Pattern Matching

- exact pattern matching
- Knuth-Morris-Pratt
- RE pattern matching
- grep

References:
Algorithms in C (2nd edition), Chapter 19
http://www.cs.princeton.edu/introalgsds/63long
http://www.cs.princeton.edu/introalgsds/72regular

- **exact pattern matching**
- Knuth-Morris-Pratt
- RE pattern matching
- grep

Exact pattern matching

Problem:

Find first match of a pattern of length M in a text stream of length N.

typically N >> M

pattern

n	e	e	d	l	e

M = 6

text

i	n	a	h	a	y	s	t	a	c	k	a	n	e	e	d	l	e	i	n	a

N = 21

Applications.

- parsers.
- spam filters.
- digital libraries.
- screen scrapers.
- word processors.
- web search engines.
- natural language processing.
- computational molecular biology.
- feature detection in digitized images.

. . .

Brute-force exact pattern match

Check for pattern starting at each text position.

h	a	y	n	e	e	d	s	a	n	n	e	e	d	l	e	x
n	e	e	d	l	e											
	n	e	e	d	l	e										
		n	e	e	d	l	e									
			n	**e**	**e**	**d**	**l**	e								
				n	e	e	d	l	e							
					n	e	e	d	l	e						
						n	e	e	d	l	e					
							n	e	e	d	l	e				
								n	e	e	d	l	e			
									n	**e**	e	d	l	e		
										n	**e**	**e**	**d**	**l**	**e**	

```
public static int search(String pattern, String text)
{
   int M = pattern.length();
   int N = text.length();

   for (int i = 0; i < N - M; i++)
   {
      int j;
      for (j = 0; j < M; j++)
         if (text.charAt(i+j) != pattern.charAt(j))
            break;
      if (j == M) return i;  ← pattern start index in text
   }
   return -1;  ← not found
}
```

Brute-force exact pattern match: worst case

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

a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	b	text length N
a	a	a	a	a	b												
	a	a	a	a	a	b											
		a	a	a	a	a	b										
			a	a	a	a	a	b									
				a	a	a	a	a	b								
					a	a	a	a	a	b							
						a	a	a	a	a	b						
							a	a	a	a	a	b					
								a	a	a	a	a	b				
									a	a	a	a	a	b			
										a	a	a	a	a	b		
											a	a	a	a	a	b	pattern length M

MN char compares

but this situation is rare in typical applications

Hence, the `indexOf()` method in Java's `String` class uses brute-force

Exact pattern matching in Java

Exact pattern matching is implemented in Java's **`String`** class

`s.indexOf(t, i)`: index of first occurrence of pattern **`t`** in string `s`, starting at offset **`i`**.

Ex: Screen scraping. Exact match to extract info from website

```
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 input = in.readAll();
      int start    = input.indexOf("Last Trade:", 0);
      int from     = input.indexOf("<b>",  start);
      int to       = input.indexOf("</b>", from);
      String price = input.substring(from + 3, to);
      System.out.println(price);
   }
}
```

```
% java StockQuote goog
688.04

% java StockQuote msft
33.75
```

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

```
...
<tr>
<td class= "yfnc_tablehead1"
width= "48%">
Last Trade:
</td>
<td class= "yfnc_tabledata1">
<big><b>688.04</b></big>
</td></tr>
<td class= "yfnc_tablehead1"
width= "48%">
Trade Time:
</td>
<td class= "yfnc_tabledata1">
```

