A. Pattern Matching with Regular Expressions using Recursive / Backtracking Search

- Concentrate on basic regular expressions built using $\varepsilon$, $\Sigma$, concatenation, $|$, $($...$)$, and Kleene star.
- Matching problem: Does a given string of symbols $w$ from $\Sigma$ conform to a given regular expression?
  - Is $w \in L(R)$ where $R$ is a regular expression?
- Easiest part of problem: Matching individual symbols concatenated together
  - Does $w = a_1 a_2 a_3 \ldots$ match regular expression $b_1, b_2, b_3 \ldots$? Check $a_1 = b_1$, $a_2 = b_2$, etc.
  - Given an r.e. $R = a_1 R_1$ (character $a$ followed by rest of expression) and string $w = b v$, verify that $a = b$ and continue search with $v$ and $R_1$. If $a \neq b$, the search fails.
  - To keep track of where we are in the search, let’s maintain the string of characters we’ve approved so far and the string of remaining characters. (Their concatenation should always match the original starting string.)
  - So to match $a_1 R_1$ with input $b v$ (and approved string $x$), check for $a_1 = b$. If they match, then continue matching with $R_1$ and $v$ and approved $x a$. If they don’t match, fail.
- For a general concatenation of expressions $R_1 R_2$ and input $w$, the process is similar except that instead of matching the first symbol of $w$ against a symbol for $R_1$, we recursively look for a match with all of $R_1$ and, if that succeeds, then continue the match with $R_2$.
  - Match $R_1$ against $w$. If it fails, we fail.
    - If it succeeds with some suffix $w'$ remaining from $w$ and approved prefix $x'$ from $w$, then match $R_2$ against $v$. If that fails, then we fail.
      - If it succeeds with some tail $w''$ and approved $x''$, then we succeed with $w''$ leftover and approved string $x' x''$.
  - For an alternation $R_1 | R_2$, the process is easier than concatenation: Search with $R_1$ and $w$; if it succeeds, we succeed; if it fails, then search with $R_2$ with $w$ (the same $w$). We succeed iff that succeeds.
- If $R$ is the empty string symbol $\varepsilon$, we succeed with $w$ unchanged and the empty string as the approved prefix.
- There’s a subtlety with the Kleene star. $R^*$ is like an infinite alternation $\varepsilon | R | R R | R R R | \ldots$. To "loop" through the sequence of $R$’s, we can just match against $R (R^*)$. So essentially we can treat $R^* = \varepsilon | R R^*$.
  - The subtle problem is that $\varepsilon$ matches any string, so with $\varepsilon | R R^*$, we’ll always succeed with just $\varepsilon$ and skip $R R^*$.
  - If we look at a concrete example, like match $a^*$ against $a a a a b$, what we probably want is to look for as many $a$’s as possible, so we should succeed, approving $a a a a$ and leaving $b$.
  - The general question is "When matching $R$ against $w$, if more than one prefix of $w$ matches, then which prefix should we approve?"
• For $R^*$, it seems like we want the longest prefix. It’s actually easy to guarantee this: Instead of treating $R^*$ like $\varepsilon \mid R R^*$, treat it like $R R^* \mid \varepsilon$. Recursively, we’ll try to match $R R^*$ which will in effect start by trying to search $R R R^*$, which will try $R R R R^*$ and so on.

• For our example, $a^*$ against $aaaab$, first we’ll match $a$ against $aaaab$, succeed with approved $a$ and leftover $aaa$, then $a'$ against $aaa$ first matches $a$ against $aab$, which succeeds approving $a$ and leaving $aab$, and so on. The recursion has to stop because eventually we reach $b$.

• Activity question: The "Treat $R^*$ like $R R^* \mid \varepsilon$" backtracking algorithm relies on an assumption about what happens when we match the $R$ part of $R R^*$ against the string. What is the assumption, and what happens if the assumption is not met?

• Activity question: This backtracking algorithm does not always return the longest matching prefix. Give an example of when this happens (i.e., give an $R$ and $w$) and discuss how to modify the algorithm so that it always returns the longest matching prefix.

•-------------------------------------------------------------------------------- ended 2019-09-16

B. Finite State Machines

• One way to implement searches using regular expressions uses Finite State Machines (FSMs).

A finite state machine has a state; it processes a string of symbols one-by-one with each state/symbol combination leading to another state. The set of state values is fixed and finite and so is the set of state/symbol-to-state transitions

• Recognizer: The simplest kind of FSM is a recognizer: It only produces 1 bit of output ("Yes" or "No"), after all the input is seen. It accepts the input if it says "Yes," and rejects the input if it says "No."

