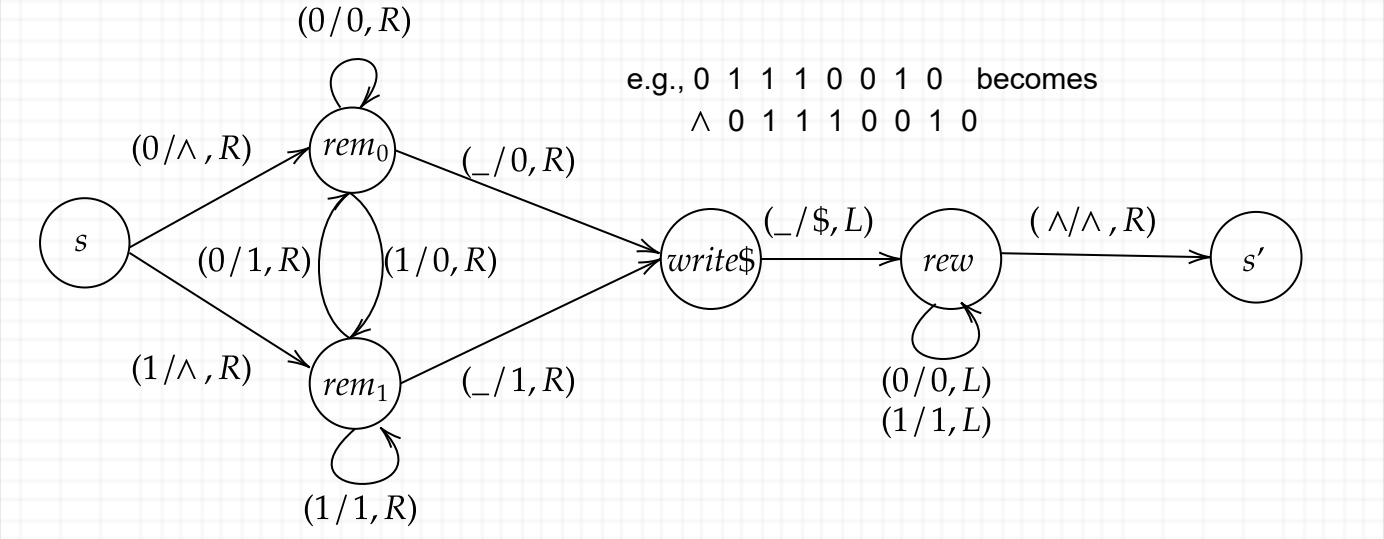
CSE491/596 Lecture Fri. 10/2: Turing Universality and Decision Procedures

What kinds of operations can Turing machines carry out? We have seen most of the following:

1. Copy a string from one tape to a second tape
2. Compare two strings on separate tapes to test whether they are equal
3. Search leftward or rightward on a tape until reaching the end or a sought-for char
4. Loop with back-and-forth *passes* until an exit condition is met
5. Multiply numbers by 2 (by appending a 0), or divide by 2 if even, or multiply by 3...
6. Add two binary numbers
7. Remember a char in a state while shifting an entire string over one cell
8. Use dedicated states to write a dedicated string to a desired location

We haven't seen the last two. Here is an example of how to insert a $\wedge$ in front of a binary string x and also put a $ after it:



This shows that the "$\wedge$ convention" for $I_0(x)$ can always be emulated by the bare-startup convention, at the cost of only $2n+1$ extra steps on inputs of length n. Moreover, if we need to insert a special char, say @, in the middle of a tape string at any given state q, we can attach this entire routine at state q by making $s = s' = q$ and using @ in place of $\wedge$, except that the last arc becomes $(@/@, S)$ so that the head is scanning @ in state $s' = q$ and so can execute an option that was not available before. This "shift-over" routine can thus act like an invokable process that returns control to its point of call. We can repeat it to make more room. We can also compose it with operation 8 by having more states in place of "$write\$$" that lay down whatever fixed string y we want to append after x.

