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

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

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

typically $N \gg M$

\begin{align*}
\text{pattern} & \rightarrow \text{NEEDLE} \\
\text{text} & \rightarrow \text{INAHAYSTACK} \text{ NEEDLE INA}
\end{align*}

match
**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 $N$.

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

Electronic surveillance.

Need to monitor all internet traffic. (security)

No way! (privacy)

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

OK. Build a machine that just looks for that.

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

Screen scraping. Extract relevant data from web page.

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

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

...<tr>
<td class= "yfnc_tablehead1" width= "48%">Last Trade:<br/>
<td class= "yfnc_tabledata1">452.92<br/>
</td>
<td class= "yfnc_tablehead1" width= "48%">Trade Time:<br/>
<td class= "yfnc_tabledata1">
...
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
Check for pattern starting at each text position.

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

Brute-force substring search

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

Return i when j is M.

Match.
Brute-force substring search: Java implementation

Check for pattern starting at each text position.

```
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;
}
```

<table>
<thead>
<tr>
<th>i</th>
<th>j</th>
<th>i+j</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A</th>
<th>C</th>
<th>A</th>
<th>D</th>
<th>A</th>
<th>B</th>
<th>R</th>
<th>A</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>3</td>
<td>7</td>
<td></td>
<td>A</td>
<td>D</td>
<td>A</td>
<td>C</td>
<td>R</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>5</td>
<td></td>
<td>A</td>
<td>D</td>
<td>A</td>
<td>C</td>
<td>R</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

index in text where pattern starts

not found
Brute-force substring search: worst case

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

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

`txt` → A A A A A A A A A B

`pat` ← match

Worst case. \( \sim MN \) char compares.
In many applications, we want to avoid backup in text stream.
- Treat input as stream of data.
- Abstract model: standard input.

Brute-force algorithm needs backup for every mismatch.

**Approach 1.** Maintain buffer of last $M$ characters.
**Approach 2.** Stay tuned.
Brute-force substring search: alternate implementation

Same sequence of char compares as previous implementation.

- \( i \) points to end of sequence of already-matched chars in text.
- \( j \) stores number of already-matched chars (end of sequence in pattern).

```
public static int search(String pat, String txt)
{
    int i, N = txt.length();
    int j, M = pat.length();
    for (i = 0, j = 0; i < N && j < M; i++)
    {
        if (txt.charAt(i) == pat.charAt(j)) j++;
        else { i -= j; j = 0;  }
    }
    if (j == M) return i - M;
    else            return N;
}
```
Algorithmic challenges in substring search

Brute-force is not always good enough.

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

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

Now is the time for all people to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for many good people to come to the aid of their party. Now is the time for a lot of good people to come to the aid of their party. Now is the time for all of the good people to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for each good person to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for all good Republicans to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for all good Democrats to come to the aid of their party. Now is the time for all people to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for many good people to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for a lot of good people to come to the aid of their party. Now is the time for all of the good people to come to the aid of their party. Now is the time for all good people to come to the aid of their party.
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 \texttt{BAAAAAAA}.  
• Suppose we match 5 chars in pattern, with mismatch on 6\textsuperscript{th} char.  
• We know previous 6 chars in text are \texttt{BAAAAA}.  
• 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.

**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>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 $dfa[c][j]$
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 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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>6</td>
</tr>
</tbody>
</table>

Diagram:

```
0 -- A, C -> 1 -> 2 -- A -> 3 -- B -> 4 -- A -> 5 -- C -> 6
```

B, C
DFA simulation

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

```
dfa[][][]j   |    0  1  3  1  5  1
A  B   0  2  3  1  5  0
B  2  0  4  0  4  4
C  0  0  0  0  6  6
```
DFA simulation

\[
\begin{array}{ccccccc}
A & B & A & B & A & B & A \\
\end{array}
\]

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

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

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

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

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

<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>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>

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

\[
daF[][j] \\
\begin{array}{ccccccc}
A & B & A & B & A & C &   \\
B & 0 & 2 & 0 & 4 & 0 & 4 \\
C & 0 & 0 & 0 & 0 & 0 & 6 \\
\end{array}
\]

[Diagram of DFA with states labeled 0 to 6 and transitions marked with 'A', 'B', or 'C', and possibly 'B, C'.]
DFA simulation

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

A A B A C A A B A B A C A A 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>

