Acknowledgement: The course slides are adapted from the slides prepared by R. Sedgewick and K. Wayne of Princeton University.
Substring search
Brute force
Knuth-Morris-Pratt
Boyer-Moore
Rabin-Karp
Goal. Find pattern of length $M$ in a text of length $N$.

Typically $N \gg M$
Substring search applications

**Goal.** Find pattern of length $M$ in a text of length $N$.

Typically $N \gg M$

**Computer forensics.** Search memory or disk for signatures, e.g., all URLs or RSA keys that the user has entered.

[Links to http://citp.princeton.edu/memory]
Substring search applications

Goal. Find pattern of length $M$ in a text of length $N$.

typically $N >> M$

Identify patterns indicative of spam.

- PROFITS
- L0SE WE1GHT
- There is no catch.
- This is a one-time mailing.
- This message is sent in compliance with spam regulations.
Substring search applications

Electronic surveillance.

Need to monitor all internet traffic. (security)

No way! (privacy)

Well, we’re mainly interested in “ATTACK AT DAWN”

OK. Build a machine that just looks for that.

“ATTACK AT DAWN” substring search machine found
Substring search applications

**Screen scraping.** Extract relevant data from web page.

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

```
http://finance.yahoo.com/q?s=goog
```

...<tr>
<td class="yfnc_tablehead1" width="48%">
**Last Trade:**
</td>  
<td class="yfnc_tabledata1">
<b>452.92</b>
</td></tr>
```

**Trade Time:**
```
http://finance.yahoo.com/q?s=goog
```
Screen scraping: Java implementation

Java library. The `indexOf()` method in Java's string library 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

% java StockQuote msft
24.84
Substring Search

- Brute force
- Knuth-Morris-Pratt
- Boyer-Moore
- Rabin-Karp
### Brute-force substring search

Check for pattern starting at each text position.

<table>
<thead>
<tr>
<th>i</th>
<th>j</th>
<th>i+j</th>
<th>txt</th>
<th>pat</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>2</td>
<td>A</td>
<td>BRA</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>A</td>
<td>BRA</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>3</td>
<td>A</td>
<td>BRA</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>3</td>
<td>A</td>
<td>BRA</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>5</td>
<td>A</td>
<td>BRA</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>5</td>
<td>A</td>
<td>BRA</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>10</td>
<td>A</td>
<td>BRA</td>
</tr>
</tbody>
</table>

Entries in red are mismatches.

Entries in gray are for reference only.

Entries in black match the text.

Return i when j is M.

Match.
Brute-force substring search: Java implementation

Check for pattern starting at each text position.

<table>
<thead>
<tr>
<th>i</th>
<th>j</th>
<th>i+j</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>A</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>D</td>
<td>A</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>A</td>
<td>D</td>
<td>A</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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; // index in text where pattern starts
    }
    return N; // not found
Brute-force substring search: worst case

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

<table>
<thead>
<tr>
<th>i</th>
<th>j</th>
<th>i+j</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>10</td>
</tr>
</tbody>
</table>

```
txt  → A A A A A A A A A B
pat  ← A A A A A A A A A B
```

Worst case. \( \sim MN \) char compares.
In many applications, we want to avoid backup in text stream.

- Treat input as stream of data.
- Abstract model: standard input.

**Brute-force algorithm needs backup for every mismatch.**

**Approach 1.** Maintain buffer of last $M$ characters.

**Approach 2.** Stay tuned.
Brute-force substring search: alternate implementation

Same sequence of char compares as previous implementation.
• i points to end of sequence of already-matched chars in text.
• j stores number of already-matched chars (end of sequence in pattern).

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

