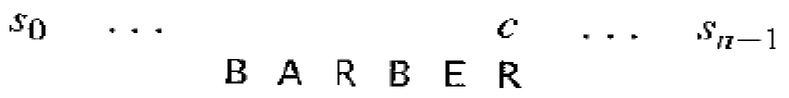
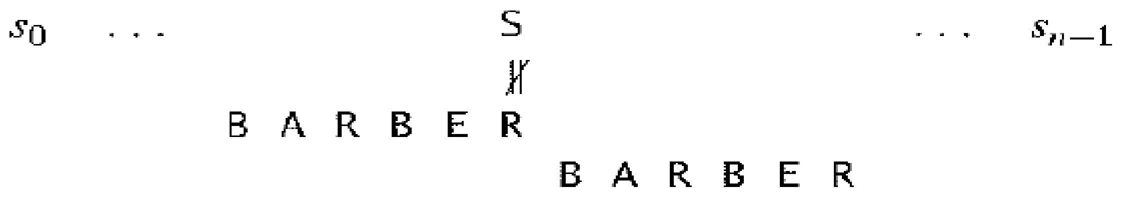
Horspool's Algorithm

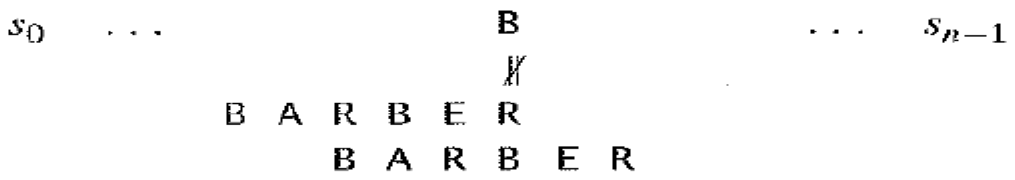
Consider, as an example, searching for the pattern BARBER in some text:



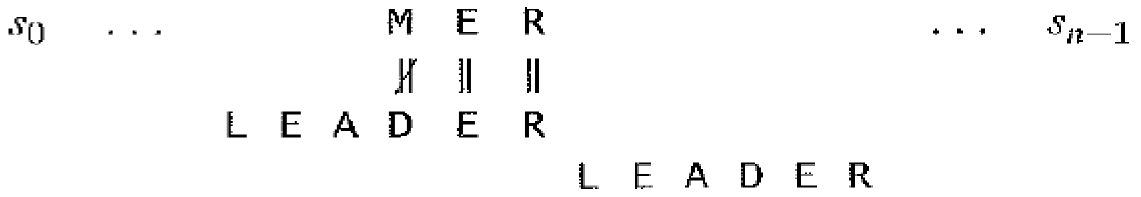
Case 1 If there are no c's in the pattern—e.g., c is letter S in our example—we can safely shift the pattern by its entire length (if we shift less, some character of the pattern would be aligned against the text's character c that is known not to be in the pattern):



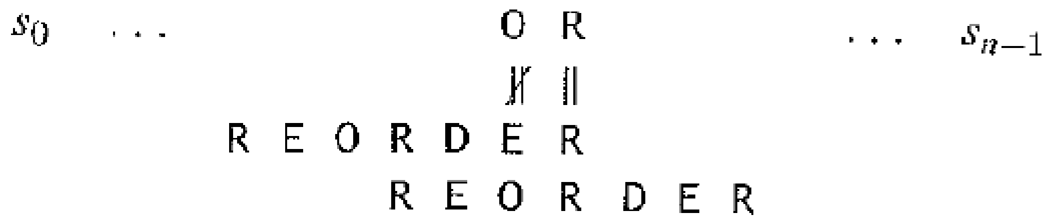
Case 2 If there are occurrences of character c in the pattern but it is not the last one there—e.g., c is letter B in our example—the shift should align the rightmost occurrence of c in the pattern with the c in the text:



Case 3 If c happens to be the last character in the pattern but there are no c's among its other $m-1$ characters, the shift should be similar to that of Case 1: the pattern should be shifted by the entire pattern's length m, e.g.,



Case 4 Finally, if c happens to be the last character in the pattern and there are other c's among its first $m-1$ characters, the shift should be similar to that of Case 2: the rightmost occurrence of c among the first $m-1$ characters in the pattern should be aligned with the text's c, e.g.,



The table will be indexed by all possible characters that can be encountered in a text, including, for natural language texts, the space, punctuation symbols, and other special characters. (Note that no other information about the text in which eventual searching will be done is required.) The table's entries will indicate the shift sizes computed by the formula

$$t(c) = \begin{cases} \text{the pattern's length } m, \\ \text{if } c \text{ is not among the first } m-1 \text{ characters of the pattern} \\ \\ \text{the distance from the rightmost } c \text{ among the first } m-1 \text{ characters} \\ \text{of the pattern to its last character, otherwise} \end{cases} \qquad \textbf{(7.1)}$$

