13.2 Pattern-Matching Algorithms

In the classic pattern-matching problem, we are given a text string of length \( n \) and a pattern string of length \( m \leq n \), and must determine whether the pattern is a substring of the text. If so, we may want to find the lowest index within the text at which the pattern begins, or perhaps all indices at which the pattern begins.

The pattern-matching problem is inherent to many behaviors of Java's String class, such as text.contains(pattern) and text.indexOf(pattern), and is a subtask of more complex string operations such as text.replace(pattern, substitute) and text.split(pattern).

In this section, we present three pattern-matching algorithms, with increasing levels of sophistication. Our implementations report the index that begins the left-most occurrence of the pattern, if found. For a failed search, we adopt the conventions of the indexOf method of Java's String class, returning \(-1\) as a sentinel.

13.2.1 Brute Force

The brute-force algorithmic design pattern is a powerful technique for algorithm design when we have something we wish to search for or when we wish to optimize some function. When applying this technique in a general situation, we typically enumerate all possible configurations of the inputs involved and pick the best of all these enumerated configurations.

In applying this technique to design a brute-force pattern-matching algorithm, we derive what is probably the first algorithm that we might think of for solving the problem—we simply test all the possible placements of the pattern relative to the text. An implementation of this algorithm is shown in Code Fragment 13.1.

```java
/** Returns the lowest index at which substring pattern begins in text (or else -1). */
public static int findBrute(char[] text, char[] pattern) {
    int n = text.length;
    int m = pattern.length;
    for (int i=0; i <= n - m; i++) { // try every starting index within text
        int k = 0; // k is index into pattern
        while (k < m && text[i+k] == pattern[k]) // kth character of pattern matches
            k++;
        if (k == m) // if we reach the end of the pattern,
            return i; // substring text[i...i+m-1] is a match
    }
    return -1; // search failed
}
```

**Code Fragment 13.1**: An implementation of the brute-force pattern-matching algorithm. (We use character arrays rather than strings to simplify indexing notation.)
Performance

The analysis of the brute-force pattern-matching algorithm could not be simpler. It consists of two nested loops, with the outer loop indexing through all possible starting indices of the pattern in the text, and the inner loop indexing through each character of the pattern, comparing it to its potentially corresponding character in the text. Thus, the correctness of the brute-force pattern-matching algorithm follows immediately from this exhaustive search approach.

The running time of brute-force pattern matching in the worst case is not good, however, because we can perform up to \( m \) character comparisons for each candidate alignment of the pattern within the text. Referring to Code Fragment 13.1, we see that the outer \texttt{for} loop is executed at most \( n - m + 1 \) times, and the inner \texttt{while} loop is executed at most \( m \) times. Thus, the worst-case running time of the brute-force method is \( O(nm) \).

\textbf{Example 13.1:} Suppose we are given the text string

\[
\text{text} = \text{"abacaabaccabacabaabb"
}
\]

and the pattern string

\[
\text{pattern} = \text{"abacab"
}
\]

\textit{Figure 13.1} illustrates the execution of the brute-force pattern-matching algorithm on this selection of text and pattern.

\begin{figure}
\centering
\includegraphics[width=\textwidth]{example_pattern_matching}
\caption{Example run of the brute-force pattern-matching algorithm. The algorithm performs 27 character comparisons, indicated above with numerical labels.}
\end{figure}
13.2.2 The Boyer-Moore Algorithm

At first, it might seem that it is always necessary to examine every character in the text in order to locate a pattern as a substring or to rule out its existence. But this is not always the case. The Boyer-Moore pattern-matching algorithm, which we will study in this section, can sometimes avoid examining a significant fraction of the character in the text. In this section, we will describe a simplified version of the original algorithm by Boyer and Moore.

The main idea of the Boyer-Moore algorithm is to improve the running time of the brute-force algorithm by adding two potentially time-saving heuristics. Roughly stated, these heuristics are as follows:

**Looking-Glass Heuristic:** When testing a possible placement of the pattern against the text, perform the comparisons against the pattern from right-to-left.

**Character-Jump Heuristic:** During the testing of a possible placement of the pattern within the text, a mismatch of character text[i] = c with the corresponding character pattern[k] is handled as follows. If c is not contained anywhere in the pattern, then shift the pattern completely past text[i] = c. Otherwise, shift the pattern until an occurrence of character c gets aligned with text[i].