B, C
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

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

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

dfa[][]

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

Diagram of DFA states and transitions.
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     1 1 3 1 5 1
A               2 0 4 0 4
B               0 0 6
C               0 0 0 0 0
```

```java
void simulateDFA(String pattern) {
    int[] dfa = ...
    for (int j = 0; j < pattern.length(); j++) {
        pat.charAt(j) = ...;
        int nextState = dfa[j][pat.charAt(j)];
        // Transition to next state
    }
}
```
DFA simulation

\[
\begin{array}{cccccc}
A & A & B & A & C & A \\
A & B & A & B & A & C \\
\end{array}
\]

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

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

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

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

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

Diagram of DFA simulation with transitions for inputs A, B, C.
DFA simulation

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

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

<table>
<thead>
<tr>
<th>$\text{txt}$</th>
<th>$\text{pat}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7 8</td>
<td>0 1 2 3 4 5</td>
</tr>
<tr>
<td>B C B A A B A C A</td>
<td>A B A B A C</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>State</th>
<th>Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>B, C</td>
</tr>
<tr>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td>A</td>
</tr>
<tr>
<td>3</td>
<td>B</td>
</tr>
<tr>
<td>4</td>
<td>A</td>
</tr>
<tr>
<td>5</td>
<td>C</td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

- length of longest prefix of $\text{pat}[]$ that is a suffix of $\text{txt}[0..i]$
- suffix of $\text{txt}[0..6]$
- prefix of $\text{pat}[]$
**Knuth-Morris-Pratt substring search: Java implementation**

**Key differences from brute-force implementation.**

- Need to precompute $\text{dfa}[][]$ 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;
}
```

**Running time.**

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

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

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

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

Constructing the DFA for KMP substring search for 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></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( \text{dfa[][]}(j) )</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>B</td>
<td></td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td>6</td>
</tr>
</tbody>
</table>

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

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

### Knuth-Morris-Pratt construction

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

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

<table>
<thead>
<tr>
<th>$\text{dfa[][]}[j]$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B$</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$A$</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$C$</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

0 $\rightarrow$ A $\rightarrow$ B $\rightarrow$ 2 $\rightarrow$ A $\rightarrow$ 3 $\rightarrow$ B $\rightarrow$ 4 $\rightarrow$ A $\rightarrow$ 5 $\rightarrow$ C $\rightarrow$ 6
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
Knuth-Morris-Pratt construction

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

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

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

- **State 0**: Start state, transitions to 1 on A, B, C.
- **State 1**: On input A, move to 2, on B move to 3, on C move to 6.
- **State 2**: On input A, move to 3, on B move to 4, on C move to 6.
- **State 3**: On input B, move to 4, on A move to 5, on C move to 6.
- **State 4**: On input A, move to 5, on B move to 6, on C move to 6.
- **State 5**: On input C, move to 6, on A move to 5.
- **State 6**: Final accepting state.
Knuth-Morris-Pratt construction

Mismatch transition: back up if c != pat.charAt(j).

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

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

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

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

```
A
B, C
```

```
A
B
```
How to build DFA from pattern?

Include one state for each character in pattern (plus accept state).
**How to build DFA from pattern?**

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

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

<table>
<thead>
<tr>
<th>( \text{pat.charAt}(j) )</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td>B</td>
<td></td>
<td>B</td>
<td>A</td>
<td>C</td>
</tr>
<tr>
<td>A</td>
<td>1</td>
<td></td>
<td>3</td>
<td></td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>( \text{dfa[]}[j] )</td>
<td>B</td>
<td></td>
<td>2</td>
<td></td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6</td>
</tr>
</tbody>
</table>

Diagram:

0 \(\rightarrow\) A \(\rightarrow\) 1 \(\rightarrow\) B \(\rightarrow\) 2 \(\rightarrow\) A \(\rightarrow\) 3 \(\rightarrow\) B \(\rightarrow\) 4 \(\rightarrow\) A \(\rightarrow\) 5 \(\rightarrow\) C \(\rightarrow\) 6
How to build DFA from pattern?

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

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

Running time. Seems to require $j$ steps.

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

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

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

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

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

Running time. Takes only constant time if we maintain state \( X \).

