Substring search

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.

http://citp.princeton.edu/memory
Substring search applications

Goal. Find pattern of length \( M \) in a text of length \( N \). Typically \( N \gg M \).

Identify patterns indicative of spam.
- PROFILES
- LOSE WEIGHT
- There is no catch.
- This is a one-time mailing.
- This message is sent in compliance with spam regulations.

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.

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

Entries in red are mismatches.

---

**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++) {
        int j;
        for (j = 0; j < M; j++)
            if (txt.charAt(i+j) != pat.charAt(j))
                break;
        if (j == M) return i;
    }
    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>A A A</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>A A A</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>A A A</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>A A A</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>A A B</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>A A A</td>
</tr>
</tbody>
</table>

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.

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

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).

```
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;
}
```

SUBSTRING SEARCH

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

Intuition. Suppose we are searching in text for pattern $\text{BA\ldots}$.
- Suppose we match 5 chars in pattern, with mismatch on 6th char.
- We know previous 6 chars in text are $\text{BA\ldots}$.
- Don’t need to back up text pointer!

Knuth-Morris-Pratt algorithm. Clever method to always avoid backup. (!)

Deterministic finite state automaton (DFA)

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.

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

If in state $j$ reading char $c$:
- if $j$ is 6 halt and accept
- else move to state $\text{dfa}[c][j]$
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</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[i][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>

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</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[i][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>
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</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>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

<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>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>

substring found
Interpretation of Knuth-Morris-Pratt DFA

Q. What is interpretation of DFA state after reading in \texttt{txt[i]}?

A. State = number of characters in pattern that have been matched.

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

Knuth-Morris-Pratt substring search: Java implementation

Key differences from brute-force implementation.

• Need to precompute \texttt{dfa[][]} from pattern.
• Text pointer \texttt{i} never decrements.

Knuth-Morris-Pratt construction

Include one state for each character in pattern (plus accept state).
**Knuth-Morris-Pratt construction**

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

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

---

**Knuth-Morris-Pratt construction**

Constructing the DFA for KMP substring search for \( \text{A B A B A C} \)

---

**Knuth-Morris-Pratt construction**

Constructing the DFA for KMP substring search for \( \text{A B A B A C} \)

---

**Knuth-Morris-Pratt construction**

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

<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>1</td>
<td>3</td>
<td>1</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>$\text{dfa}[i][j]$</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>

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).

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

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

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

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

Ex. \( \text{dfa}[\text{"A"}][5] = 1 \); \( \text{dfa}[\text{"B"}][5] = 4 \);
Knuth-Morris-Pratt construction (in linear time)

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

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

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 \(j\), \(dfa[pat.charAt(j)][j] = j+1\).

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

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

Knuth-Morris-Pratt construction (in linear time)

Mismatch transition. For state 0 and char \(c\) != \(pat.charAt(j)\), set \(dfa[c][0] = 0\).

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

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

Knuth-Morris-Pratt construction (in linear time)

Mismatch transition. For each state \(j\) and char \(c\) != \(pat.charAt(j)\), set \(dfa[c][j] = dfa[c][X]\); then update \(X = dfa[pat.charAt(j)][X]\).

\[
\begin{array}{cccccc}
0 & 1 & 2 & 3 & 4 & 5 \\
X & simulation of empty string \\
A & B & A & B & A & C \\
dfa[1]\{j\} \\
A & 1 & 3 & 5 \\
B & 2 & 4 \\
C & 0 \\
\end{array}
\]

Constructing the DFA for KMP substring search for A B A B A C
KMP construction (in linear time)

Mismatch transition. For each state $j$ and char $c \neq \text{pat.charAt}(j)$, set $\text{dfa}[c][j] = \text{dfa}[\text{pat.charAt}(j)][X]$; then update $X = \text{dfa}[\text{pat.charAt}(j)][X]$.

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

- X = simulation of B A B A C

Knuth-Morris-Pratt construction (in linear time)

Mismatch transition. For each state $j$ and char $c \neq \text{pat.charAt}(j)$, set $\text{dfa}[c][j] = \text{dfa}[\text{pat.charAt}(j)][X]$; then update $X = \text{dfa}[\text{pat.charAt}(j)][X]$.

