5.3 Substring Search

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

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

Applications

• Parsers.
• Spam filters.
• Digital libraries.
• Screen scrapers.
• Word processors.
• Web search engines.
• Electronic surveillance.
• Natural language processing.
• Computational molecular biology.
• FBI’s Digital Collection System 3000.
• Feature detection in digitized images.

Application: Spam filtering

Identify patterns indicative of spam.

• PROFITS
• LOSE WEIGHT
• herbal Viagra
• There is no catch.
• LOW MORTGAGE RATES
• This is a one-time mailing.
• This message is sent in compliance with spam regulations.
• You’re getting this message because you registered with one of our marketing partners.
Application: 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

Application: Screen scraping

Goal. Extract relevant data from web page.

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

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=" + args[0];
    In in = new In(name);
    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
256.44
% java StockQuote msft
19.68
Brute-force substring search

Check for pattern starting at each text position.

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

Brute-force substring search

Brute-force substring search: Java implementation

Check for pattern starting at each text position.

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

Brute-force substring search: worst case

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

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

Brute-force substring search (worst case)

Worst case. ~ M N char compares.

Backup

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

Brute-force algorithm needs backup for every mismatch

```
A A A A A A A A A A A A A A A A A B
A A A A A A A A A A A A A A A A A B
A A A A A A A A A A A A A A A A A B
A A A A A A A A A A A A A A A A A B
```

Backup

Approach 1. Maintain buffer of size `m` (build backup into `StdIn`).
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).

```java
public static int search(String pat, String txt)
{
    int i, N = txt.length();
    int j, M = pat.length();
    for (i = 0, j = 0; i < N && j < M; i++)
    {
        if (txt.charAt(i) == pat.charAt(j)) j++;
        else { i -= j; j = 0;  }
    }
    if (j == M) return i - M;
    else            return N;
}
```

Algorithmic challenges in substring search

Brute-force is often not good enough.

Theoretical challenge. Linear-time guarantee. ← fundamental algorithmic problem

Practical challenge. Avoid backup in text stream. ← often no room or time to save text

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 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 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 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.

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! assuming {A, B} alphabet

Remark. It is always possible to avoid backup (!)
KMP substring search preprocessing (concept)

Q. What pattern char do we compare to the next text char on match?
A. Easy: compare next pattern char to next text char.

<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>j</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>pat.charAt(j)</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>0</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>dfa[i][j]</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>0</td>
<td>6</td>
</tr>
</tbody>
</table>

Table giving pattern char to compare to the next text char

KMP substring search preprocessing (concept)

Q. What pattern char do we compare to the next text char on mismatch?
A. Check each position, working from left to right.

Check each position, working from left to right.

What pattern char do we compare to the next text char on mismatch?
A. Check each position, working from left to right.

Fill in table columns by doing computation for each possible mismatch position.
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.

KMP substring search: trace

KMP search: Java implementation

KMP implementation. Build machine for pattern, simulate it on text.

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

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: at most \( N \) character accesses.
- Build DFA: at most \( M^2 R \) character accesses (stay tuned for better method).

KMP search: Java implementation

Key differences from brute-force implementation.
- Text pointer \( i \) never decrements.
- Need to precompute \( dfa[][] \) table from pattern.
- Could use input stream.

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; }

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 i; }
Constructing the DFA for KMP substring search: example

Q. What state $X$ would the DFA be in if it were restarted to correspond to shifting the pattern one position to the right?

A. Use the (partially constructed) DFA to find $X$!

Consequence.
- We want the same transitions as $X$ for the next state on mismatch.
- Copy $dfa[0]$ to $dfa[j]$.
- But a different transition (to $j+1$) on match.
- Set $dfa[pat.charAt(j)]$ to $j+1$.

Observation. No need to restart DFA.
- Remember last restart state in $X$.
- Use DFA to update $X$.
- $X = dfa[pat.charAt(j)][X]$.