ALGORITHM *ShiftTable*($P[0..m-1]$)

```
//Fills the shift table used by Horspool's and Boyer-Moore algorithms
//Input: Pattern $P[0..m-1]$ and an alphabet of possible characters
//Output: $Table[0..size-1]$ indexed by the alphabet's characters and
//          filled with shift sizes computed by formula (7.1)
initialize all the elements of $Table$ with $m$
for $j \leftarrow 0$ to $m-2$ do $Table[P[j]] \leftarrow m-1-j$
return $Table$
```

Horspool's algorithm

Step 1 For a given pattern of length m and the alphabet used in both the pattern and text, construct the shift table as described above.

Step 2 Align the pattern against the beginning of the text.

Step 3 Repeat the following until either a matching substring is found or the pattern reaches beyond the last character of the text. Starting with the last character in the pattern, compare the corresponding characters in the pattern and text until either all m characters are matched (then stop) or a mismatching pair is encountered. In the latter case, retrieve the entry $t(c)$ from the c's column of the shift table where c is the text's character currently aligned against the last character of the pattern, and shift the pattern by $t(c)$ characters to the right along the text.

Here is a pseudocode of Horspool's algorithm.

```
ALGORITHM  HorspoolMatching(P[0..m − 1], T[0..n − 1])
    //Implements Horspool's algorithm for string matching
    //Input: Pattern P[0..m − 1] and text T[0..n − 1]
    //Output: The index of the left end of the first matching substring
    //        or −1 if there are no matches
    ShiftTable(P[0..m − 1])    //generate Table of shifts
    i ← m − 1                  //position of the pattern's right end
    while i ≤ n − 1 do
        k ← 0                  //number of matched characters
        while k ≤ m − 1 and P[m − 1 − k] = T[i − k] do
            k ← k + 1
        if k = m
            return i − m + 1
        else i ← i+Table[T[i]]
    return −1
```

EXAMPLE As an example of a complete application of Horspool's algorithm, consider searching for the pattern BARBER in a text that comprises English letters and spaces (denoted by underscores). The shift table, as we mentioned, is filled as follows:

character c	A	B	C	D	E	F	. . .	R	. . .	Z	_
shift $t(c)$	4	2	6	6	1	6	6	3	6	6	6

The actual search in a particular text proceeds as follows:

```
J I M _ S A W _ M E _ I N _ A _ B A R B E R S H O P
B A R B E R           B A R B E R
        B A R B E R       B A R B E R
          B A R B E R           B A R B E R
```

Boyer-Moore Algorithm

If the first comparison of the rightmost character in the pattern with the corresponding character *c* in the text fails, the algorithm does exactly the same thing as Horspool's algorithm. Namely, it shifts the pattern to the right by the number of characters retrieved from the table precomputed

The two algorithms act differently, however, after some positive number *k* ($0 < k < m$) of the pattern's characters are matched successfully before a mismatch is encountered:

$$\begin{array}{ccccccccccc}
s_0 & \cdots & & c & s_{i-k+1} & \cdots & s_i & \cdots & s_{n-1} & \text{text} \\
 & & & \not\| & \| & & \| & & & \\
 & p_0 & \cdots & p_{m-k-1} & p_{m-k} & \cdots & p_{m-1} & & & \text{pattern}
\end{array}$$

In this situation, the Boyer-Moore algorithm determines the shift size by considering two quantities.

The first one is guided by the text's character *c* that caused a mismatch with its counterpart in the pattern. Accordingly, it is called the ***bad symbol shift.***

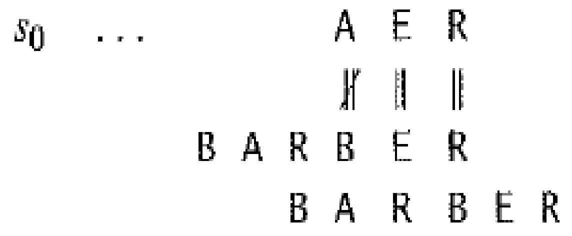
If c is not in the pattern, we shift the pattern to just pass this *c* in the text. Conveniently, the size of this shift can be computed by the formula **t1(c)- *k*,** where t1(c) is the entry in the precomputed table used by Horspool's algorithm and *k* is the number of matched characters:

$$\begin{array}{cccccccccc}
s_0 & \dots & & c & s_{i-k+1} & \dots & s_i & \dots & s_{n-1} & \text{text} \\
 & & & \not\Vert & \Vert & & \Vert & & & \\
 & p_0 & \dots & p_{m-k-1} & p_{m-k} & \dots & p_{m-1} & & & \text{pattern} \\
 & & & & p_0 & \dots & & p_{m-1} & &
\end{array}$$

