5.3 Substring Search

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

Substring search quiz 0

Do any of the algorithms we’ve studied so far have a running time that’s a decreasing function of the input size?

A. Yes
B. No
C. Haha no way
D. I don’t know.

Substring search

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

pattern — NEEDLE

text — INAHAYSTACKNEEDLEINAN

match
### Substring search applications

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

- Typical $N \gg M$

- **Pattern** → NEEDLE
- **Text** → INAHAYSTACK NEEDLE INA
- **Match**

### 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

### Electronic surveillance

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

### Latest censored keywords in China

- Female infant + vaccine + die
- Hebei + female infant + vaccine
- Panama
- Banana (Panama)
- banana (Panama)
- [Panama] Canal Papers
- [Panama] Papers
- launder money + brother-in-law
- Xi + brother-in-law
- top Chinese official + offshore
- Wen [Jiabao] clan
- Xi + explode
- Wanda + bigwig
- Leshi + [Jia] Yueting
- 50 cents + internet commentary

From http://chinadigitaltimes.net/2013/06/grass-mud-horse-list/
5.3 Substring Search

Substring search quiz 1

Suppose you want to count the number of all occurrences of some pattern string of length \( M \) in a text of length \( N \). What is the order of growth of the best-case and worst-case running time of the brute-force algorithm? Assume \( M \leq N \).

A. \( N \) and \( MN \)
B. \( N \) and \( MN^2 \)
C. \( MN \) and \( MN \)
D. \( MN \) and \( MN^2 \)
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. Streaming algorithm.
5.3 Substring Search

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

Searching for BAAAAAAA

Let \( j = \# \text{characters matched so far} \).

When \( j = 0 \):
- If we see 'A': \( j \) remains 0
- If we see 'B': \( j \) becomes 1

When \( 1 \leq j < 10 \):
- If we see 'A': \( j = j + 1 \)
- If we see 'B': \( j = 1 \)

\( j = 10 \): match!

Properties of state transition matrix:
- Depends only on pattern, not text
- \#rows = alphabet size
- \#columns = length of pattern
- In each col, exactly one row lets us increment the state \( j \)

Knuth–Morris–Pratt substring search

Intuition. Suppose we are searching in text for pattern BAAAAAAA.
- 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!

Knuth–Morris–Pratt algorithm. Clever method to always avoid backup!

Exercise

Construct the state transition matrix for the pattern ABABAC.

e.g., \( j = 3 \): we’ve matched the string 'ABA' in the text (XXXXXABA)
- If we see 'A': we’ve matched 'A' (XXXXXABAA) \( \Rightarrow j \) becomes 1
- If we see 'B': we’ve matched 'ABAB' (XXXXXABAB) \( \Rightarrow j \) becomes 4
- If we see 'C': we’ve matched nothing (XXXXXABAC) \( \Rightarrow j \) becomes 0

\[
\begin{array}{cccccc}
A & B & A & B & A & C \\
\hline
j & 0 & 1 & 2 & 3 & 4 & 5 \\
A & 1 &  &  &  &  &  \\
B & 4 &  &  &  &  &  \\
C & 0 &  &  &  &  &  \\
\end{array}
\]

old:

\[
\begin{array}{cccccc}
\hline
j & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
A & 0 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
B & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
\end{array}
\]
Construct the state transition matrix for the pattern ABABAC.

e.g., \( j = 3\): we've matched the string 'ABA' in the text (XXXXXXXXABA)
  - If we see 'A': we've matched 'A' (XXXXXXXXABA) ⇒ \( j \) becomes 1
  - If we see 'B': we've matched 'ABAB' (XXXXXXXXABAB) ⇒ \( j \) becomes 4
  - If we see 'C': we've matched nothing (XXXXXXXXABAC) ⇒ \( j \) becomes 0

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

old:

<table>
<thead>
<tr>
<th></th>
<th>B</th>
<th>A</th>
<th>A</th>
<th>A</th>
<th>A</th>
<th>A</th>
<th>A</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>
<td>6</td>
</tr>
<tr>
<td>A</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</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>
</tbody>
</table>

**Deterministic finite state automaton (DFA)**

DFA is abstract string-searching machine.
- Finite number of states (including start and halt).
- Exactly one state transition for each char in alphabet.
- Accept if sequence of state transitions leads to halt state.

