**Substring Search**

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

- Typically \( N \gg M \)

**Text.** INAHAYSTACK

**Pattern.** NEEDLE

**Substring search applications**

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

- Typically \( N \gg M \)

**Text.** INAHAYSTACK

**Pattern.** NEEDLE

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

[Image of a computer with a memory module]

[URL: 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**
- There is no catch.
- This is a one-time mailing.
- This message is sent in compliance with spam regulations.

Identify patterns indicative of spam.

**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.

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

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

```
<tr>
  <td class= "yfnc_tabledata1">
    452.92
  </td>
</tr>
```

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

```
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: Java implementation

Check for pattern starting at each text position.

```
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.

Worst case. \( \sim MN \) char compares.
Backup

In many applications, we want to avoid backup in text stream.

• Treat input as stream of data.
• Abstract model: standard input.

Brute-force algorithm needs backup for every mismatch.

Approach 1. Maintain buffer of last $M$ characters.
Approach 2. Stay tuned.

Algorithmic challenges in substring search

Brute-force is not always good enough.

Theoretical challenge. Linear-time guarantee. 

Practical challenge. Avoid backup in text stream.

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).

```
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;
}
```
Knuth-Morris-Pratt substring search

Intuition. Suppose we are searching in text for pattern $\text{BAAAAAAAAA}$. Suppose we match 5 chars in pattern, with mismatch on 6th char. We know previous 6 chars in text are $\text{BAAAAB}$. Don’t need to back up text pointer!

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

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.

Deterministic finite state automaton (DFA)

DFA simulation

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

\[
\begin{array}{c|ccccc}
\text{dfa}[][] & 0 & 1 & 2 & 3 & 4 & 5 \\
\hline
B & 0 & 2 & 0 & 4 & 0 & 4 \\
C & 0 & 0 & 0 & 0 & 6 \\
\end{array}
\]
DFA simulation

\[ \text{pat.charAt}(j) \]
\[ \text{dfa}[][] \]

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

pat.charAt(j)
A B A B A C
A 1 1 3 1 5 1
dfa[][]
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

pat.charAt(j)
A B A B A C
A 1 1 3 1 5 1
dfa[][]
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

pat.charAt(j)
A B A B A C
A 1 1 3 1 5 1
dfa[][]
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

pat.charAt(j)
A B A B A C
A 1 1 3 1 5 1
dfa[][]
B 0 2 0 4 0 4
C 0 0 0 0 0 6
**Interpretation of Knuth-Morris-Pratt DFA**

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

Ex. DFA is in state 3 after reading in `txt[0..6]`.

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

**Knuth-Morris-Pratt substring search: Java implementation**

Key differences from brute-force implementation.
- Need to precompute `dfa[]` from pattern.
- Text pointer `i` never decrements.

```
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 NOT_FOUND;
}
```

Running time.
- Simulate DFA on text: at most `N` character accesses.
- Build DFA: how to do efficiently? [warning: tricky algorithm ahead]

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

```
**Knuth-Morris-Pratt construction**

**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

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

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

**Mismatch transition:** back up if $c \neq \text{pat.charAt}(j)$.

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

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

Mismatch transition: back up if $c \neq \text{pat.charAt}(j)$.

### 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>1</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>$\text{dfa}[][]$</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>

Knuth-Morris-Pratt construction

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>1</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>$\text{dfa}[][]$</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>
How to build DFA from pattern?

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

Match transition. If in state \(j\) and next char \(c = \text{pat.charAt}(j)\), go to \(j+1\).

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 \(c\).

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

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

From state \(X\), take transition ‘A’ = \(\text{dfa}['A'][X]\); take transition ‘B’ = \(\text{dfa}['B'][X]\); take transition ‘C’ = \(\text{dfa}['C'][X]\).
Knuth-Morris-Pratt construction (in linear time)

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

```
<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>
<tr>
<td>dfa[][]</td>
<td>A</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

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

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

Knuth-Morris-Pratt construction (in linear time)

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

```
<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>
<tr>
<td>dfa[][]</td>
<td>A</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

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

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

Knuth-Morris-Pratt construction (in linear time)

Mismatch transition. For state 0 and char \( c \) != \( \text{pat.charAt}(j) \), set \( \text{dfa}[c][0] = 0 \).

