Acknowledgement: The course slides are adapted from the slides prepared by R. Sedgewick and K. Wayne of Princeton University.

Substring search

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

```
pattern  -> NEEDLE

text     -> INAHAYSTACKNEEDLEINA
```

Substring search applications

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

```
pattern  -> NEEDLE

text     -> INAHAYSTACKNEEDLEINA
```

**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.
- PROFITS
- LOSE WEIGHT
- There is no catch.
- This is a one-time mailing.
- This message is sent in compliance with spam regulations.

Substring search applications

Screen scraping. Extract relevant data from web page.

Ex. Find string delimited by <b> and </b> after first occurrence of

Screen scraping: Java implementation

Java library. The \texttt{indexOf()} method in Java’s string library returns the
index of the first occurrence of a given string, starting at a given offset.
Substring Search

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

Brute-force substring search

Check for pattern starting at each text position.

```
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;
}
```

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>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>10</td>
</tr>
</tbody>
</table>
```

```
A  B  A  C  A  D  A  B  R  A  C
```

```
A  B  A  C  A  D  A  B  R  A  C
```

```
entries in red are
mismatches
```

```
entries in gray are
for reference only
```

```
entries in black
match the text
```

```
return i when j is M
```

```
matches
```

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>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>10</td>
</tr>
</tbody>
</table>
```

```
A  A  A  A  A  A  A  A  A  B
```

```
A  A  A  A  A  A  A  A  A  B
```

```
entries in red are
mismatches
```

```
entries in gray are
for reference only
```

```
entries in black
match the text
```

```
return i when j is M
```

```
matches
```

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.

Practical challenge. Avoid backup in text stream.

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

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;
}
**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 `BAAAAB`.
- 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.

**DFA simulation**

### Internal representation

<table>
<thead>
<tr>
<th>Internal representation</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><code>dfa[0][j]</code></td>
<td>A</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td><code>dfa[1][j]</code></td>
<td>B</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td><code>dfa[2][j]</code></td>
<td>C</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>6</td>
</tr>
</tbody>
</table>

**Graphical representation**

**DFA simulation**

### Internal representation

<table>
<thead>
<tr>
<th>Internal representation</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><code>dfa[0][j]</code></td>
<td>A</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td><code>dfa[1][j]</code></td>
<td>B</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td><code>dfa[2][j]</code></td>
<td>C</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>6</td>
</tr>
</tbody>
</table>
DFA simulation

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

pat.charAt(j)
A B A B A C
A 1 1 3 1 5 1
dfa[][][j]
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)
A B A B A C
A 1 1 3 1 5 1
dfa[][][j]
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)
A B A B A C
A 1 1 3 1 5 1
dfa[][][j]
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)
A B A B A C
A 1 1 3 1 5 1
dfa[][][j]
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 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[][]

DFA simulation

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

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

DFA simulation

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

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

DFA simulation

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

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

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].

\[
\begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\end{array}
\]

Interpretation of Knuth-Morris-Pratt DFA

Knuth-Morris-Pratt substring search: Java implementation

Key differences from brute-force implementation.

