Acknowledgement: The course slides are adapted from the slides prepared by R. Sedgewick and K. Wayne of Princeton University.
TODAY

- 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 mental retardation.
• Human genome contains triplet repeats of CGG or AGG, bracketed by GCG at the beginning and CTG at the end.
• Number of repeats is variable, and correlated with syndrome.

pattern  GCG (CGG | AGG) *CTG

text  GCGCGGTGTGTCGAGAGAGTGTTTAAAGCTGGCGGGAGGCGGCTGGCGCGGAGGCTG
/*
 * Compilation:  javac NFA.java
 * Execution:    java NFA regexp text
 * Dependencies: Stack.java Bag.java Digraph.java DirectedDFS.java
 * % java NFA "(A*B|AC)D" AAAABD
 * true
 * % java NFA "(A*B|AC)D" AAAAC
 * false
 */

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> ops = new Stack<Integer>();
        G = new Digraph(M+1);
    }
}
Google code search

Search public source code

Search via regular expression, e.g. ^java/.*/java$

Search Options

<table>
<thead>
<tr>
<th>Option</th>
<th>In Search Box</th>
</tr>
</thead>
<tbody>
<tr>
<td>Package</td>
<td>package:linux-2.6</td>
</tr>
<tr>
<td>Language</td>
<td>lang:c++</td>
</tr>
<tr>
<td>File Path</td>
<td>file:(code) [^or]g)search</td>
</tr>
<tr>
<td>Class</td>
<td>class:HashMap</td>
</tr>
<tr>
<td>Function</td>
<td>function:toString</td>
</tr>
<tr>
<td>License</td>
<td>license:mozilla</td>
</tr>
<tr>
<td>Case Sensitive</td>
<td>case:yes</td>
</tr>
</tbody>
</table>

http://code.google.com/p/chromium/source/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. A "language"

<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>ABABBBBBBBBA</td>
<td>ABABABA</td>
</tr>
<tr>
<td>parentheses</td>
<td>1</td>
<td>A (A</td>
<td>B) AAB</td>
<td>AAAAB ABAAB</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(AB) *A</td>
<td>A ABABABABABABA</td>
<td>AA ABBA</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>111111111 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>RASPBERRY CRISPBREAD</td>
<td>SUBSPACE SUBSPECIES</td>
</tr>
<tr>
<td>(substring search)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[0-9]{3}-[0-9]{2}-[0-9]{4}</td>
<td>166-11-4433 166-45-1111</td>
<td>11-55555555 8675309</td>
</tr>
<tr>
<td>(Social Security numbers)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[a-z]+@[a-z]+.(edu</td>
<td>com)</td>
<td><a href="mailto:wayne@princeton.edu">wayne@princeton.edu</a> <a href="mailto:rs@princeton.edu">rs@princeton.edu</a></td>
</tr>
<tr>
<td>(email addresses)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[$_A-Za-z][$_A-Za-z0-9]*</td>
<td>ident3 PatternMatcher</td>
<td>3a ident#3</td>
</tr>
<tr>
<td>(Java identifiers)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

REs plays 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

Whenever I learn a new skill, I concoct elaborate fantasy scenarios where it lets me save the day.

Oh no! The killer must have followed her on vacation!

But to find them, we'd have to search through 200MB of emails looking for something formatted like an address!

It’s hopeless!

Everybody stand back.

I know regular expressions.

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.
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.

**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.

*Stephen Kleene*  
Princeton Ph.D. 1934

**Example:**

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

**DFA:**
- Number of 1's 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. [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. Non-deterministic 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 $\varepsilon$-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.

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

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 $\text{AAAABD}$ matched by NFA?
A. Yes, because some sequence of legal transitions ends in state 11.  
[ even though some sequences end in wrong state or stall ]

NFA corresponding to the pattern $((A*B|A*C)D)$
Q. Is \texttt{AAAC} matched by NFA?

A. No, because no sequence of legal transitions ends in state 11.

[ but need to argue about all possible sequences ]
Nondeterminism

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.

