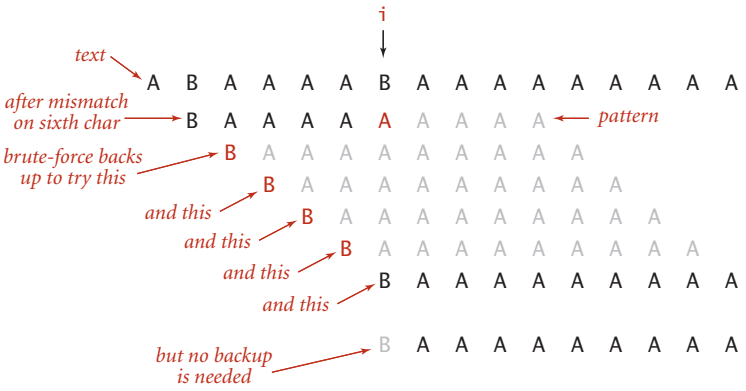
The alternate implementation at the bottom of this page is instructive. As before, the program keeps one pointer (`i`) into the text and another pointer (`j`) into the pattern. As long as they point to matching characters, both pointers are incremented. This code performs precisely the same character compares as the previous implementation. To understand it, note that `i` in this code maintains the value of `i+j` in the previous code: it points to the *end* of the sequence of already-matched characters in the text (where `i` pointed to the *beginning* of the sequence before). If `i` and `j` point to mismatching characters, then we *back up* both pointers: `j` to point to the beginning of the pattern and `i` to correspond to moving the pattern to the right one position for matching against the text.

```
public static int search(String pat, String txt)
{
   int j, M = pat.length();
   int i, N = txt.length();
   for (i = 0, j = 0; i < N && j < M; i++)
   {
      if (txt.charAt(i) == pat.charAt(j)) j++;
      else { i -= j; j = 0;  }
   }
   if (j == M) return i - M;  // found
   else        return N;  // not found
}
```

Alternate implementation of brute-force substring search (explicit backup)

Knuth-Morris-Pratt substring search The basic idea behind the algorithm discovered by Knuth, Morris, and Pratt is this: whenever we detect a mismatch, we already know some of the characters in the text (since they matched the pattern characters prior to the mismatch). We can take advantage of this information to avoid backing up the text pointer over all those known characters.

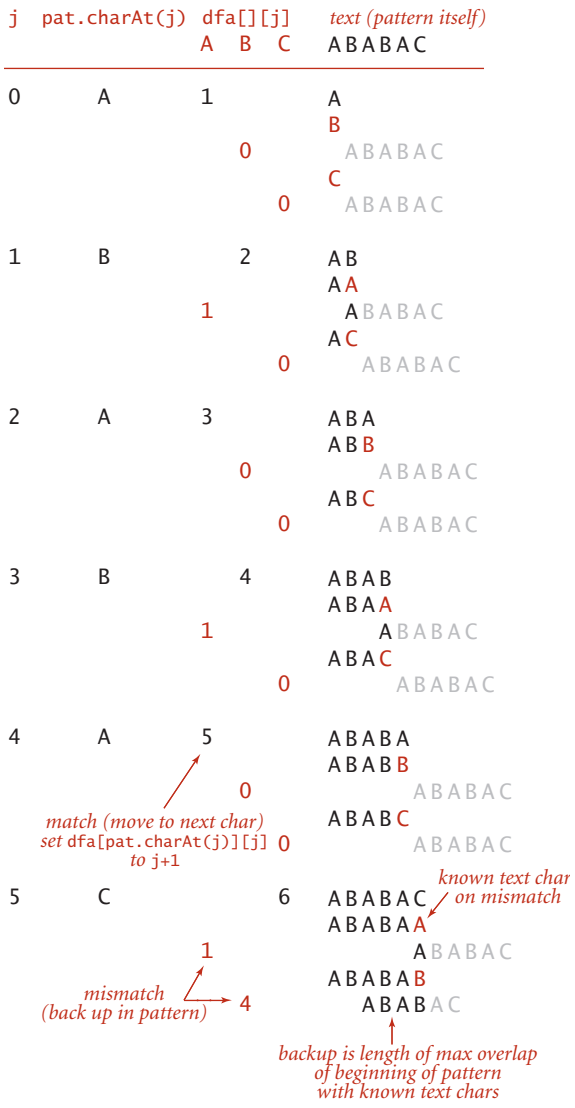
As a specific example, suppose that we have a two-character alphabet and are searching for the pattern B A A A A A A A A A. Now, suppose that we match five characters in the pattern, with a mismatch on the sixth. When the mismatch is detected, we know that the six previous characters in the text must be B A A A A B (the first five match and the sixth does not), with the text pointer now pointing at the B at the end. The key observation is that we need not back up the text pointer i, since the previous four characters in the text are all As and do not match the first character in the pattern. Furthermore, the character currently pointed to by i is a B and does match the first character in the pattern, so we can increment i and compare the next character in the text with the second character in the pattern. This argument leads to the observation that, for this pattern, we can change the else clause in the alternate brute-force implementation to just set j = 1 (and not decrement i). Since the value of i does not change within the loop, this method does at most N character compares. The practical effect of this particular change is limited to this particular pattern, but the idea is worth thinking about—the Knuth-Morris-Pratt algorithm is a generalization of it. Surprisingly, it is *always* possible to find a value to set the j pointer to on a mismatch, so that the i pointer is never decremented.



Text pointer backup in substring searching

Fully skipping past all the matched characters when detecting a mismatch will not work when the pattern could match itself at any position overlapping the point of the mismatch. For example, when searching for the pattern A A B A A A in the text A A B A A B A A A A, we first detect the mismatch at position 5, but we had better restart at position 3 to continue the search, since otherwise we would miss the match. The insight of the KMP algorithm is that we can decide ahead of time exactly how to restart the search, because that decision depends only on the pattern.

