CSE396 Spring 2021 First Lecture: Turing Machines (and Syllabus Overview)

Thoughts entering the Spring 2021 term? (Notice the word "epidemic" at bottom left.)

[Lecture showed the same Groundhog Day pic as in the first course lecture on Feb. 2. If it had snowed this morning as forecast---with an inch or two accumulating---then it would have been an even more perfect setup for the April Fool's shtick of pretending it's the first lecture of term.]

Some remarks relevant to multiple aspects of the course, including Academic Integrity:

- I paid $\$11$ for license from CartoonStock to use the groundhog picture in the classroom. (Web publishing would have been $55, print publishing $50---what does that say?)
- Whereas, the author of the following blog post reproduced not only UK currency but also a UK passport design without permission and faces extradition to the UK and millions of dollars in fines (worse, in pounds sterling). https://rjlipton.wpcomstaging.com/2021/04/01/computer-science-gets-noted/
- Probabilistic automata are not on our syllabus or in the text, nor even covered in CSE596. But the course will begin in Chapter 3 with Turing Machines, of which the automata in chapters 1 and 2 are special cases.

Alas, the pandemic is affecting a second Spring term, "Groundhog Year" one could say.
What will be the same, and what different?

[The above was the April Fool's Joke---which did, however, play into the real lecture material.]

Why Turing Machines?

We saw that DFAs M, nor even NFAs nor GNFAs, cannot recognize simple languages like $\left\{a^m\, b^n : \, m = n\right\}$. How can we augment the DFA *model* to give it the needed capability?

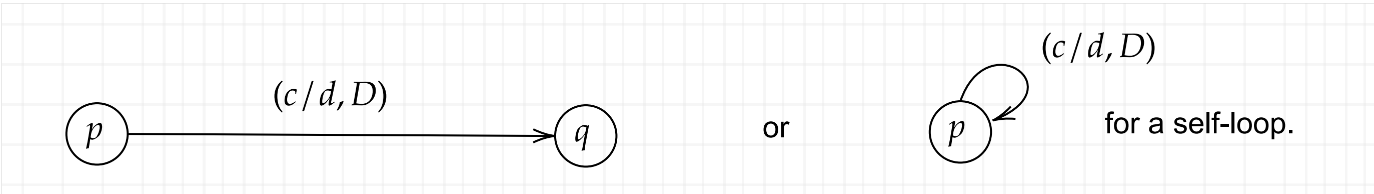
1. Allow M to change a character it reads, storing it on its tape.
2. Allow M to move its scanner left L as well as right R (or keep it stationary S).

Capability 1 by itself changes nothing: the DFA would still have to move R past the changed character. Capability 2 by itself also does not allow recognizing any nonregular languages. The proof, that every "two-way DFA" can be simulated by a simple 1-way DFA, is beyond our scope and involves another "exponential explosion" but we will cite it later to say that the class of regular languages equals "constant space" on a Turing machine.

But if we give both capabilities together, then we can do it---and lots more besides. The capabilities add two components to instructions in δ, making them 5-tuples:

$$(p, c/d, D, q) \quad \text{where } p \text{ and } q \text{ are states, } c \text{ and } d \text{ are chars, and } D \in \{\mathrm{L}, \mathrm{R}, \mathrm{S}\}$$

The meaning is that if M is in state p and scans character c, then it can change it to d, move its scanning head one position left, right, or keep it stationary, and finally transit to state q. The case (p, c, c, R, q) is the same as an ordinary FA instruction (p, c, q) where moving right is automatic. I tend to like to write a slash for the second comma to emphasize that p, c are read and d, D, q are actions taken; it also visually suggests c being changed to d. Graphically the instruction looks like:



We also regard the blank as an explicit character. I will represent it as _ in MathCha but in full LaTeX you can get "\text{\textvisiblespace}" which turns up the corners to look like more than just an underscore. My other notes call the blank B. The blank belongs not to the *input alphabet* Σ but to the work alphabet Γ (capital Gamma) which always includes Σ too. We allow going past the right end of the input string $x \in \Sigma^*$ where successive *tape cells* each initially hold the blank. We *can* also allow moving leftward of the first char of x where there are likewise blanks on a "two-way infinite tape", *or* we can stipulate that x is initially left-justified on a "one-way infinite tape" and consider any left move from the first cell to be a "crash." The *Turing Kit* package shows a two-way infinite tape and this is the default. A compromise is to use a one-way infinite tape but place a special left-endmarker char $\wedge$ in cell 0 with x occupying cells $1, \dots, n$ where $n = |x|$. If $x = \epsilon$ then the whole tape is initially blank except in the last case it has just $\wedge$ in cell 0. Then $\wedge$, as well as _, belongs to Γ but not to Σ. We will be free to put any other characters we want into Γ, but the blank (and $\wedge$ if used) are required. With all that said, the definition is crisp:

Definition: A *Turing machine* is a 7-tuple $M = (Q, \Sigma, \Gamma, \delta, \_, s, F)$ where Q, s, F and Σ are as with a DFA, the *work alphabet* Γ includes Σ and the *blank* _, and

$$\delta \subseteq (Q \times \Gamma) \times (\Gamma \times \{\mathsf{L}, \mathsf{R}, \mathsf{S}\} \times Q) .$$

It is *deterministic* (a DTM) if no two instructions share the same first two components. A DTM is "in normal form" if F consists of one state q_{acc} and there is only one other state q_{rej} in which it can halt, so that δ is a function from $(Q \setminus \{q_{acc}, q_{rej}\}) \times \Gamma$ to $(\Gamma \times \{\mathsf{L}, \mathsf{R}, \mathsf{S}\} \times Q)$. The notation then becomes $M = (Q, \Sigma, \Gamma, \delta, \_, s, q_{acc}, q_{rej})$.

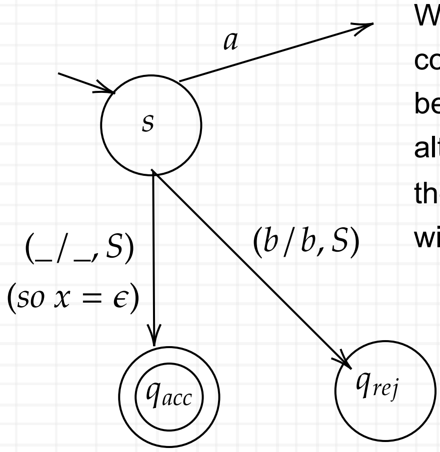
[Show the "3n+1 Game" Turing Machine as an unsolved problem about programs in general.]

To define the language $L(M)$ formally, especially when M is properly nondeterministic (an NTM), requires defining *configurations* (also called *ID*s for *instantaneous descriptions*) and *computations,* but especially with DTMs we can use the informal understanding that $L(M)$ is the set of input strings that cause M to end up in q_{acc}, while seeing some examples first.

1. $L_1 = \left\{a^m\, b^n : n = m\right\}$, by default $\epsilon \in L_1$ since $n = m = 0$ is allowed.
2. $L_2 = \left\{a^m\, b^n : n > m\right\}$. [Show this example on the Turing Kit, as "MarEx94a.tmt".]
3. $L_3 = \left\{a^m\, b^n\, a^m\, b^n : m, n \geq 0\right\}$. [Not a CFL, but conceptually not much more difficult for a Turing machine than L_1.]
4. $L_4 = \left\{ww : w \in \{a, b\}^*\right\}$. [Review how CFL Pumping Lemma proof works for both this and L_3 at the same time. Restrict $m, n \geq 1$. Show a two-tape TM for this if time allows.]

[The 4/1 lecture did L_2 but did not get to L_3 / L_4. So the Tue. 4/6 lecture will start here.]

