14. Introduction to Theoretical CS
14. Introduction to Theoretical CS

- Overview
- Regular expressions
- DFAs
- Applications
- Limitations
Introduction to theoretical computer science

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.

Surprising outcome. Sweeping and relevant statements about all computers.
Why study theory?

In theory...
• Deeper understanding of computation.
• 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
practical application."
— Yogi Berra
Abstract machines

Abstract machine
- Mathematical model of computation.
- Each machine defined by specific rules for transforming input to output.
- This lecture: Deterministic finite automata (DFAs).

Formal language
- A set of strings.
- Each defined by specific rules that characterize it.
- This lecture: Regular expressions (REs).

Questions for this lecture
- Is a given string in the language defined by a given RE, or not?
- Can a DFA help answer this question?
14. Introduction to Theoretical CS

- Overview
- Regular expressions
- DFAs
- Applications
- Limitations
Pattern matching

**Pattern matching problem.** Is a given string an element of a given set of strings?

**Example 1** (from computational biochemistry)

An *amino acid* is represented by one of the characters CAVLIMCRKHDENQSTYFWP.

A *protein* is a string of amino acids.

A **C$_2$H$_2$-type zinc finger domain signature** is
- C followed by 2, 3, or 4 amino acids, followed by
- C followed by 3 amino acids, followed by
- L, I, V, M, F, Y, W, C, or X followed by 8 amino acids, followed by
- H followed by 3, 4, or 5 amino acids, followed by H.

**Q.** Is this protein in the C$_2$H$_2$-type zinc finger domain?

**A.** Yes.

![Diagram of protein structure]
Pattern matching

Example 2 (from commercial computing)

An e-mail address is
• A sequence of letters, followed by
• the character "@", followed by
• followed by a nonempty sequence of lowercase letters, followed by the character "."
• [any number of occurrences of the previous pattern]
• "edu" or "com" (others omitted for brevity).

Q. Which of the following are e-mail addresses?

<table>
<thead>
<tr>
<th></th>
<th>A.</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="mailto:rs@cs.princeton.edu">rs@cs.princeton.edu</a></td>
<td>✓</td>
</tr>
<tr>
<td>not an e-mail address</td>
<td>✗</td>
</tr>
<tr>
<td><a href="mailto:wayne@cs.princeton.edu">wayne@cs.princeton.edu</a></td>
<td>✓</td>
</tr>
<tr>
<td>eve@airport</td>
<td>✗</td>
</tr>
<tr>
<td><a href="mailto:rs123@princeton.edu">rs123@princeton.edu</a></td>
<td>✗</td>
</tr>
</tbody>
</table>

Oops, need to fix description

Challenge. Develop a precise description of the set of strings that are legal e-mail addresses.
Pattern matching

Example 3 (from genomics)

A nucleic acid is represented by one of the letters a, c, t, or g.

A genome is a string of nucleic acids.

A Fragile X Syndrome pattern is a genome having an occurrence of gcg, followed by any number of cgg or agg triplets, followed by ctg.

Note. The number of triplets correlates with Fragile X Syndrome, a common cause of mental retardation.

Q. Does this genome contain a such a pattern?

gcgcgttgtgcgcagagagatgggttaagctg

A. Yes.
A regular expression (RE) is a notation for specifying a set of strings (a formal language).

An RE is either
- The empty set
- The empty string
- A single character or wildcard symbol
- An RE enclosed in parentheses
- The concatenation of two or more REs
- The union of two or more REs
- The closure of an RE (any number of occurrences)

<table>
<thead>
<tr>
<th>operation</th>
<th>example RE</th>
<th>matches (IN the set)</th>
<th>does not match (NOT in the set)</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>cumulus 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 abbbba</td>
<td>ab ababa</td>
</tr>
<tr>
<td>parentheses</td>
<td>a(a</td>
<td>b)aab</td>
<td>aaaaab ababa</td>
</tr>
<tr>
<td></td>
<td>(ab)*a</td>
<td>a ababababababa</td>
<td>aa abbbba</td>
</tr>
</tbody>
</table>
## More examples of regular expressions

The 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>ba<em>ba</em>)*</td>
<td>bbb</td>
</tr>
<tr>
<td>multiple of three b’s</td>
<td>aaa</td>
<td>bb</td>
</tr>
<tr>
<td></td>
<td>bbbababbaa</td>
<td>baabbbbaa</td>
</tr>
<tr>
<td>.*0....</td>
<td>1000234</td>
<td>111111111</td>
</tr>
<tr>
<td>fifth to last digit is 0</td>
<td>98701234</td>
<td>403982772</td>
</tr>
<tr>
<td>.<em>gcgcgcggctg.</em></td>
<td>gcgcgctg...</td>
<td>gcgcggcggctg...</td>
</tr>
<tr>
<td>fragile X syndrome pattern</td>
<td>gcgcgaggctg...</td>
<td>gcgcaggctg</td>
</tr>
</tbody>
</table>
### Generalized regular expressions

Additional operations further extend the utility of REs.