We can make other operations by composing two or a few of the above. By combining 2 and 3 we can solve the problem of finding a substring w inside a larger string x (by testing place-by-place, but there are also quicker ways used by compilers). We can multiply two binary numbers by using repeated addition or by using shifts to emulate the grade-school algorithm and adding up the shifted copies of the first number. This small vocabulary of machine ops suffices to simulate a rudimentary assembly

language. The following one is coded to use just one argument for each instruction, but it is fairly flexible: it even has indirect load (LDI) and store (STI):

1. LDL n : load a hard-coded literal integer n into the ALU
2. LDR Y : load the contents of register Y into the ALU
3. LDI Y : read the contents of Y as another address Z, then do as in LDR Z
4. STO Y : copy the contents of the ALU into register Y, replacing whatever is there
5. STI Y : read Y to get the indirect address Z, then do as in STO Z.
6. ADD Y : add the contents of register Y to the number currently in the ALU
7. SUB Y : subtract the contents of register Y from the number currently in the ALU
8. SHF d : shift the ALU by the hard-coded number d of places (shift left if d is negative)
9. ABS : take the absolute value of the number in the ALU
10. JMP ℓ : jump to the hard-coded instruction number ℓ if the ALU currently holds 0.

Indeed, the main reason for real assembly languages having many more primitive instructions is having different types and sizes of operands: 8-bit char, 16-bit int, 32 or 64-bit float, etc. Whereas a real "RAM computer" has fixed-size registers, our Turing machine can emulate arbitrary-size registers thanks to how the "shift-over" routine can be sprinkled into its state code to make any extra room needed to store a bigger value. Thus our "mini assembly" language is actually rich enough to be a compilation target for any high-level programming language (ignoring special object features put at machine level and the like).

The "Universal RAM Simulator" handout
`https://cse.buffalo.edu/~regan/cse396/UTMRAMsimulator.pdf`
has enough hand-drawn detail to serve as proof-of-concept. Immediately what it proves is:

Theorem 1: We have built a single DTM U such that for any mini-assembly program A and integer argument x to A, U on input $\langle A, x\rangle$ outputs the result (if any) of running A on input x. ☒

Here the angle brackets in $\langle A, x\rangle$ stand for "some transparent way of combining A and x into a single string." We've already been using this notation with IDs, in case we would want to read them as strings (as later, we will). One way to implement it is fine provided the angle brackets and comma don't occur inside A and x: we can treat them as literal characters over, say, the ASCII or UNICODE alphabets---which we can then convert to binary if we wish. Another that works in this case is to just ram the two strings together as xA or Ax, which is fine in teh handout since A begins with a ! and ends with a semicolon, both of which we suppose do not occur inside x. In general, one can regard the angle brackets as applying a *pairing function*. (Some sources devote time to pairing functions, which can be composed to encode any tuples as strings and also decode them, but we can dispense with the details.)

Now we add a further wrinkle using operation 8 above when A is a single fixed program, rather than one given on-the-fly.

Theorem 2: Given any program P in any known high-level programming language (HLL) that uses standard input and output, we can build a Turing machine U_P such that for any input x to P, U_P on input x replicates the stream output of $P(x)$. In particular, if P computes a function f then U_P computes the same function---and moreover, does so with roughly comparable efficiency.

Proof: First use our compilation target to create a mini-assembly program A_P that simulates P. Now A_P is a fixed literal string over the ASCII alphabet as encoded in the handout. We can therefore create U_P to use something like our "shift-over" routine to convert its input x into the string $\langle A_P, x \rangle$, which is just $A_P x$ in the handout, on the first tape. (Or we could use xA_P which would work similarly.) Then we rewind the head on the first tape and send control to the start state of the fixed program U we wrote in Theorem 1. Then for any x,

$$U_P(x) \;\simeq\; U(A_P x) \;\simeq\; A_P(x) \;\simeq\; P(x),$$

where the "\simeq" symbol $\simeq$ says the computations must give the same result if they converge but allows both to diverge, as of course can happen.

