$Z_{TM} = \{\langle M\rangle \mid M$ is a TM over $\Sigma = \{0\}$ ~~1}~~,
and M accepts on all strings ~~containing 0~~ $\}$

<u>Puzzle</u>: show Z_{TM} is undecidable by reducing from A_{TM}.

(That is: Assume we have a decider M_Z for Z_{TM},
and use it as a subroutine in a decider for
$A_{TM} = \{\langle M, w\rangle \mid M \text{ accepts } w\}$.)

(Decider for A_{TM})

GOAL: $M(w)$ accepts?
CAN DO: TM $\in Z_{TM}$?

"On input $\langle M, w\rangle$:

// Use M_Z to figure out if $M(w)$ accepts.

// Want: TM T such that $\langle T\rangle \in Z_{TM} \iff M(w)$ accepts

- Define a TM T such that:

~~On input $x \neq 0$: accept.~~

~~On input 0~~ On any input x, simulate $M(w)$ and accept if $M(w)$ accepts.

// If $M(w)$ accepts, $L(T) = \Sigma^*$. $T \in Z_{TM}$, $M(w)$ accepts
If $M(w)$ doesn't accept, $0 \notin L(T)$. $T \notin Z_{TM}$, $M(w)$ doesn't accept.

- Simulate $M_Z(\langle T\rangle)$ and accept if and only if the simulation accepts."

Thus, given M_Z, we can construct a decider for A_{TM}. This is a contradiction, because we know A_{TM} is undecidable.

$\therefore$ our assumption is false and M_Z cannot exist:
Z_{TM} is undecidable.

Take 2:

$ALL_{TM} = \{\langle M \rangle \mid M \text{ accepts all strings}\}$.

Assume M_2 decides ALL$_{TM}$

$M_2(\langle T \rangle)$ tells us if T accepts all strings.

($A_{TM} = \{\langle M, \omega \rangle \mid M(\omega) \text{ accepts}\}$).

To decide A_{TM}, define:

$M_{A_{TM}}$ = "On input $\langle M, \omega \rangle$:

- Let T be a TM that ignores its input, simulates $M(\omega)$, and accepts if $M(\omega)$ accepts.
- Write $\langle T \rangle$ on our tape, but don't simulate it.

((If $M(\omega)$ accepts, T accepts everything.))

- Run $M_2(\langle T \rangle)$, which tells us if T accepts all strings $\iff M(\omega)$ accepts.
- Accept if $M_2(\langle T \rangle)$ accepts."

$M_{A_{TM}}$ decides A_{TM}, which is a contradiction. Thus deciding ALL_{TM} is not possible.

Today: Complexity.

1. Complexity vs. computability (Big-O review)
2. The class P
3. The class NP.

Computability ≈ whether a TM can do something

- recognize, deciding languages.
- computing a function

Def. A function $f: \Sigma^* \to \Sigma^*$ is computable if there exist a TM that, for any input string $w \in \Sigma^*$, halts with $f(w)$ on the tape.

Complexity $\approx$ the quantity of resources a TM takes to do something.
- time (number of steps/transitions).
- space (number of tape cells).
- randomness (bits/# of coin flips).
- quantumness (qbits/quantum ops).
- communication (messages exchanged between parties, queries to some information source.)

Def. (TM running time). If M is a deterministic TM that always halts, the running time/time complexity of M is the maximum number of steps that M takes on any input of length n. (A function, $f(n)$).

Example: does the input contain 0?

N = "On input w, of length $|w| = n$:
- scan input left to right, and accept if we see an 0.
- otherwise, reject at the end of the string."

- Takes "roughly" n steps. ($n+10$, $n-1$).
  (Do we care about factors less than n?)
- $|w| = n$ matters — input length almost always important.

Back at 2:13

Big-O review:

Let f, g be positive functions, increasing in n.

- $f(n) = O(g(n))$ if there exist positive integers n_0, c such that

$$f(n) \le c \cdot g(n) \text{ for all } n \ge n_0.$$

$f < g$, up to multiplication, in the long run.

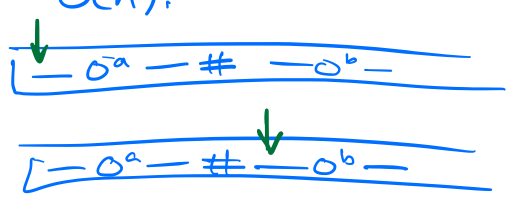
$$O(1) \le O(\log(n)) \le O(n^{0.1}) \le O(\sqrt{n}) \le O(n) \le O(n^3)$$
$$\ll O(2^n).$$

$n^{1000} < 1.0001^n$, for n really big.

$$\text{Dup} = \{\underbrace{0^a \# 0^b \# 0^c \ldots}_{m \text{ strings of } 0\text{'s}} \mid \text{there exist two } 0 \text{ substrings of the same length}\}.$$

Regular TM: shuttle back and forth $O(n)$ times, $O(n)$ distance, for all $O(m^2)$ pairs of 0 substrings to check for duplicates.
$O(n^2 \cdot m^2)$.

2-tape TM: copy the input, and check two substrings in time $O(n)$.



$O(n \cdot m^2)$

(2-tape NTM)
NTM: Nondeterministically branch into $O(m^2)$ branches, each of which checks a different pair of 0 substrings. (Accept if any branch accepts). $O(n^2) \to O(n)$

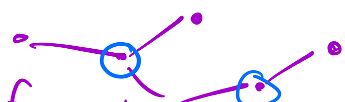
NTM's ability to "guess a solution" makes them really good at problems that involve "guess and check."

