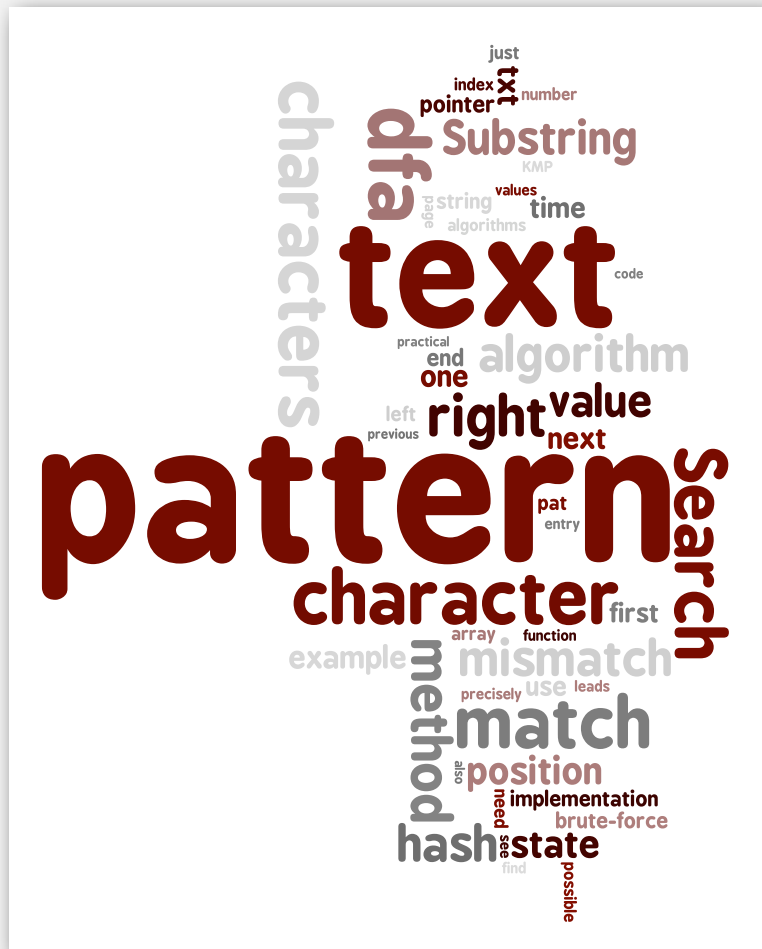


5.3 Substring Search



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

Substring search

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

typically $N \gg M$

pattern → N E E D L E

text \longrightarrow I N A H A Y S T A C K N E E D L E I N A

\uparrow

match

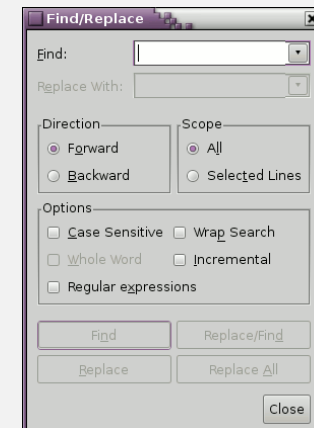
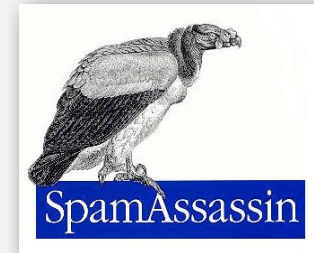
Computer forensics. Search memory or disk for signatures, e.g., all URLs or RSA keys that the user has entered.



<http://citp.princeton.edu/memory>

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.



Application: electronic surveillance



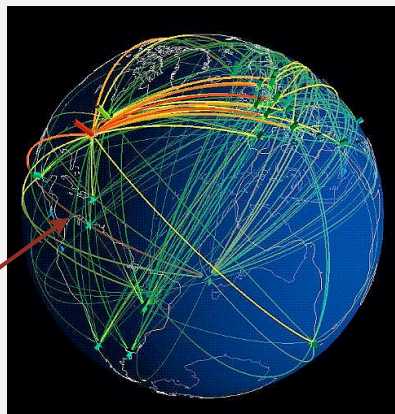
Need to monitor all
internet traffic.
(security)



Well, we're mainly
interested in
"ATTACK AT DAWN"

"ATTACK AT DAWN"
substring search
machine

found



No way!
(privacy)



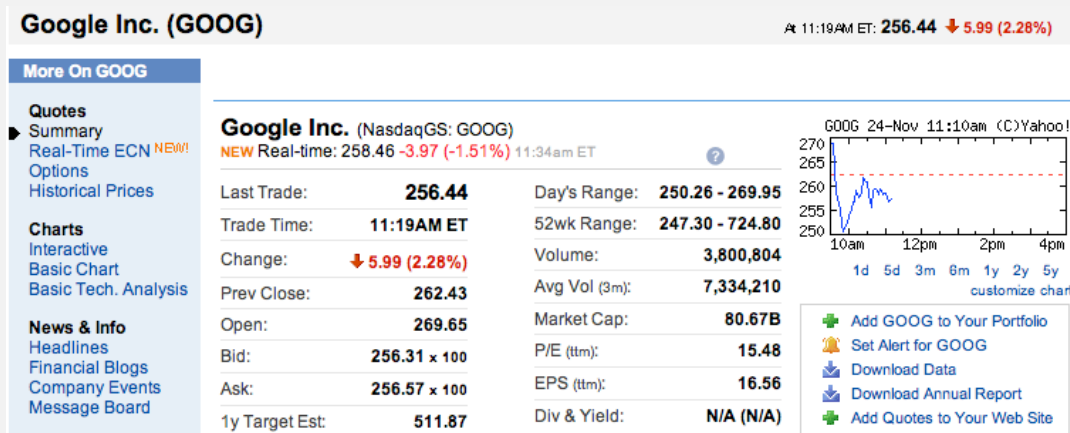
OK. Build a
machine that just
looks for that.



Application: screen scraping

Goal. Extract relevant data from web page.

Ex. Find string delimited by `` and `` after first occurrence of pattern `Last Trade:`.



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

```
...  
<tr>  
<td class= "yfnc_tablehead1"  
width= "48%">  
Last Trade:  
</td>  
<td class= "yfnc_tabledata1">  
<big><b>452.92</b></big>  
</td></tr>  
<td class= "yfnc_tablehead1"  
width= "48%">  
Trade Time:  
</td>  
<td class= "yfnc_tabledata1">  
...
```

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.

```
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
564.35
```

```
% java StockQuote msft
26.04
```

- ▶ **brute force**
- ▶ Knuth-Morris-Pratt
- ▶ Boyer-Moore
- ▶ Rabin-Karp

Brute-force substring search

Check for pattern starting at each text position.

i	j	i+j	0	1	2	3	4	5	6	7	8	9	10
txt →			A	B	A	C	A	D	A	B	R	A	C
0	2	2	A	B	R	A	← pat						
1	0	1		A	B	R	A	entries in red are mismatches					
2	1	3			A	B	R	A	entries in gray are for reference only				
3	0	3				A	B	R	A	entries in black match the text			
4	1	5					A	B	R	A	entries in gray are for reference only		
5	0	5						A	B	R	A	entries in black match the text	
6	4	10							A	B	R	A	
return i when j is M													

Brute-force substring search: Java implementation

Check for pattern starting at each text position.

