5.3 Substring Search

- introduction
- brute force
- Knuth–Morris–Pratt
- Boyer–Moore
- Rabin–Karp
Substring search quiz 0

Do any of the algorithms we’ve studied so far have a running time that's a decreasing function of the input size?

A. Yes
B. No
C. Haha no way
D. I don't know.
5.3 Substring Search

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

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

Typically $N \gg M$

```
pattern $\longrightarrow$ NEEDLE
```
```
text $\longrightarrow$ INAHAYSTACK NEEDLE INA
```
Substring search applications

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

\[\text{pattern} \rightarrow \text{NEEDLE} \]
\[\text{text} \rightarrow \text{INAHAYSTACK NEEDLE INA} \]

![Find & Replace dialog box](Image)
**Substring search applications**

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

- **pattern** $\rightarrow$ NEEDLE
- **text** $\rightarrow$ INAHAYSSTACK NEEDLE INA

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

[Image: http://citp.princeton.edu/memory]
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
Latest censored keywords in China

female infant + vaccine + die
Hebei + female infant + vaccine
Panama
Banama (Panama)
banama (Panama)
[Panama] Canal Papers
[Pa]nama Papers
launder money + brother-in-law
Xi + brother-in-law
top Chinese official + offshore
Wen [Jiabao] clan
Xi + explode
Wanda + bigwig
Leshi + [Jia] Yueting
50 cents + internet commentary

From http://chinadigitaltimes.net/2013/06/grass-mud-horse-list/
5.3 Substring Search

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

Check for pattern starting at each text position.

txt → A B A C A D A B R A C
A B R A
A B R A
A B R A
A B R A
A B R A
A B R A
A B R A
A B R A
A B R A

entries in red are mismatches
entries in gray are for reference only
entries in black match the text
match
Substring search quiz 1

Suppose you want to count the number of all occurrences of some pattern string of length $M$ in a text of length $N$. What is the order of growth of the best-case and worst-case running time of the brute-force algorithm? Assume $M \leq N$.

A. $N$ and $MN$
B. $N$ and $MN^2$
C. $MN$ and $MN$
D. $MN$ and $MN^2$
E. I don't know.
Backup

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. Streaming algorithm.
5.3 Substring Search

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

**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 BAAAAAB.
- Don't need to back up text pointer!

**Knuth–Morris–Pratt algorithm.** Clever method to always avoid backup!
Searching for BBBBBBBBBBBBBBBBBBBBBB

Let $j = \#\text{characters matched so far.}$

When $j = 0$:
- If we see ‘A’: $j$ remains 0
- If we see ‘B’: $j$ becomes 1

When $1 \leq j < 10$:
- If we see ‘A’: $j = j + 1$
- If we see ‘B’: $j = 1$

$j = 10$: match!

Properties of state transition matrix:
- Depends only on pattern, not text
- #rows = alphabet size
- #columns = length of pattern
- In each col, exactly one row lets us increment the state $j$

<table>
<thead>
<tr>
<th></th>
<th>B</th>
<th>A</th>
<th>A</th>
<th>A</th>
<th>A</th>
<th>A</th>
<th>A</th>
<th>A</th>
<th>A</th>
<th>A</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>j</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>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>0</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>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

\[\begin{array}{cccccccccccc}
\hline
j & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 &   \\
A & 0 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 &   \\
B & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1
\end{array}\]
Exercise

Construct the state transition matrix for the pattern ABABAC.

e.g., \( j = 3 \): we’ve matched the string ‘ABA’ in the text (XXXXXXXXABA)
- If we see ‘A’: we’ve matched ‘A’ (XXXXXXXXABAA) \( \Rightarrow j \) becomes 1
- If we see ‘B’: we’ve matched ‘ABAB’ (XXXXXXXXABAB) \( \Rightarrow j \) becomes 4
- If we see ‘C’: we’ve matched nothing (XXXXXXXXABAC) \( \Rightarrow j \) becomes 0

old:

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

new:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>A</th>
<th>B</th>
<th>A</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>j</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></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Exercise

Construct the state transition matrix for the pattern ABABAC.

