The Lecture Title

Scribe: Your Name

Date: Day, Mon, Date Year is a

1 Turing Machines

A *Turing Machine* is a fundamental model of computation that can be thought of as a black box that can compute things.

An *alphabet* is a set of characters or symbols. A *word* is a string of characters from the alphabet. A *language* $\mathcal{L}$ is a subset of words.

For example - a language is the set of all $\{0,1\}^*$ with an even number of 0s, or all English sentences in $[a-z]^*$ that include the word "the," or all $\{0,1\}^*$ that are prime in binary.

Languages can be anything we essentially want them to be - for example, we can encode the set of all a, b, c, n such that $a^n + b^n = c^n$ in a language. We can compute everything in this language to solve this difficult problem. (known to be the trivial language).

A finite automata can only read in inputs from left to right, and spit out an answer based on the inputs.

A *push down automata* reads a character from the input, and pushes it onto a stack that can be popped from. This allows a push down automata to figure out what even-length binary strings are palindromes.

If we have a non-deterministic PDA that will take any path that leads to an accepted state at some point, we can consider the following algorithm:

1. Reads a character from input.
2. Push that character onto the stack.
3. Repeat OR go to 4.
4. Read a character and pop off the stack.
5. Compare to next character.
6. Repeat.
7. If empty stack when out of input, ACCEPT.

The non-deterministic PDA will magically know what step is necessary in order to get to the ACCEPT state, so the algorithm will automatically go to step 4 when it reaches the middle of the word. One can think of this as having the PDA already having searched through the entire state space and already knowing what the proper step is to take.

We can modify this to accept palindromes of odd or even length, by adding the step to read and discard a character and moving on to the next step at step 3. This, again, requires knowing when to discard the middle character.

Turing machines can to all the things that PDAs can, but even more. Turing machines have an input, a state, and an infinitely long tape that can be written to and read from.

The Turing machine can:

- read from the input or the tape.
- write to the tape.
- move left or right along the tape.

We can look at a relatively simple problem - suppose we wanted to add integers x and y in unary. For example,

$$7 + 3 = 1111111 + 111 = 1111111111$$

As an input, we will style the inputs as strings of $1, 0, \#$s, and blanks. The above example would look like

$$1111111\#111$$

Our code essentially consists of the two steps: if we see a 1, we copy a 1 and move to the right, and if we see the pound sign, we skip it.

We can multiply using a more complex algorithm. We start by copying the entire input until we run into a blank space. We can copy a 1 to the right to begin with, marking all characters already copied. Once this is done, we mark a 1 and revert the first part of the input to 1. This is done while the second part of the input still has ones, after which the output will have a string that has a length that has multiplied the two numbers.

What if we want to convert a binary number to unary? We start in the first state (state 0) where we either see a 0 and do nothing, or go to state 1 after seeing a 1 and copying the 1. Once we are in state 1, if we see a 0 thereafter, we double the output string, or if we see a 1, we double the output string and add 1. This will give the correct unary representation of the binary number.

It now remains to double a number, which can be done by copying the string next to itself and adding them.

What about dividing binary numbers? We can convert to unary, and then subtract the second number from the first by overwriting ones with zeroes. Every time this is done, we increment a counter elsewhere. We terminate when there are too few digits to subtract from.

We move on to enhancing our Turing Machine. Can we improve this by adding more tapes? It turns out the answer is no - we can essentially weave the information together and encode where the information starts, so this doesn't actually improve our machine, and neither does having more than one pointer to read and write.

Can we improve our machine by introducing non-determinism? Consider the problem of finding whether or not a given binary number is composite. For a deterministic algorithm, we can consider trying every integer from 2 to x, and seeing whether it divides evenly. This takes on the order of a division times the smallest prime that divides x. Non-deterministically, we can just choose the correct number that divides x immediately, and return it if it divides x and is less than x.

In general, the deterministic algorithm runs exponentially, and the non-deterministic algorithm runs in log time. Is this always going to be the case? It turns out a fundamental question in computer science asks whether the set of languages computable by a non-deterministic Turing machine is larger than the set is computable by a deterministic Turing machine. If we restrict each of these Turing machines to polynomial time, this is a famous open problem $P = NP$.