We will formalize these heuristics shortly, but at an intuitive level, they work as an integrated team to allow us to avoid comparisons with whole groups of characters in the text. In particular, when a mismatch is found near the right end of the pattern, we may end up realigning the pattern beyond the mismatch, without ever examining several characters of the text preceding the mismatch. For example, Figure 13.2 demonstrates a few simple applications of these heuristics. Notice that when the characters s and i mismatch at the right end of the original placement of the pattern, we slide the pattern beyond the mismatched character, without ever examining the first four characters of the text.

![Figure 13.2](image)

**Figure 13.2:** A simple example demonstrating the intuition of the Boyer-Moore pattern-matching algorithm. The original comparison results in a mismatch with character e of the text. Because that character is nowhere in the pattern, the entire pattern is shifted beyond its location. The second comparison is also a mismatch, but the mismatched character s occurs elsewhere in the pattern. The pattern is then shifted so that its last occurrence of s is aligned with the corresponding s in the text. The remainder of the process is not illustrated in this figure.
The example of Figure 13.2 is rather basic, because it only involves mismatches with the last character of the pattern. More generally, when a match is found for that last character, the algorithm continues by trying to extend the match with the second-to-last character of the pattern in its current alignment. That process continues until either matching the entire pattern, or finding a mismatch at some interior position of the pattern.

If a mismatch is found, and the mismatched character of the text does not occur in the pattern, we shift the entire pattern beyond that location, as originally illustrated in Figure 13.2. If the mismatched character occurs elsewhere in the pattern, we must consider two possible subcases depending on whether its last occurrence is before or after the character of the pattern that was mismatched. Those two cases are illustrated in Figure 13.3.

In the case of Figure 13.3(b), we slide the pattern only one unit. It would be more productive to slide it rightward until finding another occurrence of mismatched character text[i] in the pattern, but we do not wish to take time to search

```
Text:  · · · · · · · · · · a · · · · · · ·
Pattern:  · a · · b ·  

(a)

Text:  · · · · · · · · · · · · · · ·
Pattern:  · · · · · · · · · · b a

(b)
```

**Figure 13.3:** Additional rules for the character-jump heuristic of the Boyer-Moore Algorithm. We let $i$ represent the index of the mismatched character in the text, $k$ represent the corresponding index in the pattern, and $j$ represent the index of the last occurrence of text[i] within the pattern. We distinguish two cases: (a) $j < k$, in which case we shift the pattern by $k - j$ units, and thus, index $i$ advances by $m - (j + 1)$ units; (b) $j > k$, in which case we shift the pattern by one unit, and index $i$ advances by $m - k$ units.
for another occurrence. The efficiency of the Boyer-Moore algorithm relies on quickly determining where a mismatched character occurs elsewhere in the pattern. In particular, we define a function last(c) as

- If c is in the pattern, last(c) is the index of the last (rightmost) occurrence of c in the pattern. Otherwise, we conventionally define last(c) = −1.

If we assume that the alphabet is of fixed, finite size, and that characters can be converted to indices of an array (for example, by using their character code), the last function can be easily implemented as a lookup table with worst-case $O(1)$-time access to the value last(c). However, the table would have length equal to the size of the alphabet (rather than the size of the pattern), and time would be required to initialize the entire table.

We prefer to use a hash table to represent the last function, with only those characters from the pattern occurring in the map. The space usage for this approach is proportional to the number of distinct alphabet symbols that occur in the pattern, and thus $O(\max(m, |\Sigma|))$. The expected lookup time remains $O(1)$ (as does the worst-case, if we consider $|\Sigma|$ a constant). Our complete implementation of the Boyer-Moore pattern-matching algorithm is given in Code Fragment 13.2.

```java
/** Returns the lowest index at which substring pattern begins in text (or else -1). */
public static int findBoyerMoore(char[] text, char[] pattern) {
    int n = text.length;
    int m = pattern.length;
    if (m == 0) return 0; // trivial search for empty string
    Map<Character, Integer> last = new HashMap<>(); // the 'last' map
    for (int i = 0; i < n; i++)
        last.put(text[i], -1); // set -1 as default for all text characters
    for (int k = 0; k < m; k++)
        last.put(pattern[k], k); // rightmost occurrence in pattern is last
    // start with the end of the pattern aligned at index m-1 of the text
    int i = m-1; // an index into the text
    int k = m-1; // an index into the pattern
    while (i < n) {
        if (text[i] == pattern[k]) {
            if (k == 0) return i; // entire pattern has been found
            i--; // otherwise, examine previous
            k--; // characters of text/pattern
        } else {
            i += m - Math.min(k, 1 + last.get(text[i])); // case analysis for jump step
            k = m - 1; // restart at end of pattern
        }
    }
    return -1; // pattern was never found
}
```

