CSE491/596 Lecture Wed. 10/11/23: Computations and (Un)Decidability

Let $\vec{c}$ encode sequences of possible IDs of a (single-tape) Turing machine. We can write

$$\vec{c} = \langle I_0, I_1, I_2, \ldots, I_t \rangle$$

Definition: This is a **valid accepting computation** of a TM M (NTM allowed) on some input x if:

1. $I_0 = I_0(x)$ which is the starting ID on input x. Single-tape TM: $I_0(x) = sx$.
2. For all j, $1 \leq j \leq t$, $I_{j-1} \vdash_M I_j$.
3. I_t is an accepting ID. With "good housekeeping", $I_t = q_{acc}1$.

A **valid halting computation** allows I_t to be a halt-and-reject ID, such as $q_{rej}0$.

Definition: For any TM M (NTM allowed), $VALCOMPS_M$ denotes the language of valid accepting computations of M. (Some sources say "...of valid halting computations of M.") And

$$VALCOMPS = \{\langle M, x, \vec{c} \rangle : \vec{c} \in VALCOMPS_M \wedge \vec{c} = \langle I_0(x), \ldots \rangle\}.$$

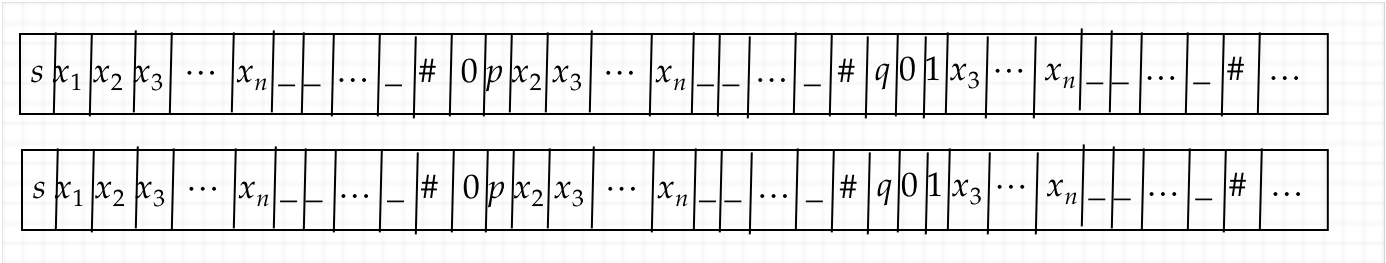
In my earlier notes and many sources, $VALCOMPS$ is called the **Kleene T-predicate**---after the same Stephen Kleene who proved the central theorem about regular languages.

Theorem: Given any fixed Turing machine $M = (Q, \Sigma, \Gamma, \delta, \_, s, F)$, we can build a two-tape deterministic Turing machine H_M that decides $VALCOMPS_M$ in linear time.

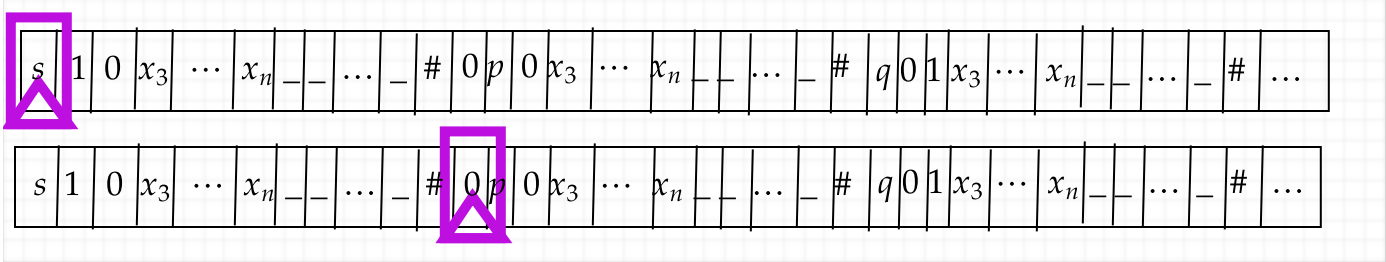
Here the time is reckoned in the length of $\vec{c}$. If $\vec{c} = \langle I_0, I_1, I_2, \ldots, I_t \rangle$ is valid to begin with, and if we suppose t is at least as big as the length n of the input x in I_0, then the length is $O(t^2)$.

