We pick up with more examples of the "proof script."

**Example 3.** \(L = \{x \in \{a, b\}^* : x^R = x\}\), where \(x^R\) means \(x\) reversed, e.g., \(abba^R = babba\). [What is \(e^R\)?] That is, \(L\) is the language of strings that are **palindromes** and has the standard name PAL.

Take \(S = \_\_\). Clearly \(S\) is infinite.

Let any \(x, y \in S \ (x \neq y)\) be given. Then we can write \(x = \_\_\) and \(y = \_\_\) where \(\_\_\) and \(m\).

Take \(z = \_\_\).

Then \(L(xz) \neq L(yz)\) because \(\_\_\in\) PAL but which is not in PAL because \(m \neq n\) and the single \(b\) prevents any other possible way of "parsing" \(yz\) as a palindrome\(\_\_\).

Because \(x, y\) are an arbitrary pair of strings in \(S\), this shows that \(S\) is PD for \(L\), and since \(S\) is infinite, it follows that \(L\) is nonregular by the Myhill-Nerode Theorem.

We did not need the "wlog." provision this time---but you can always take it even if you don't need it. We also could have started with \(S = a^*\) and made the \(b\) the first char in \(z\). Why did I put the \(b\) "up front" in \(S\)? It is to emphasize its importance and help avoid a common mistake of forgetting it altogether. The mistake (in this case---it pops up in others too) is to think that \(a^m \cdot a^n\) is not a palindrome whenever \(m \neq n\). That may be true with your breakdown but there could be others. E.g. \(a^3a^5 = a^4a^4\) which is now clearly a palindrome. Indeed, \(a^{m+n}\) is always in PAL.

**Example 4.** \(L = \{(, )^* : x\ is\ balanced\}\). What does "balanced" mean?

[Discuss if time allows, but this will be more important when we cover **pushdown automata** as a special case of Turing machines.] This language is often called BAL. It is in fact "isomorphic to" the language of the "unlimited spears" and dragons game when you win only if you leave the dungeon with zero spears. E.g., if you are holding 5 spears, you need 5 "closing dragons" to balance out. With this understood, we can re-use the proof of Example 2.
The Full MNT

We have proved only one direction. The whole theorem says:

**Theorem:** A language $L$ is regular $\iff$ all PD sets for $L$ are finite.

We've proved that if $L$ has an infinite PD set, then $L$ is not regular. This is the $\implies$ direction, though it may sound like the reverse. It is the contrapositive of the $\implies$ direction. To complete the equivalence, we need to prove the $\iff$ direction.

**Proof:** All PD sets for $L$ are finite is the same as saying the equivalence relation $\sim_L$ has only finitely many equivalence classes. Take $Q$ to be the set of equivalence classes. For any string $x \in \Sigma^*$ (where $\Sigma$ is understood to be the alphabet that $L$ is "over"), there is exactly one equivalence class $R_x$ to which it belongs. Note that $R_e$ is an equivalence class, thus a member of $Q$, and it will serve as the start state $s$ of the DFA $M$ we are building. Next define

$$F = \{R_x : x \in L\}.$$ 

Note that even though $L$ may be infinite, $F$ can be finite because $R_x$ and $R_y$ can coincide---indeed, will coincide whenever $x \sim_L y$. Indeed, $F$ must be finite, because $F$ is a subset of $Q$ which is finite by the premise of $\iff$. Finally, we define $\delta$ by the rule

$$\delta(R_x, c) = R_{xc}.$$ 

For this to be "well defined" we need to show that it depends only on the equivalence class, not on any $x$ that happens to represent it. So suppose $y \sim_L x$, i.e., that $y$ also belongs to $R_x$, so that $R_y = R_x$. We need to show that $\delta(R_y, c) = R_{xc}$ too. This follows if $R_{yc}$ is the same as $R_{xc}$. And justifying this is left as a study guide. Then $M = (Q, \Sigma, \delta, s, F)$ is a legal DFA. And $L(M) = L$ because $M$ hits its accepting states exactly on the strings $x$ that belong to $L$. Thus $L$ is regular.

**Corollary:** In the $\iff$ direction of MNT, the DFA you get not only has the least possible number of states, it is unique. Hence, every regular language has a *unique minimum-size DFA*.  

\[ \Box \]
Putting a checkbox in the corollary statement signifies that we've already essentially proved it. The notes by Debray prove instead that every DFA can be reduced to a unique minimum one via the DFA minimization algorithm. The algorithm is interesting for its own sake—-it is IMHO the easiest example of "dynamic programming"---but for us it is just a "skim". The reasoning of both halves of MNT helps us recognize minimum-size DFA cases, even extreme ones.

