18.404: Theory of Computation

Ace Chun

December 13, 2025

Contents

Any analysis of computability must start with a rigorous mathematical formulation of what a computer actually *is*, and what it is able to do. Once given, we can begin to derive what is and isn't possible, as well as what kinds of problems are harder to solve than others.

1 Regular Languages

Regular languages are the simplest class of languages studied in this course.

1.1 Deterministic Finite Automata

Regular languages are defined to be those recognized by Deterministic Finite Automata (DFAs). DFAs are defined by a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where

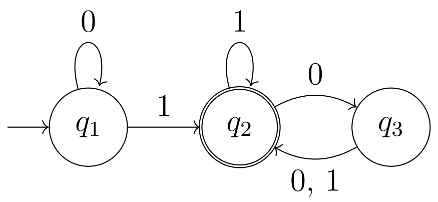
1. Q is a finite set of states.
2. Σ is a finite set of characters, defining the *alphabet* of the language.
3. $\delta : Q \times \Sigma \to Q$ is the *transition function.*
4. $q_0 \in Q$ is the *start state.*
5. $F \subseteq Q$ is the set of *accepting states.*

DFAs are essentially *pattern matchers.* For an input string $w = w_1 w_2 \cdots w_n$, a DFA M will read each character in sequence and transition along its states by following rules outlined by its transition function δ. If, after reading w_n (which is when M *terminates execution*), we end in a state $q_i \in F$, we say that w is *accepted* by M, and is a member of the language of M (notated $L(M)$). Otherwise, w is *rejected* by M, and so $w \notin L(M)$.

For example, let M_1 be a DFA such that

$$\begin{aligned}
Q &= \{q_1, q_2, q_3\} \\
\Sigma &= \{0, 1\} \\
\delta &: (q_1, 0) \mapsto q_1,\ (q_1, 1) \mapsto q_2 \\
&: (q_2, 0) \mapsto q_3,\ (q_2, 1) \mapsto q_2 \\
&: (q_3, 0) \mapsto q_2,\ (q_3, 1) \mapsto q_2 \\
q_0 &= q_1 \\
F &= \{q_2\}
\end{aligned}$$

We can draw this DFA as a state diagram:



For a DFA, the transition function must be uniquely defined for *every pair* $(q, a) \in Q \times \Sigma$ — every state must have outgoing arrows for each symbol in the alphabet that tell the DFA what to do next when it scans the string, and each destination state for such a pair must be unique.

Regular languages can be finite or infinite, depending on the existence of loops in the DFA that recognize it. Note that all finite languages are trivially regular, as we can recognize them by simply having one state and path for each distinct string in the language.

1.2 Nondeterministic Finite Automata

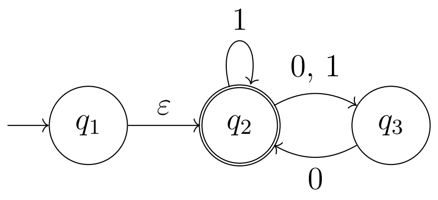
In contrast to DFAs, Nondeterministic Finite Automata (NFAs) loosen the constraint that each destination state must exist and be unique for every (q, a) pair. Instead, an NFA can map one (q, a) pair to multiple different states, which indicates the branching possibilities one could take when scanning the string. Formally, NFAs are defined very similar to DFAs; they are defined by a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where

1. Q is a finite set of states.
2. Σ is a finite set of characters, defining the *alphabet* of the language.
3. $\delta : Q \times \Sigma_\varepsilon \to \mathcal{P}(Q)$ is the *transition function.*
4. $q_0 \in Q$ is the *start state.*
5. $F \subseteq Q$ is the set of *accepting states.*

Note the change in the transition function; instead of simply mapping to the set of states, we now map to the *power set* of states; any (q, a) pair may link to any subset of our states, not just a single state. For example, if $\delta(q_1, 1) = \{q_0, q_3\}$, this means that, when we scan a 1 while in state q_1, we may either choose to go to q_0 or q_3, leading to two alternate branches (or "alternate universes") where we simulate the next scan starting from each of $\{q_0, q_3\}$.

With nondeterminism, we say that a string is accepted if *any* path through the state machine leads to an accepting state — we just need one such path out of the many branches that may arise from each decision made. Implicitly, a string is rejected on a particular branch if, on that branch, we reach a state that does not have a valid transition with the next character being scanned (we reach a "dead end").

In addition, transitions can now include the empty string ε, which corresponds to moving between states *without* scanning a symbol from the string — the symbol Σ_ε is shorthand for $\Sigma \cup \{\epsilon\}$, accounting for this.



1.2.1 Equivalence with DFAs

While it seems like NFAs are more powerful than DFAs by virtue of being able to utilize nondeterminism, the two models of computation are *equivalent* in power and capabilities. In general, two machines (that could be of different models of computation) are equivalent if they recognize the same language.

Therefore, to show that NFAs are equivalent to DFAs, we just need to show that the language of any NFA N can also be described by some DFA M, which demonstrates that the NFAs recognizes the same class of languages as DFAs. (We would also need to prove the reverse, which is that a DFA can be described by an NFA. However, this is more trivial, as an NFA is simply a generalized version of a DFA with added capabilities.)

The hint lies in the transition function. The only difference between DFAs and NFAs are that NFAs map to $\mathcal{P}(Q)$, rather than Q itself.

