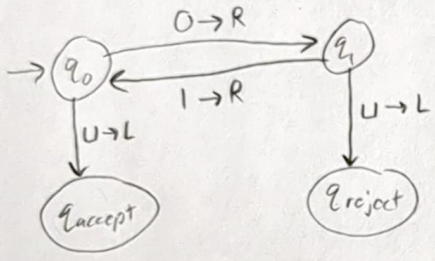
Even when languages are decidable in theory, they may not be so in practice. What good does it do to know a problem *can* be solved if it takes 50 years to do it? This is where we turn to computational complexity, a study of resource requirements (time, space, etc) for algorithms.

Consider the language $L = (01)^*$. We can give a TM or algorithm for it as it's obviously decidable (it's regular).



M = "On input w,
1) Sweep right
2) If a 1 follows a 1 or a 0 a 0, reject.
3) If w ends with 0, or starts with 1, reject
4) Accept."

If we ask how long it takes to decide an input w, it's clear that we visit each cell of w at most once, so the time $t(w)$ satisfies $t(w) \leq |w| + 1$ (+1 for a ⊔ cell). This is a worst-case runtime, but we could analyze average or best case as well or even a more exotic runtime like amortized time. Let's first give a precise definition of *worst-case* runtime.

Let M be a deterministic TM which halts on all inputs. The *runtime* or *time complexity* of M is the function $f: \mathbb{N} \to \mathbb{N}$, where $f(n)$ is the maximum number of steps M uses on any input of length n. We say that *M runs in $f(n)$ time* and that M is an $f(n)$ time TM. By custom, n is the input length.

This is useful, but functions don't compare well to each other in raw form. We need a way to classify them.

Consult CSC 301 - Asymptotics.

Now that we have a way to classify functions, we can work with the classes more easily. Speaking of, let's translate runtimes into languages.

Let $t: \mathbb{N} \to \mathbb{N}$ be a function. The <u>time complexity class</u> <u>$TIME(t(n))$</u> is the set of all languages decidable by a TM in $O(t(n))$ time.

Ex) Consider the language $L = \{0^k 1^k \mid k \geq 0\}$.

M_1 = " On input w,
1) Sweep right and reject if there is a 0 right of a 1 or a 1 left of a 0
2) Return to the start
3) Sweep right, marking a 0 and a 1
4) If neither were found, accept
5) If one but not the other were found, reject
6) Goto 2"

How long does M_1 take? Why?

$O(n^2)$. We sweep over the string of length n $\frac{n}{2}+2$ times, which takes time $O(n \cdot (\frac{n}{2}+2)) = O(n^2)$.

Is this optimal? That is does L <u>require</u> $\Omega(n^2)$ time or is it $o(n^2)$ time?

Here's an algorithm which decides M_2 asymptotically faster.

M_2 = "On input w,

1) Sweep right and reject if there is a 0 right of a 1 or a 1 left of a 0
2) Return to the start
3) While at least one 0 and at least one 1 remain
   a) Sweep right and reject if the # of unmarked inputs is odd
   b) Return to the start
   c) Sweep right, marking every other 0 then every other 1
   d) Return to the start
4) If no 0's or 1's remain, accept
5) Reject"

With M_1, we reduced the number of unmarked inputs by 2 each time (one 0 and one 1). Here we <u>divide</u> the number of unmarked inputs by 2. So instead of $O(n)$ loops, we must instead perform only $O(\log n)$ loops. M_2 the takes $O(n \log n)$ time and $L \in TIME(n \log n)$. This is optimal.

what if we have <u>two</u> tapes? Can we do better?

We can!

M_3 = " On input w (on tape 1),

1) Sweep right and reject if there is a 0 right of a 1 or a 1 left of a 0.
2) Return to the start, copying all 1's onto the second tape
3) Sweep right on both tapes and accept if #0's = #1's.
4) Reject"

In this model of computation, $L \in TIME(n)$! The model matters!

However, all reasonable models of computation are polynomially equivalent to each other. That is $\exists p \forall L$, $L \in TIME(f(n))$ in model A $\Rightarrow$ in model B, $L \in TIME(p(f(n)))$. For example, consider the following theorem which we will state without proof.

Thm) Let $t(n)$ be a function. Then for every $t(n)$ time TM, there is an equivalent $O(t(n)^2)$ time k-tape TM ($k \geq 1$).

In this theorem, $p(n) = n^2$.

Now there is a notion of non-deterministic time as well, and it will be important, but let's stick with deterministic time first. In particular, let's look at the class P.