$i = 4, j = 3$

0	1	2	3	4	5	6	7	8	9	10
A	B	A	C	A	D	A	B	R	A	C
				A	D	A	C	R		

```
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;
}
```

← index in text where pattern starts

← not found

Brute-force substring search: worst case

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

i	j	i+j	0	1	2	3	4	5	6	7	8	9
			txt → A A A A A A A A A A B									
0	4	4	A	A	A	A	B	← pat				
1	4	5		A	A	A	A	B				
2	4	6			A	A	A	A	B			
3	4	7				A	A	A	A	B		
4	4	8					A	A	A	A	B	
5	5	10						A	A	A	A	B

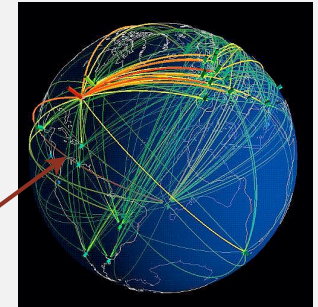
Worst case. $\sim MN$ char compares.

Backup

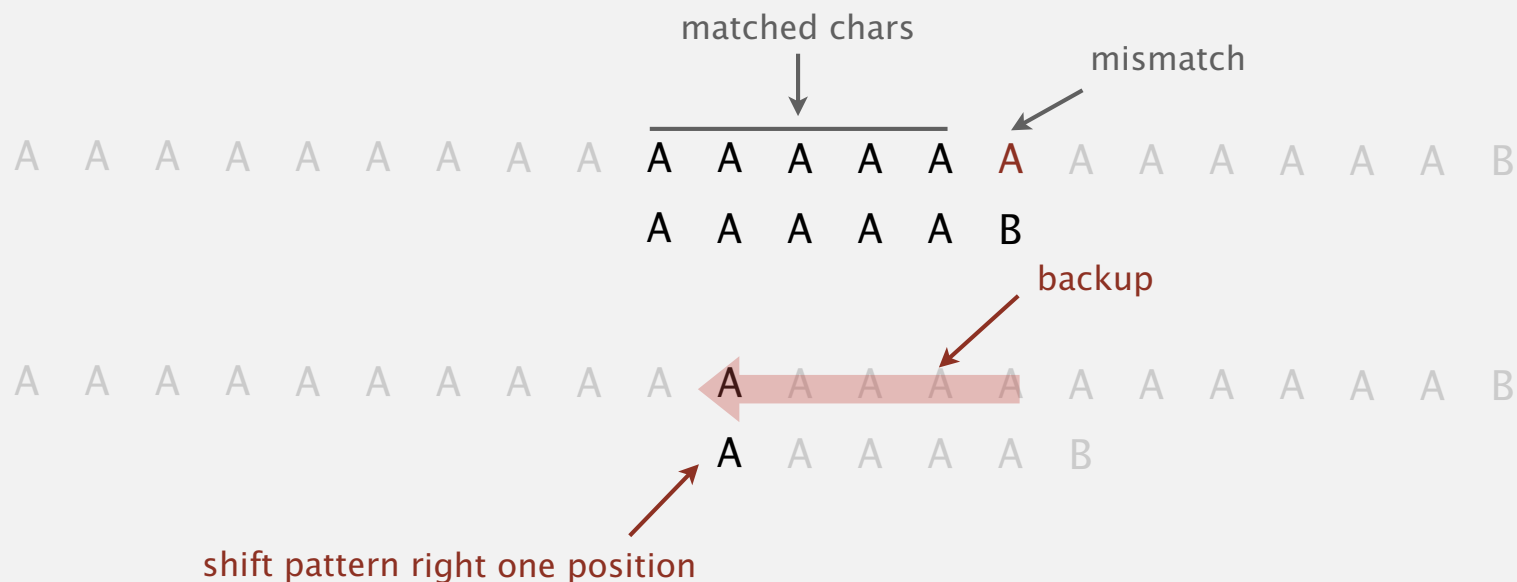
In typical applications, we want to avoid **backup** in text stream.

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

“ATTACK AT DAWN”
substring search
machine
found



Brute-force algorithm needs backup for every mismatch.



Approach 1. Maintain buffer of size M (build backup into standard input).

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).

$i = 6, j = 3$

0	1	2	3	4	5	6	7	8	9	10
A	B	A	C	A	D	A	B	R	A	C
				A	D	A	C	R		

```
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;
}
```

← backup

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 many good people to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for a lot of good people to come to the aid of their party. Now is the time for all of the good people to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for each good person to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for all good Republicans to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for many or all good people to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for all good Democrats to come to the aid of their party. Now is the time for all people to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for many good people to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for a lot of good people to come to the aid of their party. Now is the time for all of the good people to come to the aid of their party. Now is the time for all good people to come to the aid of their **attack at dawn** party. Now is the time for each person to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for all good Republicans to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for many or all good people to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for all good Democrats to come to the aid of their party.

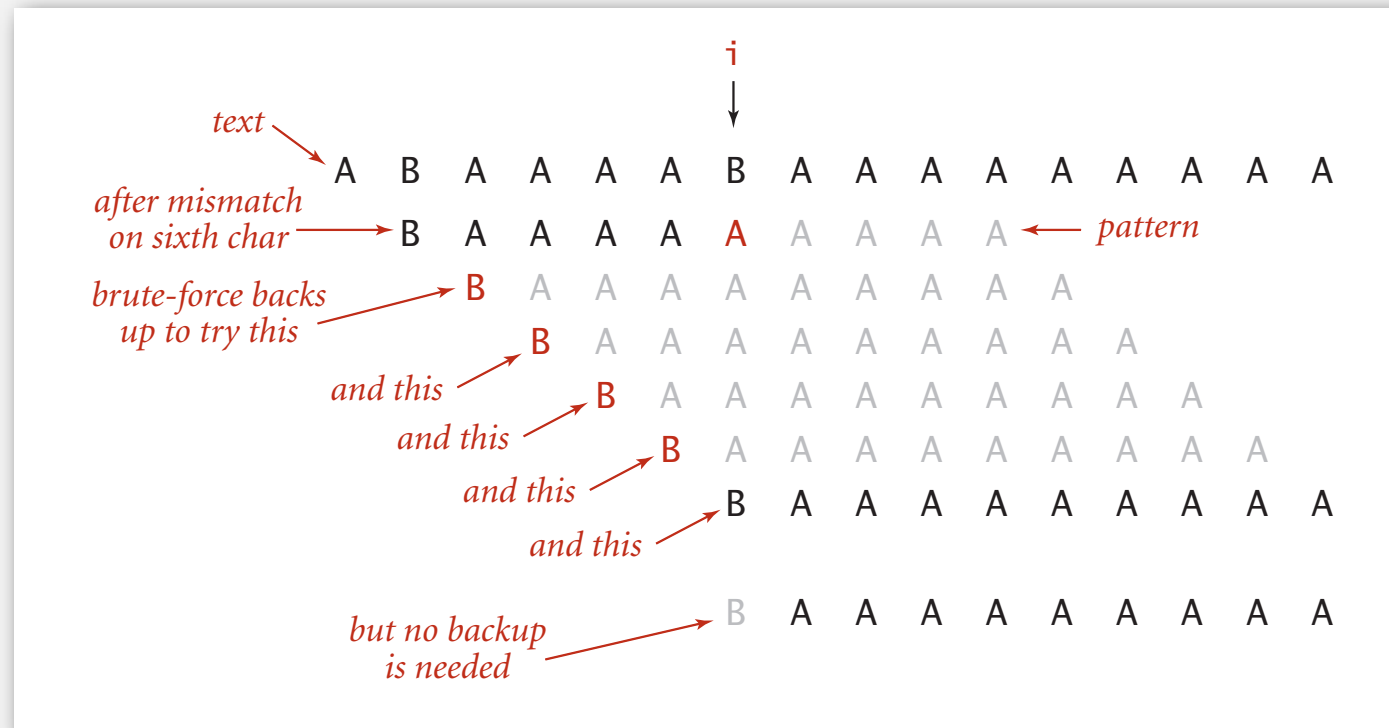
- ▶ brute force
- ▶ **Knuth-Morris-Pratt**
- ▶ Boyer-Moore
- ▶ Rabin-Karp

Knuth-Morris-Pratt substring search

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

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

assuming { A, B } alphabet



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

Deterministic finite state automaton (DFA)

DFA is abstract string-searching machine.

