5.3 **Substring Search**

- introduction
- brute force
- Knuth-Morris-Pratt
- Boyer-Moore
- Rabin-Karp

---

**Substring search**

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

- Pattern: `NEEDLE`
- Text: `INAHAYSTACK NEEDLE INA`

---

**Substring search applications**

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

- Pattern: `NEEDLE`
- Text: `INAHAYSTACK NEEDLE INA`

---

Typically $N \gg M$.
**Substring search applications**

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

- **pattern** — NEEDLE
- **text** — INAHAYSTACK NEEDLE INA
- **match** — typically $N >> M$

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

![Image of computer forensics](http://citp.princeton.edu/memory)

**Identify patterns indicative of spam.**
- PROFITS
- LOSE WEIGHT
- herbal Viagra
- There is no catch.
- This is a one-time mailing.
- This message is sent in compliance with spam regulations.

![Image of spam](SpamAssassin.png)

**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

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

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

```html
...<tr>
<td class="yfnc_tablehead1" width="48%" align="right">Last Trade:</td>
<td class="yfnc_tabledata1">452.92</td>
</tr>
```

![Stock chart](http://finance.yahoo.com/q?s=goog)
Screen scraping: Java implementation

Java library. The `indexOf()` method in Java’s String data type returns the index of the first occurrence of a given string, starting at a given offset.

```java
public class StockQuote {
    public static void main(String[] args) {
        String name = "http://finance.yahoo.com/q?s=";
        In in = new In(name + args[0]);
        String text = in.readAll();
        int start = text.indexOf("Last Trade:", 0);
        int to = text.indexOf("</b>", start);
        String price = text.substring(from + 3, to);
        StdOut.println(price);
    }
}
```

---

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

```

```

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;  // index in text where pattern starts
    }
    return N;  // not found
}
```
Substring search quiz 1

What is the worst-case running time of brute-force substring search as a function of the number of characters in the pattern $M$ and text $N$?

A. $M + N$
B. $M^2$
C. $MN$
D. $N^2$
E. I don’t know.

Backup

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

Brute-force algorithm needs backup for every mismatch.

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

Brute-force substring search: 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>4</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>5</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>5</td>
<td>10</td>
</tr>
</tbody>
</table>

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

Worst case. $\sim MN$ char compares.
```

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 # 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 &amp; j < M; i++)
        
        if (txt.charAt(i) == pat.charAt(j)) j++;
        else { i --; j = 0; }
    }
    if (j == M) return i - M;
    else return N;
}
```

```
A B A C A D A B R A C
7 3
5 0
```

```
shift pattern right one position
'''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
Algorithmic challenges in substring search

Brute-force is not always 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

Knuth-Morris-Pratt substring search

Intuition. Suppose we are searching in text for pattern BAAAAAAAAAAA.

• Suppose we match 5 chars in pattern, with mismatch on 6th char.
• We know previous 6 chars in text are BAAAAB.
• Don't need to back up text pointer!

Assuming { A, B } alphabet

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.

Internal representation

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

Deterministic finite state automaton (DFA)
**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>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>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>B</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>4</td>
</tr>
</tbody>
</table>

<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>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**

**Knuth-Morris-Pratt demo: DFA simulation**

**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].

**Interpretation of Knuth-Morris-Pratt DFA**

length of longest prefix of pat[] that is a suffix of txt[0..i]
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++)
        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 demo: DFA construction

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

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

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

Knuth-Morris-Pratt demo: DFA construction

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

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

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

```java
public int search(In in) {
    int i, j;
    for (i = 0, j = 0; !in.isEmpty() && j < M; i++)
        if (j == M) return i - M;
    else return NOT_FOUND;
}
```
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 $j+1$.

- First $j$ characters of pattern have already been matched
- Next char matches
- Now first $j + 1$ characters of pattern have been matched

**Mismatch transition.** If in state $j$ and next char $c \neq \text{pat.charAt}(j)$, then the last $j-1$ characters of input are $\text{pat}[1..j-1]$, followed by $c$.

To compute $\text{dfa}[c][j]$:
- Simulate $\text{pat}[1..j-1]$ on DFA and take transition $c$.
- Seems to require $j$ steps.

Running time. Seems to require $j$ steps.

Running time. Takes only constant time if we maintain state $x$.

