Main topics of the week:
- Pumping Lemma for CFLs
- Formal Definition of Turing Machine
- Computation of Turing Machine
- Examples of Turing Machines

Pumping Lemma for CFLs. For any context free language $A$ there is a pumping length $p$ such that for any string $s \in A$ of length $\geq p$, there are substrings $u$, $v$, $x$, $y$, and $z$ with $s = uvxyz$ and satisfying:

1. $uv^ixy^iz \in A$ for all $i \geq 0$
2. $|vy| > 0$
3. $|vxy| \leq p$

So basically, $s$ can be divided into five pieces, and the two middle flanking pieces $v$ and $y$ can be pumped (together, not individually). The second condition requires that at least one of $v$ and $y$ not be the empty string, and the third condition keeps the middle part close to the pumping length, even for a very long string. As before, this third condition will sometimes be useful in proofs.

The basic idea of this pumping lemma is that a sufficiently long string will have a tall enough parse tree so that we can find a repeated variable on a path through the tree. Because of the way variable substitution is context free in derivations, we can either collapse the tree between the repeated variable (thus cutting out the flanks of the subtree) or repeat the subtree more times at the second occurrence. The picture shows what is going on.

Let $G$ be the grammar for CFL $A$, and let $b$ be the largest number of symbols in the right hand side of all rules in $G$. Certainly $b \geq 2$ (otherwise, $A$ would just consist of a finite number of strings of terminals and the lemma would be vacuously true.) What is the significance of $b$? In a parse tree, it represents the maximum number of children of any node. This means that in the worst case, we have $b$ leaves at depth 1, $b^2$ leaves at depth 2, and in general, at most $b^h$ leaves in a parse tree of height $h$. And this translates into a string of length at most $b^h$ when the height of the tree is $h$. Put another way, this says that if the string is $> b^h$, then the height must be $> h$, i.e., $\geq h+1$.

So we choose our pumping length as follows. If $|V|$ is the number of variables in $G$, we choose $p = b^{|V|+2}$. From what we saw above, any string of length $\geq p$ (which is certainly $> b^{|V|+1}$) must require that the height of the tree is at least $|V|+2$. 
Given a string \( s \in A \) with length at least \( p \), we choose our parse tree carefully, since there may be several for the same string. In particular, we want the tree to be one with the smallest number of nodes. By what we have seen about the height, we know now that there must be a path through the tree of length at least \(|V|+2\), which means it has at least \(|V|+1\) variables (only the leaf of this path is a terminal). This in turn implies that some variable must be repeated among the \(|V|+1\) variables of the nodes. We’ll pick a repeating variable \( R \) such that \( R \) repeats among the lowest \(|V|+1\) variables (i.e., is closest to the bottom).

Now we can divide up \( s \). We let \( u \) and \( z \) be the portions of \( s \) that occur outside the subtree rooted by the higher occurrence of \( R \), we let \( x \) be the portion generated by the subtree rooted by the lower occurrence of \( R \), and we let \( v \) and \( y \) be the portions generated by the upper subtree, but excluding the nested lower subtree. As mentioned earlier, either instance of \( R \) may be substituted with either subtree and still obtain a valid tree. When we replace the upper (larger) tree with the lower (smaller) tree, we eliminate \( u \) and \( y \). When we replace the lower tree by the upper tree, we replace \( x \) by \( uxy \), thus getting the pumping up.

To see that both \( v \) and \( y \) are not the empty string, suppose that they were. Then the substitution of the small tree for the large tree would still produce \( s \), yet it would have fewer nodes (having eliminated an \( R \)), contrary to our choice of the tree. So at least one of \( v \) or \( y \) must be non-empty.

Finally, for the third condition, consider the length of \( vxy \), which is generated by the upper tree. The height of this tree is \( \leq |V|+2 \) since we made sure to choose \( R \) within the
bottom $|V|+1$ variables (and then there’s the leaf to give height $\leq |V|+2$). By our earlier argument, the length of the string generated (namely $vxy$) cannot exceed $b^{\sqrt{|V|+2}}$, which is of course our choice for $p$.

**Example of a Non-Context Free Language.** The language $A = \{a^i b^j c^k \mid k > i \text{ and } k > j\}$ is not a context free language. Suppose it is context free and let $p$ be the pumping length and let $s = a^p b^p c^{p+1}$. When this is realized as $uvxyz$, since $|vxy| \leq p$, $vxy$ cannot span more than two different symbols. If $v$ and $y$ contain no $c$'s, just $a$'s and $b$'s, then pumping up as $uvvxyyz$ would increase the count of $a$'s and/or $b$'s, but not $c$'s, and this would mean the string $uvvxyyz$ is not in the language $A$ since the count of $c$'s would not exceed both the counts of $a$'s and $b$'s. Otherwise, $v$ and $y$ must contain $c$'s, (and might contain $b$'s, but no $a$'s). Then pumping down as $uxz$ would decrease the count of $c$'s, but leave the $a$’s alone, again meaning that $uxz$ is not in the language since it would not have less $a$’s than $c$’s. In either case we have reached a contradiction, so $A$ must not be context free.
Computation in a Turing Machine.