• As an example, you can build an FSM that takes a sequence of 0’s and 1’s and accepts strings that include exactly two occurrences of 1.

• To specify a recognizer FSM you fix
  - The state set $\Sigma$ (the finite set of all possible states).
  - The start state (the one the FSM begins execution in).
  - The set of accepting states (when the FSM ends computation, it accepts the input iff it ends in an accepting state).
  - A transition function that describes how the FSM executes. It takes an input character and the current state and produces the next state for the machine (which may or may not be the same as the current state).

• FSM Execution:
  Initialize state $\leftarrow$ start state;
  while there exists more input \{
  read character of input
  set state to new state
  i.e., state $\leftarrow$ transition_function(input_char, state)
if state ∈ Accepting States then
    output “Accept”
else
    output “Reject”

**FSM Example:**

- Machine $M_1$ reads strings of a’s and b’s and accepts a string iff it contains exactly two occurrences
  of b.
- We’ll have four states $\Sigma = \{S_0, S_1, S_2, S_3\}$.
  - For $k=0, 1, \text{ or } 2$, we’ll be in state $S_k$ iff we’ve seen exactly $k$ b’s.
  - We’ll be in state $S_3$ if we’ve seen three or more b’s.
  - State $S_0$ is the initial state, state $S_2$ is the (only) accepting state.
- The transitions:
  - For $k \in \{0, 1, 2\}$, if we’re in state $S_k$ and the input is b, then we go to state $S_{k+1}$.
  - If we’re in state $S_3$ and the input is b, we stay in $S_3$.
  - For every state, if we see an a, we stay in that state.
- Sample execution:
  - One way to describe an execution of an FSM uses two rows.
  - The top row contains the symbols of the input.
  - The bottom row contains the states that the FSM is in as it reads the input.
  - We stagger the columns for visibility
  - Here’s a run with input abaabaa. Since we end in state $S_2$, we accept the input.
  - The leftmost column includes the start state; the rightmost column is the final state of the
  machine (we accept iff it is an accepting state).

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>a</th>
<th>a</th>
<th>b</th>
<th>a</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_0$</td>
<td>$S_0$</td>
<td>$S_1$</td>
<td>$S_1$</td>
<td>$S_1$</td>
<td>$S_2$</td>
<td>$S_2$</td>
<td>$S_2$</td>
</tr>
</tbody>
</table>

- (Quick question: What strings would we accept if we change the accepting state from $S_2$ to $S_3$? What
if we make every state accepting except for $S_2$?)

**C. Representing Transition Functions**

- **State Transition Table:** One way to describe a finite state machine’s transition function uses a table with three columns: The first two columns list a state and input symbol. The third column lists the state we go to if we see the given state and input symbol.
State Transition Diagram: The other technique for representing an FSM’s state transition function uses a directed graph, with a circular node for each state and an arc for each transition. The transition arc goes from the old state to the new state and is labeled with the input symbol that causes this transition.

- A tail-less arrow indicates the start state; a node gets drawn as a doubled (concentric) circle if it’s an accepting state.
- (By the way, it’s okay for the initial state to be an accepting state, though it doesn’t have to be.)

Example 1: To the right is a transition table for a the machine $M_1$.

Example 2: This is the same machine $M_1$ above, specified using a state transition diagram.

Example 3: This FSM reads strings of 0’s and 1’s and accepts iff the string contained only 0’s.
D. **Mealy Machines — A Finite State Machine With Output**

- In addition or instead of accepting its input, a **FSM with output** generates a string of symbols as it processes its input.

- A **Mealy machine** associates an output with each transition. When using a transition table, we specify the outputs by adding them as another column in the table. When using a transition diagram, we add a slash and output symbol to each arc of the graph. When showing a trace of execution, we can add a row of outputs above the inputs.

- **Example 5**: Let machine $M_2$ extend $M_1$: Output a 1 for the two transitions that lead to $S_2$; output a 0 for the other transitions. Augmenting the transition table gives:

<table>
<thead>
<tr>
<th>State</th>
<th>Input</th>
<th>New State</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_0$</td>
<td>a</td>
<td>$S_0$</td>
<td>0</td>
</tr>
<tr>
<td>$S_0$</td>
<td>b</td>
<td>$S_1$</td>
<td>0</td>
</tr>
<tr>
<td>$S_1$</td>
<td>a</td>
<td>$S_1$</td>
<td>0</td>
</tr>
<tr>
<td>$S_1$</td>
<td>b</td>
<td>$S_2$</td>
<td>1</td>
</tr>
<tr>
<td>$S_2$</td>
<td>a</td>
<td>$S_2$</td>
<td>1</td>
</tr>
<tr>
<td>$S_2$</td>
<td>b</td>
<td>$S_3$</td>
<td>0</td>
</tr>
<tr>
<td>$S_3$</td>
<td>a</td>
<td>$S_3$</td>
<td>0</td>
</tr>
<tr>
<td>$S_3$</td>
<td>b</td>
<td>$S_3$</td>
<td>0</td>
</tr>
</tbody>
</table>

Start state: $S_0$; Accepting state: $S_2$.

