5.3 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 $N \gg M$

pattern $\rightarrow$ NEEDLE

text $\rightarrow$ INAHAYSTACK NEEDLE INA

match
Substring search applications

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

typically $N \gg M$

pattern $\rightarrow$ N E E D L E

text $\rightarrow$ I N A H A Y S T A C K $\underline{N E E D L E I N A}$

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

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

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

Typically $N \gg M$

**Identify patterns indicative of spam.**

- PROFITS
- LOSE WEIGHT
- 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:`.

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

- **txt**: A B A C A D A B R A C
- **pat**: R A

<table>
<thead>
<tr>
<th>entries in black</th>
<th>entries in red</th>
<th>entries in gray</th>
</tr>
</thead>
<tbody>
<tr>
<td>match the text</td>
<td>mismatches</td>
<td>for reference only</td>
</tr>
</tbody>
</table>

**Return i when j is M**
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>4</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>5</td>
</tr>
</tbody>
</table>

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>0</td>
<td>4</td>
<td>4</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>5</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>6</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>7</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>8</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>10</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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).

<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>

\[
\begin{array}{cccccccccccc}
7 & 3 & A & D & A & C & R \\
5 & 0 & A & D & A & C & R \\
\end{array}
\]

```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.

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. 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 many or all 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 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. 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 many or all 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 all good Democrats to come to the aid of their party.

attack at dawn
- brute force
- Knuth-Morris-Pratt
- Boyer-Moore
- Rabin-Karp
**Knuth-Morris-Pratt substring search**

**Intuition.** Suppose we are searching in text for pattern BAAAAAA.

- 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.

<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>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>1</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>4</td>
<td></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>

If in state $j$ reading char $c$:
- if $j$ is 6 halt and accept
- else move to state $\text{dfa}[c][j]$
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.

```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`
Knuth-Morris-Pratt DFA construction demo
How to build DFA from pattern?

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

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

0 1 2 3 4 5 6
How to build DFA from pattern?

**Match transition.** If in state \( j \) and next char \( c = \text{pat}.\text{charAt}(j) \), 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}.\text{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>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

Diagram: 

```
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 \( c \neq \text{pat.charAt}(j) \), 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'
\[ = \text{dfa}['B'][3] \]

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

![Diagram of DFA simulation of BABA](image)
Mismatch transition. If in state $j$ and next char $c \neq \text{pat}\cdot\text{charAt}(j)$, 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.** $\text{dfa}[\text{'A'}][5] = 1$; $\text{dfa}[\text{'B'}][5] = 4$; $X' = 0$

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>

**How to build DFA from pattern?**