For example, if we search for the pattern BARBER in some text and match the last two characters before failing on letter S in the text, we can shift the pattern by $t_1(\mathsf{S}) - 2 = 6 - 2 = 4$ positions:

$$\begin{array}{cccccccccccc}
s_0 & \dots & & & & \mathsf{S} & \mathsf{E} & \mathsf{R} & & & \dots & s_{n-1} \\
 & & & & & \not\Vert & \Vert & \Vert & & & & \\
 & & \mathsf{B} & \mathsf{A} & \mathsf{R} & \mathsf{B} & \mathsf{E} & \mathsf{R} & & & & \\
 & & & & & & \mathsf{B} & \mathsf{A} & \mathsf{R} & \mathsf{B} & \mathsf{E} & \mathsf{R}
\end{array}$$

The same formula can also be used when the mismatching character c of the text occurs in the pattern, provided $t_1(c) - k > 0$. For example, if we search for the pattern BARBER in some text and match the last two characters before failing on letter A, we can shift the pattern by $t_1(\text{A}) - 2 = 4 - 2 = 2$ positions:


s_0 ... A E R ... s_{n-1}
$\nparallel$ $\parallel$ $\parallel$
B A R B E R
B A R B E R

If $t_1(c) - k \le 0$, we obviously do not want to shift the pattern by 0 or a negative number of positions. Rather, we can fall back on the brute-force thinking and simply shift the pattern by one position to the right.

To summarize, the bad-symbol shift d_1 is computed by the Boyer-Moore algorithm either as $t_1(c) - k$ if this quantity is positive or as 1 if it is negative or zero. This can be expressed by the following compact formula:

$$d_1 = \max\{t_1(c) - k,\ 1\}. \tag{7.2}$$

The second type of shift is guided by a successful match of the last $k > 0$ characters of the pattern. We refer to the ending portion of the pattern as its suffix of size k and denote it *suff(k*). Accordingly, we call this type of shift the ***good-suffix shift.*** We now apply the reasoning that guided us in filling the bad-symbol shift table, which was based on a single alphabet character c, to the pattern's suffixes of sizes 1, ... , m - 1 to fill in the good-suffix shift table

Let us first consider the case when there is another occurrence of *suff(k)* in the pattern or, to be more accurate, there is another occurrence of *suff(k)* not preceded by the same character as in its last occurrence. (It would be useless to shift the pattern to match another occurrence of *suff(k)* preceded by the same character because this would simply repeat a failed trial.) In this case, we can shift the pattern by the distance d2 between such a second rightmost occurrence (not preceded by the same character as in the last occurrence) of *suff(k)* and its rightmost occurrence.

k	**pattern**	d_2
1	$\text{ABC}\overline{\text{B}}\text{A}\underline{\text{B}}$	2
2	$\overline{\text{AB}}\text{CB}\underline{\text{AB}}$	4

What is to be done if there is no other occurrence of *suff*(k) not preceded by the same character as in its last occurrence? In most cases, we can shift the pattern by its entire length m. For example, for the pattern DBCBAB and $k = 3$, we can shift the pattern by its entire length of 6 characters:

```
s0   ...      c  B  A  B                ...  s_{n-1}
              ∦  ||  ||  ||
        D  B  C  B  A  B
                          D  B  C  B  A  B
```

Unfortunately, shifting the pattern by its entire length when there is no other occurrence of *suff*(k) not preceded by the same character as in its last occurrence is not always correct. For example, for the pattern ABCBAB and $k = 3$, shifting by 6 could miss a matching substring that starts with the text's AB aligned with the last two characters of the pattern:

```
s0   ...      c  B  A  B  C  B  A  B       ...  s_{n-1}
              ∦  ||  ||  ||
        A  B  C  B  A  B
                          A  B  C  B  A  B
```

Note that the shift by 6 is correct for the pattern DBCBAB but not for ABCBAB, because the latter pattern has the same substring AB as its prefix (beginning part of the pattern) and as its suffix (ending part of the pattern). To avoid such an erroneous shift based on a suffix of size k, for which there is no other occurrence in the pattern not preceded by the same character as in its last occurrence, we need to find the longest prefix of size $l < k$ that matches the suffix of the same size l. If such a prefix exists, the shift size d_2 is computed as the distance between this prefix and the corresponding suffix; otherwise, d_2 is set to the pattern's length m. As an example, here is the complete list of the d_2 values—the good-suffix table of the Boyer-Moore algorithm—for the pattern ABCBAB:

k	**pattern**	d_2
1	$\mathrm{ABC\overline{B}A\underline{B}}$	2
2	$\mathrm{\overline{AB}CB\underline{AB}}$	4
3	$\mathrm{\overline{AB}C\underline{BAB}}$	4
4	$\mathrm{\overline{AB}\underline{CBAB}}$	4
5	$\mathrm{\overline{A}\underline{BCBAB}}$	4

The Boyer-Moore algorithm

Step 1 For a given pattern and the alphabet used in both the pattern and the text, construct the bad-symbol shift table as described earlier.

