Data Structures and Algorithms

String (Pattern) Matching/Searching Algorithms

Pattern Search Applications

Problem Domain	Application	Input Pattern	Pattern Class
Bioinformatics	Sequence Analysis	DNA/Protein sequence	Known type of genes/patterns
Data mining	Searching for meaningful patterns	Points in multi dimension space	Compact and well separated clusters
Document classification	Internet search	Text document	Semantic categories
Document image analysis	Reading machine for the blind	Document image	Alphanumeric characters / words
Industrial automation	Printed circuit board inspection	Intensity or range image	Defective / non defective nature of product
Multimedia database retrieval	Internet search	Video clip	Video genres e.g. action, dialogue etc
Biometric recognition	Personal identification	Face, iris & finger print	Authorized user for access control
Remote sensing	Forecasting crop yield	Multispectral image	Land use categories, growth pattern of crops
Speech recognition	Telephone directory enquiry with operator	Speech waveform	Spoken words

Exact pattern matching

Problem:

Find first match of a pattern of length M in a text stream of length N.

typically N >> M

pattern

```
n e e d l e
```

M = 6

text

```
i n a h a y s t a c k a n e e d l e i n a
```

N = 21

Applications.

- parsers.
- spam filters.
- digital libraries.
- screen scrapers.
- word processors.
- web search engines.
- natural language processing.
- computational molecular biology.
- feature detection in digitized images.

. . .

String Matching Problem

- Given a text T and a pattern P, find all occurrences of P within T
- Notations:
  - n and m: lengths of P and T
  - Σ: set of alphabets (of constant size)
  - P_i: ith letter of P (1-indexed)
  - a, b, c: single letters in Σ
  - x, y, z: strings

Brute-force exact pattern match

Check for pattern starting at each text position.

```
h a y n e e d s a n n e e d l e x
n e e d l e
  n e e d l e
    n e e d l e
      n e e d l e
        n e e d l e
          n e e d l e
            n e e d l e
              n e e d l e
                n e e d l e
                  n e e d l e
                    n e e d l e
```

```
public static int search(String pattern, String text)
{
   int M = pattern.length();
   int N = text.length();

   for (int i = 0; i < N - M; i++)
   {
      int j;
      for (j = 0; j < M; j++)
         if (text.charAt(i+j) != pattern.charAt(j))
            break;
      if (j == M) return i;   ← pattern start index in text
   }
   return -1;   ← not found
}
```

Example

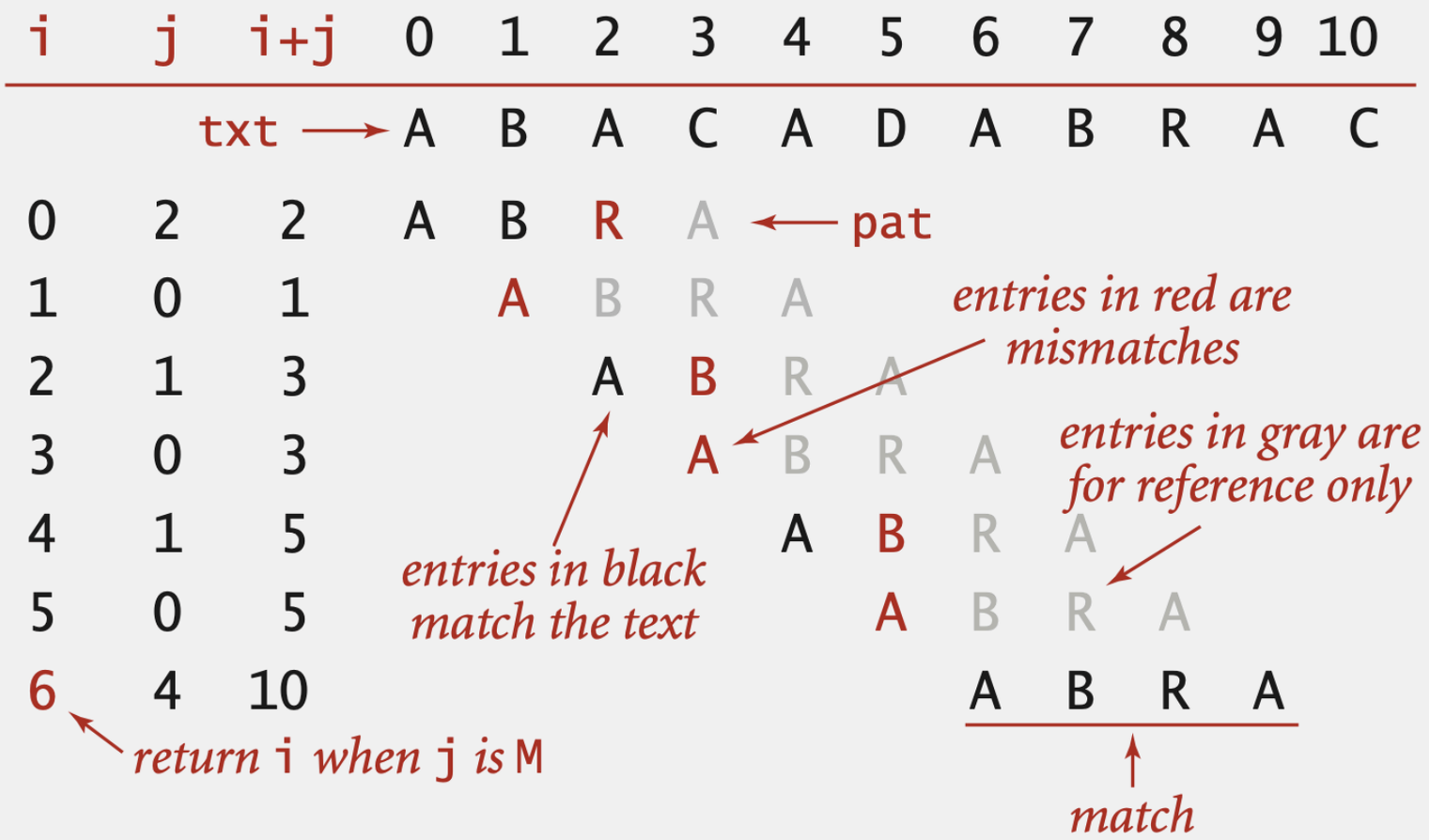
- $T = \texttt{AGCATGCTGCAGTCATGCTTAGGCTA}$
- $P = \texttt{GCT}$
- P appears three times in T