To address efficiency, note first that P and U_P use basically the same volume of memory registers, so the *space* usage is about the same. The TM U_P, however, needs extra time because it does not enjoy true *random access*---it has to scroll up and down the whole line of registers to find a desired one. The length s of that line of registers, however, includes only the ones that have been previously allocated, and those allocations used at least s steps of P. The linear search by U_P compounds that time by an $O(s)$ factor per next instruction of A_P. Since s is also at most the total time t taken by P up to a given point, the total compounded time is $O(t^2)$. There is, however, a further complication: The "shift-over" routine may need to be invoked to make more room after repeated addition, for instance. The literal code in my handout would bump up the time to $O(t^3)$ or maybe even $O(t^4)$, depending on implementation details. However, in 1972, Stephen Cook and Robert Reckhow of Toronto worked out a clever caching scheme by which, if P uses the *fair-cost* time measure, which charges for the lengths of the operands in a RAM instruction, then the time goes back down to $O(t^2)$. ☒

The import is, simply: **Turing machines have the same computing power as high-level programming languages, likewise the same power as the machines on which they run**. This is the main concrete evidence in support of the following. Alonzo Church had earlier defined notions of "recursive" and "r.e." via logical schemes of recursion, before Alan Turing's famous 1936 paper proved his machines equivalent to them. Church became Turing's PhD advisor at Princeton in 1937--38; I met him when he received an honorary doctorate from UB in 1990.

The Church-Turing Thesis (three-part version):

1. Any HLL that will ever be devised will have the same computing power as the Turing machine.
2. Any physical device that will ever be built---even quantum computers---will have no more computing power than a Turing machine.

3. For any human being H who follows a consistent functional procedure to convert (sensory) inputs x into outputs y, there exists a Turing machine M_H that on the same inputs x (under a natural string encoding, e.g., pixels for optical input) outputs the same values y. Moreover, M_H has comparable program size and efficiency to the "grey matter" of H, or better.

Plank 1 is often considered a "truism" but maybe it depends on plank 2, which survived a "quantum scare" from David Deutsch at Oxford in 1985 and is even more in play when we bring time-efficiency into the picture. Plank 3 is the philosophically controversial one; the program and memory size S needed is the threshold that "The Singularity" talks about. The "Part Deux" of the C-T thesis is often ascribed to Alan Cobham and Jack Edmonds from papers they wrote in 1965, in which they justified **polynomial time** as a benchmark for feasible problem-solving.

Polynomial-Time C-T Thesis: As above, plus the assertion that whatever the HLL and/or device physically implementing its programs, there will always be a constant k such that whatever the program/device does in time t can be emulated by $O(t^k)$ steps of the Turing machine.

This was also almost-universally believed until 1994, when Peter Shor proved that quantum computers can factor n-digit numbers in $\widetilde{O}(n^2)$ time (idealized---no one has yet built quantum technology that can *scale up*), whereas the security of most Internet commerce and many other cryptosystems relies on concrete scaling of the belief that factoring requires roughly $2^{\Omega(n^{1/3})}$ time, well maybe $2^{\Omega(n^{1/4})}$ or $2^{\Omega(n^{1/5})}$ time in most cases... [Cf. the 1992 movie *Sneakers* and the novel *Factor Man*.] Just last year, a team led by Google *claimed* building a quantum chip capable of achieving a quantum-specific task beyond the reach of classical hardware in under 10,000 years.

But as long as we stick with "classical" machines---meaning non-quantum hardware---we can take both theses as given. (Note: Actually, transistors and other chip elements *are* quantum devices, but the point is that they treat information in the classical manner of *bits*, as opposed to *qubits*.) The import is:

The classes REC, RE, *and co*-RE, *and later* P, NP, *and co*-NP, *remain the same whenever we transfer their defining notions to any HLL or classical machine model. Moreover, it is perfectly legitimate to describe Turing machines via pseudocode, provided the pseudocode gives enough detail to pin down the running time t within a linear $O(t)$, a quasi-linear $\widetilde{O}(t)$, or at worst a polynomial $t^{O(1)}$, factor.*

For example, the 2-tape TM we built to recognize $\{a^m b^n : m = n\}$ can be described by saying, "Copy leading a's to tape 2, then count against b's on the rest of tape 1, and accept iff the counts are equal and the end is reached on tape 1 without any further a appearing. Runtime: $O(m+n)$ steps, which is linear in the length $m+n$ of the input."