Ex. \( \text{dfa}['A'][5] = 1; \) from state \( X \), take transition 'A' = \( \text{dfa}['A'][X] \); 
\( \text{dfa}['B'][5] = 4; \) from state \( X \), take transition 'B' = \( \text{dfa}['B'][X] \); 
\( \cdot X' = 0 \) from state \( X \), take transition 'C' = \( \text{dfa}['C'][X] \).
Knuth-Morris-Pratt construction (in linear time)

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

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

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

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

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

<table>
<thead>
<tr>
<th>( \text{dfa[][][j]} )</th>
<th>1</th>
<th>3</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( B )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( C )</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

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

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

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

Knuth-Morris-Pratt construction (in linear time)

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

<table>
<thead>
<tr>
<th>( \text{pat.charAt}(j) )</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td></td>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td>1</td>
<td>1</td>
<td>3</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></td>
<td></td>
<td></td>
<td>6</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 $X = \text{dfa[pat.charAt(j)]}[X]$.

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

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

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

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

A B, C A 1 B 2 A 3 B 4 A 5 C 6
Constructing the DFA for KMP substring search: Java implementation

For each state $j$:

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

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

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

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

**Pf.** Each pattern 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 $\text{nfa}[]$ in time and space proportional to $M$.
Knuth-Morris-Pratt: brief history

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

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

\[
\begin{array}{cccccccccccccccccccccccc}
0 & 5 & \text{NEEDE} & \text{L} & \text{E} & \text{pattern} \\
5 & 5 & \text{NEEDE} & \text{L} & \text{E} \\
11 & 4 & \text{NEEDE} & \text{L} & \text{E} \\
15 & 0 & \text{NEEDE} & \text{L} & \text{E} \\
\end{array}
\]

\text{return } i = 15
**Boyer-Moore: mismatched character heuristic**

**Q.** How much to skip?

**Case 1.** Mismatch character not in pattern.

*before*

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

*i*

*after*

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

mismatch character 'T' not in pattern: increment i one character beyond 'T'
Q. How much to skip?

Case 2a. Mismatch character in pattern.

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

<table>
<thead>
<tr>
<th>txt</th>
<th>. . . . . . E L E . . . . . .</th>
</tr>
</thead>
<tbody>
<tr>
<td>pat</td>
<td>N E E D L E</td>
</tr>
</tbody>
</table>

Aligned with rightmost E?

<table>
<thead>
<tr>
<th>txt</th>
<th>. . . . . . E L E . . . . . .</th>
</tr>
</thead>
<tbody>
<tr>
<td>pat</td>
<td>N E E D L E</td>
</tr>
</tbody>
</table>

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).

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

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

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

**Property.** Substring search with the Boyer-Moore mismatched character heuristic takes about $\sim \frac{N}{M}$ character compares to search for a pattern of length $M$ in a text of length $N$.

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

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

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

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

Basic idea = modular hashing.

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

<table>
<thead>
<tr>
<th>$i$</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>$i$</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>txt.charAt(i)</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>pat.charAt(i)</td>
<td>0</td>
<td>3</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>5</td>
<td>2</td>
<td>6</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>% 997 = 613</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</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>% 997 = 201</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</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>% 997 = 715</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</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>% 997 = 442</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>5</td>
<td>9</td>
<td>2</td>
<td>6</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>% 997 = 929</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>9</td>
<td>2</td>
<td>6</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>% 997 = 613</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>return i = 6</td>
<td>2</td>
<td>6</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>% 997 = 613</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$\text{match}$
Efficiently computing the hash function

Modular hash function. Using the notation \( t_i \) for \( \text{txt}.\text{charAt}(i) \), we wish to compute

\[
\cdot x_i = t_i R^{M-1} + t_{i+1} R^{M-2} + \ldots + t_{i+M-1} R^{0} \pmod{Q}
\]

Intuition. \( M \)-digit, base-\( R \) integer, modulo \( Q \).

Horner's method. Linear-time method to evaluate degree-\( M \) polynomial.

\begin{verbatim}
// 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;
}
\end{verbatim}

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

\begin{align*}
    0 & \quad \% 997 = 2 \\
    1 & \quad 2 \times 10 + 6 \% 997 = 26 \\
    2 & \quad 26 \times 10 + 5 \% 997 = 265 \\
    3 & \quad 265 \times 10 + 3 \% 997 = 659 \\
    4 & \quad 659 \times 10 + 5 \% 997 = 613
\end{align*}
**Challenge.** How to efficiently compute $x_{i+1}$ given that we know $x_i$.