- A naive method takes $O(mn)$ time
  - Initiate string comparison at every starting point
  - Each comparison takes $O(m)$ time

- We can do much better!

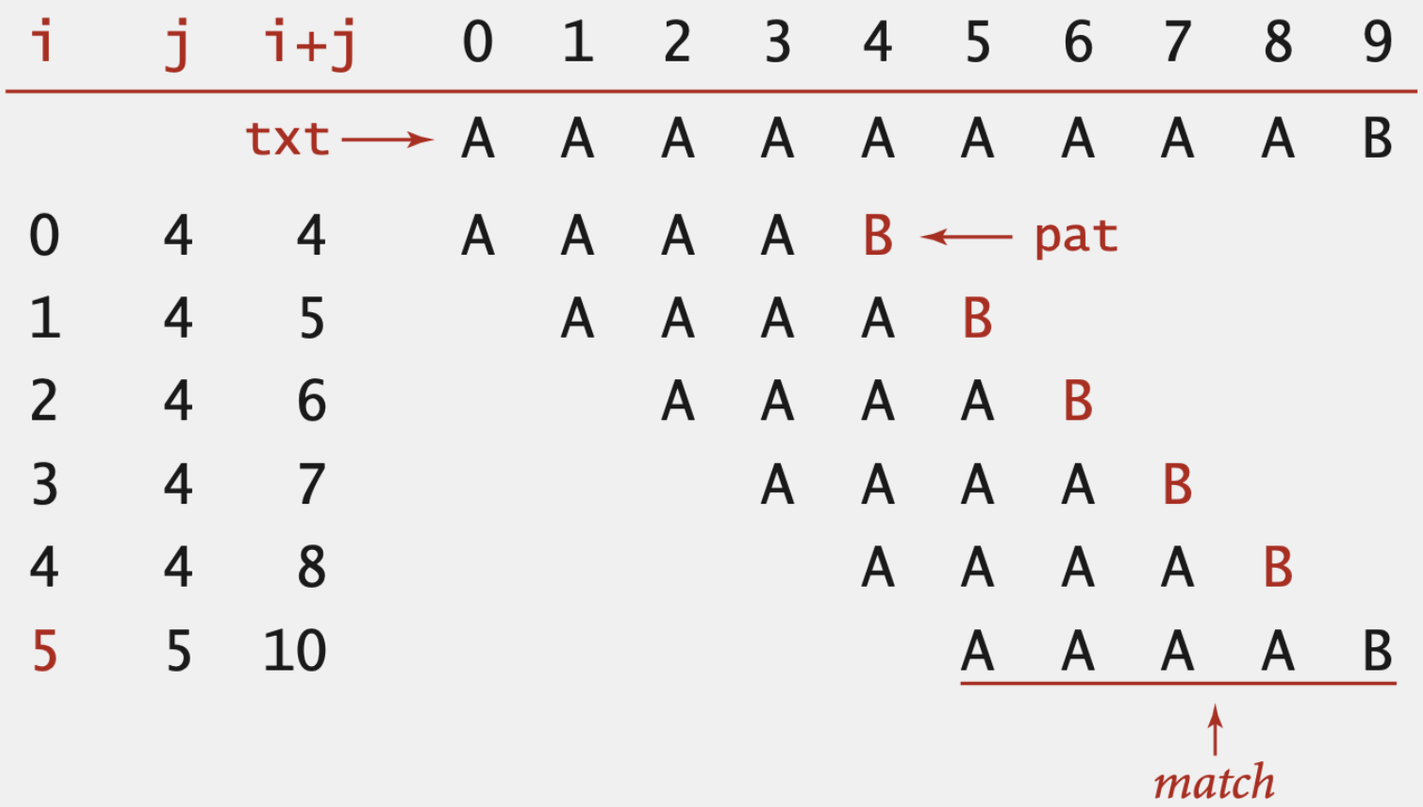
Brute-force substring search

Check for pattern starting at each text position.



Brute-force substring search: worst case

Brute-force algorithm can be slow if text and pattern are repetitive.



Worst case. $\sim M N$ char compares.

Boyer-Moore Algorithm

Boyer-Moore search

- Problem with brute force search → we keep considering too many bad options as well ~ maybe we can eliminate a lot of possibilities
- Thats why Boyer-Moore algorithm came to be
- Very efficient string search algorithm
- The algorithm needs to preprocess the pattern, but not the whole text !!!
- The algorithm runs faster as the length of the pattern increases
- The key features of the algorithm are to match on the tail of the pattern rather than the head
- Why is it good? We can skip multiple characters at the same time rather than searching every single character in the text