<table>
<thead>
<tr>
<th>operation</th>
<th>example RE</th>
<th>matches</th>
<th>does not match</th>
</tr>
</thead>
<tbody>
<tr>
<td>one or more</td>
<td>a(bc)+de</td>
<td>abcde</td>
<td>ade</td>
</tr>
<tr>
<td></td>
<td></td>
<td>abcbcde</td>
<td>bcd</td>
</tr>
<tr>
<td>character class</td>
<td>[A-Za-z][a-z]⁺</td>
<td>lowercase Capitalized</td>
<td>camelCase illegal</td>
</tr>
<tr>
<td>exactly j</td>
<td>[0–9]{5}–[0–9]{4}</td>
<td>08540–1321 19072–5541</td>
<td>111111111 166–54–1111</td>
</tr>
<tr>
<td>between j and k</td>
<td>a.{2,4}b</td>
<td>abcb</td>
<td>ab</td>
</tr>
<tr>
<td></td>
<td></td>
<td>abcbcb</td>
<td>aaaaab</td>
</tr>
<tr>
<td>negation</td>
<td>[^aeiou]{6}</td>
<td>rhythm</td>
<td>decade</td>
</tr>
<tr>
<td>whitespace</td>
<td>\s</td>
<td>any whitespace char</td>
<td>every other character</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(space, tab, newline...)</td>
<td></td>
</tr>
</tbody>
</table>

**Note.** These operations are all *shorthand.* They are very useful but not essential.

RE: \((a|b|c|d|e)(a|b|c|d|e)\)*

shorthand: (a–e)+
A \( \text{C}_2\text{H}_2 \)-type zinc finger domain signature is

- C followed by 2, 3, or 4 amino acids, followed by
- C followed by 3 amino acids, followed by
- L, I, V, M, F, Y, W, C, or X followed by 8 amino acids, followed by
- H followed by 3, 4, or 5 amino acids, followed by

Q. Give a generalized RE for all such signatures.

A. \( \text{C}\{2,4\}\text{C...}[\text{LIVMFYWXC}]\{8\}\text{H}\{3,5\}\text{H} \)

"Wildcard" matches any of the letters

CAVLMCRKHDENQSTYFWP
Example of a real-world RE application: PROSITE
Another example of describing a pattern with a generalized RE

An **e-mail address** is
- A sequence of letters, followed by
- the character "@", followed by
- the character ".", followed by a nonempty sequence of lowercase letters, followed by
- [any number of occurrences of the previous pattern]
- "edu" or "com" (others omitted for brevity).

**Q.** Give a generalized RE for e-mail addresses.

**A.** `[a-z]+@[a-z]+\.[a-z]+(edu|com)`

**Exercise.** Extend to handle `rs123@princeton.edu`, more suffixes such as `.org`, and any other extensions you can think of.

**Next.** Determining whether a given string matches a given RE.
Pop quiz 1 on REs

Q. Which of the following strings match the RE \( a^*bb(ab|ba)^* \)?

1. abb
2. aaba
3. abba
4. bbbaab
5. cbb
6. bbababbab
Pop quiz 2 on REs

Q. Give an RE for genes
   • Characters are a, c, t or g.
   • Starts with atg (a start codon).
   • Length is a multiple of 3.
   • Ends with tag, taa, or ttg (a stop codon).
Image sources

http://en.wikipedia.org/wiki/Homology_modeling#/media/File:DHR57B_homology_model.png
14. Introduction to Theoretical CS

- Overview
- Regular expressions
- DFAs
- Applications
- Limitations
Deterministic finite automata (DFA)

A DFA is an abstract machine that solves a pattern matching problem.
- A string is specified on an input tape (no limit on its length).
- The DFA reads each character on input tape once, moving left to right.
- The DFA lights "YES" if it recognizes the string, "NO" otherwise.

Each DFA defines a language (the set of strings that it recognizes).
Deterministic finite automata details and example

A **DFA** is an abstract machine with a finite number of states, each labeled Y or N, and transitions between states, each labeled with a symbol. One state is the start state.

- Begin in the start state.
- Read an input symbol and move to the indicated state.
- Repeat until the last input symbol has been read.
- Turn on the "YES" or "NO" light according to the label on the current state.

Does this DFA recognize this string? b b a a b b a b b
A **DFA** is an abstract machine with a finite number of states, each labeled Y or N, and transitions between states, each labeled with a symbol. One state is the start state.