[I have taken extra time to include two concrete steps as examples in the proof.]

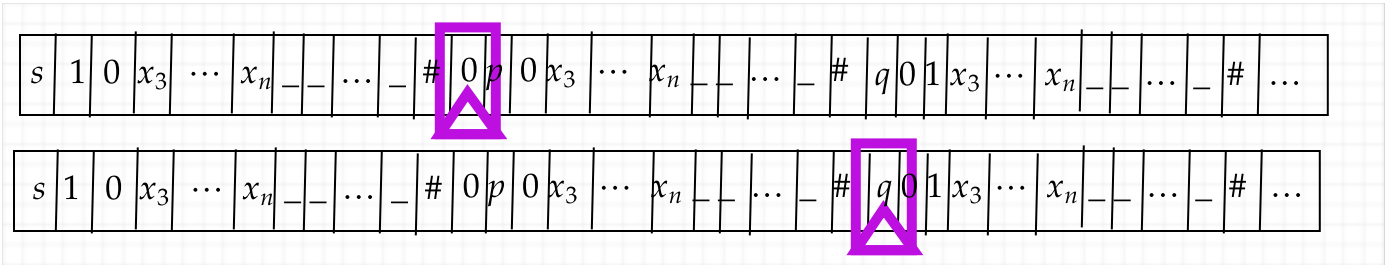
Proof: We may suppose that $\vec{c}$ is encoded as a string over the alphabet $\Gamma' = Q \cup \Gamma \cup \{\#, \%\}$, where # is used to terminate IDs, rather than comma to separate them, and % is used to represent blanks inside IDs. Note that since the state set Q of M is finite, it is completely legitimate to make it part of a bigger alphabet. (And ultimately, any alphabet Γ' like this can be re-encoded via strings over $\{0, 1\}$.) We build $H_M = (Q', \Sigma', \Gamma', \delta'. \_, s', F')$, where Σ' is Γ' minus the actual blank _, as follows. H_M first copies the purported computation given on its input tape to its second tape like so:



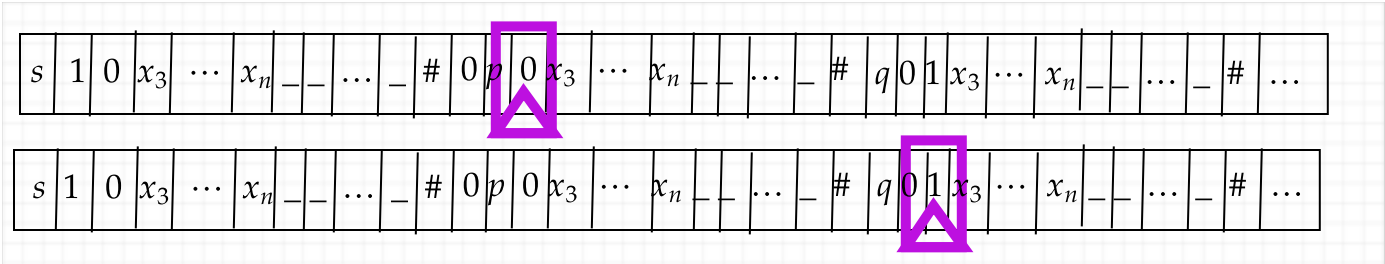
For the concrete example, let us suppose $x_1 = 1$ and $x_2 = 0$ are the first two bits of the input x. Then M is shown executing the instructions $(s, 1/0, R, p)$ and $(p, 0/1, L, q)$. They leave M in state q scanning the 0 that it wrote in the first step. To begin verifying the results of these instructions, H_M rewinds its first tape head to the left and advances its second tape head to the cell after the first # separator---here I have filled in $x_1 = 1$ and $x_2 = 0$:



