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

String. Sequence of characters.

Important fundamental abstraction.
- Information processing.
- Genomic sequences.
- Communication systems (e.g., email).
- Programming systems (e.g., Java programs).
- ...

“The digital information that underlies biochemistry, cell biology, and development can be represented by a simple string of G's, A's, T's and C's. This string is the root data structure of an organism's biology.” — M. V. Olson
The char data type

C char data type. Typically an 8-bit integer.
- Supports 7-bit ASCII.
- Need more bits to represent certain characters.

<table>
<thead>
<tr>
<th></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>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NUL</td>
<td>SOH</td>
<td>STX</td>
<td>ETX</td>
<td>EOT</td>
<td>ENQ</td>
<td>ACK</td>
<td>BEL</td>
<td>BS</td>
<td>HT</td>
<td>LF</td>
<td>VT</td>
<td>FF</td>
<td>CR</td>
<td>SO</td>
<td>SI</td>
</tr>
<tr>
<td>1</td>
<td>DLE</td>
<td>DC1</td>
<td>DC2</td>
<td>D3C</td>
<td>DC4</td>
<td>NAK</td>
<td>SYN</td>
<td>ETB</td>
<td>CAN</td>
<td>EM</td>
<td>SUB</td>
<td>ESC</td>
<td>FS</td>
<td>GS</td>
<td>RS</td>
<td>US</td>
</tr>
<tr>
<td>2</td>
<td>SP</td>
<td>!</td>
<td>“</td>
<td>#</td>
<td>$</td>
<td>%</td>
<td>&amp;</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>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>:</td>
<td>;</td>
<td>&lt;</td>
<td>=</td>
<td>&gt;</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>@</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>N</td>
<td>O</td>
</tr>
<tr>
<td>5</td>
<td>P</td>
<td>Q</td>
<td>R</td>
<td>S</td>
<td>T</td>
<td>U</td>
<td>V</td>
<td>W</td>
<td>X</td>
<td>Y</td>
<td>Z</td>
<td>[</td>
<td>\</td>
<td>]</td>
<td>^</td>
<td>_</td>
</tr>
<tr>
<td>6</td>
<td>'</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>e</td>
<td>f</td>
<td>g</td>
<td>h</td>
<td>i</td>
<td>j</td>
<td>k</td>
<td>l</td>
<td>m</td>
<td>n</td>
<td>o</td>
</tr>
<tr>
<td>7</td>
<td>p</td>
<td>q</td>
<td>r</td>
<td>s</td>
<td>t</td>
<td>u</td>
<td>v</td>
<td>w</td>
<td>x</td>
<td>y</td>
<td>z</td>
<td>{</td>
<td></td>
<td></td>
<td>~</td>
<td>DEL</td>
</tr>
</tbody>
</table>

Hexadecimal to ASCII conversion table

Java char data type. A 16-bit unsigned integer.
- Supports original 16-bit Unicode.
- Supports 21-bit Unicode 3.0 (awkwardly).

Unicode characters
I (heart) Unicode
**The String data type**

**String data type.** Sequence of characters (immutable).

**Length.** Number of characters.

**Indexing.** Get the $i^{th}$ character.

**Substring extraction.** Get a contiguous sequence of characters.

**String concatenation.** Append one character to end of another string.

---

```plaintext
s.length()
s.charAt(3)
s.substring(7, 11)
```

---

$s \rightarrow$ ATTAACKATAWDNW

0 1 2 3 4 5 6 7 8 9 10 11 12
The String data type: Java implementation

```java
public final class String implements Comparable<String>
{
    private char[] val; // characters
    private int offset; // index of first char in array
    private int length; // length of string
    private int hash; // cache of hashCode()

    public int length()
    {  return length;  }

    public char charAt(int i)
    {  return value[i + offset];  }

    private String(int offset, int length, char[] val)
    {
        this.offset = offset;
        this.length = length;
        this.val    = val;
    }

    public String substring(int from, int to)
    {  return new String(offset + from, to - from, val);  }
    ...
```
The String data type: performance

String data type. Sequence of characters (immutable).

Design Choice. Immutable, cache or share the backing array

Underlying implementation. Immutable char[] array, offset, and length.