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

 simulate BABA; take transition 'A'
  = $\text{dfa}[\text{'A'}][3]$

 simulate BABA; take transition 'B'
  = $\text{dfa}[\text{'B'}][3]$

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

 from state $x$,
 take transition 'A'
  = $\text{dfa}[\text{'A'}][x]$

 from state $x$,
 take transition 'B'
  = $\text{dfa}[\text{'B'}][x]$

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

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

 from state $x$,
 take transition 'A'
  = $\text{dfa}[\text{'A'}][x]$

 from state $x$,
 take transition 'B'
  = $\text{dfa}[\text{'B'}][x]$

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

from state $x$;

take transition 'A'
  = $\text{dfa}[\text{'A'}][x]$

from state $x$;

take transition 'B'
  = $\text{dfa}[\text{'B'}][x]$

from state $x$;

take transition 'C'
  = $\text{dfa}[\text{'C'}][x]$
Knuth-Morris-Pratt demo: DFA construction in linear time

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

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

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

0 1 2 3 4 5

Constructing the DFA for KMP substring search: Java implementation

For each state j:
- Copy dfa[][][x] to dfa[][][j] for mismatch case.
- Set dfa[pat.charAt(j)][j] to j+1 for match case.
- Update x.

```java
public KMP(String pat)
{
    this.pat = pat;
    M = pat.length();
    dfa = new int[R][M];
    dfa[pat.charAt(0)][0] = 1;
    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];
    }
}
```

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 dfa[][] in time and space proportional to $R M$.

Larger alphabets. Improved version of KMP constructs 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 algorithm
  - Pratt: made running time independent of alphabet size
  - Morris: built a text editor for the CDC 6400 computer
- Theory meets practice.

Cyclic Rotation

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

<table>
<thead>
<tr>
<th>yes</th>
<th>no</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROATEDSTRING</td>
<td>ROTATEDSTRING</td>
</tr>
<tr>
<td>STRINGROTATED</td>
<td>GNITSDETAOT</td>
</tr>
</tbody>
</table>

Problem. Given two binary strings $s$ and $t$, design a linear-time algorithm to determine if $s$ is a cyclic rotation of $t$.

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.
Boyer-Moore: mismatched character heuristic

Q. How much to skip?

Case 1. Mismatch character not in pattern.

before
\[
\begin{align*}
txt & \quad \quad \quad T \quad L \\ pat & \quad N \quad E \quad E \quad D \quad L \quad E
\end{align*}
\]

after
\[
\begin{align*}
txt & \quad \quad \quad T \quad L \\ pat & \quad N \quad E \quad E \quad D \quad L \quad E
\end{align*}
\]

mismatch character 'T' not in pattern: increment i one character beyond 'T'

Case 2a. Mismatch character in pattern.

before
\[
\begin{align*}
txt & \quad \quad \quad N \quad L \quad E \\ pat & \quad N \quad E \quad E \quad D \quad L \quad E
\end{align*}
\]

after
\[
\begin{align*}
txt & \quad \quad \quad N \quad L \quad E \\ pat & \quad N \quad E \quad E \quad D \quad L \quad E
\end{align*}
\]

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

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

before
\[
\begin{align*}
txt & \quad \quad \quad E \quad L \quad E \\ pat & \quad N \quad E \quad E \quad D \quad L \quad E
\end{align*}
\]

aligned with rightmost E?
\[
\begin{align*}
txt & \quad \quad \quad E \quad L \quad E \\ pat & \quad N \quad E \quad E \quad D \quad L \quad E
\end{align*}
\]

mismatch character 'E' in pattern: align text 'E' with rightmost pattern 'E'?

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

before
\[
\begin{align*}
txt & \quad \quad \quad E \quad L \quad E \\ pat & \quad N \quad E \quad E \quad D \quad L \quad E
\end{align*}
\]

after
\[
\begin{align*}
txt & \quad \quad \quad E \quad L \quad E \\ pat & \quad N \quad E \quad E \quad D \quad L \quad E
\end{align*}
\]

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)

Precompute index of rightmost occurrence of character $c$ in pattern.

$$right = \text{new int}[R];$$
$$\text{for } (\text{int } c = 0; c < R; c++)$$
$$\text{right}[c] = -1;$$
$$\text{for } (\text{int } j = 0; j < M; j++)$$
$$\text{right}[\text{pat.charAt}(j)] = j;$$

$$\begin{array}{ccccccc}
\text{c} & 0 & 1 & 2 & 3 & 4 & 5 \\
A & -1 & -1 & -1 & -1 & -1 & -1 \\
B & -1 & -1 & -1 & -1 & -1 & -1 \\
C & -1 & -1 & -1 & -1 & -1 & -1 \\
D & -1 & -1 & -1 & -1 & ~3 & ~3 \\
E & -1 & -1 & 1 & 2 & 2 & 2 \\
\ldots & & & & & & \\
\end{array}$$

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;
}
```

Boyer-Moore: analysis

Property. Substring search with the Boyer-Moore mismatched character heuristic takes about $\sim N/M$ character compares to search for a pattern of length $M$ in a text of length $N$.

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

5.3 Substring Search

- introduction
- brute force
- Knuth-Morris-Pratt
- Boyer-Moore
- Rabin-Karp

Boyer-Moore variant. Can improve worst case to $\sim 3N$ character compares by adding a KMP-like rule to guard against repetitive patterns.
Rabin-Karp fingerprint search

Basic idea = modular hashing.
- Compute a hash of $\text{pat}[0..M-1]$.
- For each $i$, compute a hash of $\text{txt}[i..M+i-1]$.
- If pattern hash = text substring hash, check for a match.