```java
public static int search(String pat, String txt) {
    int i, N = txt.length();
    int j, M = pat.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;
    else            return N;
}
```
Algorithmic challenges in substring search

Brute-force is not always good enough.

Theoretical challenge. Linear-time guarantee. ← fundamental algorithmic problem

Practical challenge. Avoid backup in text stream. ← often no room or time to save text

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 each good person 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 Democrats 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 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 Republicans to come to the aid of their party. Now is the time for all good Democrats to come to the aid of their party.
Substring Search

- Brute force
- Knuth-Morris-Pratt
- Boyer-Moore
- Rabin-Karp
**Intuition.** Suppose we are searching in text for pattern `BAAAAAAAAA`.

- Suppose we match 5 chars in pattern, with mismatch on 6th char.
- We know previous 6 chars in text are `BAAAAB`.
- Don't need to back up text pointer!

**Knuth-Morris-Pratt algorithm.** Clever method to always avoid backup. (!)
DFA is abstract string-searching machine.

- Finite number of states (including start and halt).
- Exactly one transition for each char in alphabet.
- Accept if sequence of transitions leads to halt state.

### Internal Representation

<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>dfa[j][c]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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>

If in state \( j \) reading char \( c \):
- If \( j \) is 6 halt and accept
- Else move to state \( \text{dfa}[c][j] \)

### Graphical Representation

```
0 --- B,C
    ^
    | 1
    | A
    v
  2 --- A
      |     A
      v     v
  3 --- B
      |     C
      v     v
  4 --- A
      |     B
      v     v
  5 --- C
      |     B,C
      v     v
  6 --- A
       |     C
       v     v
       |     B,C
       v     v
       |     A
```
DFA simulation

\[
\text{pat.charAt(j)} = \begin{array}{cccccc}
0 & 1 & 2 & 3 & 4 & 5 \\
A & B & A & B & A & C \\
A & 1 & 1 & 3 & 1 & 5 & 1 \\
B & 0 & 2 & 0 & 4 & 0 & 4 \\
C & 0 & 0 & 0 & 0 & 0 & 6 \\
\end{array}
\]

\[
\text{dfa[][]} = \begin{array}{ccc}
0 & 1 & 2 \\
1 & 3 & 1 \\
2 & 4 & 0 \\
3 & 5 & 4 \\
4 & 6 & 0 \\
5 & 0 & 0 \\
6 & 1 & 2 \\
\end{array}
\]
DFA simulation

\[
\begin{array}{cccccc}
0 & 1 & 2 & 3 & 4 & 5 \\
A & B & A & B & A & C \\
A & 1 & 1 & 3 & 1 & 5 & 1 \\
B & 0 & 2 & 0 & 4 & 0 & 4 \\
C & 0 & 0 & 0 & 0 & 0 & 6 \\
\end{array}
\]

\[
\begin{array}{ccc}
1 & 2 & 3 \\
A & A & B \\
A & B & A \\
B & A & B \\
\end{array}
\]

\[
\begin{array}{ccc}
4 & 5 & 6 \\
B & A & C \\
C & B & A \\
B, C & B, C & B, C \\
\end{array}
\]
DFA simulation

\[
\begin{array}{ccccccc}
0 & 1 & 2 & 3 & 4 & 5 \\
A & B & A & B & A & C \\
A & 1 & 1 & 3 & 1 & 5 & 1 \\
B & 0 & 2 & 0 & 4 & 0 & 4 \\
C & 0 & 0 & 0 & 0 & 0 & 6 \\
\end{array}
\]
DFA simulation

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

B, C

A

B, C

C

B, C

A

B

A

C

B, C

B, C

A

B

A

C

B, C

B, C

A

B

A

C

B, C
DFA simulation

\[ \text{pat.charAt}(j) \]

<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>A</td>
<td>B</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>C</td>
</tr>
<tr>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</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>
<tr>
<td>dfa[][]</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>6</td>
</tr>
</tbody>
</table>

A → A → B → A → C → A → A

\[ \text{dfa[][]} \]
DFA simulation

```
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 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 | 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