<table>
<thead>
<tr>
<th>operation</th>
<th>guarantee</th>
<th>extra space</th>
</tr>
</thead>
<tbody>
<tr>
<td>length()</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>charAt()</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>substring()</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>concat()</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>

Memory. 40 + 2N bytes for a virgin String of length N.

---

can use byte[] or char[] instead of String to save space
(but lose convenience of String data type)
The **StringBuilder data type**

**StringBuilder data type.** Sequence of characters (mutable).

**Design Choice.** Easier to update, can’t cache or share array.

**Underlying implementation.** Resizing `char[]` array and length.

<table>
<thead>
<tr>
<th>operation</th>
<th>String</th>
<th></th>
<th>StringBuilder</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>guarantee</td>
<td>extra space</td>
<td>guarantee</td>
<td>extra space</td>
</tr>
<tr>
<td>length()</td>
<td>I</td>
<td>I</td>
<td>I</td>
<td>I</td>
</tr>
<tr>
<td>charAt()</td>
<td>I</td>
<td>I</td>
<td>I</td>
<td>I</td>
</tr>
<tr>
<td>substring()</td>
<td>I</td>
<td>I</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>concat()</td>
<td>N</td>
<td>N</td>
<td>I*</td>
<td>I*</td>
</tr>
</tbody>
</table>

* amortized

**Remark.** **StringBuffer** data type is similar, but thread safe (and slower).
**String vs. StringBuilder**

**Q.** How to efficiently reverse a string?

**A.**

```java
public static String reverse(String s) {
    String rev = "";
    for (int i = s.length() - 1; i >= 0; i--)
        rev += s.charAt(i);
    return rev;
}
```

- Quadratic time
- String concatenation creates a new String and all chars in backing array are copied to new one.

**B.**

```java
public static String reverse(String s) {
    StringBuilder rev = new StringBuilder();
    for (int i = s.length() - 1; i >= 0; i--)
        rev.append(s.charAt(i));
    return rev.toString();
}
```

- Linear time
- The backing array is updated. Sometimes may need to expand the array but amortised cost is O(1)
String challenge: array of suffixes

Q. How to efficiently form array of suffixes?

<table>
<thead>
<tr>
<th>input string</th>
</tr>
</thead>
<tbody>
<tr>
<td>a a c a a g t t t t a c a a g c</td>
</tr>
<tr>
<td>0 1 2 3 4 5 6 7 8 9 10 11 12 13 14</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>suffixes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 a a c a a g t t t t a c a a g c</td>
</tr>
<tr>
<td>a c a a g t t t t a c a a g c</td>
</tr>
<tr>
<td>2 c a a g t t t t a c a a g c</td>
</tr>
<tr>
<td>3 a a g t t t t a c a a g c</td>
</tr>
<tr>
<td>4 a g t t t t a c a a g c</td>
</tr>
<tr>
<td>5 g t t t t a c a a g c</td>
</tr>
<tr>
<td>6 t t t a c a a g c</td>
</tr>
<tr>
<td>7 t t a c a a g c</td>
</tr>
<tr>
<td>8 t a c a a g c</td>
</tr>
<tr>
<td>9 a c a a g c</td>
</tr>
<tr>
<td>10 c a a g c</td>
</tr>
<tr>
<td>11 a a g c</td>
</tr>
<tr>
<td>12 a g c</td>
</tr>
<tr>
<td>13 g c</td>
</tr>
<tr>
<td>14 c</td>
</tr>
</tbody>
</table>
String vs. StringBuilder

Q. How to efficiently form array of suffixes?

A.

```java
public static String[] suffixes(String s)
{
    int N = s.length();
    String[] suffixes = new String[N];
    for (int i = 0; i < N; i++)
        suffixes[i] = s.substring(i, N);
    return suffixes;
}
```

B.

```java
public static String[] suffixes(String s)
{
    int N = s.length();
    StringBuilder sb = new StringBuilder(s);
    String[] suffixes = new String[N];
    for (int i = 0; i < N; i++)
        suffixes[i] = sb.substring(i, N);
    return suffixes;
}
```

Since Strings are immutable, the backing array of larger String can be shared with substring. In Java 1.7 they changed it, now cost is the same as below!