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

**Key property.** Can update hash function in constant time!

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

---

**Table:**

<table>
<thead>
<tr>
<th>i</th>
<th>...</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>
<td>5</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>
<td></td>
</tr>
</tbody>
</table>

Text:

- Current value
- Subtract leading digit
- Multiply by radix
- Add new trailing digit
- (can precompute $R^{M-2}$)
Rabin-Karp substring search example

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

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

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

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

Theory. If $Q$ is a sufficiently large random prime (about $M N^2$), 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>backup in input?</th>
<th>correct?</th>
<th>extra space</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>guarantee</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>brute force</td>
<td>—</td>
<td>$MN$</td>
<td>yes</td>
<td>yes</td>
<td>1</td>
</tr>
<tr>
<td>Knuth-Morris-Pratt</td>
<td>full DFA (Algorithm 5.6)</td>
<td>$2N$</td>
<td>no</td>
<td>yes</td>
<td>MR</td>
</tr>
<tr>
<td></td>
<td>mismatch transitions only</td>
<td>$3N$</td>
<td>no</td>
<td>yes</td>
<td>$M$</td>
</tr>
<tr>
<td>Boyer-Moore</td>
<td>full algorithm</td>
<td>$3N$</td>
<td>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>$7N$</td>
<td>no</td>
<td>yes†</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Las Vegas</td>
<td>$7N$ †</td>
<td>yes</td>
<td>yes</td>
<td>1</td>
</tr>
</tbody>
</table>

† probabilistic guarantee, with uniform hash function
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 \text{CGG} or \text{AGG}, bracketed by \text{GCG} at the beginning and \text{CTG} at the end.
- Number of repeats is variable, and correlated with syndrome.

\begin{align*}
\text{pattern} & \quad \text{GCG (CGG | AGG) *CTG} \\
\text{text} & \quad \text{GCGCGTGCTGTGCCGAGAGATGGTTTAAGCTGGCGCGGAGGCGGCTGGCGCAGGCTG}
\end{align*}
public class NFA {

    private Digraph G;    // digraph of epsilon transitions
    private String regexp; // regular expression
    private int M;        // number of characters in regular expression

    // Create the NFA for the given RE
    public NFA(String regexp) {
        this.regexp = regexp;
        M = regexp.length();
        Stack<Integer> ops = new Stack<Integer>();
        G = new Digraph(M+1);
    }
}
Google code search

![Search public source code](http://code.google.com/p/chromium/source/search)

<table>
<thead>
<tr>
<th>Search Options</th>
<th>In Search Box</th>
</tr>
</thead>
<tbody>
<tr>
<td>Package</td>
<td>package:linux-2.6</td>
</tr>
<tr>
<td>Language</td>
<td>lang:c++</td>
</tr>
<tr>
<td>File Path</td>
<td>file:(code)</td>
</tr>
<tr>
<td>Class</td>
<td>class:HashMap</td>
</tr>
<tr>
<td>Function</td>
<td>function:toString</td>
</tr>
<tr>
<td>License</td>
<td>license:mozilla</td>
</tr>
<tr>
<td>Case Sensitive</td>
<td>case:yes</td>
</tr>
</tbody>
</table>
Pattern matching: applications

Test if a string matches some pattern.
• Process natural language.
• Scan for virus signatures.
• Specify a programming language.
• Access information in digital libraries.
• 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.
  ...
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>AA ABBBBBBBBBBA</td>
<td>AB ABABA</td>
</tr>
<tr>
<td>parentheses</td>
<td>1</td>
<td>A (A</td>
<td>B) AAB</td>
<td>AAAAB ABABAB</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(AB) *A</td>
<td>A ABABABABABA</td>
<td>AA ABBA</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</td>
<td>SUCCUBUS TUMULTUOUS</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>ADE BCDE</td>
</tr>
<tr>
<td>exactly k</td>
<td>[0-9]{5}−[0-9]{4}</td>
<td>08540−1321 19072−5541</td>
<td>111111111 166−54−111</td>
</tr>
<tr>
<td>complement</td>
<td>[^AEIOU]{6}</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 CRISPBREAD</td>
<td>SUBSPACE SUBSPECIES</td>
</tr>
<tr>
<td>(substring search)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[0-9]{3}-[0-9]{2}-[0-9]{4}</td>
<td>166-11-4433 166-45-1111</td>
<td>11-55555555 8675309</td>
</tr>
<tr>
<td>(Social Security numbers)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[a-z]+@[a-z]+.(edu</td>
<td>com)</td>
<td><a href="mailto:wayne@princeton.edu">wayne@princeton.edu</a> <a href="mailto:rs@princeton.edu">rs@princeton.edu</a></td>
</tr>
<tr>
<td>(email addresses)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[$_A-Za-z][$_A-Za-z0-9]*</td>
<td>ident3 PatternMatcher</td>
<td>3a ident#3</td>
</tr>
<tr>
<td>(Java identifiers)</td>
<td></td>
<td></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.
Regular expressions to the rescue