- Need to precompute \text{dfa[][]} from pattern.
- Text pointer \text{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 NOT_FOUND;
}
```

Running time.

- Simulate DFA on text: at most \(N\) character accesses.
- Build DFA: how to do efficiently? [warning: tricky algorithm ahead]

Knuth-Morris-Pratt construction

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

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

Constructing the DFA for KMP substring search for \text{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>A</td>
<td></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>3</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>2</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>6</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:** 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></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>3</td>
<td>5</td>
<td></td>
<td></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></td>
<td>0</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

**Knuth-Morris-Pratt construction**

**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></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>3</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>2</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>6</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
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**

$$\text{pat.charAt}(j) = \begin{array}{l}
A \quad B \\
A \quad B \\
A \quad C \\
A \\
B \\
C
\end{array}$$

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

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**

$$\text{pat.charAt}(j) = \begin{array}{l}
A \quad B \\
A \quad B \\
A \quad C \\
A \\
B \\
C
\end{array}$$

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

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**

$$\text{pat.charAt}(j) = \begin{array}{l}
A \quad B \\
A \quad B \\
A \quad C \\
A \\
B \\
C
\end{array}$$

$$\text{dfa}[][] = \begin{array}{cccccc}
0 & 1 & 2 & 3 & 4 & 5 \\
0 & 2 & 0 & 4 & 0 & 0 \\
1 & 1 & 3 & 1 & 5 & 1 \\
B & 0 & 0 & 0 & 0 & 0 \\
C & A & B & A & B & A \\
A & B & A & C & A & C
\end{array}$$
How to build DFA from pattern?

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

<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>
</tbody>
</table>

\[
\text{dfa}[j][j] = \begin{cases} 0 & \text{if } \text{pat.charAt(j)} = A \\ 1 & \text{if } \text{pat.charAt(j)} = B \\ 2 & \text{if } \text{pat.charAt(j)} = C \end{cases}
\]

How to build DFA from pattern?

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. Seems to require \( j \) steps.

Ex. \( \text{dfa}[\text{A'}][3] = 1 \); \( \text{dfa}[\text{B'}][5] = 4 \)

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

Ex. \( \text{dfa}[\text{A'}][5] = 1 \); \( \text{dfa}[\text{B'}][5] = 4 \); \( X' = 0 \)

from state \( X \)

take transition 'A' = \( \text{dfa}[\text{A'}][5] \)

from state \( X \)

take transition 'B' = \( \text{dfa}[\text{B'}][5] \)

from state \( X \)

take transition 'C' = \( \text{dfa}[\text{C'}][5] \)

Running time. Takes only constant time if we maintain state \( X \).
Knuth-Morris-Pratt construction (in linear time)

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

```
<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>df[a][j]</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>
```

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

```
<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>df[a][j]</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>
```

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 \( c \neq pat.charAt(j) \), set \( df[a][0] = 0 \).

```
<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>df[a][j]</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>
```

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 \( c \neq pat.charAt(j) \), set \( df[a][j] = df[a][x] \); then update \( x = df[pat.charAt(j)][x] \).

```
<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>df[a][j]</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>4</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 $\text{dfa}[c][j] = \text{dfa}[c][X]$; then update $X = \text{dfa}[\text{pat.charAt}(j)][X]$.

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

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

Knuth-Morris-Pratt construction (in linear time)

Constructing the DFA for KMP substring search for A B A B A C
**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 $X = \text{dfa}[\text{pat.charAt}(j)][X]$.

<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>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>$\text{dfa}[c][j]$</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>C</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

**Knuth-Morris-Pratt construction (in linear time)**

<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>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>$\text{dfa}[c][j]$</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>C</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

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

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

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

**Larger alphabets.** Improved version of KMP constructs $\text{dfa}[][]$ 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.

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.

Boyer-Moore: mismatched character heuristic

Q. How much to skip!

Case 1. Mismatch character not in pattern.

Before

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

Pattern: NEEDLE

Return $i = 15$

After

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

Mismatch character 'T' not in pattern: increment $i$ one character beyond 'T'
Boyer-Moore: mismatched character heuristic

Q. How much to skip?

Case 2a. Mismatch character in pattern.

before
\[
\begin{array}{c}
\text{txt} \ldots N \ldots E \ldots L \ldots E \\
\text{pat} \quad N \quad E \quad E \quad D \quad L \quad E
\end{array}
\]

after

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

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

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

after

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

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

Boyer-Moore: mismatched character heuristic

Q. How much to skip?

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

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

\[
\begin{array}{ccccccc}
\text{N} & \text{E} & \text{E} & \text{D} & \text{L} & \text{E} & \text{right}[] \\
\hline
0 & 1 & 2 & 3 & 4 & 5 \\
A & -1 & -1 & -1 & -1 & -1 & -1 \\
B & -1 & -1 & -1 & -1 & -1 & -1 \\
C & -1 & -1 & -1 & -1 & -1 & -1 \\
D & -1 & -1 & -1 & 3 & 3 & 3 \\
E & -1 & -1 & 1 & 2 & 2 & 5 \\
\ldots & -1 & -1 & -1 & 4 & 4 & -1 \\
L & -1 & -1 & -1 & 0 & 0 & 0 \\
M & -1 & -1 & -1 & 0 & 0 & 0 \\
N & -1 & 0 & 0 & 0 & 0 & 0 \\
\ldots & -1 & -1 & -1 & -1 & -1 & -1 \\
\end{array}
\]

Boyer-Moore skip table computation
**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 \( N \).

