Pattern matching

Substring search. Find a single string in text.

Pattern matching. Find one of a specified set of strings in text.

Ex. [genomics]
- Fragile X syndrome is a common cause of mental retardation.
- Human genome contains triplet repeats of CCG or AGG, bracketed by GCG at the beginning and CTG at the end.
- Number of repeats is variable, and correlated with syndrome.

Pattern text

\text{CGC} \text{(CGG|AGG)} \text{CTG}

\text{GCGCGGCTGCTCGAGAGATGGTTTAAAGCTG}
\text{GCGCGGAGGCGGCTG}
\text{GCGCGGAGGCTG}

Syntax highlighting

```java
public class NFA {
    private Digraph G; // digraph of epsilon transitions
    private String regexp; // regular expression
    private int M; // number of characters in regular expression

    // Create the NFA for the given RE
    public NFA(String regexp) {
        this.regexp = regexp;
        M = regexp.length();
        Stack<Integer> efa = new Stack<Integer>();
        G = new Digraph(M);
    }

    // Output
    public boolean matches(String text) {
        // Implementation...
    }
}
```
Google code search

Pattern matching: applications

Test if a string matches some pattern.
- Process natural language.
- Scan for virus signatures.
- Specify a programming language.
- Access information in digital libraries.
- Search genome using PROSITE patterns.
- Filter text (spam, NetNanny, Carnivore, malware).
- Validate data-entry fields (dates, email, URL, credit card).

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

Regular expressions

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

<table>
<thead>
<tr>
<th>operation</th>
<th>order</th>
<th>example RE</th>
<th>matches</th>
<th>does not match</th>
</tr>
</thead>
<tbody>
<tr>
<td>concatenation</td>
<td>3</td>
<td>AABAAB</td>
<td>AABAAB</td>
<td>every other string</td>
</tr>
<tr>
<td>or</td>
<td>4</td>
<td>AA</td>
<td>BAAB</td>
<td>AA</td>
</tr>
<tr>
<td>closure</td>
<td>2</td>
<td>AB*A</td>
<td>AA</td>
<td>ABBB</td>
</tr>
<tr>
<td>parentheses</td>
<td>1</td>
<td>A(A</td>
<td>B)AAB</td>
<td>AABA</td>
</tr>
</tbody>
</table>

Regular expression shortcuts

Additional operations are often added for convenience.

<table>
<thead>
<tr>
<th>operation</th>
<th>example RE</th>
<th>matches</th>
<th>does not match</th>
</tr>
</thead>
<tbody>
<tr>
<td>wildcard</td>
<td>.U.U.U.</td>
<td>CUMULUS JUGULUM</td>
<td>SUCCUBUS TUMULTUOUS</td>
</tr>
<tr>
<td>character class</td>
<td>[A-Za-z]+[a-z]*</td>
<td>word Capitalized</td>
<td>camelCase 4illegal</td>
</tr>
<tr>
<td>at least 1</td>
<td>A(BC)+DE</td>
<td>ABCDE ABCBCDE</td>
<td>ADE BCDE</td>
</tr>
<tr>
<td>exactly k</td>
<td>[0-9]{5}-[0-9]{4}</td>
<td>08540-1321 19072-5541</td>
<td>11111111 166-54-111</td>
</tr>
<tr>
<td>complement</td>
<td>[*AEIOU]{6}</td>
<td>RHYTHM</td>
<td>DECADE</td>
</tr>
</tbody>
</table>

Ex. \[A-E]+ is shorthand for \((A\|B\|C\|D\|E)\)A\(B\|C\|D\|E\)+
Regular expression examples

**RE notation is surprisingly expressive**

<table>
<thead>
<tr>
<th>regular expression</th>
<th>matches</th>
<th>does not match</th>
</tr>
</thead>
<tbody>
<tr>
<td>.<em>SPB.</em></td>
<td>SPB</td>
<td>SPB'</td>
</tr>
<tr>
<td>(0-9)[3-4]</td>
<td>12</td>
<td>03, 34</td>
</tr>
<tr>
<td>\b[a-z]+@([a-z]+.)+(edu</td>
<td>com)</td>
<td>email addresses</td>
</tr>
<tr>
<td>[0-9]{3}-[0-9]{2}-[0-9]{4}</td>
<td>Social Security numbers</td>
<td>123-45-6789</td>
</tr>
<tr>
<td>[$_A-Za-z_][$_A-Za-z0-9]*</td>
<td>Java identifiers</td>
<td>ident3</td>
</tr>
</tbody>
</table>

REs play a well-understood role in the theory of computation.

Can the average web surfer learn to use REs?

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

**Regular expressions to the rescue**

http://xkcd.com/208

Can the average programmer learn to use REs?