The array of StringBuilder can change, so can't share with substring.
Longest common prefix

Q. How long to compute length of longest common prefix?

![Example strings](image)

```java
public static int lcp(String s, String t) {
    int N = Math.min(s.length(), t.length());
    for (int i = 0; i < N; i++)
        if (s.charAt(i) != t.charAt(i))
            return i;
    return N;
}
```

Running time. Proportional to length $D$ of longest common prefix.

Remark. Also can compute `compareTo()` in sublinear time.
Digital key. Sequence of digits over fixed alphabet.

Radix. Number of digits $R$ in alphabet.

Complexity of some algorithms will depend on this

<table>
<thead>
<tr>
<th>name</th>
<th>$R()$</th>
<th>$\lg R()$</th>
<th>characters</th>
</tr>
</thead>
<tbody>
<tr>
<td>BINARY</td>
<td>2</td>
<td>1</td>
<td>01</td>
</tr>
<tr>
<td>OCTAL</td>
<td>8</td>
<td>3</td>
<td>01234567</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>10</td>
<td>4</td>
<td>0123456789</td>
</tr>
<tr>
<td>HEXADECIMAL</td>
<td>16</td>
<td>4</td>
<td>0123456789ABCDEF</td>
</tr>
<tr>
<td>DNA</td>
<td>4</td>
<td>2</td>
<td>ACTG</td>
</tr>
<tr>
<td>LOWERCASE</td>
<td>26</td>
<td>5</td>
<td>abcdefghijklmnopqrstuvwxyz</td>
</tr>
<tr>
<td>UPPERCASE</td>
<td>26</td>
<td>5</td>
<td>ABCDEFGHIJKLMNOPQRSTUVWXYZ</td>
</tr>
<tr>
<td>PROTEIN</td>
<td>20</td>
<td>5</td>
<td>ACDEFGHIJKLMNOPQRSTUVWXYZ</td>
</tr>
</tbody>
</table>
| BASE64         | 64    | 6         | ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789+/
| ASCII          | 128   | 7         | ASCII characters                       |
| EXTENDED_ASCII | 256   | 8         | extended ASCII characters              |
| UNICODE16      | 65536 | 16        | Unicode characters                     |
• 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 $\rightarrow$ INA HAY STACK NEEDLE INA

match
**Substring search applications**

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

Typically $N >> M$

**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 $N$.

- Find patterns indicative of spam.
  - PROFITS
  - LOSE WEIGHT
  - There is no catch.
  - This is a one-time mailing.
  - This message is sent in compliance with spam regulations.
Substring search applications

Electronic surveillance.

Need to monitor all internet traffic. (security)

No way! (privacy)

Well, we’re mainly interested in “ATTACK AT DAWN”

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
Substring Search

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

Check for pattern starting at each text position.

```
  i  j  i+j  0  1  2  3  4  5  6  7  8  9  10
  txt → A  B  A  C  A  D  A  B  R  A  C
0  2  2  A  B  R  A  pat
1  0  1  A  B  R  A
2  1  3  A  B  R  A
3  0  3  A  B  R  A
4  1  5  A  B  R  A
5  0  5  A  B  R  A
6  4  10 A  B  R  A
```

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>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>

| A | B | A | C | A | D | A | B | R | A | C |

4 3 7  A D A C R

5 0 5  A D A C R

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

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

<table>
<thead>
<tr>
<th>i</th>
<th>j</th>
<th>i+j</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
</tr>
</tbody>
</table>

```
txt  A A A A A A A A A B
pat  B
```