Backing up the pattern pointer. In KMP substring search, we never back up the text pointer `i`, and we use an array `dfa[][]` to record how far to back up the pattern pointer `j` when a mismatch is detected. For every character `c`, `dfa[c][j]` is the pattern position to compare against the next text position after comparing `c` with `pat.charAt(j)`. During the search, `dfa[txt.charAt(i)][j]` is the pattern position to compare with `txt.charAt(i+1)` after comparing `txt.charAt(i)` with `pat.charAt(j)`. For a match, we want to just move on to the next character, so `dfa[pat.charAt(j)][j]` is always `j+1`. For a mismatch, we know not just `txt.charAt(i)`, but also the `j-1` previous characters in the text: *they are the first `j-1` characters in the pattern*. For each character `c`, imagine that we slide a copy of the pattern over these `j` characters (the first `j-1` characters in the pattern followed by `c`—we are deciding what to do when these characters are `txt.charAt(i-j+1..i)`), from left to right, stopping when all overlapping characters match (or there are none). This gives the next possible place the pattern could match. The index of the pattern character to compare with `txt.charAt(i+1)` (`dfa[txt.charAt(i)][j]`) is precisely the number of overlapping characters.



Pattern backup for A B A B A C in KMP substring search

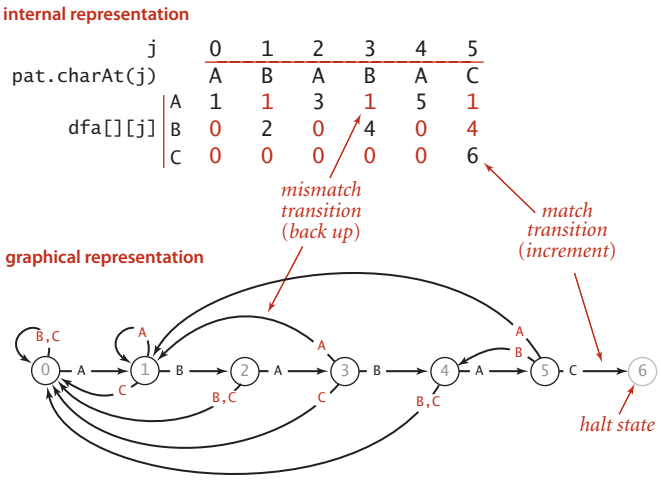
KMP search method. Once we have computed the `dfa[][]` array, we have the substring search method at the top of the next page: when `i` and `j` point to mismatching characters (testing for a pattern match beginning at position `i-j+1` in the text string), then the next possible position for a pattern match is beginning at position `i-dfa[txt.charAt(i)][j]`. But by construction, the first `dfa[txt.charAt(i)][j]` characters at that position match the first `dfa[txt.charAt(i)][j]` characters of the pattern, so there is no need to back up the `i` pointer: we can simply set `j` to `dfa[txt.charAt(i)][j]` and increment `i`, which is precisely what we do when `i` and `j` point to matching characters.

DFA simulation. A useful way to describe this process is in terms of a *deterministic finite-state automaton* (DFA). Indeed, as indicated by its name, our `dfa[][]` array precisely defines a DFA. The graphical DFA represention shown at the bottom of this page consists of states (indicated by circled numbers) and transitions (indicated by labeled lines). There is one state for each character in the pattern, each such state having one transition leaving it for each character in the alphabet. For the substring-matching DFAs that we are considering, one of the transitions is a *match* transition (going from `j` to `j+1` and labeled with `pat.charAt(j)`) and all the others are *mismatch* transition (going left). The states correspond to character compares, one for each value of the pattern index. The transitions correspond to changing the value of the pattern index. When examining the text character `i` when in the state labeled `j`, the machine does the following: "Take the transition to `dfa[txt.charAt(i)][j]` and move to the next character (by incrementing `i`)." For a match transition, we move to the right one position because `dfa[pat.charAt(j)][j]` is always `j+1`; for a mismatch transition we move to the left. The automaton reads the text characters one at a time, from left to right, moving to a new state each time it reads a character. We also include a *halt* state `M` that has no transitions. We start the machine at state 0: if the machine reaches state `M`, then a substring of the text matching the pattern has been found (and we say that the DFA *recognizes* the pattern); if the machine reaches the end of the text before reaching state `M`, then we know the pattern does not appear as a substring of the text. Each pattern corresponds to an automaton (which is represented by the `dfa[][]` array that gives the transitions). The KMP substring `search()` method is a Java program that simulates the operation of such an automaton.

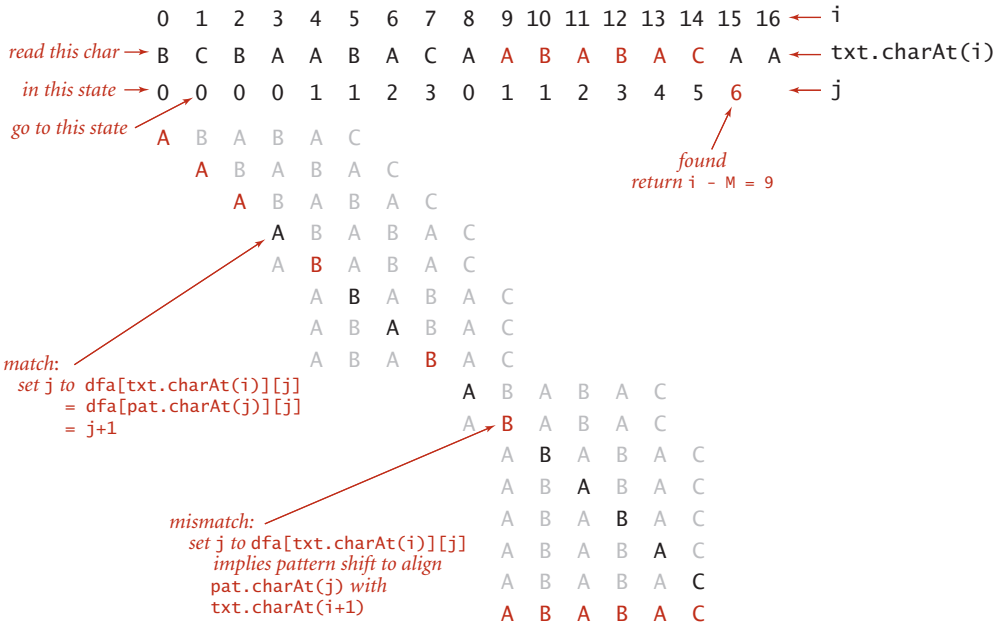
```
public int search(String txt)
{  // Simulate operation of DFA on txt.
   int i, j, N = txt.length();
   for (i = 0, j = 0; i < N && j < M; i++)
      j = dfa[txt.charAt(i)][j];
   if (j == M) return i - M;  // found
   else        return N;      // not found
}
```

