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** $\rightarrow$ NEEDLE
- **text** $\rightarrow$ INAHAYSTACK NEEDLE INA

- **match**
Substring search applications

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

```
pattern → N E E D L E

text → I N A H A Y S T A C K N E E D L E I N A
```

typically $N >> M$

```
Find: search
Replace: 
```

Find & Replace
Simple  Advanced
Replace All  Replace  Replace & Find  Previous  Next
Substring search applications

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

```
    pattern → N E E D L E
    text   → I N A H A Y S T A C K N E E D L E I N A

match
```

typically $N \gg M$

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

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

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

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

...<tr>
  <td class="yfnc_tablehead1" width="48%">
  Last Trade:
  </td>
  <td class="yfnc_tabledata1">582.93</td>
  <td class="yfnc_tablehead1" width="48%">
  Trade Time:
  </td>
  <td class="yfnc_tabledata1">Nov 29</td>
...```
Screen scraping: Java implementation

**Java library.** The `indexOf()` method in Java's `String` data type 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);
    }
}
```

```sh
% java StockQuote goog
582.93
```

**Caveat.** Must update program if Yahoo format changes.
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>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>
<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>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>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; // pattern starts
    }
    return N; // not found
}
What is the worst-case running time of brute-force substring search as a function of the number of characters in the pattern $M$ and text $N$?

A. $M + N$

B. $M^2$

C. $MN$

D. $N^2$

E. I don't know.
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.
Brute-force substring search: alternate implementation

Same sequence of character compares as previous implementation.
- i points to end of sequence of already-matched characters in text.
- j stores # of already-matched characters (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>

A B A C A D A B R A C

```
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 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 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 Democrats 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 each 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 a lot of 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 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.

fundamental algorithmic problem

often no space (or time) to save text
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 BAAAAAAAAAAA.
- 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!

Assuming \{ A, B \} alphabet

**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 state transition for each char in alphabet.
- Accept if sequence of state 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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>6</td>
</tr>
</tbody>
</table>

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

**graphical representation**

- The automaton transitions are represented with arrows.
- States are represented with circles.
- Edges indicate the transition based on the input character.
- Distinguish start state (0) and accept states (6).
Knuth–Morris–Pratt demo: DFA simulation

A A B A C A A B A B A C A 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>
<tr>
<td>B</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
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>i</th>
<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></td>
<td>0</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>
<td></td>
</tr>
</tbody>
</table>

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

<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>A</td>
<td>B</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>C</td>
</tr>
</tbody>
</table>

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

length of longest prefix of \( \text{pat}[] \) that is a suffix of \( \text{txt}[0..i] \)
Which state is the DFA in after processing the following input?


A. 0

B. 1

C. 3

D. 4

E. I don't know.
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]
Key differences from brute-force implementation.

- Need to precompute $\text{dfa}[][]$ 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 demo: DFA construction

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

```
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 \( c \neq \text{pat}.\text{charAt}(j) \), 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. Seems to require \( j \) steps.

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

simulate BABAA

simulate BABAB

<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}.\text{charAt}(j) )</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>C</td>
</tr>
</tbody>
</table>