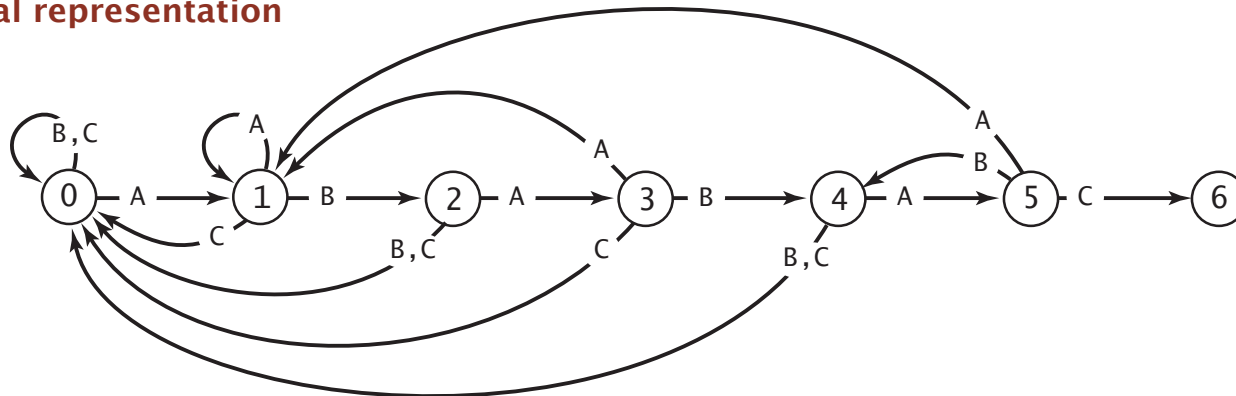
- Finite number of states (including start and halt).
- Exactly one transition for each char in alphabet.
- Accept if sequence of transitions leads to halt state.

internal representation

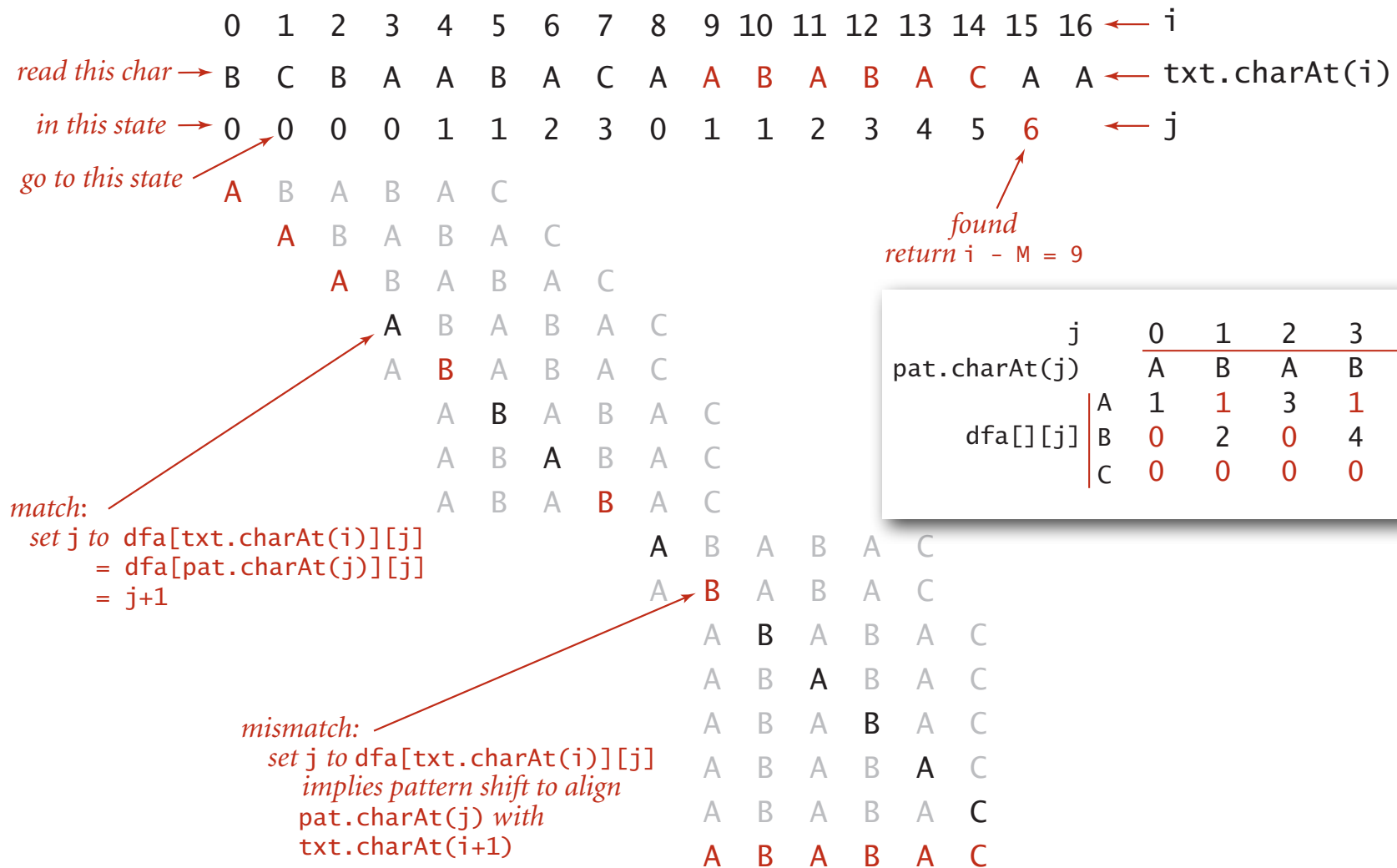
j	0	1	2	3	4	5
pat.charAt(j)	A	B	A	B	A	C
dfa[][j]	A	1	1	3	1	5
	B	0	2	0	4	0
	C	0	0	0	0	6

If in state j reading char c :
if j is 6 halt and accept
else move to state $dfa[c][j]$

graphical representation



KMP substring search: trace



Trace of KMP substring search (DFA simulation) for A B A B A C

Interpretation of Knuth-Morris-Pratt DFA

- Q.** What is interpretation of DFA state after reading in $\text{txt}[i]$?
- A.** State = number of characters in pattern that have been matched.
(length of longest prefix of $\text{pat}[]$ that is a suffix of $\text{txt}[0..i]$)

Ex. DFA is in state 3 after reading in character $\text{txt}[6]$.

