5.4 **Regular Expressions**

- regular expressions
- REs and NFAs
- NFA simulation
- NFA construction
- applications

https://algs4.cs.princeton.edu
5.4 Regular Expressions

- regular expressions
- REs and NFAs
- NFA simulation
- NFA construction
- applications
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 intellectual disability.
- A human’s genome is a string.
- It contains triplet repeats of CGG or AGG, bracketed by GCC at the beginning and CTG at the end.
- Number of repeats is variable and is correlated to syndrome.

```
pattern       GCC(CGG|AGG)*CTG

GCGGCCGTGTGTCGAGAGAGGCTGTTAAAGCTG____GCGGAGGCCTG____GCCGGAGGCTG
```
Pattern matching: applications

Test if a string matches some pattern.

- Scan for virus signatures.
- Process natural language.
- Specify a programming language.
- Access information in digital libraries.
- Search genome using Prosite patterns.
- Validate forms (dates, email, URL, credit card).
- Filter text (spam, NetNanny, Carnivore, malware).
  ...

Parse text files.

- Compile a Java program.
- Crawl and index the Web.
- Read 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.

typically infinite

Crucial difference from substring search: entire text must match RE.

<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>A B A A B</td>
<td>A A B A A B</td>
<td>every other string</td>
</tr>
<tr>
<td>or</td>
<td>4</td>
<td>A A</td>
<td>B A A B</td>
<td>A A B A A B</td>
</tr>
<tr>
<td>closure</td>
<td>2</td>
<td>A B * A</td>
<td>A A</td>
<td>A B A B A</td>
</tr>
<tr>
<td>parentheses</td>
<td>1</td>
<td>A (A</td>
<td>B ) A A B</td>
<td>A A A A B</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(A B ) * A</td>
<td>A</td>
<td>A A</td>
</tr>
</tbody>
</table>

Parentheses are crucial due to the order of operations.
Which one of the following strings is not matched by the regular expression \((A B \mid C^* D)^*\) ?

A. A B A B A B
B. C D C C D D D D D
C. A B C C D A B
D. A B D A B C C A B D
Regular expression shortcuts

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><strong>wildcard</strong></td>
<td>.U.U.U.</td>
<td>CUMULUS JUGULUM</td>
<td>SUCCUBUS TUMULTUOUS</td>
</tr>
<tr>
<td><strong>character class</strong></td>
<td>[A-Za-z][a-z]*</td>
<td>word Capitalized</td>
<td>camelCase 4illegal</td>
</tr>
<tr>
<td><strong>one or more</strong></td>
<td>A(BC)+DE</td>
<td>ABCDE ABCBCDE</td>
<td>ADE BCDE</td>
</tr>
<tr>
<td><strong>exactly k</strong></td>
<td>[0-9]{5}-[0-9]{4}</td>
<td>08540-1321 19072-5541</td>
<td>111111111 166-54-111</td>
</tr>
</tbody>
</table>

**Note.** These operations are useful but not essential.

**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>RASPBERRY</td>
<td>SUBSPACE</td>
</tr>
<tr>
<td>(substring search)</td>
<td>CRISPBREAD</td>
<td>SUBSPECIES</td>
</tr>
<tr>
<td>[0-9]{3}-[0-9]{2}-[0-9]{4}</td>
<td>166-11-4433</td>
<td>11-55555555</td>
</tr>
<tr>
<td>(U.S. Social Security numbers)</td>
<td>166-45-1111</td>
<td>8675309</td>
</tr>
<tr>
<td>[a-z]+@[a-z]+.+.(edu</td>
<td>com)</td>
<td><a href="mailto:wayne@princeton.edu">wayne@princeton.edu</a></td>
</tr>
<tr>
<td>(simplified email addresses)</td>
<td><a href="mailto:rs@princeton.edu">rs@princeton.edu</a></td>
<td></td>
</tr>
<tr>
<td>[$_A-Za-z][$_A-Za-z0-9]*</td>
<td>ident3</td>
<td>3a</td>
</tr>
<tr>
<td>(Java identifiers)</td>
<td>PatternMatcher</td>
<td>ident#3</td>
</tr>
</tbody>
</table>

REs play a well-understood role in the theory of computation.
Perl RE for valid RFC822 email addresses

REs are not always the right tool
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

**Bottom line.** REs are amazingly powerful and expressive; using them can be amazingly complex and error-prone.
Which of the following REs match genes:

1. alphabet is \{ A, C, G, T \}
2. length is a multiple of 3
3. starts with ATG (a start codon)
4. ends with TAG or TAA or TTG (a stop codon)

A. \text{ATG}\((A|C|G|T)(A|C|G|T)(A|C|G|T))^*\((\text{TAG}|\text{TAA}|\text{TTG})\)

B. \text{ATG}\(\{\text{ACGT}\}{3}\)^*\((\text{TAG}|\text{TAA}|\text{TTG})\)

C. Both A and B.

D. Neither A nor B.
5.4 Regular Expressions

- regular expressions
- *REs and NFAs*
- NFA simulation
- NFA construction
- applications
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.