The point is that in this and the next steps, the heads will read the s and 1 from the first tape and the 0 and p from the second tape. These contents, plus the p being to the right of where the s was, uniquely specify the instruction $(s, 1/0, R, p)$ as the only one that could have been executed. H_M itself is hard-coded to allow only the combinations specified in turn by the instructions of M. Beyond the limited range of that instruction, the characters in the IDs must be identical, since no change took place there. The heads of H_M verify that in lockstep until they hit the next pair of # signs and move one cell further forward:

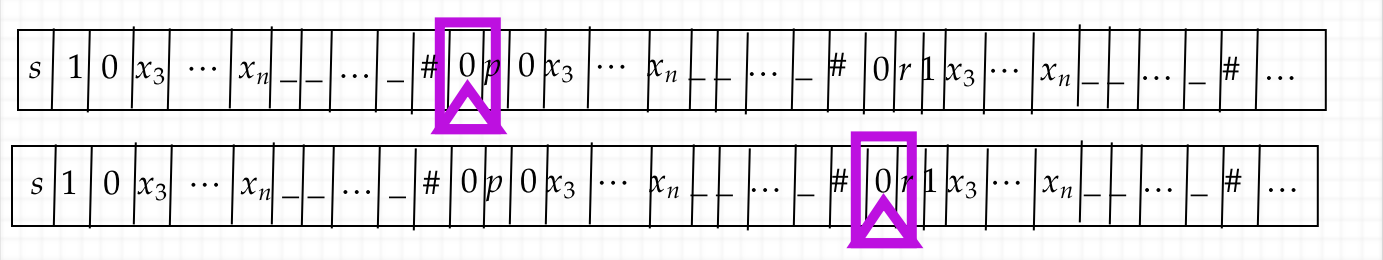


Well, they need to move two cells more:

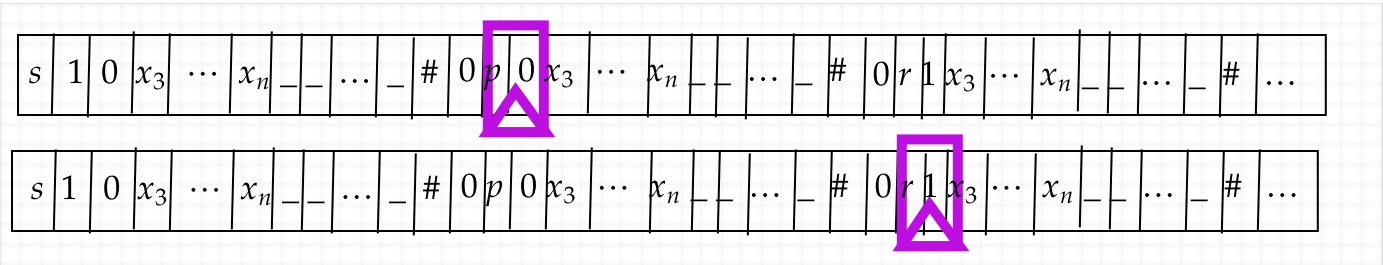


At this point, H_M remembers that it has seen $0p0$ on tape 1 and $q01$ on tape 2. This says that the instruction must have been $(p, 0/1, L, q)$. This instruction is present, so H_M allows it and moves on. If it were absent, H_M would immediately reject that combination as an "illegal move." H_M will accept a trace string $\vec{c}$ if and only if all of the computation steps check out.

If the instruction had been $(p, 0/1, S, r)$ with a stay move instead, then the tapes at the after-# point would look like this:



Here H_M would already know that the two scanned 0s could not have been part of any instruction, because we are using the convention of putting the state to the left of the character scanned by the tape head. They are thus part of the "inert character matching" between the IDs that just happens to come before rather than after the two-or-three-char region where the tape head of M is. Since the two 0s matched (if one or the other was a 1 this would be a mismatch for immediate rejection), the heads move on:



The combination $p0$ and $r1$ is correct for the instruction $(p, 0/1, S, r)$. If M is nondeterministic, then both instructions would be allowed. Note that this doesn't make H_M nondeterministic---rather, its input (i.e., the computation trace $\vec{c}$) would change from being the first form with q to being the second form with r and different happenings in nearby cells of the tape.

