Other Automata

We can create PDAs with multiple stacks. At each step we look at the current state, the current input symbol, and the top of each stack. From all of this information we decide what state to transition to and what to write on each stack.

Theorem: A TM can simulate a PDA with k stacks.
Proof: Use a TM with k+1 tapes: one for the input, the other tapes each representing one stack.

Theorem: A PDA with 2 stacks can simulate a TM.
Proof: Suppose the current configuration is 10BXq00Y
We will store the tape as

X	
B	0
0	0
1	Y
Z_0	Z_0
stack 1	stack 2

The current tape symbol is always at the top of stack 2; the symbol to its left is at the top of stack 1. We move Left on the tape by popping stack 1 onto stack 2; we move Right by popping stack 2 onto stack 1. The PDA controller can simulate the actions of the TM controller: read the tape symbol (top of stack 2), write a new tape symbol (pop stack 2, push new symbol) and move left or right.

Counter Machines

A k-counter machine has a finite state controller and k counters in place of a stack or tape. Each counter is a non-negative integer. The only operations with a counter are

- Add 1 to the counter
- Subtract 1 from the counter
- Test the counter for being 0

Note that we can represent a counter with a stack. To increment the counter push something onto the stack; to decrement the counter pop the stack; to test for 0 ask if the stack is empty.

Theorem: Any language accepted by a 1-counter machine is context-free.

Theorem: Any language accepted by a k-counter machine is recursively enumerable (accepted by a TM)

We will now show that we can simulate a TM with a 3-counter machine. We know we can simulate a TM on a PDA with two stacks so we'll show we can simulate the latter with 3 counters.

Suppose the stack alphabet has (r-1) symbols. Identify these symbols with the digits 1...(r-1). (We deliberately don't use 0.) Then represent the stack

X_1
X_2
...
X_n
Z_0

by $X_1+X_2r+X_3r^2+...+X_nr^{n-1}$; i.e., treat the stack as a number in base r.

For example, suppose we have 9 stack symbols A,B,C,D,E,F,G,H, I, which we identify with 1,2,...9. Then the stack

D

B

B

C

A

Z_0

is represented by 13224.

To push symbol X onto the stack represented by a counter, multiply the counter by r and add X.

To pop the stack, divide by r and ignore any remainder.

To get the value at the top of the stack, take the counter mod r.

To simulate 2 stacks with 3 counters, we will use 2 counters to represent the contents of the two stacks and keep the other counter free.

1. We can have states that add a fixed number to a counter (by doing that many increments).
2. To multiply a counter A by r, first empty the free counter. Add r to the free counter and decrement A. Again add r to the free counter and decrement A. Continue until A is 0. The free counter now has A*r. Copy this to counter A.
3. To divide counter A by r, first empty the free counter. Decrement A r times then increment the free counter. Repeat until A can't be decremented r times. The value in the free counter is A/r.

4. We need to be able to compute A%r without changing A. Make a cycle of r nodes m_0 m_1 ... m_{r-1} m_0. Empty the free counter. Walk around the cycle of nodes at each step decrementing A and incrementing the free counter. When A is empty the free counter has the original value of A and the index i of m_i is the value of A%r. So copy the free counter to A and then increment the free counter i times.

These operations let us simulate the top, push and pop operations of two stacks with 3 counters.

Theorem: We can simulate a 3-counter machine with just 2 counters.

Proof: Suppose the 3 counters have values i, j, k. Represent all three with one counter $2^i3^j5^k$.

a) To increment a counter multiply by 2, 3, or 5.
b) To decrement a counter, divide by 2, 3, or 5.
c) To determine if a counter is 0, look at the remainder when dividing by 2, 3, or 5. If the remainder is 0, the counter is not 0.

Moral: a 2-counter machine is equivalent to a TM.

We can use a TM to simulate a computer

Go to the assembly-language level of the computer.

Use 4 tapes:

- One with 2 tracks for memory contents and addresses
- One for registers, including instruction pointer
- One for user input
- One for scratch work

The instruction cycle is

a) Copy the next instruction onto the scratch tape
b) Fetch the arguments
c) Perform the instruction
d) Store the results

All of this can be done on a TM.

We can also simulate a TM on a computer if we have a way to arbitrarily expand its memory. More on this later.

Here are 3 models of computation:

A. Turing Machines
B. Unrestricted grammars
C. Recursive functions (Church, Kleene) -- functions that can be derived primitive functions via composition and recursion

These are equivalent in the sense that

- Any language accepted by a TM can be derived from an unrestricted grammar.
- Any language derived from a grammar can be "enumerated" by a recursive function in the sense that f(n) is an encoding of the nth string in the language.
- Any recursive function can be computed by a TM.

Church's Thesis (Turing, Church 1936): Anything we might describe as an algorithm can be implemented on a TM.

We say a programming language is *Turing Complete* if it is powerful enough to simulate a TM. Any algorithm can be implemented in a Turing Complete language.

We can simulate a TM on a computer and a computer on a TM. How do the running times compare?

Theorem: There are polynomials p_1 and p_2 so that we can simulate n steps of a program on a computer with $O(p_1(n))$ steps on a TM and n steps of a TM with $O(p_2(n))$ steps on a computer.

Proof: We can simulate a computer with a 4-tape TM. Each move on the computer requires a lookup of data. In n moves we can write at most n values into memory, so each lookup takes at most O(n) steps on the TM. Each move on a 4-tape TM requires looking up the current value at each of the tape heads, which takes time O(n). (continued..)

Altogether, each move on the computer can be simulated in $O(n^2)$ moves on a standard TM, so $p_1(n)$ is n^3.

We can simulate a TM on a computer where we represent the tape by two arrays -- one for positive indexes and one for negative indexes. In n steps we can write at most n symbols on the tape, so any index will be between -n and n. If n is huge incrementing and decrementing an index will no longer be constant time but it won't be any worse than O(n). $p_2(n)$ is n^2.

Moral: Any problem solvable on a TM in polynomial time is solvable on a computer in polynomial time, and vice versa.