Perl RE for valid RFC822 email addresses

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

“Some people, when confronted with a problem, think 'I know I'll use regular expressions.' Now they have two problems.” — Jamie Zawinski (flame war on alt.religion.emacs)

Bottom line. REs are amazingly powerful and expressive, but using them in applications can be amazingly complex and error-prone.

Duality between REs and DFAs

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.

Pattern matching implementation: basic plan (first attempt)

Overview is the same as for KMP.
• No backup in text input stream.
• Linear-time guarantee.

Underlying abstraction. Deterministic finite state automata (DFA).

Basic plan. [apply Kleene's theorem]
• Build DFA from RE.
• Simulate DFA with text as input.

Bad news. Basic plan is infeasible (DFA may have exponential # of states).
Pattern matching implementation: basic plan (revised)

Overview is similar to KMP.
- No backup in text input stream.
- Quadratic-time guarantee (linear-time typical).

Underlying abstraction. **Nondeterministic finite state automata** (NFA).

Basic plan. [apply Kleene’s theorem]
- Build NFA from RE.
- Simulate NFA with text as input.

**Q. What is an NFA?**

**Nondeterministic finite-state automata**

**Regular-expression-matching NFA.**
- RE enclosed in parentheses.
- One state per RE character (start = 0, accept = M).
- Red ε-transition (change state, but don’t scan text).
- Black match transition (change state and scan to next text char).
- Accept if any sequence of transitions ends in accept state.

Nondeterminism.
- One view: machine can guess the proper sequence of state transitions.
- Another view: sequence is a proof that the machine accepts the text.

**Q. Is AAAABD matched by NFA?**

**A. Yes, because some sequence of legal transitions ends in state 11.**

**NFA corresponding to the pattern ((A*B|A*C)D)**
Q. Is $AAAC$ matched by NFA?
A. No, because no sequence of legal transitions ends in state $11$.
   [but need to argue about all possible sequences]

Nondeterministic finite-state automata

$A A A A C$

$0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 2 \rightarrow 3 \rightarrow 2 \rightarrow 3 \rightarrow 4$

no way out of state $4$

NFA corresponding to the pattern ($(A^* B I A C) D$)

Q. How to determine whether a string is matched by an automaton?

DFA. Deterministic ⇒ exactly one applicable transition.

NFA. Nondeterministic ⇒ can be several applicable transitions; need to select the right one!

Q. How to simulate NFA?
A. Systematically consider all possible transition sequences.

Nondeterminism

R. REGULAR EXPRESSIONS

- REs and NFAs
- NFA simulation
- NFA construction
- Applications

Regular Expressions

- State names. Integers from 0 to $M$.
- Match-transitions. Keep regular expression in array $re[]$.
- $\varepsilon$-transitions. Store in a digraph $G$.

0 → 1, 1 → 2, 1 → 6, 2 → 3, 3 → 2, 3 → 4, 5 → 8, 8 → 9, 10 → 11

NFA representation

NFA corresponding to the pattern ($(A^* B I A C) D$)
NFA simulation

Q. How to efficiently simulate an NFA?
A. Maintain set of all possible states that NFA could be in after reading in the first i text characters.

Q. How to perform reachability?

NFA simulation

Goal. Check whether input matches pattern.

set of states reachable from start: 0

set of states reachable via ε-transitions from start
NFA simulation

Read next input character.
• Find states reachable by match transitions.
• Find states reachable by ε-transitions

set of states reachable via ε-transitions from start : { 0, 1, 2, 4, 6 }

NFA simulation

Read next input character.
• Find states reachable by match transitions.
• Find states reachable by ε-transitions

set of states reachable after matching A : { 3, 7 }

NFA simulation

Read next input character.
• Find states reachable by match transitions.
• Find states reachable by ε-transitions

set of states reachable via ε-transitions after matching A
NFA simulation

Read next input character.
• Find states reachable by match transitions.
• Find states reachable by $\varepsilon$-transitions

input: A A B D

set of states reachable via $\varepsilon$-transitions after matching A: { 2, 3, 4, 7 }

NFA simulation

Read next input character.
• Find states reachable by match transitions.
• Find states reachable by $\varepsilon$-transitions

input: A A B D

match A transitions

set of states reachable after matching A A

NFA simulation

Read next input character.
• Find states reachable by match transitions.
• Find states reachable by $\varepsilon$-transitions

input: A A B D

set of states reachable after matching A A: { 3 }

NFA simulation

Read next input character.
• Find states reachable by match transitions.
• Find states reachable by $\varepsilon$-transitions

input: A A B D

$\varepsilon$-transitions

set of states reachable via $\varepsilon$-transitions after matching A A
NFA simulation