B, C

B, C

B, C

pat.charAt(j) for DFA simulation.
DFA simulation

\[\text{pat}.\text{charAt}(j)\]

\[
\begin{array}{c|ccccc}
0 & 1 & 2 & 3 & 4 & 5 \\
\hline
A & B & A & B & A & C \\
A & 1 & 1 & 3 & 1 & 5 \\
B & 0 & 2 & 0 & 4 & 0 \\
C & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\]
DFA simulation

\[
\begin{array}{ccccccc}
0 & 1 & 2 & 3 & 4 & 5 \\
A & B & A & B & A & C \\
A & 1 & 1 & 3 & 1 & 5 & 1 \\
B & 0 & 2 & 0 & 4 & 0 & 4 \\
C & 0 & 0 & 0 & 0 & 0 & 6 \\
\end{array}
\]

\[
\text{pat.charAt(j)}
\]

\[
\text{dfa[][][j]}
\]
DFA simulation

<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 B A B A C A</td>
<td>A</td>
<td>B</td>
<td>A B A C A</td>
<td>A</td>
<td>B</td>
<td>A C</td>
</tr>
<tr>
<td>A A 1 3 1 5 1</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>B 0 0 0 0 0 6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>6</td>
</tr>
</tbody>
</table>

A A B A C A A B A B A C A A
DFA simulation

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

<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>1</td>
<td>3</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>dfa[][][]j</td>
<td>B</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>0</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>

A

0 1 2 3 4 5

0

0

0

0

0

0

0

0

0

0

0

0

0

0

0

0

0

0

0

0
DFA simulation

```
<table>
<thead>
<tr>
<th>dfa[][][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>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>
```

```
pat.charAt(j)  | A  | B  | A  | B  | A  | C  |
```

```
A A B A C A A B A B A C A A

0 1 2 3 4 5

B, C A A B, C A B, C A

0 1 2 3 4 5

pat.charAt(j)

A  B  A  B  A  C
```

```
A B A B A C A A

0 1 2 3 4 5

B, C A A B, C A B, C A

0 1 2 3 4 5

pat.charAt(j)

A  B  A  B  A  C
```
DFA simulation

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

<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>

 DFA simulation diagram

- States: 0, 1, 2, 3, 4, 5, 6
- Transitions:
  - From state 0:
    - On 'A': to state 1
    - On 'B': to state 2
    - On 'C': to state 3
  - From state 1:
    - On 'A': to state 1
    - On 'B': to state 2
    - On 'C': to state 3
  - From state 2:
    - On 'A': to state 3
    - On 'B': to state 4
    - On 'C': to state 3
  - From state 3:
    - On 'A': to state 3
    - On 'B': to state 4
    - On 'C': to state 5
  - From state 4:
    - On 'A': to state 5
    - On 'B': to state 6
    - On 'C': to state 5
  - From state 5:
    - On 'C': to state 6
- Initial state: 0
- Accepting states: 5, 6
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 | A B A B A C
B | 1 1 3 1 5 1
C | 0 2 0 4 0 4

da[][][j] | 0 0 0 0 0 0 6
A

B, C

A

B, C

C

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

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

![Diagram of DFA]

- **length of longest prefix of pat[]**
  that is a suffix of $\text{txt}[0..i]$

- **Suffix of text[0..6]**

- **Prefix of pat[]**
Knuth-Morris-Pratt substring search: Java implementation

Key differences from brute-force implementation.

- Need to precompute $dfa[]$ from pattern.
- Text pointer $i$ never decrements.

```
public int search(String 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;
    else return N;
}
```

Running time.

- Simulate DFA on text: at most $N$ character accesses.
- Build DFA: how to do efficiently? [warning: tricky algorithm ahead]
Knuth-Morris-Pratt substring search: Java implementation

Key differences from brute-force implementation.
• Need to precompute $dfa[][]$ from pattern.
• Text pointer $i$ never decrements.
• Could use input stream.

```java
public int search(In in)
{
    int i, j;
    for (i = 0, j = 0; !in.isEmpty() && j < M; i++)
        j = dfa[in.readChar()][j];
    if (j == M) return i - M;
    else return NOT_FOUND;
}
```
Knuth-Morris-Pratt construction

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

Constructing the DFA for KMP substring search for \texttt{A B A B A C}
**Knuth-Morris-Pratt construction**

**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></td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>C</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>1</td>
<td>3</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td>2</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Constructing the DFA for KMP substring search for $A B A B A C$
Mismatch transition: back up if \( c \neq \text{pat.charAt}(j) \).

Constructing the DFA for KMP substring search for A B A B A C
Mismatch transition: back up if \( c \neq \text{pat.charAt}(j) \).