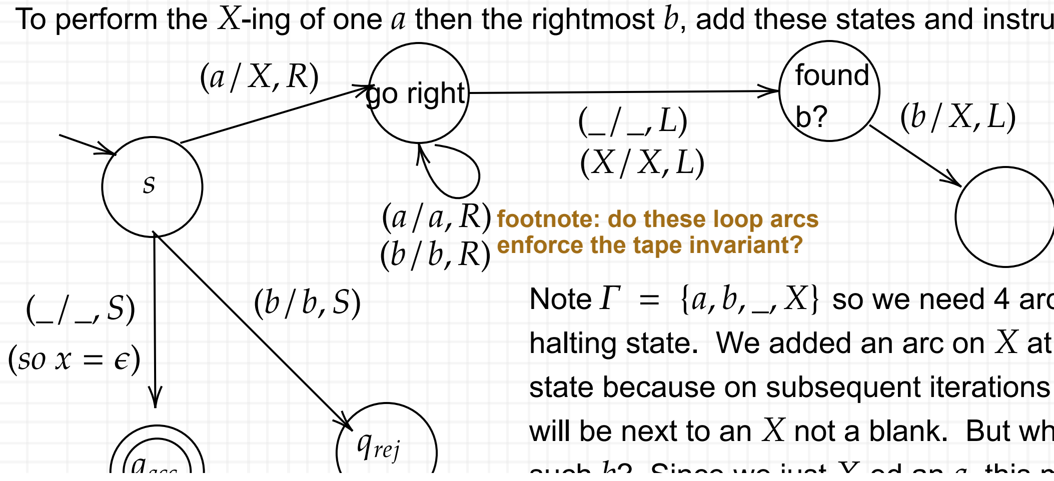
By default, n, m are natural numbers, so $n = m = 0$ is allowed, and so $\epsilon \in L_1$. When the input x is ϵ, the TM tape starts off completely blank. Otherwise, the TM starts in the configuration of scanning the first char of x, with the rest of the tape blank. So an initial scan of _ means that $x = \epsilon$ and we can make M accept right away. And if x starts with b then it cannot be in L, so we can make M reject right away. A Turing machine is not required to scan its entire input, though we can impose this requirement (and when we discuss time complexity classes, we will). This gives us a good beginning on how to build M to recognize L_1 step-by-step with goal-oriented reasoning. [Lecture might work on the diagram "interactively"; here we show some stages.]



We've already been able to handle immediate accept and reject conditions in the start state. Now we decide strategy when x begins with a. The idea is to X-out a's and b's one-by-one in alternation. If we X-out always the leftmost a and the rightmost b then the string between (which after the first iteration is $a^{m-1}b^{n-1}$) will belong to L if and only if x does. So we can recurse and keep:

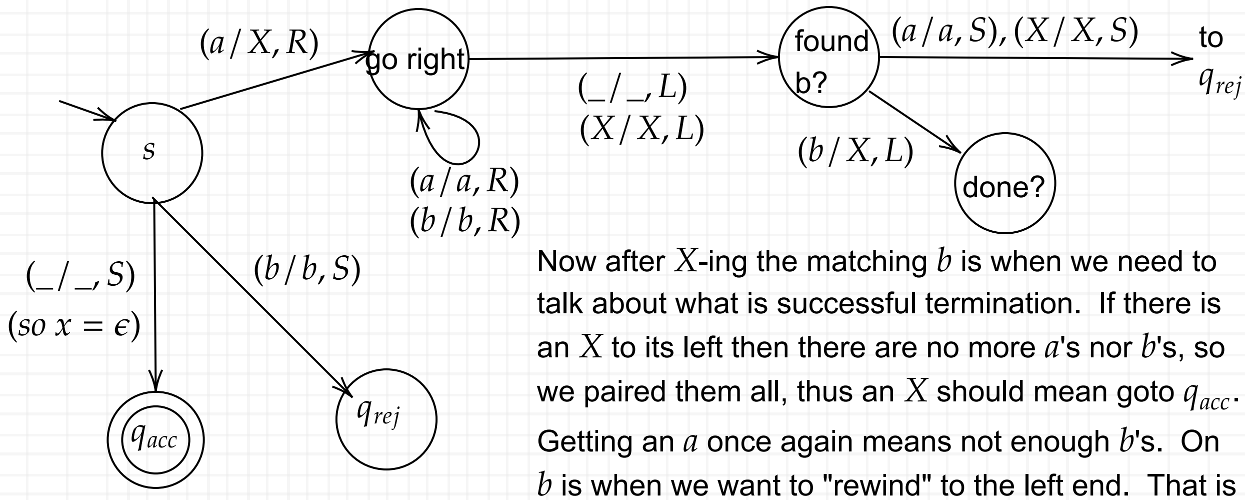
> **Tape Invariant**: $X^*\ a^*\ b^*\ X^*$ and after X-ing a b the numbers of Xes on left and right are the same, so the string between them belongs to L if and only if the original x does.