Whenever I learn a new skill I concoct elaborate fantasy scenarios where it lets me save the day.

Oh no! The killer must have followed her on vacation!

But to find them we’d have to search through 200 MB of emails looking for something formatted like an address!

It’s hopeless!

Everybody stand back.

I know regular expressions.

http://xkcd.com/208
Perl RE for valid RFC822 email addresses

http://www.ex-parrot.com/~pdw/Mail-RFC822-Address.html
Regular expression caveat

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.
REs and NFAs
NFA simulation
NFA construction
Applications
RE. Concise way to describe a set of strings.

DFA. Machine to recognize whether a given string is in a given set.

Kleene's theorem.
- For any DFA, there exists a RE that describes the same set of strings.
- For any RE, there exists a DFA that recognizes the same set of strings.

**RE** 0* | (0*10*10*10*)*

**DFA**

Stephen Kleene
Princeton Ph.D. 1934
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 (start = 0, accept = M).
- Red $\epsilon$-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.

![Diagram of NFA corresponding to the pattern ( ( A * B | A C ) D )](image-url)
Q. Is $\text{AAAAABD}$ matched by NFA?
A. Yes, because some sequence of legal transitions ends in state 11.
Q. Is $\text{AAAAABD}$ matched by NFA?
A. Yes, because some sequence of legal transitions ends in state 11.
   [ even though some sequences end in wrong state or stall ]

NFA corresponding to the pattern $((A \ast B \mid A C)D)$
**Q.** Is $\text{AAAC}$ matched by NFA?

**A.** No, because no sequence of legal transitions ends in state 11.

[ but need to argue about all possible sequences ]

Nondeterministic finite-state automata

NFA corresponding to the pattern \(( ( A \ast B \mid A \ C ) D )\)
Q. How to determine whether a string is matched by an automaton?

DFA. Deterministic $\Rightarrow$ exactly one applicable transition.

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

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

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

- REs and NFAs
- NFA simulation
- NFA construction
- Applications
**NFA representation**

**State names.** Integers from 0 to $M$.

**Match-transitions.** Keep regular expression in array `re[]`.

**$\epsilon$-transitions.** Store in a digraph $G$.

- $0 \to 1$, $1 \to 2$, $1 \to 6$, $2 \to 3$, $3 \to 2$, $3 \to 4$, $5 \to 8$, $8 \to 9$, $10 \to 11$

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

Q. How to perform reachability?
Goal. Check whether input matches pattern.

NFA corresponding to the pattern \(( ( A \ast B \mid A C ) D )\)
Read next input character.

- Find states reachable by match transitions.
- Find states reachable by ε-transitions
NFA simulation

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

set of states reachable via $\varepsilon$–transitions from start
NFA simulation

Read next input character.

- Find states reachable by match transitions.
- Find states reachable by $\varepsilon$-transitions

set of states reachable via $\varepsilon$-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 \( \varepsilon \)-transitions

![Diagram of NFA simulation]

set of states reachable after matching A
Read next input character.

- Find states reachable by match transitions.
- Find states reachable by $\varepsilon$-transitions

set of states reachable after matching $A$: \{3, 7\}
Read next input character.

- Find states reachable by match transitions.
- Find states reachable by $\varepsilon$-transitions

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

Read next input character.

• Find states reachable by match transitions.
• Find states reachable by \( \varepsilon \)-transitions

set of states reachable via \( \varepsilon \)-transitions after matching A: \{ 2, 3, 4, 7 \}
Read next input character.

