5.3 Substring Search

- introduction
- brute force
- Knuth–Morris–Pratt
- Boyer–Moore
5.3 Substring Search

- introduction
- brute force
- Knuth–Morris–Pratt
- Boyer–Moore
Substring search

Goal. Find pattern of length $m$ in a text of length $n$. Typically $n \gg m$.

\begin{align*}
\text{pattern} & \rightarrow \text{NEEDE LE} \\
\text{text} & \rightarrow \text{INAHAYSTACK NEEDE LE INA}
\end{align*}
Substring search applications

**Goal.** Find pattern of length $m$ in a text of length $n$.

`pattern` → NEEDLE

`ttext` → INAHAYSTACK NEEDLE INA

`match` typically $n \gg m$

Search in a word processor or IDE.
Substring search applications

Goal. Find pattern of length $m$ in a text of length $n$. 

- \text{pattern} \rightarrow \text{NEEDLE}
- \text{text} \rightarrow \text{INA HAYSTACK NEEDLE INA}

Identify patterns indicative of spam.

- PROFITS
- LOSE WEIGHT
- herbal Viagra
- There is no catch.
- This is a one-time mailing.
- This message is sent in compliance with spam regulations.
Substring search applications

Web scraping. Extract relevant data from web page.

Ex. Find string delimited by `<b>` and `</b>` after first occurrence of pattern Last Trade:
Web scraping: Java implementation

Java library. The `indexOf()` method in Java’s `String` data type returns the index of the first occurrence of a given string, starting at a given offset.

```java
public class StockQuote {
    public static void main(String[] args) {
        String name = "http://finance.yahoo.com/q?s=";
        In in = new In(name + args[0]);
        String text = in.readAll();
        int start = text.indexOf("Last Trade:", 0);
        int from = text.indexOf("<b>", start);
        int to = text.indexOf("</b>", from);
        String price = text.substring(from + 3, to);
        StdOut.println(price);
    }
}
```

% java StockQuote goog
582.93

Caveat. Must update program whenever Yahoo format changes.
5.3 Substring Search

- introduction
- brute force
- Knuth–Morris–Pratt
- Boyer–Moore
Brute-force substring search

Check for pattern starting at each text position.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>txt</strong></td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>C</td>
<td>A</td>
<td>D</td>
<td>A</td>
<td>B</td>
<td>R</td>
<td>A</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>A</td>
<td>B</td>
<td>R</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>A</td>
<td>B</td>
<td>R</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>A</td>
<td>B</td>
<td>R</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>A</td>
<td>B</td>
<td>R</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>A</td>
<td>B</td>
<td>R</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>A</td>
<td>B</td>
<td>R</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>A</td>
<td>B</td>
<td>R</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>A</td>
<td>B</td>
<td>R</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>A</td>
<td>B</td>
<td>R</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>A</td>
<td>B</td>
<td>R</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>pat</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Entries in red are mismatches.
Entries in gray are for reference only.
Entries in black match the text.

Match
Brute-force substring search: Java implementation

Check for pattern starting at each text position.

```java
public static int search(String pat, String txt) {
    int m = pat.length();
    int n = txt.length();

    for (int i = 0; i <= n - m; i++) { // for each possible offset i
        for (int j = 0; j <= m; j++) { // check for match at offset i
            if (j == m) return i; // match at offset i
            if (pat.charAt(j) != txt.charAt(i+j)) { // no match at offset i
                break;
            }
        }
    }
    return n; // no match
}
```
What is the worst-case running time of brute-force substring search as a function of the pattern length $m$ and text length $n$?

A. $m + n$
B. $m^2$
C. $mn$
D. $n^2$
Algorithmic challenges in substring search

Fundamental algorithmic challenge. Linear-time guarantee.

Now is the time for all people to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for many good people to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for a lot of good people to come to the aid of their party. Now is the time for all of the good people to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for all good Republicans to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for many or all good people to come to the aid of their party. Now is the time for all good Democrats to come to the aid of their party. Now is the time for all people to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for all good Democrats to come to the aid of their party. Now is the time for all good Republicans to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for an attack at dawn party. Now is the time for each person to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for all good Republicans to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for many or all good people to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for all good Democrats to come to the aid of their party.
5.3 Substring Search

- introduction
- brute force
- Knuth–Morris–Pratt
- Boyer–Moore
Knuth–Morris–Pratt substring search

**Intuition.** Suppose we are searching in text for pattern \texttt{B A A A A A A A A A A A A A A A A A A.}

- Suppose we match 5 chars in pattern, with mismatch on 6\textsuperscript{th} char.
Knuth–Morris–Pratt substring search

**Intuition.** Suppose we are searching in text for pattern B A A A A A A A A A A A A A A A A.
- Suppose we match 5 chars in pattern, with mismatch on 6th char.
- We know previous 6 chars in text must be B A A A A A .
- Don’t need to compare any text character twice.

assuming \{A, B\} alphabet

Knuth–Morris–Pratt algorithm. Clever method to always avoid comparing a text character more than once!
Deterministic finite state automaton (DFA)

DFA is abstract string-searching machine.
- Finite number of states (including start and halt).
- Exactly one state transition for each character in alphabet.
- Accept if sequence of state transitions ever enters halt state.