$\text{txt} \longrightarrow$

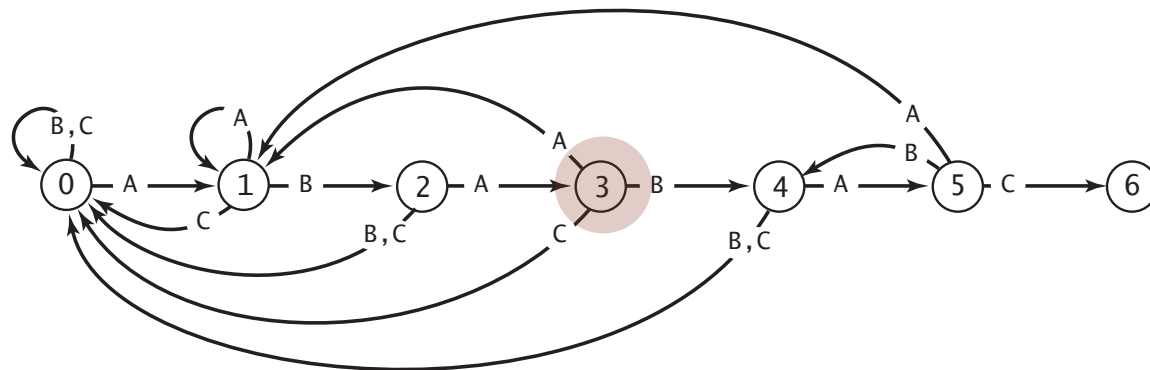
0	1	2	3	4	5	6	7	8
B	C	B	A	A	B	A	C	A

suffix of $\text{txt}[0..6]$

$\text{pat} \longrightarrow$

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

prefix of $\text{pat}[]$



KMP search: Java implementation

Key differences from brute-force implementation.

- Text pointer i never decrements.
- Need to precompute $dfa[][]$ 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;
}
```

← no backup

Running time.

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

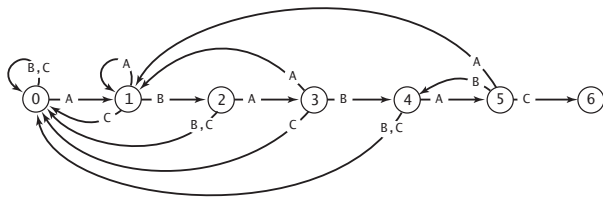
KMP search: Java implementation

Key differences from brute-force implementation.

- Text pointer i never decrements.
- Need to precompute $dfa[][]$ from pattern.
- Could use **input stream**.

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

← no backup



Building DFA from pattern: easy case

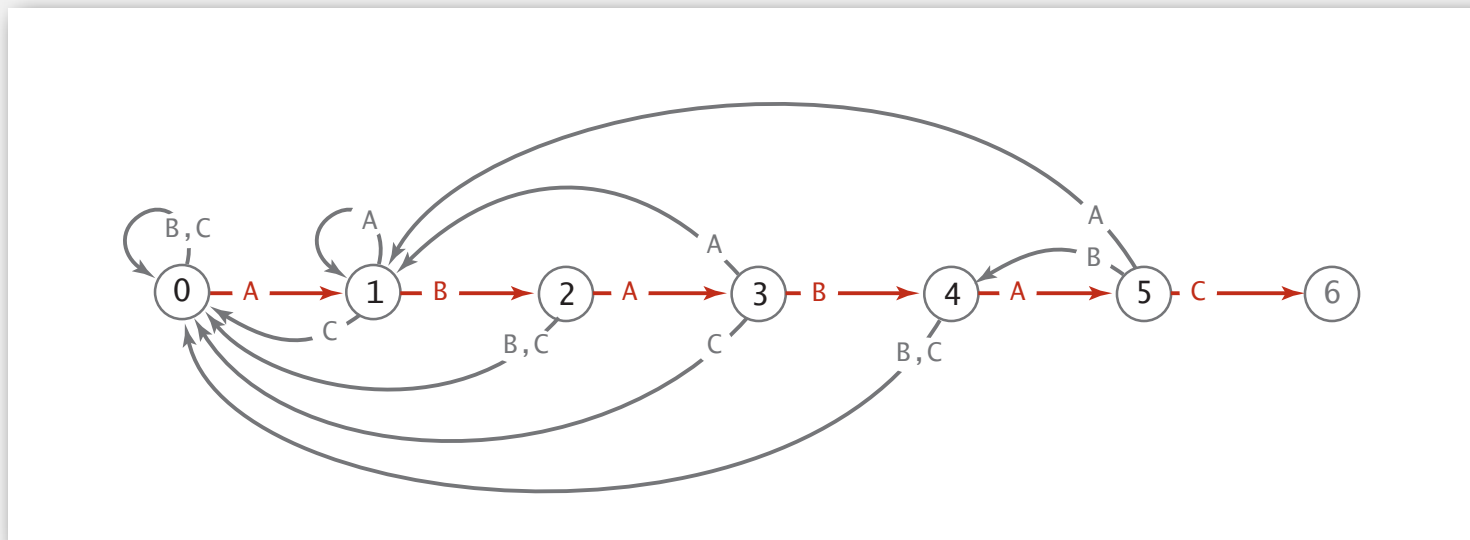
Match transition. If in state j and next char $c == \text{pat.charAt}(j)$,
then go to state $j+1$.

↑
now first $j+1$ characters of
pattern have been matched

↑
first j characters of pattern
have already been matched

↑
next char matches

j	0	1	2	3	4	5
$\text{pat.charAt}(j)$	A	B	A	B	A	C



Knuth-Morris-Pratt construction

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

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

①

②

③

④

⑤

⑥

⑦

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

Knuth-Morris-Pratt construction

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

↑
first j characters of pattern
have already been matched

↑
now first $j+1$ characters of
pattern have been matched

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



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

Building DFA from pattern: tricky case

Mismatch transition.

Suppose DFA is in state j and next char is $c \neq \text{pat.charAt}(j)$

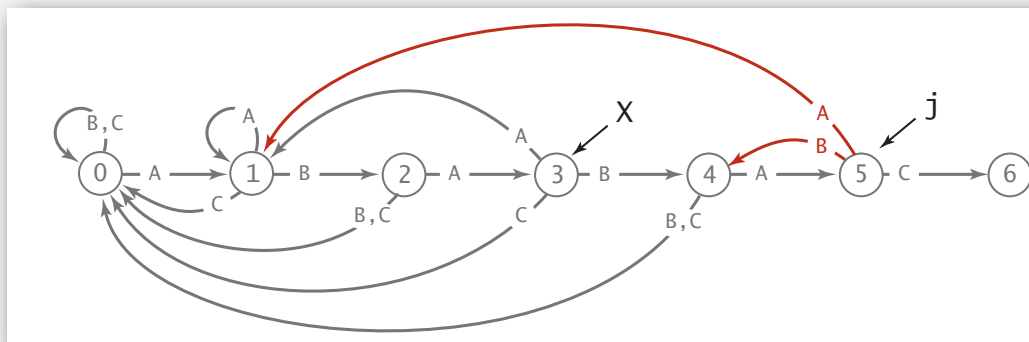
. . . A B A B A C . . .	← match
A B A B A A	← mismatch case 1
A B A B A B	← mismatch case 2

Brute-force solution: Back up $j-1$ chars and restart.

Key idea 1. The last j characters of input are known to be $\text{pat}[1..j-1]$, so use **the DFA itself** to find what state (X) it would be in if restarted after backup

Key idea 2. **Maintain the value of X while constructing DFA.**

Ex.



$\text{pat}[1..j-1] = \text{B A B A}$

$X = 3$

$\text{dfa}['A'][5] = \text{dfa}['A'][X] = 1$

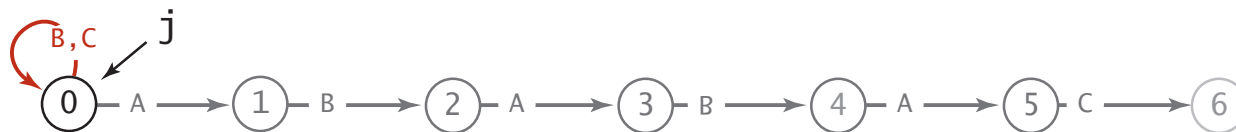
$\text{dfa}['B'][5] = \text{dfa}['B'][X] = 4$

$\text{dfa}['C'][5] = 6$

$\text{new } X = \text{dfa}['C'][X] = 0$

Knuth-Morris-Pratt construction

j	0	1	2	3	4	5
pat.charAt(j)	A	B	A	B	A	C
dfa[][j] A	1		3		5	
B	0	2		4		
C	0					6



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

Knuth-Morris-Pratt construction

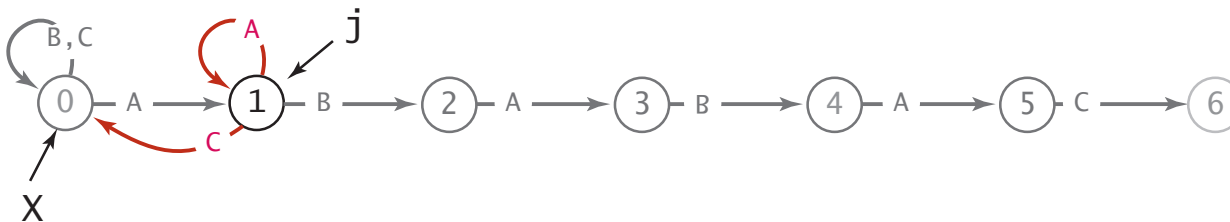
Mismatch transition. For each state j and char $c \neq \text{pat.charAt}(j)$,
 $\text{dfa}[c][j] = \text{dfa}[c][x]$; then update $x = \text{dfa}[\text{pat.charAt}(j)][x]$.

		X = simulation of empty string					
		0	1	2	3	4	5
pat.charAt(j)	j	A	B	A	B	A	C
dfa[][j]	A	1	1	3		5	
	B	0	2		4		
	C	0	0				6

