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 \( N \gg M \)

\[
\begin{array}{c}
\text{pattern} \rightarrow N \ E \ E \ D \ L \ E \\
\text{text} \rightarrow I \ N \ A \ H \ A \ Y \ S \ T \ A \ C \ K \ N \ E \ E \ D \ L \ E \ I \ N \ A \\
\text{match}
\end{array}
\]

Substring search applications

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

Typically \( N \gg M \)

\[
\begin{array}{c}
\text{pattern} \rightarrow N \ E \ E \ D \ L \ E \\
\text{text} \rightarrow I \ N \ A \ H \ A \ Y \ S \ T \ A \ C \ K \ N \ E \ E \ D \ L \ E \ I \ N \ A \\
\text{match}
\end{array}
\]
Substring search applications

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

- Typically \( N \gg M \)

**Pattern** \( \text{NEEDLE} \)

**Text** \( \text{INAHAYSTACKNEEDLEINA} \)

**Match**

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

![Computer forensics image](http://citp.princeton.edu/memory)

**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.

![SpamAssassin image](http://www.spamassassin.org)

**Screen scraping.** Extract relevant data from web page.

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

```html
<br /&gt;
	<td class="yfnc_tablehead1" width="48%">

Last Trade:
	</td>
	<td class="yfnc_tabledata1">

452.92
	</td>
	</tr>
	<td class="yfnc_tablehead1" width="48%">

Trade Time:
	</td>
	<td class="yfnc_tabledata1">

15:17
	</td>
	</tr>
```

![Screen scraping example](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**

### 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>C</td>
<td>D</td>
<td>A</td>
<td>B</td>
<td>R</td>
<td>A</td>
<td>C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>2</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>1</td>
<td>0</td>
<td>1</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</td>
<td>1</td>
<td>3</td>
<td>A</td>
<td>B</td>
<td>R</td>
<td></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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>5</td>
<td>A</td>
<td>B</td>
<td>R</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>5</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</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>

Return i when j is M

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

Worst case. $\sim MN$ char compares.

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).

<table>
<thead>
<tr>
<th>i</th>
<th>j</th>
<th>i+j</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</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>
</tbody>
</table>

Backup

In many applications, we want to avoid backup in text stream.
- Treat input as stream of data.
- Abstract model: standard input.

Brute-force algorithm needs backup for every mismatch.

Approach 1. Maintain buffer of last M characters.

Approach 2. Stay tuned.

Algorithmic challenges in substring search

Brute-force is not always good enough.

Theoretical challenge. Linear-time guarantee. $\rightarrow$ fundamental algorithmic problem

Practical challenge. Avoid backup in text stream. $\leftarrow$ often no room or time to save text

public static int search(String pat, String txt)
{
    int i, 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;
}
5.3 Substring Search

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 BAAAAAB.
- Don’t need to back up text pointer!

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

Deterministic finite state automaton (DFA)

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

DFA simulation demo
DFA simulation demo

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

<table>
<thead>
<tr>
<th>dfa[i][j]</th>
<th>B</th>
<th>C</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>0</td>
<td>6</td>
</tr>
</tbody>
</table>

Knuth-Morris-Pratt substring search: Java implementation

Key differences from brute-force implementation.
- Need to precompute \( dfa[] \) from pattern.
- Text pointer 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]