e.g., $j = 3$: we’ve matched the string ‘ABA’ in the text (XXXXXXABA)
- If we see ‘A’: we’ve matched ‘A’ (XXXXXXABAA) ⇒ $j$ becomes 1
- If we see ‘B’: we’ve matched ‘ABAB’ (XXXXXXXABAB) ⇒ $j$ becomes 4
- If we see ‘C’: we’ve matched nothing (XXXXXXXABAC) ⇒ $j$ becomes 0

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>A</th>
<th>B</th>
<th>A</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>$j$</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>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>

old:

<table>
<thead>
<tr>
<th></th>
<th>B</th>
<th>A</th>
<th>A</th>
<th>A</th>
<th>A</th>
<th>A</th>
<th>A</th>
<th>A</th>
<th>A</th>
<th>A</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>$j$</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>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>A</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
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 char in alphabet.
- Accept if sequence of state transitions leads to 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[][j]</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 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  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[][][j]
Knuth–Morris–Pratt demo: DFA simulation

A A B A C A A B A B A C A 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[] [j]
Knuth–Morris–Pratt demo: DFA simulation

A A B A C A A B A B A C A A A
Knuth–Morris–Pratt demo: DFA simulation

```
A A B A C A A B A B A C A 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
```
Knuth–Morris–Pratt demo: DFA simulation

A A B A C A A B A B A C A 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 simulation diagram...
Knuth–Morris–Pratt demo: DFA simulation

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

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

dfa[][] | 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  | 0  |
Knuth–Morris–Pratt demo: 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>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>

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

A
B
C

1
2
3
4
5
6
A
A
A
A
A
A
B
B
B
B
B
B
C
C
C
C
C
C
B, C
B, C
B, C
B, C
B, C
B, C

Knuth–Morris–Pratt demo: 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></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>dfa[][][j]</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>6</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>

Diagram of DFA simulation with transitions:

- From state 0: A to 1, B, C to 0
- From state 1: A, B, C to 2, 3, 4
- From state 2: A to 3, C to 1
- From state 3: B to 4
- From state 4: A to 5
- From state 5: C to 6
Knuth–Morris–Pratt demo: DFA simulation

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

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

\begin{align*}
\text{pat.charAt(j)}
\end{align*}

\[
\begin{array}{cccccc}
0 & 1 & 2 & 3 & 4 & 5 \\
A & B & A & B & A & C \\
A & 1 & 1 & 3 & 1 & 5 & 1 \\
B & 0 & 2 & 0 & 4 & 0 & 4 \\
C & 0 & 0 & 0 & 0 & 0 & 6 \\
\end{array}
\]
Knuth–Morris–Pratt demo: DFA simulation

\[
\begin{array}{ccccccc}
\end{array}
\]

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

\[
\begin{array}{ccccccc}
\text{pat.charAt}(j) & B & A & B & A & C \\
\text{dfa[][][j]} & 1 & 1 & 3 & 1 & 5 & 1 \\
B & 0 & 2 & 0 & 4 & 0 & 4 \\
C & 0 & 0 & 0 & 0 & 0 & 6 \\
\end{array}
\]
Knuth–Morris–Pratt demo: 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>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>1</td>
<td>3</td>
<td>1</td>
<td>5</td>
<td>1</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>
```

```
0 ----> 1 ----> 2 ----> 3 ----> 4 ----> 5 ----> 6
```

```java
B, C
```

```java
A, B
```

```java
C
```

```java
B, C
```

```java
A
```

```java
B
```

```java
C
```

```java
B, C
```
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
C | 1 1 3 1 5 1
C | 0 2 0 4 0 4
C | 0 0 0 0 0 6

0 1 2 3 4 5

A

0 1 2 3 4 5 6

0 1 2 3 4 5 6

A B C
Knuth–Morris–Pratt demo: DFA simulation

| A | A | B | A | C | A | A | B | A | B | A | C | A | A |

---

```
<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>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>
```

---

**Diagram: DFA Simulation**

```
0 -> 1 (A, B, C)
1 -> 2 (A)
2 -> 3 (A)
3 -> 4 (B)
4 -> 5 (A)