**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.
Efficiently computing the hash function

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

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

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

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

Key property. Can update hash function in constant time!

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

Rabin-Karp substring search example

<table>
<thead>
<tr>
<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>t</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>3</td>
<td>1</td>
<td>1</td>
<td>9</td>
<td>2</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>$t_i$</td>
<td>0</td>
<td>2</td>
<td>6</td>
<td>5</td>
<td>997 = 2</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>$t_{i-1}$</td>
<td>1</td>
<td>2</td>
<td>6</td>
<td>5</td>
<td>997 = (2*10 + 6) % 997 = 26</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>$t_{i-2}$</td>
<td>2</td>
<td>2</td>
<td>6</td>
<td>5</td>
<td>997 = (26*10 + 5) % 997 = 265</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>$t_{i-3}$</td>
<td>2</td>
<td>6</td>
<td>5</td>
<td>3</td>
<td>997 = (265*10 + 3) % 997 = 659</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>$t_{i-4}$</td>
<td>4</td>
<td>2</td>
<td>6</td>
<td>5</td>
<td>3</td>
<td>997 = (659*10 + 3) % 997 = 613</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Efficiently computing the hash function

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

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

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

Rabin-Karp: Java implementation

```java
public class RabinKarp
{   private long patHash;    // pattern hash value    private int M;           // pattern length   // Compute hash for M-digit keyprivate 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;}
   public int search(String txt) { /* see next slide */  }

   public RabinKarp(String pat) {   // Compute hash for M-digit key   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 RM;   // radix    private int R;   // modulus    private long Q;   // pattern hash value
}
```

public int search(String txt) { /* see next slide */  }
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: Java implementation (continued)

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*R + txt.charAt(i)) % Q;
        txtHash = (txtHash + Q - RM*txt.charAt(i-M) % Q) % 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 $M^2$), 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 $M N$).

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>Brute force</td>
<td>—</td>
<td>$M N$</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>$2 N$</td>
<td>no</td>
<td>yes</td>
<td>MR</td>
</tr>
<tr>
<td></td>
<td>mismatch transitions only</td>
<td>$3 N$</td>
<td>no</td>
<td>yes</td>
<td>M</td>
</tr>
<tr>
<td>Boyer-Moore</td>
<td>full algorithm</td>
<td>$3 N$</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>yes</td>
<td>yes</td>
<td>R</td>
</tr>
<tr>
<td>Rabin-Karp†</td>
<td>Monte Carlo (Algorithm 5.8)</td>
<td>$7 N$</td>
<td>no</td>
<td>yes</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Las Vegas</td>
<td>$7 N$</td>
<td>yes</td>
<td>yes</td>
<td>1</td>
</tr>
</tbody>
</table>

† probabilistic guarantee, with uniform hash function
Substring search, Regular Expressions

- Substring search
- Regular Expressions
- REs and NFAs
- NFA simulation
- NFA construction
- Applications

Pattern matching

Substring search. Find a single string in text.

Pattern matching. Find one of a specified set of strings in text.

Ex. [genomics]

- Fragile X syndrome is a common cause of mental retardation.
- Human genome contains triplet repeats of CGG or AGG, bracketed by GCC at the beginning and CTG at the end.
- Number of repeats is variable, and correlated with syndrome.

pattern: `GCC (CGG | AGG) * CTG`

text: `GCCGCGTGTGTGCAGAGAGTGCTTAAAGCTGCCGCGAGGGCTGGCGCGGAGGCTG`

Syntax highlighting

```java
public class NFA {
    private Digraph G; // digraph of epsilon transitions
    private String regexp; // regular expression
    private int M; // number of characters in regular expression

    public NFA(String regexp) {
        this.regexp = regexp;
        M = regexp.length();
        Stack<Integer> ops = new Stack<Integer>();
        G = new Digraph(M);
    }
}
```

Google code search

Search public source code

Search via regular expression, e.g. `java\.*\|java\$`

Search Options
- In Search Box
- Package: `package:linux-2.6`
- Language: `Any language` `lang:c++`
- File Path: `file:(code)` `!for|search`
- Class: `class:HashMap`
- Function: `function:toString`
- License: `Any license` `license:mozilla`
- Case Sensitive: `Case sensitive: false` `case:yes`

http://code.google.com/p/chromium/source/search
**Pattern matching: applications**

Test if a string matches some pattern.
- Process natural language.
- Scan for virus signatures.
- Specify a programming language.
- Search genome using PROSITE patterns.
- Filter text (spam, NetNanny, Carnivore, malware).
- Validate data-entry fields (dates, email, URL, credit card).