Constructing the DFA for KMP substring search for A B A B A C
Knuth-Morris-Pratt construction

Mismatch transition: back up if \( c \neq \text{pat.charAt}(j) \).

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

<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>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[][]</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

B: 0 2 0 4
C: 0 0 0 6
**Mismatch transition:** back up if $c \neq \text{pat.charAt}(j)$.

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

<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></td>
<td></td>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>5</td>
<td></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>

\[
\begin{array}{cccccc}
0 & 1 & 2 & 3 & 4 & 5 \\
A & B & A & B & A & C \\
A & 1 & 1 & 3 & 1 & 5 \\
B & 0 & 2 & 0 & 4 &   \\
C & 0 & 0 & 0 & 0 & 6 \\
\end{array}
\]
Mismatch transition: back up if \( c \neq \text{pat.charAt}(j) \).

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

<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>A</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>B</td>
<td>A</td>
</tr>
<tr>
<td>A</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>( \text{dfa[][]}[j] )</td>
<td>B</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>0</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>

\( j \)
Knuth-Morris-Pratt construction

Mismatch transition: back up if $c \neq \text{pat}.\text{charAt}(j)$.

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></td>
<td></td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>C</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>dfa[][][j]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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>

Graph: (Diagram of DFA with states 0 to 6, transitions for 'A', 'B', and 'C' with labels 'B, C' for unreachable states.)
Knuth-Morris-Pratt construction

<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><code>pat.charAt(j)</code></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 for: A B A B A C
How to build DFA from pattern?

Include one state for each character in pattern (plus accept state).
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>B</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>2</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
\begin{array}{c|cccccc}
\text{dfa[]}[j] & 0 & 1 & 2 & 3 & 4 & 5 \\
\hline
A & B & A & B & A & C \\
B & 1 & 3 & 5 \\
C & 2 & 4 \\
\end{array}
\]

\[
\begin{array}{c|ccccccc}
& 0 & 1 & 2 & 3 & 4 & 5 & 6 \\
\hline
0 & A & 1 & B & 2 & A & 3 & B & 4 & A & 5 & C & 6
\end{array}
\]
How to build DFA from pattern?

Mismatch transition. If in state $j$ and next char $c \neq \text{pat.charAt}(j)$, 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. $\text{dfa}[A][5] = 1$; $\text{dfa}[B][5] = 4$

simulate BABA; take transition 'A'
= $\text{dfa}'A'|[3]$

simulate BABA; take transition 'B'
= $\text{dfa}'B'|[3]$
Mismatch transition. If in state $j$ and next char $c \neq \text{pat.charAt}(j)$, 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' = $\text{dfa}['C'][X]$

\[
\begin{align*}
0 & \quad 1 & \quad 2 & \quad 3 & \quad 4 & \quad 5 \\
\hline
A & B & A & B & A & C
\end{align*}
\]
Knuth-Morris-Pratt construction (in linear time)

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

Constructing the DFA for KMP substring search for  A B A B A C
**Knuth-Morris-Pratt construction (in linear time)**