Polynomial time algorithms are, generally speaking, efficient. They exploit some structure of a problem to avoid a brute force search for a solution. Consider sorting. If we want to brute force the problem, there are $n!$ permutations of a set of n elements. Determining how to permute your input via brute force to get to the identity permutation (i.e. to sort your input) thus takes $O(n!)$ time. Alternatively, we know that $\leq$ is transitive, which makes sorting way easier since we can just sort 1 element at a time.

Another reason why poly time algorithms are important is because we said all reasonable models of computation are polynomially equivalent to each other. So if L is polytime in one model, then L is in any other model.

Now let's formally define the set of polytime languages P.

$$P = \bigcup_{k \in \mathbb{N}} TIME(n^k).$$

So P is the set of all languages L for which there is a TM M that decides L in $O(n^k)$ time for some $k \in \mathbb{N}$.

Let's look at some examples of languages in P.

Ex) $PATH = \{\langle G,s,t\rangle \mid G$ is a directed graph with a directed path from s to t$\}$.

We give a polytime TM M for PATH.

M = "On input $\langle G,s,t\rangle$, ← reject if the input is in the wrong format

0) If $s=t$, accept
1) Let $G=(V,E)$
2) Let $L=V$ be a list of unvisited vertices
3) Let P be an empty path
4) Put s into P and remove s from L
5) While L is not empty
   a) Let u be the last vertex of P
   b) If $u=t$, accept
   c) Otherwise pick a neighbor v of u in L
   d) If no such v exists,
      i) Remove u from P
      ii) If P is empty, reject
   e) Else
      i) Add v to P
      ii) Remove v from L
6) Reject"

M is performing a DFS. Conventionally, we would call this algorithm $O(V+E)$, but with the TM model of computation we have to account for scanning the tape and how we represent vertices and edges since we don't have RAM or registers. Even so, the TM is polynomially equivalent, so we're good.

Here's a less conventional way to do this algorithm that makes more sense in the TM model. Suppose E is given to us as a $|V| \times |V|$ bit matrix with value 1 at (i,j) if $(i,j) \in E$ and 0 if $(i,j) \notin E$.

M = "On input $\langle G, s, t \rangle$,
0) If $s = t$, accept
1) Let $G = (V, E)$
2) Let $L = 0^{|V|}$ and set bit s to 1 (marked)
3) Repeat until no bit is marked
   a) Scan E
      i) If $(i,j) \in E$ has i marked and j unmarked, mark j
      ii) If $j = t$, accept
4) Reject"

This algorithm scans E at most V times, and we can track L on a second tape efficiently, so with an ordinary TM, the runtime is

$$O\left((V \cdot V^2)^2\right) = O(V^6).$$

Ex) Every regular language is in P.

pf) There is a DFA D for any regular language L.
D decides L in linear time(1 input / 1 transition).
We can simulate D by tracking the state and looking up transitions (a roughly constant time operation since D is a finite object [we may have to transition across our input]).
The exact runtime depends on implementation, but it's probably $O(n^3)$.

Ex) $RELPRIME = \{\langle a,b \rangle \mid \gcd(a,b) = 1\}$

E = "On input $\langle a,b \rangle$,
1) Repeat until $b = 0$
   a) Assign $a = a \bmod b$
   b) Swap a and b
2) Output a"

R = "On input $\langle a,b \rangle$,
1) Run E on $\langle a,b \rangle$
2) If E returns 1, accept
3) Reject"

E reduces a,b by at least half every other time, so E loop $O(n)$ times $\Rightarrow$ R runs in poly time.

Here's a list of other cool P languages.

- $\{\langle p \rangle \mid p \text{ is prime}\}$
- L such that L is context-free
- $2SAT = \{\langle \phi \rangle \mid \phi \text{ is a satisfiable 2-cnf-formula}\}$

  Ex) $\phi(x_1, x_2, x_3) = (x_1 \vee x_2) \wedge (\overline{x_1} \vee x_3) \wedge (\overline{x_2} \vee x_3)$
- $MINDFA = \{\langle D \rangle \mid D \text{ is a minimum state DFA}\}$
- $CYCLE = \{\langle G \rangle \mid G \text{ contains a cycle}\}$
- $ECYCLE = \{\langle G \rangle \mid G \text{ contains a Euler cycle}\}$

  visit each edge exactly once
- $SUM = \{\langle x \rangle + \langle y \rangle = \langle z \rangle \mid x, y, z \in \mathbb{N} \text{ and } x + y = z\}$