To be able to formally define computation in a Turing Machine, we must have the notion of a configuration of the machine. A configuration consists of the state, the tape contents, and the head location. A single step according to the transition function of the machine will bring us to another configuration. The starting configuration consists of the start state, the tape with the input on it, and head location on the first cell of the tape. A configuration can be specified as a string over the tape alphabet, with a state appearing somewhere in the string. This captures the idea of the tape contents, the state, and the head position is the symbol immediately after the state as it appears in the string. The following diagram shows one configuration that yields another, i.e., is achieved by a transition step.

\[
\begin{align*}
    \delta(q_i, b) &= (q_j, c, L) \\
    uaq_i bv &\text{ yields } uq_j acv
\end{align*}
\]

With this idea of configurations and yielding, we can more formally state that a Turing Machine M accepts input w if there is a sequence C_1, C_2, ..., C_k of configurations where
1. C_1 is the start configuration of M on input w,
2. C_i yields C_{i+1} for 1 ≤ i < k,
3. C_k is an accepting configuration (the state is q_{accept}).

Likewise, we can define reject. Note that accepting and rejecting configurations are halting configurations, so cannot yield another configuration.

In previous automata, we thought of the machines as being driven by the input, reading characters one at a time, and deciding what to do based on the transition function. The difference with Turing Machines is that the “input” all appears on the tape at the start, and we have the head positioned at the first character and use the transition function to decide what to do next. But after that start state, there is not really a concept of reading the input unless the machine is designed that way – that is, the actions are governed not so much by what is the next character, but what is the character under the head. And since the head can move left or right, it might not be what we think of as the next input.
character as we have been doing. And especially since the character can be overwritten, we can’t keep thinking of it as the input character. It’s just the tape, with an initial character configuration on it.

**Example of a Turing Machine.**

Consider a Turing Machine that accepts the language \( \{a^n b^n c^n \mid n \geq 0\} \). We have seen that this is not a context free language, so know that we cannot design a PDA to recognize it. The informal description of the TM is:

1) If no unmarked symbols, *accept*, otherwise mark an a.
2) If no a, *reject*, otherwise skip to next b and mark it.
3) If no b, *reject*, otherwise skip to next c and mark it.
4) If no c, *reject*, otherwise rewind and go to step 1.

In the diagram we have not included the reject state, but assume that any unspecified transition leads to it.

Notice that we use the tape alphabet character ‘x’ to mark off matched b’s and c’s. We use \( \omega \) to mark a’s and this also serves to allow us to rewind to the beginning. When we are at the start state, we skip over any x’s, and if we read a \( \omega \), then we accept since we must have matched everything. If we see an ‘a’, we mark it with \( \omega \) and skip over subsequent a’s. Then we mark a ‘b’ with ‘x’ and skip over subsequent b’s. Then we mark a ‘c’ with ‘x’ and skip over further c’s, stopping when we get to a \( \omega \). At this point, we rewind back till we encounter a \( \omega \), which must have been one of the a’s marked off. The cycle thus repeats. There are lots of places to reject in this machine: we might not see an ‘a’ from the start state, or we might see an ‘a’ or ‘c’ when looking for a ‘b’, and so on.
Turing Machines may also be viewed as a way of performing what we typically think of as computation, rather than just accepting or rejecting strings (although recognizing a language in the formal sense we have seen is completely equivalent to any computational problem). But seeing how we could perform an arithmetic computation shows how similar Turing Machines are to the familiar notion of programming on computers. In this example, we omit the accept and reject states because all we care about is how the tape looks when we halt. That is, the “computation” is to consider how the machine transforms the input string on the tape.

The example here computes the difference between two numbers $m$ and $n$, or zero if $n$ exceeds $m$. We use unary representation of numbers, that is, $m$ will be represented by a string of $m$ zeroes, and $n$ will be represented by a string of $n$ zeroes. Likewise, our result will be similarly represented, and we will use 1s to separate the values. So the tape will begin with the string $0^m10^n$. If we get the transitions right, the machine will halt with the tape having the string $0^{m-n}$, where $m-n$ is the difference or zero, whichever is greater (formally called the *monus* operation).

We start by replacing the initial 0 by a blank so we can find the beginning of the tape again. Then we move to find the separating 1 and change the zero that follows it to a 1. Then we rewind to the beginning zero and repeat. Basically, we’re replacing the 0s in $m$ by blanks, and replacing the corresponding 0s in $n$ by 1s. If we run out of 0s in $n$, then we back up, replacing 1s by blanks, and replace one 0 by a blank. So, we have blanked out $n$ 0s from the $m$ string, and blanked out all of $n$’s 0s. We adjust for the initial blanking of a zero from $m$ by resetting one blank back to a zero. If, on the other hand, we run out of 0s in $m$, then we blank out everything, which indicates a result of zero.
You can consider the various places where the machine halts, e.g., if the input is not of the required form. In these cases, we assume a transition to a reject state. Otherwise, the machine will end in the accept state, and the tape will be left recording the desired result.

Compute $\max(m-n,0)$