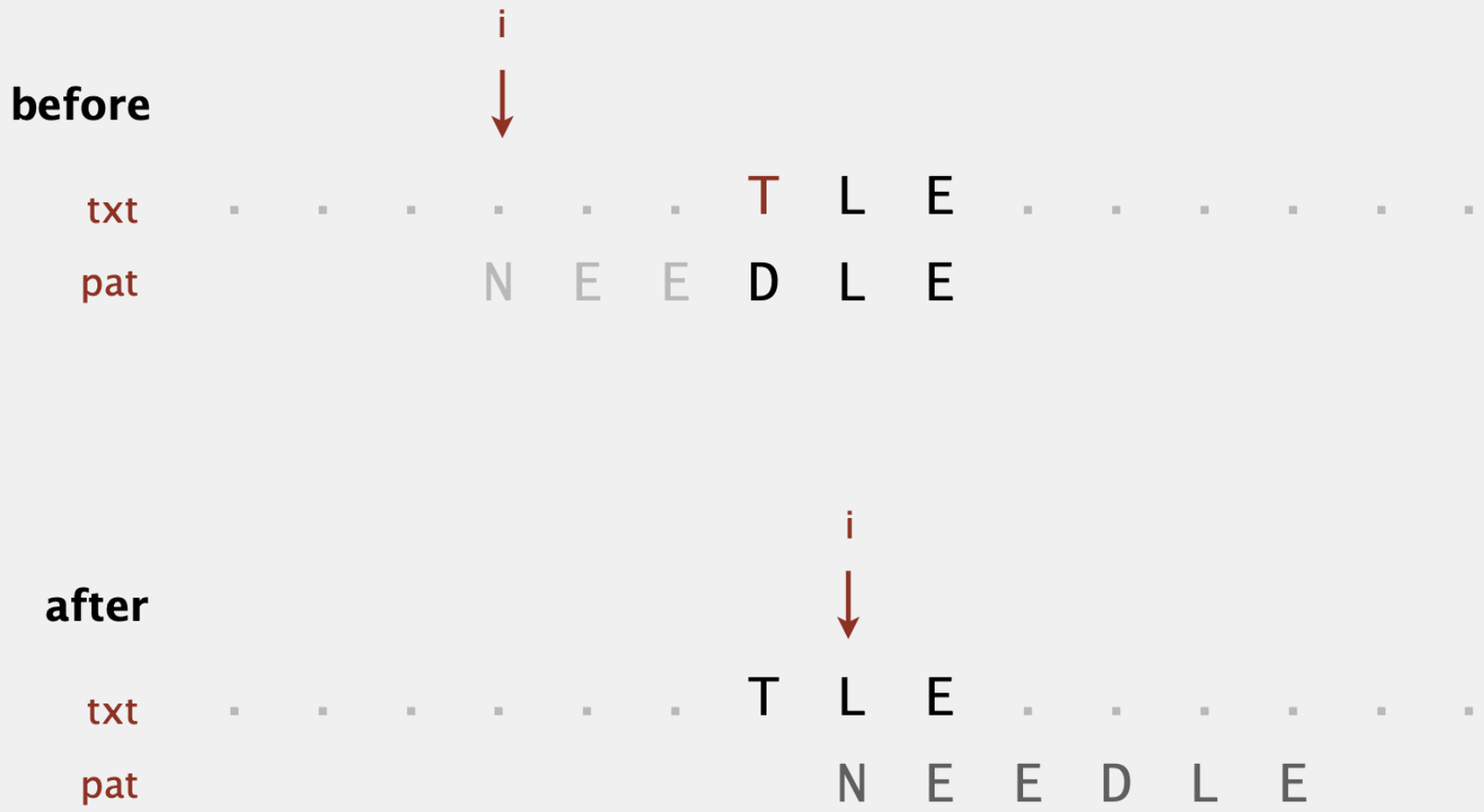
Boyer-Moore Algorithm

- Bad Symbol Shift
- Good Suffix Shift

Boyer-Moore: mismatched character heuristic

Q. How much to skip?

Case 1. Mismatch character not in pattern.