- Begin in the start state.
- Read an input symbol and move to the indicated state.
- Repeat until the last input symbol has been read.
- Turn on the "YES" or "NO" light according to the label on the current state.

![DFA Diagram](image)

Does this DFA recognize this string?

```
bbababb
```
Simulating the operation of a DFA

```java
public class DFA {
    private int start;
    private boolean[] action;
    private ST<Character, Integer>[] next;

    public DFA(String filename) {
        // Fill in data structures */
    }

    public boolean recognizes(String input) {
        int state = start;
        for (int i = 0; i < input.length(); i++)
            state = next[state].get(input.charAt(i));
        return action[state];
    }

    public static void main(String[] args) {
        DFA dfa = new DFA(args[0]);
        while (!StdIn.isEmpty())
            { input = StdIn.readString();
              if (dfa.recognizes(input)) StdOut.println("Yes");
              else StdOut.println("No");
            }
    }
}
```

Symbol table to map characters a, b, ... to next state 0, 1, ...

<table>
<thead>
<tr>
<th>action[]</th>
<th>next[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>True</td>
<td>0</td>
</tr>
<tr>
<td>False</td>
<td>1</td>
</tr>
<tr>
<td>False</td>
<td>2</td>
</tr>
</tbody>
</table>

States:
- Start state:
  - 0

Alphabet:
- ab

Transition Table:
- a: False 1 2
- b: False 2 0

Sample run:
```
% more b3.txt
bababa
Yes
bb
No
ab abb ab bab a
Yes
```

Sample output:
```
Enter an input string ("end" to finish):
bababa
Yes
bb
No
```
Pop quiz 1 on DFAs

Q. Which of the following strings does this DFA accept?

1. Bitstrings that end in 1
2. Bitstrings with an equal number of occurrences of 01 and 10
3. Bitstrings with more 1s than 0s
4. Bitstrings with an equal number of occurrences of 0 and 1
5. Bitstrings with at least one 1
Q. Which of the following strings does this DFA accept?

1. Bitstrings with at least one 1
2. Bitstrings with an equal number of occurrences of 01 and 10
3. Bitstrings with more 1s than 0s
4. Bitstrings with an equal number of occurrences of 0 and 1
5. Bitstrings that end in 1
Kleene's theorem

Two ways to define a set of strings (language)
- Regular expressions (REs).
- Deterministic finite automata (DFAs).

Remarkable fact. DFAs and REs are equivalent.

Equivalence theorem (Kleene)
Given any RE, there exists a DFA that accepts the same set of strings.
Given any DFA, there exists an RE that matches the same set of strings.

Consequence: A way to solve the RE pattern matching problem
- Build the DFA corresponding to the given RE.
- Simulate the operation of the DFA.
Image sources

http://math.library.wisc.edu/images/skleene.gif
14. Introduction to Theoretical CS

- Overview
- Regular expressions
- DFAs
- Applications
- Limitations
GREP: a solution to the RE pattern matching problem

"GREP" (Generalized Regular Expression Pattern matcher).
- Developed by Ken Thompson, who designed and implemented Unix.
- Indispensable programming tool for decades.
- Found in most development environments, including Java.

**Practical difficulty:** The DFA might have *exponentially* many states.

A more efficient algorithm: use Nondeterministic Finite Automata (NFA)
- Build the NFA corresponding to the given RE.
- Simulate the operation of the NFA.
### REs in Java

Java's String class implements GREP.

```java
public class String {
    ...
    boolean matches(String re) {
        // does this string match the given RE?
    }
    ...
}
```

```java
String re = "C.{2,4}C...[LIVMFYWC].{8}H.{3,5}H";
String zincFinger = "CAASCGGPYACGGAAGYHAGAH";
boolean test = zincFinger.matches(re);
```

true!
Java RE client example: Validation

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

Does a given string match a given RE?
- Take RE from command line.
- Take strings from StdIn.

Applications
- Scientific research.
- Compilers and interpreters.
- Internet commerce.
- ...

% java Validate "C.{2,4}C...[LIVMFYWC]{8}H.{3,5}H"
CAASCGPYACGGAAGYHAGAH
true
CAASCGPYACGGAAGYHAGAH
false

% java Validate "[$_A-Za-z]$_[A-Za-z0-9]*"
ident123
true
123ident
false

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

C\textsubscript{2}H\textsubscript{2} type zinc finger domain
legal Java identifier
valid email address (simplified)

need quotes to "escape" the shell
Beyond matching

Java's String class contains other useful RE-related methods.
- RE search and replace
- RE delimited parsing

```java
public class String {
    ...
    String replaceAll(String re, String to) // replace all occurrences of substrings matching RE with to
    String[] split(String re) // split the string around matches of the given RE
    ...
}
```

Tricky notation (typical in string processing): \ signals "special character" so "\\" means "\" and "\s" means "\s"

Examples using the RE "\\s+" (matches one or more whitespace characters).

