• Homework 9 due Friday at 2pm
• Final Exam:
  • Friday, May 13 from 5:10pm to 7:00pm in WWH 101
  • Must bring a Laptop! (Email me asap if you do not have one!)
Exercise: Give a DFA for $\Sigma = \{a, b\}$ that accepts any string with aababb as a substring
Exercise: Give a DFA for $\Sigma = \{a, b\}$ that accepts any string with $aababb$ as a substring.
Exercise: Suppose you are given many texts (strings) $T_1, \ldots, T_n$ and one pattern string $P$. You want to determine which texts have the pattern $P$ as a substring. What is the total runtime of doing this using DFAs?
• String matching with a DFA is very fast
• How do you build the DFA?
• Knuth-Morris-Pratt
  • Quickly builds the DFA
  • Compactly stores the DFA
  • Simulates the DFA
• Given a pattern P and a string T
  • The array can be built in $O(|P|)$ time
  • The string can be matched in $O(|T|)$ time
• General idea: build an array where each entry corresponds to a DFA state
• Recall from previous exercises that each state of the DFA corresponds to the “progress” we’ve made in pattern matching
  • Suppose each entry in the array corresponds to a prefix $z$ of the pattern $P$
• When we get a letter that doesn’t make progress, which state do we return to?
• The array entry will refer to the index of the longest pattern prefix which is a proper suffix of $z$
**Example: For pattern ABABAC**

<table>
<thead>
<tr>
<th>Index</th>
<th>Prefix z</th>
<th>Longest proper suffix that is also a prefix of z</th>
<th>Refers to…</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>⊖</td>
<td>⊖</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>A</td>
<td>⊖</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>AB</td>
<td>⊖</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>ABA</td>
<td>A</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>ABAB</td>
<td>AB</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>ABABA</td>
<td>ABA</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>ABABAC</td>
<td>⊖</td>
<td>0</td>
</tr>
</tbody>
</table>
public static int[] buildKMPTable(String pattern) {
    int[] table = new int[pattern.length()+1];
    for (int i = 2; i < table.length; ++i) {
        int j = table[i-1];
        while (true) {
            if (pattern.charAt(j) == pattern.charAt(i-1)) { table[i] = j+1; break; }
            else if (j == 0) break;
            else j = table[j];
        }
    }
    return table;
}

Exercise: build the KMP array for aababb (from previous exercise)
public static int simulate(int[] table, String text, String pattern)
{
    int state = 0;
    for (int i = 0; i < text.length(); ++i)
    {
        while (true)
        {
            if (text.charAt(i) == pattern.charAt(state)) { state++; break; }
            else if (state == 0) break;
            state = table[state];
        }
        if (state == table.length -1) break;
    }
    return state;
}
**Exercise**: Show how to find all occurrences of a pattern P in a text T using the KMP algorithm

**Exercise**: Given a string z find the shortest string x such that z = x^k for some k > 0. That is, z is x repeated k times.
• Finite automata are a graph-based way to represent patterns
  • Deterministic finite automata is convertible in a simple way into a program for pattern matching
  • Non-deterministic finite automata can be converted to deterministic finite automata using subset construction

• Regular expressions are an algebra for expressing the same kind of patterns that can be described by automata
  • Regular expressions can be converted to automata and vice versa
• The automata we have discussed so far are all deterministic
  • From each state $s$ and input character $x$, there is exactly one transition out of $s$ with label $x$
• **Non-deterministic finite automata** are allowed to have two or more transitions containing the same symbol out of one state
  • Not directly translatable to programs (unlike DFAs)
  • Useful conceptually
  • A NFA accepts a string if any of its paths lead to an accepted state
• In order to convert a NFA into a program, you can first convert the NFA to a DFA using **subset construction**
  • Intuition: each state in the DFA represents a set of states in the NFA
• **Example**: convert the following NFA to a DFA:

![NFA Diagram]

- Begin with new state in DFA \{0\}, representing the start state of the NFA
- Determine the transitions from state \{0\}:
  - For \(\Lambda\)-m, it goes to state 0 (same as state \{0\} in DFA)
  - For m, it goes to both 0 and 1, so create a new state in DFA representing \{0, 1\}
Consider transitions from state \{0, 1\}

- For $\Lambda$-m-a, it goes to state \{0\}
- On m, state 1 goes nowhere, but state 0 goes to 0 and 1, so we are back at \{0, 1\}
- On a, state 0 goes to itself, and state 1 goes to 2, so we need a new state \{0, 2\}
Consider transitions from state \{0, 2\}
- For \(\Lambda - m - n\), state 0 goes to 0, and state 2 goes nowhere (so back to \{0\})
- On \(m\), state 2 goes nowhere, but state 0 goes to both 0 and 1 (so transition to \{0, 1\})
- On \(n\), state 0 goes to 0, and state 2 goes to 3 (so transition to new state \{0, 3\})
Consider transitions from state \{0, 3\}
  - Notice that state 3 goes nowhere on any input, so we only need to worry about state 0
  - For \(\Lambda-\text{m}\), state 0 goes to 0 (so back to \{0\})
  - On \text{m}, state 0 goes to both 0 and 1 (so transition to \{0, 1\})

No more new states
• Regular expressions are an algebra to define patterns, with operators and operands
• Operands:
  • Character, $\varepsilon$ (empty string), $\varnothing$, variable whose value is any pattern defined by a regex
• Operators (in order of precedence)
  • (Kleene) closure $\ast$ (highest), concatenation, union $|$ (lowest)
  • Closure: $R^* = \varepsilon | R | RR | RRR \ldots$
• Unix extensions:
  • Character classes, e.g., [a-zA-Z]
  • Line beginning $^\wedge$ / ending $\$$
Exercise: Suppose you are grepping through `/usr/share/dict/words`. Write regular expressions for the finding following:

1. All words that end in “dous”
2. All words that have only one vowel
3. All words that have alternating consonants and vowels
Competitive Programming 5.8, 6.1—6.4

More on Automata and Regexes:

Rob Pike’s implementation of a regex parser from The Practice of Programming:
http://www.cs.princeton.edu/courses/archive/spr09/cos333/beautiful.html