[From here on, back to the lecture as I gave it.]

Now we can simply cut the long story short and say the two heads of H_M zip straight across left-to-right and check the whole computation-trace string $\vec{c}$ in one swoop. At the end, H_M accepts if its leading head scanned the state q_{acc} in the final step. This is just three passes through the input $\vec{c}$, so this takes linear (hence polynomial) time. ☒

Application to Undecidability

Not only can we build H_M from any given M, but if we are also given a specific input x to M, then we can build a machine $H_{M,x}$ which includes the extra check that the first ID has the same x as its nonblank string. The **key observation** is:

- If M accepts x, then it has a computation trace $\vec{c}$ on x that $H_{M,x}$ accepts, so $L(H_{M,x}) \neq \emptyset$.
- If M does not accept x, then there is no trace that $H_{M,x}$ accepts, so $L(H_{M,x}) = \emptyset$.

For the upshot, recall that according to Sipser's uniform naming scheme for computational problems about automata, the problem NE_{TM} is: "Given a TM R, is $L(R) \neq \emptyset$?"

Theorem: The problem NE_{TM} is undecidable. Indeed, it remains undecidable even when are told in advance that the TM R is deterministic and runs in linear time.

Proof: Given an arbitrary instance $\langle M, x \rangle$ of the Acceptance Problem A_{TM}, we can build $H_{M,x}$ as above. If we could decide the NE_{TM} problem for these machines --- i.e., decide whether $L(H_{M,x}) \neq \emptyset$ --- then we could decide whether M accepts x. But that is undecidable. So NE_{TM} is undecidable. ☒

In my older notes, this kind of thing is formalized as a many-one reduction from A_{TM} to NE_{TM} with the reduction function f being $f(\langle M, x \rangle) = \langle H_{M,x} \rangle$. I have a way of drawing these reductions as little pictures that draw Turing machines like old-style flowcharts. For us this year, it is enough to note a few more cases of this kind of dependent reasoning:

Theorem: The language of the E_{TM} problem is co-c.e. and undecidable.

Proof: The NE_{TM} *language* is computably enumerable, as it is the language of a nondeterministic TM that on input $\langle R \rangle$ guesses an x and $\vec{c}$, builds $H_{R,x}$, and accepts if and only if $H_{R,x}$ accepts $\vec{c}$). The E_{TM} language is basically the complement of the NE_{TM} language, so ti is co-c.e. And it is likewise undecidable, since the class of decidable languages is closed under taking complements.

Now for a problem/language that is "even more undecidable" than these:

Theorem: The language $ALL_{TM} = \left\{ \langle M \rangle : L(M) = \Sigma^* \right\}$ is undecidable. Furthermore, it is neither c.e. nor co-c.e.

Proof: Note that because the machines $H_{M,x}$ are total (indeed, they run in linear time), the trick of swapping the states q_{acc} and q_{rej} works. The resulting machine $H'_{M,x}$ always recognizes the complment of the language $L(H_{M,x})$. Pulling in the key observations from above, we now have:

- M accepts $x \implies L(H_{M,x}) \neq \emptyset \implies L(H'_{M,x}) \neq \Sigma^* \implies \langle H'_{M,x} \rangle \notin ALL_{TM}$.
- M does not accept $x \implies L(H_{M,x}) = \emptyset \implies L(H'_{M,x}) = \Sigma^* \implies \langle H'_{M,x} \rangle \in ALL_{TM}$.

Thus if we could decide ALL_{TM}, we could decide A_{TM} --- which we cannot do. Technically, we have shown that the language ALL_{TM} is not c.e. either, because the function $f'(\langle M, x \rangle) = \langle H'_{M,x} \rangle$ constitutes a mapping reduction from *the complement of* A_{TM} to ALL_{TM}.