Theorem 2 also allows us to shortcut proofs of two other theorems:

Theorem 3: We can build a *universal Turing machine*, that is, a DTM U_T such that, given any transparent encoding $\langle M, x \rangle$ of a DTM M and an input string x over the alphabet of M, U_T on input $\langle M, x \rangle$ simulates the computation $M(x)$.

Proof: The *Turing Kit* is a Java program T that takes inputs $x' = \langle M, x \rangle$ where M comes from a .tmt file (a "Turing Machine Text" file) and x is an input to M likewise encoded via ASCII characters, and simulates $M(x)$, giving the same output if $M(x) \downarrow$. By Theorem 2 we can compile T first to A_T in our mini-assembly and then to U_T such that:

$$U_T(\langle M, x \rangle) \simeq U_T(x') \simeq A_T(x') \simeq T(x') \simeq M(x). \boxtimes$$

Note, incidentally, that this also neatly handles the issue of M being allowed any number of tapes whereas U_T has a fixed number---well, the U_T in my handout has 3 tapes but we will see we can cut it to 1 tape, on pain of (only) another quadratic overhead in running time, so Cook and Reckhow's $O(t^2)$ overhead bumps up to $O(t^4)$---but even that is still *polynomial time*.

Theorem 4: For every NTM N we can build a DTM U such that $L(U) = L(N)$.

Proof: We can imagine extending the *Turing Kit* to T' so that given an NTM N and an input x to N, it does an open-ended loop $t = 1, 2, 3, 4, \ldots$ and for each t it tries all possible ways N can execute t steps on x. If any such way is found to reach the q_{acc} of N, then T' prints "String accepted" and halts. Now given any N, we can fix it so that we have a Java program $T'_N(x)$. Now convert T'_N to a DTM U as in Theorem 3. Then $U(x) \simeq T'_N(x)$, so U accepts x if and only if N accepts x, thus $L(U) = L(N)$. $\boxtimes$

This shortcut proof, however, somewhat conceals the detail in more customary NTM-to-DTM proofs of how the simulation may have to fan out exponentially over sequences of IDs of N. That still happens underneath the words "all possible ways" in this proof. The point of appealing to the *Turing Kit* is that it is easier to imagine writing the control structure for this process in Java rather than directly into "Turing machine code." The point is that the running time $u(n)$ of the U you get is definitely **not** bounded by a polynomial in $t = t(n)$. Whether a faster simulation can be done, so that $u(n) = t(n)^{O(1)}$ for all N, is exactly the famous P = ? NP question, which anchors the second half of this course. But we can say that every NTM N accepts a c.e. language, and once we agree on a notion of "total" for NTMs, those give the same class of decidable languages.

[The transit to Monday will probably be here.]

This all also means we do not have to be picky about how languages of practical *decision problems* are formalized. We can specify the problem in a format popularized by the 1983 book *Computers and Intractability* by Michael Garey and David Johnson, known as "Garey & Johnson":

A_{DFA} (the "Acceptance Problem for DFAs):
Instance: $\langle M, x \rangle$, where M is a DFA and x is an input to M
Question: Does M accept x?

For any decision problem named in this fashion, the language is the set of syntactically-proper instances for which the answer to the Question is *yes*. We can make the name of the problem do double-duty as the name of the language, so as a language,

$$A_{DFA} = \{\langle M, x \rangle : M \text{ is a DFA and } x \in L(M)\}.$$

Proposition: A_{DFA} is decidable.

Proof (by algorithm): Given $x' = \langle M, x \rangle$, simply simulate $M(x)$, and accept x' if and only if M accepts x. Since a DFA always halts, the simulation always halts, so our algorithm is total. ☒

Now we consider a harder problem---but we've talked about it before.

A_{NFA} (the "Acceptance Problem for NFAs):
Instance: $\langle N, x \rangle$, where N is an NFA and x is an input to N
Question: Does N accept x?

If we only care about decidable-versus-undecidable, the following algorithm is fine:

1. Convert N into an equivalent DFA M
2. Now apply the algorithm for A_{DFA} on $\langle M, x \rangle$, and accept if and only if it accepts.