DFA simulations to compute restart states for $A$ $B$ $A$ $B$ $A$ $C$.
Constructing the DFA for KMP substring search: Java implementation

For each $j$:
- Copy $\text{dfa}[][][X]$ to $\text{dfa}[][][j]$ for mismatch case.
- Set $\text{dfa}[\text{pat}.charAt(j)][j]$ to $j+1$ for match case.
- Update $X$.

Running time. $M$ character accesses.

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.** We access each pattern char once when constructing the DFA, and each text char once (in the worst case) when simulating the DFA.

**Remark.** Takes time and space proportional to $R \cdot M$ to construct $\text{dfa}[][]$, but with cleverness, can reduce time and space to $M$.

Knuth-Morris-Pratt: brief history

**Brief history.**
- Inspired by esoteric theorem of Cook.
- Discovered in 1976 independently by two theoreticians and a hacker.
  - Knuth: discovered linear-time algorithm
  - Pratt: made running time independent of alphabet
  - Morris: trying to build a text editor
- Theory meets practice.

```
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];
}
```
Boyer-Moore: mismatched character heuristic

Intuition.
- Scan characters in pattern from right to left.
- Can skip M text chars when finding one not in the pattern.

Mismatched character heuristic for right-to-left (Boyer-Moore) substring search

```plaintext
<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>
</tr>
</thead>
<tbody>
<tr>
<td>text</td>
<td>H A Y S T A C K N E E D L E I N A</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>0 5</td>
<td>N E E D L 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>
</tr>
<tr>
<td>5 5</td>
<td>N E E D L 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>
</tr>
<tr>
<td>8 5</td>
<td>N E E D L 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>
</tr>
<tr>
<td>8 0</td>
<td>return i = 8</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>
```

Boyermoore skip table computation

Q. How much to skip?
A. Compute right[c] = rightmost occurrence of character c in pat[].

```plaintext
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

```
<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>3</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>-1</td>
<td>-1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>L</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>M</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>N</td>
<td>-1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>
```

Q. How much to skip?
A. Compute right[c] = rightmost occurrence of character c in pat[].

```plaintext
increment i by j - right[‘N’] to line up text with ‘N’ in pattern
reset j to M-1
```
Boyer-Moore: mismatched character heuristic

Q. How much to skip?
A. Compute right[c] = rightmost occurrence of character c in pat[].

**Easy fix.** Set right[c] to -1 for characters not in pattern.

Boyer-Moore: analysis

**Property.** Substring search with the Boyer-Moore mismatched character heuristic takes about ~ 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 ~ M N.

**Boyer-Moore variant.** Can improve worst case to ~ 3 N by adding a KMP-like rule to guard against repetitive patterns.

---

**Boyer-Moore: Java implementation**

```java
public int search(String txt) {
    int N = txt.length();
    int M = pat.length();
    int skip = 0;
    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;
}
```

---

**Mismatched character heuristic (mismatch in pattern)**

increment i by j - right[‘N’] to line up text with ‘N’ in pattern
reset j to M-1

```
i j
. . . . . . . T L E . . . . . N E E DEL E
. . . . . . . T L E . . . . . N E E DEL E
. . . . . . . T L E . . . . . N E E DEL E
. . . . . . . T L E . . . . . N E E DEL E
. . . . . . . T L E . . . . . N E E DEL E

```

---

**Mismatched character heuristic (mismatch not in pattern)**

increment i by j+1 reset j to M-1

```
i j
. . . . . . . . T L E . . . . . N E E DEL E
. . . . . . . . T L E . . . . . N E E DEL E
. . . . . . . . T L E . . . . . N E E DEL E
. . . . . . . . T L E . . . . . N E E DEL E
. . . . . . . . T L E . . . . . N E E DEL E

```

---

compute skip value
Rabin-Karp fingerprint search

Basic idea.
- 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 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 int hash(String key) {
    int h = 0;
    for (int i = 0; i < M; i++)
        h = (R * h + key.charAt(i)) % Q;
    return h;
}
```

Efficiently computing the hash function

Challenge. How to efficiently compute \( x_{i+1} \) given that we know \( x_i \).