**Theorem: DFAs and REs are equivalent.**
- 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.

---

**RE**

\[
0^* \mid (0^*10^*10^*10^*10^*)^*
\]

**DFA**

Number of 1s is a multiple of 3
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.
- 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.
- Slower than DFA simulation (stay tuned).

Underlying abstraction. Non-deterministic finite state automata (NFA).

Basic plan.
- Build NFA from RE.
- Simulate NFA with text as input.

Q. What is an NFA?
Regular-expression-matching NFA.

- We assume RE enclosed in parentheses.
- One state per RE character (start = 0, accept = $m$).
- Match transition (change state and scan to next text char).
- Dashed $\varepsilon$-transition (change state, but don’t scan text).
- Accept if any sequence of transitions ends in accept state.

NFA corresponding to the pattern $( (A \ast B \mid A C) D )$
Nondeterministic finite-state automata

Q. Is A A A A B D matched by NFA?
A. Yes, because some sequence of legal transitions ends in state 11.

NFA corresponding to the pattern \(( ( A \ast B \mid A C ) D )\)
Q. Is A A A A B D matched by NFA?
A. Yes, because some sequence of legal transitions ends in state 11. [even though some sequences end in wrong state or get stuck]

NFA corresponding to the pattern ( ( A * B | A C ) D )
Nondeterministic finite-state automata

Q. Is A A A C matched by NFA?

A. No, because no sequence of legal transitions ends in state 11.
   [ must argue about all possible sequences (not just the one below) ]

A A A A A C

NFA corresponding to the pattern ( ( A * B | A C ) D )
Which of the following strings are matched by the NFA?

A. B A A A A

B. A A B A A B A A

C. Both A and B.

D. Neither A nor B.
Nondeterminism

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

DFA. Deterministic $\Rightarrow$ easy (at each step, only one applicable transition).

NFA. Nondeterministic $\Rightarrow$ hard (at each step, can be several applicable transitions; machine “guesses” the correct one!)

Q. How to simulate NFA?
A. Systematically consider all possible transition sequences.  [up next]

NFA corresponding to the pattern ( ( A * B | A C ) D )
5.4 Regular Expressions

- regular expressions
- REs and NFAs
- NFA simulation
- NFA construction
- applications
NFA representation

State names. Integers from 0 to $m$.

Match-transitions. Keep regular expression in array $re[]$.

$$
\begin{array}{cccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
\end{array}
\begin{array}{c}
( & ( & A & \cdot & B & | & A & C & ) & D & ) \\
\end{array}
$$

ɛ-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 corresponding to the pattern ( ( A * B | 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.

**Good news.** There are only $m$ possible states, so this is only a linear slowdown compared to DFA simulation.

**Q.** How to perform reachability?
**Goal.** Check whether input matches pattern.

NFA corresponding to the pattern \(( ( A \ast B \mid A \ C ) \ D )\)
Digraph reachability review

**Goal.** Find all vertices reachable from a given set of vertices.

- Recall Section 4.2

```java
public class DirectedDFS {

    DirectedDFS(Digraph G, int s) {
        find vertices reachable from s
    }

    DirectedDFS(Digraph G, Iterable<Integer> s) {
        find vertices reachable from sources
    }

    boolean marked(int v) {
        is v reachable from source(s)?
    }
}
```

**Solution.** Run DFS from each source, without unmarking vertices.

**Performance.** Runs in time proportional to $E + V$. 
public class NFA
{
    private char[] re;        // match transitions
    private Digraph G;        // epsilon transition digraph
    private int m;            // number of states

    public NFA(String regexp)
    {
        m = regexp.length();
        re = regexp.toCharArray();
        G = buildEpsilonTransitionDigraph();
    }

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

    public Digraph buildEpsilonTransitionDigraph()
    { /* stay tuned */ }
}

NFA simulation: Java implementation
NFA simulation: Java implementation

```java
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> states = new Bag<Integer>();
            for (int v : pc)
            {
                if (v == m) continue;
                if (((re[v] == txt.charAt(i)) || re[v] == '.'))  
                    states.add(v+1);
            }

            dfs = new DirectedDFS(G, states);
            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;
}
```

- States reachable from start by $\varepsilon$-transitions
- Set of states reachable after scanning past `txt.charAt(i)`
- Not necessarily a match (RE needs to match full text)
- Follow $\varepsilon$-transitions
- Accept if can end in state $m$
NFA simulation: analysis

Proposition. Determining whether an $n$-character text is recognized by the NFA corresponding to an $m$-character pattern takes time proportional to $mn$ in the worst case.