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>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
</tr>
</tbody>
</table>

```
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 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 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.
Substring Search

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

**Intuition.** Suppose we are searching in text for pattern `BAAAAAAAAA`.
- 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.

**internal representation**

<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><code>pat.charAt(j)</code></td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>C</td>
</tr>
<tr>
<td><code>dfa[i][j]</code></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 DFA is illustrated with states and transitions labeled with characters. States are represented by circles, and transitions are labeled with the input characters that cause a move to another state.
DFA simulation

A A B A C A A B A B A C A 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></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>6</td>
</tr>
</tbody>
</table>

![DFA Simulation Diagram]
DFA simulation

\[
\begin{array}{ccccccc}
\end{array}
\]

\[
\begin{array}{c|ccccccc}
& 0 & 1 & 2 & 3 & 4 & 5 \\
pat.charAt(j) & A & B & A & B & A & C \\
dfa[][j] & A & 1 & 3 & 1 & 5 & 1 \\
B & 0 & 2 & 0 & 4 & 0 & 4 \\
C & 0 & 0 & 0 & 0 & 0 & 6 \\
\end{array}
\]

\[
\begin{array}{ccccccc}
& 0 \\
B, C & A
\end{array}
\]

\[
\begin{array}{ccccccc}
0 & 1 & 2 & 3 & 4 & 5 \\
A & B & A & B & A & C \\
1 & 1 & 3 & 1 & 5 & 1 \\
0 & 2 & 0 & 4 & 0 & 4 \\
0 & 0 & 0 & 0 & 0 & 6 \\
\end{array}
\]

\[
\begin{array}{ccccccc}
& 0 \\
B, C & A
\end{array}
\]

\[
\begin{array}{ccccccc}
& 0 \\
B, C & A
\end{array}
\]
DFA simulation

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

\[
\begin{array}{c|cccccc}
\text{pat.charAt(j)} & 0 & 1 & 2 & 3 & 4 & 5 \\
A & B & A & B & A & C \\
A & 1 & 1 & 3 & 1 & 5 & 1 \\
B & 0 & 2 & 0 & 4 & 0 & 4 \\
C & 0 & 0 & 0 & 0 & 0 & 6 \\
\end{array}
\]
DFA simulation

```
A A B A C A A B A B A C A A
```

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

```
DFA simulation

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

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

dfa[][]

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

pat.charAt(j)

<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>
<tr>
<td>B</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
DFA simulation

A A B A C

A A B A B A C A 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>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>dfa[][j]</td>
<td>B</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>4</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>

A

B, C

B, C

B, C

B, C

B, C

B, C

B, C

B, C

B, C
DFA simulation
DFA simulation

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

\[
\begin{array}{c|cccccc}
\text{pat.charAt(j)} & 0 & 1 & 2 & 3 & 4 & 5 \\
\hline
A & B & A & B & A & C \\
A & 1 & 1 & 3 & 1 & 5 & 1 \\
B & 0 & 2 & 0 & 4 & 0 & 4 \\
C & 0 & 0 & 0 & 0 & 0 & 6 \\
\end{array}
\]
DFA simulation

\[
\begin{array}{cccccc}
A & A & B & A & C & A \\
A & B & A & B & A & C \\
\end{array}
\]

\[
\begin{array}{|c|c|c|c|c|c|}
\hline
\text{pat.charAt(j)} & 0 & 1 & 2 & 3 & 4 & 5 \\
\hline
A & B & A & B & A & C \\
A & 1 & 1 & 3 & 1 & 5 & 1 \\
B & 0 & 2 & 0 & 4 & 0 & 4 \\
C & 0 & 0 & 0 & 0 & 0 & 6 \\
\hline
\end{array}
\]
DFA simulation

A A B A C A A B A B A C A 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>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>1</td>
<td>3</td>
<td>1</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>dfa[][][j]</td>
<td>B</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>4</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>

0 1 2 3 4 5

DFA simulation

\[
\begin{array}{cccccc}
\text{pat.charAt(j)} & 0 & 1 & 2 & 3 & 4 & 5 \\
A & B & A & B & A & C \\
A & 1 & 1 & 3 & 1 & 5 & 1 \\
B & 0 & 2 & 0 & 4 & 0 & 4 \\
C & 0 & 0 & 0 & 0 & 0 & 6 \\
\end{array}
\]
DFA simulation

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[][]
DFA simulation

```
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 B 1 1 3 1 5 1
A C B 0 2 0 4 0 4
C 0 0 0 0 0 0 6
```

substring found

```
Q. What is interpretation of DFA state after reading in \( \text{txt}[i] \)?

A. State = number of characters in pattern that have been matched.

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

```
<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>txt</td>
<td>B</td>
<td>C</td>
<td>B</td>
<td>A</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>C</td>
<td>A</td>
</tr>
<tr>
<td>pat</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>C</td>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
</tbody>
</table>
```

Interpretation of Knuth-Morris-Pratt DFA

- The length of the longest prefix of \( \text{pat}[\cdot] \) that is a suffix of \( \text{txt}[0..i] \).

DFA States:
- State 0
  - Transitions: (A, B, C)
- State 1
  - Transitions: (A)
- State 2
  - Transitions: (A)
- State 3
  - Transitions: (B)
- State 4
  - Transitions: (A)
- State 5
  - Transitions: (C)
- State 6
  - Transitions: (B, C)
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 $\text{dfa}[] []$ from pattern.
- Text pointer $i$ never decrements.
- Could use input stream.