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

- X = simulation of B A B A C

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

Knuth-Morris-Pratt construction (in linear time)

Mismatch transition. For each state $j$ and char $c \neq \text{pat.charAt}(j)$, set $\text{dfa}[c][j] = \text{dfa}[\text{pat.charAt}(j)][X]$; then update $X = \text{dfa}[\text{pat.charAt}(j)][X]$.

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

- X = simulation of B A B A C

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

Knuth-Morris-Pratt construction (in linear time)

Mismatch transition. For each state $j$ and char $c \neq \text{pat.charAt}(j)$, set $\text{dfa}[c][j] = \text{dfa}[\text{pat.charAt}(j)][X]$; then update $X = \text{dfa}[\text{pat.charAt}(j)][X]$.

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

- X = simulation of B A B A C

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

Knuth-Morris-Pratt construction (in linear time)

Mismatch transition. For each state $j$ and char $c \neq \text{pat.charAt}(j)$, set $\text{dfa}[c][j] = \text{dfa}[\text{pat.charAt}(j)][X]$; then update $X = \text{dfa}[\text{pat.charAt}(j)][X]$.

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

- X = simulation of B A B A C

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

Knuth-Morris-Pratt construction (in linear time)

Mismatch transition. For each state $j$ and char $c \neq \text{pat.charAt}(j)$, set $\text{dfa}[c][j] = \text{dfa}[\text{pat.charAt}(j)][X]$; then update $X = \text{dfa}[\text{pat.charAt}(j)][X]$.

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

- X = simulation of B A B A C

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

Knuth-Morris-Pratt construction (in linear time)

Mismatch transition. For each state $j$ and char $c \neq \text{pat.charAt}(j)$, set $\text{dfa}[c][j] = \text{dfa}[\text{pat.charAt}(j)][X]$; then update $X = \text{dfa}[\text{pat.charAt}(j)][X]$.

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

- X = simulation of B A B A C

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

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 $X = \text{dfa}[\text{pat.charAt}(j)][X]$.

```
X = simulation of B A B A C
```

<table>
<thead>
<tr>
<th>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>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 $\text{dfa}[c][X]$ to $\text{dfa}[c][j]$ for mismatch case.
- Set $\text{dfa}[\text{pat.charAt}(j)][j] = j+1$ for match case.
- Update $X$.

```
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 $\text{dfa}[\cdot][]$ in time and space proportional to $RM$.

Larger alphabets. Improved version of KMP constructs $\text{nfa}[\cdot][]$ in time and space proportional to $M$. 