**Code Fragment 13.2**: An implementation of the Boyer-Moore algorithm.
The correctness of the Boyer-Moore pattern-matching algorithm follows from the fact that each time the method makes a shift, it is guaranteed not to "skip" over any possible matches. For \( \text{last}(c) \) is the location of the last occurrence of \( c \) in the pattern. In Figure 13.4, we illustrate the execution of the Boyer-Moore pattern-matching algorithm on an input string similar to Example 13.1.

<table>
<thead>
<tr>
<th>( c )</th>
<th>( a )</th>
<th>( b )</th>
<th>( c )</th>
<th>( d )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{last}(c) )</td>
<td>4</td>
<td>5</td>
<td>3</td>
<td>-1</td>
</tr>
</tbody>
</table>

Text: \[ a b a c a a b a d c a b a c a b a a b b \]

Pattern: \[ a b a c a b \]

Figure 13.4: An illustration of the Boyer-Moore pattern-matching algorithm, including a summary of the \( \text{last}(c) \) function. The algorithm performs 13 character comparisons, which are indicated with numerical labels.

Performance

If using a traditional lookup table, the worst-case running time of the Boyer-Moore algorithm is \( O(nm + |\Sigma|) \). The computation of the last function takes \( O(m + |\Sigma|) \) time, although the dependence on \( |\Sigma| \) is removed if using a hash table. The actual search for the pattern takes \( O(nm) \) time in the worst case—the same as the brute-force algorithm. An example that achieves the worst case for Boyer-Moore is

\[
\text{text} = \underbrace{a a a a a a}_{n} \cdots a
\]

\[
\text{pattern} = \underbrace{b a a \cdots a}_{m-1}
\]

The worst-case performance, however, is unlikely to be achieved for English text; in that case, the Boyer-Moore algorithm is often able to skip large portions of text. Experimental evidence on English text shows that the average number of comparisons done per character is 0.24 for a five-character pattern string.

We have actually presented a simplified version of the Boyer-Moore algorithm. The original algorithm achieves worst-case running time \( O(n + m + |\Sigma|) \) by using an alternative shift heuristic for a partially matched text string, whenever it shifts the pattern more than the character-jump heuristic. This alternative shift heuristic is based on applying the main idea from the Knuth-Morris-Pratt pattern-matching algorithm, which we discuss next.
13.5.2 DNA and Text Sequence Alignment

A common text-processing problem, which arises in genetics and software engineering, is to test the similarity between two text strings. In a genetics application, the two strings could correspond to two strands of DNA, for which we want to compute similarities. Likewise, in a software engineering application, the two strings could come from two versions of source code for the same program, for which we want to determine changes made from one version to the next. Indeed, determining the similarity between two strings is so common that the Unix and Linux operating systems have a built-in program, named `diff`, for comparing text files.

Given a string $X = x_0x_1x_2 \cdots x_{n-1}$, a subsequence of $X$ is any string that is of the form $x_{i_j}x_{i_j+1} \cdots x_{i_k}$, where $i_j < i_{j+1}$; that is, it is a sequence of characters that are not necessarily contiguous but are nevertheless taken in order from $X$. For example, the string `AAAG` is a subsequence of the string `CGATAATTGAGA`.

The DNA and text similarity problem we address here is the longest common subsequence (LCS) problem. In this problem, we are given two character strings, $X = x_0x_1x_2 \cdots x_{n-1}$ and $Y = y_0y_1y_2 \cdots y_{m-1}$, over some alphabet (such as the alphabet $\{A,C,G,T\}$ common in computational genomics) and are asked to find a longest string $S$ that is a subsequence of both $X$ and $Y$. One way to solve the longest common subsequence problem is to enumerate all subsequences of $X$ and take the largest one that is also a subsequence of $Y$. Since each character of $X$ is either in or not in a subsequence, there are potentially $2^n$ different subsequences of $X$, each of which requires $O(m)$ time to determine whether it is a subsequence of $Y$. Thus, this brute-force approach yields an exponential-time algorithm that runs in $O(2^m)$ time, which is very inefficient. Fortunately, the LCS problem is efficiently solvable using dynamic programming.