Algorithmic challenges in pattern matching

Brute-force is not good enough for all applications

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.
```

- exact pattern matching
- **Knuth-Morris-Pratt**
- RE pattern matching
- grep

Knuth-Morris-Pratt (KMP) exact pattern-matching algorithm

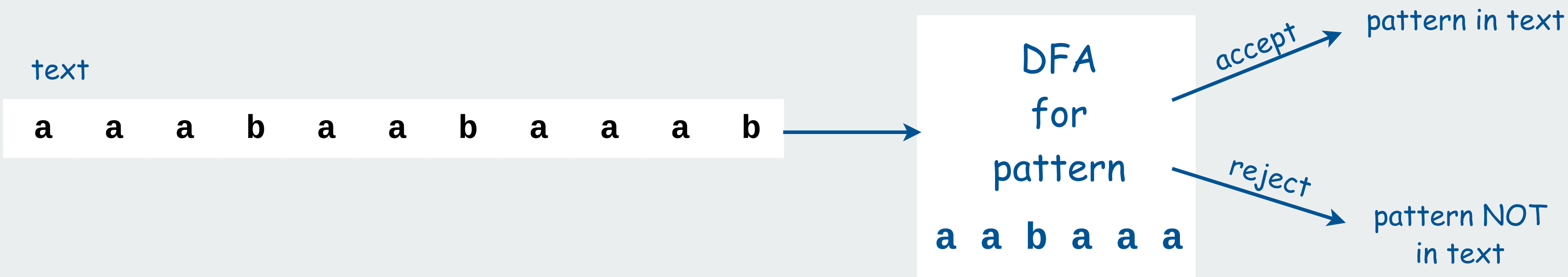
Classic algorithm that meets both challenges
- linear-time guarantee
- no backup in text stream


Don Knuth


Jim Morris


Vaughan Pratt

Basic plan (for binary alphabet)
- build DFA from pattern
- simulate DFA with text as input



No backup in a DFA
Linear-time because each step is just a state change

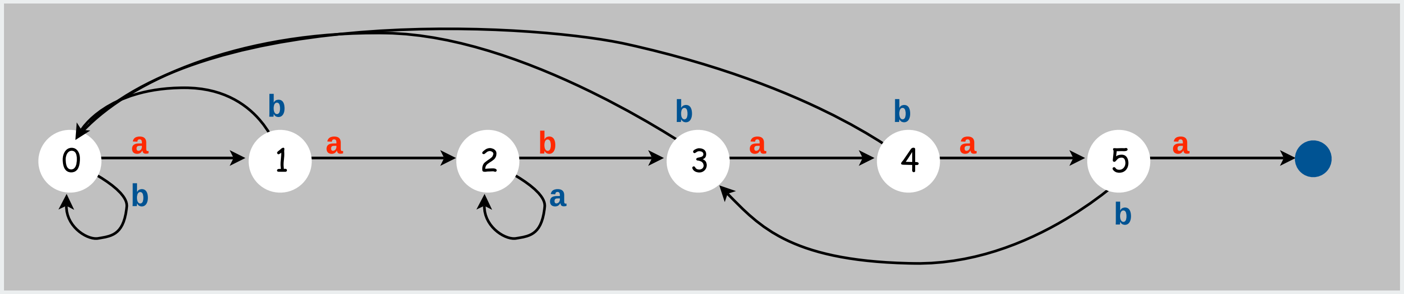
Knuth-Morris-Pratt DFA example

One state for each pattern character

- Match input character: move from i to i+1
- Mismatch: move to previous state

DFA
for
pattern

a a b a a a



How to construct? Stay tuned

Knuth-Morris-Pratt DFA simulation

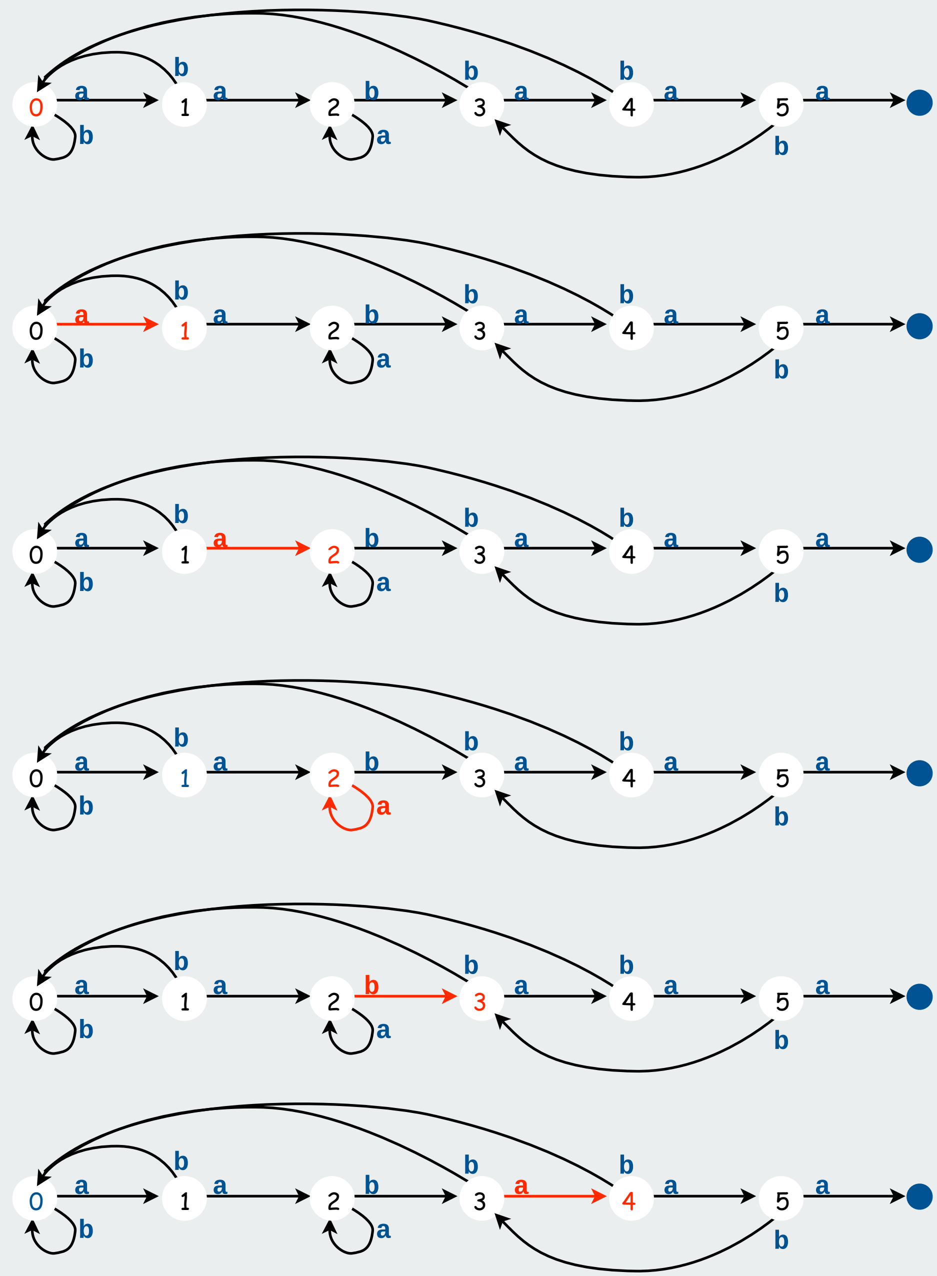
0 `a a a b a a b a a a b`

1 `a a a b a a b a a a b`

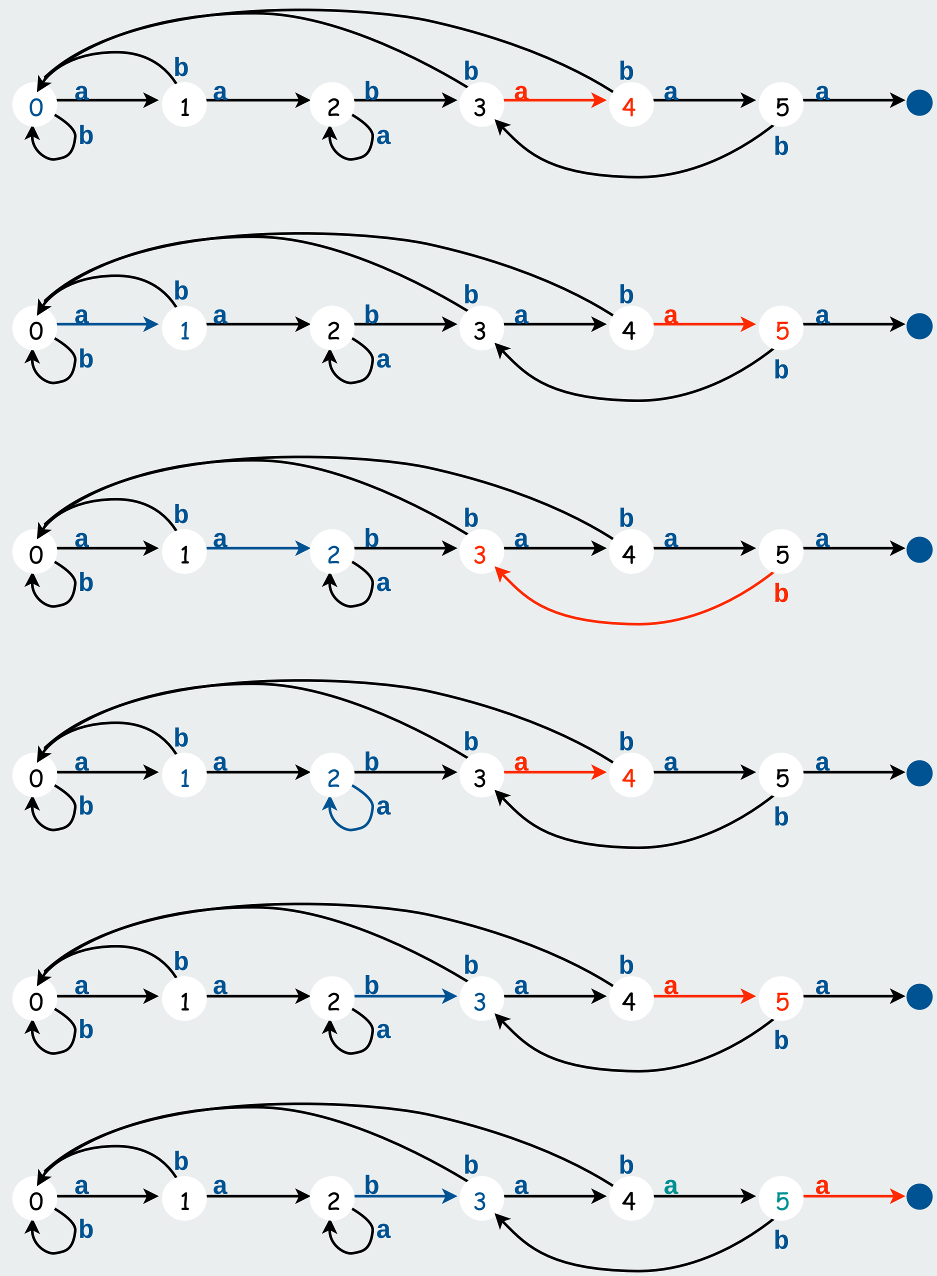
2 `a a a b a a b a a a b`

2 `a a a b a a b a a a b`

3 `a a a b a a b a a a b`



Knuth-Morris-Pratt DFA simulation



4 a a a b a a b a a a b

5 a a a b a a b a a a b

3 a a a b a a b a a a b

4 a a a b a a b a a a b

5 a a a b a a b a a a b

accept!

Knuth-Morris-Pratt DFA simulation

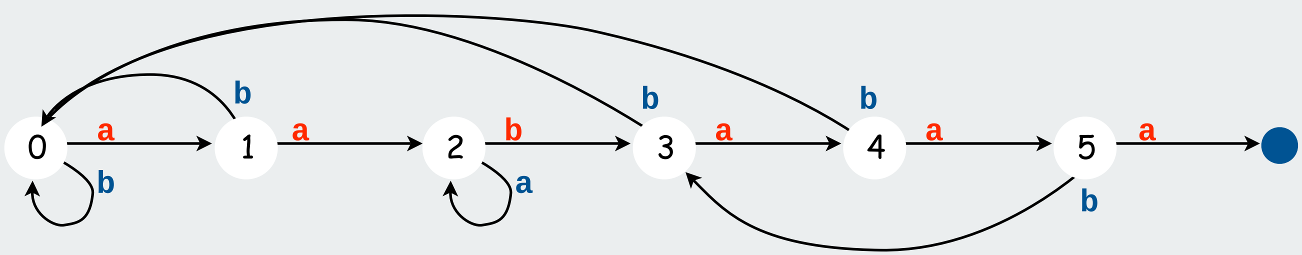
When in state i:

- have found match in i previous input chars
- that is the longest such match

Ex. End in state 4 iff text ends in **aaba**.

Ex. End in state 2 iff text ends in **aa** (but not **aabaa** or **aabaaa**).

0	a	a	a	b	a	a	b	a	a	a	b
1	a	a	a	b	a	a	b	a	a	a	b
2	a	a	a	b	a	a	b	a	a	a	b
2	a	a	a	b	a	a	b	a	a	a	b
3	a	a	a	b	a	a	b	a	a	a	b
4	a	a	a	b	a	a	b	a	a	a	b
5	a	a	a	b	a	a	b	a	a	a	b
3	a	a	a	b	a	a	b	a	a	a	b
4	a	a	a	b	a	a	b	a	a	a	b
5	a	a	a	b	a	a	b	a	a	a	b
	a	a	a	b	a	a	b	a	a	a	b



KMP implementation

DFA representation: a single state-indexed array `next[]`

- Upon character match in state `j`, go forward to state `j+1`.
- Upon character mismatch in state `j`, go back to state `next[j]`.

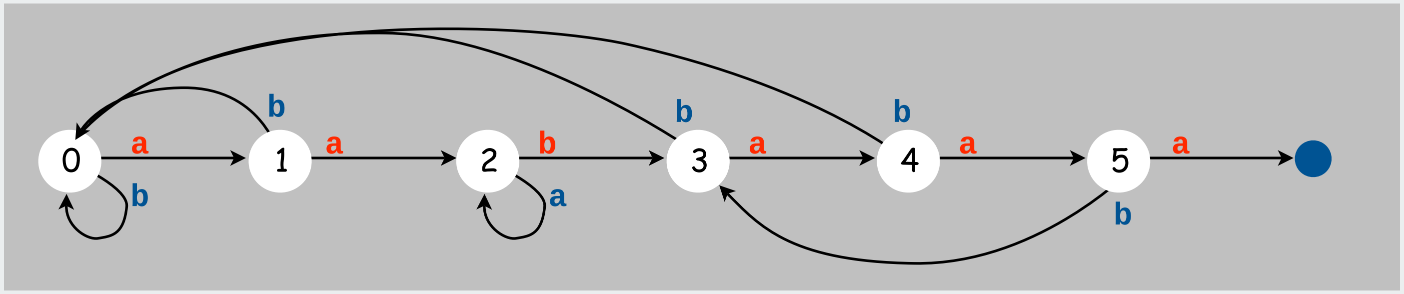
DFA for pattern

a a b a a a

	0	1	2	3	4	5
a	1	2	2	4	5	6
b	0	0	3	0	0	3

next	0	0	2	0	0	3

← only need to store mismatches



KMP implementation

Two key differences from brute-force implementation:

- Text pointer `i` never decrements
- Need to precompute `next[]` table (DFA) from pattern.