```
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

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>
</tbody>
</table>

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

0 1 2 3 4 5 6
**Match transition.** If in state \(j\) and next char \(c = \text{pat.charAt}(j)\), go to \(j+1\).

<table>
<thead>
<tr>
<th>(\text{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>B</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>C</td>
</tr>
<tr>
<td>A</td>
<td>1</td>
<td></td>
<td>3</td>
<td></td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td>2</td>
<td></td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6</td>
</tr>
</tbody>
</table>

Constructing the DFA for KMP substring search for \(A B A B A C\)
Mismatch transition: back up if $c \neq \text{pat.charAt}(j)$.

Constructing the DFA for KMP substring search for $\text{A B A B A C}$
Mismatch transition: back up if \( c \neq \text{pat.charAt}(j) \).

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

<table>
<thead>
<tr>
<th>( \text{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>1</td>
<td>3</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>6</td>
</tr>
</tbody>
</table>

0 \rightarrow 1 \rightarrow B \rightarrow 2 \rightarrow A \rightarrow 3 \rightarrow B \rightarrow 4 \rightarrow A \rightarrow 5 \rightarrow C \rightarrow 6
Mismatch transition: back up if \( c \neq \text{pat.charAt}(j) \).

### Knuth-Morris-Pratt construction

Constructing the DFA for KMP substring search for \( A B A B A C \)

<table>
<thead>
<tr>
<th>( \text{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>1</td>
<td>3</td>
<td>5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( \text{dfa}[][j] )</th>
<th>B</th>
<th>0</th>
<th>2</th>
<th>0</th>
<th>4</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td>6</td>
</tr>
</tbody>
</table>

```plaintext
0 -> B, C
1 -> A
2 -> B
3 -> A
4 -> B
5 -> A
6 -> C
```

```plaintext
B, C
```
Mismatch transition: back up if \( c \neq \text{pat.charAt}(j) \).

Constructing the DFA for KMP substring search for A B A B A C
Mismatch transition: back up if c != pat.charAt(j).

Knuth-Morris-Pratt construction

Constructing the DFA for KMP substring search for A B A B A C
**Knuth-Morris-Pratt construction**

**Mismatch transition:** back up if \( c \neq \text{pat.charAt}(j) \).
Knuth-Morris-Pratt construction

Constructing the DFA for KMP substring search for \textit{A B A B A C}

<table>
<thead>
<tr>
<th>\textbf{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>\textit{A}</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>C</td>
</tr>
<tr>
<td>\textit{B}</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>\textit{C}</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>\textit{B, C}</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>6</td>
</tr>
</tbody>
</table>
How to build DFA from pattern?

Include one state for each character in pattern (plus accept 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>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>$\text{dfa}[][j]$</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></td>
<td></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>
Mismatch transition. If in state \( j \) and next character \( c \neq \text{pat.charAt}(j) \), 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 \( \text{BABA} \);
- take transition 'A'
  = \( \text{dfa}['A'][3] \)

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

Ex. $\text{dfa[}'A'\text{'][5]} = 1$; from state $X$, take transition 'A'
    $\quad = \text{dfa[}'A'\text{'][X]}$
    $\quad = \text{dfa[}'A'\text{'][X]}$

$\text{dfa[}'B'\text{'][5]} = 4$; from state $X$, take transition 'B'
    $\quad = \text{dfa[}'B'\text{'][X]}$
    $\quad = \text{dfa[}'B'\text{'][X]}$

$X' = 0$ from state $X$, take transition 'C'
    $\quad = \text{dfa[}'C'\text{'][X]}$

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

How to build DFA from pattern?
Knuth-Morris-Pratt construction (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
**Knuth-Morris-Pratt construction (in linear time)**

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

First \( j \) characters of pattern have already been matched

Now first \( j+1 \) characters of pattern have been matched

<table>
<thead>
<tr>
<th>( \text{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>B</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>C</td>
</tr>
<tr>
<td>A</td>
<td>1</td>
<td></td>
<td>3</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td>2</td>
<td></td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6</td>
</tr>
</tbody>
</table>

Constructing the DFA for KMP substring search for A B A B A C
Mismatch transition. For state 0 and char \( c \neq \text{pat.charAt}(j) \), set \( \text{dfa}[c][0] = 0 \).