```
KMP NFA for ABABAC
```
**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.

---

**FAST PATTERN MATCHING IN STRINGS**

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

**Abstract.** An algorithm is presented which, with all commonness of one given string within another, can scan any string presented to the user of the lengths of the string. The amount of preprocessing is low enough to make the algorithm economical and, if the procedure can be extended to include some more general pattern matching situations. A theoretical analysis of the algorithm shows that the use of substitution of non-equal characters is made quite simple. A theoretical comparison of the algorithm reveals that the use of substitution of non-equal characters is made quite simple. A theoretical comparison of the algorithm reveals that the use of substitution of non-equal characters is made quite simple. A theoretical comparison of the algorithm reveals that the use of substitution of non-equal characters is made quite simple. A theoretical comparison of the algorithm reveals that the use of substitution of non-equal characters is made quite simple. A theoretical comparison of the algorithm reveals that the use of substitution of non-equal characters is made quite simple. A theoretical comparison of the algorithm reveals that the use of substitution of non-equal characters is made quite simple. A theoretical comparison of the algorithm reveals that the use of substitution of non-equal characters is made quite simple. A theoretical comparison of the algorithm reveals that the use of substitution of non-equal characters is made quite simple. A theoretical comparison of the algorithm reveals that the use of substitution of non-equal characters is made quite simple. A theoretical comparison of the algorithm reveals that the use of substitution of non-equal characters is made quite simple. A theoretical comparison of the algorithm reveals that the use of substitution of non-equal characters is made quite simple. A theoretical comparison of the algorithm reveals that the use of substitution of non-equal characters is made quite simple. A theoretical comparison of the algorithm reveals that the use of substitution of non-equal characters is made quite simple.

---

**Boyer-Moore Intuition**

- Scan the text with a window of M chars (length of pattern)

- Case 1: Scan Window is exactly on top of the searched pattern  
  - Starting from one end check if all characters are equal. (We must check!)

- Case 2: Scan Window starts after the pattern starts.

---

**Substring Search**

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

---

**Boyer-Moore Intuition (2)**

- Case 3: Scan Window starts before the pattern starts

- Case 4: Independent
  - In case 4, simply shift window M characters
  - Avoid Case 2
  - Convert Case 3 to Case 1, by shifting appropriately
Boyer Moore Intuition (3)

- If we can recognise the character in the scan window end-point, we can find how many characters to shift.

- So, for example D is the 4th character, we must shift window 4 characters so that they overlap.

Boyer Moore Intuition (4)

- A potential problem, the character in the text can repeat.

- For example, pattern = XXAXX and the text is

  A X A X A X A X A X A X A X

- Solution: be conservative, choose the instance with the least Shift (so we cannot miss the others).

Boyer Moore Intuition (5)

- A X A X A X A X A X A X A X

  Search: XXAXX

- So, for the example when it is A at the endpoint we must shift for 2 characters.

- text:AAAAX we have a mismatch in last A, now we must shift only once, so that we can check the configuration where the A we found moves to middle.

- text:AAYXX we have a mismatch in Y, now we must shift 3 times as we know that the last 2 characters are in pattern and they can be repeating in the first 3 characters.

Boyer-Moore: mismatched character heuristic

Intuition.

- Can skip as many as $M$ text chars when finding one not in the pattern.

  - First we check the character in index pattern.length()-1
  
    - It is N which is not E, so we know that first 5 characters is not a match. Shift text 5 characters
  
    - $S != E$ so shift 5, $E == E$ so we can check for the pattern.length()-2, $L != N$, skip 4.
Boyer-Moore: mismatched character heuristic

Q. How much to skip?

Case 1. Mismatch character not in pattern.

before

\[
\text{txt } \ldots \ldots T \ldots \ldots \\
\text{pat } \text{NEEDLE}
\]

after

\[
\text{txt } \ldots \ldots T \ldots \ldots \\
\text{pat } \text{NEEDLE}
\]

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

Case 2a. Mismatch character in pattern.

before

\[
\text{txt } \ldots \ldots N \ldots \ldots \\
\text{pat } \text{NEEDLE}
\]

after

\[
\text{txt } \ldots \ldots N \ldots \ldots \\
\text{pat } \text{NEEDLE}
\]

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

Boyer-Moore: mismatched character heuristic

Q. How much to skip?

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

before

\[
\text{txt } \ldots \ldots E \ldots \ldots \\
\text{pat } \text{NEEDLE}
\]

after

\[
\text{txt } \ldots \ldots E \ldots \ldots \\
\text{pat } \text{NEEDLE}
\]

aligned with rightmost E?

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

```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;
```

Boyer-Moore skip table computation

---

Another Example

SEARCH FOR: XXXX

```
A X A X A X X X A X X X X X A A A A
```

If the window scan points to an unrecognised character, we can skip past that character. For this example, for the initial step we first match X at the end, when check for previous character (A) which is not in the string we skip 3 steps. The X at the end, matched can still be the first character of the pattern, so we do not skip that.

---

Boyer-Moore: Java implementation

```java
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 $\sim \frac{N}{M}$ character compares to search for a pattern of length $M$ in a text of length $N$. sublinear

Worst-case. Can be as bad as $\sim M \cdot N$.

Boyer-Moore variant. Can improve worst case to $\sim 3 \cdot N$ 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.