```
j = 1
pat[1..j-1] = ''

X = 0
dfa['A'][1] = dfa['A'][X] = 1
dfa['B'][1] = 2
dfa['C'][1] = dfa['C'][X] = 0

new X = dfa['B'][X] = 0
```



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

Knuth-Morris-Pratt construction

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

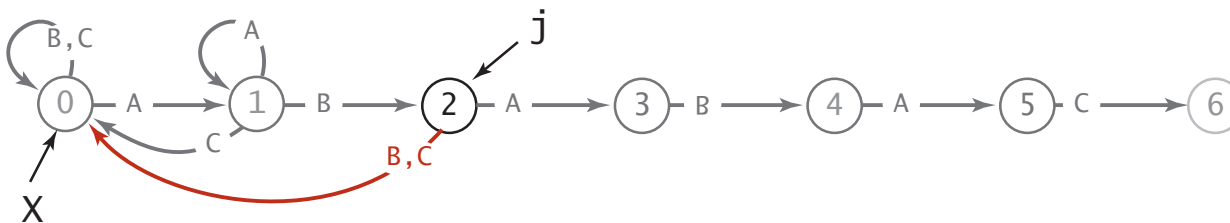
$X = \text{simulation of } B$
↓

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

```
j = 2
pat[1..j-1] = 'B'

x = 0
dfa['A'][2] = 3
dfa['B'][2] = dfa['B'][x] = 0
dfa['C'][2] = dfa['C'][x] = 0

new x = dfa['A'][x] = 1
```



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

Knuth-Morris-Pratt construction

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

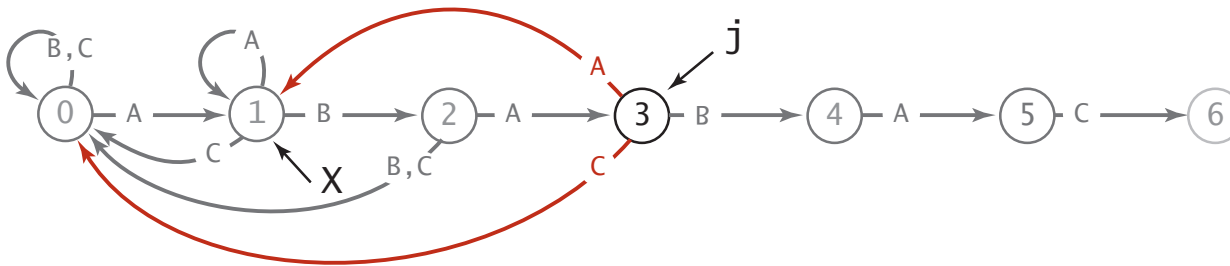
$X = \text{simulation of BA}$
↓

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

```
j = 3
pat[1..j-1] = 'BA'

x = 1
dfa['A'][3] = dfa['A'][x] = 1
dfa['B'][3] = 4
dfa['C'][3] = dfa['C'][x] = 0

new x = dfa['B'][x] = 2
```



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

Knuth-Morris-Pratt construction

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

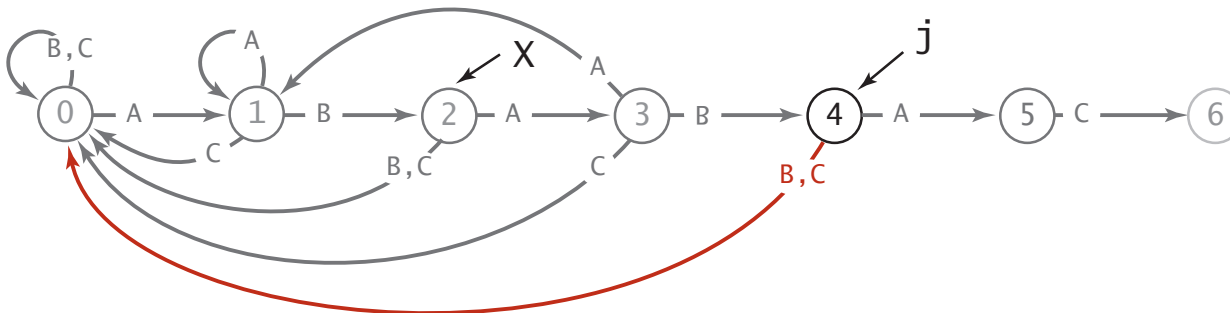
$X = \text{simulation of BAB}$
 \downarrow

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