Consider an NFA $N = (Q_N, \Sigma, \delta_N, q_{0N}, F_N)$. We can construct a DFA $M = (Q_M, \Sigma, \delta_M, q_{0M}, F_M)$ such that each of its states correspond to a subset of Q_N.

$$Q_M = \mathcal{P}(Q_N)$$

For every transition, we wish to map $\delta_N(q_1, a)$ to the state corresponding to the set of all of its possible destinations. In addition, given a state in M, we would like to map it to the state containing all of the possible ways the states in M could transition, having read the symbol a. Therefore, for $S \subseteq Q_N$,

$$\delta_M(S, a) = \bigcup_{s \in S} \delta_N(s, a)$$

For ε-transitions, i.e., for every (q_1, S) pair such that $\delta(q_1, \varepsilon) = S$, we union S with the set of outgoing arrows from states containing q_1.

$$\delta_M(S, a) = \bigcup_{s \in S} \{\delta_N(s, a) \cup \delta_N(s, \varepsilon)\}$$

For our starting state, similarly, we would like to simulate starting from q_{0N}:

$$q_{0M} = \{q_{0N}\} \cup \{q \in Q \big| q \in \delta_N(q_{0N}, \varepsilon)\}$$

Finally, we designate all subsets that contain an accepting state in N to be an accepting state in M.

$$F_M = \bigcup_{f \in F_N} \{s \big| s \subseteq Q_N, f \in s\}$$

This completes our construction of an equivalent DFA for a given NFA. We have not actually changed any logical content of the transitions of the machine; we've simply shifted around and changed the labels on the states and transitions we are dealing with.

1.3 Regular Operations

There are three basic operations that we may perform on languages (though they are set operations that can be applied to languages in general). Given languages A and B,

1. **Union:** $A \cup B = \{x \big| x \in A \vee x \in B\}$
2. **Concatenation:** $A \circ B = AB = \{xy \big| x \in A,\ y \in B\}$
3. **Star:** $A^* = \{x_1 x_2 \cdots x_k \big| \{k \geq 0,\ x_i \in A\}$

We can demonstrate all of these operations in terms of constructions on DFAs, which therefore proves that regular languages are *closed* under all three of these operations.

Given machines $M_1 = (Q_1, \Sigma, \delta_1, q_{10}, F_1)$ and $M_2 = (Q_2, \Sigma, \delta_2, q_{20}, F_2)$, their *union* can be constructed by taking advantage of nondeterminism; we nondeterministically simulate M_1 and M_2, and if either machine accepts some input, we accept. Formally, $L(M_1 \cup M_2)$ is recognized by the machine

$$M = (Q_1 \cup Q_2, \Sigma, \delta_{12}, q_0, F_1 \cup F_2)$$

where δ_{12} preserves all of the transitions from M_1 and M_2 and adds an additional transition

$$\delta_{12}(q_0, \varepsilon) = \{q_{10}, q_{20}\}$$

To recognize the concatenation of the languages of M_1 and M_2, we want to determine if $w = xy$, where $x \in L(M_1)$ and $y \in L(M_2)$. We first pass an input w through M_1, and nondeterministically choose where the "break" in w is, i.e., where x transitions into y. x will transition into y at an accept state of M_1, so we add ε-transitions into M_2. If the nondeterministically determined y portion is accepted by M_2, $w = xy \in L(M_1) \circ L(M_2)$. This language is therefore recognized by the machine

$$M = (Q_1 \cup Q_2, \Sigma, \delta_{12}, q_{10}, F_2)$$

where δ_{12} preserves all transitions from M_1 and M_2, and adds

$$\delta_{12}(q, \varepsilon) = q_{20} \ \forall q \in F_1$$

The star construction works in a very similar way. If $w \in L(M_1)^*$, $w = x_1 x_2 \cdots$, where each $x_i \in L(M_1)$. Here, instead of nondeterministically finding a single transition point, we wish to "find" and recognize boundaries between each $x_i x_{i+1}$ segment. To do this, we add a nondeterministic ε-transition back to the start state of the machine, q_{10}. In addition, we append a new start state that also functions as an accepting state in order to recognize ε, and link it with a ε-transition to q_{10}.

1.4 Regular Expressions

Regular expressions are the *generative* counterpart to DFAs — that is, a regular expression generates the strings of a regular language given some specified production rules, whereas a DFA just recognizes and accepts them. The two are equivalent in expressive power, and can be proven to be such.

Regular expressions are constructed in terms of the regular operations. Formally, we define them recursively: R is a regular expression if R is one of the following:

1. a for some $a \in \Sigma$
2. ε
3. $\emptyset$
4. $(R_1 \cup R_2)$ for regular expressions R_1 and R_2
5. $(R_1 \circ R_2)$ for regular expressions R_1 and R_2
6. (R_1^*) for regular expression R_1.

The first three conditions correspond to our "base cases" — they are the most atomic-level descriptions a regular language could be. The latter three correspond to our "recursive cases" — they describe how to build larger regular expressions out of smaller ones. This is known as an *inductive definition.*

We can start from any symbol in our alphabet (or, the empty symbol or set) and repeatedly apply the regular operations to generate regular expressions that describe new languages.

1.4.1 Equivalence with automata

Regular expressions are equivalent to the automata model of computing, despite seeming like a fundamentally different approach to forming a language. The class of languages described by any regular expression are the regular languages. We can show this by establishing an equivalence between finite automata and regular expressions, which requires us to provide constructions for an equivalent finite automaton M given a regular expression R, and vice versa.

1.5 Pumping Lemma

To determine if a language is not regular, we keep track of an invariant known as the *Pumping Lemma*, which we know must hold for all regular languages.