```java
pat.charAt(i)
0 1 2 3 4
2 6 5 3 5 % 997 = 613
```

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

Modular arithmetic

**Math trick.** To keep numbers small, take intermediate results modulo $Q$.

**Ex.**
- $(10000 + 535) \times 1000 \equiv (30 + 535) \times 3 \pmod{997}$
- $10000 \bmod{997} = 3$
- $1000 \bmod{997} = 3$
- $= 1695 \pmod{997}$
- $= 698 \pmod{997}$

$$(a + b) \bmod{Q} = ((a \bmod{Q}) + (b \bmod{Q})) \bmod{Q}$$

$$(a \times b) \bmod{Q} = ((a \bmod{Q}) \times (b \bmod{Q})) \bmod{Q}$$

two useful modular arithmetic identities

Efficiently computing the hash function

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

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

**Intuition.** $M$-digit, base-$R$ integer, modulo $Q$.

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

```java
private long hash(String key, int M) {
    long h = 0;
    for (int j = 0; j < M; j++)
        h = (h * R + key.charAt(j)) % Q;
    return h;
}
```

```java
26535 = 2*10000 + 6*1000 + 5*3 + 10 + 5
= (((2) *10 + 6) * 10 + 5) * 10 + 5
```

Efficiently computing the hash function

**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 “rolling” hash function in constant time!

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

```java
i... 2 3 4 5 6 7...
current value 4 1 5 9 2 6 5
new value 4 1 5 9 2 6 5
```

(text)

```java
4 1 5 9 2 6 5
- 4 0 0 0 0
subtract leading digit
* 1 0
1 5 9 2 0
+ 6
add new trailing digit
1 5 9 2 6
```
**Rabin-Karp substring search example**

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

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

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

\[
\text{Rabin-Karp substring search example (continued)}
\]

**Monte Carlo version.** Return match if hash match.

```java
public int search(String txt) {
    int N = txt.length();
    int txtHash = hash(txt, M);
    if (pathHash == txtHash) return 0;
    for (int i = M; i < N; i++) {
        txtHash = (txtHash + Q - R*M*txt.charAt(i-M) % Q) % Q;
        if (pathHash == txtHash) return i - M + 1;
    }
    return N;
}
```

**Las Vegas version.** Modify code to check for substring match if hash match; continue search if false collision.

**Rabin-Karp: Java implementation**

```java
public class RabinKarp {
    private long patHash; // pattern hash value
    private int M; // pattern length
    private long Q; // modulus
    private int R; // radix
    private long RM1; // R^(M-1) % Q

    public RabinKarp(String pat) {
        M = pat.length();
        R = 256;
        Q = 1000000007;
        RM1 = 1;
        for (int i = 1; i <= M-1; i++)
            RM1 = (R * RM1) % Q;
        patHash = hash(pat, M);
    }

    private long hash(String key, int M) {
        long hashValue = 0;
        for (int i = 0; i < M; i++)
            hashValue = (hashValue * R + key.charAt(i)) % Q;
        return hashValue;
    }

    public int search(String txt) {
        // as before
    }
}
```

**Rabin-Karp analysis**

**Theory.** If \( Q \) is a sufficiently large random prime (about \( M \cdot N^2 \)), then the probability of a false collision is about \( 1 / N \).

**Practice.** Choose \( Q \) to be a large prime (but not so large to cause overflow). Under reasonable assumptions, probability of a collision is about \( 1 / Q \).

**Monte Carlo version.**
- Always runs in linear time.
- Extremely likely to return correct answer (but not always!).

**Las Vegas version.**
- Always returns correct answer.
- Extremely likely to run in linear time (but worst case is \( M \cdot N \)).
Rabin-Karp fingerprint search

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

Disadvantages.
- Arithmetic ops slower than char compares.
- Las Vegas version requires backup.
- Poor worst-case guarantee.

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

Substring search cost summary

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

<table>
<thead>
<tr>
<th>algorithm</th>
<th>version</th>
<th>operation count</th>
<th>backup in input?</th>
<th>correct?</th>
<th>extra space</th>
</tr>
</thead>
<tbody>
<tr>
<td>brute force</td>
<td>—</td>
<td>( MN )</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 )</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 )</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>( \text{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