**Match transition.** For each state $j$, $\text{dfa}[\text{pat.charAt}(j)][j] = j+1$.

- First $j$ characters of pattern have already been matched
- 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></td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>C</td>
</tr>
<tr>
<td>$\text{dfa}[][][j]$</td>
<td></td>
<td>1</td>
<td>3</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td>2</td>
<td>4</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>
```

Constructing the DFA for KMP substring search for $A B A B A C$
Mismatch transition. For state 0 and char $c \neq \text{pat.charAt}(j)$, set $\text{dfa}[c][0] = 0$.

Constructing the DFA for KMP substring search for $A B A B A C$
Mismatch transition. For each state $j$ and character $c \neq \text{pat.charAt}(j)$, set $\text{dfa}[c][j] = \text{dfa}[c][X]$; then update $X = \text{dfa[pat.charAt(j)]}[X]$.

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

$X = \text{simulation of empty string}$

\[
\begin{array}{cccccc}
0 & 1 & 2 & 3 & 4 & 5 \\
\text{pat.charAt}(j) & A & B & A & B & A & C \\
A & 1 & 3 & 5 \\
B & 0 & 2 & 4 \\
C & 0 & & & & & 6 \\
\end{array}
\]
Mismatch transition. For each state \( j \) and char \( c \neq \text{pat.charAt}(j) \), set \( \text{dfa}[c][j] = \text{dfa}[c][X] \); then update \( X = \text{dfa[pat.charAt(j)][X]} \).
Mismatch transition. For each state $j$ and char $c \neq \text{pat.charAt}(j)$, set $\text{dfa}[c][j] = \text{dfa}[c][X]$; then update $X = \text{dfa}[	ext{pat.charAt}(j)][X]$.

$\text{X} = \text{simulation of B A}$

<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></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></td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

Constructing the DFA for KMP substring search for A B A B A C
Mismatch transition. For each state $j$ and char $c \neq \text{pat.charAt}(j)$, set 
$dfa[c][j] = dfa[c][X]$; then update $X = dfa[\text{pat.charAt}(j)][X]$.

Knuth-Morris-Pratt construction (in linear time)

Constructing the DFA for KMP substring search for A B A B A C
Mismatch transition. For each state $j$ and char $c \neq \text{pat.charAt}(j)$, set $\text{dfa}[c][j] = \text{dfa}[c][X]$; then update $X = \text{dfa[pat.charAt(j)]}[X]$.

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

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

X = simulation of B A B A

$$
\begin{array}{cccccc}
0 & 1 & 2 & 3 & 4 & 5 \\
\hline
0 & 0 & 0 & 0 & 0 & 0 \\
A & 0 & 2 & 0 & 4 & 0 \\
B & 1 & 1 & 3 & 1 & 5 \\
C & A & B & A & B & A \\
\end{array}
$$
Mismatch transition. For each state $j$ and char $c \neq \text{pat.charAt}(j)$, set $\text{dfa}[c][j] = \text{dfa}[c][X]$; then update $X = \text{dfa}[	ext{pat.charAt}(j)][X]$.

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

$X = \text{simulation of B A B A C}$

<table>
<thead>
<tr>
<th>$\text{pat.charAt}(j)$</th>
<th>A</th>
<th>B</th>
<th>A</th>
<th>B</th>
<th>A</th>
<th>C</th>
</tr>
</thead>
<tbody>
<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>A</td>
<td></td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>dfa[][][j]</td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>6</td>
</tr>
</tbody>
</table>

\[ \text{dfa}[c][X] = \text{dfa}[c][X] \]
Knuth-Morris-Pratt 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>B</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>dfa[][][j]</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>6</td>
</tr>
</tbody>
</table>

Diagram of DFA:

- States: 0, 1, 2, 3, 4, 5, 6
- Edges:
  - 0 → 1 (A)
  - 0 → 1 (B, C)
  - 1 → 2 (B)
  - 1 → 3 (A)
  - 1 → 1 (A)
  - 2 → 3 (A)
  - 2 → 4 (B)
  - 2 → 2 (B, C)
  - 3 → 4 (B)
  - 3 → 5 (A)
  - 4 → 5 (A)
  - 5 → 6 (C)
  - 5 → 5 (B, C)
  - 6 → 6 (A)
  - 6 → 6 (B, C)

A, B, C transitions
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 $\text{dfa[pat.charAt(j)]}[j]$ 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 (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];
    }
}
```

Running time. $M$ character accesses (but space proportional to $RM$).
**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 char accessed once when constructing the DFA; each text char accessed once (in the worst case) when simulating the DFA.

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

**Larger alphabets.** Improved version of KMP constructs $\text{nfa}[\text{}]$ 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-time algorithm
  - Pratt: made running time independent of alphabet size
  - Morris: built a text editor for the CDC 6400 computer

- Theory meets practice.