The pumping lemma states that, if A is a regular language, then there exists some number p (called the pumping length) such that, for any $s \in A$ with $|s| \geq p$, $s = xyz$ such that

1. $\forall i \in \mathbb{N},\ xy^iz \in A$
2. $|y| > 0$
3. $|xy| \leq p$

The pumping lemma is frequently employed in proofs by contradiction. If we wish to prove that some language B is non-regular using the pumping lemma, we can follow the proof skeleton:

> Suppose, for the sake of contradiction, that B is regular. Then, by the pumping lemma, there must exist some pumping length p such that, for any $s \in B$ with $|s| \geq p$, $s = xyz$ that satisfies:
>
> 1. $xy^iz \in B$.
> 2. $|y| > 0$.
> 3. $|xy| \leq p$.
>
> We then construct an adversarial s that cannot be pumped (i.e. be modified into xy^iz) and remain in B — s serves as a counterexample to demonstrate that the pumping lemma does not hold for B, and therefore, B cannot be regular. This step depends on the definition of B, so it will vary from language to language.

Crucially, finite automata cannot really deal with holding *memory*, or a record of what has previously been read while processing a string, outside of some finite set of properties(which can be recorded in states). This presents a fundamental limitation to the expressive power of regular languages. For example, the language

$$A = \{0^n1^n \big| n \geq 0\}$$

is not regular. This can be shown by proving that any string in A cannot be pumped and still remain in A. Intuitively, however, we can discern that A is not regular by the fact that some finite automaton would require a way of keeping track of the number of 0s processed and somehow match them to each 1 scanned in the second half of the string, which is a capability that a finite automaton has.

1.6 Closure properties

Regular languages are closed by definition under all of the regular operations; that is, if a regular operation is performed on regular language(s), the resulting set is also a regular language. In addition, regular languages are closed under:

1. **Complement:** $A^c = \overline{A} = \{x \big| x \notin A\}$
2. **Intersection:** $A \cap B = \{x \big| x \in A \wedge x \in B\}$

For a machine $M = (Q, \Sigma, \delta, q_0, F)$, $\overline{L(M)}$ is recognized by

$$M' = (Q, \Sigma, \delta, q_0, Q - F)$$

Given $M_1 = (Q_1, \Sigma, \delta_1, q_{10}, F_1)$ and $M_2 = (Q_2, \Sigma, \delta_2, q_{20}, F_2)$, $L(M_1) \cap L(M_2)$ is recognized by

$$M_{12} = (\{q_0\} \cup Q_1 \times Q_2, \Sigma, \delta_{12}, q_0, F_1 \times F_2)$$

where

$$\delta_{12} : ((q_1, q_2), a) \mapsto (\delta_1(q_1, a), \delta_2 q_2, a)$$

and

$$\delta_{12}(q_0, \epsilon) = (q_{10}, q_{20})$$

We can use closure properties to prove certain languages not regular. What ends up being the most helpful is utilizing closure under complement and intersection to "mold" our language into one that is known to be non-regular, which is a contradiction; since regular languages are closed under complement and intersection with other regular languages, any outcome of a sequence of these operations must also be regular, provided that we started with a regular language. If the outcome is not regular, we know that we cannot have started with a regular language.

2 Context Free Languages

The class of context-free languages (CFLs) covers a larger scope than that of regular languages. The two primary models that define CFLs are Pushdown Automata (PDAs) and Context Free Grammars (CFGs).

2.1 Pushdown Automata

Pushdown automata are composed of two parts: a finite control (akin to a DFA) and a stack, which introduces memory capabilities. In this way, PDAs encompass all finite automata; all finite automata can be simulated by a PDA that just ignores its own stack.

A PDA reads through its input string, just as a DFA does. However, at each step, it also has the option to pop and/or push a symbol from its stack, which adds an element of complexity to the states it can describe. Formally, a PDA is defined as a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$ where

1. Q is a finite set of states.
2. Σ is the input alphabet, or the set of all characters that can be in an input string.
3. Γ is the stack alphabet, or the set of all characters that can be pushed onto the stack.
4. $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to \mathcal{P}(Q \times \Gamma_\varepsilon)$ is the transition function.
5. $q_0 \in Q$ is the start state.
6. $F \subseteq Q$ is the set of accepting states.

PDAs that recognize CFLs must necessarily be *nondeterministic*; if, instead, we fix PDAs to be deterministic, they will recognize a smaller class of languages called *Deterministic Context Free Languages.*

2.2 Context Free Grammars

Just like regular expressions are generative counterparts to DFAs, CFGs are generative counterparts to PDAs. A CFG consists of a set of production/-substitution rules, with each rule being comprised of a single *variable* symbol on the left and a string of variables and *terminal symbols* it can be replaced with on the right. For example,

$$S \to 0S1|\varepsilon$$

is a single-rule grammar that describes the language $A = \{0^n1^n \big| n \geq 0\}$.

Formally, a CFG is a 4-tuple (V, Σ, R, S), where

1. V is a finite set of variables.
2. Σ is a finite set (disjoint from V) of terminals.
3. R is a finite set of rules, with each rule taking a single variable to a sequence of variables and terminals.
4. $S \in V$ is the start variable.

Generating a string in the language of the grammar follows the procedure:

1. Write down the start variable S.
2. For each variable in what is currently written, find a rule that starts with that variable. Replace the variable with a the string of variables and terminals corresponding to the right hand side of the production rule.
3. Repeat step 2 until no variables are left in the string, and it is composed of terminals.

The sequence of substitutions that lead to the generation of a particular string is called a *derivation*. Note that the derivation for a string need not be unique; if there are multiple valid derivations for a particular string from a grammar, we say that the string is *ambiguously derived*, and that its grammar is itself *ambiguous*.