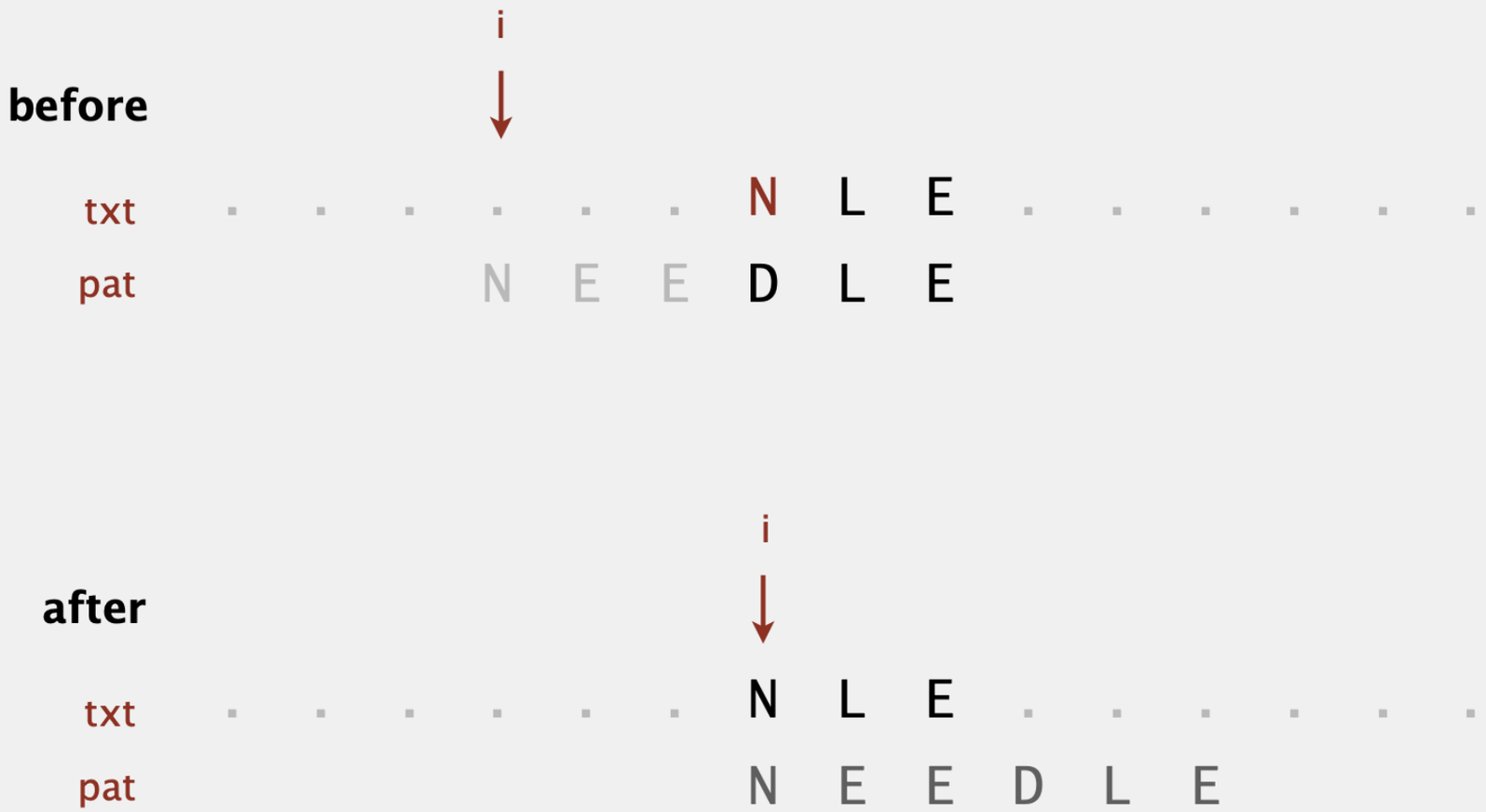


mismatch character 'T' not in pattern: increment i one character beyond 'T'

Boyer-Moore: mismatched character heuristic

Q. How much to skip?

Case 2a. Mismatch character in pattern.