```
int j = 0;
for (int i = 0; i < N; i++)
{
   if (t.charAt(i) == p.charAt(j)) j++;   // match
   else j = next[j];                      // mismatch
   if (j == M) return i - M + 1;          // found
}
return -1;                                // not found
```

Simulation of KMP DFA

Knuth-Morris-Pratt: Iterative DFA construction

DFA for first i states contains the information needed to build state i+1

Ex: given DFA for pattern **aabaaa**.
how to compute DFA for pattern **aabaaab** ?

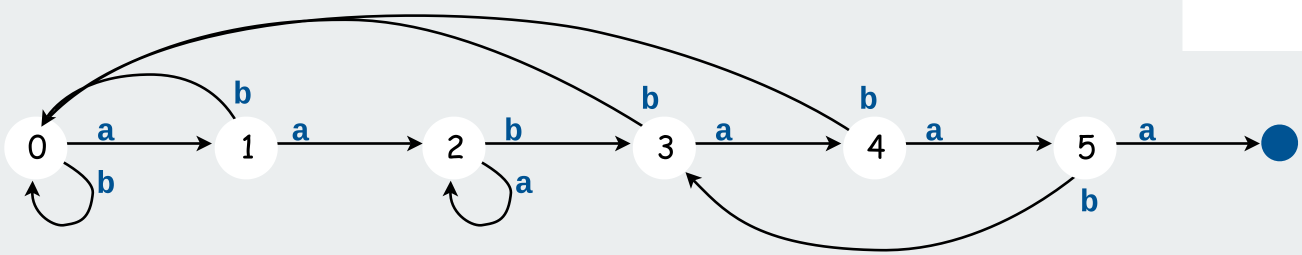
Key idea

- on mismatch at 7th char, need to simulate 6-char backup
- previous 6 chars are known (**abaaaa** in example)
- 6-state DFA (known) determines next state!

Keep track of DFA state for start at 2nd char of pattern

- compare char at that position with next pattern char
- match/mismatch provides all needed info

0	a	b	a	a	a	a
1	a	b	a	a	a	a
0	a	b	a	a	a	a
1	a	b	a	a	a	a
2	a	b	a	a	a	a
2	a	b	a	a	a	a
2	a	b	a	a	a	a



KMP iterative DFA construction: two cases

Let X be the next state in the simulation and j the next state to build.

If p[X] and p[j] match, copy and increment

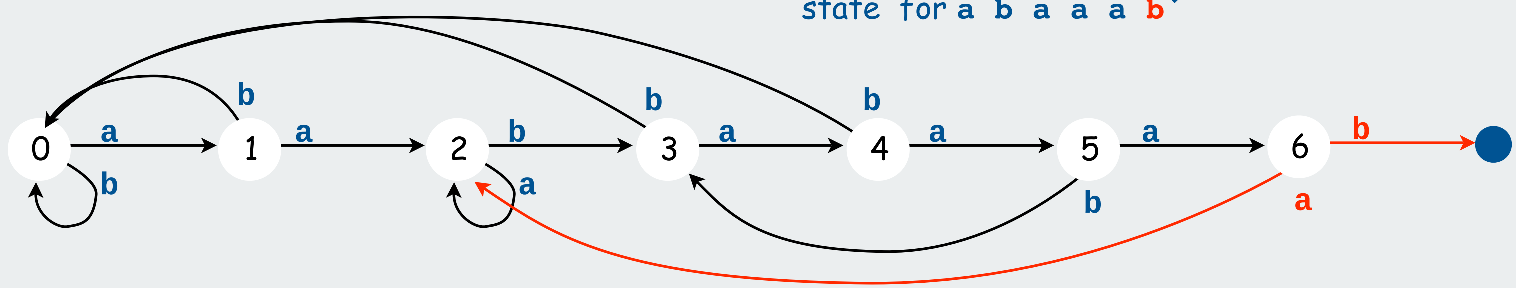
```
next[j] = next[X];
X = X+1
```

	0	1	2	3	4	5	6
p[]	a	a	b	a	a	a	b
next[]	0	0	2	0	0	3	2

x X j

state for a b a a a b

DFA for a a b a a a b



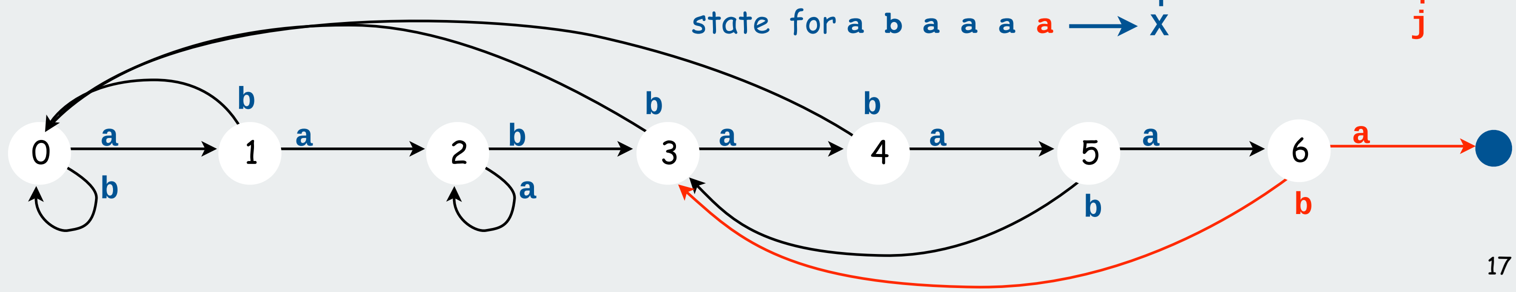
If p[X] and p[j] mismatch, do the opposite

```
next[j] = X+1;
X = next[X];
```

	0	1	2	3	4	5	6
p[]	a	a	b	a	a	a	a
next[]	0	0	2	0	0	3	3

state for a b a a a a → X j

DFA for a a b a a a a



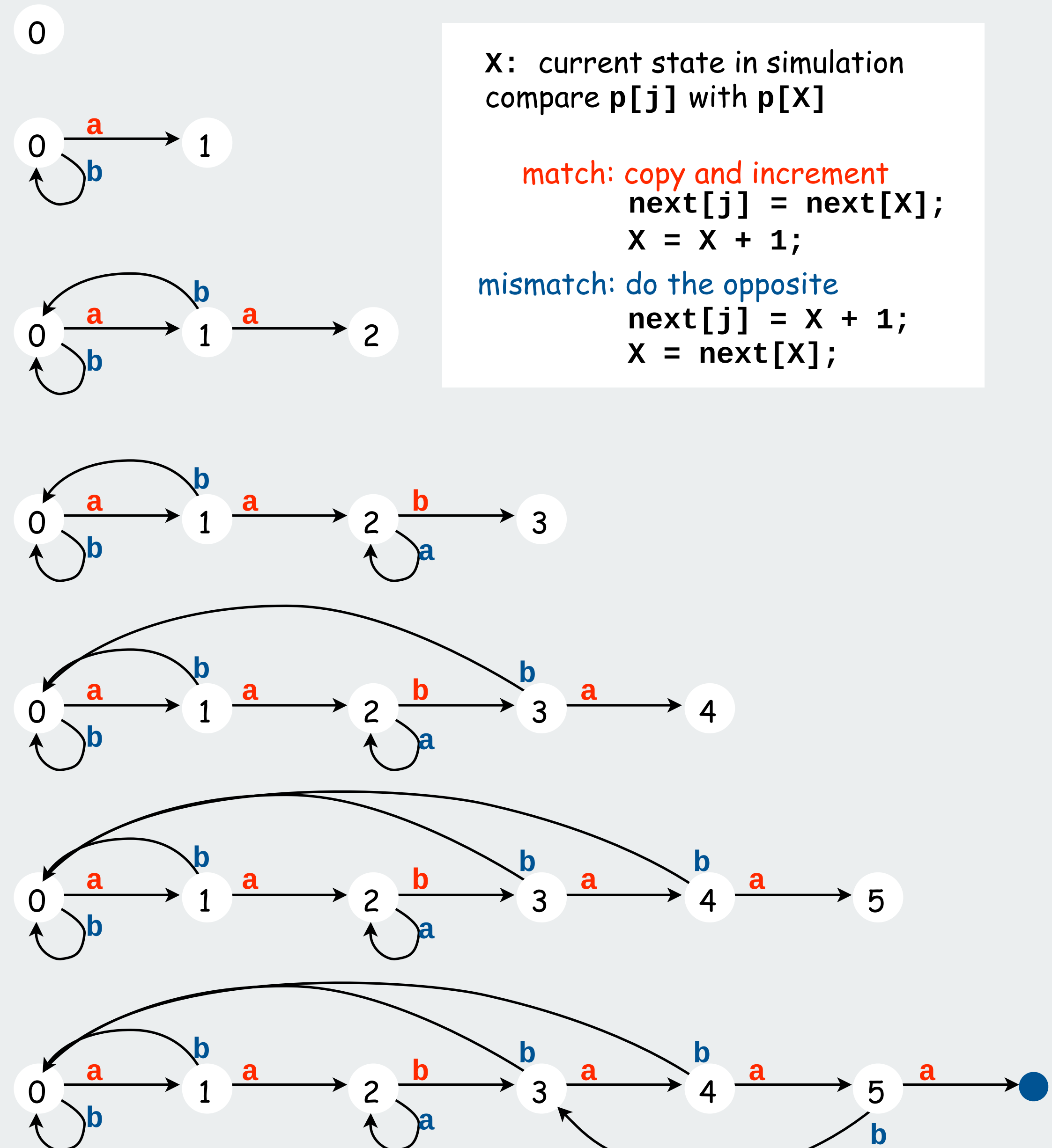
Knuth-Morris-Pratt DFA construction

DFA

0				
a				
0				

0 1		match		
a a				
0 0				
X j				

DFA		p[1..j-1]	X
0 a 0			0
0 1 a a 0 0 X j	match	a	1
0 1 2 a a b 0 0 2	mismatch	a b	0
0 1 2 3 a a b a 0 0 2 0	match	a b a	1
0 1 2 3 4 a a b a a 0 0 2 0 0	match	a b a a	2
0 1 2 3 4 5 a a b a a a 0 0 2 0 0 3	mismatch		



X: current state in simulation
compare **p[j]** with **p[X]**

match: copy and increment
```
next[j] = next[X];
X = X + 1;
```
mismatch: do the opposite
```
next[j] = X + 1;
X = next[X];
```

Knuth-Morris-Pratt DFA construction examples

ex: a a b a a a b