```
<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>
<tr>
<td>dfa[][]</td>
<td>A</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

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

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

Knuth-Morris-Pratt construction (in linear time)

Mismatch transition. For each state \( j \) and char \( c != \text{pat.charAt}(j) \), set \( \text{dfa}[c][] = \text{dfa}[c][\text{X}] \); then update \( \text{X} = \text{dfa}[\text{pat.charAt}(j)][\text{X}] \).

```
<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>B</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>dfa[][]</td>
<td>A</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

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

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

X = simulation of empty string
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 $X = \text{dfa}[	ext{pat.charAt}(j)][X]$.

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

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

**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 $X = \text{dfa[pat.charAt(j)]}[X]$.

**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 $M$.
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.

Committee:

Friday, 2 June 1977

FAST PATTERN MATCHING IN STRINGS

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

Abstract: An algorithm is presented which finds all occurrences of one given string within
another, in running time proportional to the sum of the lengths of the strings. The constant of
proportionality is low enough to make the algorithm of practical use, and the procedure can also
be used to find all occurrences of any set of strings. The algorithm is based on the use of
substrings of the pattern string to determine in advance positions in the text string where a
match must begin. Several variations of the algorithm are described, and the one used is
recognized to be the most efficient. Other algorithms which can even be run on the average are
also considered.

Don Knuth  Jim Morris  Vaughan Pratt

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 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

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

- If we can recognise the character in the scan window end-point, we can find how many characters to shift.

![Diagram showing how to shift window to overlap]

- So, for example D is the 4th character, we must shift window 4 characters so that they overlap.

Boyer Moore Intuition (4)

- A potential problem, the character in the text can repeat.

- For example, pattern = XXAXX and the text is A X A X A X X A X A X X A X A X

- Solution: be conservative, choose the instance with the least Shift (so we cannot miss the others).

- So, for the example above when end-point is X, we should shift for 1 characters, when it is A shift for 2 characters.

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.
  - 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
  - \( S \neq E \) so shift 5, \( E = E \) so we can check for the pattern.length()-2, LInN, skip 4.

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

- Before
  - Text: T L E
  - Pattern: NEEDLE

- After
  - Text: T L E
  - Pattern: NEEDLE

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.

before

<table>
<thead>
<tr>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>N L E</td>
</tr>
<tr>
<td>N E E D L E</td>
</tr>
</tbody>
</table>

after

<table>
<thead>
<tr>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>N L E</td>
</tr>
<tr>
<td>N E E D L E</td>
</tr>
</tbody>
</table>

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

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

before

<table>
<thead>
<tr>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>E L E</td>
</tr>
<tr>
<td>N E E D L E</td>
</tr>
</tbody>
</table>

after

<table>
<thead>
<tr>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>E L E</td>
</tr>
<tr>
<td>N E E D L E</td>
</tr>
</tbody>
</table>

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

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[pat.charAt(j)] = j;
```

Boyer-Moore skip table computation
Boyer-Moore: Java implementation

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

Another Example

SEARCH FOR: XXXX

```
A X A X A X X A X X A X X X A A A
```

If the window scan points to an unrecognised character, we can skip M steps (gray scan). For this example, when we see an A skip 4. When we see an X skip 1 step.

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 \( \sim MN \).

<table>
<thead>
<tr>
<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>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>pat</td>
<td>A</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</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.
Rabin-Karp fingerprint search

Basic idea = modular hashing.

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

Efficiently computing the hash function

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

$$x_i = t_i R^M + t_{i+1} R^{M-1} + ... + t_{i+M-1} R^0 \pmod{Q}$$

**Horner’s method.** Linear-time method to evaluate degree-$M$ polynomial.

Key property. Can update hash function in constant time!

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

**Rabin-Karp substring search example**

```
// 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;
}
```
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) {
        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;
    }
}
```

Rabin-Karp: Java implementation (continued)

Monte Carlo version. Return match if hash match.

```java
public int search(String txt) {
    int M = 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>brute force</td>
<td>—</td>
<td>$M N$</td>
<td>yes</td>
<td>yes</td>
<td>1</td>
</tr>
<tr>
<td>Knuth-Morris-Pratt</td>
<td>full DFA (Algorithm 5.6)</td>
<td>$2 N$</td>
<td>no</td>
<td>yes</td>
<td>$MR$</td>
</tr>
<tr>
<td></td>
<td>mismatch transitions only</td>
<td>$3 N$</td>
<td>no</td>
<td>yes</td>
<td>$M$</td>
</tr>
<tr>
<td>Boyer-Moore</td>
<td>full algorithm</td>
<td>$3 N / M$</td>
<td>yes</td>
<td>yes</td>
<td>$R$</td>
</tr>
<tr>
<td></td>
<td>mismatched char heuristic only (Algorithm 5.7)</td>
<td>$MN / M$</td>
<td>yes</td>
<td>yes</td>
<td>$R$</td>
</tr>
<tr>
<td>Rabin-Karp†</td>
<td>Monte Carlo (Algorithm 5.8)</td>
<td>$7 N$</td>
<td>no</td>
<td>yes†</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Las Vegas</td>
<td>$7 N$</td>
<td>yes</td>
<td>yes</td>
<td>1</td>
</tr>
</tbody>
</table>

† probabilistic guarantee, with uniform hash function