CFGs with certain restrictions can be converted into Deterministic CFGs, which similarly produce the language of Deterministic CFLs.

2.2.1 Chomsky Normal Form

Chomsky Normal Form, or CNF, is a convenient way of codifying CFGs. All rules in a grammar in CNF follow the format

$$A \to BC$$
$$A \to a$$

for any variables A, B, C (with B, C not being the start variable) and any terminal a. All CFGs can be converted into CNF via finite construction, so grammars in CNF are equivalent in power to CFGs in general.

CNF is useful because it provides a guarantee on the generating structure of some string from the grammar's production rules. It can be proven that, with a grammar in CNF, a string w can be derived from exactly $2|w| - 1$ steps.

2.3 Context-free Pumping Lemma

Similar to the regular languages, we can define the boundary of what is and isn't a CFL by introducing an invariant, the pumping lemma for context free languages. For a CFL A, there must exist a pumping length p such that, for any $s \in A$ with $|s| \geq p$, s can be divided into $s = uvxyz$ such that

1. $uv^ixy^iz \in A \ \forall i \in \mathbb{N}$
2. $|vy| > 0$
3. $|vxy| \leq p$

Proving that a language B is not context free using the context free pumping lemma follows a very similar format to that of regular languages.

Suppose, for the sake of contradiction, that B is context free. Then, by the pumping lemma, there must exist some pumping length p such that, for any $s \in B$ with $|s| \geq p$, $s = uvxyz$ that satisfies:

1. $uv^ixy^iz \in B$.
2. $|vy| > 0$.
3. $|vxy| \leq p$.

We similarly construct an adversarial s that cannot be pumped and remain in B, and therefore, B is not context free. Note that we no longer have the guarantee that the pumpable portion of s is within the first p characters, like we do with regular languages.

2.4 Closure Properties

CFLs are closed under all of the regular operations $\cup$, $\circ$, $*$. These can be demonstrated easily from CFGs. Given two grammars G_1, G_2 with corresponding start symbols $S_1,\ S_2$, we can construct the rules

$$\begin{aligned} A &\to S_1|S_2 \\ B &\to S_1S_2 \\ C &\to CS_1|\varepsilon \end{aligned}$$

where A is the start symbol of the grammar describing $L(G_1) \cup L(G_2)$, B is the start symbol of the grammar describing $L(G_1) \circ L(G_2)$, and C is the start symbol of the grammar describing $L(G_1)^\star$.

However, CFLs are *not* closed under complement or intersection. However, the intersection of a CFL and a regular language is itself a CFL.

3 Turing Machines

A Turing machine (TM) is a much more powerful model of computation that allows for unrestricted memory and tape access. A Turing machine consists of a finite control, which keeps track of state, and an infinite tape that has a written portion with its *contents* w, and an infinite sequence of blanks $\sqcup$ following it. The finite control has a pointer (also called its *head*) that points to the current symbol being read by the machine.

Crucially, a Turing machine can go back and forth (L, R) on its tape; it does not "lose" any information by advancing forward and reading the tape. In addition, it does not have separate read-only and write-only stores for memory; it can write directly on its input, which is on its singular tape.

Turing machines are the most powerful models of computation that are dealt with in this course, and most metrics of computability are compared against what is capable with a Turing machine.

Formally, a Turing machine is defined by a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_{acc}, q_{rej})$, where

1. Q is a finite set of states.
2. Σ is the input alphabet, the set of all possible characters that constitute an input string.
3. Γ is the tape alphabet, which is the set of all possible characters that can exist on the tape. This includes the blank symbol $\sqcup$.
4. $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is the transition function.

5. $q_0 \in Q$ is the start state.

6. $q_{acc} \in Q$ is the accepting state.

7. $q_{rej} \in Q$ is the rejecting state.

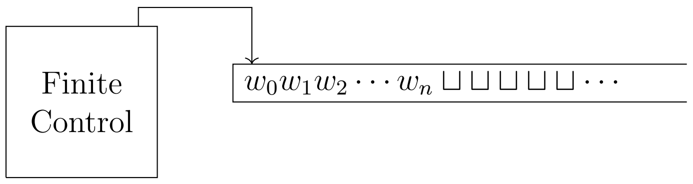
A Turing machine will follow its transition function on a given input until one of the following happens:

1. It enters q_{acc}. At this point, it *halts* and accepts the input string into its language.

2. It enters q_{rej}. Similarly, it halts, but rejects the input string.

3. It keeps looping on the input, never entering a terminal state. If this happens, we say that the Turing machine *rejects by looping.*

If a Turing machine halts on all of its inputs, it is called a *decider*, and its language is *decidable.* Otherwise, its language is just *T-recognizable.*


Finite Control
$w_0w_1w_2 \cdots w_n \sqcup \sqcup \sqcup \sqcup \sqcup \cdots$

We note that, in its current definition, a Turing machine is *deterministic* — each singular configuration leads directly and deterministically to a successive configuration. Unlike PDAs and DPDAs, however, it turns out that deterministic TMs are equivalent in power to nondeterministic TMs.

3.0.1 Configurations

A *configuration* of a Turing machine at a given point over the course of its computation is essentially a written record or snapshot of its entire state at that point. This includes the head position, the tape contents, and the current state of the machine.

For a state q and tape contents uv, where the head is pointing at the first symbol in v, we write the current configuration $C_i = uqv$.

We say that one configuration C_i yields another configuration $C_j$if the machine can go from C_i to C_j with a valid transition in δ.