- Augmenting the transition diagram gives:

```
S0  a/0  b/0
    ↓  ↓  ↓  ↓
S1  a/0  b/1
    ↓  ↓  ↓  ↓
S2  a/1  b/0
    ↓  ↓  ↓  ↓
S3  a,b/0
```

- For input `abababab` we get output `0000111`. Since we produce an output with each transition, this output is one symbol shorter than the output for Example 4.

```
0  0  0  0  1  1  1
a  b  a  a  b  a  a

S0  S0  S1  S1  S1  S2  S2  S2
```

- A recognizer FSM can be viewed as a Mealy machine that produces output 1 if it's in an accepting state and 0 if it's not.
E. Echoing/Remembering the Input

• Where a Mealy machine produces output according the transitions it follows, a Moore machine produces output according to what state it is in. A Moore machine that produces "Yes" for each accepting state and "No" for each rejecting state is like a recognizer machine altered to produce a trace of the state acceptance properties as it executes.

• One handy Mealy machine echoes the input symbol with each transition — the output is the string of symbols it processes. If we modify the machine to buffer the output as it's built, then on termination, the buffer contains the input string that was accepted.

F. Nondeterministic Finite Automata (NFA)

• There are two kinds of finite state machines that differ according to whether there is always one unique next state to go to (a deterministic automaton) or where there may be a choice as to which state to go to (a nondeterministic automaton).

• [added 9/23] In a Deterministic Finite State Automaton (DFA), there is never a choice: The current state and input symbol lead to exactly one new state. In a state transition diagram for a DFA, every state has exactly one outgoing arrow for each symbol in the alphabet $\Sigma$. In a state transition table, every row/column entry contains exactly one state.

• In a Nondeterministic Finite State Automaton (NFA), there are two kinds of choices.
  
  • Two transitions leaving the same state labeled with the same character. The transition arcs have to go to different target states because we don’t allow multi-edges (two edges that are exactly the same).
  
  • An $\epsilon$-transition, which is an arc labeled by the empty string. We have the choice of remaining in the current state or following the $\epsilon$-transition to its target. (For a state with multiple $\epsilon$-transitions, we can choose to follow any of them or stay in the current state.) Following an $\epsilon$-transition doesn’t consume any of the input.

• A general NFA also allows a state to have no arc labeled with a particular character.
  
  • E.g., you might have a node with out-arrows labeled $a$ and $b$ but not $c$.
  
  • If you’re in that state and see character $c$, the matching fails. (People sometimes say you get "stuck" and can’t proceed.) If this is the only possible state you could be in, then execution halts and you reject the input even though you have only seen part of it.

• We can think of execution of an NFA in two basic ways:
  
  • Backtracking search
    
    • When a path through the automaton can’t be followed any further (because we get stuck), we backtrack to the most recent nondeterministic choice and try an alternative choice.
    
    • At the end of the input, if any of the paths we try lead to an accepting state, then the string is accepted, regardless of what happens along other possible paths.
    
    • To reject a string, we must try all possible paths of execution, and each path must either get stuck or end in a non-accepting state.

  • Being “in” multiple states simultaneously.
• During execution, we maintain the set of all the possible states we could be in given the previous nondeterministic choices. You can think of us being in one of those states (but we don't know which), or you can think of us as being in all of them simultaneously.

• To transition from the current set of states to the next state, we have to look at all states in our current set, find all the relevant transitions from them, and collect the target states of the transitions to form the set of all possible next states. E.g., if from state $S_1$ we have a nondeterministic choice of going to state $S_2$ or $S_3$, then both states go into the set of possible next states.

• When input ends, we accept if any of the states in the current state are accepting and reject only if all the current possible states are rejecting.

• To implement this kind of execution, one way is to maintain a vector listing the set of all possible states.

• Since this slows down execution, it's possible to use preprocessing to convert an NFA to an equivalent DFA. (We'll see this algorithm in the next lecture.)
Activity Problems, Lecture 7

Problems

1. Consider the state diagram above for a finite state machine.
   a. What is the start state? The accepting state(s)?
   b. Trace the execution of this machine on the input 0100010.
   c. What is the pattern of strings accepted by this machine?
   d. Complete the state transition table for this machine.

