7. Theory of Computation

Q. What can a computer do?
Q. 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.

Pioneering work in the 1930s.
- Princeton == center of universe.
- Automata, languages, computability, universality, complexity, logic.

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.

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

Regular Expressions
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.
- 8 more amino acids.
- 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.
- Access information in digital libraries.
- 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>matches</th>
<th>does not match</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>.u.u.u.</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>aaaaab</td>
</tr>
<tr>
<td></td>
<td>(ab)*a</td>
<td>abababa</td>
<td>aa abba</td>
</tr>
</tbody>
</table>

Regular Expressions: Examples

Regular expression. 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>raspberry</td>
<td>subspace</td>
</tr>
<tr>
<td>contains the trigraph spb</td>
<td>crispbread</td>
<td>subspecies</td>
</tr>
<tr>
<td>a*</td>
<td>(a<em>ba</em>a<em>ba</em>)*</td>
<td>multiple of three b’s</td>
</tr>
<tr>
<td>fifth to last digit is 0</td>
<td>1000234 98701234</td>
<td>11111111 403982772</td>
</tr>
<tr>
<td>gcg(ogg</td>
<td>agg)*ctg</td>
<td>fragile X syndrome indicator</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>matches</th>
<th>does not match</th>
</tr>
</thead>
<tbody>
<tr>
<td>one or more</td>
<td>a(bc)+de</td>
<td>abode abodecode</td>
<td>ade bode</td>
</tr>
<tr>
<td>character class</td>
<td>[A-Za-z][a-z]*</td>
<td>lowercase</td>
<td>capitalized 4illegal</td>
</tr>
<tr>
<td>exactly k</td>
<td>[0-9]{5}-[0-9]{4}</td>
<td>08540-1321 19072-5541</td>
<td>11111111</td>
</tr>
<tr>
<td>negation</td>
<td>[^aeiou]{6}</td>
<td>rhythm</td>
<td>decade</td>
</tr>
</tbody>
</table>

String Searching Methods

```java
public class String { (Java's String library)
    boolean matches(String re)
    replace all occurrences of regular expression
    String replaceAll(String re, String str)
    int indexOf(String r, int from)
    split the string around matches of the given regular expression
    String[] split(String re)
}
```

```java
String s = StdIn.readAll();
s = s.replaceAll("\s+", " ");
```
replace all sequences of whitespace characters with a single space

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

```java
public 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" CAASCGGYPACGGAAGYHAGAH
true
legal Java identifier
% java Validate "[_$A-Za-z\[$_A-Za-z0-9]*" ident123
true
valid email address (simplified)
% java Validate \[a-z]+@[a-z]+\.[edu](edu|com) "wayne@cs.princeton.edu"
true
need quotes to "escape" the shell

String Searching Methods

```java
String s = StdIn.readAll();
String[] words = s.split("\s+"个百分
create array of words in document
regular expression that matches any whitespace character
```

```java
String s = StdIn.readAll();
String[] words = s.split("\s+"个百分
create array of words in document
regular expression that matches any whitespace character
```
DFAs

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.
**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.
  ```
  % java Harvester "gcg(cgg|agg)*ctg" chromosomeX.txt
  gcgcggcgcggcggcggcggctg
  gcgcggcgcggcggcggcggctg
  gcgcggcgcggcggcggcggcggctg
  ```

- Harvest email addresses from web for spam campaign.
  ```
  % java Harvester "[a-z]+@[a-z]+(.)+(edu|com)" http://www.princeton.edu/~cos126
  rs@cs.princeton.edu
  maia@cs.princeton.edu
  doug@cs.princeton.edu
  wayne@cs.princeton.edu
  ```

**Application: Parsing a Data File**

Ex: parsing an NCBI genome data file.

```java
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()) {
            StdOut.println(matcher.group());
            // the match most recently found
        }
    }
}
```
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()) {
        extract the part of match in parens
    }
}
```

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.