7. Theory of Computation

Two fundamental questions.
- What can a computer do?
- What can a computer do with limited resources?

General approach.
- Don’t talk about specific machines or problems.
- Consider minimal abstract machines.
- Consider general classes of problems.

Why Learn Theory?

In theory ...
- Deeper understanding of what is a computer and computing.
- Foundation of all modern computers.
- Pure science.
- Philosophical implications.

In practice ...
- Web search: theory of pattern matching.
- Sequential circuits: theory of finite state automata.
- Compilers: theory of context free grammars.
- Cryptography: theory of computational complexity.
- Data compression: theory of information.

Regular Expressions

“In theory there is no difference between theory and practice. In practice there is.” – Yogi Berra
Describing a Pattern

PROSITE. Huge database of protein families and domains.

Q. How to describe a protein motif?

Ex. [signature of the C2H2-type zinc finger domain]
   - C
   - Between 2 and 4 amino acids.
   - C
   - 3 more amino acids.
   - One of the following amino acids: LIVMFYWCX.
   - H
   - Between 3 and 5 more amino acids.
   - H

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, ads, 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 TOY input file format.
- Automatically create Java documentation from Javadoc comments.

Regular Expressions: Basic Operations

Regular expression. Notation to specify a set of strings.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Regular Expression</th>
<th>Yes</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concatenation</td>
<td>aabaab</td>
<td>aabaab</td>
<td>every other string</td>
</tr>
<tr>
<td>Wildcard</td>
<td>. . . .</td>
<td>cumulus jugulum</td>
<td>succubus tumultuous</td>
</tr>
<tr>
<td>Union</td>
<td>aa</td>
<td>baab</td>
<td>aa baab</td>
</tr>
<tr>
<td>Closure</td>
<td>ab+a</td>
<td>aa abba</td>
<td>ab ababa</td>
</tr>
<tr>
<td>Parentheses</td>
<td>a(a</td>
<td>b)aab</td>
<td>aabababab</td>
</tr>
<tr>
<td></td>
<td>(ab)*a</td>
<td>a ababababa</td>
<td>abbbba</td>
</tr>
</tbody>
</table>

Regular Expressions: Examples

Regular expression. Notation is surprisingly expressive.

<table>
<thead>
<tr>
<th>Regular Expression</th>
<th>Yes</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>.<em>spb.</em></td>
<td>raspberry</td>
<td>subspace</td>
</tr>
<tr>
<td>contains the trigraph spb</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a*</td>
<td>(a<em>ba</em>ba*)*</td>
<td>bbb</td>
</tr>
<tr>
<td>multiple of three b’s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>aa abbaabbbbaa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bbbabaabbbbaa</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.*0 ....</td>
<td>1000234</td>
<td>111111111</td>
</tr>
<tr>
<td>fifth to last digit is 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>98701234</td>
<td>403982772</td>
<td></td>
</tr>
<tr>
<td>gcg(cgg</td>
<td>agg)*ctg</td>
<td>gcgcgctg</td>
</tr>
<tr>
<td>fragile X syndrome indicator</td>
<td></td>
<td></td>
</tr>
<tr>
<td>gcgcgagggctg</td>
<td></td>
<td></td>
</tr>
<tr>
<td>gcgcggctg</td>
<td></td>
<td></td>
</tr>
<tr>
<td>gcgcgg</td>
<td></td>
<td></td>
</tr>
<tr>
<td>gcgcggctg</td>
<td></td>
<td></td>
</tr>
<tr>
<td>gcgcggctg</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Generalized Regular Expressions

Regular expressions are a standard programmer's tool.
- Built in to Java, Perl, Unix, Python, ...
- Additional operations typically added for convenience.
- Ex: \[a-e]+ is shorthand for \((a|b|c|d|e)(a|b|c|d|e)\)*.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Regular Expression</th>
<th>Yes</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>One or more</td>
<td>a(bc) +de</td>
<td>abode</td>
<td>ade</td>
</tr>
<tr>
<td>Character classes</td>
<td>[A-Za-z][a-z]*</td>
<td>lowercase CamelCase</td>
<td>illegal</td>
</tr>
<tr>
<td>Exactly k</td>
<td>[0-9]{5}-[0-9]{4}</td>
<td>08540-1321</td>
<td>1111111111</td>
</tr>
<tr>
<td>Negations</td>
<td>[^aeiou]{6}</td>
<td>rhythm</td>
<td>decade</td>
</tr>
</tbody>
</table>

Regular Expressions in Java

Validity checking. Is input in the set described by the re?