Step 2 Using the pattern, construct the good-suffix shift table as described.

Step 3 Align the pattern against the beginning of the text.

Step 4 Repeat the following step until either a matching substring is found or the pattern reaches beyond the last character of the text. Starting with the last character in the pattern, compare the corresponding characters in the pattern and the text until either all m character pairs are matched (then stop) or a mismatching pair is encountered after $k \geq 0$ character pairs are matched successfully. In the latter case, retrieve the entry $t_1(c)$ from the c's column of the bad-symbol table where c is the text's mismatched character. If $k > 0$, also retrieve the corresponding d_2 entry from the good-suffix table. Shift the pattern to the right by the number of positions computed by the formula

$$d = \begin{cases} d_1 & \text{if } k = 0 \\ \max\{d_1, d_2\} & \text{if } k > 0 \end{cases}, \tag{7.3}$$

where $d_1 = \max\{t_1(c) - k, 1\}$.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
B	E	S	S	_	K	N	E	W	_	A	B	O	U	T	_	B	A	O	B	A	B	S
B	A	O	B	A	B																	
$d_1 = t_1(K) - 0 = 6$						B	A	O	B	A	B											
						$d_1 = t_1(\_) - 2 = 4$					B	A	O	B	A	B						
						$d_2 = 5$					$d_1 = t_1(\_) - 1 = 5$											
						$d = \max\{4, 5\} = 5$					$d_2 = 2$											
											$d = \max\{5, 2\} = 5$											
																B	A	O	B	A	B	

FIGURE 7.3 Example of string matching with the Boyer-Moore algorithm

The good-suffix table is filled as follows:

k	**pattern**	d_2
1	BAOBAB	2
2	BAOBAB	5
3	BAOBAB	5
4	BAOBAB	5
5	BAOBAB	5

c	A	B	C	D	. . .	O	. . .	Z	_
$t_1(c)$	1	2	6	6	6	3	6	6	6

Bad Symbol Shift table

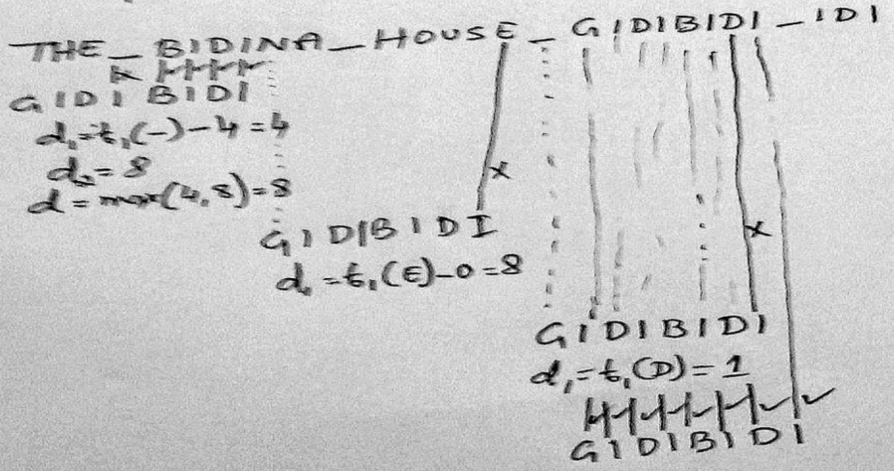
A	B	C	D	G	I	...	Z	_
8	3	8	1	7	2	8	8	8

0.5 M

Good suffix shift table

k		
1	GIDIBIDI	2
2	GIDIBIDI	8
3	GIDIBIDI	4
4	GIDIBIDI	8
5	GIDIBIDI	8
6	GIDIBIDI	8
7	GIDIBIDI	8

0.5 M



THE_BIDINA_HOUSE_GIDIBIDI_IDI

GIDIBIDI

$d_1 = t_1(\_) - 4 = 4$

$d_2 = 8$

$d = \max(4, 8) = 8$

GIDIBIDI

$d_1 = t_1(E) - 0 = 8$

GIDIBIDI

$d_1 = t_1(D) = 1$

GIDIBIDI

1 M