```
        0
        a
        0

        0 1
        a a
next[]  0 0             match
        ↑ ↑
        X j

        0 1 2
        a a b           mismatch
        0 0 2
          ↑ ↑

        0 1 2 3
        a a b a         match
        0 0 2 0
        ↑     ↑

        0 1 2 3 4
        a a b a a       match
        0 0 2 0 0
          ↑     ↑

        0 1 2 3 4 5
        a a b a a a     mismatch
        0 0 2 0 0 3
            ↑     ↑

        0 1 2 3 4 5 6
        a a b a a a b   match
        0 0 2 0 0 3 2
            ↑       ↑
```

ex: a b b a b b b

```
0
a
0

0 1
a b             mismatch
0 1
↑ ↑
X j

0 1 2
a b b           mismatch
0 1 1
↑   ↑

0 1 2 3
a b b a         match
0 1 1 0
↑     ↑

0 1 2 3 4
a b b a b       match
0 1 1 0 1
  ↑     ↑

0 1 2 3 4 5
a b b a b b     match
0 1 1 0 1 1
    ↑     ↑

0 1 2 3 4 5 6
a b b a b b b   mismatch
0 1 1 0 1 1 4
      ↑     ↑
```

X: current state in simulation
compare **p[j]** with **p[X]**

match: copy and increment

```
next[j] = next[X];
X = X + 1;
```

mismatch: do the opposite

```
next[j] = X + 1;
X = next[X];
```

DFA construction for KMP: Java implementation

Takes time and space proportional to pattern length.

```
int X = 0;
int[] next = new int[M];
for (int j = 1; j < M; j++)
{
   if (p.charAt(X) == p.charAt(j))
   {  // match
      next[j] = next[X];
      X = X + 1;
   }
   else
   {  // mismatch
      next[j] = X + 1;
      X = next[X];
   }
}
```

DFA Construction for KMP (assumes binary alphabet)

Optimized KMP implementation

Ultimate search program for any given pattern:

- one statement comparing each pattern character to next
- match: proceed to next statement
- mismatch: go back as dictated by DFA
- translates to machine language (three instructions per pattern char)

```
int kmpsearch(char t[])
{
   int i = 0;
   s0: if (t[i++] != 'a') goto s0;
   s1: if (t[i++] != 'a') goto s0;
   s2: if (t[i++] != 'b') goto s2;
   s3: if (t[i++] != 'a') goto s0;
   s4: if (t[i++] != 'a') goto s0;
   s5: if (t[i++] != 'a') goto s3;
   s6: if (t[i++] != 'b') goto s2;
   s7: if (t[i++] != 'b') goto s4;
   return i - 8;
}
```

pattern[]   next[]

assumes pattern is in text (o/w use sentinel)

Lesson: Your computer is a DFA!

KMP summary

General alphabet

- more difficult
- easy with `next[][]` indexed by mismatch position, character
- KMP paper has ingenious solution that is not difficult to implement
  [build NFA, then prove that it finishes in 2N steps]

Bottom line: linear-time pattern matching is possible (and practical)

Short history:

- inspired by esoteric theorem of Cook
  [linear time 2-way pushdown automata simulation is possible]
- 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

Exact pattern matching: other approaches

Rabin-Karp: make a digital signature of the pattern

- hashing without the table
- linear-time probabilistic guarantee
- plus: extends to 2D patterns
- minus: arithmetic ops much slower than char comparisons

Boyer-Moore: scan from right to left in pattern

- main idea: can skip M text chars when finding one not in the pattern
- needs additional KMP-like heuristic
- plus: possibility of sublinear-time performance (~ N/M)
- used in Unix, emacs

pattern	s	y	z	y	g	y															
text	a	a	a	b	b	a	a	b	a	b	a	a	a	b	b	a	a	a	b	a	a
	s	y	z	y	g	y															
							s	y	z	y	g	y									
													s	y	z	y	g	y			

Exact pattern match cost summary

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

algorithm	typical	worst-case
brute-force	1.1 N char compares †	M N char compares
Karp-Rabin	3N arithmetic ops	3N arithmetic ops ‡
KMP	1.1 N char compares †	2N char compares
Boyer-Moore	~ N/M char compares †	3N char compares

† assumes appropriate model
‡ randomized

- exact pattern matching
- Knuth-Morris-Pratt
- **RE pattern matching**
- grep

Exact pattern matching:

Search for occurrences of a single pattern in a text file.

Regular expression (RE) pattern matching:

Search for occurrences of one of multiple patterns in a text file.

Ex. (genomics)

- Fragile X syndrome is a common cause of mental retardation.
- human genome contains triplet repeats of `cgg` or `agg` bracketed by `gcg` at the beginning and `ctg` at the end
- number of repeats is variable, and correlated with syndrome
- use regular expression to specify pattern: `gcg(cgg|agg)*ctg`
- do RE pattern match on person's genome to detect Fragile X

pattern (RE) `gcg(cgg|agg)*ctg`

text `gcggcgtgtgtgcgagagagtgggtttaaagctggcgcggaggcggctggcgcggaggctg`

RE pattern matching: applications

Test if a string matches some pattern.

- Process natural language.
- Scan for virus signatures.
- Search for information using Google.
- Access information in digital libraries.
- Retrieve information from Lexis/Nexis.
- Search-and-replace in a word processors.
- Filter text (spam, NetNanny, Carnivore, malware).
- Validate data-entry fields (dates, email, URL, credit card).
- Search for markers in human genome using PROSITE patterns.

Parse text files.

- Compile a Java program.
- Crawl and index the Web.
- Read in data stored in ad hoc input file format.
- Automatically create Java documentation from Javadoc comments.

Regular expression examples

A regular expression is a notation to specify a set of strings.

operation	example RE	in set	not in set
concatenation	`aabaab`	`aabaab`	every other string
wildcard	`.u.u.u.`	`cumulus` `jugulum`	`succubus` `tumultuous`
union	`aa | baab`	`aa` `baab`	every other string
closure	`ab*a`	`aa` `abbba`	`ab` `ababa`
parentheses	`a(a|b)aab`	`aaaab` `abaab`	every other string
	`(ab)*a`	`a` `ababababa`	`aa` `abbba`

Regular expression examples (continued)

Notation is surprisingly expressive

regular expression	in set	not in set
.*spb.* contains the trigraph **spb**	**raspberry** **crispbread**	**subspace** **subspecies**
a* | (a*ba*ba*ba*)* number of **b**'s is a multiple of 3	**bbb** **aaa** **bbbaababbaa**	**b** **bb** **baabbbaa**
.*0.... fifth to last digit is **0**	**1000234** **98701234**	**111111111** **403982772**
gcg(cgg|agg)*ctg fragile X syndrome indicator	**gcgctg** **gcgcggctg** **gcgcggaggctg**	**gcgcgg** **cggcggcggctg** **gcgcaggctg**

and plays a well-understood role in the theory of computation

Generalized regular expressions

Additional operations are often added

- Ex: `[a-e]+` is shorthand for `(a|b|c|d|e)(a|b|c|d|e)*`
- for convenience only
- need to be alert for non-regular additions (Ex: Java /)

operation	example	in set	not in set
one or more	`a(bc)+de`	`abcde` `abcbcde`	`ade` `bcde`
character classes	`[A-Za-z][a-z]*`	`word` `Capitalized`	`camelCase` `4illegal`
exactly k	`[0-9]{5}-[0-9]{4}`	`08540-1321` `19072-5541`	`111111111` `166-54-111`
negations	`[^aeiou]{6}`	`rhythm`	`decade`

Regular expressions in Java

RE pattern matching is implemented in Java's **`String`** class

- basic: **`match()`** method
- various other methods also available (stay tuned)

Ex: Validity checking. Is **`input`** in the set described by the **`re`**?

```
public class Validate
{
   public static void main(String[] args)
   {
      String re    = args[0];
      String input = args[1];
      System.out.println(input.matches(re));
   }
}
```

```
% java Validate "..oo..oo." bloodroot
true

% java Validate "[$_A-Za-z][$_A-Za-z0-9]*" ident123
true

% java Validate "[a-z]+@([a-z]+\.)+(edu|com)" rs@cs.princeton.edu
true

% java Validate "[0-9]{3}-[0-9]{2}-[0-9]{4}" 166-11-4433
true
```

need help solving crosswords?

legal Java identifier

valid email address (simplified)

Social Security number

Regular expressions in other languages

Broadly applicable programmer's tool.

- originated in UNIX in the 1970s
- many languages support extended regular expressions
- built into grep, awk, emacs, Perl, PHP, Python, JavaScript

```
grep NEWLINE */*.java
```

print all lines containing **`NEWLINE`** which occurs in any file with a **`.java`** extension

```
egrep '^[qwertyuiop]*[zxcvbnm]*$' dict.txt | egrep '...........'
```

PERL. Practical Extraction and Report Language.