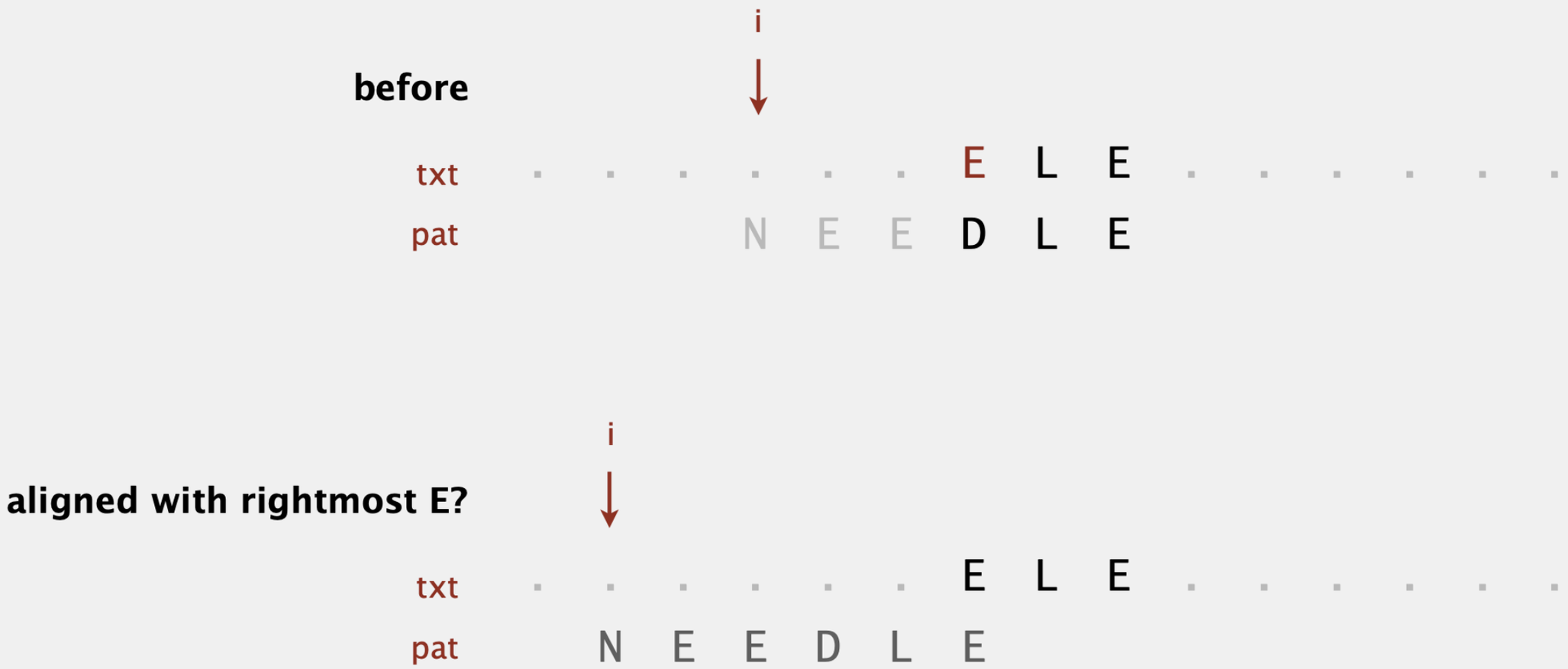


mismatch character 'N' in pattern: align text 'N' with rightmost pattern 'N'

Boyer-Moore: mismatched character heuristic

Q. How much to skip?

Case 2b. Mismatch character in pattern (but heuristic no help).