<table>
<thead>
<tr>
<th>State</th>
<th>Input</th>
<th>New State</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>None</td>
<td>1</td>
<td>None</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>00</td>
</tr>
<tr>
<td>00</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Acc</td>
<td></td>
<td>0, 1</td>
</tr>
</tbody>
</table>

2. Design a finite state machine that takes an input string of 0’s and 1’s and has two states E and O (for even and odd), which it uses to keep track of whether the input has an even or odd number of 0’s. It should accept strings with even numbers of 0’s.
   a. Draw a state diagram for the machine.
   b. Give a state transition table for the machine.
   c. Give a regular expression for this language.

   (A side note: It’s easy to get a machine or reg expr for the language that has an even number of 1’s (just flip the 0 & 1 transitions). If we want a machine that accepts strings that meet both criteria (even number of 0’s and even number of 1’s), it’s straightforward to build a new machine that combines the old machines, but finding a reg expr for the intersection language is definitely more complicated.)

3. Design a finite state machine that takes an input string of a’s and b’s and accepts iff every a is immediately followed by at least one b. E.g., bbabbab should be accepted but aab and baab shouldn’t (because of their first a’s) and ababa shouldn’t (because of the final a).
   a. Give a state transition diagram for this machine. Hint: You’ll need 3 states, one for “ready to see an a”, one for “just saw an a”, and one for “reject this string”.
   b. Give a state transition table for the machine.
   c. Give a trace of execution for bbabbab.
d. Give a regular expression for the language accepted by this machine.

4. Take your machine from Problem 3 [10/2] and form a new machine by swapping a’s for b’s in all the transitions. The result will be a machine that accepts an input iff every b in it is immediately followed by at least one a. Now form a third machine that has a brand-new start state with ε-transitions from it to the start states of the first two machines. The result will be a machine that accepts any string that matches either criteria of all a’s followed by ≥ b, or all b’s followed by ≥ a. (That is, the language of this new machine is the union of the languages for the first two machines.)

Try executing this NFA on various inputs using the two different styles of executing an NFA (backtracking or current set of states). At least try an input accepted by machine 1 and one accepted by machine 2. (Trick question: Is there a string that would be accepted by both machines?)

5. Discuss briefly: Should Mealy and Moore machines be deterministic? In other words, how do you imagine executing a nondeterministic Mealy or Moore machine and is this more complicated than you think we should put up with?
Solutions to Selected Activity 7 Problems

1a. (Execution trace):

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>0</td>
<td>None</td>
<td>0</td>
<td>00</td>
<td>00</td>
<td>Acc</td>
<td>Acc</td>
</tr>
</tbody>
</table>

1b. It accepts any string that contains a 001.


<table>
<thead>
<tr>
<th>State</th>
<th>Input</th>
<th>New State</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>None</td>
<td>1</td>
<td>None</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>None</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>Acc</td>
</tr>
<tr>
<td>Acc</td>
<td>0</td>
<td>Acc</td>
</tr>
<tr>
<td>Acc</td>
<td>1</td>
<td>Acc</td>
</tr>
</tbody>
</table>

2. (Even/odd numbers of 0’s)^

2a. Note that since we’re tracking only the 0’s, the 1’s don’t change the state. Each 0 changes the state from even to odd (or vice versa).

2b. The initial and accepting state is E.

2c. One way to get a regular expression is to ask for all the general ways we can get to the accepting state from the start state. A trip from E to O and back looks like 0 1* 0; a trip from E back to E looks like 1. A single trips is one or the other, which gives (1 | 0 1* 0), and we can take as many trips as we like, which gives (1 | 0 1* 0)*. (This is not the only reg expr that’s possible.)

---

* Ignore the red color of the state machine.
3. (Look for each a is followed by ≥ one b) The three states are $E = \text{“Ready to see an a”}$, $J = \text{“Just saw an a”}$, and $X = \text{“Reject this string”}$. '

<table>
<thead>
<tr>
<th>State</th>
<th>Input</th>
<th>New State</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E$</td>
<td>a</td>
<td>$J$</td>
</tr>
<tr>
<td>$E$</td>
<td>b</td>
<td>$E$</td>
</tr>
<tr>
<td>$J$</td>
<td>a</td>
<td>$X$</td>
</tr>
<tr>
<td>$J$</td>
<td>b</td>
<td>$E$</td>
</tr>
<tr>
<td>$X$</td>
<td>a</td>
<td>$X$</td>
</tr>
<tr>
<td>$X$</td>
<td>b</td>
<td>$X$</td>
</tr>
</tbody>
</table>

[10/2: Transition graph was missing $E \rightarrow_a J$ arrow]

d. $b^*(ab^+)^*$ is one regular expression for this language.

' Ignore the red color of the state machine.