5.3 SUBSTRING SEARCH

- introduction
- brute force
- Knuth-Morris-Pratt
- Boyer-Moore
- Rabin-Karp
5.3 Substring Search

- Introduction
- Brute Force
- Knuth-Morris-Pratt
- Boyer-Moore
- Rabin-Karp
**Substring search**

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

- **pattern** → NEEDLE
- **text** → INAHAYSTACK NEEDLE INA

*match*
Substring search applications

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

$\text{pattern} \rightarrow \text{NEEDEL}$

$\text{text} \rightarrow \text{INAHAYSTACK} \text{NEEDEL} \text{INA}$

typically $N \gg M$

**match**
Substring search applications

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

(pattern $\rightarrow$ NEEDLE)

(text $\rightarrow$ INA HAY STACK NEEDLE INA)

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

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

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

 Typically $N \gg M$

<table>
<thead>
<tr>
<th>pattern</th>
<th>NEEDLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>text</td>
<td>INAHAYSTACK NEEDLE INA</td>
</tr>
</tbody>
</table>

Identify patterns indicative of spam.

- PROFITS
- LOSE WEIGHT
- herbal Viagra
- 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:

```
...<tr><td class="yfnc_tablehead1" width="48%">
 Last Trade:
</td><td class="yfnc_tabledata1">452.92</td><td class="yfnc_tabledata1">
Trade Time:
</td></tr>
```

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

- introduction
- brute force
- Knuth-Morris-Pratt
- Boyer-Moore
- Rabin-Karp
5.3 SUBSTRING SEARCH

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

Check for pattern starting at each text position.

<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>C</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>2</td>
<td></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>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>

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.

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

A B A C A D A B R A C

4 3 7
5 0 5

```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; // index in text where pattern starts
    }
    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>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>A</td>
<td>A</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>4</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A B</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A B</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A B</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A B</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A B</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 # 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;
}
```

explicit backup

<table>
<thead>
<tr>
<th>i</th>
<th>j</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>9</td>
<td>10</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>7</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Algorithmic challenges in substring search

Brute-force is not always good enough.

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

**Practical challenge.** Avoid backup in text stream.
5.3 Substring Search

- introduction
- brute force
- Knuth-Morris-Pratt
- Boyer-Moore
- Rabin-Karp
5.3 SUBSTRING SEARCH

- Introduction
- Brute Force
- Knuth-Morris-Pratt
- Boyer-Moore
- Rabin-Karp
Knuth-Morris-Pratt substring search

**Intuition.** Suppose we are searching in text for pattern BAAAAAAA.
- Suppose we match 5 chars in pattern, with mismatch on 6th char.
- We know previous 6 chars in text are BAAAAB.
- Don't need to back up text pointer!

Knuth-Morris-Pratt algorithm. Clever method to always avoid backup. (!)
Deterministic finite state automaton (DFA)

DFA is abstract string-searching machine.

- Finite number of states (including start and halt).
- Exactly one transition for each char in alphabet.
- Accept if sequence of transitions leads to halt state.

**Internal representation**

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

If in state j reading char C:
if j is 6 halt and accept
else move to state dfa[c][j]

**Graphical representation**

- States: 0, 1, 2, 3, 4, 5, 6
- Transitions:
  - 0 → 1: A, B, C
  - 1 → 2: A
  - 2 → 3: A
  - 3 → 4: B
  - 4 → 5: A
  - 5 → 6: C
  - 0 → 0: A, B, C
DFA simulation demo

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 demo

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

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>A</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>C</td>
</tr>
<tr>
<td>A</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>5</td>
<td>1</td>
</tr>
</tbody>
</table>

dfa[][]

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

substring found
Interpretation of Knuth-Morris-Pratt DFA

Q. What is interpretation of DFA state after reading in \( \text{txt}[i] \)?
A. State = number of characters in pattern that have been matched.

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

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

Prefix of \( \text{pat}[] \)

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

Suffix of \( \text{txt}[0..6] \)
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.

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

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

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

dfa[j] = pat.charAt(j)

no backup
Knuth-Morris-Pratt construction demo

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 demo

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

DFA

0 → A → 1 → B → 2 → A → 3 → B → 4 → A → 5 → C → 6
How to build DFA from pattern?

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

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

Running time. Seems to require \( j \) steps.

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

- simulate BABA; take transition 'A' \( \Rightarrow \texttt{dfa['A'][3]} \)
- simulate BABA; take transition 'B' \( \Rightarrow \texttt{dfa['B'][3]} \)
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. Takes only constant time if we maintain state $X$.