Interpretation of Knuth-Morris-Pratt DFA

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

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

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;
}
```
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

How to build DFA from pattern?

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

Match transition. If in state j and next char c == 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
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'
- Simulate BABA; take transition 'B'

<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>$\text{pat.charAt(j)}$</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>C</td>
</tr>
<tr>
<td>$\text{dfa[][]}$</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>B</td>
<td>C</td>
<td></td>
</tr>
</tbody>
</table>

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

- From state $X$, take transition 'A'
- From state $X$, take transition 'B'

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

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

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

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

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

Knuth-Morris-Pratt: brief history

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 \( dfa[][] \) in time and space proportional to \( R M \).

Larger alphabets. Improved version of KMP constructs \( nfa[][] \) in time and space proportional to \( M \).
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.

![Diagram](image1)

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

before
```
txt  . . . .  T L E . . . .
pat  NEEDLE
```

after
```
txt  . . . .  T L E . . . .
pat  NEEDLE
```

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

**Case 2a.** Mismatch character in pattern.

before
```
txt  . . . .  N L E . . . .
pat  NEEDLE
```

after
```
txt  . . . .  N L E . . . .
pat  NEEDLE
```

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

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

before
```
txt  . . . .  E L E . . . .
pat  NEEDLE
```

aligned with rightmost 'E'?
```
txt  . . . .  E L E . . . .
pat  NEEDLE
```

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).

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

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

Boyer-Moore: Java implementation

```java
public int search(String txt) {
    int N = txt.length();
    int M = pat.length();
    int skip;
    for (int i = 0; i <= N-M; i += skip) {
        skip = 0;
        for (int j = M-1; j >= 0; j--)
            if (pat.charAt(j) != txt.charAt(i+j))
                skip = Math.max(1, j - right[txt.charAt(i+j)]);
        break;
    }
    if (skip == 0) return i;  // match
    return N;
}
```

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).

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

compute skip value

in case other term is nonpositive

Boyer-Moore skip table computation

Property. Substring search with the Boyer-Moore mismatched character heuristic takes about ~N/M characters compares to search for a pattern of length M in a text of length N. \(\text{sublinear!}\)

Worst-case. Can be as bad as ~MN.

Boyer-Moore variant. Can improve worst case to ~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**

#### 5.3.1 Substring Search

**Rabin-Karp fingerprint search**

**Basic idea = modular hashing.**

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

**Efficiently computing the hash function**

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

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

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

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

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

**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!

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

(can precompute \( R^{M-1} \))
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>0</td>
<td>3</td>
<td>3</td>
<td>% 997 = 3</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</td>
<td>3</td>
<td>1</td>
<td>% 997 = (3*10 + 1) % 997 = 31</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</td>
<td>3</td>
<td>1</td>
<td>4</td>
<td>% 997 = (31*10 + 4) % 997 = 314</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</td>
<td>3</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>% 997 = (314*10 + 1) % 997 = 150</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</td>
<td>3</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>5</td>
<td>% 997 = (150*10 + 5) % 997 = 508</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>5</td>
<td>9</td>
<td>% 997 = ((508 + 3*(997 - 30))*10 + 9) % 997 = 201</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>1</td>
<td>5</td>
<td>9</td>
<td>2</td>
<td>% 997 = ((201 + 1*(997 - 30))*10 + 2) % 997 = 715</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>5</td>
<td>9</td>
<td>2</td>
<td>6</td>
<td>5</td>
<td>% 997 = ((715 + 4*(997 - 30))*10 + 6) % 997 = 971</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>5</td>
<td>9</td>
<td>2</td>
<td>6</td>
<td>5</td>
<td>3</td>
<td>% 997 = ((442 + 5*(997 - 30))*10 + 3) % 997 = 929</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>2</td>
<td>6</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>% 997 = ((929 + 9*(997 - 30))*10 + 5) % 997 = 613</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

return i-M+1 = 6

Rabin-Karp: Java implementation

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

    public RabinKarp(String pat) {
        M = pat.length();
        R = 256;
        Q = LongRandomPrime();
        RM = 1;
        for (int i = 1; i < M; i++)
            RM = (RM * R) % Q;
        patHash = hash(pat, M);
    }
    private long hash(String key, int M) {
        return key.substring(M-1).hashCode();
    }
    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++)
            if (patHash == txtHash) return i - M + 1;
        return N;
    }
}
```

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: Java implementation (continued)

**Monte Carlo version.** Return match if hash match.

```java
public int search(String txt)
{
    int N = txt.length();
    int txtHash = hash(txt, M);
    if (patHash == txtHash) return 0;
    for (int i = M; i < N; i++)
    {
        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 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</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>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</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