- Find states reachable by match transitions.
- Find states reachable by $\varepsilon$-transitions

set of states reachable after matching A A
Read next input character.

- Find states reachable by match transitions.
- Find states reachable by $\varepsilon$-transitions

\[
\begin{array}{cccc}
\text{input} & A & A & B & D \\
\end{array}
\]

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

input

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 $\varepsilon$-transitions

set of states reachable via $\varepsilon$-transitions after matching A A : \{ 2, 3, 4 \}
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 A B
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 B: \{ 5 \}
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 B
NFA simulation

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

set of states reachable via $\varepsilon$–transitions after matching A A B : { 5, 8, 9 }
Read next input character.

- Find states reachable by match transitions.
- Find states reachable by $\varepsilon$-transitions

**NFA simulation**

The input is `A A B D`.

The set of states reachable after matching `A A B D` is shown in the diagram.

The match D transition is indicated by the arrow pointing from state 9 to state 10.
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 D : \{ 10 \}
Read next input character.

- Find states reachable by match transitions.
- Find states reachable by $\varepsilon$-transitions

NFA simulation

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

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

set of states reachable via $\varepsilon$–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

set of states reachable: \{ 10, 11 \}
## Digraph reachability

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

<table>
<thead>
<tr>
<th>public class DirectedDFS</th>
</tr>
</thead>
<tbody>
<tr>
<td>DirectedDFS(Digraph G, int s)</td>
</tr>
<tr>
<td>DirectedDFS(Digraph G, Iterable&lt;Integer&gt; s)</td>
</tr>
<tr>
<td>boolean marked(int v)</td>
</tr>
</tbody>
</table>

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

**Performance.** Runs in time proportional to $E + V$. 
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 = buildEpsilonTransitionsDigraph();
    }

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

    public Digraph buildEpsilonTransitionDigraph()
    { /* stay tuned */ }
}
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 $\leq 3M$.]

![NFA diagram](image)

NFA corresponding to the pattern \(( ( A \ast B \mid A C ) D )\)
REs and NFAs
NFA simulation
NFA construction
Applications
Building an NFA corresponding to an RE

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

![NFA diagram](image)

NFA corresponding to the pattern \( ( ( A \ast B | A C ) D ) \)
**Building an NFA corresponding to an RE**

**Concatenation.** Add match-transition edge from state corresponding to characters in the alphabet to next state.

**Alphabet.**  A  B  C  D  

**Metacharacters.**  ( )  .  *  |  

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

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

NFA corresponding to the pattern \(( ( A * B | A C ) D )\)
Building an NFA corresponding to an RE

**Closure.** Add three \( \varepsilon \)-transition edges for each \( * \) operator.

\[
\begin{align*}
\text{single-character closure} & \quad \text{closure expression} \\
& \quad \text{G.addEdge}(i, i+1); \\
& \quad \text{G.addEdge}(i+1, i); \\
& \quad \text{G.addEdge}(lp, i+1); \\
& \quad \text{G.addEdge}(i+1, lp);
\end{align*}
\]

NFA corresponding to the pattern \(((A*B|A*C)D)\)
**Building an NFA corresponding to an RE**

**Or.** Add two $\varepsilon$-transition edges for each | operator.

![Diagram of NFA construction rules](image)

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

(((A*B|A*C)*D))
Left parenthesis.

- Add $\varepsilon$-transition to next state.
- Push index of state corresponding to ( onto stack.
Left parenthesis.

- Add $\varepsilon$-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 \( \varepsilon \)-transitions if next character is \(*\).
NFA construction

Alphabet symbol.

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

```plaintext
(( A * B | A C ) D )
```

Stack:
- 1
- 0
Closure symbol.

- Add $\varepsilon$-transition to next state.
NFA construction

Alphabet symbol.

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

\[
( ( A \ast B | A C ) D )
\]
NFA construction

Or symbol.

- Push index of state corresponding to $|$ onto stack.
NFA construction

Alphabet symbol.

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

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

$( ( ( A \ast B \mid A C ) D )$
Right parenthesis.

- Add $\varepsilon$-transition to next state.
- Pop corresponding ( and possibly intervening |; add $\varepsilon$-transition edges for or.
- Do one-character lookahead:
  add $\varepsilon$-transitions if next character is *.
NFA construction

Alphabet symbol.