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

from state $X$, take transition 'A'
= $\text{dfa['A']}[X]$

from state $X$, take transition 'B'
= $\text{dfa['B']}[X]$

from state $X$, take transition 'C'
= $\text{dfa['C']}[X]$

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

\[ 0 \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \]

\[ A \quad B \quad A \quad B \quad A \quad C \]
Knuth-Morris-Pratt construction demo (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

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

0 1 2 3 4 5 6
Knuth-Morris-Pratt construction demo (in linear time)

Constructing the DFA for KMP substring search for A B A B A C
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 $\text{dfa}[	ext{pat.charAt}(j)][j]$ 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/time proportional to $RM$).
KMP substring search analysis

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

**Pf.** Each pattern char accessed once when constructing the DFA; each text char accessed once (in the worst case) when simulating the DFA.

**Proposition.** KMP constructs $\text{dfa}[][]$ in time and space proportional to $R M$.

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

![KMP NFA for ABABAC](image)
Knuth-Morris-Pratt: brief history

- Independently discovered by two theoreticians and a hacker.
  - Knuth: inspired by esoteric theorem, discovered linear algorithm
  - Pratt: made running time independent of alphabet size
  - Morris: built a text editor for the CDC 6400 computer
- Theory meets practice.

SIAM J. COMPUT.
Vol. 6, No. 2, June 1977

FAST PATTERN MATCHING IN STRINGS*

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

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

- introduction
- brute force
- Knuth-Morris-Pratt
- Boyer-Moore
- Rabin-Karp
5.3 Substring Search

- introduction
- 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.

![Boyer-Moore algorithm diagram]

```
0   5       NEEDLE  ← pattern
5   5       NEEDLE
11  4
15  0
```

*return i = 15*
Boyer-Moore: mismatched character heuristic

Q. How much to skip?

Case 1. Mismatch character not in pattern.

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

Q. How much to skip?

Case 2a. Mismatch character in pattern.

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

| txt | . . . . . . E L E . . . . . |
| pat | N E E D L E |

aligned with rightmost E?

| txt | . . . . . . E L E . . . . . |
| pat | N E E D L E |

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).

```
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>c</th>
<th>N E E D L E</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>-1 -1 -1 -1 -1 -1</td>
</tr>
<tr>
<td>B</td>
<td>-1 -1 -1 -1 -1 -1</td>
</tr>
<tr>
<td>C</td>
<td>-1 -1 -1 -1 -1 -1</td>
</tr>
<tr>
<td>D</td>
<td>-1 -1 -1 -1 3 3 3</td>
</tr>
<tr>
<td>E</td>
<td>-1 -1 1 2 2 2 5</td>
</tr>
<tr>
<td>...</td>
<td>-1</td>
</tr>
<tr>
<td>L</td>
<td>-1 -1 -1 -1 -1 4 4 4</td>
</tr>
<tr>
<td>M</td>
<td>-1 -1 -1 -1 -1 -1 -1</td>
</tr>
<tr>
<td>N</td>
<td>-1 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>...</td>
<td>-1</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</td>
<td>0</td>
<td></td>
<td>A</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td>A</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td></td>
<td>A</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td></td>
<td>A</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td></td>
<td>A</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td></td>
<td>A</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

- Introduction
- Brute force
- Knuth-Morris-Pratt
- Boyer-Moore
- Rabin-Karp
5.3 SUBSTRING SEARCH

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

Michael Rabin, Turing Award '76
Dick Karp, Turing Award '85
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>( pat ).charAt(( i ))</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>( i )</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>--------------------------</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>2 6 5 3 5 % 997 = 613</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( txt ).charAt(( i ))</th>
<th>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>--------------------------</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
</tr>
<tr>
<td>3 1 4 1 5 % 997 = 508</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 4 1 5 % 997 = 201</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 4 1 5 % 997 = 715</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 1 5 % 997 = 971</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 5 2 6 % 997 = 442</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5 9 2 6 5 % 997 = 929</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6 return ( i = 6 )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>--------------------------</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
</tr>
<tr>
<td>2 6 5 3 5 % 997 = 613</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Efficiently computing the hash function

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

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

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

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

---

```
// Compute hash for M-digit key
private long hash(String key, int M) {
    long h = 0;
    for (int j = 0; j < M; j++)
        h = (R * h + key.charAt(j)) % Q;
    return h;
}
```
Efficiently computing the hash function

**Challenge.** How to efficiently compute \( x_{i+1} \) given that we know \( x_i \).

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

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

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

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

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

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