[I stopped the discussion of undecidability here, without stating or proving the "Furthermore" part, but it is not hard to complete it. We need one more idea, which my previous-year notes call the "All-or-Nothing Switch": Given any $\langle M, x \rangle$ pair, define M'_x to be a deterministic TM that has x hard-wired inside it and otherwise mostly contains the code of M. Whatever input string w you formally give M'_x, it just erases w and uses its hard-coded states to write x in its place. Then it goes to its M code. The upshot is:

- M accepts $x \implies M'_x$ accepts any $w \implies L(M'_x) = \Sigma^* \implies \langle M'_x \rangle \in ALL_{TM}$.
- M does not accept $x \implies M'_x$ does not accept any w
  $\implies L(M'_x) = \varnothing \implies L(M'_x) \neq \Sigma^* \implies \langle M'_x \rangle \notin ALL_{TM}$.

This incidentally provides another way of showing that E_{TM} is not c.e., since it reduces from the complement of A_{TM}. But most of note, it does the reduction to ALL_{TM} with the yes/no outcomes switched. So it reduces the original A_{TM} language to the ALL_{TM} language, which bears the consequence that the ALL_{TM} language is not co-c.e. either. So it is *neither c.e. nor co-c.e.* ☒

I briefly mentioned that the "Halting Problem", while defined as "given M and x, does $M(x)$ halt?", is considered synonymous with the Acceptance Problem. Here is a supplement to this remark that pulls in other things I've said, so I regard it as at the top level of course notes.]

One more important problem is whether a given Turing machine halts for *all* inputs. This is the problem that confronts us with the machine shown for the Collatz $3n + 1$ problem. The corresponding language can be written symbolically as $\{\langle M \rangle : (\forall x)\, M(x) \downarrow \}$ and called $TOTAL_{TM}$ (I actually shorten this to TOT). This is in fact virtually the same language as ALL_{TM}: If we want to know whether a given Turing machine M has $L(M) = \Sigma^*$ then---presuming M is in "good housekeeping form"---we can create an "evil" machine M'' by adding arcs to make a never-halting loop at q_{rej}. Then M'' is total if and only if $L(M) = \Sigma^*$ (so that the rejecting state is not reached by any input). Going the other way, if you want to know whether a given Turing machine M is total, make an "angelic" machine M''' that routes all arcs of M that go to q_{rej} to go to q_{acc} instead. Then $L(M''') = \Sigma^*$ if and only if M is total. So the languages ALL_{TM} and $TOTAL_{TM}$ mapping reduce to each other.

Moreover, the mappings are so simple that historically, "halting" and "accepting" were considered identical concepts. This is also why accepting states are called "final" states. (I personally believe that keeping the concepts of "accepting" and "halting" separate is more illuminating.)

Application to NP

If we put a time limit on computations, then this translates into a limit on the length of possible computation strings $\vec{c}$ that we need to consider. The key case is where the time limit t is polynomial in