```
j = 4
pat[1..j-1] = 'BAB'

x = 2
dfa['A'][4] = 5
dfa['B'][4] = dfa['B'][x] = 0
dfa['C'][4] = dfa['C'][x] = 0

new x = dfa['B'][x] = 3
```



Knuth-Morris-Pratt construction

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

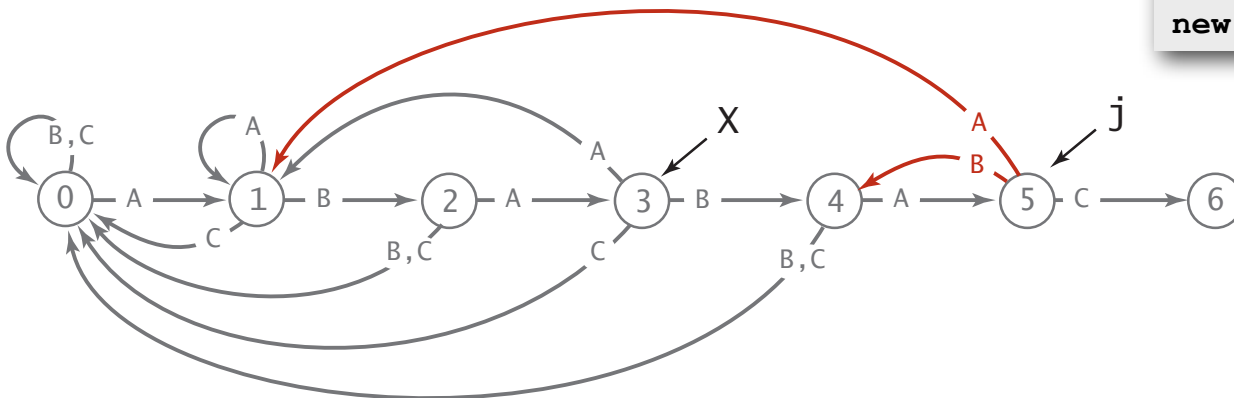
$X = \text{simulation of BABA}$
 \downarrow

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

```
j = 5
pat[1..j-1] = 'BABA'

x = 3
dfa['A'][5] = dfa['A'][x] = 1
dfa['B'][5] = dfa['B'][x] = 4
dfa['C'][5] = 6

new x = dfa['C'][x] = 0
```



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 .

```
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];
    }
}
```

← copy mismatch cases

← set match case

← update restart state

Running time. M character accesses (but space proportional to $R M$).

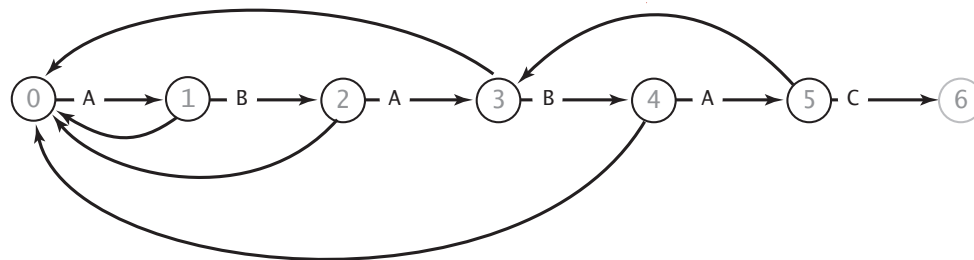
KMP substring search analysis

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

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

Proposition. KMP constructs $\text{dfa}[][]$ in time and space proportional to $R M$.

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.

SIAM J. COMPUT.
Vol. 6, No. 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 this algorithm of practical use, and the procedure can also be extended to deal with some more general pattern-matching problems. A theoretical application of the algorithm shows that the set of concatenations of even palindromes, i.e., the language $\{\alpha\alpha^R\}^*$, can be recognized in linear time. Other algorithms which run even faster on the average are also considered.



Don Knuth



Jim Morris



Vaughan Pratt

Knuth-Morris-Pratt application

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 .

yes

ROTATEDSTRING
STRINGROTATED

yes

ABABABBBABBABA
BABBBABBABAABA

no

ROTATEDSTRING
GNIRTSDETATOR

Problem. Given two strings s and t , design a linear-time algorithm that determines if s is a cyclic rotation of t .

Solution.

- Check that s and t are the same length.
- Search for s in $t + t$ using KMP.

- ▶ brute force
- ▶ Knuth-Morris-Pratt
- ▶ **Boyer-Moore**
- ▶ Rabin-Karp



Robert Boyer



J. Strother Moore

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.

i	j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
	<i>text</i> →	F	I	N	D	I	N	A	H	A	Y	S	T	A	C	K	N	E	E	D	L	E	I	N	A
0	5	N	E	E	D	L	E																		
5	5						N	E	E	D	L	E													
11	4												N	E	E	D	L	E							
15	0																N	E	E	D	L	E			

← *pattern*

← *return i = 15*

Boyer-Moore: mismatched character heuristic

Q. How much to skip?

A. Compute $\text{right}[c]$ = rightmost occurrence of character c in pat .

