**BBM 202 - ALGORITHMS**

HACETTEPE UNIVERSITY

DEPT. OF COMPUTER ENGINEERING

ERKUT ERDEM

**SUBSTRING SEARCH**

Apr. 29, 2014

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

---

**TODAY**

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

---

**Substring search**

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

Typically $N \gg M$

```plaintext
<table>
<thead>
<tr>
<th>Pattern</th>
<th>Needle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Text</td>
<td>Inahaystackneedleina</td>
</tr>
</tbody>
</table>
```

```plaintext
match
```

---

**Substring search applications**

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

Typically $N \gg M$

```plaintext
<table>
<thead>
<tr>
<th>Pattern</th>
<th>Needle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Text</td>
<td>Inahaystackneedleina</td>
</tr>
</tbody>
</table>
```

```plaintext
match
```

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

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”

“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_tabledata1">
    452.92
  </td>
</tr>
```

```
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

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.
## Substring Search

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

### Brute-force substring search

Check for pattern starting at each text position.

<table>
<thead>
<tr>
<th>i</th>
<th>j</th>
<th>i+j</th>
<th>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>txt</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>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>2</td>
<td>pat</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>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td>A</td>
<td>B</td>
<td>R</td>
<td></td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>3</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>3</td>
<td>0</td>
<td>3</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>4</td>
<td>1</td>
<td>5</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>5</td>
<td>0</td>
<td>5</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>6</td>
<td>4</td>
<td>10</td>
<td>return i when j is M</td>
<td></td>
<td>A</td>
<td>B</td>
<td>R</td>
<td>A</td>
<td>A</td>
<td>B</td>
<td>R</td>
<td>A</td>
<td>A</td>
</tr>
</tbody>
</table>

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

### Brute-force substring search: Java implementation

Check for pattern starting at each text position.

```java
public static int search(String pat, String txt) {
    int M = pat.length();
    int N = txt.length();
    for (int i = 0; i <= N - M; i++) {
        int j;
        for (j = 0; j < M; j++)
            if (txt.charAt(i+j) != pat.charAt(j))
                break;
        if (j == M) return i;
    }
    return N; // not found
}
```

### Brute-force substring search: worst case

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

<table>
<thead>
<tr>
<th>i</th>
<th>j</th>
<th>i+j</th>
<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></td>
<td></td>
<td></td>
<td>txt</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>4</td>
<td>4</td>
<td>pat</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>5</td>
<td></td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>6</td>
<td></td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>7</td>
<td></td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>8</td>
<td></td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>10</td>
<td>return i when j is M</td>
<td></td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
</tbody>
</table>

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

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

Brute-force algorithm needs backup for every mismatch.

**Approach 1.** Maintain buffer of last $M$ characters.
**Approach 2.** Stay tuned.

**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).

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

**Algorithmic challenges in substring search**

Brute-force is not always good enough.

**Theoretical challenge.** Linear-time guarantee.

**Practical challenge.** Avoid backup in text stream.
Knuth-Morris-Pratt substring search

Intuition. Suppose we are searching in text for pattern \texttt{BAAAAAAAAA}.
- Suppose we match 5 chars in pattern, with mismatch on 6th 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. (!)

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.

Constructing the DFA for KMP substring search for \texttt{A B A A A A B A A A A A A A A A A A B A}.

DFA simulation

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

DFA simulation

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

A A B A C A A

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

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

String 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>$i$</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>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{txt}$</td>
<td>B</td>
<td>C</td>
<td>B</td>
<td>A</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>C</td>
</tr>
</tbody>
</table>