The Components of a Dynamic Programming Solution

As mentioned above, the dynamic programming technique is used primarily for optimization problems, where we wish to find the “best” way of doing something. We can apply the dynamic programming technique in such situations if the problem has certain properties:

**Simple Subproblems:** There has to be some way of repeatedly breaking the global optimization problem into subproblems. Moreover, there should be a way to parameterize subproblems with just a few indices, like $i$, $j$, $k$, and so on.

**Subproblem Optimization:** An optimal solution to the global problem must be a composition of optimal subproblem solutions.

**Subproblem Overlap:** Optimal solutions to unrelated subproblems can contain subproblems in common.
Applying Dynamic Programming to the LCS Problem

Recall that in the LCS problem, we are given two character strings, $X$ and $Y$, of length $n$ and $m$, respectively, and are asked to find a longest string $S$ that is a subsequence of both $X$ and $Y$. Since $X$ and $Y$ are character strings, we have a natural set of indices with which to define subproblems—indices into the strings $X$ and $Y$. Let us define a subproblem, therefore, as that of computing the value $L_{j,k}$, which we will use to denote the length of a longest string that is a subsequence of both the first $j$ characters of $X$ and the first $k$ characters of $Y$, that is of prefixes $X[0..j-1]$ and $Y[0..k-1]$. If either $j = 0$ or $k = 0$, then $L_{j,k}$ is trivially defined as 0.

When both $j \geq 1$ and $k \geq 1$, this definition allows us to rewrite $L_{j,k}$ recursively in terms of optimal subproblem solutions. This definition depends on which of two cases we are in. (See Figure 13.13.)

- $x_{j-1} = y_{k-1}$. In this case, we have a match between the last character of $X[0..j-1]$ and the last character of $Y[0..k-1]$. We claim that this character belongs to a longest common subsequence of $X[0..j-1]$ and $Y[0..k-1]$. To justify this claim, let us suppose it is not true. There must be some longest common subsequence $x_{a_1}x_{a_2}...x_{a_c} = y_{b_1}y_{b_2}...y_{b_c}$. If $x_{a_c} = x_{j-1}$ or $y_{b_c} = y_{k-1}$, then we get the same sequence by setting $a_c = j - 1$ and $b_c = k - 1$. Alternately, if $x_{a_c} \neq x_{j-1}$ and $y_{b_c} \neq y_{k-1}$, then we can get an even longer common subsequence by adding $x_{j-1} = y_{k-1}$ to the end. Thus, a longest common subsequence of $X[0..j-1]$ and $Y[0..k-1]$ ends with $x_{j-1}$. Therefore, we set

$$L_{j,k} = L_{j-1,k-1} + 1$$

if $x_{j-1} = y_{k-1}$.

- $x_{j-1} \neq y_{k-1}$. In this case, we cannot have a common subsequence that includes both $x_{j-1}$ and $y_{k-1}$. That is, we can have a common subsequence end with $x_{j-1}$ or one that ends with $y_{k-1}$ (or possibly neither), but certainly not both. Therefore, we set

$$L_{j,k} = \max\{L_{j-1,k}, L_{j,k-1}\}$$

if $x_{j-1} \neq y_{k-1}$.

\[L_{10,12} = 1 + L_{9,11}\]

\[L_{9,11} = \max(L_{9,10}, L_{8,11})\]

Figure 13.13: The two cases in the longest common subsequence algorithm for computing $L_{j,k}$ when $j,k \geq 1$: (a) $x_{j-1} = y_{k-1}$; (b) $x_{j-1} \neq y_{k-1}$.
The LCS Algorithm

The definition of $L_{j,k}$ satisfies subproblem optimization, for we cannot have a longest common subsequence without also having longest common subsequences for the subproblems. Also, it uses subproblem overlap, because a subproblem solution $L_{j,k}$ can be used in several other problems (namely, the problems $L_{j+1,k}$, $L_{j,k+1}$, and $L_{j+1,k+1}$). Turning this definition of $L_{j,k}$ into an algorithm is actually quite straightforward. We create an $(n+1) \times (m+1)$ array, $L$, defined for $0 \leq j \leq n$ and $0 \leq k \leq m$. We initialize all entries to 0, in particular so that all entries of the form $L_{j,0}$ and $L_{0,k}$ are zero. Then, we iteratively build up values in $L$ until we have $L_{n,m}$, the length of a longest common subsequence of $X$ and $Y$. We give a Java implementation of this algorithm in Code Fragment 13.7.

```java
/** Returns table such that L[j][k] is length of LCS for X[0..j-1] and Y[0..k-1]. */
public static int[][] LCS(char[] X, char[] Y) {
    int n = X.length;
    int m = Y.length;
    int[][] L = new int[n+1][m+1];
    for (int j=0; j < n; j++)
        for (int k=0; k < m; k++)
            if (X[j] == Y[k]) // align this match
                L[j+1][k+1] = L[j][k] + 1;
            else // choose to ignore one character
                L[j+1][k+1] = Math.max(L[j][k+1], L[j+1][k]);
    return L;
}
```