```
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;
```

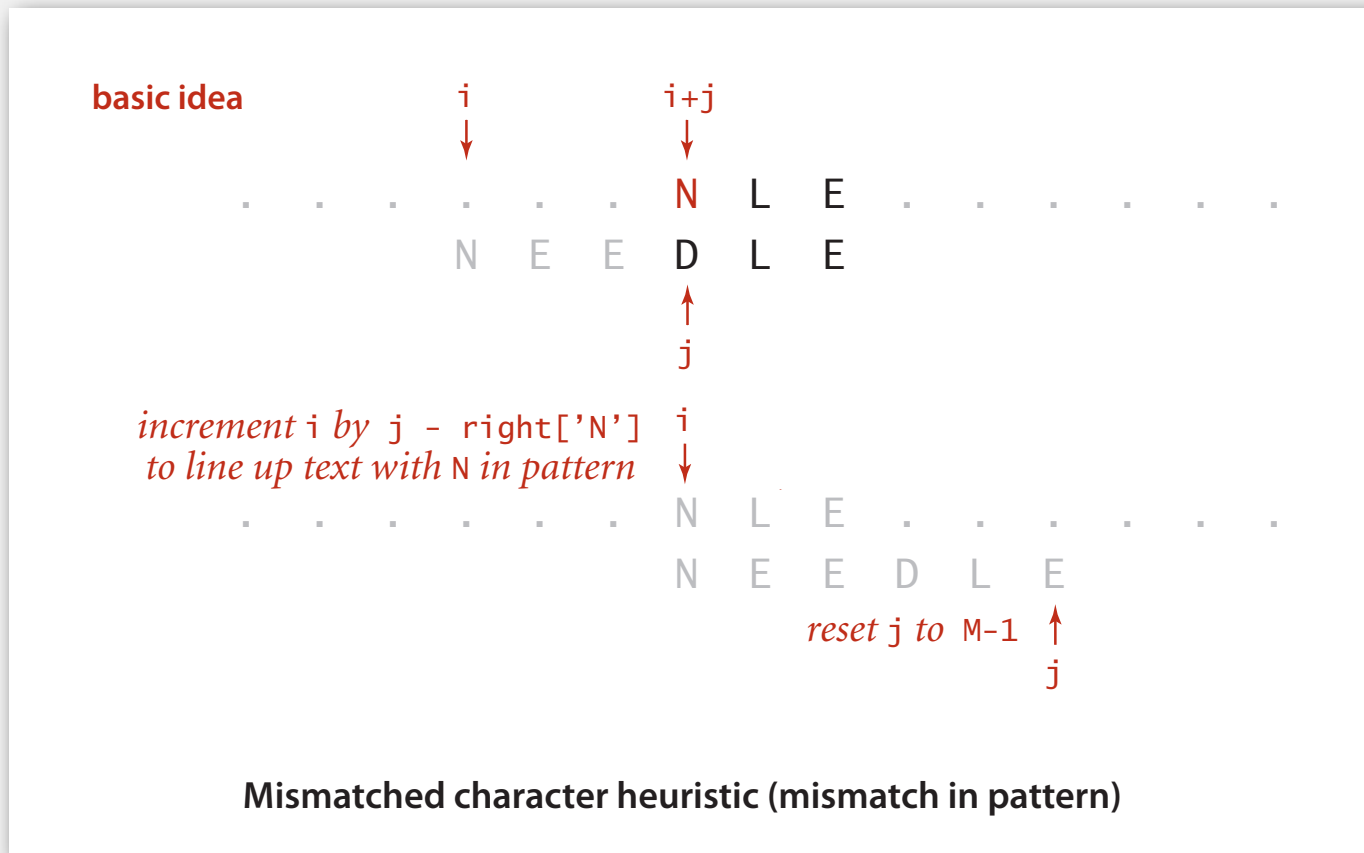
		N	E	E	D	L	E	
<u>c</u>		0	1	2	3	4	5	<u>right[c]</u>
A	-1	-1	-1	-1	-1	-1	-1	-1
B	-1	-1	-1	-1	-1	-1	-1	-1
C	-1	-1	-1	-1	-1	-1	-1	-1
D	-1	-1	-1	-1	3	3	3	3
E	-1	-1	1	2	2	2	5	5
...								-1
L	-1	-1	-1	-1	-1	4	4	4
M	-1	-1	-1	-1	-1	-1	-1	-1
N	-1	0	0	0	0	0	0	0
...								-1

Boyer-Moore skip table computation

Boyer-Moore: mismatched character heuristic

Q. How much to skip?

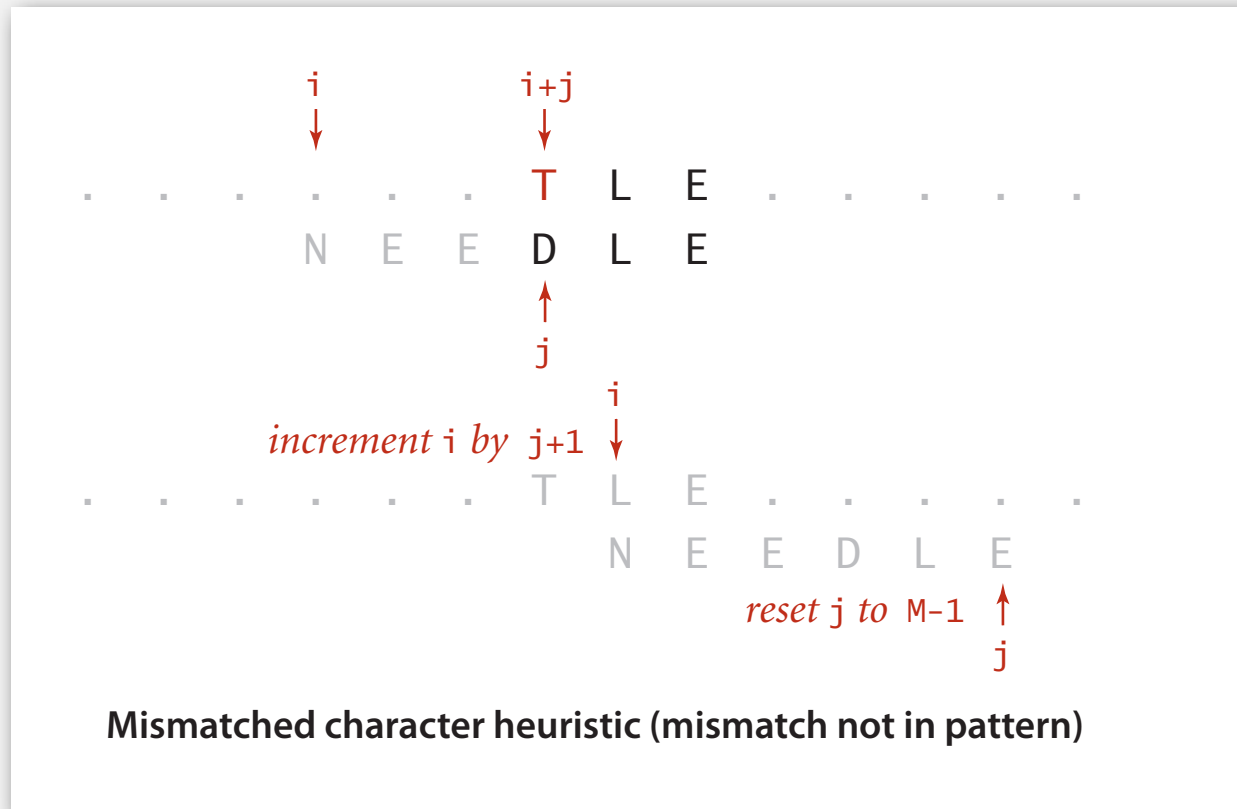
A. Compute $\text{right}[c]$ = rightmost occurrence of character c in pat .



Boyer-Moore: mismatched character heuristic

Q. How much to skip?

A. Compute $\text{right}[c]$ = rightmost occurrence of character c in pat .

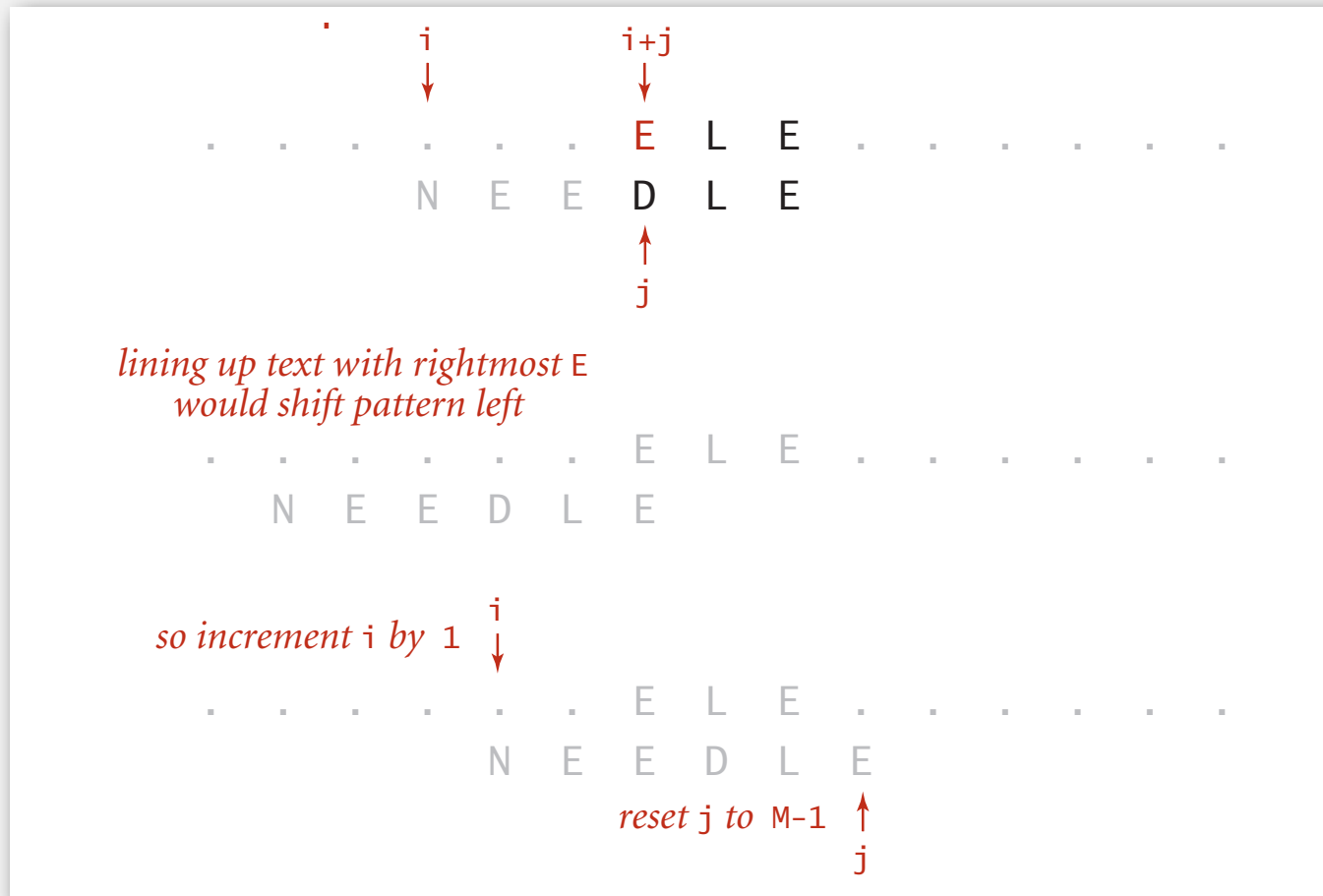


Character not in pattern? Set $\text{right}[c]$ to -1 .

Boyer-Moore: mismatched character heuristic

Q. How much to skip?

A. Compute $\text{right}[c]$ = rightmost occurrence of character c in pat .



Heuristic no help? Increment i and reset j to $M-1$

Boyer-Moore: Java implementation