We say that a configuration is a *start configuration* C_{start} if it represents the starting state of the TM on some input w.

$$C_{start} = q_0 w$$

An accepting configuration is a configuration that contains the symbol q_{acc} somewhere in its string. Similarly, a rejecting configuration is a configuration that contains q_{rej}. Whether or not these configurations are reachable is determined by producing a valid sequence of configurations. In general, C_n is reachable if there exists a sequence $C_{start}, C_1, C_2, \cdots C_n$ such that each C_i yields its successive C_{i+1} for $i \in 1, 2, \cdots, n-1$.

3.1 Equivalent Variants

(Single tape) Turing machines are equivalent to many other models of computation, meaning that they both recognize the same set of languages. The way this is shown is through *simulation*; in effect, for some arbitrary model of computation A, we can show that there is a direct correspondence between a Turing machine M_{TM} and an instance of A, M_A. Note that there are two directions that we must prove.

1. For an arbitrary Turing Machine M_{TM}, there exists an M_A that can simulate the operations of M_{TM} and describe the same language that M_{TM} does. This is shown by constructing an abstracted model M_A that can perform any required operations of a a Turing machine.

2. For any arbitrary instance of the other model of computation M_A, we can simulate all of its operations with a Turing machine. This involves constructing a machine M_{TM} and defining subroutines of atomic Turing machine operations that simulate any capability of M_A.

3.1.1 Multitape Turing Machines

One could ask what happens to the language of a Turing machine if we are allowed more memory stores, i.e., multiple tapes instead of just a singular tape. However, it turns out that adding more tapes to a Turing machine does not add any capability to a Turing machine; it just makes record-keeping more "convenient." In other words, single tape and multi-tape Turing machines are equivalent.

To show this, we require a construction (both ways) from one to the other. In one direction, the construction is trivial; a 1-tape TM is just a special case of a k-tape TM, with $k = 1$.

The other direction requires more work. We can convert any k-tape TM into an equivalent 1-tape TM by concatenating by each of the k tape contents (truncating each before their infinite sequence of $\sqcup$s)with some kind of unique separator symbol (like #). In addition, for each tape symbol $a \in \Gamma$, we add a symbol to the tape alphabet (say, $\dot{a}$) to indicate that a particular head is at that present symbol in its corresponding tape. We require this addition because we must be able to simulate all k heads of a k-tape machine at once when constructing an equivalent 1-tape TM.

Simulating the operations of a k-tape TM, similarly, requires us to *scan through* the entirety of the concatenated tape once to keep track of its current state before any modifications, then sequentially applying desired transitions to each of its k subsections after doing the scan. If, at any point in the j-th tape, we need to utilize its blank portion, we will just shift every section to the right of the j-th simulated tape to the right by one cell to make room. This newly constructed 1-tape TM, though much more *complicated* than a k-tape TM, is logically equivalent, and can simulate any operation that can be done on the k-tape TM; k-tape TMs therefore recognize the same class of languages.

Adding multiple tapes doesn't add any expressive power to our TM, but it does add ease of description and conciseness, which we can leverage in demonstrating certain properties.

3.1.2 Nondeterministic Turing Machines

Using the fact that multi-tape TMs are equivalent to single-tape TMs, we can use them as a throughline for proving that nondeterministic TMs are equivalent to deterministic TMs. In particular, we simulate a *breadth-first search* through the state tree generated by a nondeterministic TM on a particular input (with each branch at any node representing a different path the machine could have taken at a particular configuration). We create three tapes: one for the input string, another for the "simulation" string, and one last tape that effectively functions as a queue for the BFS, keeping track of addresses of configurations (encoded as a sequence $abcd\cdots$, where each symbol corresponds to taking a particular choice at a given node).

When performing the BFS, if we ever encounter an accepting configuration, we accept; if we encounter a rejecting configuration, we just erase its entry from the queue and continue simulating. Notice that this resulting simulation may loop if the origial nondeterministic TM loops on an input, and will halt otherwise — this detail is why we need to do a breadth-first search, as opposed to a depth-first search.

3.1.3 Enumerators

Just like we had generative counterparts for finite automata and PDAs, enumerators function as generators for languages recognized by a TM. Enumerators are TMs with a new device, referred to as a *printer*, attached to it; a string is sent to the printer and printed out whenever indicated by the finite control. The language of an enumerator E is the set of all strings it ever prints out; E may produce an infinite list of strings, in which case its language is infinite.

3.2 Church-Turing Thesis

The reason Turing machines are so important and fundamental is because they formalize the notion of an *algorithm*, or a procedure of steps applied in order to solve a particular problem. Prior to the development of computability theory, an algorithm was more of an intuitive measure for indicating how a particular task or construction was accomplished; however, there was no rigorous way of defining the correctness of an algorithm, or even determine whether or not it was even possible to produce an algorithm to solve a particular problem.

Turing machines provide a mathematical and logically rigorous way to answer these questions. Formally, the Church-Turing thesis states that any function on $\mathbb{N}$ can be calculated via a specified method or algorithm iff it is possible to implement its computation on a Turing machine.

3.2.1 Procedures on objects

At a very high level, we can describe the processes of a Turing machine in terms of a sequence of steps (not unlike how we would descrbe an algorithm in pseudocode). For some object G that we are operating on, we abstract its encoding into a TM-readable string to place on the tape, and we note this as $\langle G \rangle$. $\langle G \rangle$ is the *string representation* of the object G, whether G be a graph, equation, or even another Turing machine. $\langle G \rangle$ contains all of the information needed to describe G and its properties, which we can use in our Turing machine description to determine something about it.

This abstraction allows us to not have to worry about the low level specifics about operating with Turing machines; we only care that a Turing machine can reasonably access a particular property of an object it needs through a sequence of atomic TM operations, which is encapsulated in $\langle G \rangle$.