KMP substring search (DFA simulation)

To get a feeling for the operation of a substring search DFA, consider two of the simplest things that it does. At



DFA corresponding to the string A B A B A C

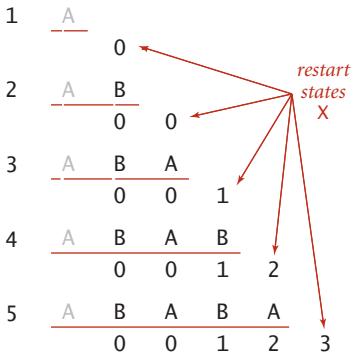


Trace of KMP substring search (DFA simulation) for A B A B A C

the beginning of the process, when started in state 0 at the beginning of the text, it stays in state 0, scanning text characters, until it finds a text character that is equal to the first pattern character, when it moves to the next state and is off and running. At the end of the process, when it finds a match, it matches pattern characters with the right end of the text, incrementing the state until reaching state `M`. The trace at the top of this page gives a typical example of the operation of our example DFA. Each match moves the DFA to the next state (which is equivalent to incrementing the pattern index `j`); each mismatch moves the DFA to an earlier state (which is equivalent to setting the pattern index `j` to a smaller value). The text index `i` marches from left to right, one position at a time, while the pattern index `j` bounces around in the pattern as directed by the DFA.

Constructing the DFA. Now that you understand the mechanism, we are ready to address the key question for the KMP algorithm: How do we compute the `dfa[][]` array corresponding to a given pattern? Remarkably, the answer to this question lies in the DFA *itself* (!) using the ingenious (and rather tricky) construction that was developed by Knuth, Morris, and Pratt. When we have a mismatch at `pat.charAt(j)`, our interest is in knowing in what state the DFA *would be* if we were to back up the text index and rescan the text characters that we just saw after shifting to the right one position. We do not want to actually do the backup, just restart the DFA *as if* we had done the backup.

The key observation is that the characters in the text that would need to be rescanned are precisely `pat.charAt(1)` through `pat.charAt(j-1)`: we drop the first character to shift right one position and the last character because of the mismatch. These are pattern characters that we know, so we can figure out ahead of time, for each possible mismatch position, the state where we need to restart the DFA. The figure at left shows the possibilities for our example. *Be sure that you understand this concept.*



DFA simulations to compute restart states for A B A B A C

What should the DFA do with the next character? Exactly what it would have done if we had backed up, *except* if it finds a match with `pat.charAt(j)`, when it should go to state `j+1`. For example, to decide what the DFA should do when we have a mismatch at `j = 5` for `A B A B A C`, we use the DFA to learn that a full backup would leave us in state 3 for `B A B A`, so we can copy `dfa[][3]` to `dfa[][5]`, then set the entry for `C` to 6 because `pat.charAt(5)` is `C` (a match). Since we only need to know how the DFA runs for `j-1` characters when we are building the `j`th state, we can always get the information that we need from the partially built DFA.

The final crucial detail to the computation is to observe that maintaining the restart position `X` when working on column `j` of `dfa[][]` is easy because `X < j` so that we can use the partially built DFA to do the job—the next value of `X` is `dfa[pat.charAt(j)][X]`). Continuing our example from the previous paragraph, we would update the value of `X` to `dfa['C'][3] = 0` (but we do not use that value because the DFA construction is complete).

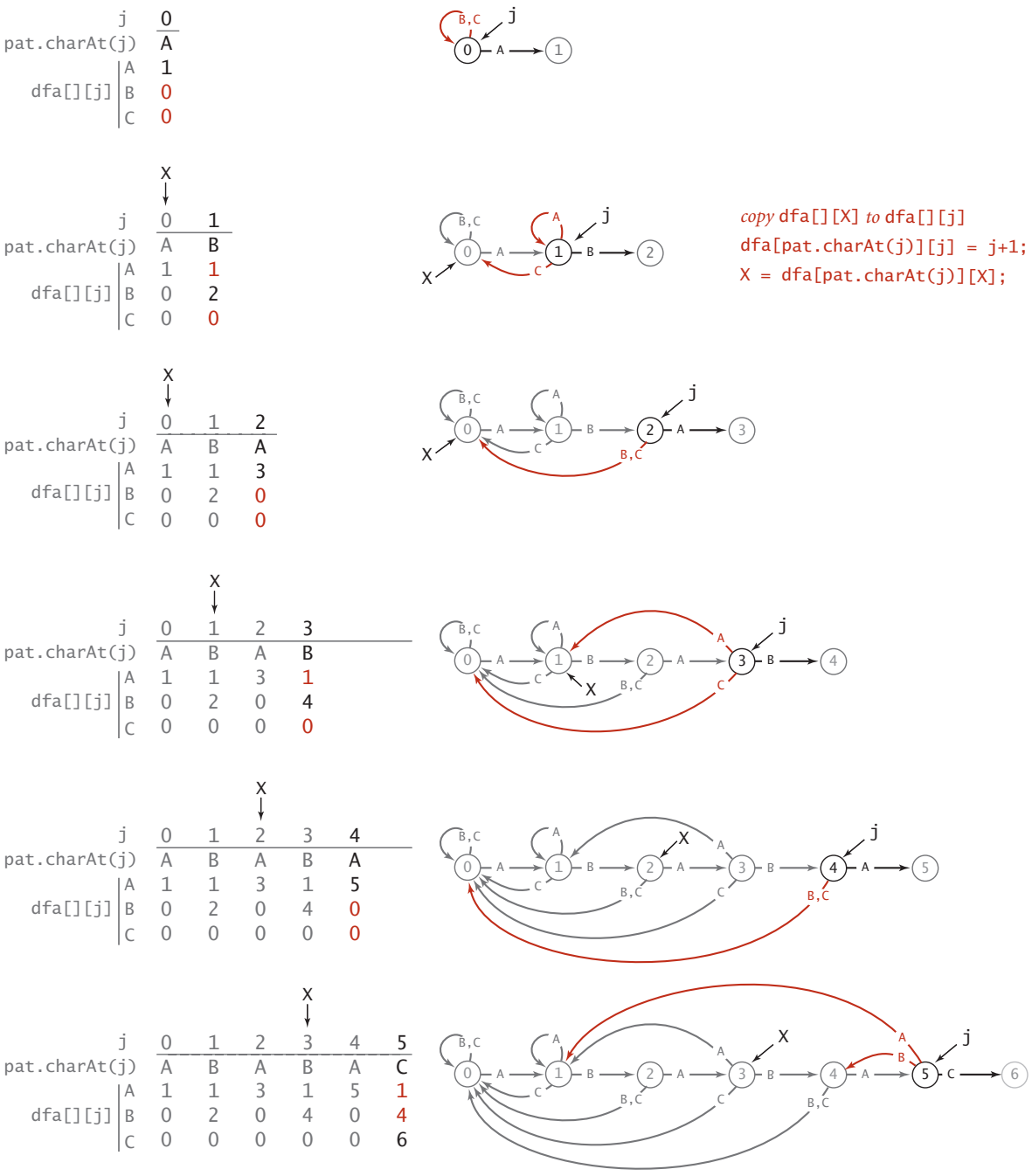
The discussion above leads to the remarkably compact code below for constructing the DFA corresponding to a given pattern. For each `j`, it