Replace each sequence of at least one whitespace character with a single space.
```
String s = StdIn.readLine();
s = s.replaceAll("\\s+", " ");
```

Create an array of the words in StdIn (basis for StdIn.readLineStrings() method)
```
String s = StdIn.readLine();
String[] words = s.split("\\s+"超级);
```
Way beyond matching

Java's Pattern and Matcher classes give fine control over the GREP implementation.

<table>
<thead>
<tr>
<th>public class Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
</tr>
<tr>
<td>static Pattern compile(String re)</td>
</tr>
<tr>
<td>Matcher matcher(String input)</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>public class Matcher</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
</tr>
<tr>
<td>boolean find()</td>
</tr>
<tr>
<td>String group()</td>
</tr>
<tr>
<td>String group(int k)</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

[A sophisticated interface designed for pros, but very useful for everyone.]
Java pattern matcher client example: Harvester

```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());
    }
}
```

Harvest information from input stream

- Take RE from command line.
- Take input from file or web page.
- Print all substrings matching RE.

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

% java Harvester "[a-z]+@[a-z]+\.(edu|com)" http://www.cs.princeton.edu/people/faculty
... rs@cs.princeton.edu
... wayne@cs.princeton.edu
...
```

harvest patterns from DNA

harvest email addresses from web for spam campaign.
(no email addresses on that site any more)
Applications of REs

**Pattern matching and beyond.**

- Compile a Java program.
- Scan for virus signatures.
- Crawl and index the Web.
- Process natural language.
- Access information in digital libraries.
- Search-and-replace in a word processors.
- Process NCBI and other scientific data files.
- 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.
- Automatically create Java documentation from Javadoc comments.

GREP and related facilities are built in to Java, Unix shell, PERL, Python ...

*virtually every computing environment*
Image sources

14. Introduction to Theoretical CS

- Overview
- Regular expressions
- DFAs
- Applications
- Limitations
Summary

Programmers
- Regular expressions are a powerful pattern matching tool.
- Equivalent DFA/NFA paradigm facilitates implementation.
- Combination greatly facilitates real-world string data

Theoreticians
- REs provide compact descriptions of sets of strings.
- DFAs are abstract machines with equivalent descriptive power.
- Are there languages and machines with more descriptive power?

You
- CS core principles provide useful tools that you can exploit now.
- REs and DFAs provide an introduction to theoretical CS.
Basic questions

Q. Are there sets of strings that cannot be described by any RE?
A. Yes.
   • Bitstrings with equal number of 0s and 1s (stay tuned).
   • Strings that represent legal REs.
   • Decimal strings that represent prime numbers.
   • DNA strings that are Watson-Crick complemented palindromes.
   • ...

Q. Are there sets of strings that cannot be described by any DFA?
A. Yes.
   • Bit strings with equal number of 0s and 1s (see next slide).
   • Strings that represent legal REs.
   • Decimal strings that represent prime numbers.
   • DNA strings that are Watson-Crick complemented palindromes.
   • ...

The same question, by Kleene's theorem
A limit on the power of REs and DFAs

**Proposition.** There exists a set of strings that cannot be described by any RE or DFA.

**Proof sketch.** No DFA can recognize the set of bitstrings with equal number of 0s and 1s.

- **Assume that you have such a DFA**, with $N$ states.
- It recognizes the string with $N + 1$ 0s followed by $N + 1$ 1s.
- Some state is *revisited* when scanning the 0s in that string.
- Delete the substring of 0s between visits of that state.
- DFA recognizes that string, too.
- It does not have equal number of 0s and 1s.
- **Proof by contradiction:** the assumption that such a DFA exists must be false.

**Ex. $N = 10$**

| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 0 | 3 | 5 | 9 | 8 | 7 | 5 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . |

| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 0 | 3 | 5 | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . | . |
Another basic question

Q. Are there abstract machines that are more powerful than DFAs?

A. Yes. A 1-stack DFA can recognize
   • Bitstrings with equal number of 0s and 1s.
   • Strings that represent legal REs.

Proof. [details omitted]
Yet another basic question

**Q.** Are there abstract machines that are more powerful than a 1-stack DFA?

**A.** Yes. A 2-stack DFA can recognize

- Decimal strings that represent prime numbers.
- Strings that represent legal Java programs.
- ...

[stay tuned for next lecture]
One last basic question

Q. Are there machines that are more powerful than a 2-stack DFA?
A. No! Not even a roomful of supercomputers (!!!)

[stay tuned for next lecture]

two machines with equal computational power
Image sources

https://openclipart.org/detail/211418/thenobel-programming
14. Introduction to Theoretical CS