We describe a Turing machine that operates on a particular object with the skeleton structure:

```
$M =$ "On input $\langle G \rangle$,
    1. List operations on $G$.
    2. State conditions on $G$ for which $M$ accepts or rejects."
```

3.2.2 Computable Functions

A function $f : \Sigma^* \rightarrow \Sigma^\star$ (i.e. a function that maps strings to strings) is *computable* if there exists some Turing machine T that, when given an input x, will execute and halt with $f(x)$ written on its tape.

4 Decidability and Reducibility

As previously delineated, a Turing machine can be a *decider*, meaning that it is guaranteed to halt on all of its inputs, or it may simply *recognize* its language (meaning that it may reject some input w by looping). The trouble with looping is that we can never definitively say whether or not a TM is actually looping, or if it simply hasn't finished its execution on its input; we *cannot* make a decision, therefore, by "testing for looping" (i.e. "If M loops on w, do x" is not a valid instruction).

A language is said to be *decidable* if it is recognized by a decider. A language is said to be *T-recognizable* if it is recognized by any Turing machine. All decidable languages are T-recognizable, but the converse is not true.

4.1 Decision Problems

For a given computational problem, we can formulate its outcomes as a language to determine its computability. If we wish to ask if some object X has a particular property y, we can formulate a language

$$A = \{\langle X \rangle \big| X \text{ has property } y\}$$

If X fits the conditions, $\langle X \rangle \in A$; if not, then $\langle X \rangle \notin A$.

We can then try to formulate a Turing machine M that describes the language A ($L(M) = A$). If A is a decidable language, the problem of determining if X has property y is decidable: there exists a definitive algorithm that can determine this and provides a (correct) answer for any input. If A is not decidable, then we can conclude that there is no algorithm that will give us a "yes/no" answer for any input to the problem.

4.2 Diagonalization

In 1891, Georg Cantor introduced the *diagonalization argument* to show that the set of real numbers $\mathbb{R}$ is uncountable. The general gist of this argument is as follows:

Suppose $\mathbb{R}$ is countable. For simplicity, we'll consider the set of real numbers between $[0, 1)$ — this is the same cardinality as $\mathbb{R}$, which can be shown by projecting a circle of circumference 1 to the rest of the real number line. In addition, we will be representing these numbers as binary decimals — this does not change the cardinality of the set.

Then, there exists some bijective function $f : \mathbb{N} \to \mathbb{R}_{[0,1)}$ — we can enumerate all numbers in $\mathbb{R}_{[0,1)}$ in a numbered list, so that $f(n) \in \mathbb{R}\ \forall n \in \mathbb{N}$.

n	$f(n)$
0	0.010101...
1	0.101010...
2	0.001000...
$\vdots$	$\vdots$

For each $f(n)$, we'll notate its digit at the i'th place after the decimal place is $f(n)[i]$.

Now, we construct a new number x. We set x's n'th digit after the decimal place adversarially, such that

$$x[n] = 1 - f(n)[n]$$

x's nth digit is the opposite of whatever the nth enumerated number's nth digit is — if we read down the diagonal indices of the numbers in the table, x's nth digit is the opposite of the nth diagonal entry.

Now, we try to find an index $y \in \mathbb{N}$ such that $f(y) = x$. However, we have specifically constructed x to contradict any diagonal index of the outputs of f. If $f(y) = x$, that means that $f(y)[y] = x[y]$, but we have already stated that $x[y] = 1 - f(y)[y]$. Therefore, we have a contradiction, and $\mathbb{R}_{[0,1)}$ is uncountable (and so is $\mathbb{R}$).

Now, consider the following language:

$$A_{TM} = \{\langle M, w\rangle \big| \text{the Turing machine } M \text{ accepts } w\}$$

A_{TM} is the language corresponding to the *acceptance problem* for Turing machines. Using a similar diagonal argument, we can prove that A_{TM} is undecidable — there cannot exist a decider H that recognizes A_{TM}.

Suppose A_{TM} is decidable, and is decided by some TM H. By definition, H on some input $\langle M, w\rangle$ will accept and halt if the machine described by $\langle M\rangle$ accepts w, and reject and halt if not.

Now, we construct an adversarial machine D.

D = "On input $\langle M\rangle$,

1. Run the machine H on $\langle M, \langle M\rangle\rangle$.
2. Do the opposite of H — accept if H rejects, and reject if H accepts."

Because we have supposed that H is a decider, D is a machine that we can construct, as we will have a definitive "yes/no" answer on any given input (which we can then flip). D will indicate (the opposite of) whether or not an arbitrary TM M accepts or rejects its own description.

Now suppose that we enumerate all Turing machines $M_1, M_2, \cdots$

and their descriptions. We can place these into a table, with each entry describing whether or not a particular machine M_i accepts (1) or rejects (0) the description of another machine $\langle M_j \rangle$ (i.e. the output of H on the input $\langle M_i \langle M_j \rangle \rangle$).

	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\cdots$	$\langle D \rangle$
M_1	1	0	$\cdots$	1
M_2	1	0	$\cdots$	0
$\vdots$			$\ddots$	
D	0	1	$\cdots$	?

By construction, D will not accept its own description. However, D cannot do the *opposite* of what H outputs on $\langle D \langle D \rangle \rangle$; if H outputs accept, then D would have to reject, and vice versa. Therefore, it is impossible to enumerate and define all outputs of H in this manner, so H cannot exist by contradiction.

Therefore, A_{TM} is undecidable.