Knuth-Morris-Pratt construction (in linear time)

Constructing the DFA for KMP substring search for \( \text{A B A B A C} \)
Mismatch transition. For each state $j$ and char $c != \text{pat.charAt}(j)$, set $\text{dfa}[c][j] = \text{dfa}[c][X]$; then update $X = \text{dfa}[$pat.charAt$(j)][][X]$.

Knuth-Morris-Pratt construction (in linear time)

Constructing the DFA for KMP substring search for A B A B A C
Mismatch transition. For each state $j$ and char $c \neq \text{pat.charAt}(j)$, set $\text{dfa}[c][j] = \text{dfa}[c][X]$; then update $X = \text{dfa[pat.charAt}(j)][X]$.

Knuth-Morris-Pratt construction (in linear time)

Constructing the DFA for KMP substring search for A B A B A C
Mismatch transition. For each state $j$ and char $c \neq \text{pat.charAt}(j)$, set $\text{dfa}[c][j] = \text{dfa}[c][X]$; then update $X = \text{dfa[pat.charAt(j)]}[X]$.

**Knuth-Morris-Pratt construction (in linear time)**

Constructing the DFA for KMP substring search for A B A B A C
Mismatch transition. For each state $j$ and char $c \neq \text{pat.charAt}(j)$, set $\text{dfa}[c][j] = \text{dfa}[c][X]$; then update $X = \text{dfa[pat.charAt(j)]}[X]$.

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

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

$X = \text{simulation of B A B}$
Mismatch transition. For each state \( j \) and char \( c \neq \text{pat.charAt}(j) \), set\[ \text{dfa}[c][j] = \text{dfa}[c][\text{X}] \]; then update \( \text{X} = \text{dfa}[	ext{pat.charAt}(j)][\text{X}] \).
Mismatch transition. For each state $j$ and char $c \neq \text{pat.charAt}(j)$, set $\text{dfa}[c][j] = \text{dfa}[c][X]$; then update $X = \text{dfa}[\text{pat.charAt}(j)][X]$.

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

<table>
<thead>
<tr>
<th>$\text{pat.charAt}(j)$</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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<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>$A$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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>

$X = \text{simulation of B A B A C}$
Knuth-Morris-Pratt 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 $\text{dfa}[][X]$ to $\text{dfa}[][j]$ for mismatch case.
- Set $\text{dfa}[$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;
    for (int X = 0, j = 1; j < M; j++) {
        for (int c = 0; c < R; c++)
            dfa[c][j] = dfa[c][X];
        dfa[pat.charAt(j)][j] = j+1;
        X = dfa[pat.charAt(j)][X];
    }
}
```

Running time. $M$ character accesses (but space proportional to $RM$).
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 $\text{dfa}[\cdot][\cdot]$ in time and space proportional to $RM$.

**Larger alphabets.** Improved version of KMP constructs $nfa[\cdot]$ 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-time 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 \(\{a^n b^n\}^*\), can be recognized in linear time. Other algorithms which run even faster on the average are also considered.
Substring Search

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

- Scan the text with a window of $M$ chars (length of pattern)

  Pattern in Text (M)

  Scan Window (M)

- Case 1: Scan Window is exactly on top of the searched pattern

  - Starting from one end check if all characters are equal. (We must check!)

- Case 2: Scan Window starts after the pattern starts.
Boyer Moore Intuition (2)

- Case 3: Scan Window starts before the pattern starts

- Case 4: Independent

- In case 4, simply shift window M characters
- Avoid Case 2
- Convert Case 3 to Case 1, by shifting appropriately
Intuition.

- Scan characters in pattern from right to left.
- Can skip as many as $M$ text chars when finding one not in the pattern.
  - First we check the character in index pattern.length()-1
  - It is N which is not E, so we know that first 5 characters is not a match. Shift text 5 characters
  - S != E so shift 5, E == E so we can check for the pattern.length()-2, L! = N, skip 4.

| i  | j  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 |
|----|----|----|----|----|----|----|----|----|----|----|----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| text | FIND I N A H A Y S T A C K NEED LE I N A |
| 0 5 | NEED LE | \(\text{pattern}\) |
| 5 5 | NEED LE |
| 11 4 | NEED LE |
| 15 0 | NEED LE |
| return i = 15 |
**Boyer-Moore: mismatched character heuristic**

**Q.** How much to skip?

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

Before

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

<table>
<thead>
<tr>
<th>i</th>
</tr>
</thead>
</table>

After

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

mismatch character 'T' not in pattern: increment i one character beyond 'T'
Q. How much to skip?

Case 2a. Mismatch character in pattern.

mismatch character 'N' in pattern: align text 'N' with rightmost pattern 'N'
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'?
**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).