Parse text files.
- Compile a Java program.
- Crawl and index the Web.
- Read in data stored in ad hoc input file format.
- Create Java documentation from Javadoc comments.

**Regular expressions**

A regular expression is a notation to specify a set of strings.

<table>
<thead>
<tr>
<th>operation</th>
<th>order</th>
<th>example RE</th>
<th>matches</th>
<th>does not match</th>
</tr>
</thead>
<tbody>
<tr>
<td>concatenation</td>
<td>3</td>
<td>AABAAB</td>
<td>AABAAB</td>
<td>every other string</td>
</tr>
<tr>
<td>or</td>
<td>4</td>
<td>AA</td>
<td>BAAB</td>
<td>AA BAAB</td>
</tr>
<tr>
<td>closure</td>
<td>2</td>
<td>AB*A</td>
<td>AB ABABABABA</td>
<td>AB ABABABA</td>
</tr>
<tr>
<td>parentheses</td>
<td>1</td>
<td>A (A</td>
<td>B)AAB</td>
<td>AAAAB</td>
</tr>
</tbody>
</table>

**Regular expression shortcuts**

Additional operations are often added for convenience.

<table>
<thead>
<tr>
<th>operation</th>
<th>example RE</th>
<th>matches</th>
<th>does not match</th>
</tr>
</thead>
<tbody>
<tr>
<td>wildcard</td>
<td>.U.U.U.</td>
<td>CUMULUS JUGULUM</td>
<td>SUCCUS TUNMULTIOUS</td>
</tr>
<tr>
<td>character class</td>
<td>[A-Za-z][a-z]*</td>
<td>word Capitalized</td>
<td>camelCase 4illegal</td>
</tr>
<tr>
<td>at least 1</td>
<td>A(BC)+DE</td>
<td>ABCDE</td>
<td>ABDE</td>
</tr>
<tr>
<td>exactly k</td>
<td>[0-9]{3}-[0-9]{2}-[0-9]{4}</td>
<td>166-11-4433</td>
<td>166-45-1111</td>
</tr>
<tr>
<td>complement</td>
<td>![^A-Za-z]</td>
<td>RHYTHM</td>
<td>DECADE</td>
</tr>
</tbody>
</table>

Ex. [A-E]+ is shorthand for (A|B|C|D|E) (A|B|C|D|E)*

**Regular expression examples**

RE notation is surprisingly expressive

<table>
<thead>
<tr>
<th>regular expression</th>
<th>matches</th>
<th>does not match</th>
</tr>
</thead>
<tbody>
<tr>
<td>.<em>SPB.</em></td>
<td>RASPBERRY CRISPHEAD</td>
<td>SUBSPACE SUBSPECIES</td>
</tr>
<tr>
<td>[0-9]{3}-[0-9]{2}-[0-9]{4} (Social Security numbers)</td>
<td>166-11-4433</td>
<td>11-55555555 8675309</td>
</tr>
<tr>
<td>[a-z]+@[a-z]+.(edu</td>
<td>com) (email addresses)</td>
<td><a href="mailto:wayne@princeton.edu">wayne@princeton.edu</a></td>
</tr>
<tr>
<td>$_A-Za-z$</td>
<td>ident3</td>
<td>PatternMatcher</td>
</tr>
<tr>
<td>$_A-Za-z$</td>
<td>3a</td>
<td>ident$3</td>
</tr>
</tbody>
</table>

REs plays a well-understood role in the theory of computation.
Can the average web surfer learn to use REs?

Google. Supports * for full word wildcard and | for union.

Can the average programmer learn to use REs?

Perl RE for valid RFC822 email addresses

http://www.ex-parrot.com/~perl5/FAQ/mail-RFC822-Address.html

Regular expressions to the rescue

Writing a RE is like writing a program.
- Need to understand programming model.
- Can be easier to write than read.
- Can be difficult to debug.

"Some people, when confronted with a problem, think 'I know I'll use regular expressions.' Now they have two problems."