```
perl -p -i -e 's|from|to|g' input.txt
```

replace all occurrences of **`from`** with **`to`** in the file **`input.txt`**

```
perl -n -e 'print if /^[A-Za-z][a-z]*$/' dict.txt
```

do for each line

Regular expression caveat

Writing a RE is like writing a program.

- need to understand programming model
- can be easier to write than read
- can be difficult to debug

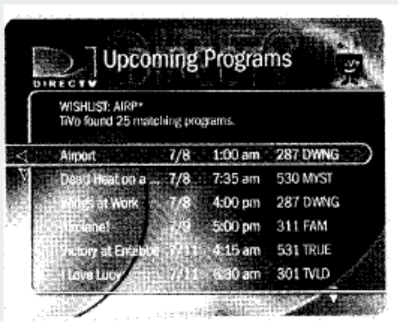
"Sometimes you have a programming problem and it seems like the best solution is to use regular expressions; now you have two problems."

Can the average web surfer learn to use REs?

Google. Supports * for full word wildcard and | for union.

Can the average TV viewer learn to use REs?

TiVo. WishList has very limited pattern matching.



Using * in WishList Searches. To search for similar words in Keyword and Title WishList searches, use the asterisk (*) as a special symbol that replaces the endings of words. For example, the keyword *AIRP** would find shows containing "airport," "airplane," "airplanes," as well as the movie "Airplane!" To enter an asterisk, press the SLOW ( ) button as you are spelling out your keyword or title.

The asterisk can be helpful when you're looking for a range of similar words, as in the example above, or if you're just not sure how something is spelled. Pop quiz: is it "irresistible" or "irresistable?" Use the keyword *IRRESIST** and don't worry about it! Two things to note about using the asterisk:

- It can only be used at a word's end; it cannot be used to omit letters at the beginning or in the middle of a word. (For example, *AIR*NE* or **PLANE* would not work.)

Reference: page 76, Hughes DirectTV TiVo manual

Can the average programmer learn to use REs?

Perl RE for Valid RFC822 Email Addresses

Reference: http://www.ex-parrot.com/~pdw/Mail-RFC822-Address.html

```
(?:(?:\r\n)?[ \t])*(?:(?:(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t]
)+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:
\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(
?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[
\t]))*"(?:(?:\r\n)?[ \t])*))*@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\0
31]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\
](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+
(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:
(?:\r\n)?[ \t])*))*|(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z
|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)
?[ \t])*)*\<(?:(?:\r\n)?[ \t])*(?:@(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\
r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[
 \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)
?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t]
)*))*(?:,@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
 \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*
)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t]
)+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*)
*:(?:(?:\r\n)?[ \t])*)?(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+
|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r
\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:
\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t
]))*"(?:(?:\r\n)?[ \t])*))*@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031
]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](
?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?
:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?
:\r\n)?[ \t])*))*\>(?:(?:\r\n)?[ \t])*)|(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?
:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?
[ \t]))*"(?:(?:\r\n)?[ \t])*)*:(?:(?:\r\n)?[ \t])*(?:(?:(?:[^()<>@,;:\\".\[\]
\000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|
\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>
@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"
(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*))*@(?:(?:\r\n)?[ \t]
)*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\
".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?
:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[
\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*|(?:[^()<>@,;:\\".\[\] \000-
\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(
?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)*\<(?:(?:\r\n)?[ \t])*(?:@(?:[^()<>@,;
:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([
^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\"
.\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\
]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*(?:,@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\
[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\
r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\]
```

"Implementing validation with regular expressions somewhat pushes the limits of what it is sensible to do with regular expressions, although Perl copes well."

37 more lines

- exact pattern matching
- Knuth-Morris-Pratt
- RE pattern matching
- **grep**

GREP implementation: basic plan

Overview is the same as for KMP !

- linear-time guarantee
- no backup in text stream

Ken Thompson

Basic plan for GREP

- build DFA from RE
- simulate DFA with text as input

text

actgtgcaggaggcggcgcggcggaggaggctggcga → DFA for pattern **gcg(cgg|agg)*ctg**

accept → pattern in text

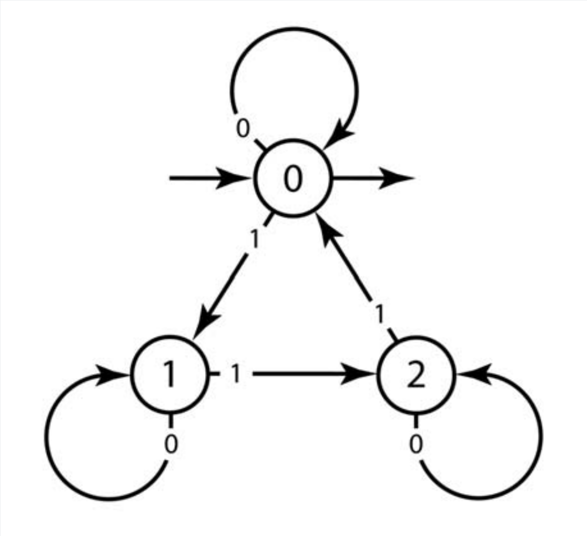
reject → pattern NOT in text

No backup in a DFA

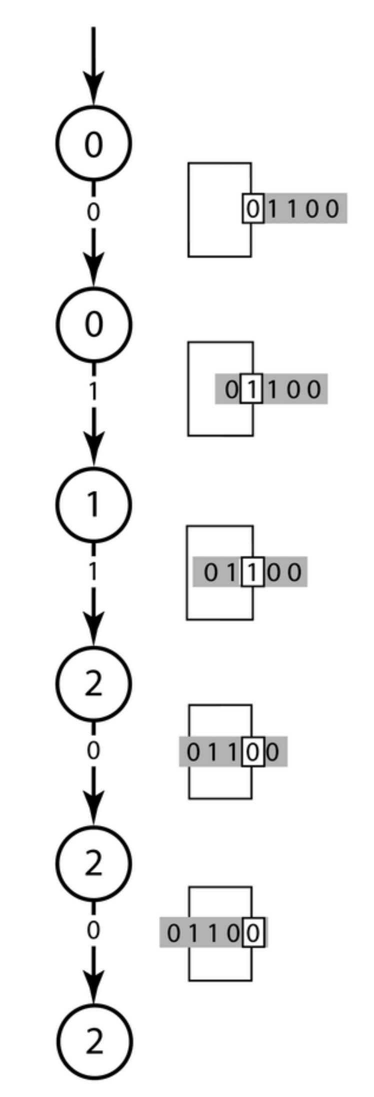
Linear-time because each step is just a state change

Deterministic finite-state automata

DFA review.




```
int pc = 0;
while (!tape.isEmpty())
{
   boolean bit = tape.read();
   if      (pc == 0) { if (!bit) pc = 0; else pc = 1; }
   else if (pc == 1) { if (!bit) pc = 1; else pc = 2; }
   else if (pc == 2) { if (!bit) pc = 2; else pc = 0; }
}
if (pc == 0) System.out.println("accepted");
else         System.out.println("rejected");
```



Duality

RE. Concise way to describe a set of strings.

DFA. Machine to recognize whether a given string is in a given set.

Kleene's theorem.

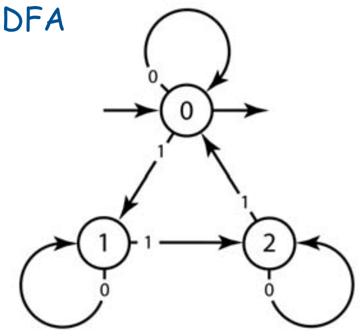
- for any DFA, there exists a RE that describes the same set of strings
- for any RE, there exists a DFA that recognizes the same set of strings

Ex: set of strings whose number of 1's is a multiple of 3

RE