```plaintext

<table>
<thead>
<tr>
<th>i</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>current value</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>5</td>
<td>9</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>new value</td>
<td></td>
<td>4</td>
<td>1</td>
<td>5</td>
<td>9</td>
<td>2</td>
<td>6</td>
</tr>
</tbody>
</table>

current value: current value
subtract leading digit: 4 - 4 = 0
multiply by radix: 0 * 1 = 0
add new trailing digit: 0 + 6 = 6
new value: 1 5 9 2 6
```
Rabin-Karp substring search example

<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td></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>
</tbody>
</table>

0 | 3 | % 997 = 3
1 | 3 | 1 | % 997 = (3*10 + 1) % 997 = 31
2 | 3 | 1 | 4 | % 997 = (31*10 + 4) % 997 = 314
3 | 3 | 1 | 4 | 1 | % 997 = (314*10 + 1) % 997 = 150
4 | 3 | 1 | 4 | 1 | 5 | % 997 = (150*10 + 5) % 997 = 508
5 | 1 | 4 | 1 | 5 | 9 | % 997 = ((508 + 3*(997 - 30))*10 + 9) % 997 = 201
6 | 4 | 1 | 5 | 9 | 2 | % 997 = ((201 + 1*(997 - 30))*10 + 2) % 997 = 715
7 | 1 | 5 | 9 | 2 | 6 | % 997 = (((715 + 4*(997 - 30))*10 + 6) % 997 = 971
8 | 5 | 9 | 2 | 6 | 5 | % 997 = (((971 + 1*(997 - 30))*10 + 5) % 997 = 442
9 | 9 | 2 | 6 | 5 | 3 | % 997 = (((442 + 5*(997 - 30))*10 + 3) % 997 = 929
10 | return i-M+1 = 6 | 2 | 6 | 5 | 3 | 5 | % 997 = ((929 + 9*(997 - 30))*10 + 5) % 997 = 613

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 */ }
}

precompute R^{M-1} (mod Q)
a large prime (but avoid overflow)
Monte Carlo version. Return match if hash match.

```java
public int search(String txt)
{
    int N = txt.length();
    int txtHash = hash(txt, M);
    if (patHash == txtHash) return 0;
    for (int i = M; i < N; i++)
    {
        txtHash = (txtHash + Q - 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 $MN^2$), then the probability of a false collision is about $1/N$.

**Practice.** Choose $Q$ to be a large prime (but not so large to cause overflow). Under reasonable assumptions, probability of a collision is about $1/Q$.

**Monte Carlo version.**
- Always runs in linear time.
- Extremely likely to return correct answer (but not always!).

**Las Vegas version.**
- Always returns correct answer.
- Extremely likely to run in linear time (but worst case is $MN$).
Rabin-Karp fingerprint search

Advantages.

- Extends to 2d patterns.
- Extends to finding multiple patterns.

Disadvantages.

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

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

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

<table>
<thead>
<tr>
<th>algorithm</th>
<th>version</th>
<th>operation count guarantee</th>
<th>operation count typical</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>1.1 $N$</td>
<td>yes</td>
<td>yes</td>
<td>1</td>
</tr>
<tr>
<td>Knuth-Morris-Pratt</td>
<td>full DFA (Algorithm 5.6)</td>
<td>2 $N$</td>
<td>1.1 $N$</td>
<td>no</td>
<td>yes</td>
<td>MR</td>
</tr>
<tr>
<td></td>
<td>mismatch transitions only</td>
<td>3 $N$</td>
<td>1.1 $N$</td>
<td>no</td>
<td>yes</td>
<td>M</td>
</tr>
<tr>
<td>Boyer-Moore</td>
<td>full algorithm</td>
<td>3 $N$</td>
<td>$N / M$</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>$N / M$</td>
<td>yes</td>
<td>yes</td>
<td>R</td>
</tr>
<tr>
<td>Rabin-Karp†</td>
<td>Monte Carlo (Algorithm 5.8)</td>
<td>7 $N$</td>
<td>7 $N$</td>
<td>no</td>
<td>yes †</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Las Vegas</td>
<td>7 $N †$</td>
<td>7 $N$</td>
<td>yes</td>
<td>yes</td>
<td>1</td>
</tr>
</tbody>
</table>

† probabilistic guarantee, with uniform hash function
5.3 Substring Search

- introduction
- brute force
- Knuth-Morris-Pratt
- Boyer-Moore
- Rabin-Karp
5.3 Substring Search

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