— Jamie Zawinski (flame war on alt.religion.emacs)

Bottom line. REs are amazingly powerful and expressive, but using them in applications can be amazingly complex and error-prone.
### Pattern matching implementation: basic plan (first attempt)

**Overview** is the same as for KMP.
- No backup in text input stream.
- Linear-time guarantee.

**Underlying abstraction.** Deterministic finite state automata (DFA).

**Basic plan.** [apply Kleene’s theorem]
- Build DFA from RE.
- Simulate DFA with text as input.

**Bad news.** Basic plan is infeasible (DFA may have exponential # of states).

### Pattern matching implementation: basic plan (revised)

**Overview** is similar to KMP.
- No backup in text input stream.
- **Quadratic-time guarantee** (linear-time typical).

**Underlying abstraction.** Non-deterministic finite state automata (NFA).

**Basic plan.** [apply Kleene’s theorem]
- Build NFA from RE.
- Simulate NFA with text as input.

**Q.** What is an NFA?
Nondeterministic finite-state automata

Regular-expression-matching NFA.
- RE enclosed in parentheses.
- One state per RE character \( (\text{start } = 0, \text{accept } = M) \).
- Red \( \varepsilon \)-transition (change state, but don’t scan text).
- Black match transition (change state and scan to next text char).
- Accept if any sequence of transitions ends in accept state.

Nondeterminism.
- One view: machine can guess the proper sequence of state transitions.
- Another view: sequence is a proof that the machine accepts the text.

Q. Is AAAABD matched by NFA?
A. Yes, because some sequence of legal transitions ends in state 11.

A     A     A     A     B        D
0   1   2   3   2   3   2   3   2   3   4   5   8   9   10   11

match transition: scan to next input character and change state
\( \varepsilon \)-transition: change state with no match
accept state reached and all text characters scanned: pattern found

Wrong guess if input is
A     A     A     A     B        D
0   1   2   3   2   3   2   3   2   3   4

no way out of state 4

Stalling sequences for 
\( (A^*B | A C)D \) NFA
- no way out of state 4
- no way out of state 7

A     A     A     A     C
0   1   2   3   2   3   2   3   2   3   4

no way out of state 4

Q. Is AAAAC matched by NFA?
A. No, because no sequence of legal transitions ends in state 11.

\[ \text{[but need to argue about all possible sequences]} \]
**Nondeterminism**

Q. How to determine whether a string is matched by an automaton?

DFA. Deterministic ⇒ exactly one applicable transition.

NFA. Nondeterministic ⇒ can be several applicable transitions; need to select the right one!

Q. How to simulate NFA?
A. Systematically consider all possible transition sequences.

![NFA diagram](image1)

NFA corresponding to the pattern \((A \ast B | AC)D\)

**NFA representation**

**State names.** Integers from 0 to \(M\), where \(M\) is the number of symbols in RE

**Match-transitions.** Keep regular expression in array \(re[\]\).

**\(\epsilon\)-transitions.** Store in a digraph \(G\).

- 0→1, 1→2, 1→6, 2→3, 3→2, 3→4, 5→8, 8→9, 10→11

![NFA diagram](image2)

NFA corresponding to the pattern \((A \ast B | AC)D\)

**NFA simulation**

Q. How to efficiently simulate an NFA?
A. Maintain set of all possible states that NFA could be in after reading in the first \(i\) text characters.

![NFA simulation](image3)

Q. How to perform reachability?
**NFA simulation**

**Goal.** Check whether input matches pattern.

![NFA simulation diagram](image)

- **Input:** A A B D
- **ε-transitions:**
- **Match transitions:**

*NFA corresponding to the pattern ((A* B | A C) D)*

**NFA simulation**

Read next input character.
- Find states reachable by match transitions.
- Find states reachable by ε-transitions

**Input:** A A B D

![NFA simulation diagram](image)

- **ε-transitions:**
- **States reachable via ε-transitions from start:** {0, 1, 2, 3, 4, 6}

**NFA simulation**

Read next input character.
- Find states reachable by match transitions.
- Find states reachable by ε-transitions

**Input:** A A B D

![NFA simulation diagram](image)

- **States reachable via ε-transitions from start:** {0, 1, 2, 3, 4, 6}
Read next input character.
- Find states reachable by match transitions.
- Find states reachable by $\varepsilon$-transitions

NFA simulation

set of states reachable after matching $A$

set of states reachable via $\varepsilon$-transitions after matching $A$

Read next input character.
- Find states reachable by match transitions.
- Find states reachable by $\varepsilon$-transitions