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

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>A</th>
<th>B</th>
<th>A</th>
<th>C</th>
<th>A</th>
<th>A</th>
<th>A</th>
<th>A</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>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dfa<a href="c">( j )</a></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Knuth–Morris–Pratt demo: DFA simulation
Knuth–Morris–Pratt demo: 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
dfa[][j] B 0 2 0 4 0 4
C 0 0 0 0 0 6

Knuth–Morris–Pratt demo: 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
dfa[][j] B 0 2 0 4 0 4
C 0 0 0 0 0 6

Knuth–Morris–Pratt demo: 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
dfa[][j] B 0 2 0 4 0 4
C 0 0 0 0 0 6

Knuth–Morris–Pratt demo: 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
dfa[][j] B 0 2 0 4 0 4
C 0 0 0 0 0 6
Knuth–Morris–Pratt demo: DFA simulation

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

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

```
dfa[][][] A    B    A    B
B    0    2    0    4    0    4
C    0    0    0    0    0    6
```

Knuth–Morris–Pratt demo: DFA simulation

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

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

```
dfa[][][] A    B    A    B
B    0    2    0    4    0    4
C    0    0    0    0    0    6
```
Knuth–Morris–Pratt demo: DFA simulation

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

pat.charAt(i) | 0 1 2 3 4 5
---|---
A B A B A C
A 1 1 3 1 5 1
dfa[i][] | 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].

Substring search quiz 2

Which state is the DFA in after processing the following input?

A. 0
B. 1
C. 3
D. 4
E. I don’t know.

Knuth–Morris–Pratt substring search: Java implementation

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

```java
public int search(String txt) {
    int i, j, N = txt.length();
    for (i = 0, j = 0; i < N && j < M; i++)
        j = dfa[txt.charAt(i)][]];
    if (j == M) return i - M;
    else return N;
}
```
Knuth–Morris–Pratt substring search: Java implementation