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

**Interpretation of Knuth-Morris-Pratt DFA**

**Knuth-Morris-Pratt substring search: Java implementation**

Key differences from brute-force implementation.
- Need to precompute $\text{dfa}[\cdot][\cdot]$ 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 NOT_FOUND;
}
```

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

**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 $j+1$.

1. first $j$ characters of pattern have already been matched
2. next char matches
3. now first $j+1$ characters of pattern have been matched

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

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

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
How to build DFA from pattern?

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

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

\[\begin{align*}
0 & \rightarrow 1 & 2 & 3 & 4 & 5 \\
A & \rightarrow B & A & B & A & C \\
\end{align*}\]

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. Starts under construction (!)

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

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

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

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

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

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

0 1 2 3 4 5

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

\[ \text{first } j \text{ characters of pattern have already been matched} \]

\[ \text{now first } j+1 \text{ characters of pattern have been matched} \]

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

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

0 1 2 3 4 5

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

\[ \text{X} = \text{simulation of empty string} \]

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

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

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

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

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

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

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

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

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

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

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

Knuth-Morris-Pratt construction (in linear time)

Constructing the DFA for KMP substring search: Java implementation

For each state \( j \):
- Copy \( \text{dfa}[j][X] \) to \( \text{dfa}[j][j] \) for mismatch case.
- Set \( \text{dfa}[	ext{pat.charAt}(j)][j] \) to \( j+1 \) for match case.
- Update \( X \).

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

Running time. \( M \) character accesses (but space proportional to \( RM \)).

Knuth-Morris-Pratt construction (in linear time)

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.

**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.

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

<table>
<thead>
<tr>
<th>i</th>
<th>j</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>18</td>
<td>19</td>
</tr>
<tr>
<td>20</td>
<td>21</td>
</tr>
<tr>
<td>22</td>
<td>23</td>
</tr>
</tbody>
</table>

**Substring Search**

- Brute force
- Knuth-Morris-Pratt
- Boyer-Moore
- Rabin-Karp
Boyer-Moore: mismatched character heuristic

Q. How much to skip?

Case 2a. Mismatch character in pattern.

![Diagram](image1)

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

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

![Diagram](image2)

Mismatch character 'E' in pattern: increment i by 1

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

$$x_i = t_i R^{M-1} + t_{i+1} R^{M-2} + \ldots + t_{i+M-1} R^0 \quad (\text{mod } Q)$$

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

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

```
public long hash(String key, int M) {
    long h = 0;
    for (int j = 0; j < M; j++)
        h = (R * h + key.charAt(j)) % Q;
    return h;
}
```

• $x_i = t_i R^{M-1} + t_{i+1} R^{M-2} + \ldots + t_{i+M-1} R^0 \quad (\text{mod } Q)$

Efficiently computing the hash function

```java
public class RabinKarp {
    private long patHash; // pattern hash value
    private int M; // pattern length
    private long Q; // modulus
    private int R; // radix
    private long RM; // $R^{(M-1)} \text{ (mod } Q)$

    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 */
    }
}
```
### Rabin-Karp: Java implementation (continued)

#### Monte Carlo version

Return match if hash match.

```java
class RabinKarp {
  private int M, Q, R;
  private char[] txt, pat;
  private int txtHash, patHash, i, j;

  public int search(String txt) {
    this.txt = txt.toCharArray();
    this.M = this.txt.length();
    this.R = (int) Math.pow(this.R, this.M - 1) % this.Q;
    this.Q = this.txt[0];
    this.pat = pat.toCharArray();
    this.patHash = 0;
    this.i = 0;
    for (j = 0; j < this.M; j++)
      this.patHash = (this.patHash * this.R + this.pat[j]) % this.Q;
    for (i = 0; i < this.M - this.N; i++)
      this.txtHash = (this.txtHash * this.R + this.txt[i]) % this.Q;
    for (i = this.M - this.N; i < this.M; i++)
      this.txtHash = (this.txtHash - this.R * this.txt[i - this.M] % this.Q) % this.Q;
    for (i = this.M - this.N; i < this.M; i++)
      if (this.patHash == this.txtHash)
        return i - this.N + 1;
    return -1;
  }
}
```

#### Las Vegas version

Check for substring match if hash match; continue search if false collision.

---

### 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 guarantee</th>
<th>Backup in input?</th>
<th>Correct?</th>
<th>Extra space</th>
</tr>
</thead>
<tbody>
<tr>
<td>brute force</td>
<td>—</td>
<td>MN</td>
<td>yes</td>
<td>yes</td>
<td>1</td>
</tr>
<tr>
<td>Knuth-Morris-Pratt</td>
<td>fast 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>fast algorithm</td>
<td>MN</td>
<td>yes</td>
<td>yes</td>
<td>R</td>
</tr>
<tr>
<td></td>
<td>mismatched characteristic only (Algorithm 5.7)</td>
<td>MN</td>
<td>yes</td>
<td>yes</td>
<td>R</td>
</tr>
<tr>
<td>Rabin-Karp†</td>
<td>Las Vegas (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.