NFA simulation

set of states reachable after matching $A$ : $\{3, 7\}$

set of states reachable via $\varepsilon$-transitions after matching $A$ : $\{2, 4, 7\}$
NFA simulation

Read next input character.
• Find states reachable by match transitions.
• Find states reachable by ε-transitions

set of states reachable after matching A A

NFA simulation

Read next input character.
• Find states reachable by match transitions.
• Find states reachable by ε-transitions

set of states reachable via ε-transitions after matching A A

NFA simulation

Read next input character.
• Find states reachable by match transitions.
• Find states reachable by ε-transitions

set of states reachable after matching A A : { 3 }

NFA simulation

Read next input character.
• Find states reachable by match transitions.
• Find states reachable by ε-transitions

set of states reachable via ε-transitions after matching A A : { 2, 3, 4 }
NFA simulation

Read next input character.
- Find states reachable by match transitions.
- Find states reachable by $\varepsilon$-transitions

match B transition

set of states reachable after matching A A B

input A A B D

$\varepsilon$-transitions

set of states reachable via $\varepsilon$-transitions after matching A A B

NFA simulation

Read next input character.
- Find states reachable by match transitions.
- Find states reachable by $\varepsilon$-transitions

set of states reachable after matching A A B :  \{ 5 \}

set of states reachable via $\varepsilon$-transitions after matching A A B :  \{ 5, 8 \}
Find states reachable by match transitions.
Find states reachable by ε-transitions

NFA simulation

Input: A A B D

set of states reachable after matching A A B D

match D transition

NFA simulation

Input: A A B D

set of states reachable after matching A A B D: {10}

ε-transitions

NFA simulation

Input: A A B D

set of states reachable via ε-transitions after matching A A B D

NFA simulation

Input: A A B D

set of states reachable via ε-transitions after matching A A B D: {10, 11}
When no more input characters:

- Accept if any state reachable is an accept state.
- Reject otherwise.

NFA simulation

| input | A | A | B | D |

set of states reachable: {10, 11}

Digraph reachability

Digraph reachability. Find all vertices reachable from a given source or set of vertices.

Solution. Run DFS from each source, without unmarking vertices.

Performance. Runs in time proportional to $E + V$.

NFA simulation: Java implementation

```java
public class NFA
{
    private char[] re; // match transitions
    private Digraph G; // epsilon transition digraph
    private int M; // number of states

    public NFA(String regexp)
    {
        M = regexp.length();
        re = regexp.toCharArray();
        G = buildEpsilonTransitionDigraph();
    }

    public boolean recognizes(String txt)
    {
        /* see next slide */
    }

    public Digraph buildEpsilonTransitionDigraph()
    {
        /* stay tuned */
    }
}
```

NFA simulation: Java implementation

```java
public boolean recognizes(String txt)
{
    Bag<Integer> pc = new Bag<Integer>();
    DirectedDFS dfs = new DirectedDFS(G, 0);
    for (int v = 0; v < G.V(); v++)
        if (dfs.marked(v)) pc.add(v);

    for (int i = 0; i < txt.length(); i++)
    {
        Bag<Integer> match = new Bag<Integer>();
        for (int v : pc)
            if (v == M) continue;
            if (re[v] == txt.charAt(i) || re[v] == '.')
                match.add(v+1);
        dfs = new DirectedDFS(G, match);
        pc = new Bag<Integer>();
        for (int v = 0; v < G.V(); v++)
            if (dfs.marked(v)) pc.add(v);
    }

    for (int v : pc)
        if (v == M) return true;
    return false;
}
```
**NFA simulation: analysis**

**Proposition.** Determining whether an $N$-character text is recognized by the NFA corresponding to an $M$-character pattern takes time proportional to $MN$ in the worst case.

**Pf.** For each of the $N$ text characters, we iterate through a set of states of size no more than $M$ and run DFS on the graph of $\varepsilon$-transitions. [The NFA construction we will consider ensures the number of edges $\leq 3M$.]