```plaintext
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]}
```
Knuth-Morris-Pratt DFA construction (in linear time) demo
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 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 $M$.

---

**KMP NFA for ABABAC**
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.
Knuth-Morris-Pratt application

A string $s$ is a **cyclic rotation** of $t$ if $s$ and $t$ have the same length and $s$ is a suffix of $t$ followed by a prefix of $t$.

<table>
<thead>
<tr>
<th>yes</th>
<th>yes</th>
<th>no</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROTATEDSTRING</td>
<td>ABA</td>
<td>ABABABBABABABA</td>
</tr>
<tr>
<td>STRINGROTATED</td>
<td>BABBABBBABABA</td>
<td>ROTATEDSTRING</td>
</tr>
<tr>
<td>STR</td>
<td>GNI RTSDE TATOR</td>
<td></td>
</tr>
</tbody>
</table>

**Problem.** Given two strings $s$ and $t$, design a linear-time algorithm that determines if $s$ is a cyclic rotation of $t$.

**Solution.**

- Check that $s$ and $t$ are the same length.
- Search for $s$ in $t + t$ using KMP.

$$
\begin{array}{ccc}
 t + t & \rightarrow & \text{STRINGROTATEDSTRINGROTATED} \\
 s & \rightarrow & \text{ROTATEDSTRING} \\
\end{array}
$$
brute force
Knuth-Morris-Pratt
Boyer-Moore
Rabin-Karp

Robert Boyer  J. Strother Moore
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.

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

`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).

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).

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>

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

After:

<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>
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</th>
<th>E</th>
<th>E</th>
<th>D</th>
<th>L</th>
<th>E</th>
<th>right[c]</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>-1</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>
<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>
<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>
<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>
<td>5</td>
</tr>
<tr>
<td>...</td>
<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>4</td>
<td>4</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>
<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>
<td>0</td>
</tr>
<tr>
<td>...</td>
<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 $N$. sublinear!

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

Boyer-Moore variant. Can improve worst case to $\sim 3N$ by adding a KMP-like rule to guard against repetitive patterns.
 › 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>( i )</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{pat.charAt}(i) )</td>
<td>2</td>
<td>6</td>
<td>5</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>( \text{txt.charAt}(i) )</td>
<td>3</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>5</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>1</td>
<td>5</td>
<td>9</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>5</td>
<td>9</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>9</td>
<td>2</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>9</td>
<td>2</td>
<td>6</td>
<td>5</td>
<td>3</td>
</tr>
</tbody>
</table>

\[ \text{return } i = 6 \]

\[ 2 \ 6 \ 5 \ 3 \ 5 \ % \ 997 = 613 \]

match
Efficiently computing the hash function

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

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

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;
}
**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!

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

<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>text</td>
</tr>
</tbody>
</table>

 current value

 Subtract leading digit

 Multiply by radix

 Add new trailing digit

4 1 5 9 2  current value

- 4 0 0 0 0  subtract leading digit

* 1 0  multiply by radix

1 5 9 2 0  add new trailing digit

+ 6  new value

1 5 9 2 6  new value
Rabin-Karp substring search example

\[
\begin{array}{cccccccccccccccc}
  i & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\
  \hline
  0 & 3 & 1 & 4 & 1 & 5 & 9 & 2 & 6 & 5 & 3 & 5 & 8 & 9 & 7 & 9 & 3 \\
  1 & 3 & 1 & % 997 = (3*10 + 1) % 997 = 31 \\
  2 & 3 & 1 & 4 & % 997 = (31*10 + 4) % 997 = 314 \\
  3 & 3 & 1 & 4 & 1 & % 997 = (314*10 + 1) % 997 = 150 \\
  4 & 3 & 1 & 4 & 1 & 5 & % 997 = (150*10 + 5) % 997 = 508 \\
  5 & 1 & 4 & 1 & 5 & 9 & % 997 = ((508 + 3*(997 - 30))\times10 + 9) % 997 = 201 \\
  6 & 4 & 1 & 5 & 9 & 2 & % 997 = ((201 + 1*(997 - 30))\times10 + 2) % 997 = 715 \\
  7 & 1 & 5 & 9 & 2 & 6 & % 997 = ((715 + 4*(997 - 30))\times10 + 6) % 997 = 971 \\
  8 & 5 & 9 & 2 & 6 & 5 & % 997 = ((971 + 1*(997 - 30))\times10 + 5) % 997 = 442 \\
  9 & 9 & 2 & 6 & 5 & 3 & % 997 = ((442 + 5*(997 - 30))\times10 + 3) % 997 = 929 \\
  10 & \text{return } i-M+1 = 6 & 2 & 6 & 5 & 3 & 5 & % 997 = ((929 + 9*(997 - 30))\times10 + 5) % 997 = 613 \\
\end{array}
\]
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-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 */
    }
}
```

- A large prime (but avoid overflow)
- Precompute $R^{M-1} \pmod{Q}$
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 $MN^2$), then the probability of a false collision is about $1/N$.

**Practice.** Choose $Q$ to be a large prime (but not so large 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 $MN$).
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</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>typical</td>
<td></td>
<td></td>
</tr>
<tr>
<td>brute force</td>
<td>—</td>
<td>$MN$</td>
<td>1.1$N$</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Knuth-Morris-Pratt</td>
<td>full DFA (Algorithm 5.6)</td>
<td>$2N$</td>
<td>1.1$N$</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>mismatch transitions only</td>
<td>$3N$</td>
<td>1.1$N$</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Boyer-Moore</td>
<td>full algorithm</td>
<td>$3N$</td>
<td>$N/M$</td>
<td>yes</td>
<td>yes</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>
</tr>
<tr>
<td>Rabin-Karp†</td>
<td>Monte Carlo (Algorithm 5.8)</td>
<td>$7N$</td>
<td>7$N$</td>
<td>no</td>
<td>yes†</td>
</tr>
<tr>
<td></td>
<td>Las Vegas</td>
<td>$7N$†</td>
<td>7$N$</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

† probabilistic guarantee, with uniform hash function