- Copies `dfa[][X]` to `dfa[][j]` (for mismatch cases)
- Sets `dfa[pat.charAt(j)][j]` to `j+1` (for the match case)
- Updates `X`

The diagram on the facing page traces this code for our example. To make sure that you understand it, work EXERCISE 5.3.2 and EXERCISE 5.3.3.

```
dfa[pat.charAt(0)][0] = 1;
for (int X = 0, j = 1; j < M; j++)
{  // Compute dfa[][j].
   for (int c = 0; c < R; c++)
      dfa[c][j] = dfa[c][X];
   dfa[pat.charAt(j)][j] = j+1;

   X = dfa[pat.charAt(j)][X];
}
```

Constructing the DFA for KMP substring search



copy `dfa[][X]` *to* `dfa[][j]`

```
dfa[pat.charAt(j)][j] = j+1;
X = dfa[pat.charAt(j)][X];
```

Constructing the DFA for KMP substring search for A B A B A C

ALGORITHM 5.6 Knuth-Morris-Pratt substring search

```
public class KMP
{
   private String pat;
   private int[][] dfa;

   public KMP(String pat)
   {  // Build DFA from pattern.
      this.pat = pat;
      int M = pat.length();
      int R = 256;
      dfa = new int[R][M];
      dfa[pat.charAt(0)][0] = 1;
      for (int X = 0, j = 1; j < M; j++)
      {  // Compute dfa[][j].
         for (int c = 0; c < R; c++)
            dfa[c][j] = dfa[c][X];            // Copy mismatch cases.
         dfa[pat.charAt(j)][j] = j+1;         // Set match case.
         X = dfa[pat.charAt(j)][X];           // Update restart state.
      }
   }

   public int search(String txt)
   {  // Simulate operation of DFA on txt.
      int i, j, N = txt.length(), M = pat.length();
      for (i = 0, j = 0; i < N && j < M; i++)
         j = dfa[txt.charAt(i)][j];
      if (j == M) return i - M;  // found (hit end of pattern)
      else        return N;      // not found (hit end of text)
   }

   public static void main(String[] args)
   // See page 769.
}
```

The constructor in this implementation of the Knuth-Morris-Pratt algorithm for substring search builds a DFA from a pattern string, to support a `search()` method that can find the pattern in a given text string. This program does the same job as the brute-force method, but it runs faster for patterns that are self-repetitive.

```
% java KMP AACAA AABRAACADABRAACAADABRA
text:    AABRAACADABRAACAADABRA
pattern:             AACAA
```

ALGORITHM 5.6 on the facing page implements the following API:

public class KMP	
KMP(String pat)	*create a DFA that can search for* pat
int search(String txt)	*find index of* pat *in* txt

Substring search API

You can see a typical test client at the bottom of this page. The constructor builds a DFA from a pattern that the `search()` method uses to search for the pattern in a given text.

Proposition N. Knuth-Morris-Pratt substring search accesses no more than $M + N$ characters to search for a pattern of length M in a text of length N.

Proof. Immediate from the code: we access each pattern character once when computing `dfa[][]` and each text character once (in the worst case) in `search()`.

Another parameter comes into play: for an R-character alphabet, the total running time (and space) required to build the DFA is proportional to MR. It is possible to remove the factor of R by building a DFA where each state has a match transition and a mismatch transition (not transitions for each possible character), though the construction is somewhat more intricate.

The linear-time worst-case guarantee provided by the KMP algorithm is a significant theoretical result. In practice, the speedup over the brute-force method is not often important because few applications involve searching for highly self-repetitive patterns in highly self-repetitive text. Still, the method has the practical advantage that it never backs up in the input. This property makes KMP substring search more convenient for use on an input stream of undetermined length (such as standard input) than algorithms requiring backup, which need some complicated buffering in this situation. Ironically, when backup is easy, we can do significantly better than KMP. Next, we consider a method that generally leads to substantial performance gains precisely because it *can* back up in the text.

```
public static void main(String[] args)
{
   String pat = args[0];
   String txt = args[1];
   KMP kmp = new KMP(pat);
   StdOut.println("text:    " + txt);
   int offset = kmp.search(txt);
   StdOut.print("pattern: ");
   for (int i = 0; i < offset; i++)
      StdOut.print(" ");
   StdOut.println(pat);

}
```

KMP substring search test client

Boyer-Moore substring search When backup in the text string is not a problem, we can develop a significantly faster substring-searching method by scanning the pattern from *right to left* when trying to match it against the text. For example, when searching for the substring `BAABBAA`, if we find matches on the seventh and sixth characters but not on the fifth, then we can immediately slide the pattern seven positions to the right, and check the 14th character in the text next, because our partial match found `XAA` where `X` is not `B`, which does not appear elsewhere in the pattern. In general, the pattern at the end might appear elsewhere, so we need an array of restart positions as for Knuth-Morris-Pratt. We will not explore this approach in further detail because it is quite similar to our implementation of the Knuth-Morris-Pratt method. Instead, we will consider another suggestion by Boyer and Moore that is typically even more effective in right-to-left pattern scanning.

