1 Classical Computation Models

1.1 Turing Machines

Though not quite the first, *Turing Machines* are perhaps one of the most important models of classical computation. A Turing Machine (or TM) has an infinite read/write tape, of which all but a finite portion is blank. There is some finite *alphabet* of symbols that can be written on the tape, in addition to the blank symbol □. There is a "head" which does the computation. The head (or *control*) can be in a finite set of states. The program is a set of rules of the form

state	symbol	state	symbol	movement
1	'0'	2	'1'	R
1	'1'	2	'1'	R
1	'□'	2	'1'	L
...				

Turing proved that Turing Machines are universal: that is, that there is a universal Turing Machine program that is able to simulate any other Turing Machine's behavior on any input, when that other TM is written appropriately as part of the input to the universal TM.

There is a special state called "HALT". Each TM computes a function f where $f(s)$ for some string s is whatever is left on the tape when the state reaches "HALT," where initially, the machine is in its initial state, the input s is on the tape, and the head is on the leftmost symbol of the input. Actually, this is not quite a function in all cases, because not every TM will ever reach "HALT" on every input, so for some values of s, it may be that $f(s)$ is undefined. When we say that a TM computes a particular function f, this means that it computes that f on every input, and thus it must halt on every input.

1.2 The Halting Problem

This is the famous Halting Problem: given a TM M and an input j, does M halt on j? This is an uncomputable (also called *undecidable*) function.

If the Halting Problem were computable, there would be a TM M which on input i halts if and only if TM number i does *not* halt on input i. But then M would be TM number j for some j, and M could neither halt nor not halt on input j, which is a contradiction. Thus, the halting problem is undecidable.

1.3 The Wires and Gates model

Here, the idea is that computation devices are circuits consisting of a finite number of *gates* joined up by finitely many *wires,* which carry 0s and 1s on them. Each gate is a function of one or two bits. The three basic gates are the NOT gate, the AND gate, and the OR gate, whose operation is simple.

x	y	NOT(x)	AND(x,y)	OR(x,y)
0	0	1	0	0
0	1	1	0	1
1	0	0	0	1
1	1	0	1	1

A theorem: any boolean function can be composed out of AND, OR, and NOT gates. In fact, these three can be computed using just a NAND gate, which gives the opposite output from an AND gate.

An *acyclic* circuit is a directed graph with gates on the notes, where NOT gates have one input and AND and OR gates have two inputs; each gate may have many outputs. Certain edges may have no gate at their beginning; these are the *input* wires. Others may have no gate at their end; these are the *output* wires. In order to compute the output from the input, we need only propagate values along the wires, computing each gate as we get to it, and using that output on every wire out of the gate. The values that end up on the output wires make up the output of the circuit. (Alternatively, we can place special "input" and "output" gates at the beginning of input wires and the end of output wires, respectively.)

A function f is computable by circuits if there exists a *uniform* family of circuits C_n such that C_n maps x to $f(x)$ for all strings x of length n.

Note: there is a family of circuits such that C_n computes the following function: $f(0^n) = 0$ if TM number n halts on input n, or 1 if TM number n does not halt on input n, and $f(x)$ for $x \neq 0^n$ is 0. However, this family is not *uniform.*

C_n is uniform if the description of C_n can be output by a TM on input n. C_n is polynomially uniform if C_n can be output by a TM on input n, in $\leq p(n)$ steps for some polynomial p.

1.4 Circuits can simulate TMs

In order to show this, we conside the *history* of a TM computation. For each step, we can write down a configuration: the contents of the tape, where the head is, and which state the head is in.

For each space on the tape, we have a wire or two to determine what symbol, if any, belongs there, and a wire to say whether or not the head is there, and some other wires to determine which state the head is in, if it is at this space. Given the wires for a particular space, and for the spaces to the left and right, a circuit can compute what belongs in that space on the tape at the next step, going out on wires similar to the ones that came in. Thus, by making enough of these circuits for individual tape spaces at individual times, we can compute what the Turing Machine computes.. almost.

The problem is the infinite tape. We can make this circuit/tape arbitrarily broad, and arbitrarily deep, but how deep does it need to be, and how broad? Certainly if the TM in question takes time $\leq p(n)$ for all inputs of size n, then there is a uniform family of circuits C_n computing the same function with $|C_n| \leq \alpha p(n)^2$, where we simply make the circuit as broad as it is deep, with dimensions $p(n)$ for each.