```
0* | (0*10*10*10*)*
```

DFA



Good news: The basic plan works (build DFA from RE and run with text as input)

Bad news : The DFA can be exponentially large (can't afford to build it).

Consequence: We need a smaller abstract machine.

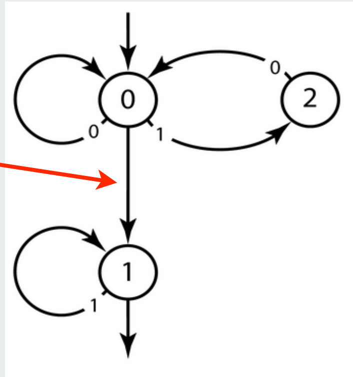
Nondeterministic finite-state automata

NFA.

- may have 0, 1, or more transitions for each input symbol
- may have ε-transitions (move to another state without reading input)
- accept if any sequence of transitions leads to accept state

Ex: set of strings that do not contain **110**

convention:
unlabelled arrows
are ε - transitions



in set: **111**, **00011**, **101001011**

not in set: **110**, **00011011**, **00110**

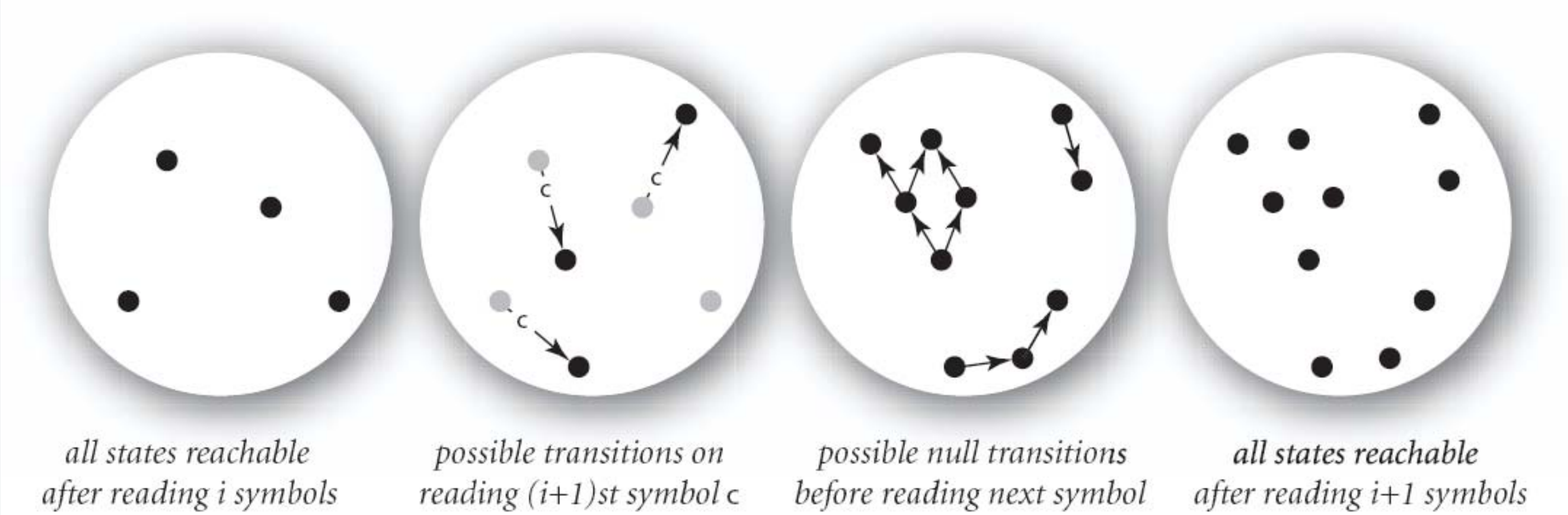
Implication of proof of Kleene's theorem: RE -> NFA -> DFA

Basic plan for GREP (revised)

- build NFA from RE
- simulate NFA with text as input
- give up on linear-time guarantee

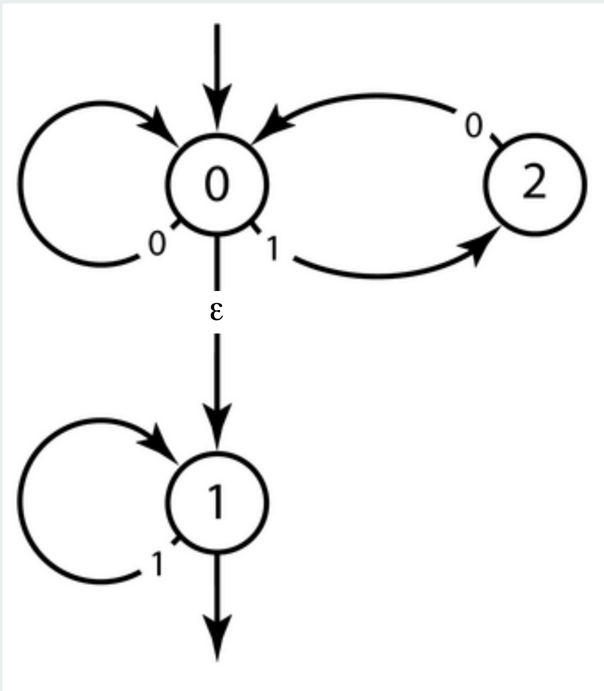
Simulating an NFA

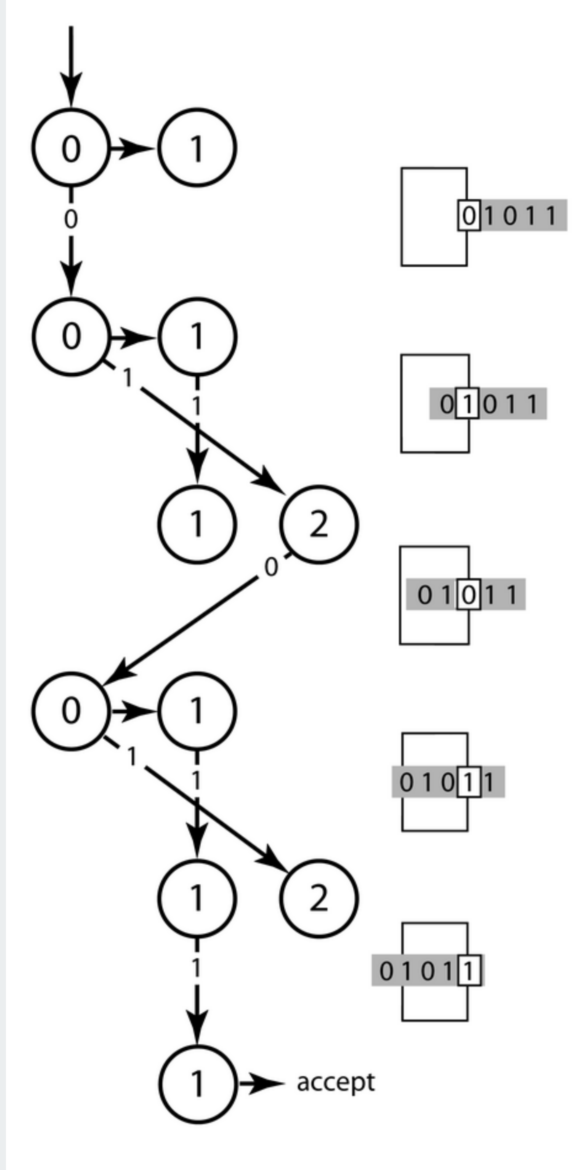
How to simulate an NFA? Maintain set of all possible states that NFA could be in after reading in the first i symbols.



One step in simulating an NFA

NFA Simulation

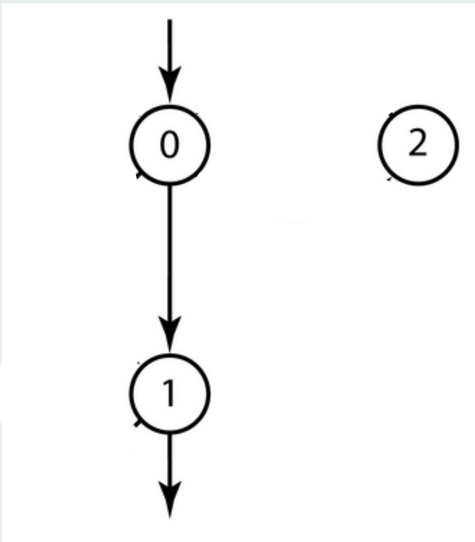




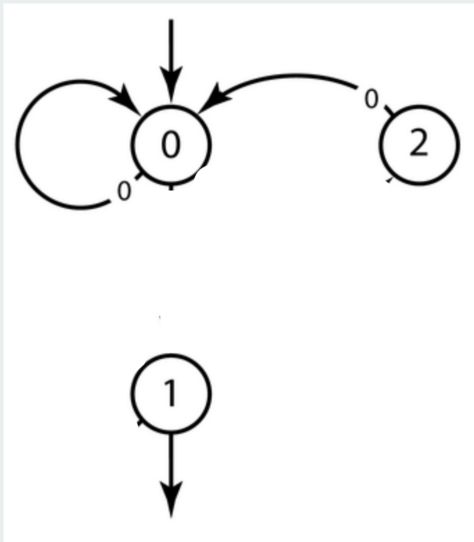
An NFA trace

NFA Representation

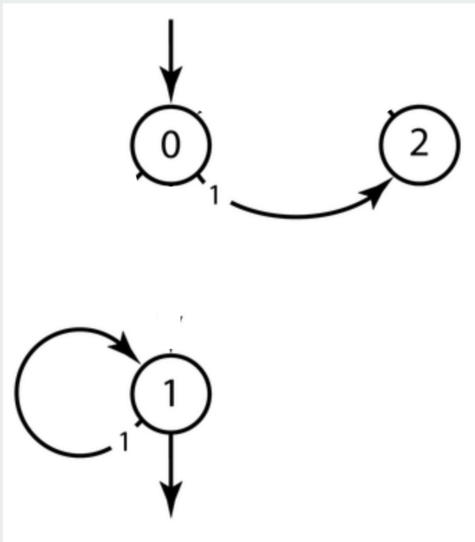
NFA representation. Maintain several digraphs, one for each symbol in the alphabet, plus one for ε.



ε-graph



0-graph



1-graph

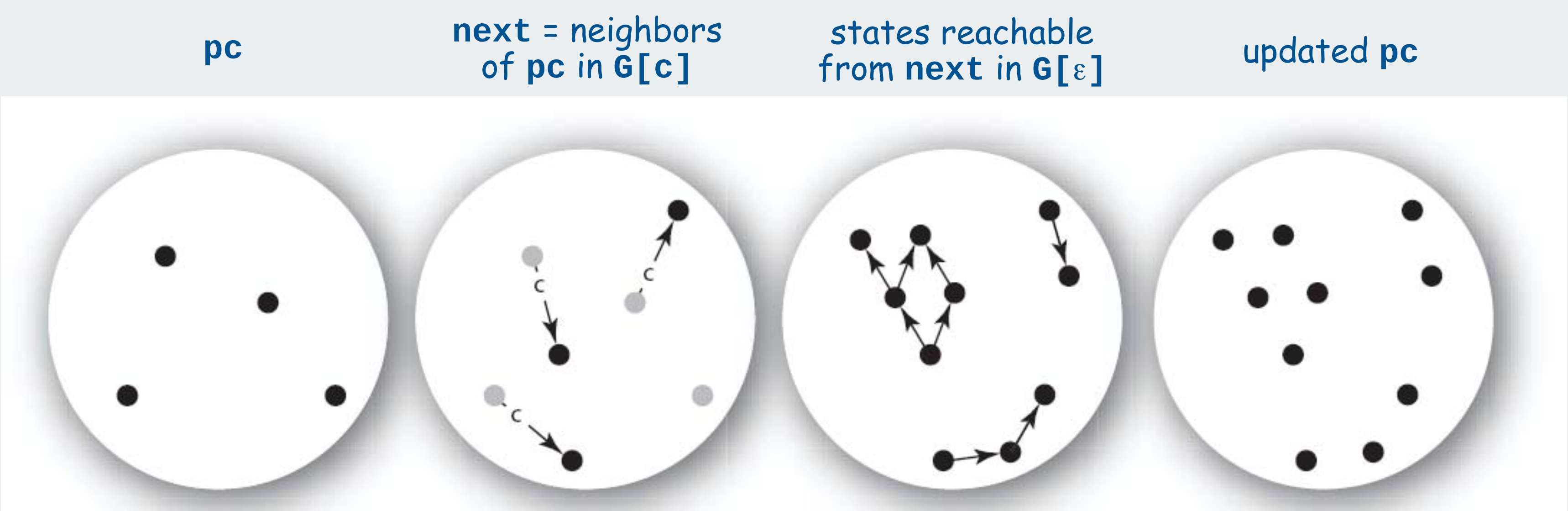
NFA: Java Implementation