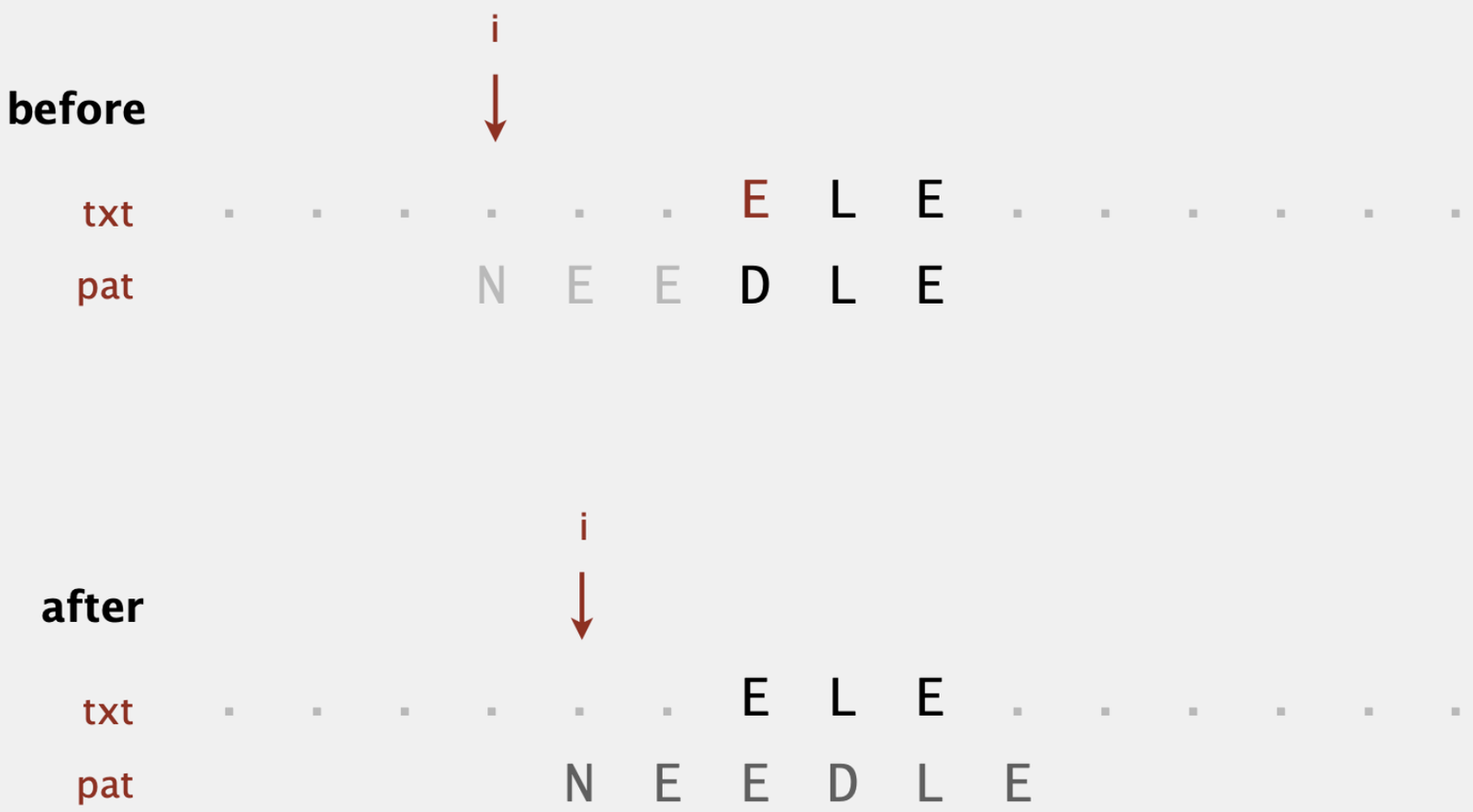


mismatch character 'E' in pattern: align text 'E' with rightmost pattern 'E' ?