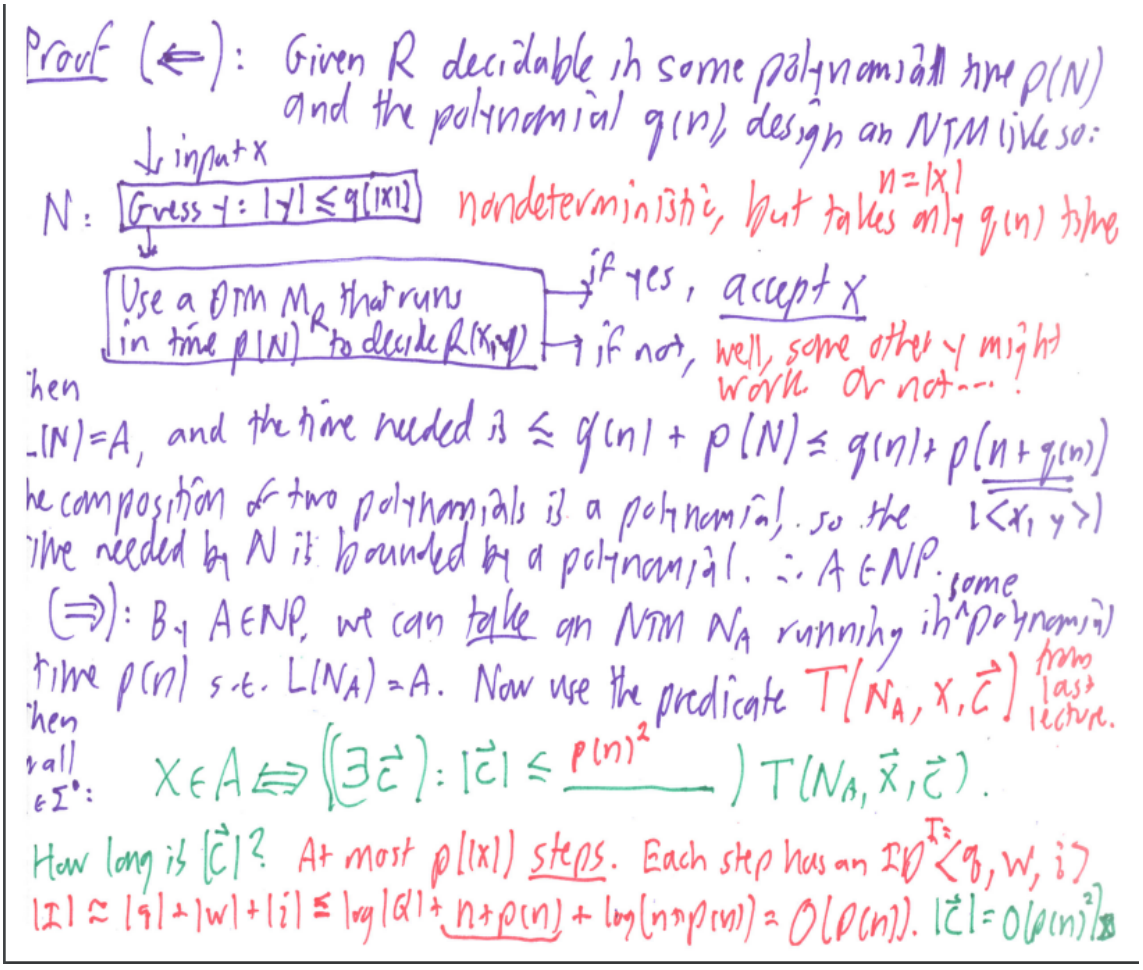
the length n of the string "x" involved, that is, $t = n^{O(1)}$.The following theorem draws the parallel between the cases with-and-without a polynomial time limit.

Theorem (connecting Theorems 10.2 and 13.12 in Debray's notes): For any language L,

- L is c.e. if and only if there is a decidable predicate $R(x, y)$ such that for all $x \in \Sigma^*$,
$$x \in L \iff (\exists y \in \Sigma^*) R(x, y).$$
- $L \in$ NP if and only if there are a polynomial-time decidable predicate $R(x, y)$ and a polynomial $q(n)$ such that for all $x \in \Sigma^*$,
$$x \in L \iff (\exists y \in \Sigma^* : |y| \leq q(|x|)) R(x, y).$$

In both cases, $R(x, y)$ can be linear-time decidable; indeed, R can be the Kleene T-predicate $T(N, x, \vec{c})$ as defined also for NTMs. (The key difference is the polynomial length bound q, not R.)

In the lecture I wrote "Kleene T" as $VALCOMPS(N, x, \vec{c})$ instead. A sketch of the proof using the older "T" notation:



[I wrapped this into the subsequent lecture. I also briefly drew a picture of the idea of stacking IDs vertically to enable conversion to Boolean circuits, because "checking computations" was the theme of the lecture, but I covered that in full in the subsequent lectures.]