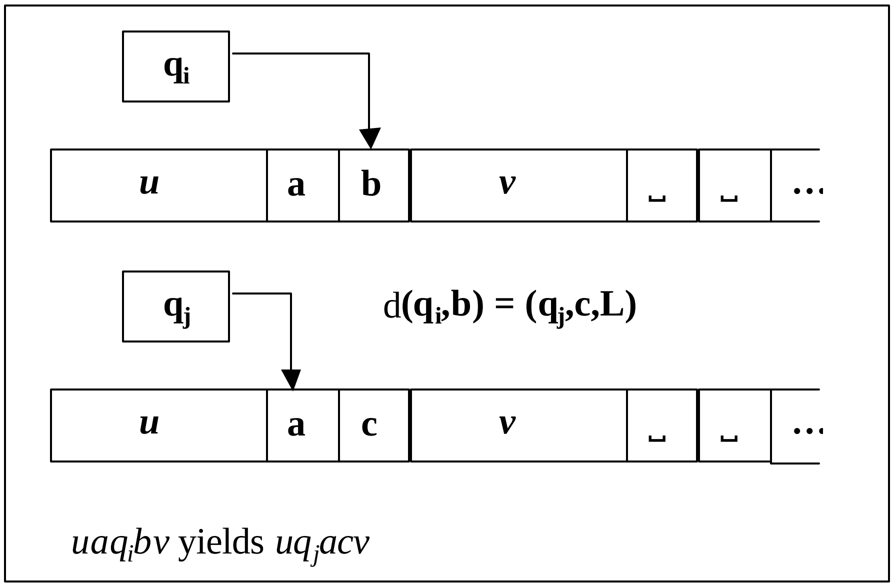
Main topics of the week:

- **Formal Definition of Turing Machine**
- **Computation of Turing Machine**
- **Definition of recognizable, decidable**
- **Examples of Turing Machines**
- **Variations of Turing Machines**

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.



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 $C_1,C_2,\ldots,C_k$ of configurations where

1. C_1 is the start configuration of M on input w,
2. C_i yields C_{i+1} for $1 \leq i < k$,
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.

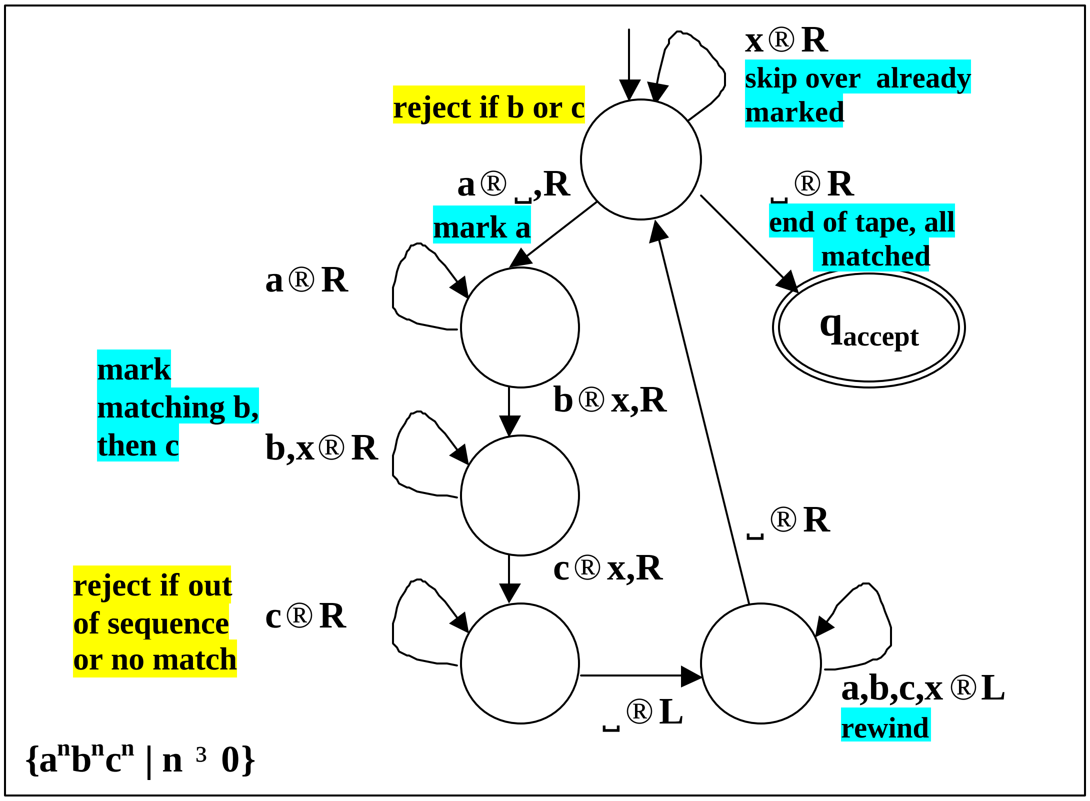
In previous automata, we thought of the machines as being driven by the input, reading characters one at a time, and deciding what to do based on the transition function. The difference with Turing Machines is that the "input" all appears on the tape at the start, and we have the head positioned at the 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.