Boyer-Moore: mismatched character heuristic

Q. How much to skip?

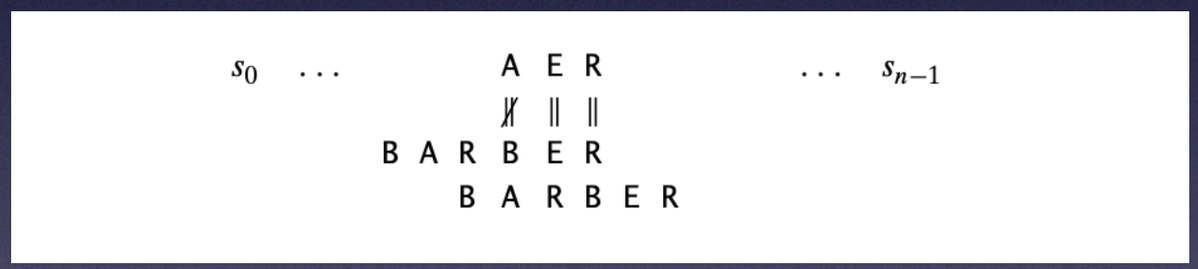
Case 2b. Mismatch character in pattern (but heuristic no help).



mismatch character 'E' in pattern: increment i by 1

Bad-Symbol Shift

$$t_1(\mathsf{S}) - 2 = 6 - 2 = 4$$

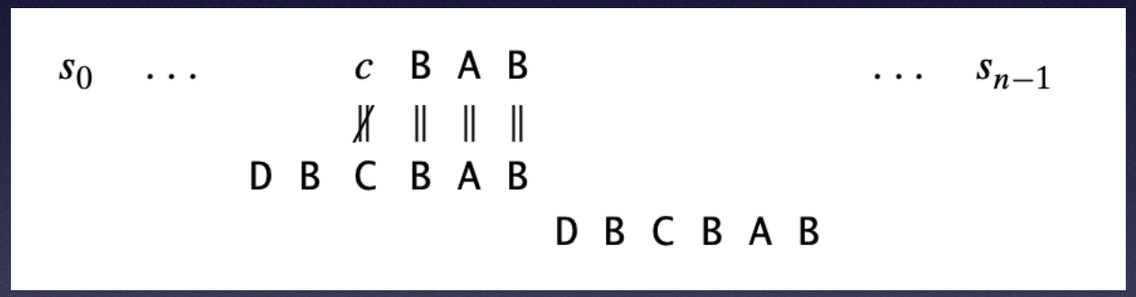


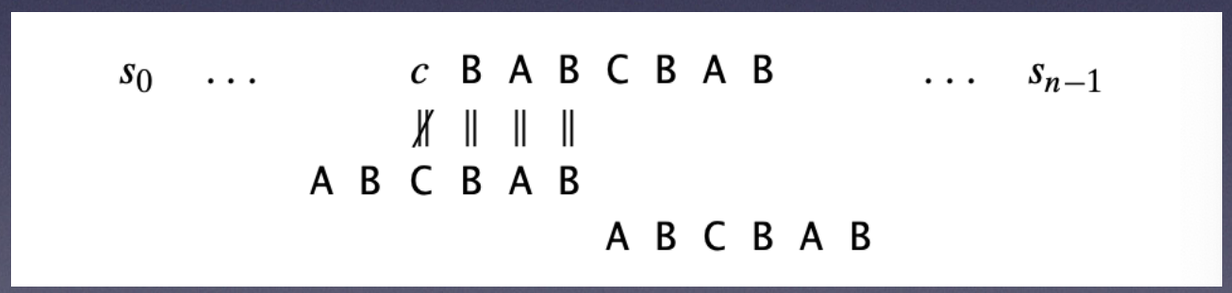
$$t_1(\mathsf{A}) - 2 = 4 - 2 = 2$$

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

Good Suffix Shift

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





k	**pattern**	d_2
1	ABC$\overline{\text{B}}$A$\underline{\text{B}}$	2
2	$\overline{\text{AB}}$CBA$\underline{\text{B}}$	4
3	$\overline{\text{AB}}$CB$\underline{\text{AB}}$	4
4	$\overline{\text{AB}}$C$\underline{\text{BAB}}$	4
5	$\overline{\text{AB}}$CBAB	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 earlier.