**internal representation**

<table>
<thead>
<tr>
<th>j</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>pat.charAt(j)</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>C</td>
</tr>
<tr>
<td>dfa[i][j]</td>
<td>A</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

If in state j reading char C:
- if j is 6 halt and accept
- else move to state dfa[c][j]

**graphical representation**
Knuth–Morris–Pratt demo: DFA simulation

A A B A C A A B A B A C A A

pat.charAt(j) 0 1 2 3 4 5
A B A B A C
A 1 1 3 1 5 1
B 0 2 0 4 0 4
C 0 0 0 0 0 6

dfa[][]
Interpretation of Knuth–Morris–Pratt DFA

Q. What is interpretation of DFA state after processing \( \text{txt}[i] \)?
A. State = number of characters in pattern that have been matched.

Ex. DFA is in state 3 after reading in \( \text{txt}[0..6] \).

<table>
<thead>
<tr>
<th>( \text{txt}[] )</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{pat}[] )</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>prefix of ( \text{pat}[] )</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>C</td>
<td>A</td>
<td>A</td>
<td>C</td>
</tr>
<tr>
<td>suffix of ( \text{txt}[0..6] )</td>
<td>B</td>
<td>C</td>
<td>B</td>
<td>A</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>C</td>
<td>A</td>
</tr>
</tbody>
</table>

length of longest prefix of \( \text{pat}[] \) that is a suffix of \( \text{txt}[0..i] \)
Which state is the DFA in after processing the following input?

B A A B A B A B

A. 0
B. 1
C. 3
D. 4
Which state is the DFA in after processing the following input?


A. 0
B. 1
C. 3
D. 4
Knuth–Morris–Pratt substring search: Java implementation

Key differences from brute-force implementation.

- Need to precompute dfa[][] from pattern.
- Each text character compared (at most) once.

```java
public int search(String txt)
{
    int m = pat.length(), n = txt.length();
    int j = 0; // current state
    for (int i = 0; i < n; i++)
    {
        if (j == m) return i - m; // halt state (match found)
        j = dfa[txt.charAt(i)][j];
    }
    return n; // no match
}
```

Running time.

- Simulate DFA on text: at most $n$ character accesses.
- Build DFA: how to do efficiently? [warning: tricky algorithm ahead]
Knuth–Morris–Pratt demo: DFA construction

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

<table>
<thead>
<tr>
<th>pat.charAt(j)</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Diagram:

- Start state: 0
- Transitions:
  - A from 0 to 1
  - B from 0 to 2
  - C from 0 to 4
  - A from 1 to 2
  - B from 1 to 4
  - C from 2 to 3
  - A from 2 to 3
  - B from 2 to 5
  - C from 3 to 6
  - A from 3 to 4
  - B from 4 to 5
  - C from 5 to 6
- Accepting states: 3, 5, 6

 DFA[][]:

```
<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
```
How to build DFA from pattern?

Include one state for each character in pattern (plus accept state).

<table>
<thead>
<tr>
<th>DFA States</th>
<th>Pattern Concentration</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A</td>
</tr>
<tr>
<td>1</td>
<td>AB</td>
</tr>
<tr>
<td>2</td>
<td>ABA</td>
</tr>
<tr>
<td>3</td>
<td>ABAB</td>
</tr>
<tr>
<td>4</td>
<td>ABABA</td>
</tr>
<tr>
<td>5</td>
<td>ABABAC</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>pat.charAt(j)</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>C</td>
</tr>
<tr>
<td>B</td>
<td>A</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>A</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
How to build DFA from pattern?