NFA corresponding to the pattern ( ( A * B | A C ) D )
REs and NFAs
NFA simulation
NFA construction
Applications
**NFA representation**

**State names.** Integers from 0 to \( M \).

**Match-transitions.** Keep regular expression in array \( \text{re}[\cdot] \).

**\( \varepsilon \)-transitions.** Store in a digraph \( G \).

- \( 0 \rightarrow 1 \), \( 1 \rightarrow 2 \), \( 1 \rightarrow 6 \), \( 2 \rightarrow 3 \), \( 3 \rightarrow 2 \), \( 3 \rightarrow 4 \), \( 5 \rightarrow 8 \), \( 8 \rightarrow 9 \), \( 10 \rightarrow 11 \)

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

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

NFA simulation

NFA corresponding to the pattern \(( ( A^* B | A C ) D )\)
Read next input character.

- Find states reachable by match transitions.
- Find states reachable by $\varepsilon$-transitions

Set of states reachable from start: 0
NFA simulation

Read next input character.

- Find states reachable by match transitions.
- Find states reachable by $\varepsilon$-transitions

set of states reachable via $\varepsilon$–transitions from start
Read next input character.

- Find states reachable by match transitions.
- Find states reachable by $\varepsilon$-transitions

**NFA simulation**

**set of states reachable via $\varepsilon$-transitions from start**: \{ 0, 1, 2, 3, 4, 6 \}
NFA simulation

Read next input character.

- Find states reachable by match transitions.
- Find states reachable by $\varepsilon$-transitions

set of states reachable after matching A
Read next input character.

- Find states reachable by match transitions.
- Find states reachable by $\varepsilon$-transitions

set of states reachable after matching $A$: \{ 3, 7 \}
Read next input character.

- Find states reachable by match transitions.
- Find states reachable by $\varepsilon$-transitions

**NFA simulation**

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

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

```
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>(</td>
<td>(</td>
<td>A</td>
<td>*</td>
<td>B</td>
<td>D</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>C</td>
<td>D</td>
<td></td>
</tr>
</tbody>
</table>
```

set of states reachable after matching A A
Read next input character.

- Find states reachable by match transitions.
- Find states reachable by $\varepsilon$-transitions

set of states reachable after matching A A : \{ 3 \}
Read next input character.

- Find states reachable by match transitions.
- Find states reachable by $\varepsilon$-transitions

set of states reachable via $\varepsilon$-transitions after matching A A
Read next input character.

- Find states reachable by match transitions.
- Find states reachable by $\epsilon$-transitions

set of states reachable via $\epsilon$-transitions after matching A A : { 2, 3, 4 }
NFA simulation

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

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 $\varepsilon$-transitions

set of states reachable after matching A A B : { 5 }
Read next input character.
• Find states reachable by match transitions.
• Find states reachable by $\varepsilon$-transitions

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

Read next input character.

• Find states reachable by match transitions.
• Find states reachable by $\varepsilon$-transitions

set of states reachable via $\varepsilon$–transitions after matching A A B : \{ 5, 8, 9 \}
Read next input character.

- Find states reachable by match transitions.
- Find states reachable by $\varepsilon$-transitions

NFA simulation

set of states reachable after matching A A B D
Read next input character.

- Find states reachable by match transitions.
- Find states reachable by $\varepsilon$-transitions

set of states reachable after matching A A B D : { 10 }
Read next input character.

- Find states reachable by match transitions.
- Find states reachable by $\varepsilon$-transitions

$\varepsilon$-transitions

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

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

set of states reachable via $\varepsilon$-transitions after matching A A B D: \{ 10, 11 \}
When no more input characters:
  • Accept if any state reachable is an accept state.
  • Reject otherwise.

set of states reachable: \{10, 11\}
Digraph reachability

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

```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 = 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;
}
**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 $\leq 3M$.]

![NFA diagram for the pattern ( ( A * B | A C ) D )](image-url)
Regular Expressions

- REs and NFAs
- NFA simulation
- NFA construction
- Applications
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.** ( ) . * |
Building an NFA corresponding to an RE

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

NFA corresponding to the pattern $( ( A \ast B \mid A C ) D )$
Building an NFA corresponding to an RE

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

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

**Or.** Add two $\varepsilon$-transition edges for each | operator.