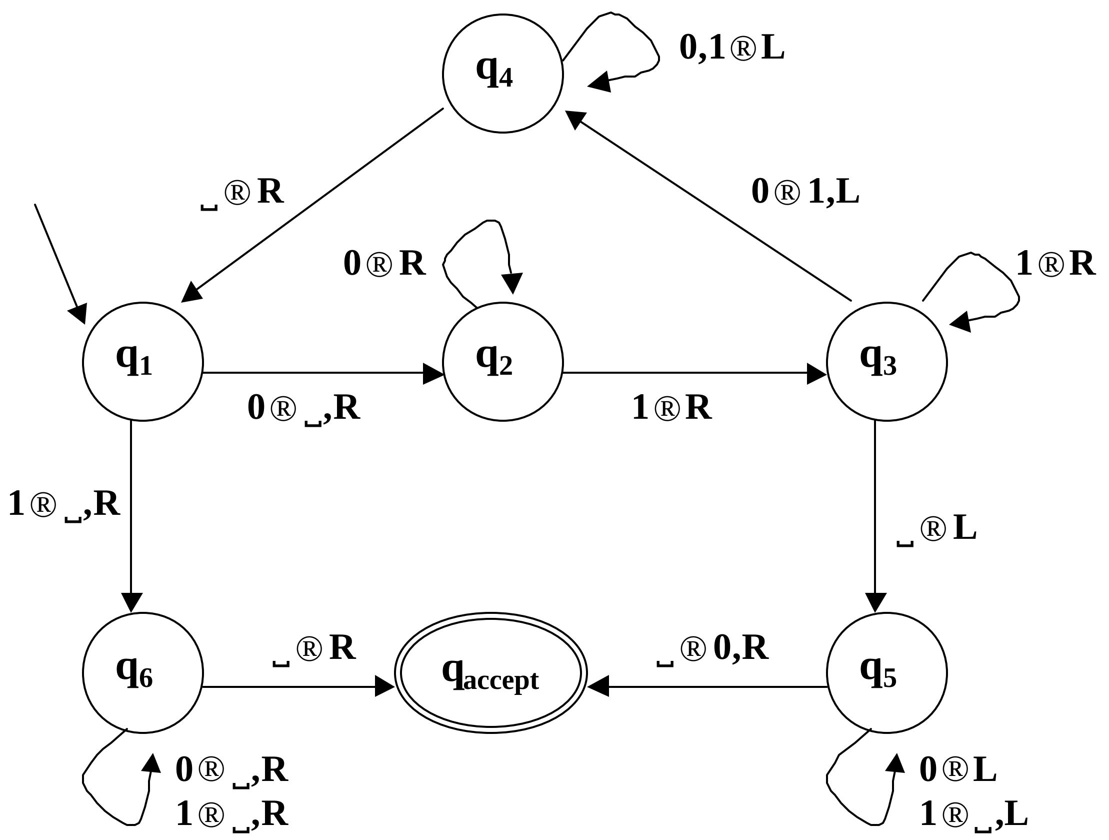
Notice that we use the tape alphabet character ‘x’ to mark off matched b’s and c’s. We use ␣ to mark a’s and this also serves to allow us to rewind to the beginning. When we are at the start state, we skip over any x’s, and if we read a ␣, then we accept since we must have matched everything. If we see an ‘a’, we mark it with ␣ 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 ␣. At this point, we rewind back till we encounter a ␣, 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 $0^{m\sim n}$, 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)