```
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;
}
```

← compute skip value

← match

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 .  sublinear

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

i skip		0	1	2	3	4	5	6	7	8	9
		txt → B B B B B B B B B B B									
0	0	A	B	B	B	B	← pat				
1	1		A	B	B	B	B				
2	1			A	B	B	B	B			
3	1				A	B	B	B	B		
4	1					A	B	B	B	B	
5	1						A	B	B	B	B

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

- ▶ brute force
- ▶ Knuth-Morris-Pratt
- ▶ Boyer-Moore
- ▶ **Rabin-Karp**



**Michael Rabin, Turing Award '76
and Dick Karp, Turing Award '85**

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.

pat.charAt(i)																		
i	0	1	2	3	4													
	2	6	5	3	5	% 997 = 613												
						txt.charAt(i)												
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	% 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							
6	← return i = 6						2	6	5	3	5	% 997 = 613					match	

match
↙

Efficiently computing the hash function

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

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

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

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

	pat.charAt()				
i	0	1	2	3	4
	2	6	5	3	5
0	2	% 997 = 2			
1	2	6	% 997 = (2*10 + 6) % 997 = 26		
2	2	6	5	% 997 = (26*10 + 5) % 997 = 265	
3	2	6	5	3	% 997 = (265*10 + 3) % 997 = 659
4	2	6	5	3	5 % 997 = (659*10 + 5) % 997 = 613

Computing the hash value for the pattern with Horner's method

```
// 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;
}
```

Efficiently computing the hash function

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

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

$$x_{i+1} = t_{i+1} R^{M-1} + t_{i+2} R^{M-2} + \dots + t_{i+M} R^0$$

Key property. Can update hash function in constant time!

$$x_{i+1} = \underset{\substack{\uparrow \\ \text{shift} \\ \text{left}}}{x_i R} - \underset{\substack{\uparrow \\ \text{subtract} \\ \text{leftmost digit}}}{t_i R^M} + \underset{\substack{\uparrow \\ \text{add new} \\ \text{rightmost digit}}}{t_{i+M}}$$

i	...	2	3	4	5	6	7	...
current value	1	4	1	5	9	2	6	5
new value		4	1	5	9	2	6	5
		4	1	5	9	2		
	-	4	0	0	0	0		
			1	5	9	2		
				*	1	0		
		1	5	9	2	0		
					+	6		
		1	5	9	2	6		

current value
new value
subtract leading digit
multiply by radix
add new trailing digit
new value

text

Rabin-Karp substring search example

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	% 997 = 3														
1	3	1	% 997 = (3*10 + 1) % 997 = 31													
2	3	1	4	% 997 = (31*10 + 4) % 997 = 314												
3	3	1	4	1	% 997 = (314*10 + 1) % 997 = 150											
4	3	1	4	1	5	% 997 = (150*10 + 5) % 997 = 508										
5		1	4	1	5	9	% 997 = ((508 + 3*(997 - 30))*10 + 9) % 997 = 201									
6			4	1	5	9	2	% 997 = ((201 + 1*(997 - 30))*10 + 2) % 997 = 715								
7				1	5	9	2	6	% 997 = ((715 + 4*(997 - 30))*10 + 6) % 997 = 971							
8					5	9	2	6	5	% 997 = ((971 + 1*(997 - 30))*10 + 5) % 997 = 442						
9						9	2	6	5	3	% 997 = ((442 + 5*(997 - 30))*10 + 3) % 997 = 929					
10	← return i-M+1 = 6						2	6	5	3	5	% 997 = ((929 + 9*(997 - 30))*10 + 5) % 997 = 613				

Rabin-Karp: Java implementation

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

    public RabinKarp(String pat) {
        M = pat.length();
        R = 256;
        Q = longRandomPrime();

        RM = 1;
        for (int i = 1; i <= M-1; i++)
            RM = (R * RM) % Q;
        patHash = hash(pat, M);
    }

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

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

← a large prime (but not so large to cause long overflow)

← precompute $R^M - 1 \pmod{Q}$

Rabin-Karp: Java implementation (continued)

Monte Carlo version. Return match if hash match.

```
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;
}
```

check for hash collision
using rolling hash function

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.
- 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 ?



Substring search cost summary

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

algorithm	version	operation count		backup in input?	correct?	extra space
		guarantee	typical			
brute force	—	MN	$1.1 N$	<i>yes</i>	<i>yes</i>	1
Knuth-Morris-Pratt	<i>full DFA</i> (Algorithm 5.6)	$2 N$	$1.1 N$	<i>no</i>	<i>yes</i>	MR
	<i>mismatch</i> <i>transitions only</i>	$3 N$	$1.1 N$	<i>no</i>	<i>yes</i>	M
	<i>full algorithm</i>	$3 N$	N / M	<i>yes</i>	<i>yes</i>	R
Boyer-Moore	<i>mismatched char</i> <i>heuristic only</i> (Algorithm 5.7)	MN	N / M	<i>yes</i>	<i>yes</i>	R
Rabin-Karp [†]	<i>Monte Carlo</i> (Algorithm 5.8)	$7 N$	$7 N$	<i>no</i>	<i>yes</i> [†]	1
	<i>Las Vegas</i>	$7 N$ [†]	$7 N$	<i>yes</i>	<i>yes</i>	1

[†] probabilistic guarantee, with uniform hash function