Pf. For each of the $n$ text characters, we iterate through a set of states of size no more than $m$ and run DFS on the graph of $\varepsilon$-transitions. [The NFA construction we will consider ensures the number of edges $\leq 3m$.]
5.4 Regular Expressions

- regular expressions
- REs and NFAs
- NFA simulation
- NFA construction
- applications
Building an NFA corresponding to an RE

**States.** Include a state for each symbol in the RE, plus an accept state.

NFA corresponding to the pattern ( ( A * B | A C ) D )
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.** ( ) . * |

NFA corresponding to the pattern ( ( A * B | A C ) D )
Building an NFA corresponding to an RE

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

NFA corresponding to the pattern $( ( A * B | A C ) D )$
Building an NFA corresponding to an RE

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

NFA corresponding to the pattern $(A^* B | A C) D$
Building an NFA corresponding to an RE

2-way or. Add two $\varepsilon$-transition edges for each | operator.

2-way or expression

NFA corresponding to the pattern ( ( A * B | A C ) D )
Building an NFA corresponding to an RE

**States.** Include a state for each symbol in the RE, plus an accept state.

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

**Parentheses.** Add \( \varepsilon \)-transition edge from parentheses to next state.

**Closure.** Add three \( \varepsilon \)-transition edges for each \( \ast \) operator.

**2-way or.** Add two \( \varepsilon \)-transition edges for each \( | \) operator.
How would you modify the NFA below to match \(((ABC^*)^+)\) ?

A. Remove $\epsilon$-transition edge $1 \rightarrow 7$.
B. Remove $\epsilon$-transition edge $7 \rightarrow 1$.
C. Remove $\epsilon$-transition edges $1 \rightarrow 7$ and $7 \rightarrow 1$.
D. None of the above.

NFA corresponding to the pattern \(((A B C^*)^+)\)
NFA construction: implementation

**Goal.** Write a program to build the $\varepsilon$-transition digraph.

**Challenges.** Remember left parentheses to implement closure and 2-way or; remember | symbols to implement 2-way or.

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

NFA corresponding to the pattern ( ( A * B | A C ) D )
NFA construction: analysis

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

Pf. For each of the $m$ characters in the RE, we add at most three $\varepsilon$-transitions and execute at most two stack operations.
Exercise.

- Write a regular expression that matches all strings that contain the substring ABABAC.
- Draw the NFA corresponding to the regular expression.
- Simulate the NFA on the string AABACABAABABACAA.
- Recall that the DFA for matching the pattern ABABAC required backward edges (shown below). Why does the NFA not need such edges?
5.4 Regular Expressions

- regular expressions
- REs and NFAs
- NFA simulation
- NFA construction
- applications
Industrial-strength grep implementation

To complete the implementation:

- Add multiway or.
- Handle metacharacters.
- Support character classes.
- 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>`?
Validity checking. Does the input match the regexp?

Java string library. Use `input.matches(regexp)` for basic RE matching.

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

Example usage:

```
% 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
```

- Legal Java identifier
- Valid email address
- Simplified
- Social Security number
Harvesting information

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

```
% java Harvester "GCG(CGG|AGG)*CTG" chromosomeX.txt
GCGCGCCGCGCGCGGCTG
GCGCTG
GCGCTG
GCGCGCCGCGCGGAGGGGAGGCGGCTG
```

harvest patterns from DNA

```
% java Harvester "http://(\w+\.)*(\w+)" http://www.cs.princeton.edu
http://www.w3.org
http://www.cs.princeton.edu
http://drupal.org
http://csguide.cs.princeton.edu
http://www.cs.princeton.edu
http://www.princeton.edu
```

harvest links from website
Harvesting information

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());
        }
    }
}
```

- `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()`
What is the worst-case running time of Java’s matches() method?

A. $m + n$

B. $mn$

C. $mn^2$

D. $2^n$

$m = \text{pattern length}$
$n = \text{text length}$
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 $\Rightarrow$ DFA.
- **grep** RE $\Rightarrow$ NFA.
- **javac** Java language $\Rightarrow$ Java byte code.

<table>
<thead>
<tr>
<th></th>
<th>KMP</th>
<th>grep</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>pattern</strong></td>
<td>string</td>
<td>RE</td>
<td>program</td>
</tr>
<tr>
<td><strong>parser</strong></td>
<td>unnecessary</td>
<td>check if legal</td>
<td>check if legal</td>
</tr>
<tr>
<td><strong>compiler output</strong></td>
<td>DFA</td>
<td>NFA</td>
<td>byte code</td>
</tr>
<tr>
<td><strong>simulator</strong></td>
<td>DFA simulator</td>
<td>NFA simulator</td>
<td>JVM</td>
</tr>
</tbody>
</table>
Summary of pattern-matching algorithms

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

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

You.
- Core CS principles provide useful tools that you can exploit now.
- REs and NFAs provide introduction to theory of computing.

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