As with our implementation of KMP substring search, we decide what to do next on the basis of the character that caused the mismatch in the *text* as well as the pattern. The preprocessing step is to decide, for each possible character that could occur in the text, what we would do if that character were to cause the mismatch. The simplest realization of this idea leads immediately to an efficient and useful substring search method.

Mismatched character heuristic. Consider the figure at the bottom of this page, which shows a search for the pattern `NEEDLE` in the text `FINDINAHAYSTACKNEEDLE`. Proceeding from right to left to match the pattern, we first compare the rightmost `E` in the pattern with the `N` (the character at position 5) in the text. Since `N` appears in the pattern, we slide the pattern five positions to the right to line up the `N` in the text with the (rightmost) `N` in the pattern. Then we compare the rightmost `E` in the pattern with the `S` (the character at position 10) in the text. This is also a mismatch, but `S` *does not* appear in the pattern, so we can slide the pattern six positions to the right.We match the rightmost `E` in the pattern against the `E` at position 16 in the text, then find a mismatch and discover the `N` at position 15 and slide to the right five positions, as at the beginning. Finally, we verify, moving from right to left starting at position 20, that the pattern is in the text. This method brings us to the match position at a cost of only four character compares (and six more to verify the match)!

Mismatched character heuristic for right-to-left (Boyer-Moore) substring search

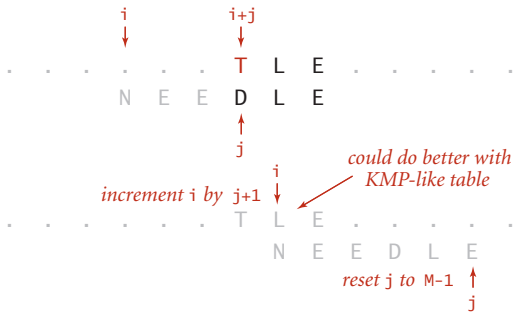
Starting point. To implement the mismatched character heuristic, we use an array `right[]` that gives, for each character in the alphabet, the index of its *rightmost occurrence* in the pattern (or `-1` if the character is not in the pattern). This value tells us precisely how far to skip if that character appears in the text and causes a mismatch during the string search. To initialize the `right[]` array, we set all entries to `-1` and then, for `j` from `0` to `M-1`, set `right[pat.charAt(j)]` to `j`, as shown in the example at right for our example pattern `NEEDLE`.

		N	E	E	D	L	E	
c		0	1	2	3	4	5	right[c]
A	-1	-1	-1	-1	-1	-1	-1	-1
B	-1	-1	-1	-1	-1	-1	-1	-1
C	-1	-1	-1	-1	-1	-1	-1	-1
D	-1	-1	-1	-1	3	3	3	3
E	-1	-1	1	2	2	2	5	5
...								-1
L	-1	-1	-1	-1	-1	4	4	4
M	-1	-1	-1	-1	-1	-1	-1	-1
N	-1	0	0	0	0	0	0	0
...								-1

Boyer-Moore skip table computation

Substring search. With the `right[]` array precomputed, the implementation in ALGORITHM 5.7 is straightforward. We have an index `i` moving from left to right through the text and an index `j` moving from right to left through the pattern. The inner loop tests whether the pattern aligns with the text at position `i`. If `txt.charAt(i+j)` is equal to `pat.charAt(j)` for all `j` from `M-1` down to `0`, then there is a match. Otherwise, there is a character mismatch, and we have one of the following three cases:

- If the character causing the mismatch is not found in the pattern, we can slide the pattern `j+1` positions to the right (increment `i` by `j+1`). Anything less would align that character with some pattern character. Actually, this move aligns some known characters at the beginning of the pattern with known characters at the end of the pattern so that we could further increase `i` by precomputing a KMP-like table (see example at right).



Mismatched character heuristic (mismatch not in pattern)

- If the character `c` causing the mismatch is found in the pattern, we use the `right[]` array to line up the pattern with the text so that character will match its rightmost occurrence in the pattern. To do so, we increment `i` by `j` minus `right[c]`. Again, anything less would align that text character with a pattern character it could not match (one to the right of its rightmost occurrence). Again, there is a possibility that we could do better with a KMP-like table, as indicated in the top example in the figure on page 773.

ALGORITHM 5.7 Boyer-Moore substring search (mismatched character heuristic)