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 = = W) return i - W;
    if (j = = W) return NOT_FOUND;
    return NOT_FOUND;
}
```

Knuth–Morris–Pratt substring search analysis

Proposition. KMP substring search accesses no more than $M + N$ chars to search for a pattern of length $M$ in a text of length $N$.

Pf. Each pattern character accessed once when constructing the DFA; each text character accessed once (in the worst case) when simulating the DFA.

Proposition. KMP constructs DFA[][] in time and space proportional to $RM$.

Larger alphabets. Improved version of KMP constructs DFA[][] in time and space proportional to $M$.

Knuth–Morris–Pratt Running time

Running time.
- Simulate DFA on text: at most $N$ character accesses.
- Build DFA: how to do efficiently? See textbook/video.
  - In the vast majority of applications, the running time of building the DFA is irrelevant. [Arvind’s opinion.]

Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered.

We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.

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.

In 1977, Donald E. Knuth, James H. Morris, Jr. and Vaughan R. Pratt published their classic paper “Fast Pattern Matching in Strings” in the SIAM Journal on Computing. The algorithm they described, now known as the KMP algorithm, is a cornerstone of string matching and has had a profound impact on computer science. It has applications in bioinformatics, where it is used to search for patterns in DNA sequences, and in text processing, where it is used to efficiently search for substrings in large documents. The KMP algorithm is a testament to the power of theoretical research in solving practical problems, and it remains a fundamental tool in the computer scientist’s toolkit today.
Cyclic Rotation

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

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

---

**Boyer–Moore: mismatched character heuristic**

**Intuition.**

- Scan characters in pattern from right to left.
- Can skip as many as $\Delta$ text chars when finding one not in the pattern.

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

- **Before**
  - $i$ before mismatch character 'T' not in pattern: increment $i$ one character beyond 'T'

- **After**
  - $i$ after mismatch character 'T' not in pattern: increment $i$ one character beyond 'T'
**Boyer–Moore: mismatched character heuristic**

**Case 2a.** Mismatch character in pattern.

**before**

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

**after**

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

mismatch character 'N' in pattern: align text 'N' with rightmost (why?) pattern 'N'

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

**before**

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

**after**

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

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

---

**Boyzer-Moore skip table computation**

<table>
<thead>
<tr>
<th>c</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>-1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>-1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>-1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>L</td>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N</td>
<td>-1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(mismatch character 'c' in pattern: increment i by right(c))
Boyer–Moore: analysis

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

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

Q. What’s the worst-case input?

<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></td>
<td>txt</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td>0</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 \( \sim 3N \) character compares by adding a KMP-like rule to guard against repetitive patterns.

5.3 Substring search

- Introduction
- Brute force
- Knuth–Morris–Pratt
- Boyer–Moore
- Rabin–Karp

**Simplified example**

Assume 10-character alphabet: abcdefghij

Text: beachheadacidifiedjadedbeheadbeefheadbeef
Pattern: beheaded

"Hash" of string: number obtained replacing each char by corresponding digit

\[ h_0 = 14727740 \quad \text{Precompute hash of pattern: } h = 14740343 \]

Compute \( h_0, h_1, h_2, \ldots \). Match if \( h = h_0 \).

**Simplified example**

Assume 10-character alphabet: abcdefghij

Text: beachheadacidifiedjadedbeheadbeheadbeef
Pattern: beheaded

"Hash" of string: number obtained replacing each char by corresponding digit

\[ h_0 = 14727740 \quad \text{Precompute hash of pattern: } h = 14740343 \]

\[ h_1 = 47277403 \]
**Simplified example**

Assume 10-character alphabet: abcdefghij

Text: beachheadacidifiedheadbeheadbeef

Pattern: beheaded

“Hash” of string: number obtained replacing each char by corresponding digit

<table>
<thead>
<tr>
<th>b</th>
<th>e</th>
<th>a</th>
<th>c</th>
<th>h</th>
<th>e</th>
<th>a</th>
<th>d</th>
<th>a</th>
<th>c</th>
<th>i</th>
<th>d</th>
<th>f</th>
<th>i</th>
<th>e</th>
<th>d</th>
<th>j</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>7</td>
<td>2</td>
<td>7</td>
<td>4</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>2</td>
<td>8</td>
<td>3</td>
<td>8</td>
<td>5</td>
<td>8</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>7</td>
<td>4</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>b</td>
<td>e</td>
<td>h</td>
<td>e</td>
<td>a</td>
<td>d</td>
<td>e</td>
<td>d</td>
<td></td>
</tr>
</tbody>
</table>

$h_1 = 47277403$

Precompute hash of pattern: $h = 14740343$

$h_2 = 72774030$

Q. Express $h_{h+i}$ in terms of $h$, $i\{0..N\}$ (digits corresponding to text) and M

A. $h_{h+i} = (h - i \cdot 10^{|i|}) \cdot 10 + i \cdot M$

**Rabin–Karp fingerprint search**

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

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

| txt.charAt(i) | 1 | 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 | % 997 = 508 |
| 1 | 1 | 4 | 1 | 5 | 9 | % 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 |

**Basic idea of Rabin-Karp**

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

<table>
<thead>
<tr>
<th>b</th>
<th>e</th>
<th>a</th>
<th>c</th>
<th>h</th>
<th>e</th>
<th>a</th>
<th>d</th>
<th>a</th>
<th>c</th>
<th>i</th>
<th>d</th>
<th>f</th>
<th>i</th>
<th>e</th>
<th>d</th>
<th>j</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>7</td>
<td>2</td>
<td>7</td>
<td>4</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>2</td>
<td>8</td>
<td>3</td>
<td>8</td>
<td>5</td>
<td>8</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>7</td>
<td>4</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>b</td>
<td>e</td>
<td>h</td>
<td>e</td>
<td>a</td>
<td>d</td>
<td>d</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Problem 1: alphabet size $R$ may not be 10
Problem 2: integer overflow if $M$ is too long ($M \geq 10$ for 32-bit ints)

Solution 1: use base $R$

**Modular arithmetic**

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

Ex.

<table>
<thead>
<tr>
<th>$(10000 + 535) \cdot 1000 \mod 997$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10000 \mod 997 = 10$</td>
</tr>
<tr>
<td>$997 \cdot 3 \mod 997$</td>
</tr>
<tr>
<td>$= 1971 \mod 997$</td>
</tr>
<tr>
<td>$= 608 \mod 997$</td>
</tr>
</tbody>
</table>

For more depth

- take COS 340