SUDOKU $= \{\langle S\rangle \mid S$ is a sudoku puzzle (nxn generalization) and S has a correct solution.$\}$.

NTM can guess all possible solutions and check.

SALESMAN $= \{\langle G, t\rangle \mid$ G is A graph of cities with edges labeled by distance, and some tour of the graph visits every city and traverses a path of length at most t.$\}$.

Vertex Cover: Given a graph find a subset of the vertices of size at most t that is adjacent to every vertex.



Subset Sum: Given a set of numbers, is there any subset that adds up to a target t?

Def. P is the class of languages that a TM can decide in polynomial time.

$O(n^k)$ for some fixed k, also written "poly(n)".

$n^{1000} \ll 1.00001^n$.

Note: poly(n)-time deciders can use each other as subroutines and stay poly(n).

Example: M_1 runs in time $O(n^c)$

M_2 "calls" M_1 $O(n^d)$ times:

total runtime is $O(n^c \cdot n^d) = O(n^{c+d})$.

Takeaway: P = poly(n)-time decidable
= "efficiently" decidable languages.

Languages in P:

- regular languages $\subseteq P$.
  (logic: any DFA takes n steps, so an equivalent TM takes $O(n)$ steps.)
- CFLs $\subseteq P$ (proof slightly harder.)
- searching, arithmetic $(+, \times, \div, -)$, basic problems on sets, graphs, (element distinctness, connectivity...).

Back at 3:00

$\{0^n 1^n \mid n \geq 0\}$.

NP.

Def 1. NP is the class of languages that a NTM can decide in time poly(n).

Def 2. NP is the class of all languages in which every string can be "verified" by a short "proof."

Def A Verifier for a language L is a deterministic TM V_L such that, for every $w \in L$, there exists a 'proof' or 'certificate' string c that makes $V_L(\langle w, c\rangle)$ accept.

Def 2 (formal). NP is the class of languages that have a poly(n)-time verifier V.

Takeaway: NP $\approx$ efficiently verifiable languages.

Theorem: Def 1 (languages an NTM can decide in time poly(n)) and Def 2 (languages with a poly(n)-time verifier) are equivalent.

Proof.

$\Longrightarrow$. Say L is decided by a poly(n)-time NTM N_L.

- Some branch, or series of choices/transitions, causes N_L to reach an accept state in time poly(n).
- Our verifier, V_L, will verify a string $w \in L$ when given a certificate c that summarizes the choices made by the accepting branch of N_L.

V_L = "On input $\langle w, c \rangle$:

- Simulate $N_L(w)$ but <u>only</u> the one branch indicated by c.

(Our simulation is deterministic.

- Accept if and only if our simulation accepts."

$\Longleftarrow$. Say L is a language with a poly(n)-time verifier V_L. Thus, for each $w \in L$, there exists a certificate c such that $V_L(\langle w, c \rangle)$ accepts in time poly(n).

Build an NTM that decides L by nondeterministically guessing c, and simulating $V_L(\langle w, c \rangle)$ for every possible c on a different branch. Accept if any branch accepts.

(* w.l.o.g., we can assume $|c| = \text{poly}(n)$ – otherwise, V_L can't read it.)

P = efficiently decidable by TM.

NP = efficiently decidable by NTM OR efficiently verifiable.

$P \neq NP$, right? Unproved.

($P \subseteq NP$.)

Next time: elaborate on NP

NP-completeness: "being at least as hard as any problem in NP."

Course evals — bring device.

Snacks.

Segue into final review.

Final:

Problems reproduced on final:

HW 2, 2.

HW 3, 1.

HW 3, 2.

HW 4, 2.

$$L_1 = \{0^a \# 1^a \# 2^{2a} \mid a \geq 0\}$$

Goal: Show L_1 is not CF using the CFPL.

- Assume for contradiction that L_1 is CF.
- Thus the CFPL applies: the CFPL states that there exists a number p, such that for all $w \in L_1$, $|w| \geq p$,

  there is some way to divide w into 5 pieces $w = uvxyz$, such that:

  (1) $uv^ixy^iz \in L$, for any $i \geq 0$.

  (2) $|vy| > 0$,

  (3) $|vxy| \leq p$.

- Now we'll pick a specific string $w \in L_1$, $|w| \geq p$, and show that no way of dividing w into $uvxyz$ satisfies 1, 2, and 3.

$w = 0^p \# 1^p \# 2^{2p}$. ($w \in L$, $|w| \geq p$. $|w| = 4p+2$.)

Case 1: either v or y contains a #.

Now: cond'n (1) says $uv^2xy^2z \in L$ $(i=2)$. But this string repeats v and y, so it has at least 3 #'s. X

(For any $i \neq 1$, uv^ixy^iz has too many/too few #'s.)

Case 2: v and y contain more 0's than 1's,
(or more 1's than 0's.)

Then, uv^2xy^2z has more 0's than 1's (more 1's than 0's).

Case 3: v and y contain the same number of 0's and 1's.

(3a) ↳ v and y contain at least one 1, and one 0.

Claim: v is a substring of 0's, and y is a substring of 1's.
v comes before y.
case 1 is ruled out, so v, y contain no #'s.

Now: uv^2xy^2z has more 0's + 1's than 2's.

(3b) ↳ v and y contain no 0's and no 1's.
Because we're not in case 1, v and y have no #'s,
and thus v, y contain only 2's.
$|vy| > 0$ (condition 2) implies v, y contain at least one 2.
So uv^2xy^2z has more 2's than 1's + 0's.