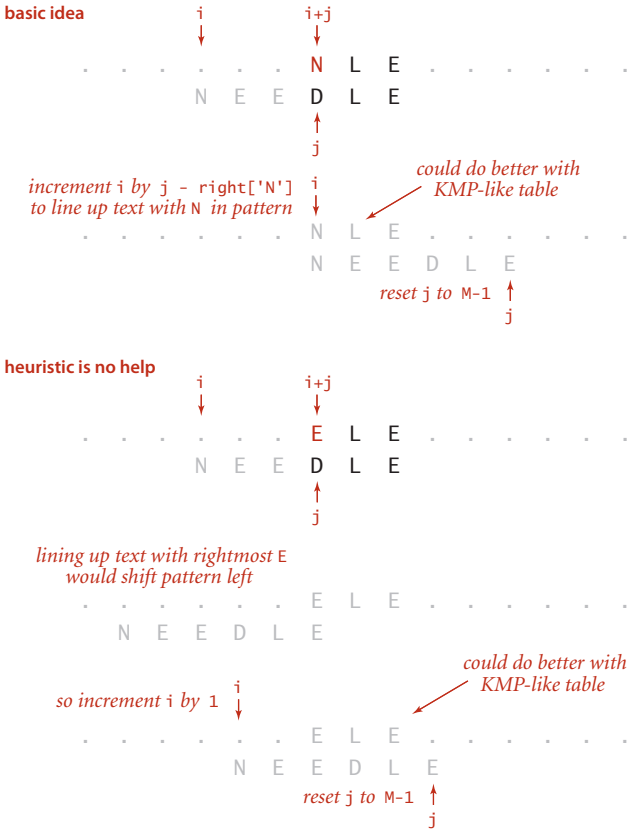
```
public class BoyerMoore
{
   private int[] right;
   private String pat;

   BoyerMoore(String pat)
   {  // Compute skip table.
      this.pat = pat;
      int M = pat.length();
      int R = 256;
      right = new int[R];
      for (int c = 0; c < R; c++)
         right[c] = -1;                // -1 for chars not in pattern
      for (int j = 0; j < M; j++)      // rightmost position for
         right[pat.charAt(j)] = j;     //   chars in pattern
   }

   public int search(String txt)
   {  // Search for pattern in txt.
      int N = txt.length();
      int M = pat.length();
      int skip;
      for (int i = 0; i <= N-M; i += skip)
      {  // Does the pattern match the text at position i ?
         skip = 0;
         for (int j = M-1; j >= 0; j--)
            if (pat.charAt(j) != txt.charAt(i+j))
            {
               skip = j - right[txt.charAt(i+j)];
               if (skip < 1) skip = 1;
               break;
            }
         if (skip == 0) return i;          // found.
      }
      return N;                            // not found.
    }

   public static void main(String[] args)  // See page 769.
}
```

The constructor in this substring search algorithm builds a table giving the rightmost occurrence in the pattern of each possible character. The search method scans from right to left in the pattern, skipping to align any character causing a mismatch with its rightmost occurrence in the pattern.

- If this computation would not increase `i`, we just increment `i` instead, to make sure that the pattern always slides at least one position to the right. The bottom example in the figure at right illustrates this situation.

ALGORITHM 5.7 is a straightforward implementation of this process. Note that the convention of using `-1` in the `right[]` array entries corresponding to characters that do not appear in the pattern unifies the first two cases (increment `i` by `j - right[txt.charAt(i+j)]`).

The full Boyer-Moore algorithm takes into account precomputed mismatches of the pattern with itself (in a manner similar to the KMP algorithm) and provides a linear-time worst-case guarantee (whereas ALGORITHM 5.7 can take time proportional to NM in the worst case—see EXERCISE 5.3.19). We omit this computation because the mismatched character heuristic controls the performance in typical practical applications.



Mismatched character heuristic (mismatch in pattern)

Property O. On typical inputs, substring search with the Boyer-Moore mismatched character heuristic uses ~N/M character compares to search for a pattern of length M in a text of length N.

Discussion: This result can be proved for various random string models, but such models tend to be unrealistic, so we shall skip the details. In many practical situations it is true that all but a few of the alphabet characters appear nowhere in the pattern, so nearly all compares lead to M characters being skipped, which gives the stated result.

Rabin-Karp fingerprint search The method developed by M. O. Rabin and R. A. Karp is a completely different approach to substring search that is based on hashing. We compute a hash function for the pattern and then look for a match by using the same hash function for each possible M-character substring of the text. If we find a text substring with the same hash value as the pattern, we can check for a match. This process is equivalent to storing the pattern in a hash table, then doing a search for each substring of the text, but we do not need to reserve the memory for the hash table because it would have just one entry. A straightforward implementation based on this description would be much slower than a brute-force search (since computing a hash function that involves every character is likely to be much more expensive than just comparing characters), but Rabin and Karp showed that it is easy to compute hash functions for M-character substrings in *constant* time (after some preprocessing), which leads to a *linear*-time substring search in practical situations.

Basic plan. A string of length `M` corresponds to an `M`-digit base-`R` number. To use a hash table of size `Q` for keys of this type, we need a hash function to convert an `M`-digit base-`R` number to an `int` value between `0` and `Q-1`. Modular hashing (see SECTION 3.4) provides an answer: take the remainder when dividing the number by `Q`. In practice, we use a random prime `Q`, taking as large a value as possible while avoiding overflow (because we do not actually need to store a hash table). The method is simplest to understand for small `Q` and `R = 10`, shown in the example below. To find the pattern `2 6 5 3 5` in the text `3 1 4 1 5 9 2 6 5 3 5 8 9 7 9 3`, we choose a table size `Q` (`997` in the example), compute the hash value `26535 % 997 = 613`, and then look for a match by computing hash values for each five-digit substring in the text. In the example, we get the hash values `508`, `201`, `715`, `971`, `442`, and `929` before finding the match `613`.

```
      pat.charAt(j)
j     0  1  2  3  4
      2  6  5  3  5  % 997 = 613

                                 txt.charAt(i)
i     0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
      3  1  4  1  5  9  2  6  5  3  5  8  9  7  9  3
0     3  1  4  1  5  % 997 = 508
1        1  4  1  5  9  % 997 = 201
2           4  1  5  9  2  % 997 = 715
3              1  5  9  2  6  % 997 = 971
4                 5  9  2  6  5  % 997 = 442
5                    9  2  6  5  3  % 997 = 929     match
6 <-- return i = 6      2  6  5  3  5  % 997 = 613   <-
```

Basis for Rabin-Karp substring search

Computing the hash function. With five-digit values, we could just do all the necessary calculations with `int` values, but what do we do when `M` is 100 or 1,000? A simple application of Horner's method, precisely like the method that we examined in SECTION 3.4 for strings and other types of keys with multiple values,

leads to the code shown at right, which computes the hash function for an `M`-digit base-`R` number represented as a `char` array in time proportional to `M`. (We pass `M` as an argument so that we can use the method for both the pattern and the text, as you will see.) For each digit in the number, we multiply by `R`, add the digit, and take the remainder when divided by `Q`. For example, computing the hash function for our pattern using this process is shown at the bottom of the page. The same method can work for computing the hash functions in the text, but the cost for the substring search would be a multiplication, addition, and remainder calculation for each text character, for a total of NM operations in the worst case, no improvement over the brute-force method.

```
private long hash(String key, int M)
{  // Compute hash for key[0..M-1].
   long h = 0;
   for (int j = 0; j < M; j++)
      h = (R * h + key.charAt(j)) % Q;
   return h;
}
```