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

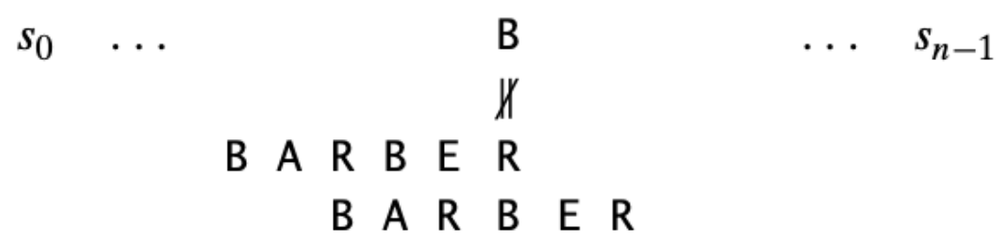
$$d = \begin{cases} d_1 & \text{if } k = 0, \\ \max\{d_1, d_2\} & \text{if } k > 0, \end{cases} \qquad \textbf{(7.3)}$$

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

Boyer-Moore Horspool Algorithm

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—e.g., c is letter R in our example—the situation is similar to that of Case 1 and the pattern should be shifted by the entire pattern's length m:

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—e.g., c is letter R in our example—the situation is similar to that of Case 2 and the rightmost occurrence of c among the first $m - 1$ characters in the pattern should be aligned with the text's c:

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

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
```

■

Horspool Algorithm

Preprocessing

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)
for $i \leftarrow 0$ **to** $size-1$ **do** $Table[i] \leftarrow m$
for $j \leftarrow 0$ **to** $m-2$ **do** $Table[P[j]] \leftarrow m-1-j$
return *Table*

Horspool 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
```

Rabin-Karp Algorithm

Rabin-Karp fingerprint search

Basic idea = modular hashing.

- Compute a hash of `pat[0..M-1]`.
- For each `i`, compute a hash of `txt[i..M+i-1]`.
- If pattern hash = text substring hash, check for a match.

```
      pat.charAt(i)
i     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
6  ← return i = 6       2  6  5  3  5   % 997 = 613
                                        match
```

modular hashing with R = 10 and hash(s) = s (mod 997)

Rabin-Karp Algorithm

Rabin-Karp

- The Rabin-Karp string searching algorithm calculates a **hash value** for the pattern, and for each M-character subsequence of text to be compared.
- If the hash values are unequal, the algorithm will calculate the hash value for next M-character sequence.
- If the hash values are equal, the algorithm will do a Brute Force comparison between the pattern and the M-character sequence.
- In this way, there is only one comparison per text subsequence, and Brute Force is only needed when hash values match.
- Perhaps a figure will clarify some things...

Modular arithmetic

Math trick. To keep numbers small, take intermediate results modulo Q.

Ex.

$$\begin{aligned}
(10000 + 535) * 1000 &\pmod{997} \\
= (30 + 535) * 3 &\pmod{997} \\
= 1695 &\pmod{997} \\
= 698 &\pmod{997}
\end{aligned}$$

$$(a + b) \bmod Q = ((a \bmod Q) + (b \bmod Q)) \bmod Q$$

$$(a * b) \bmod Q = ((a \bmod Q) * (b \bmod Q)) \bmod Q$$

two useful modular arithmetic identities

Efficiently computing the hash function

Modular hash function. Using the notation t_i for `txt.charAt(i)`, we wish to compute

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

Intuition. M-digit, base-R integer, modulo Q.

Horner's method. Linear-time method to evaluate degree-M polynomial.

	pat.charAt()				
i	0	1	2	3	4
	2	6	5	3	5

R Q

```
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
```

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

26535 = 2*10000 + 6*1000 + 5*100 + 3*10 + 5

= ((((2) *10 + 6) * 10 + 5) * 10 + 3) * 10 + 5

Efficiently computing the hash function

Challenge. How to efficiently compute x_{i+1} given that we know x_i.

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

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

Key property. Can update "rolling" hash function in constant time!

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

- x_i: current value
- $t_i R^{M-1}$: subtract leading digit
- R: multiply by radix
- t_{i+M}: add new trailing digit

(can precompute R^{M-1})

i	...	2	3	4	5	6	7	...	
current value	1	4	1	5	9	2	6	5	*text*
new value		4	1	5	9	2	6	5	

	4	1	5	9	2	*current value*
-	4	0	0	0	0	
		1	5	9	2	*subtract leading digit*
			*	1	0	*multiply by radix*
	1	5	9	2	0	
				+	6	*add new trailing digit*
	1	5	9	2	6	*new value*

Exact pattern match cost summary

Cost of searching for M-character pattern in N-character text

algorithm	typical	worst-case
brute-force	1.1 N char compares †	M N char compares
Karp-Rabin	3N arithmetic ops	3N arithmetic ops ‡
KMP	1.1 N char compares †	2N char compares
Boyer-Moore	~ N/M char compares †	3N char compares

† assumes appropriate model
‡ randomized

Substring search cost summary

Cost of searching for an M-character pattern in an N-character text.

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 hash function*

Performance Comparison of some String Matching Algorithms