```
public class NFA
{
   private int START  = 0;                   // start state
   private int ACCEPT = 1;                   // accept state
   private int N      = 2;                   // number of states
   private String ALPHABET = "01";           // RE alphabet
   private int EPS = ALPHABET.length(); // symbols in alphabet
   private Digraph[] G;

   public NFA(String re)
   {
      G = new Digraph[EPS + 1];
      for (int i = 0; i <= EPS; i++)
         G[i] = new Digraph();
      build(0, 1, re);
   }

   private void build(int from, int to, String re) { }
   public boolean simulate(Tape tape)              { }
}
```

NFA Simulation

How to simulate an NFA?

- Maintain a **SET** of all possible states that NFA could be in after reading in the first i symbols.
- Use **Digraph** adjacency and reachability ops to update.



all states reachable after reading i symbols | *possible transitions on reading (i+1)st symbol c* | *possible null transitions before reading next symbol* | *all states reachable after reading i+1 symbols*

NFA Simulation: Java Implementation

```
public boolean simulate(Tape tape)
{
   SET<Integer> pc = G[EPS].reachable(START);

   while (!tape.isEmpty())
   {  // Simulate NFA taking input from tape.
      char c = tape.read();
      int  i = ALPHABET.indexOf(c);
      SET<Integer> next = G[i].neighbors(pc);

      pc = G[EPS].reachable(next);
   }

   for (int state : pc)
      if (state == ACCEPT) return true;
   return false;
}
```

states reachable from start by ε-transitions

all possible states after reading character c from tape

follow ε-transitions

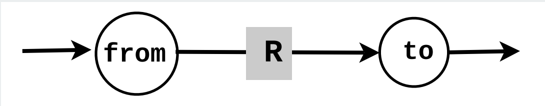
check whether in accept state at end

Converting from an RE to an NFA: basic transformations

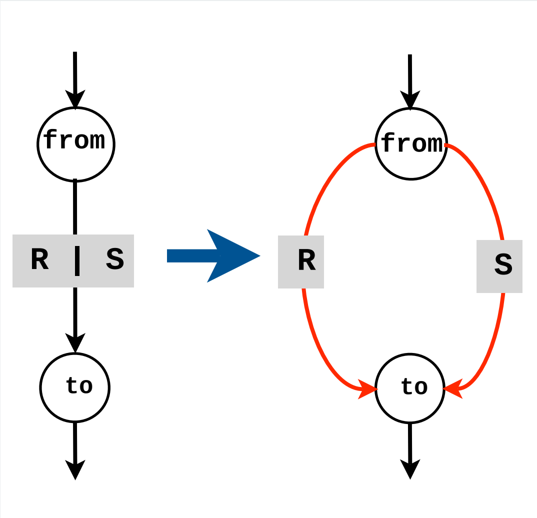
Use generalized NFA with full RE on trasitions arrows

- start with one transition having given RE
- remove operators with transformations given below
- goal: standard NFA (all single-character or epsilon-transitions)

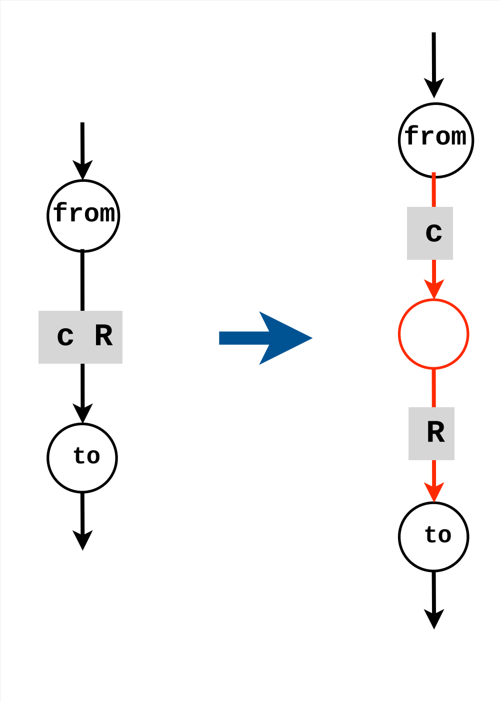
start



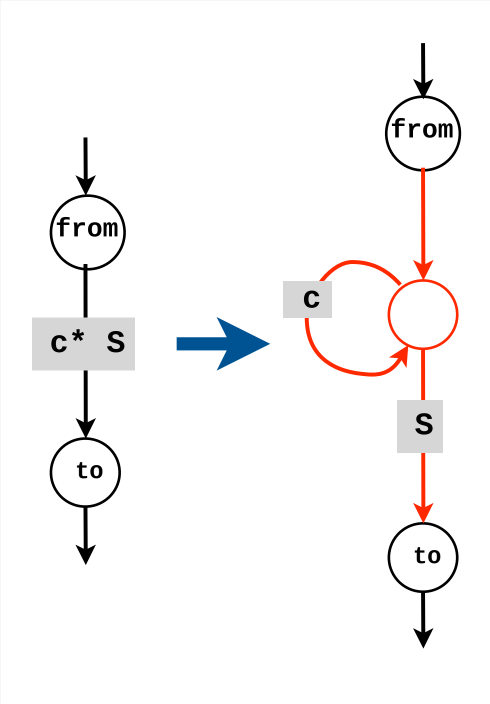
union



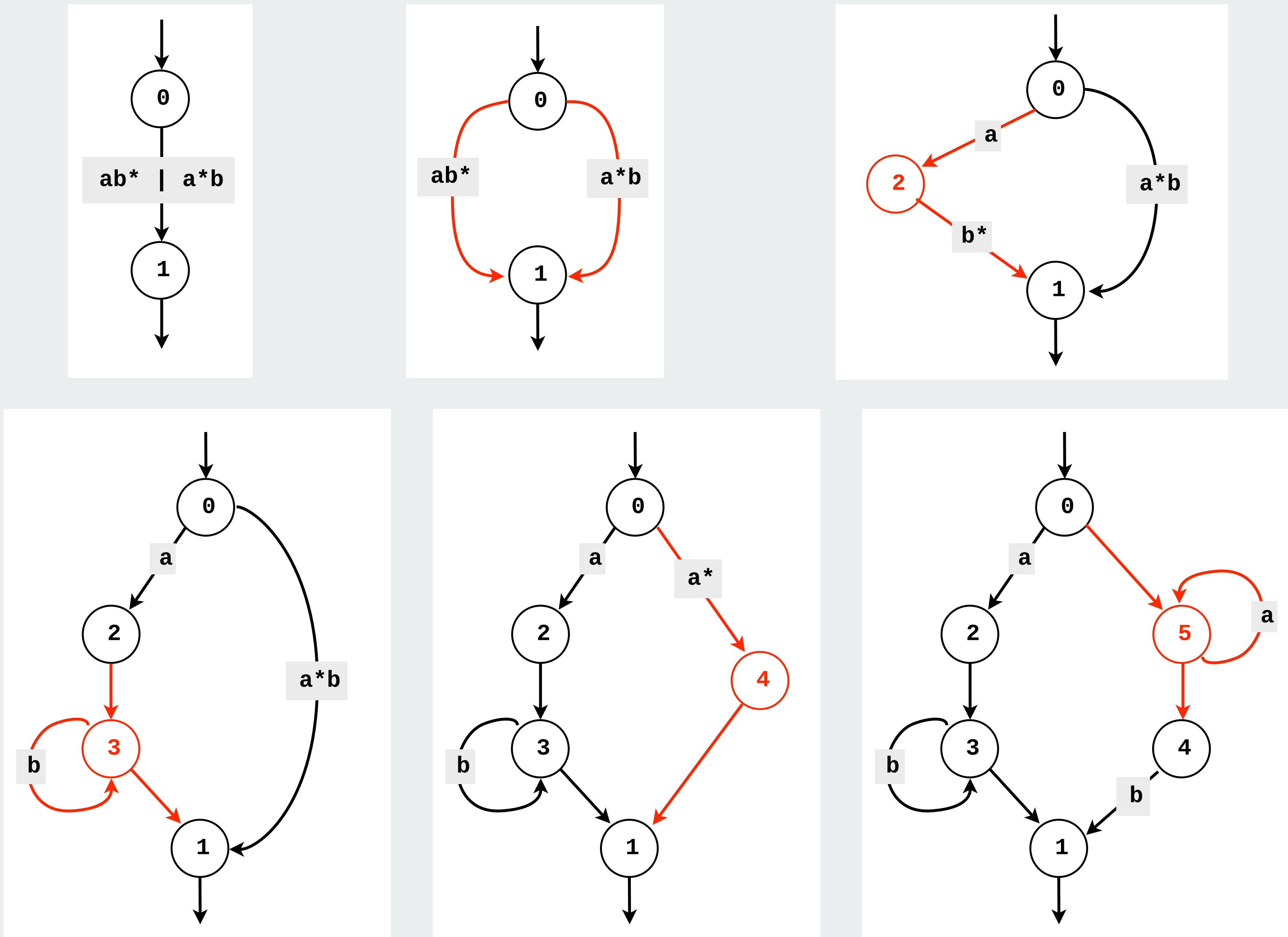
concatenation



closure



Converting from an RE to an NFA example: **ab* | ab***



NFA Construction: Java Implementation