1.5 Reversible computation

A result of Landower, mentioned in the first lecture: erasing a bit takes energy. Some gates are reversible (NOT) while others are not (AND). The FANOUT gate (a gate that replicates its input twice, to represent branches in the wires) is also reversible.

We can, however, do computation reversibly, using Toffoli gates. The essential Toffoli gate is the controlled NOT gate, which takes 3 inputs and has 3 outputs. If the first two bits are 11, the output switches the third bit and leaves the first two alone. Otherwise, the output is the same as the input. This gate is universal: we can compute a NOT gate by using two wires with 1s on them with the wire we wish to put a NOT gate on. We can compute an AND gate by computing the Toffoli gate on $(x, y, 0)$, which will output $(x, y, x \wedge y)$. (We can also do a FANOUT gate: $(1, x, 0)$ goes to $(1, x, x)$. Of course, doing this leads to having junk wires, which is a bit problematic. However, Bennett [1973] showed that any circuit computation can be made reversible *without* junk wires if the input is kept around.

The idea of Bennett's argument is that we can do the following:

INPUT	00000000000000000000	0000000000
INPUT	OUTPUT JUNK	0000000000
INPUT	OUTPUT JUNK	OUTPUT
INPUT	00000000000000000000	OUTPUT

The idea here is that we compute the first transition by computing the function using Toffoli gates, which produces some junk. Then, we can use the zeros that are untouched to copy the output, and finally, we can just compute the first part backwards on the INPUT-OUTPUT-JUNK, which gives us the INPUT and the 0s again, thus getting only INPUT, OUTPUT and 0s. Actually, we don't have to keep the input around in the middle; if it gets corrupted in the first step, this will be undone in the third.

2 Quantum Gates

This idea of gates is fairly similar to what we can do in quantum computers. A quantum gate is a linear map (matrix) on a vector space. Single qubit gates are 2×2 matrices, 2-qubit gates are 4×4 matrices, and 3-qubit gates are 8×8 matrices. It is open whether we need any larger gates than 3-qubit gates. In any case, they are very difficult to construct.

Here are the 1-qubit gates we have.

$$\sigma_x = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

$$\sigma_y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}$$

$$\sigma_z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

$$H = \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{pmatrix}$$

$S = \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix}$

$T = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{pmatrix}$

These are all 1-qubit gates. Here is a 2-qubit gate:

$$\text{CNOT} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

so that

$$\begin{array}{ccc} 00 & \to & 00 \\ 01 & \to & 01 \\ 10 & \to & 11 \\ 11 & \to & 10 \end{array}$$

What happens if we change the basis and look at the CNOT gate? Let $|+\rangle = 1/\sqrt{2}(|0\rangle + |1\rangle), |-\rangle = 1/\sqrt{2}(|0\rangle - |1\rangle)$.

$$|+\rangle|+\rangle = 1/2(|00\rangle + |01\rangle + |10\rangle + |11\rangle) \overset{CNOT}{\longrightarrow} |+\rangle|+\rangle$$

$$|+\rangle|-\rangle = 1/2(|00\rangle - |01\rangle + |10\rangle - |11\rangle) \overset{CNOT}{\longrightarrow} 1/2(|00\rangle - |01\rangle - |10\rangle + |11\rangle) = |-\rangle|-\rangle.$$

Since $CNOT$ is symmetric, $CNOT(|-\rangle|-\rangle) = |+\rangle|-\rangle$ and $CNOT(|-\rangle|+\rangle) = |-\rangle|+\rangle$.

This is quite interesting: CNOT remains a controlled not bit, but now, it is the *second* bit that determines whether or not we flip the first!

(Note: H is the matrix that changes the $|0\rangle, |1\rangle$ basis to the $|+\rangle, |-\rangle$.)

$$\begin{array}{ccccc} H & -- & \cdot & -- & H \\ & & | & & \\ H & -- & \oplus & -- & H \end{array}$$

yeilds a controlled not switched the other way:

$$\begin{array}{ccc} -- & \oplus & -- \\ & | & \\ -- & \cdot & -- \end{array}$$