**Proposition**: For each $k \geq 1$, the unique minimum DFA for $L_k = (0 + 1)^*1(0 + 1)^{k-1}$ has $2^k$ states.

Proof: Take $S = \{0, 1\}^k$. Then $S$ has size $2^k$. We claim that $S$ is PD for $L_k$: Let any $x, y \in S, x \neq y$, be given. By $x \neq y$, there is some position $i$ (let's number from 1) in which they differ. Take $z = 0^{i-1}$. Then $xz$ and $yz$ differ in position $k$ from the end, so $L_k(xz) \neq L_k(yz)$. This proves the claim, so the consequence is that any DFA $M_k$ such that $L(M_k) = L_k$ needs at least $2^k$ states. Well, we can build a correct $M_k$ of that size by having one state $q_w$ for each possible combination $w$ of last $k$ bits read (treating an initial small string like 10 as if it had $k - 2$ leading 0s) and defining $\delta(q_{bv}, c) = q_{vc}$. The final states are $q_w$ for those $w$ that begin with 1: since $|w| = k$, this 1 is in the $k$th position from the right. So $M_k$ is the unique minimum DFA for $L_k$. \(\square\)

Note that the NFA $N_k$ from an earlier lecture only needs $k + 1$ states. Thus this also demonstrates cases where the NFA-to-DFA construction has an unavoidable "exponential explosion." Furthermore, the regular expression for $L_k$ in the proposition statement (call it $r_k$) needs only $12 + \log_2(k)$ symbols, the log part for the bits in the number $k - 1$. This is an exponential step down in size. The upshot is that NFAs can sometimes be exponentially more succinct than DFAs, and regular expressions (with numerical powering) can be even more succinct in some cases.

**Using MNT For Design Hints (as time allows)**

We can use this $\iff$ direction to help us understand regular languages and build DFAs for them. Let's revisit the example $L = \{x \in \Sigma^* : \#0(x) \text{ is even}\}$. Then $x \sim_L y$ iff the numbers of 0s in $x$ and $y$ are both even or both odd. Hence the relation $\sim_L$ has just two equivalence classes. Here is the DFA:
Now let's try a trickier example by conjoining "even 0s" with another condition of not having 00 as a substring:

\[ L = \{ x : \#0(x) \text{ is even and } x \text{ has no } 00 \} \]  

[In regular expression terms, \( L \) equals \((1*01*0)^*1^* \setminus (0 + 1)^*00(0 + 1)^* \) but set-minus \( \setminus \) is not a native regular operator so that doesn't help us even build an NFA, let alone a DFA, to accept \( L \). So let's ignore this attempt and try using (1) to build a DFA \( M \) by "MNT-enlightened trial and error."\] We know that \( \epsilon \in L \), so the start state will be accepting, and that 0 and 00 are both not in \( L \). Indeed, 00 causes a "dead condition" because no string beginning with 00 can possibly belong to \( L \), so it should go to a dead state. That gives us part of the machine:

How about the string 1? It can still be a loop at the start state. At the left end of a string it makes no difference to having a possible 00, so 1 \( \in R_e \). But what about the loop on 1 which we had at the "odd" state? Can we still direct it back to that state? It is equivalent to ask whether 0 \( \sim_L 01 \). To see why not, consider \( x = 01 \) and \( y = 01 \). Take \( z = 0 \). Then \( xz = 00 \) is not in \( L \) but \( yz = 010 \) is indeed in \( L \), because the 1 helped us avoid a 00. For the same reason, 01 \( \not\sim_L 00 \), and clearly 01 \( \not\sim \epsilon \) because \( \epsilon \) is in \( L \) and 01 is not (technically, they are distinguished by \( z = \epsilon \)). Thus \( S' = \{ \epsilon, 0, 00, 01 \} \) is a PD set of size 4, and so we need a fourth state to process it to. Now, what about that string 010? It is in \( L \), but is it in \( R_e \)?

It is not, but finding a string \( z \) such that \( L(\epsilon \cdot z) \neq L(010 \cdot z) \) is not so fast. We need to activate the "no 00" condition by making \( z \) begin with 0, but then we need another 0-
--but not right away---to make \( z \in L \). Thus \( z = 010 \) is the shortest distinguishing string. This gives us:

\[
\begin{align*}
\text{Is } \epsilon &\sim_L 010? \\
\text{Is } L(0) &\neq L(010 \cdot 0)? \quad \text{no} \\
\text{Is } L(010) &\neq L(010 \cdot 010)? \quad \text{yes: 010 is in L but not 010010.} \\
S'' &\ = \ {\epsilon, 0, 00, 01, 010}
\end{align*}
\]

So we wound up needing 5 states. Is that enough? Well, can we complete the machine with arcs from the "even 0s, last char 0" state? Clearly 0 goes to \textit{dead}, and 1 must go to an accepting state. If 1 can go to the start state, then we're done. Can it? Yes---by similar reasoning to putting a loop on 1 at the start state. So \( M \) is done and \( S'' \) is a largest possible PD set.

There is another kind of reasoning we could have done. \( L \) is the \( \cap \) of two languages represented by the 2-state DFA above and the following simple 3-state DFA for the "no substring 00" condition:
Doing the Cartesian Product construction seems to suggest the final DFA will have \(2 \times 3 = 6\) states. But the operation is intersection, so the "dead" condition in the upper DFA knocks-on to make the whole third column dead in the product machine. Since you don't need two separate dead states, the number goes down to 5 after all. It is a good exercise to carry out the construction and verify that you get the same 5-state DFA as above.

[On tap Wednesday: Turing Machines.]