4.3 General Reductions

We can prove the undecidability of many languages using the diagonalization argument, simply by constructing some adversarial input. However, this method requires some heavy machinery that isn't always necessary. Instead, we can *reduce* a language to one that is known to be undecidable to prove undecidability.

More concretely, reducing some problem A to B means finding a procedure for solving B that involves knowing how to solve A. We would be asking the question:

"If we know a process for deciding A, can we use this to decide B?"

Or, logically,

$$A \text{ is decidable} \to B \text{ is decidable}$$

If we know B to be undecidable through some other method (i.e. diagonalization), then we know that A cannot be decidable either. Logically, establishing the above statement would also make the contrapositive

$$B \text{ is undecidable} \to A \text{ is undecidable}$$

true. Since we know the antecedent to be true, the consequent must also be true. Put otherwise, A being decidable would introduce a contradiction (since we would conclude that B is simultaneously decidable and undecidable), so A cannot be decidable.

The process for establishing this implication is called a *reduction*. We notate "A reduces to B" as

$$A \leq_T B$$

where $\leq_T$ indicates a *Turing reduction*, otherwise known as a "general reduction."

In most cases, we will take B to be A_{TM}, and A to be the language we are trying to prove undecidable. We want to demonstrate that solving A can be used to solve A_{TM}, and work from there. The skeleton of a reduction proof is as follows:

> Suppose the language A is decided by some decider R. Then, construct a machine S:
>
> > $S =$ "on input $\langle M, w\rangle$,
> >
> > 1. Run R on $\langle M, w\rangle$.
> > 2. Use the output of R to determine whether or not M accepts w, and accept/reject accordingly.
>
> This construction demonstrates that we can map the outputs of R to the acceptance problem for M on w via a finite procedure; we have constructed a decider for A_{TM} out of a decider for A. Therefore, if A is decided by R, then this would mean that A_{TM} is decidable. However, we know this cannot be the case, so A is undecidable by contradiction.

General reductions cannot necessarily be used to demonstrate *unrecognizability*, however; if constructing S depends on negating some output of R, we cannot be certain that this is a valid step, since we would only be supposing that R is recognizable (and not necessarily decidable).

For a language A,

$$A \leq_T \overline{A}$$

as, if A were decidable, we would be able to use the decider for A to decide

$\overline{A}$. Put otherwise,

$$A \text{ is decidable} \to \overline{A} \text{ is decidable}$$

Note that this also implies

$$\overline{A} \text{ is decidable} \to A \text{ is decidable}$$

A and $\overline{A}$ can either both be decidable, or neither can be. In fact, both A and $\overline{A}$ are recognizable iff A and $\overline{A}$ are both decidable; if A is not decidable but recognizable, then $\overline{A}$ is unrecognizable.

If $\overline{A}$ is recognizable, then A is said to be co-recognizable.

4.4 Mapping Reductions

A mapping reduction is very similar to a general reduction, but it can be used to solve broader problems (including proving unrecognizability). We say that A is mapping-reducible to B

$$A \leq_M B$$

if any instance of A can be "translated" into B. Formally, this translation is described in terms of a *computable function* f; that is, if there exists a computable function f on strings x such that

$$x \in A \to f(x) \in B$$

and

$$x \notin A \to f(x) \notin B$$

then $A \leq_M B$. We can summarize this in the following picture:

$$\begin{tikzcd} A \arrow[r, "f"] & B \\ \overline{A} \arrow[r] & \overline{B} \end{tikzcd}$$

Proving a language is undecidable/unrecognizable through a mapping reduction follows the skeleton:

Suppose the language A is decided/recognized by some machine R. Then, construct a machine S:

```
S = "on input $\langle M, w \rangle$,
    1. Construct some input $x$ that can be given to $R$ from $\langle M, w \rangle$:
       define a computable function such that $f(\langle M, w \rangle) = x$.
    2. Feed this input $x$ into $R$.
    3. Conclude whether or $M$ accepts $w$ by using the output of
       $R$ on $x$."
```

Similarly to general reductions, this construction demonstrates that we can map the outputs of R to the acceptance problem for M on w. We can translate the acceptance problem into a problem statement for A, and then used the machine for A to determine A_{TM}. We can do this, as the mapping f gives us a relationship between the strings in A and A_{TM} (and, conversely, the relationship between strings in $\overline{A}$ and $\overline{A_{TM}}$).

For languages A and B such that $A \leq_M B$, we have the following properties:

1. $A \leq_M B \rightarrow \overline{A} \leq_M \overline{B}$
2. If B is decidable/recognizable/co-recognizable, then so is A.
3. If A is undecidable/unrecognizable/un-co-recognizable, the so is B.

In effect, if $A \leq_M B$, then B is a "harder problem" than A.

4.5 Computation History Method

The above methods of reducibility, to some extent, require us to deal with models of similar computational power to Turing machines. However, if we want to prove the undecidability of a language that has to do with a *weaker* computational model (like a PDA, for example), then we cannot necessarily just attempt to do some simulation of a Turing machine as a part of our reduction. Instead, the computation history method allows us to encode the acceptance problem of M on w into a string, which can be read by any model of computation.

Recall that we can encode successive configurations of a Turing machine M on an input w. We then define a computation history as a *finite sequence* of configurations of M on w (crucially, if M loops on w, then there is no computation history for M on w). A valid computation history $C_0, C_1, C_2, \cdots, C_\ell$ follows the criteria:

1. $C_0 = q_0 w$ is a valid starting configuration for M on w.
2. For each $i \in 0, 1, 2, \cdots \ell$, each successive configuration is derived from a valid transition. That is, each C_i yields C_{i+1}.
3. C_ℓ is a valid terminal configuration; it contains either q_{acc} or q_{rej}.