Read next input character.
• Find states reachable by match transitions.
• Find states reachable by ε-transitions

input A A B D

set of states reachable via ε-transitions after matching A A :  { 2, 3, 4 }

NFA simulation

Read next input character.
• Find states reachable by match transitions.
• Find states reachable by ε-transitions

input A A B D

match B transition

set of states reachable after matching A A B

NFA simulation

Read next input character.
• Find states reachable by match transitions.
• Find states reachable by ε-transitions

input A A B D

set of states reachable after matching A A B :  { 5 }

NFA simulation

Read next input character.
• Find states reachable by match transitions.
• Find states reachable by ε-transitions

input A A B D

ε-transitions

set of states reachable via ε-transitions after matching A A B
**NFA simulation**

Read next input character.
- Find states reachable by match transitions.
- Find states reachable by ε-transitions

input: A A B D

set of states reachable via ε-transitions after matching A A B: { 5, 8, 9 }

match D transition

set of states reachable after matching A A B D: { 10 }

set of states reachable via ε-transitions after matching A A B D: { 10 }

input: A A B D
NFA simulation

Read next input character.
• Find states reachable by match transitions.
• Find states reachable by ε-transitions

NFA simulation

When no more input characters:
• Accept if any state reachable is an accept state.
• Reject otherwise.

Digraph reachability

Digraph reachability. Find all vertices reachable from a given source or set of vertices.

Solution. Run DFS from each source, without unmarking vertices.
Performance. Runs in time proportional to $E + V$.

NFA simulation: Java implementation

```java
class NFA {
  private char[] re; // match transitions
  private Digraph G; // ε-transition digraph
  private int M; // number of states

  public NFA(String regexp) {
    M = regexp.length();
    re = regexp.toCharArray();
    G = buildEpsilonTransitionsDigraph();
  }

  public boolean recognizes(String txt) {
    /* see next slide */
  }

  public Digraph buildEpsilonTransitionDigraph() {
    /* stay tuned */
  }
}
```
public boolean recognizes(String txt) {
    Bag<Integer> pc = new Bag<Integer>();
    DirectedDFS dfs = new DirectedDFS(G, 0);
    for (int v = 0; v < G.V(); v++)
        if (dfs.marked(v)) pc.add(v);
    for (int i = 0; i < txt.length(); i++) {
        Bag<Integer> match = new Bag<Integer>();
        for (int v : pc) {
            if (v == M) continue;
            if ((re[v] == txt.charAt(i)) || re[v] == '.')
                match.add(v+1);
        }
        dfs = new DirectedDFS(G, match);
        pc = new Bag<Integer>();
        for (int v = 0; v < G.V(); v++)
            if (dfs.marked(v)) pc.add(v);
    }
    for (int v : pc)
        if (v == M) return true;
    return false;
}
Building an NFA corresponding to an RE

**Concatenation.** Add match-transition edge from state corresponding to characters in the alphabet to next state.

**Alphabet.** A B C D

**Metacharacters.** ( ) . * |

Building an NFA corresponding to an RE

**Parentheses.** Add $\epsilon$-transition edge from parentheses to next state.

Building an NFA corresponding to an RE

**Closure.** Add three $\epsilon$-transition edges for each * operator.

Building an NFA corresponding to an RE

**Or.** Add two $\epsilon$-transition edges for each | operator.
Goal. Write a program to build the $\varepsilon$-transition digraph.

Challenges. Remember left parentheses to implement closure and or; need to remember $|$ to implement or.

Solution. Maintain a stack.
- $($ symbol: push $($ onto stack.
- $|$ symbol: push $|$ onto stack.
- $)$ symbol: pop corresponding $($ and possibly intervening $|$; add $\varepsilon$-transition edges for closure/or.

NFA construction: implementation

NFA construction

NFA construction

NFA construction
NFA construction

Alphabet symbol.
- Add match transition to next state.
- Do one-character lookahead:
  add $\varepsilon$-transitions if next character is $\ast$.

NFA construction

Alphabet symbol.
- Add match transition to next state.
- Do one-character lookahead:
  add $\varepsilon$-transitions if next character is $\ast$.

NFA construction

Closure symbol.
- Add $\varepsilon$-transition to next state.

NFA construction

Alphabet symbol.
- Add match transition to next state.
- Do one-character lookahead:
  add $\varepsilon$-transitions if next character is $\ast$. 
NFA construction

Or symbol.
• Push index of state corresponding to | onto stack.

stack

0 1 2 3 4 5

( ( A * B | A C ) D )

NFA construction

Alphabet symbol.
• Add match transition to next state.
• Do one-character lookahead:
  add $\varepsilon$-transitions if next character is *.