**Match transition.** If in state \( j \) and next char \( c = \text{pat.charAt}(j) \), go to \( j+1 \).

- First \( j \) characters of pattern have already been matched
- Next char matches
- Now first \( j + 1 \) characters of pattern have been matched

<table>
<thead>
<tr>
<th>( \text{pat.charAt}(j) )</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( \text{dfa[]<a href="j"></a>} )</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

0 \rightarrow A \rightarrow 1 \rightarrow B \rightarrow 2 \rightarrow A \rightarrow 3 \rightarrow B \rightarrow 4 \rightarrow A \rightarrow 5 \rightarrow C \rightarrow 6

\( A, AB, ABA, ABAB, ABABA, ABABAC \)
How to build DFA from pattern?

Mismatch transition. If in state \( j \) and next char \( c \neq \text{pat}\.\text{charAt}(j) \), then the last \( j-1 \) characters of input are \( \text{pat}[1..j-1] \), followed by \( c \).

To compute \( \text{dfa}[c][j] \): Simulate \( \text{pat}[1..j-1] \) on DFA and take transition \( c \).

Running time. Seems to require \( j \) steps.

Ex. \( \text{dfa}['A'][5] = 1 \) \( \text{dfa}['B'][5] = 4 \)

\[
\begin{array}{c|ccccccc}
\text{pat}\.\text{charAt}(j) & 0 & 1 & 2 & 3 & 4 & 5 \\
\hline
A & B & A & B & A & C
\end{array}
\]

simulate BABAA
simulate BABAB

\[
\begin{array}{c}
0 & 1 & 2 & 3 & 4 & 5 \\
\hline
\end{array}
\]

simulation of BABA
How to build DFA from pattern?

Mismatch transition. If in state $j$ and next char $c \neq \text{pat} . \text{charAt}(j)$, then the last $j-1$ characters of input are $\text{pat}[1..j-1]$, followed by $c$.

To compute $\text{dfa}[c][j]$: Simulate $\text{pat}[1..j-1]$ on DFA and take transition $c$.

Running time. Takes only constant time if we maintain state $x$.

**Ex.** $\text{dfa['A'][5]} = 1$
- from state $x$,
- take transition 'A'
  \[ = \text{dfa['A'][x]} \]

$\text{dfa['B'][5]} = 4$
- from state $x$,
- take transition 'B'
  \[ = \text{dfa['B'][x]} \]

$x' = 0$
- from state $x$,
- take transition 'C'
  \[ = \text{dfa['C'][x]} \]

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>C</td>
<td></td>
</tr>
</tbody>
</table>

![DFA Diagram]

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>C</td>
<td></td>
</tr>
</tbody>
</table>
Knuth–Morris–Pratt demo: DFA construction in linear time

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

<table>
<thead>
<tr>
<th>pat.charAt(j)</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>C</td>
</tr>
<tr>
<td>A</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>6</td>
</tr>
</tbody>
</table>
Constructing the DFA for KMP substring search: Java implementation

For each state $j$:
- Copy $\text{dfa}[][x]$ to $\text{dfa}[][j]$ for mismatch case.
- Set $\text{dfa}[	ext{pat}.\text{charAt}(j)][j]$ to $j+1$ for match case.
- Update $x$.

```java
public KMP(String pat)
{
    this.pat = pat;
    m = pat.length();
    dfa = new int[R][m];
    dfa[pat.charAt(0)][0] = 1;
    for (int x = 0, j = 1; j < m; j++)
    {
        for (char c = 0; c < R; c++)
            dfa[c][j] = dfa[c][x];
        dfa[pat.charAt(j)][j] = j+1;
        x = dfa[pat.charAt(j)][x];
    }
}
```

Running time. $m$ character accesses (but space/time proportional to $R m$).
**KMP substring search analysis**

**Proposition.** KMP substring search accesses no more than $m + n$ chars to search for a pattern of length $m$ in a text of length $n$.

**Pf.** Each pattern character accessed once when constructing the DFA; each text character accessed (at most) once when simulating the DFA.

**Proposition.** KMP constructs $\text{dfa}[][]$ in time and space proportional to $Rm$.

**Larger alphabets.** Improved version of KMP constructs $\text{nfa}[]$ in time and space proportional to $m$.
Knuth–Morris–Pratt: brief history

- Independently discovered by two theoreticians and a hacker.
  - Knuth: inspired by esoteric theorem, discovered linear algorithm
  - Pratt: made running time independent of alphabet size
  - Morris: built a text editor for the CDC 6400 computer
- Theory meets practice.
A string $s$ is a cyclic rotation of $t$ if $s$ and $t$ have the same length and $s$ is a suffix of $t$ followed by a prefix of $t$.

### Problem.
Given two binary strings $s$ and $t$, design a linear-time algorithm to determine if $s$ is a cyclic rotation of $t$. 