```
(a + b) mod Q = ((a mod Q) + (b mod Q)) mod Q
(a \cdot b) mod Q = ((a mod Q) \ast (b mod Q)) mod Q
```

two useful modular arithmetic identities
Efficiently computing the hash function

Modular hash function. Using the notation \( a \) for \( \text{txt.charAt}(i) \), we wish to compute
\[
x_i = a_i R^{i-1} + a_{i-1} R^{i-2} + \ldots + a_1 R^0 \pmod{Q}
\]

Intuition. \( M \)-digit, base-\( R \) integer, modulo \( Q \).

Horner's method. Linear-time method to evaluate degree-\( M \) polynomial.

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 = (long)RandomPrime();
    RM1 = 1;
    for (int i = 1; i < M; i++)
      RM1 = (R + RM1) % Q;
    patHash = hash(pat, M);
  }

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

  public int search(String txt) {
    // see next slide */
  }
}
```

Key property. Can update "rolling" hash function in constant time!
\[
x_{i+1} = \left( x_i - \left( R^{M-1} \times t_i \right) \right) R + t_{i+1}
\]

Rabin-Karp: Java implementation (continued)

Monte Carlo version. Return match if hash match.

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

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

**Theory.** If $Q$ is a sufficiently large random prime (about $M N^2$), 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 $M N$).

---

**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 guarantee</th>
<th>backup in typical</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>
<td></td>
</tr>
<tr>
<td>Knuth-Morris-Pratt</td>
<td>full DFA (Algorithm 5.6)</td>
<td>$2N$</td>
<td>no</td>
<td>yes</td>
<td>$MB$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>transitions only</td>
<td>$3N$</td>
<td>no</td>
<td>yes</td>
<td>$M$</td>
<td></td>
</tr>
<tr>
<td>Boyer-Moore</td>
<td>full algorithm</td>
<td>$3N$</td>
<td>yes</td>
<td>yes</td>
<td>$R$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>mismatched char</td>
<td>$3N$</td>
<td>yes</td>
<td>yes</td>
<td>$R$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>heuristic only (Algorithm 5.7)</td>
<td>$MN$</td>
<td>yes</td>
<td>yes</td>
<td>$R$</td>
<td></td>
</tr>
<tr>
<td>Rabin-Karp*</td>
<td>Monte Carlo (Algorithm 5.8)</td>
<td>$7N$</td>
<td>yes</td>
<td>yes*</td>
<td>$1$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Las Vegas</td>
<td>$7N^2$</td>
<td>yes</td>
<td>yes</td>
<td>$1$</td>
<td></td>
</tr>
</tbody>
</table>

*probabilistic guarantee, with uniform hash function

---

**Rabin–Karp fingerprint search**

**Advantages.**
- Extends to two-dimensional patterns.
- Extends to finding multiple patterns.

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

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

---

**Substring search quiz**

Which of today’s algorithms do you like the best?

A. Knuth-Morris-Pratt (finite automaton).
B. Boyer–Moore (skip–ahead heuristic).
C. Rabin-Karp (rolling hash function).
D. It’s all a blur.