Horner's method, applied to modular hashing

Key idea. The Rabin-Karp method is based on efficiently computing the hash function for position `i+1` in the text, given its value for position `i`. It follows directly from a simple mathematical formulation. Using the notation t_i for `txt.charAt(i)`, the number corresponding to the M-character substring of `txt` that starts at position `i` is

$$x_i = t_i R^{M-1} + t_{i+1} R^{M-2} + \ldots + t_{i+M-1} R^0$$

and we can assume that we know the value of $h(x_i) = x_i \bmod Q$. Shifting one position right in the text corresponds to replacing x_i by

$$x_{i+1} = (x_i - t_i R^{M-1}) R + t_{i+M} .$$

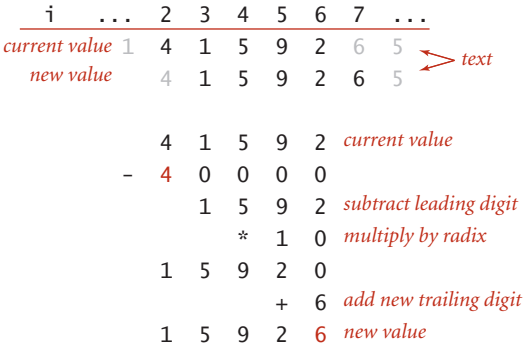
We subtract off the leading digit, multiply by `R`, then add the trailing digit. Now, the crucial point is that we do not have to maintain the values of the numbers, just the values of their remainders when divided by `Q`. A fundamental property of the modulus operation is that if we take the remainder when divided by `Q` after each arithmetic operation, then we get the same answer as if we were to perform all of the arithmetic operations, then take the remainder when divided by `Q`. We took advantage of this property once before, when implementing modular hashing with Horner's method (see page 460). The result is that we can effectively move right one position in the text in *constant* time, whether `M` is 5 or 100 or 1,000.

```
                pat.charAt(j)
 i    0  1  2  3  4
      2  6  5  3  5
 0    2  % 997 = 2
 1    2  6  % 997 = (2*10 + 6) % 997 = 26
 2    2  6  5  % 997 = (26*10 + 5) % 997 = 265
 3    2  6  5  3  % 997 = (265*10 + 3) % 997 = 659
 4    2  6  5  3  5  % 997 = (659*10 + 5) % 997 = 613
```

R Q

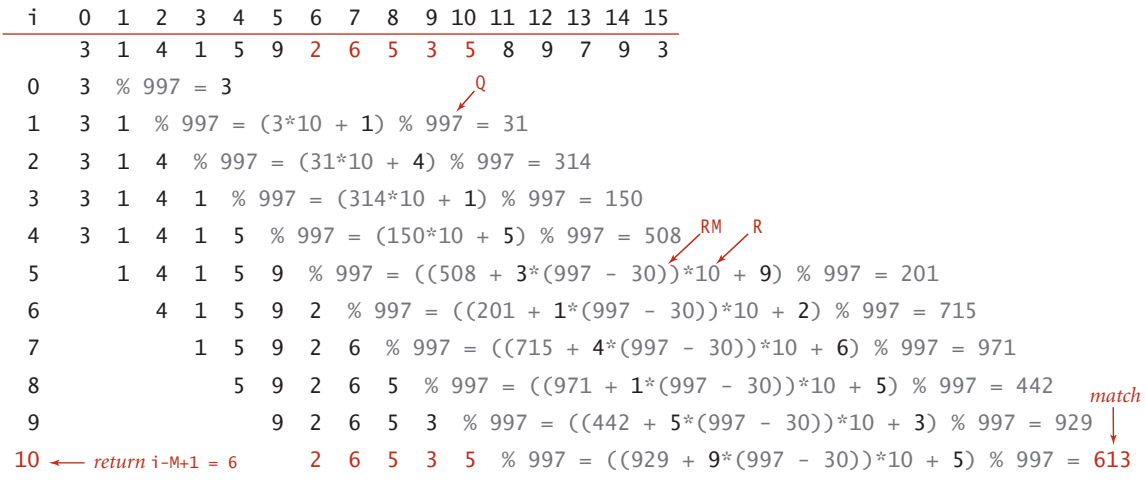
Computing the hash value for the pattern with Horner's method

Implementation. This discussion leads directly to the substring search implementation in ALGORITHM 5.8. The constructor computes a hash value `patHash` for the pattern; it also computes the value of $R^{M-1} \bmod Q$ in the variable `RM`. The `hashSearch()` method begins by computing the hash function for the first `M` characters of the text and comparing that value against the hash value for the pattern. If that is not a match, it proceeds through the text string, using the technique above to maintain the hash function for the `M` characters starting at position `i` for each `i` in a variable `txtHash` and comparing each new hash value to `patHash`. (An extra `Q` is added during the `txtHash` calculation to make sure that everything stays positive so that the remainder operation works as it should.)



Key computation in Rabin-Karp substring search (move right one position in the text)

A trick: Monte Carlo correctness. After finding a hash value for an M-character substring of `txt` that matches the pattern hash value, you might expect to see code to compare those characters with the pattern to ensure that we have a true match, not just a hash collision. We do not do that test because using it requires backup in the text string. Instead, we make the hash table "size" `Q` as large as we wish, since we are not actually building a hash table, just testing for a collision with one key, our pattern. We will use a `long` value greater than 10^{20}, making the probability that a random key hashes to the