```java
right = new int[R];
for (int c = 0; c < R; c++)
    right[c] = -1;
for (int j = 0; j < M; j++)
    right[pattern.charAt(j)] = j;
```

<table>
<thead>
<tr>
<th>$c$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</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>-1</td>
</tr>
<tr>
<td>L</td>
<td>-1</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>
<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>-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;
}
SEARCH FOR: XXXX

A X A X A X A X X X A X A X X X X X A A A

If the window scan points to an unrecognised character, we can skip past that character. For this example, for the initial step we first match X at the end, when check for previous character (A) which is not in the string we skip 3 steps. The X at the end, we matched can still be the first character of the pattern, so we do not skip that.
**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</th>
<th>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></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></td>
</tr>
<tr>
<td>0</td>
<td>0</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>1</td>
<td>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</td>
<td>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</td>
<td>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</td>
<td>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</td>
<td>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 \( \sim 3N \) by adding a KMP-like rule to guard against repetitive patterns.
Substring Search

- Brute force
- Knuth-Morris-Pratt
- Boyer-Moore
- Rabin-Karp
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>i</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>i</td>
<td>0</td>
<td>2</td>
<td>6</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>% 997 = 613</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>txt.charAt(i)</th>
<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>i</td>
<td>0</td>
<td>3</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>5</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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>% 997 = 508</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</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>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>% 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>
</tr>
<tr>
<td></td>
<td>2</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>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>% 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>
</tr>
<tr>
<td></td>
<td>3</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>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>% 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>
</tr>
<tr>
<td></td>
<td>4</td>
<td>5</td>
<td>9</td>
<td>2</td>
<td>6</td>
<td>5</td>
<td>% 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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>% 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>
</tr>
<tr>
<td></td>
<td>5</td>
<td>9</td>
<td>2</td>
<td>6</td>
<td>5</td>
<td>3</td>
<td>% 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></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></td>
<td>6</td>
<td>return i = 6</td>
<td>2</td>
<td>6</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>% 997 = 613</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.

```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!

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

(current value subtract leading digit multiply by radix add new trailing digit (can precompute $R^{M-2}$))

- **Table:**
  
  | $i$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | ...
  |-----|---|---|---|---|---|---|---|---
  | current value | 1 | 4 | 1 | 5 | 9 | 2 | 6 | 5 |...
  | new value | 4 | 1 | 5 | 9 | 2 | 6 | 5 |

  - **Current value:**
    - Subtract leading digit:
      - $4 - 0 = 4$
    - Multiply by radix:
      - $4 \times 1 = 4$
    - Add new trailing digit:
      - $4 + 6 = 10$
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>
<tr>
<td>0</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>% 997 = 3</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>% 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>
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 */
    }
}
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 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.
## 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>$1.1N$</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>$1.1N$</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>mismatch transitions only</td>
<td>3$N$</td>
<td>$1.1N$</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 (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>7$N$</td>
<td>7$N$</td>
<td>no</td>
<td>yes†</td>
</tr>
<tr>
<td></td>
<td>Las Vegas</td>
<td>7$N^\dagger$</td>
<td>7$N$</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

† probabilisitic guarantee, with uniform hash function