Right parenthesis.
• Add $\varepsilon$-transition to next state.
• Pop corresponding ( and possibly intervening | ;
  add $\varepsilon$-transition edges for or.
• Do one-character lookahead:
  add $\varepsilon$-transitions if next character is *.
NFA construction

Alphabet symbol.
- Add match transition to next state.
- Do one-character lookahead:
  add $\epsilon$-transitions if next character is $\ast$.

$((A^*B I A C) D)$

End of regular expression.
- Add accept state.

$((A^*B I A C) D)$

NFA construction

Right parenthesis.
- Add $\epsilon$-transition to next state.
- Pop corresponding ( and possibly intervening );
  add $\epsilon$-transition edges for or.
- Do one-character lookahead:
  add $\epsilon$-transitions if next character is $\ast$.

$((A^*B I A C) D)$

NFA corresponding to the pattern $((A^*B I A C) D)$
NFA construction: Java implementation

```java
private Digraph buildEpsilonTransitionDigraph() {
    Digraph G = new Digraph(M+1);
    Stack<Integer> ops = new Stack<Integer>;
    for (int i = 0; i < M; i++) {
        int lp = i;
        if (re[i] == '(' || re[i] == '|') ops.push(i);
        else if (re[i] == ')') {
            int or = ops.pop();
            if (re[or] == '|') {
                lp = ops.pop();
                G.addEdge(lp, or+1);
                G.addEdge(or, i);
            }
            else lp = or;
        }
        if (i < M-1 && re[i+1] == '*') {
            G.addEdge(lp, i+1);
            G.addEdge(i+1, lp);
        }
        if (re[i] == '(' || re[i] == '*' || re[i] == ')')
            G.addEdge(i, i+1);
    }
    return G;
}
```

NFA construction: analysis

Proposition. Building the NFA corresponding to an $M$-character RE takes time and space proportional to $M$.

Proof. For each of the $M$ characters in the RE, we add at most three $\epsilon$-transitions and execute at most two stack operations.

---

Generalized regular expression print

Grep. Take a RE as a command-line argument and print the lines from standard input having some substring that is matched by the RE.

```java
public class GREP
{
    public static void main(String[] args)
    {
        String regexp = "\.* + args[0] + \.*\);
        NFA nfa = new NFA(regexp);
        while (StdIn.hasNextLine())
        {
            String line = StdIn.readLine();
            if (nfa.recognizes(line))
                StdOut.println(line);
        }
    }
}
```

Bottom line. Worst-case for grep (proportional to $MN$) is the same as for brute-force substring search.
Typical grep application: crossword puzzles

% more words.txt
a
aback
abacus
abalone
abandon
...
% grep "s..ict.." words.txt
constrictor
strict
strict

Industrial-strength grep implementation

To complete the implementation:
• Add character classes.
• Handle metacharacters.
• Add capturing capabilities.
• Extend the closure operator.
• Error checking and recovery.
• Greedy vs. reluctant matching.

Ex. Which substring(s) should be matched by the RE <blink>.*</blink>?

\[\text{reluctant}\]  \[\text{reluctant}\]
<blink>text</blink><\text{some text</blink><\text{more text</blink>\]</\text{greedy}>

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]*$’ words.txt | egrep ‘...........’

typewritten

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-Z][A-Za-z]*$/ ' words.txt — print all words that start with uppercase letter
do for each line

Regular expressions in Java

Validity checking. Does the input match the regexp?
Java string library. Use input.matches(regexp) for basic RE matching.

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

% java Validate "[A-Za-z][A-Za-z-0-9]*" ident12
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
Harvesting information

Goal. Print all substrings of input that match a RE.

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

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

RE pattern matching is implemented in Java's `java.util.regex.Pattern` and `java.util.regex.Matcher` classes.

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

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

Harvesting information

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()

Algorithmic complexity attacks

Warning. Typical implementations do not guarantee performance!

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

SpamAssassin regular expression.

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

• Takes exponential time on pathological email addresses.
• Troublemaker can use such addresses to DOS a mail server.

Not-so-regular expressions

Back-references.
• \1 notation matches subexpression that was matched earlier.
• Supported by typical RE implementations.

```regex
(\.+)\1 // beriberi couscous
1?\$|\([^1+]?\)\1+ // 1111 111111 1111111111
```

Some non-regular languages.
• Strings of the form w w for some string w: beriberi.
• Unary strings with a composite number of 1s: 111111.
• Bistreams with an equal number of 0s and 1s: 01110100.
• Watson-Crick complemented palindromes: atttggaat.

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.

**Summary of pattern-matching algorithms**

**Programmer.**
- Implement substring search via DFA simulation.
- Implement RE pattern matching via NFA simulation.

**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 computer science principles.

**Example of essential paradigm in computer science.**
- Build intermediate abstractions.
- Pick the right ones!
- Solve important practical problems.