```
lp

or expression

G.addEdge(lp, or+1);
G.addEdge(or, i);

i

NFA corresponding to the pattern (( A * B I A C ) D )
```
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 corresponding to the pattern $( A \ast B | A C ) D$
NFA construction

$(((A^*B|AC)D))$
NFA construction

Left parenthesis.

• Add $\varepsilon$-transition to next state.
• Push index of state corresponding to ( onto stack.
NFA construction

Left parenthesis.

• Add $\varepsilon$-transition to next state.

• Push index of state corresponding to ( onto stack.
Alphabet symbol.

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

\[
\text{(((A*B|A|C|D))}
\]
NFA construction

Alphabet symbol.

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

$((A^*BIA C)D)$
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 $*$. 

$((A*B\mid A*C)D)$
Or symbol.

- Push index of state corresponding to '|' onto stack.
NFA construction

Alphabet symbol.

- Add match transition to next state.
- Do one-character lookahead:
  add \( \varepsilon \)-transitions if next character is \(*\).

\[( ( (A * B | A C) D ) \]
NFA construction

Alphabet symbol.

- Add match transition to next state.
- Do one-character lookahead:
  add \( \epsilon \)-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 *.

$\text{NFA construction}$

$((A \ast B \mid A \ast C) \ast D)$
NFA construction

Alphabet symbol.

- Add match transition to next state.
- Do one-character lookahead:
  - add ε-transitions if next character is *.

((A*B|A*C)D)
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

$$ ((A^*B|A\ C)D) $$
NFA construction

End of regular expression.

- Add accept state.
NFA construction

NFA corresponding to the pattern \(( ( A \star B \mid A C ) D )\)
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;
}
**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.
REGULAR EXPRESSIONS

› REs and NFAs
› NFA simulation
› NFA construction
› Applications
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 $M N$) is the same as for brute-force substring search.
Typical grep application: crossword puzzles

% grep "s..ict.." words.txt
constrictor
strict
strict

% more words.txt
a
aback
abacus
abalone
abandon
...

dictionary (standard in Unix)
also on booksite
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>`?

```plaintext
<blink>text</blink> some text <blink>more text</blink>
```

reluctant          reluctant
reluctant          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
```
do for each line
print all words that start with uppercase letter
Regular expressions in Java

Validity checking. Does the input match the regexp?

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

```java
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-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
Harvesting information

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

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

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

harvest patterns from DNA

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()`
**Algorithmic complexity attacks**

**Warning.** Typical implementations do **not** guarantee performance!

Unix grep, Java, Perl

```
% java Validate "(a|aa)*b" aaaaaaaaaaaaaaaaaaaaaaaaaaaaaac 1.6 seconds
% java Validate "(a|aa)*b" aaaaaaaaaaaaaaaaaaaaaaaaaaaaaac 3.7 seconds
% java Validate "(a|aa)*b" aaaaaaaaaaaaaaaaaaaaaaaaaaaaaac 9.7 seconds
% java Validate "(a|aa)*b" aaaaaaaaaaaaaaaaaaaaaaaaaaaaaac 23.2 seconds
% java Validate "(a|aa)*b" aaaaaaaaaaaaaaaaaaaaaaaaaaaaaac 62.2 seconds
% java Validate "(a|aa)*b" aaaaaaaaaaaaaaaaaaaaaaaaaaaaaac 161.6 seconds
```

**SpamAssassin regular expression.**

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

Some non-regular languages.
- Strings of the form \(ww\) for some string \(w\): beriberi.
- Unary strings with a composite number of 1s: 111111.
- Bitstrings with an equal number of 0s and 1s: 01110100.
- Watson-Crick complemented palindromes: atttcggaaat.

Remark. Pattern matching with back-references is intractable.
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>pattern</td>
<td>string</td>
<td>RE</td>
<td>program</td>
</tr>
<tr>
<td>parser</td>
<td>unnecessary</td>
<td>check if legal</td>
<td>check if legal</td>
</tr>
<tr>
<td>compiler output</td>
<td>DFA</td>
<td>NFA</td>
<td>byte code</td>
</tr>
<tr>
<td>simulator</td>
<td>DFA simulator</td>
<td>NFA simulator</td>
<td>JVM</td>
</tr>
</tbody>
</table>
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.