![Diagram of NFA](image)

**Building an NFA corresponding to an RE**

**States.** Include a state for each symbol in the RE, plus an accept state.

![Diagram of NFA](image)
**Building an NFA corresponding to an RE**

**Parentheses.** Add ε-transition edge from parentheses to next state.

![Diagram](image1)

NFA corresponding to the pattern \((\text{A}^* \text{B} | \text{A} \text{C}) \text{D}\)

**Building an NFA corresponding to an RE**

**Or.** Add two ε-transition edges for each | operator.

![Diagram](image2)

NFA corresponding to the pattern \((\text{A}^* \text{B} | \text{A} \text{C}) \text{D}\)

**Building an NFA corresponding to an RE**

**Closure.** Add three ε-transition edges for each * operator.

![Diagram](image3)

NFA corresponding to the pattern \((\text{A}^* \text{B} | \text{A} \text{C}) \text{D}\)

**NFA construction: implementation**

**Goal.** Write a program to build the ε-transition digraph.

**Challenges.** Remember left parentheses to implement closure and or; need to remember | to implement or.

**Solution.** Maintain a stack.
- ( symbol: push ( onto stack.
- | symbol: push | onto stack.
- ) symbol: pop corresponding ( and possibly intervening |; add ε-transition edges for closure/or.
NFA construction

Left parenthesis.
• Add ε-transition to next state.
• Push index of state corresponding to ( onto stack.

NFA construction

Alphabet symbol.
• Add match transition to next state.
• Do one-character lookahead:
  add ε-transitions if next character is *.
NFA construction

Alphabet symbol.

• Add match transition to next state.
• Do one-character lookahead:
  add $\epsilon$-transitions if next character is $\ast$.

0 1 2 3 4
1 2 3

$((A\ast B\mid A C)D)$

NFA construction

Closure symbol.

• Add $\epsilon$-transition to next state.

0 1 2 3
1 2 3

$((A\ast B\mid A C)D)$

NFA construction

Alphabet symbol.

• Add match transition to next state.
• Do one-character lookahead:
  add $\epsilon$-transitions if next character is $\ast$.

0 1 2 3 4
1 2 3 4

$((A\ast B\mid A C)D)$

NFA construction

Or symbol.

• Push index of state corresponding to $|$ onto stack.

0 1 2 3 4 5
1 2 3 4 5

$((A\ast B\mid A C)D)$
NFA construction

Alphabet symbol.
• Add match transition to next state.
  • Do one-character lookahead:
    add ε-transitions if next character is *.

Add match transition to next state.
• Do one-character lookahead:
  add ε-transitions if next character is *.

Right parenthesis.
• Add ε-transition to next state.
• Pop corresponding ( and possibly intervening |:
  add ε-transition edges for or.
  • Do one-character lookahead:
    add ε-transitions if next character is *.

Add match transition to next state.
• Do one-character lookahead:
  add ε-transitions if next character is *.
NFA construction

Right parenthesis.
- Add ε-transition to next state.
- Pop corresponding ( and possibly intervening |; add ε-transition edges for or.
- Do one-character lookahead:
  add ε-transitions if next character is *.

NFA corresponding to the pattern ( ( A * B | A C ) D )

NFA construction

End of regular expression.
- Add accept state.

NFA corresponding to the pattern ( ( A * B | A C ) D )

NFA construction: Java implementation

```java
private Digraph buildEpsilonTransitionDigraph() {
    Digraph G = new Digraph(M+1);
    Stack<Integer> ops = new Stack<Integer>();
    for (int i = 0; i < M; i++) {
        int lp = i;
        if (re[i] == '(' || re[i] == '|') ops.push(i);
        else if (re[i] == ')') {
            int or = ops.pop();
            if (re[or] == '|') {
                lp = ops.pop();
                G.addEdge(lp, or+1);
                G.addEdge(or, i);
            } else lp = or;
        }
        if (i < M-1 && re[i+1] == '*') {
            G.addEdge(lp, i+1);
            G.addEdge(i+1, lp);
        } else if (re[i] == '(' || re[i] == '*' || re[i] == ')')
            G.addEdge(i, i+1);
    }
    return G;
}
```

Closure (needs 1-character lookahead)
or
left parentheses and |
metasymbols

**NFA construction: analysis**

**Proposition.** Building the NFA corresponding to an $M$-character RE takes time and space proportional to $M$.

**Pf.** For each of the $M$ characters in the RE, we add at most three $\varepsilon$-transitions and execute at most two stack operations.