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

Key property. Can do it in constant time!

\[
x_{i+1} = (x_i - t_i R^{M-1}) R + I_{i+1}
\]
Rabin-Karp: Java implementation

```java
public class RabinKarp {
    private String pat; // the pattern
    private int patHash; // pattern hash value
    private int M; // pattern length
    private int Q = 8355967; // modulus
    private int R; // radix
    private int RM; // R^(M-1) % Q

    public RabinKarp(String pat) {
        this.R = 256;
        this.pat = pat;
        this.M = pat.length;
        RM = 1;
        for (int i = 1; i <= M-1; i++)
            RM = (R * RM) % Q;
        patHash = hash(pat);
    }

    private int hash(String key) {
        /* as before */
    }

    public int search(String txt) {
        int N = txt.length();
        if (N < M) return N;
        int offset = hashSearch(txt);
        if (offset == N) return N;
        for (int i = 0; i < M; i++)
            if (pat.charAt(i) != txt.charAt(offset + i))
                return N;
        return offset;
    }

    private int hashSearch(String txt) {
        int N = txt.length();
        int txtHash = hash(txt);
        if (patHash == txtHash) return 0;
        for (int i = M; i < N; i++)
            {  // precompute R^i % (mod Q)
                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 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>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>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</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 = (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 + 9*(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 + 9*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 = 4</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>

Rabin-Karp: Java implementation (continued)

```java
public int search(String txt) {
    int N = txt.length();
    if (N < M) return N;
    int offset = hashSearch(txt);
    if (offset == N) return N;
    for (int i = 0; i < M; i++)
        if (pat.charAt(i) != txt.charAt(offset + i))
            return N;
    return offset;
}
```

private int hashSearch(String txt) {
    int N = txt.length();
    int txtHash = hash(txt);
    if (patHash == txtHash) return 0;
    for (int i = M; i < N; i++)
        {  // check hash collision
            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 analysis

**Proposition.** Rabin-Karp substring search is extremely likely to be linear-time.

**Worst-case.** Takes time proportional to MN.
- In worst case, all substrings hash to same value.
- Then, need to check for match at each text position.

**Theory.** If Q is a sufficiently large random prime (about $MN^2$), then probability of a false collision is about $1/N \Rightarrow$ expected running time is linear.

**Practice.** Choose Q to avoid integer overflow. Under reasonable assumptions, probability of a collision is about $1/Q \Rightarrow$ linear in practice.
Rabin-Karp fingerprint search

Advantages.
- Extends to 2D patterns.
- Extends to finding multiple patterns.

Disadvantages.
- Arithmetic ops slower than char compares.
- Poor worst-case guarantee.
- Requires backup.

Q. How would you extend Rabin-Karp to efficiently search for any one of \( P \) possible patterns in a text of length \( N \)?

Cost of searching for an \( M \)-character pattern in an \( N \)-character text.

<table>
<thead>
<tr>
<th>Algorithm (data structure)</th>
<th>Operation count</th>
<th>Backup in input?</th>
<th>Space grows with</th>
</tr>
</thead>
<tbody>
<tr>
<td>brute force</td>
<td>( MN )</td>
<td>yes</td>
<td>1</td>
</tr>
<tr>
<td>Knuth-Morris-Pratt (full DFA)</td>
<td>2 ( N )</td>
<td>no</td>
<td>MR</td>
</tr>
<tr>
<td>Knuth-Morris-Pratt (mismatch transitions only)</td>
<td>3 ( N )</td>
<td>no</td>
<td>M</td>
</tr>
<tr>
<td>Boyer-Moore</td>
<td>3 ( N )</td>
<td>yes</td>
<td>( R )</td>
</tr>
<tr>
<td>Boyer-Moore (mismatched character heuristic only)</td>
<td>( MN )</td>
<td>yes</td>
<td>( R )</td>
</tr>
<tr>
<td>Rabin-Karp(^*)</td>
<td>7 ( N )</td>
<td>no</td>
<td>1</td>
</tr>
</tbody>
</table>

\(^*\) probabilistic guarantee, with uniform hash function