A -> 1
B -> 2
C -> 6
```

```
B, C
```

```
A
```

```
B
```

```
C
```

```
B, C
```

```
B, C
```

```
A
```

```
B
```

```
C
```

```
B, C
```

```
A
```

```
B
```

```
C
```

```
B, C
```
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

substring found
**Interpretation of Knuth–Morris–Pratt DFA**

**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 $\text{txt}[0..6]$.

The DFA shown above visualizes the states and transitions of the Knuth–Morris–Pratt algorithm. The states are labeled 0 through 6, and the transitions are labeled A, B, and C. The states 0, 1, and 2 are the starting states for the alphabet A, B, and C, respectively. The state 3 is the only state that can match the pattern A, B, A, which is the suffix of $\text{txt}[0..6]$.
Substring search quiz 2

Which state is the DFA in after processing the following input?


A. 0
B. 1
C. 3
D. 4
E. I don't know.
Knuth–Morris–Pratt substring search: Java implementation

Key differences from brute-force implementation.

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

```java
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;
}
```
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;
}
```

Constructing the DFA for KMP substring search for \( A\ B\ A\ B\ A\ C \):

<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>( 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>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>6</td>
</tr>
</tbody>
</table>

\( dfa[j][k] \)
Knuth-Morris-Pratt Running time

Running time.

• Simulate DFA on text: at most $N$ character accesses.
• Build DFA: how to do efficiently? See textbook/video.
  – In the vast majority of applications, the running time of building the DFA is irrelevant. [Arvind’s opinion.]

“Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered.

We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.
**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 once (in the worst case) 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 $M$. 

[Diagram of KMP NFA for ABABAC]
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.

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 \( \{a^n b^n\}^* \), can be recognized in linear time. Other algorithms which run even faster on the average are also considered.

Don Knuth  
Jim Morris  
Vaughan Pratt
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 $t$.

<table>
<thead>
<tr>
<th>yes</th>
<th>yes</th>
<th>no</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{ROTATEDSTRING}$</td>
<td>$\text{ABABABBBABBABA}$</td>
<td>$\text{ROTATEDSTRING}$</td>
</tr>
<tr>
<td>$\text{STRINGROTATED}$</td>
<td>$\text{BABBABBBABABA}$</td>
<td>$\text{GNIRTSDETAOR}$</td>
</tr>
</tbody>
</table>

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

- introduction
- brute force
- Knuth–Morris–Pratt
- Boyer–Moore
- Rabin–Karp

Robert Boyer  
J. Strother 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.

---

![algorithm visualization]

```
<table>
<thead>
<tr>
<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>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>18</th>
<th>19</th>
<th>20</th>
<th>21</th>
<th>22</th>
<th>23</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5</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>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>5</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>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>4</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>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>0</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>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

\[ \text{return } i = 15 \]
Boyer–Moore: mismatched character heuristic

Q. How much to skip?

Case 1. Mismatch character not in pattern.

before

```
taxt  . . . . . . T L E . . . . . .
pat    N E E D L E
```

after

```
taxt  . . . . . . T L E . . . . . .
pat    N E E D L E
```

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

\[
\begin{align*}
\text{txt} & \quad \cdots \quad N \quad L \quad E \quad \cdots \quad \cdots \\
\text{pat} & \quad N \quad E \quad E \quad D \quad L \quad E
\end{align*}
\]

after

\[
\begin{align*}
\text{txt} & \quad \cdots \quad N \quad L \quad E \quad \cdots \quad \cdots \\
\text{pat} & \quad N \quad E \quad E \quad D \quad L \quad E
\end{align*}
\]

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).

before

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

aligned with rightmost E?

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

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
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)

<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>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>E</td>
<td>-1</td>
<td>1</td>
<td>2</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>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>M</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>N</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</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
**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 $N$. the longer the pattern, the faster to search!

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

**Q.** What’s the worst-case input?

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

**Boyer–Moore variant.** Can improve worst case to $\sim 3N$ character compares by adding a KMP-like rule to guard against repetitive patterns.
5.3 SUBSTRING SEARCH

- introduction
- brute force
- Knuth–Morris–Pratt
- Boyer–Moore
- Rabin–Karp
Simplified example

Assume 10-character alphabet: abcdefghij

Text: beachheadacidifiedjadedheadbeheadeadbeef
Pattern: beheaded

“Hash” of string: number obtained replacing each char by corresponding digit

\[
\begin{array}{cccccccccccc}
\text{b} & \text{e} & \text{a} & \text{c} & \text{h} & \text{e} & \text{a} & \text{d} & \text{a} & \text{c} & \text{i} & \text{d} & \text{i} & \text{f} & \text{i} & \text{e} & \text{d} & \text{j} & \text{a} \\
1 & 4 & 7 & 2 & 7 & 7 & 4 & 0 & 3 & 0 & 2 & 8 & 3 & 8 & 5 & 8 & 4 & 3 & 9 & 0 \\
1 & 4 & 7 & 4 & 0 & 3 & 4 & 3 \\
\text{b} & \text{e} & \text{h} & \text{e} & \text{a} & \text{d} & \text{e} & \text{d} \\
\end{array}
\]

\( h_0 = 14727740 \) \quad \text{Precompute hash of pattern: } h = 14740343

Compute \( h_0, h_0, h_2 \ldots \) Match if \( h = h_0 \).
Simplified example

Assume 10-character alphabet: abcdefghij

Text: beachheadacidifiedjadedheadbeheadeadbeef
Pattern: beheaded

“Hash” of string: number obtained replacing each char by corresponding digit