Rabin-Karp substring search example

ALGORITHM 5.8 Rabin-Karp fingerprint substring search

```
public class RabinKarp
{
   private String pat;       // pattern (only needed for Las Vegas)
   private long patHash;     // pattern hash value
   private int M;            // pattern length
   private long Q;           // a large prime
   private int R = 256;      // alphabet size
   private long RM;          // R^(M-1) % Q

   public RabinKarp(String pat)
   {
      this.pat = pat;        // save pattern (only needed for Las Vegas)
      this.M = pat.length();
      Q = longRandomPrime();           // See Exercise 5.3.33.
      RM = 1;
      for (int i = 1; i <= M-1; i++)   // Compute R^(M-1) % Q for use
         RM = (R * RM) % Q;            //   in removing leading digit.
      patHash = hash(pat, M);
   }

   public boolean check(int i)  // Monte Carlo (See text.)
   {  return true;  }  //   For Las Vegas, check pat vs txt(i..i-M+1).

   private long hash(String key, int M)
   // See text (page 775).
   private int search(String txt)
   {  // Search for hash match in text.
      int N = txt.length();
      long txtHash = hash(txt, M);
      if (patHash == txtHash) return 0;            // Match at beginning.
      for (int i = M; i < N; i++)
      {  // Remove leading digit, add trailing digit, check for match.
         txtHash = (txtHash + Q - RM*txt.charAt(i-M) % Q) % Q;
         txtHash = (txtHash*R + txt.charAt(i)) % Q;
         if (patHash == txtHash)
           if (check(i - M + 1)) return i - M + 1; // match
      }
      return N;                                    // no match found
    }
}
```

This substring search algorithm is based on hashing. It computes a hash value for the pattern in the constructor, then searches through the text looking for a hash match.

same value as our pattern less than 10^{-20}, an exceedingly small value. If that value is not small enough for you, you could run the algorithms again to get a probability of failure of less than 10^{-40}. This algorithm is an early and famous example of a *Monte Carlo* algorithm that has a guaranteed completion time but fails to output a correct answer with a small probability. The alternative method of checking for a match could be slow (it might amount to the brute-force algorithm, with a very small probability) but is guaranteed correct. Such an algorithm is known as a *Las Vegas* algorithm.

Property P. The Monte Carlo version of Rabin-Karp substring search is linear-time and extremely likely to be correct, and the Las Vegas version of Rabin-Karp substring search is correct and extremely likely to be linear-time.

Discussion: The use of the very large value of `Q`, made possible by the fact that we need not maintain an actual hash table, makes it extremely unlikely that a collision will occur. Rabin and Karp showed that when `Q` is properly chosen, we get a hash collision for random strings with probability `1/Q`, which implies that, for practical values of the variables, there are no hash matches when there are no substring matches and only one hash match if there is a substring match. Theoretically, a text position could lead to a hash collision and not a substring match, but in practice it can be relied upon to find a match.

If your belief in probability theory (or in the random string model and the code we use to generate random numbers) is more half-hearted than resolute, you can add to `check()` the code to check that the text matches the pattern, which turns ALGORITHM 5.8 into the Las Vegas version of the algorithm (see EXERCISE 5.3.12). If you also add a check to see whether that code is ever executed, you might develop more faith in probability theory as time wears on.

RABIN-KARP SUBSTRING SEARCH is known as a *fingerprint* search because it uses a small amount of information to represent a (potentially very large) pattern. Then it looks for this fingerprint (the hash value) in the text. The algorithm is efficient because the fingerprints can be efficiently computed and compared.

Summary The table at the bottom of the page summarizes the algorithms that we have discussed for substring search. As is often the case when we have several algorithms for the same task, each of them has attractive features. Brute-force search is easy to implement and works well in typical cases (Java's `indexOf()` method in `String` uses brute-force search); Knuth-Morris-Pratt is guaranteed linear-time with no backup in the input; Boyer-Moore is sublinear (by a factor of M) in typical situations; and Rabin-Karp is linear. Each also has drawbacks: brute-force might require time proportional to MN; Knuth-Morris-Pratt and Boyer-Moore use extra space; and Rabin-Karp has a relatively long inner loop (several arithmetic operations, as opposed to character compares in the other methods). These characteristics are summarized in the table below.

algorithm	version	operation count		backup in input?	correct?	extra space
		guarantee	typical			
brute force	—	MN	$1.1\,N$	*yes*	*yes*	1
Knuth-Morris-Pratt	*full DFA* (*Algorithm 5.6*)	$2N$	$1.1\,N$	*no*	*yes*	MR
	mismatch transitions only	$3N$	$1.1\,N$	*no*	*yes*	M
Boyer-Moore	*full algorithm*	$3N$	N / M	*yes*	*yes*	R
	mismatched char heuristic only (*Algorithm 5.7*)	MN	N / M	*yes*	*yes*	R
Rabin-Karp†	*Monte Carlo* (*Algorithm 5.8*)	$7\,N$	$7\,N$	*no*	*yes*†	1
	Las Vegas	$7\,N$†	$7\,N$	*yes*	*yes*	1

† *probabilisitic guarantee, with uniform and independent hash function*

Cost summary for substring search implementations

Q&A

Q. This substring search problem seems like a bit of a toy problem. Do I really need to understand these complicated algorithms?

A. Well, the factor of *M* speedup available with Boyer-Moore can be quite impressive in practice. Also, the ability to stream input (no backup) leads to many practical applications for KMP and Rabin-Karp. Beyond these direct practical applications, this topic provides an interesting introduction to the use of abstract machines and randomization in algorithm design.

Q. Why not simplify things by converting each character to binary, treating all text as binary text?

A. That idea is not quite effective because of false matches across character boundaries.

EXERCISES

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

EXERCISES *(continued)*

`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: AAAAAAAAAAAAAAAAAAAAAAAB

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 $(N - M + 1)\ (1 - R^{-M})\ /\ (1 - R^{-1}) \leq 2(N - M + 1)$.

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.

CREATIVE PROBLEMS

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<Integer>` 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 `abcabcababcababcababcabcabcab`.

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?

Answer: None. The method

```
public boolean search(char[] txt)
{  return false; }
```

is quite effective for this problem, since the chances of a random pattern of length 100 appearing in any text are so low that you may consider it to be 0.

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.

```
    int i = -1;
sm: i++;
s0: if (txt[i]) != 'A' goto sm;
s1: if (txt[i]) != 'A' goto s0;
s2: if (txt[i]) != 'B' goto s0;
s3: if (txt[i]) != 'A' goto s2;
s4: if (txt[i]) != 'A' goto s0;
s5: if (txt[i]) != 'A' goto s3;
    return i-8;
```

Straight-line substring search for A A B A A A

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 2^b 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 2^b. For example, if you take 2^b 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.

EXPERIMENTS

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.

This page intentionally left blank