```plaintext
 simulation of BABA
```

```plaintext
A
```

```plaintext
B
```

```plaintext
C
```

```plaintext
B, C
```
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 \( \text{pat}[1.\ldots j-1] \), followed by \( c \).

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

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

Ex. \( \text{dfa}['A'][5] = 1 \) from state \( x \), take transition 'A' = \( \text{dfa}['A'][x] \)  
\( \text{dfa}['B'][5] = 4 \) from state \( x \), take transition 'B' = \( \text{dfa}['B'][x] \)  
update \( x = 0 \) from state \( x \), take transition 'C' = \( \text{dfa}['C'][x] \)
Knuth–Morris–Pratt demo: DFA construction 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 dfa[][x] to dfa[][j] for mismatch case.
- Set dfa[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;
    int x = 0;
    for (int 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 $R M$).
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 character accessed once when constructing the DFA; each text character 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 algorithm
  - Pratt: made running time independent of alphabet size
  - Morris: built a text editor for the CDC 6400 computer
- Theory meets practice.

---

**FAST PATTERN MATCHING IN STRINGS**

DONALD E. KNUTH*, JAMES H. MORRIS, JR., AND VAUGHAN R. PRATT

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

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 $t$.

**Problem.** Given two binary strings $s$ and $t$, design a linear-time algorithm to determine if $s$ is a cyclic rotation of $t$. 
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>ROTATED STRING</td>
<td>A B A B B A B B A B A</td>
<td>ROTATED STRING</td>
</tr>
<tr>
<td>STRING ROTATED</td>
<td>B A B B A B A B A A B A</td>
<td>GNIRTSDETATOR</td>
</tr>
</tbody>
</table>

**Problem.** Given two binary strings $s$ and $t$, design a linear-time algorithm to determine 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 Knuth–Morris–Pratt.
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.

<table>
<thead>
<tr>
<th>$i$</th>
<th>$j$</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>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>18</th>
<th>19</th>
<th>20</th>
<th>21</th>
<th>22</th>
<th>23</th>
</tr>
</thead>
<tbody>
<tr>
<td>text</td>
<td>FIN</td>
<td>DIN</td>
<td>I</td>
<td>N</td>
<td>A</td>
<td>H</td>
<td>A</td>
<td>Y</td>
<td>S</td>
<td>T</td>
<td>A</td>
<td>C</td>
<td>K</td>
<td>NEED</td>
<td>L</td>
<td>E</td>
<td>I</td>
<td>N</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>5</td>
<td>NEED</td>
<td>L</td>
<td>E</td>
<td>pattern</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></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>NEED</td>
<td>L</td>
<td>E</td>
<td>no S in pattern</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></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>4</td>
<td>align N in text with</td>
<td>NEED</td>
<td>L</td>
<td>E</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></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>0</td>
<td>align N in text with</td>
<td>NEED</td>
<td>L</td>
<td>E</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></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

return $i = 15$

align N in text with
N in pattern
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 (why?) pattern 'N'
Boyer–Moore: mismatched character heuristic

Q. How much to skip?

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

before

\[
\begin{array}{c}
txt \quad \cdot \cdot \cdot \cdot \cdot \cdot \cdot \quad E \quad L \quad E \quad \cdot \cdot \cdot \cdot \cdot \cdot \cdot \\
pat \quad N \quad E \quad E \quad D \quad L \quad E
\end{array}
\]

aligned with rightmost E?

\[
\begin{array}{c}
txt \quad \cdot \cdot \cdot \cdot \cdot \cdot \cdot \quad E \quad L \quad E \quad \cdot \cdot \cdot \cdot \cdot \cdot \cdot \\
pat \quad N \quad E \quad E \quad D \quad L \quad E
\end{array}
\]

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)

```c
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>
</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>
</tr>
<tr>
<td>B</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>C</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>D</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>E</td>
<td>-1</td>
<td>-1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>L</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>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>
</tr>
<tr>
<td>N</td>
<td>-1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
...
public int search(String txt)
{
    int N = txt.length();
    int M = pat.length();
    int skip;
    for (int i = 0; i <= N-M; i += skip)
    {
        skip = 0;
        for (int j = M-1; j >= 0; j--)
        {
            if (pat.charAt(j) != txt.charAt(i+j))
            {
                skip = Math.max(1, j - right[txt.charAt(i+j)]);
                break;
            }
        }
        if (skip == 0) return i;
    }
    return N;
}
Boyer–Moore: analysis

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

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

<table>
<thead>
<tr>
<th>i skip</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>txt</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td>0 0</td>
<td>A</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>pat</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1</td>
<td>A</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 1</td>
<td>A</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 1</td>
<td>A</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 1</td>
<td>A</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5 1</td>
<td>A</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td></td>
<td></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
Rabin–Karp fingerprint search

Basic idea = modular hashing.