---

SIAM J. COMPUT.
Vol. 6, No. 2, June 1977

**FAST PATTERN MATCHING IN STRINGS**

DONALD E. KNUTH†, JAMES H. MORRIS, JR.‡ AND VAUGHAN R. PRATT¶

Abstract. An algorithm is presented which finds all occurrences of one given string within another, in running time proportional to the sum of the lengths of the strings. The constant of proportionality is low enough to make this algorithm of practical use, and the procedure can also be extended to deal with some more general pattern-matching problems. A theoretical application of the algorithm shows that the set of concatenations of even palindromes, i.e., the language \(\{\alpha^n\}\)\(^*\), can be recognized in linear time. Other algorithms which run even faster on the average are also considered.

---

Don Knuth  
Jim Morris  
Vaughan Pratt
Substring Search

- Brute force
- Knuth-Morris-Pratt
- Boyer-Moore
- Rabin-Karp
Intuition.
- Scan characters in pattern from right to left.
- Can skip as many as $M$ text chars when finding one not in the pattern.

```
i  j  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
  text → FINDIINAHAYSTACKNEELDEINAN
0   5  NEEDLE ← pattern
  5   5  NEEDLE
 11   4  NEEDLE
 15   0  NEEDLE
```

return $i = 15$
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'
**Case 2b.** Mismatch character in pattern (but heuristic no help).

---

*Question:* How much to skip?

---

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
**Q.** How much to skip?

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

```java
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></th>
<th>N</th>
<th>E</th>
<th>E</th>
<th>D</th>
<th>L</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<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></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>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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></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>
</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;
}
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 $N$.

Worst-case. Can be as bad as $\sim MN$.

Boyer-Moore variant. Can improve worst case to $\sim 3N$ by adding a KMP-like rule to guard against repetitive patterns.
Substring Search

- Brute force
- Knuth-Morris-Pratt
- Boyer-Moore
- Rabin-Karp
Rabin-Karp fingerprint search

Basic idea = modular hashing.

• Compute a hash of pattern characters 0 to $M - 1$.
• For each $i$, compute a hash of text characters $i$ to $M + i - 1$.
• If pattern hash = text substring hash, check for a match.

<table>
<thead>
<tr>
<th>pat.charAt(i)</th>
<th>i</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>i</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>6</td>
<td>5</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>% 997 = 613</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

|               | i | 2 | 6 | 5 | 3 | 5 | % 997 = 613 |
|---------------|---|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|
| match         | i | 6 | 2 | 6 | 5 | 3 | 5 | % 997 = 613 |
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.

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

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

<table>
<thead>
<tr>
<th>i</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>\ldots</th>
</tr>
</thead>
<tbody>
<tr>
<td>current value</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>5</td>
<td>9</td>
<td>2</td>
<td>\ldots</td>
</tr>
<tr>
<td>new value</td>
<td>4</td>
<td>1</td>
<td>5</td>
<td>9</td>
<td>2</td>
<td>6</td>
<td>5</td>
</tr>
</tbody>
</table>

- current value
- subtract leading digit
- multiply by radix
- add new trailing digit (can precompute $R^{M-2}$)

```
  \begin{array}{l}
  \text{current value} \\
  \text{new value} \\
  \hline
  \text{current value} \\
  4 \quad 1 \quad 5 \quad 9 \quad 2 \\
  \text{subtract leading digit} \\
  - 4 \quad 0 \quad 0 \quad 0 \quad 0 \\
  \text{multiply by radix} \\
  1 \quad 5 \quad 9 \quad 2 \quad 0 \\
  \text{add new trailing digit} \\
  \times 1 \quad 0 \\
  1 \quad 5 \quad 9 \quad 2 \quad 0 \\
  \text{new value} \\
  + 6 \\
  1 \quad 5 \quad 9 \quad 2 \quad 6
  \end{array}
```
Rabin-Karp substring search example