To perform the X-ing of one a then the rightmost b, add these states and instructions:

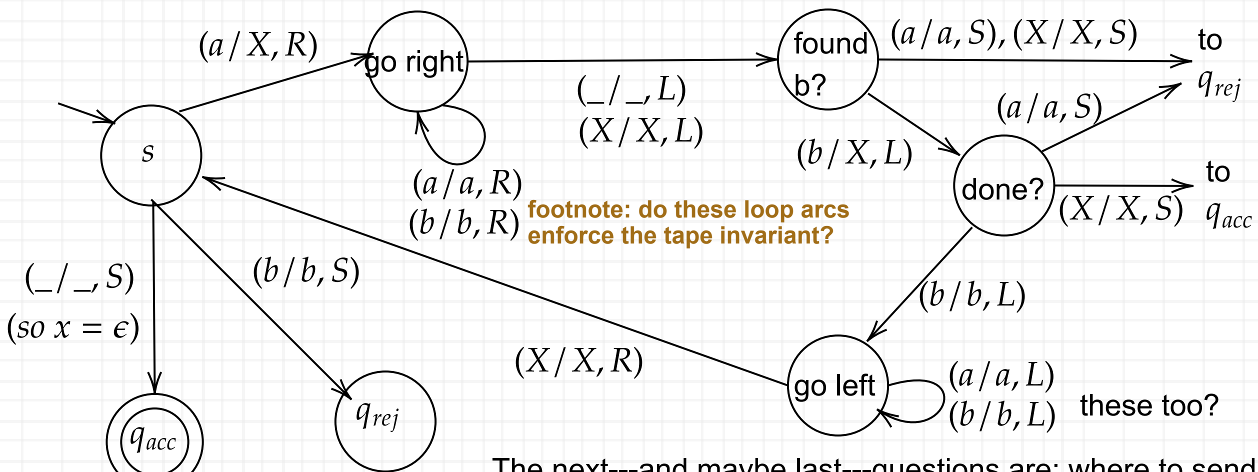


Note $\Gamma = \{a, b, \_, X\}$ so we need 4 arcs at each non-halting state. We added an arc on X at the "go right" state because on subsequent iterations the rightmost b will be next to an X not a blank. But what if there is no such b? Since we just X-ed an a, this means there were

such b? Since we just X-ed an a, this means there were initially more a's than b's, so we should reject.



Now after X-ing the matching b is when we need to talk about what is successful termination. If there is an X to its left then there are no more a's nor b's, so we paired them all, thus an X should mean goto q_{acc}. Getting an a once again means not enough b's. On b is when we want to "rewind" to the left end. That is when we need X to stop a leftward loop. So we cannot loop at the "done?" state itself but need another state:



The next---and maybe last---questions are: where to send the arc on X, and what actions to do? Most in particular:

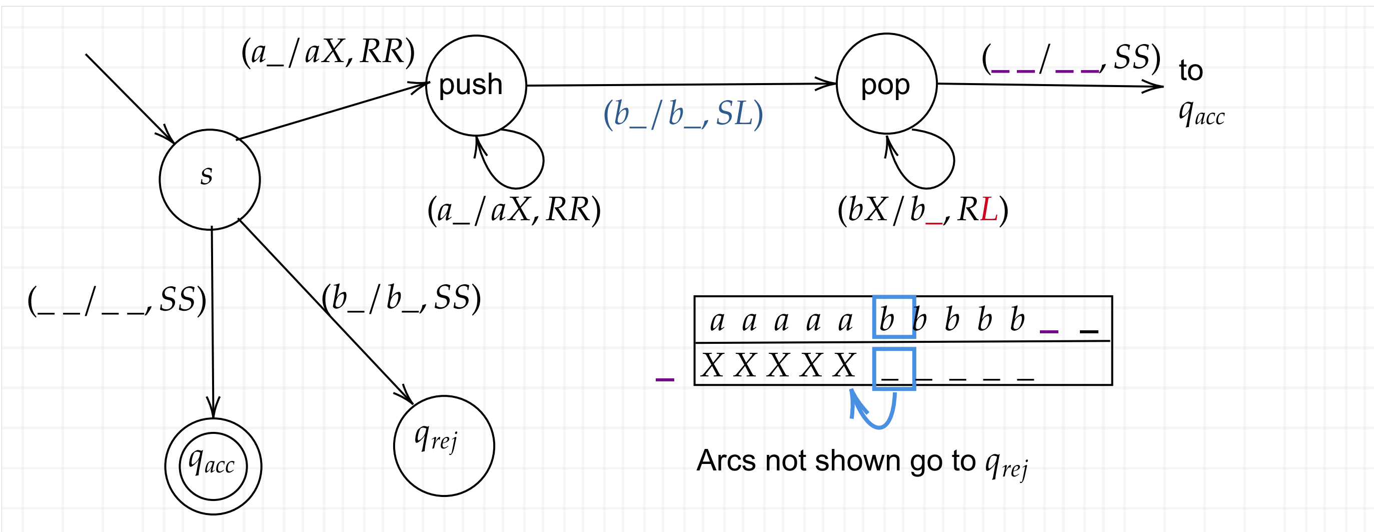
Can we complete the loop and the machine by making it be $(X/X,R)$ going back to start?

One thing to note is that if the char seen after executing $(X/X,R)$ is a b, then by the tape invariant it means there are no more a's but still at least one b since we went from "done" to "go left", so this is the case $m < n$. Well, in that case we should reject, and the arc on b going to q_{rej} is already there from the initial design. So: *this is OK and M is complete.*

Note that the input x can belong to $a^*\ b^*$ without belonging to L. Those strings abide by the tape invariant initially, and we can already see that M works correctly on those strings. But what if x is something like $aababb$? Will our M accept when it shouldn't? **That's what the footnote is about.**

Two-Tape Turing Machines (also Tue. Apr. 6)

Assuming M is correct---or quickly fixable if not---we can ask, how long does it take to accept a good $x = a^n b^n$ in terms of n? The answer is, it takes $\Theta(n^2)$ steps, owing to lots of backing-and-forthing. Can we make it run faster? There is a way to make it run much faster on one tape, in $O(n \log n)$ time, but we can get an optimal $O(n)$ running time by using a second tape:



Note the straightforwardness of the design as well as the efficiency. Also note the usefulness of having the second tape be two-way infinite with a blank to the left of the "column" initially holding the first a in x (if any). An alternative convention is to make both tapes one-way infinite but with a special char $\wedge$ in cell 0 at the left end on tape 1---so that the *initial configuration* I_0 has $\wedge x_1 \cdots x_n$ on tape 1 and just $\wedge$ on tape 2 "underneath" the $\wedge$ on tape 1. We can still start with the tape heads scanning the cells in "column 1" even if both are blank (so $x = \epsilon$). Then the final accepting instruction in the "pop" state becomes $(\_\wedge / \_\wedge, SS)$.

This two-tape DTM has the properties that:

- the input tape head never moves L and never changes a character;
- whenever the second tape moves L, it writes a blank in the cell it just left.

The second condition forces the second tape to behave like a ***stack*** (except for some "flex" in how top-of-stack is treated). A TM obeying these condiitons is formally equivalent to a **pushdown automaton** (PDA). A language is *context-free* (and belongs to the class CFL) if it is recognized by some PDA that may be nondeterministic (an NPDA); if the machine is deterministic (hence a DPDA) then it belongs to the class DCFL. Every regular language is a DCFL, and $\{a^n b^n\}$ is an example of a DCFL that is not regular.