**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;
}
```

```plaintext
pat.charAt(i)  1  0  2  3  4  5  6  7  8  9 10 11 12 13 14 15
2  6  5  3  5  % 997 = 613

// Compute hash for M-digit key
private long hash(String key, int M) {
    long h = 0;
    for (int j = 0; j < M; j++)
        h = (R * h + 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 hash function in constant time!

$$X_{i+1} = (X_i - t_i R^{M-1}) R + t_{i+M}$$

```plaintext
i   ...  2  3  4  5  6  7  ...  0  2  6  % 997 = 2
1  2  6  % 997 = (2*10 + 6) % 997 = 26
2  6  5  % 997 = (6*10 + 5) % 997 = 525
3  2  6  5  % 997 = (25*10 + 5) % 997 = 659
4  2  6  5  3  % 997 = (659*10 + 3) % 997 = 2
5  2  6  5  3  % 997 = 929
6  return i = 4  2  6  5  3  5  % 997 = 613
```

```java
public 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;
}
```
Rabin-Karp substring search example

<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>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>3</td>
<td>% 997 = 3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>% 997 = (3^10 + 1) % 997 = 31</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>4</td>
<td>% 997 = (31^20 + 4) % 997 = 114</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>4</td>
<td>3</td>
<td>% 997 = (314^10 + 1) % 997 = 150</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>1</td>
<td>4</td>
<td>3</td>
<td>1</td>
<td>5</td>
<td>% 997 = (310 + 5) % 997 = 508</td>
<td>5</td>
<td>1</td>
<td>4</td>
<td>3</td>
<td>1</td>
<td>5</td>
<td>% 997 = (508 + 3^997 - 30) * 10 + 9) % 997 = 201</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>1</td>
<td>5</td>
<td>9</td>
<td>2</td>
<td>% 997 = ((201 + 1*997 - 30) * 10 + 2) % 997 = 715</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>5</td>
<td>2</td>
<td>6</td>
<td>% 997 = ((715 + 4*(997 - 30)) * 10 + 6) % 997 = 971</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>9</td>
<td>2</td>
<td>6</td>
<td>5</td>
<td>% 997 = ((971 + 1*(997 - 30)) * 10 + 5) % 997 = 442</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>6</td>
<td>5</td>
<td>3</td>
<td>% 997 = ((442 + 5*(997 - 30)) * 10 + 3) % 997 = 929</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>7</td>
<td>9</td>
<td>2</td>
<td>6</td>
<td>5</td>
<td>3</td>
<td>% 997 = ((929 + 9*(997 - 30)) * 10 + 5) % 997 = 613</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>return i-M+1 = 6</td>
<td>2</td>
<td>6</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>9</td>
<td>7</td>
<td>9</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Rabin-Karp: Java implementation

```java
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);
    }

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

    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++)
            // check for hash collision
            txtHash = (txtHash + Q - RM*txt.charAt(i-M) % Q) % Q;
            // using rolling hash function
            txtHash = (txtHash*R + txt.charAt(i)) % Q;
            if (patHash == txtHash) return i - M + 1;
        return N;
    }
}
```

Rabin-Karp: Java implementation (continued)

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++)
        // check for hash collision
        txtHash = (txtHash + Q - RM*txt.charAt(i-M) % Q) % Q;
        // using rolling hash function
        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 \( M N^2 \)), 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 \( M N \)).
**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.

---

**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>Architecture</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>linear</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>linear</td>
<td>no</td>
<td>yes</td>
<td>MR</td>
</tr>
<tr>
<td></td>
<td>mismatch transitions only</td>
<td>( 3N )</td>
<td>linear</td>
<td>no</td>
<td>yes</td>
<td>M</td>
</tr>
<tr>
<td>Boyer-Moore</td>
<td>full algorithm</td>
<td>( 3N )</td>
<td>linear</td>
<td>yes</td>
<td>yes</td>
<td>R</td>
</tr>
<tr>
<td></td>
<td>mismatched char heuristic only (Algorithm 5.7)</td>
<td>( MN )</td>
<td>linear</td>
<td>yes</td>
<td>yes</td>
<td>R</td>
</tr>
<tr>
<td>Rabin-Karp†</td>
<td>Monte Carlo (Algorithm 5.8)</td>
<td>( 7N )</td>
<td>linear</td>
<td>no†</td>
<td>yes</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Las Vegas</td>
<td>( 7N )†</td>
<td>linear</td>
<td>yes</td>
<td>yes</td>
<td>1</td>
</tr>
</tbody>
</table>

† probabilistic guarantee, with uniform hash function