```
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  % 997 = 3       Q
1  3  1  % 997 = (3*10 + 1) % 997 = 31
2  3  1  4  % 997 = (31*10 + 4) % 997 = 314
3  3  1  4  1  % 997 = (314*10 + 1) % 997 = 150
4  3  1  4  1  5  % 997 = (150*10 + 5) % 997 = 508
5  1  4  1  5  9  % 997 = ((508 + 3*(997 - 30))*10 + 9) % 997 = 201
6  4  1  5  9  2  % 997 = (((201 + 1*(997 - 30)))*10 + 2) % 997 = 715
7  1  5  9  2  6  % 997 = (((715 + 4*(997 - 30)))*10 + 6) % 997 = 971
8  5  9  2  6  5  % 997 = (((971 + 1*(997 - 30)))*10 + 5) % 997 = 442
9  9  2  6  5  3  % 997 = (((442 + 5*(997 - 30)))*10 + 3) % 997 = 929
10 return i - M + 1 = 6  2  6  5  3  5  % 997 = (((929 + 9*(997 - 30)))*10 + 5) % 997 = 613
```
public class RabinKarp
{
    private long patHash;       // pattern hash value
    private int M;              // pattern length
    private long Q;             // modulus
    private int R;              // radix
    private long RM;            // R^(M-1) % Q

    public RabinKarp(String pat) {
        M = pat.length();
        R = 256;
        Q = longRandomPrime();

        RM = 1;
        for (int i = 1; i <= M-1; i++)
            RM = (R * RM) % Q;
        patHash = hash(pat, M);
    }

    public int search(String txt)
    {
        /* see next slide */
    }

    private long hash(String key, int M)
    {
        /* as before */
    }

    /* see next slide */
}
Monte Carlo version. Return match if hash match.

```
public int search(String txt)
{
    int N = txt.length();
    int txtHash = hash(txt, M);
    if (patHash == txtHash) return 0;
    for (int i = M; i < N; i++)
    {
        txtHash = (txtHash + Q - RM*txt.charAt(i-M) % Q) % Q;
        txtHash = (txtHash*R + txt.charAt(i)) % Q;
        if (patHash == txtHash) return i - M + 1;
    }
    return N;
}
```

Las Vegas version. Check for substring match if hash match; continue search if false collision.
Rabin-Karp analysis

**Theory.** If $Q$ is a sufficiently large random prime (about $MN^2$), then the probability of a false collision is about $1/N$.

**Practice.** Choose $Q$ to be a large prime (but not so large as to cause overflow). Under reasonable assumptions, probability of a collision is about $1/Q$.

**Monte Carlo version.**
- Always runs in linear time.
- Extremely likely to return correct answer (but not always!).

**Las Vegas version.**
- Always returns correct answer.
- Extremely likely to run in linear time (but worst case is $MN$).
Rabin-Karp fingerprint search

Advantages.
• Extends to 2d patterns.
• Extends to finding multiple patterns.

Disadvantages.
• Arithmetic ops slower than char compares.
• Las Vegas version requires backup.
• Poor worst-case guarantee.
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.

<table>
<thead>
<tr>
<th>algorithm</th>
<th>version</th>
<th>operation count</th>
<th>backup in input?</th>
<th>correct?</th>
<th>extra space</th>
</tr>
</thead>
<tbody>
<tr>
<td>brute force</td>
<td>—</td>
<td>$MN$</td>
<td>yes</td>
<td>yes</td>
<td>1</td>
</tr>
<tr>
<td>Knuth-Morris-Pratt</td>
<td>full DFA (Algorithm 5.6)</td>
<td>$2N$</td>
<td>no</td>
<td>yes</td>
<td>$MR$</td>
</tr>
<tr>
<td></td>
<td>mismatch transitions only</td>
<td>$3N$</td>
<td>no</td>
<td>yes</td>
<td>$M$</td>
</tr>
<tr>
<td>Boyer-Moore</td>
<td>full algorithm</td>
<td>$3N$</td>
<td>$N \div M$</td>
<td>yes</td>
<td>$R$</td>
</tr>
<tr>
<td></td>
<td>mismatched char heuristic only</td>
<td>$MN$</td>
<td>$N \div M$</td>
<td>yes</td>
<td>$R$</td>
</tr>
<tr>
<td>Rabin-Karp†</td>
<td>Monte Carlo (Algorithm 5.8)</td>
<td>$7N$</td>
<td>no</td>
<td>yes†</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Las Vegas</td>
<td>$7N$†</td>
<td>yes</td>
<td>yes</td>
<td>1</td>
</tr>
</tbody>
</table>

† probabilistic guarantee, with uniform hash function