- Compute a hash of $\text{pat}[0..M)$.
- For each $i$, compute a hash of $\text{txt}[i..M+i)$.
- If pattern hash = text substring hash, check for a match.

```
pat.charAt(i)
i  0  1  2  3  4
  2  6  5  3  5 % 997 = 613
```
```
txt.charAt(i)
i  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
  3  1  4  1  5  9  2  6  5  3  5  8  9  7  9  3
% 997 = 508
  1  4  1  5  9  2  % 997 = 715
% 997 = 201
  2  4  1  5  9  2  % 997 = 971
% 997 = 442
  3  1  5  9  2  6  % 997 = 929
% 997 = 929
  4  5  9  2  6  % 997 = 613
% 997 = 442
  5  2  6  5  3  % 997 = 613
% 997 = 929
  6 ← return $i = 6$
  2  6  5  3  5 % 997 = 613

modular hashing with $R = 10$ and $\text{hash}(s) = s \pmod{997}$
Modular arithmetic

**Math trick.** To keep numbers small, take intermediate results modulo \( Q \).

**Ex.**

\[
(10000 + 535) \times 1000 \pmod{997} = (30 + 535) \times 3 \pmod{997} = 1695 \pmod{997} = 698 \pmod{997}
\]

10000 mod 997 = 30

1000 mod 997 = 3

\[
(a + b) \mod Q = \left( (a \mod Q) + (b \mod Q) \right) \mod Q
\]

\[
(a \times b) \mod Q = \left( (a \mod Q) \times (b \mod Q) \right) \mod Q
\]

two useful modular arithmetic identities
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.

```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 = (h * R + key.charAt(j)) % Q;
    return h;
}
```

\[
26535 = 2 \times 10000 + 6 \times 1000 + 5 \times 100 + 3 \times 10 + 5
= (((((2) \times 10 + 6) \times 10 + 5) \times 10 + 3) \times 10 + 5
\]
Efficiently computing the hash function

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

\[
\begin{align*}
  x_i &= t_i R^{M-1} + t_{i+1} R^{M-2} + \ldots + t_{i+M-1} R^0 \\
  x_{i+1} &= t_{i+1} R^{M-1} + t_{i+2} R^{M-2} + \ldots + t_{i+M} R^0
\end{align*}
\]

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

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

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

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

```
  current value
  4 1 5 9 2
- 4 0 0 0 0
  1 5 9 2
* 1 0
  1 5 9 2
+ 6
  1 5 9 2 6
  new value
text
```
Rabin–Karp substring search example

**First R entries:** Use Horner's rule.

**Remaining entries:** Use rolling hash (and % to avoid overflow).

<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>
<tr>
<td>0</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>
<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 = (314*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>% 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>
<td></td>
</tr>
<tr>
<td>8</td>
<td>5</td>
<td>9</td>
<td>2</td>
<td>6</td>
<td>5</td>
<td>% 997 = ((971 + 1*(997 - 30))*10 + 5) % 997 = 442</td>
<td></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>% 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>
<td></td>
</tr>
<tr>
<td>10</td>
<td>return ( i - M + 1 = 6 )</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>

- \( -30 \text{ (mod 997)} = 997 - 30 \)
- \( 10000 \text{ (mod 997)} = 30 \)
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 RM1; // R^(M-1) % Q

    public RabinKarp(String pat) {
        M = pat.length();
        R = 256;
        Q = longRandomPrime();

        RM1 = 1;
        for (int i = 1; i <= M-1; i++)
            RM1 = (R * RM1) % 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.

```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++) {
        int 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. Modify code to 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 two-dimensional 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>2(N)</td>
<td>1.1(N)</td>
<td>no</td>
<td>yes</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>
</tr>
<tr>
<td>Boyer-Moore</td>
<td>full algorithm</td>
<td>3(N)</td>
<td>(N/M)</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>mismatched char heuristic only</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>7(N)</td>
<td>7(N)</td>
<td>no</td>
<td>yes†</td>
</tr>
<tr>
<td></td>
<td>Las Vegas</td>
<td>7(N)†</td>
<td>7(N)</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

† probabilistic guarantee, with uniform hash function