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

$((A*B|A*C)D)$
NFA construction

Right parenthesis.

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

(stack)

(((A*B|A*C)*D))
End of regular expression.

- Add accept state.
NFA construction

NFA corresponding to the pattern \(( ( A \ast B \mid A C ) D )\)
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);
        }

        if (re[i] == '(' || re[i] == '*' || re[i] == ')')
            G.addEdge(i, i+1);
    }
    return G;
}
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.
REGULAR EXPRESSIONS

- REs and NFAs
- NFA simulation
- NFA construction
- Applications
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

```
% more words.txt
a
aback
abacus
abalone
abandon
...
% grep "s..ict.." words.txt
constrictor
stricter
stricture
```

dictionary (standard in Unix)
also on booksite
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>` ?

```plaintext
<brink>text</blink>some text<brink>more text</blink>
```

reluctant  greedy  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, ...

% grep 'NEWLINE' */*.java ← print all lines containing \texttt{NEWLINE} which occurs in any file with a .java extension

% egrep '^[qwertyuiop]*[zxcvbnm]*$' words.txt | egrep '.............'

typewritten

**PERL.** Practical Extraction and Report Language.

% perl -p -i -e 's|from|to|g' input.txt ← replace all occurrences of from with to in the file input.txt

% perl -n -e 'print if /^[A-Z][A-Za-z]*$/' words.txt ← print all words that start with uppercase letter

do for each line
Validity checking. Does the input match the regexp?

Java string library. Use `input.matches(regexp)` for basic RE matching.

```java
class Validate {
    public static void main(String[] args) {
        String regexp = args[0];
        String input  = args[1];
        StdOut.println(input.matches(regexp));
    }
}
```

% java Validate "[$_A-Za-z][$_A-Za-z0-9]*" ident123
true

% java Validate "[a-z]+@[a-z]+.(edu|com)" rs@cs.princeton.edu
true

% java Validate "[0-9]{3}-[0-9]{2}-[0-9]{4}" 166-11-4433
true
**Harvesting information**

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

```plaintext
% java Harvester "gcg(cgg|agg)*ctg" chromosomeX.txt
gcgcggcggcggcgcggctg
gcgctg
gcgctg
gcgcggcggcggaggcggaggcggctg

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

Harvest patterns from DNA

Harvest links from website
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());
        }
    }
}
```

- `compile()` creates a `Pattern` (NFA) from RE
- `matcher()` creates a `Matcher` (NFA simulator) from NFA and text
- `find()` looks for the next match
- `group()` returns the substring most recently found by `find()`
Algorithmic complexity attacks

**Warning.** Typical implementations do not guarantee performance!

Unix grep, Java, Perl

```
% java Validate "(a|aa)*b" aaaaaaaaaaaaaaaaaaaaaaaaaaaaaac 1.6 seconds
% java Validate "(a|aa)*b" aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaac 3.7 seconds
% java Validate "(a|aa)*b" aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaac 9.7 seconds
% java Validate "(a|aa)*b" aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaac 23.2 seconds
% java Validate "(a|aa)*b" aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaac 62.2 seconds
% java Validate "(a|aa)*b" aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaac 161.6 seconds
```

SpamAssassin regular expression.

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

- Takes exponential time on pathological email addresses.
- Troublemaker can use such addresses to DOS a mail server.
Not-so-regular expressions

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

\begin{verbatim}
(.+)'\1  // beriberi couscous
1?$|^(11+?)'\1+     // 1111 111111 111111111
\end{verbatim}

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: attcggaat.

Remark. Pattern matching with back-references is intractable.
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 \(\Rightarrow\) DFA.
• grep \(\Rightarrow\) NFA.
• javac Java language \(\Rightarrow\) Java byte code.

<table>
<thead>
<tr>
<th></th>
<th>KMP</th>
<th>grep</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>pattern</td>
<td>string</td>
<td>RE</td>
<td>program</td>
</tr>
<tr>
<td>parser</td>
<td>unnecessary</td>
<td>check if legal</td>
<td>check if legal</td>
</tr>
<tr>
<td>compiler output</td>
<td>DFA</td>
<td>NFA</td>
<td>byte code</td>
</tr>
<tr>
<td>simulator</td>
<td>DFA simulator</td>
<td>NFA simulator</td>
<td>JVM</td>
</tr>
</tbody>
</table>
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.