```
beachheadacidifiedjadedheadbeheadeadbeef
1 4 7 2 7 7 4 0 3 0 2 8 3 8 5 8 4 3 9 0
```

$h_0 = 14727740$  
$h_1 = 47277403$  

Precompute hash of pattern: $h = 14740343$
Simplified example

Assume 10-character alphabet: abcdefghij

Text: beachheadacidifiedjadedheadbeheadeadbeef
Pattern: beheaded

“Hash” of string: number obtained replacing each char by corresponding digit

```
| b | e | a | c | h | e | a | d | a | c | i | d | i | f | i | e | d | j | a |
| 1 | 4 | 7 | 2 | 7 | 7 | 4 | 0 | 3 | 0 | 2 | 8 | 3 | 8 | 5 | 8 | 4 | 3 | 9 | 0 |
| 1 | 4 | 7 | 4 | 0 | 3 | 4 | 3 |
| b | e | h | e | a | d | e | d |
```

\[ h_1 = 47277403 \]
\[ h_2 = 72774030 \]

Precompute hash of pattern: \( h = 14740343 \)

Q. Express \( h_{i+1} \) in terms of \( h_i, t[0..N] \) (digits corresponding to text) and \( M \)
A. \( h_{i+1} = (h_i - t_i \cdot 10^{M-1}) \cdot 10 + t_{i+M} \)
Basic idea of Rabin-Karp

- Compute a hash of $\text{pat}[0..M]$.
- For each $i$, compute a hash of $\text{txt}[i..M+i]$.
- If pattern hash = text substring hash, declare match.

| b | e | a | c | h | e | a | d | a | c | i | d | i | f | i | e | d | j | a |
| 1 | 4 | 7 | 2 | 7 | 7 | 4 | 0 | 3 | 0 | 2 | 8 | 3 | 8 | 5 | 8 | 4 | 3 | 9 | 0 |
| 1 | 4 | 7 | 4 | 0 | 3 | 4 | 3 |

Problem 1: alphabet size $R$ may not be 10
Problem 2: integer overflow if $M$ is too long ($M \geq 10$ for 32-bit ints)

Solution 1: use base $R$
Solution 2: do modulo $Q$ arithmetic, where $Q$ is a prime

Now it is an actual hash function — collisions exist. Hash equality does not guarantee substring equality.
Rabin–Karp fingerprint search

Modular hashing.

- Compute a hash of $\text{pat}[0..M]$.
- For each $i$, compute a hash of $\text{txt}[i..M+i]$.
- 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>3</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>5</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>2</td>
<td>6</td>
<td>5</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>% 997 = 613</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>txt.charAt(i)</th>
<th>i</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>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</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>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>5</td>
<td>9</td>
<td>2</td>
<td>6</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>8</td>
<td>9</td>
<td>7</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>3</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>5</td>
<td>% 997 = 508</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>5</td>
<td>9</td>
<td>% 997 = 201</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>5</td>
<td>9</td>
<td>2</td>
<td>% 997 = 715</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>1</td>
<td>5</td>
<td>9</td>
<td>2</td>
<td>6</td>
<td>% 997 = 971</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>5</td>
<td>9</td>
<td>2</td>
<td>6</td>
<td>5</td>
<td>% 997 = 442</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>5</td>
<td>9</td>
<td>2</td>
<td>6</td>
<td>5</td>
<td>3</td>
<td>% 997 = 929</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>6</td>
<td>return $i = 6$</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>
<td></td>
</tr>
</tbody>
</table>

|               | 2 | 6 | 5 | 3 | 5 | % 997 = 613 |

*modular hashing with $R = 10$ and hash(s) = s (mod 997)*
Modular arithmetic

**Math trick.** To keep numbers small, take intermediate results modulo \( Q \).

**Ex.**

\[
(10000 + 535) \times 1000 \pmod{997}
\]

\[
= (30 + 535) \times 3 \pmod{997}
\]

\[
= 1695 \pmod{997}
\]

\[
= 698 \pmod{997}
\]

\( 10000 \pmod{997} = 30 \)

\( 1000 \pmod{997} = 3 \)

\( (a + b) \pmod{Q} = ((a \pmod{Q}) + (b \pmod{Q})) \pmod{Q} \)