<table>
<thead>
<tr>
<th>Rotated String</th>
<th>Yes</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s$</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>$t$</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>$s\text{rotated}$</td>
<td>$ABA\text{BABBABAABA}$</td>
<td>$GNIRTSDETA\text{TOR}$</td>
</tr>
</tbody>
</table>
5.3 Substring Search

- introduction
- brute force
- Knuth–Morris–Pratt
- Boyer–Moore
**Boyer–Moore: mismatched character heuristic**

**Intuition.**

- Scan characters in pattern from right to left.
- Can skip as many as \( m \) text chars when finding one not in the pattern.

```
<table>
<thead>
<tr>
<th>i</th>
<th>j</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>11</td>
<td>4</td>
</tr>
<tr>
<td>15</td>
<td>0</td>
</tr>
</tbody>
</table>
```

```
return i = 15
```
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.

before

\[
\begin{array}{cccccc}
\text{txt} & \cdot & \cdot & \cdot & \cdot & \cdot & N & L & E & \cdot & \cdot & \cdot & \cdot & \cdot \\
\text{pat} & & & & & & & & & & & & & N & E & E & D & L & E \\
\end{array}
\]

\[
\begin{array}{cccccc}
\text{txt} & \cdot & \cdot & \cdot & \cdot & \cdot & N & L & E & \cdot & \cdot & \cdot & \cdot & \cdot \\
\text{pat} & & & & & & & & & & & & & N & E & E & D & L & E \\
\end{array}
\]

after

mismatch character N in pattern: align text N with rightmost (why?) 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).

```
before     i

  txt . . . . . . . E L E . . . . . .
  pat   NEED L E

after     i

  txt . . . . . . . E L E . . . . . .
  pat   NEED L E
```

mismatch character E in pattern: increment i by 1
Which text character is compared next with pattern character E?

A. R (index 5)
B. O (index 6)
C. O (index 12)
D. O (index 13)
Substring search: quiz 6

Which text character is compared next with pattern character E?

A. O  
B. R  
C. E  
D. J

<table>
<thead>
<tr>
<th>text</th>
<th>B</th>
<th>O</th>
<th>O</th>
<th>Y</th>
<th>E</th>
<th>R</th>
<th>O</th>
<th>B</th>
<th>E</th>
<th>R</th>
<th>T</th>
<th>M</th>
<th>O</th>
<th>O</th>
<th>R</th>
<th>E</th>
<th>J</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>pattern</td>
<td>M</td>
<td>O</td>
<td>O</td>
<td>R</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>M</td>
<td>O</td>
<td>O</td>
<td>R</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>M</td>
<td>O</td>
<td>O</td>
<td>R</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Boyer–Moore: mismatched character heuristic

Q. How much to skip?

A. Precompute index of rightmost occurrence of character c in pattern. 
   (−1 if character not in pattern)

right = new int[R];
for (int c = 0; c < R; c++)
    right[c] = -1;
for (int j = 0; j < m; j++)
    right[pat.charAt(j)] = j;

<table>
<thead>
<tr>
<th>c</th>
<th>N</th>
<th>E</th>
<th>E</th>
<th>D</th>
<th>L</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>B</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>C</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>D</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>E</td>
<td>-1</td>
<td>-1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>M</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>N</td>
<td>-1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Boyer-Moore skip table computation
public int search(String txt) {
    int n = txt.length();
    int m = pat.length();
    int skip;
    for (int i = 0; i <= n-m; i += skip) {
        skip = 0;
        for (int j = m-1; j >= 0; j--) {
            if (pat.charAt(j) != txt.charAt(i+j)) {
                skip = Math.max(1, j - right[txt.charAt(i+j)]);
                break;
            }
        }
        if (skip == 0) return i;
    }
    return n;
}
**Boyer–Moore: analysis**

**Property.** Substring search with the Boyer–Moore mismatched character heuristic takes about $\sim n/m$ character compares to search for a pattern of length $m$ in a text of length $n$. sublinear!

**Worst-case.** Can be as bad as $\sim m n$.

<table>
<thead>
<tr>
<th>i</th>
<th>skip</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>txt</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td></td>
</tr>
</tbody>
</table>

0 0 A B B B B pat
1 1 A B B B B
2 1 A B B B B
3 1 A B B B B
4 1 A B B B B
5 1 A B B B B

**Boyer–Moore variant.** Can improve worst case to $\sim 3 n$ character compares by adding a KMP-like rule to guard against repetitive patterns.
Which substring search algorithm does Java’s indexOf() method use?

A. Brute-force search
B. Knuth–Morris–Pratt
C. Boyer–Moore
D. Rabin–Karp