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

```
% java Validate "C.{2,4}C...[LIVMFYWC].{8}H.{3,5}H" CAASCGGPyACGDGAAGYHAGAH
true
% java Validate "[$_A-Za-z][$_A-Za-z0-9]*" ident123
true
% java Validate "([A-Za-z]+([-A-Za-z0-9]+))"wayne@cs.princeton.edu
true
```

String Searching Methods

```java
class String {  
    // Java's String library
    boolean matches(String re) {
        // does this string match the given regular expression
    }
    String replaceAll(String re, String str) {
        // replace all occurrences of regular expression with the replacement string
    }
    int indexOf(String r, int from) {
        // return the index of the first occurrence of the string r after the index from
    }
    String[] split(String re) {
        // split the string around matches of the given regular expression
    }
}
```

```
String s = StdIn.readAll();
words = s.split("\s+");
```

String Searching Methods

```java
class String {  
    // Java's String library
    boolean matches(String re) {
        // does this string match the given regular expression
    }
    String replaceAll(String re, String str) {
        // replace all occurrences of regular expression with the replacement string
    }
    int indexOf(String r, int from) {
        // return the index of the first occurrence of the string r after the index from
    }
    String[] split(String re) {
        // split the string around matches of the given regular expression
    }
}
```

```
String s = StdIn.readAll();
s = s.replaceAll("\s+", "");
```
Solving the Pattern Match Problem

Regular expressions are a concise way to describe patterns.
- How would you implement the method `matches()`?
- Hardware: build a deterministic finite state automaton (DFA).
- Software: simulate a DFA.

**DFA:** simple machine that solves a pattern match problem.
- Different machine for each pattern.
- Accepts or rejects string specified on input tape.
- Focus on true or false questions for simplicity.

---

**Deterministic Finite State Automaton (DFA)**

Simple machine with \( N \) states.
- Begin in start state.
- Read first input symbol.
- Move to new state, depending on current state and input symbol.
- Repeat until last input symbol read.
- Accept input string if last state is labeled \( Y \).

---

**DFA and RE Duality**

**RE.** Concise way to describe a set of strings.
**DFA.** Machine to recognize whether a given string is in a given set.

**Duality.** 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.

**Practical consequence of duality proof:** to match RE, (i) build DFA and (ii) simulate DFA on input string.
Implementing a Pattern Matcher

Problem. Given a RE, create program that tests whether given input is in set of strings described.

Step 1. Build the DFA.
- A compiler!
- See COS 226 or COS 320.

Step 2. Simulate it with given input.

```java
State state = start;
while (!StdIn.isEmpty()) {
    char c = StdIn.readChar();
    state = state.next(c);
}
StdOut.println(state.accept());
```

Application: Harvester

Harvest information from input stream.

- Harvest patterns from DNA.
- Harvest email addresses from web for spam campaign.

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

public class Harvester {
    public static void main(String[] args) {
        String re = "[a-z]+\@[a-zA-Z0-9.]*\.(edu|com)";
        Matcher matcher = Pattern.compile(re).matcher(filename);
        while (matcher.find()) {
            StdOut.println(matcher.group());
        }
    }
}
```

Application: Parsing a Data File

Ex: parsing an NCBI genome data file.

```java
String re = "\[\]*\[0-9\]+(\[actg \]*).*";
Pattern pattern = Pattern.compile(re);
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
    }
}
```
Summary

Programmer.
- Regular expressions are a powerful pattern matching tool.
- Implement regular expressions with finite state machines.

Theoretician.
- RE is a compact description of a set of strings.
- DFA is an abstract machine that solves RE pattern match problem.

You. Practical application of core CS principles.

Fundamental Questions

Q. Are there patterns that cannot be described by any RE/DFA?
A. Yes.
- Bit strings with equal number of 0s and 1s.
- Decimal strings that represent prime numbers.
- DNA strings that are Watson-Crick complemented palindromes.

Q. Can we extend RE/DFA to describe richer patterns?
A. Yes.
- Context free grammar (e.g., Java).
- Turing machines.