```
private void build(int from, int to, String re)
{
   int or = re.indexOf('|');

   if (re.length() == 0) G[EPSILON].addEdge(from, to);

   else if (re.length() == 1)
   {
```

single char

```
      char c = re.charAt(0);
      for (int i = 0; i < EPSILON; i++)
         if (c == ALPHABET.charAt(i) || c == '.')
            G[i].addEdge(from, to);
   }

   else if (or != -1)
   {
```

union

```
      build(from, to, re.substring(0, or));
      build(from, to, re.substring(or + 1));
   }

   else if (re.charAt(1) == '*')
   {
```

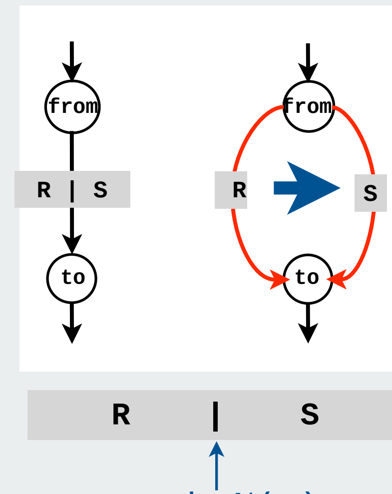
closure

```
      G[EPSILON].addEdge(from, N);
      build(N, N, re.substring(0, 1));
      build(N++, to, re.substring(2));
   }

   else
   {
```

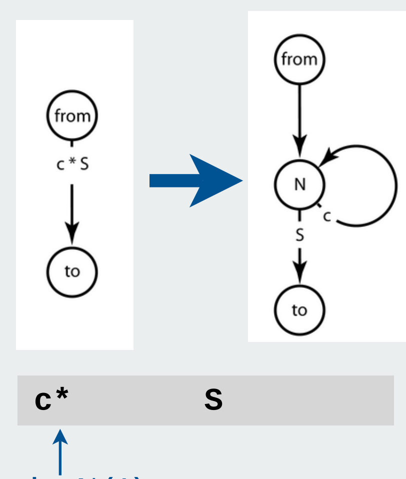
concatenation

```
      build(from, N, re.substring(0, 1));
      build(N++, to, re.substring(1));
   }
}
```

union



re.charAt(or)



re.charAt(1)

Grep running time

Input. Text with N characters, RE with M characters.

Claim. The number of edges in the NFA is at most 2M.

- Single character: consumes 1 symbol, creates 1 edge.
- Wildcard character: consumes 1 symbol, creates 2 edges.
- Concatenation: consumes 1 symbols, creates 0 edges.
- Union: consumes 1 symbol, creates 1 edges.
- Closure: consumes one symbol, creates 2 edges.

NFA simulation. O(MN) since NFA has 2M transitions

- bottleneck: 1 graph reachability per input character
- can be substantially faster in practice if few ε-transitions

NFA construction. Ours is $O(M^2)$ but not hard to make O(M).

Surprising bottom line:

Worst-case cost for grep is the same as for elementary exact match!

Industrial-strength grep implementation

To complete the implementation,

- Deal with parentheses.
- Extend the alphabet.
- Add character classes.
- Add capturing capabilities.
- Deal with meta characters.
- Extend the closure operator.
- Error checking and recovery.
- Greedy vs. reluctant matching.

Regular expressions in Java (revisited)

RE pattern matching is implemented in Java's **`Pattern`** and **`Matcher`** classes

Ex: Harvesting. Print substrings of **`input`** that match **`re`**

```
import java.util.regex.Pattern;
import java.util.regex.Matcher;

public class Harvester
{
   public static void main(String[] args)
   {
      String re       = args[0];
      In in           = new In(args[1]);
      String input    = in.readAll();
      Pattern pattern = Pattern.compile(re);
      Matcher matcher = pattern.matcher(input);
      while (matcher.find())
         System.out.println(matcher.group());
   }
}
```

`compile()` creates a **`Pattern`** (NFA) from RE

`matcher()` creates a **`Matcher`** (NFA simulator) from NFA and text

`find()` looks for the next match

`group()` returns the substring most recently found by **`find()`**

```
% java Harvester "gcg(cgg|agg)*ctg" chromosomeX.txt
gcgcggcggcggcggcggctg
gcgctg
gcgctg
gcgcggcggcggaggcggaggcggctg

% java Harvester "http://(\\w+\\.)*(\\w+)" http://www.cs.princeton.edu
http://www.princeton.edu
http://www.google.com
```

harvest patterns from DNA

harvest links from website

Typical application: Parsing a data file

Example. NCBI genome file, …

```
LOCUS AC146846 128142 bp DNA linear HTG 13-NOV-2003
DEFINITION Ornithorhynchus anatinus clone CLM1-393H9,
ACCESSION AC146846
KEYWORDS HTG; HTGS_PHASE2; HTGS_DRAFT.
SOURCE Ornithorhynchus anatinus (platypus)
ORIGIN
     1 tgtatttcat ttgaccgtgc tgttttttcc cggtttttca gtacggtgtt agggagccac
    61 gtgattctgt ttgttttatg ctgccgaata gctgctcgat gaatctctgc atagacagct  // a comment
   121 gccgcaggga gaaatgacca gtttgtgatg acaaaatgta ggaaagctgt ttcttcataa
   ...
128101 ggaaatgcga cccccacgct aatgtacagc ttctttagat tg
```

```
String regexp   = "[ ]*[0-9]+([actg ]*).*";
Pattern pattern = Pattern.compile(regexp);
In in = new In(filename);
while (!in.isEmpty())
{
   String line = in.readLine();
   Matcher matcher = pattern.matcher(line);
   if (matcher.find())
   {
      String s = matcher.group(1).replaceAll(" ", "");
      // Do something with s.
   }
}
```

replace this RE   with this string

the part of the match delimited
by the first group of parentheses

Algorithmic complexity attacks

Warning. Typical implementations do not guarantee performance!

grep, Java, Perl

```
java Validate "(a|aa)*b" aaaaaaaaaaaaaaaaaaaaaaaaaaaaaac            1.6 seconds
java Validate "(a|aa)*b" aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaac          3.7 seconds
java Validate "(a|aa)*b" aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaac        9.7 seconds
java Validate "(a|aa)*b" aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaac     23.2 seconds
java Validate "(a|aa)*b" aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaac   62.2 seconds
java Validate "(a|aa)*b" aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaac 161.6 seconds
```

SpamAssassin regular expression.

```
java RE "[a-z]+@[a-z]+([a-z\.]+\.)+[a-z]+" spammer@x......................
```

- Takes exponential time.
- Spammer can use a pathological email address to DOS a mail server.

Not-so-regular expressions

Back-references.

- `\1` notation matches sub-expression that was matched earlier.
- Supported by typical RE implementations.

```
java Harvester "\b(.+)\1\b"  dictionary.txt
beriberi
couscous
```

word boundary

Some non-regular languages.

- set of strings of the form ww for some string w: **beriberi**.
- set of bitstrings with an equal number of 0s and 1s: **01110100**.
- set of Watson-Crick complemented palindromes: **atttcggaaat**.

Remark. Pattern matching with back-references is intractable.

Context

Abstract machines, languages, and nondeterminism.

- basis of the theory of computation
- intensively studied since the 1930s
- basis of programming languages

Compiler. A program that translates a program to machine code.

- KMP string ⇒ DFA.
- `grep` RE ⇒ NFA.
- `javac` Java language ⇒ Java byte code.

	KMP	grep	Java
pattern	string	RE	program
parser	unnecessary	check if legal	check if legal
compiler output	DFA	NFA	byte code
simulator	DFA simulator	NFA simulator	JVM

Summary of pattern-matching algorithms

Programmer:

- Implement exact pattern matching by DFA simulation (KMP).
- REs are a powerful pattern matching tool.
- Implement RE pattern matching by NFA simulation (grep).

Theoretician:

- RE is a compact description of a set of strings.
- NFA is an abstract machine equivalent in power to RE.
- DFAs and REs have limitations.

You: Practical application of core CS principles.

Example of essential paradigm in computer science.

- Build intermediate abstractions.
- Pick the right ones!
- Solve important practical problems.