![NFA diagram](image)

NFA corresponding to the pattern $( ( A \ast B | A C ) D )$

---

**Generalized regular expression print**

**Grep.** Take a RE as a command-line argument and print the lines from standard input having some substring that is matched by the RE.

```java
public class GREP {
    public static void main(String[] args) {
        String regexp = "(.*" + args[0] + ".*)";
        NFA nfa = new NFA(regexp);
        while (StdIn.hasNextLine()) {
            String line = StdIn.readLine();
            if (nfa.recognizes(line))
                StdOut.println(line);
        }
    }
}
```

**Bottom line.** Worst-case for grep (proportional to $MN$) is the same as for brute-force substring search.

---

**Typical grep application: crossword puzzles**

```bash
% grep "s..ict.." words.txt
constrictor
stricter
stricture

% more words.txt
aaback
abacus
abalone
abandon
...

% grep "s...ict..." words.txt
constrictor
stricter
stricture
```

---

**Regular Expressions**

- REs and NFAs
- NFA simulation
- NFA construction
- Applications
Industrial-strength grep implementation

To complete the implementation:
- Add character classes.
- Handle metacharacters.
- Add capturing capabilities.
- Extend the closure operator.
- Error checking and recovery.
- Greedy vs. reluctant matching.

Ex. Which substring(s) should be matched by the RE <blink>.*</blink> ?

Greedy vs. Reluctant

Regular expressions in other languages

Broadly applicable programmer’s tool.
- Originated in Unix in the 1970s.
- Many languages support extended regular expressions.
- Built into grep, awk, emacs, Perl, PHP, Python, JavaScript, ...

PERL. Practical Extraction and Report Language.

Regular expressions in Java

Validity checking. Does the input match the regexp?
Java string library. Use input.matches(regexp) for basic RE matching.

Harvesting information

Goal. Print all substrings of input that match a RE.

% java Harvester "gcg(cgg|agg)*ctg" chromosomeX.txt
gcgccggcggcggcggcggcggctg
gcgctg

gcgctg
gcgccggcggcggaggcggaggcggctg

% java Harvester "http://(\w+\.)*(\w+)"
http://www.cs.princeton.edu
http://www.princeton.edu
http://www.google.com
http://www.cs.princeton.edu/news
Harvesting information

RE pattern matching is implemented in Java’s java.util.regex.Pattern and java.util.regex.Matcher classes.

```java
import java.util.regex.Pattern;
import java.util.regex.Matcher;

public class Harvester
{
    public static void main(String[] args)
    {
        String regexp = args[0];
        In in = new In(args[1]);
        String input = in.readAll();
        Pattern pattern = Pattern.compile(regexp);
        Matcher matcher = pattern.matcher(input);
        while (matcher.find())
        {
            StdOut.println(matcher.group());
        }
    }
}
```

Algorithmic complexity attacks

Warning. Typical implementations do not guarantee performance!

SpamAssassin regular expression.

```java
% java RE "([a-z]+@([a-z]+\.(?!\.[a-z]+))\.[a-z]+)" spammer@x......................
```

Not-so-regular expressions

Back-references.
- \1 notation matches subexpression that was matched earlier.
- Supported by typical RE implementations.

```java
\1\1          // beriberi couscous
1?5|^(11+?)\1+  // 1111 111111 1111111111
```

Some non-regular languages.
- Strings of the form \w\w for some string \w: beriberi.
- Unary strings with a composite number of 1s: 111111.
- Bitstrings with an equal number of 0s and 1s: 01110100.
- Watson-Crick complemented palindromes: atttggaat.

Remark. Pattern matching with back-references is intractable.

Context

Abstract machines, languages, and nondeterminism.
- Basis of the theory of computation.
- Intensively studied since the 1930s.
- Basis of programming languages.

Compiler. A program that translates a program to machine code.
- KMP string ⇒ DFA.
- grep RE ⇒ NFA.
- javac Java language ⇒ Java byte code.
Summary of pattern-matching algorithms

Programmer.
• Implement substring search via DFA simulation.
• Implement RE pattern matching via NFA simulation.

Theoretician.
• RE is a compact description of a set of strings.
• NFA is an abstract machine equivalent in power to RE.
• DFAs and REs have limitations.

You. Practical application of core computer science principles.

Example of essential paradigm in computer science.
• Build intermediate abstractions.
• Pick the right ones!
• Solve important practical problems.