\( (a \times b) \pmod{Q} = ((a \pmod{Q}) \times (b \pmod{Q})) \pmod{Q} \)

*two useful modular arithmetic identities*

For more depth, take COS 340
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  |

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>6</td>
<td>5</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>6</td>
<td>5</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>6</td>
<td>5</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>6</td>
<td>5</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>6</td>
<td>5</td>
<td>3</td>
<td>5</td>
</tr>
</tbody>
</table>

26535 = $2 \times 10000 + 6 \times 1000 + 5 \times 100 + 3 \times 10 + 5$

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

// 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;
}
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!

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

- current value
- subtract leading digit
- multiply by radix
- add new trailing digit

(can precompute $R^{M-1}$)

<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>...</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>6</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>

4 1 5 9 2 \text{ current value}

- 4 0 0 0 0 0 \text{ subtract leading digit}

* 1 0 \text{ multiply by radix}

1 5 9 2 0 \text{ add new trailing digit}

+ 6 \text{ new value}
public class RabinKarp
{
    private long patHash; // pattern hash value
    private int M; // pattern length
    private long Q; // modulus
    private int R; // radix
    private long RM1; // R^(M-1) % Q

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

        RM1 = 1;
        for (int i = 1; i <= M-1; i++)
            RM1 = (R * RM1) % Q;
        patHash = hash(pat, M);
    }

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

    public int search(String txt)
    { /* see next slide */ }
}
Monte Carlo version. Return match if hash match.

```java
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 - R*M*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. Modify code to 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 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 two-dimensional patterns.
- Extends to finding multiple patterns.

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

Q. How would you extend Rabin–Karp to efficiently search for any one of $P$ possible patterns in a text of length $N$?
## Substring search cost summary

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

<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></td>
<td></td>
<td>guarantee</td>
<td>typical</td>
<td></td>
<td></td>
</tr>
<tr>
<td>brute force</td>
<td>—</td>
<td>$MN$</td>
<td>$1.1N$</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Knuth-Morris-Pratt</td>
<td>full DFA (Algorithm 5.6)</td>
<td>$2N$</td>
<td>$1.1N$</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>mismatch transitions only</td>
<td>$3N$</td>
<td>$1.1N$</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Boyer-Moore</td>
<td>full algorithm</td>
<td>$3N$</td>
<td>$N/M$</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>mismatched char heuristic only (Algorithm 5.7)</td>
<td>$MN$</td>
<td>$N/M$</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Rabin-Karp†</td>
<td>Monte Carlo (Algorithm 5.8)</td>
<td>$7N$</td>
<td>$7N$</td>
<td>no</td>
<td>yes†</td>
</tr>
<tr>
<td></td>
<td>Las Vegas</td>
<td>$7N†$</td>
<td>$7N$</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

† probabilistic guarantee, with uniform hash function
Substring search quiz

Which of today’s algorithms do you like the best?

A. Knuth-Morris-Pratt (finite automaton).

B. Boyer-Moore (skip-ahead heuristic).

C. Rabin-Karp (rolling hash function).

D. It’s all a blur.