Code Fragment 13.7: Dynamic programming algorithm for the LCS problem.

The running time of the algorithm of the LCS algorithm is easy to analyze, for it is dominated by two nested for loops, with the outer one iterating $n$ times and the inner one iterating $m$ times. Since the if-statement and assignment inside the loop each requires $O(1)$ primitive operations, this algorithm runs in $O(nm)$ time. Thus, the dynamic programming technique can be applied to the longest common subsequence problem to improve significantly over the exponential-time brute-force solution to the LCS problem.

The LCS method of Code Fragment 13.7 computes the length of the longest common subsequence (stored as $L_{n,m}$), but not the subsequence itself. Fortunately, it is easy to extract the actual longest common subsequence if given the complete table of $L_{j,k}$ values computed by the LCS method. The solution can be reconstructed back to front by reverse engineering the calculation of length $L_{n,m}$. At any position $L_{j,k}$, if $x_j = y_k$, then the length is based on the common subsequence associated with length $L_{j-1,k-1}$, followed by common character $x_j$. We can record $x_j$ as part of the sequence, and then continue the analysis from $L_{j-1,k-1}$. If $x_j \neq y_k$, then we can move to the larger of $L_{j,k-1}$ and $L_{j-1,k}$. We continue this process until reaching
some $L_{j,k} = 0$ (for example, if $j$ or $k$ is 0 as a boundary case). A Java implementation of this strategy is given in Code Fragment 13.8. This method constructs a longest common subsequence in $O(n+m)$ additional time, since each pass of the \textit{while} loop decrements either $j$ or $k$ (or both). An illustration of the algorithm for computing the longest common subsequence is given in Figure 13.14.

```java
/** Returns the longest common substring of X and Y, given LCS table L. */
public static char[] reconstructLCS(char[] X, char[] Y, int[][] L) {
    StringBuilder solution = new StringBuilder;
    int j = X.length;
    int k = Y.length;
    while (L[j][k] > 0) // common characters remain
        if (X[j-1] == Y[k-1]) {
            j--;
            k--;
        } else if (L[j-1][k] >= L[j][k-1])
            j--;
        else
            k--;
    // return left-to-right version, as char array
    return solution.reverse().toString().toCharArray();
}
```

\textbf{Code Fragment 13.8: Reconstructing the longest common subsequence.}

\begin{figure}[h]
\centering
\begin{tabular}{cccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
2 & 0 & 0 & 1 & 1 & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 \\
3 & 0 & 0 & 1 & 1 & 2 & 2 & 2 & 2 & 3 & 3 & 3 & 3 \\
4 & 0 & 1 & 1 & 1 & 2 & 2 & 2 & 2 & 3 & 3 & 3 & 3 \\
5 & 0 & 1 & 1 & 1 & 2 & 2 & 2 & 2 & 3 & 3 & 3 & 3 \\
6 & 0 & 1 & 1 & 1 & 2 & 2 & 2 & 3 & 4 & 4 & 4 & 4 \\
7 & 0 & 1 & 1 & 2 & 2 & 3 & 3 & 4 & 4 & 4 & 5 & 5 \\
8 & 0 & 1 & 1 & 2 & 2 & 3 & 4 & 4 & 4 & 5 & 5 & 5 \\
9 & 0 & 1 & 1 & 2 & 3 & 3 & 4 & 5 & 5 & 5 & 5 & 5 \\
10 & 0 & 1 & 1 & 2 & 3 & 4 & 5 & 5 & 5 & 6 & 6 & 6 \\
\end{tabular}
\end{figure}

Figure 13.14: Illustration of the algorithm for constructing a longest common subsequence from the array $L$. A diagonal step on the highlighted path represents the use of a common character (with that character’s respective indices in the sequences highlighted in the margins).