If C_ℓ contains q_{acc}, then we say that the computation history is accepting.

Typically, we will encode a computation history by demarcating each configuration with a special symbol, like #.

$$w = \#C_0\#C_1\# \cdots \#C_\ell\#$$

Since w is just a string, and there are very simple and checkable conditions w must follow in order to be an accepting computation history for M on w, we can use this method to determine emptiness/acceptance problems for weaker models of computation.

4.6 Examples

4.6.1 Decidable Languages

Let $E_{DFA} = \{\langle B\rangle \big| B \text{ is a DFA where } L(B) = \emptyset\}$.

E_{DFA} is decidable. We construct a Turing machine $S =$ "on input $\langle B\rangle$,

1. Mark the start state of B.
2. Repeat until no new states are marked, or all states of B have been checked:
   (a) Mark any state reachable from any state that has already been marked.
   (b) If an accept state is marked, reject on $\langle B\rangle$.

3. If the process terminates without having marked an accept state, accept on $\langle B \rangle$."

Let $EQ_{DFA} = \{\langle A, B \rangle \big| A, B$ are DFAs, and $L(A) = L(B)\}$.

EQ_{DFA} is decidable. We construct a Turing machine $S =$ "on input $\langle A, B \rangle$,

1. Construct a DFA C that describes the *symmetric difference* of $L(A)$ and $L(B)$: $L(C) = (L(A) \cap \overline{L(B)}) \cup (\overline{L(A)} \cap \overline{L}(B))$. $L(C)$ is regular due to closure properties.
2. Run the decider for E_{DFA} on C to determine whether $L(C) = \emptyset$.
3. If $L(C) = \emptyset$, accept — there are no strings in $L(A)$ that aren't in $L(B)$, and vice versa.
4. If $L(C) \neq \emptyset$, reject.

Let $A_{CFG} = \{\langle G, w \rangle \big| G$ is a CFG, $w \in L(G)\}$.

A_{CFG} is decidable. We construct a Turing machine $D =$ "on input $\langle G, w \rangle$,

1. Convert G to Chomsky normal form to produce an equivalent grammar G'; there exists a finite construction/algorithm that performs this conversion.
2. Recall that all strings of length n can be derived in exactly $2n-1$ substitutions. We can therefore test every derivation of $2|w| - 1$ steps from G' in finite time.
3. If any derivation matches w, accept. Otherwise, reject."

Let $E_{CFG} = \{\langle G \rangle \big| G$ is a CFG, $L(G) = \emptyset$.

E_{CFG} is decidable. We construct a Turing machine $E =$ "on input $\langle G \rangle$,

1. Mark all terminal symbols.
2. Repeatedly mark a given variable S if there exists a production

rule

$$S \to a_1 a_2 \cdots a_\ell$$

where each a_i are symbols that have been marked.

3. Repeat step 2 until no new variable symbols are marked.

4. Reject if the starting symbol was not marked. Otherwise, accept."

Crucially, the above language tests for the reachability of any symbol consisting only of terminals (i.e. any string of terminals in the languageof G) from the start symbol.

4.6.2 Undecidable Languages

Let $HALT_{TM} = \{\langle M, w\rangle \big| M \text{ halts on } w\}$.

Suppose $HALT_{TM}$ were decidable and recognized by a decider R. We construct the following machine $S =$ "on input $\langle M, w\rangle$,

1. Run R on $\langle M, w\rangle$. If R rejects, then reject.

2. If R accepts, M must halt on w. Simulate M on w and accept if M accepts, and reject if M rejects."

S is a decider for A_{TM} that uses the existence of R as a subroutine. However, we know that A_{TM} is undecidable via other methods, so S cannot be a decider, and therefore, R cannot be a decider. $HALT_{TM}$ is undecidable.

Let $E_{TM} = \{\langle M\rangle \big| L(M) = \emptyset\}$.

Suppose E_{TM} is decidable, and it is decided by R. We construct $S =$ "On input $\langle M, w\rangle$,

1. We first define the following machine M_W:

M_w = "on input x,

(a) If $x \neq w$, reject.

(b) Else, run M on x. If M accepts, accept; else, reject."

2. We run R on $\langle M_w \rangle$.
3. If R accepts, $L(M_w)$ is empty, so w was not accepted. Reject.
4. If R rejects, accept.

S decides A_{TM}, but we know that A_{TM} is undecidable. Therefore, R cannot be a decider.

The key part of this proof is that we construct a machine M_w whose properties depend on M accepting w; however, we never actually *run* M_w itself, so we do not find ourselves in danger of looping.

4.6.3 Unrecognizable Languages

Let $EQ_{TM} = \{\langle M, N\rangle \big| L(M) = L(N)\}$.

We can show that $\overline{A_{TM}} \leq_M EQ_{TM}$ by providing some computable function f, such that

$$\langle T, w\rangle \in \overline{A_{TM}} \longleftrightarrow f(\langle T, w\rangle) = \langle M, N\rangle \in EQ_{TM}$$

Given $\langle T, w\rangle$, we provide the following machines

M = "on input x, always reject."

N = "on input x

1. Ignore x and run T on w.
2. Reject x if T rejects, and accept if T accepts.

$L(M) = L(N) = \emptyset$ if $\langle T, w\rangle \in \overline{A_{TM}}$.

We can similarly demonstrate $\overline{EQ_{TM}}$ is unrecognizable constructing the same N and by defining

```
$M =$ "on input $x$, always accept."
```

4.7 Recursion Theorem

The recursion Theorem states that a Turing machine M can obtain a representation of itself, $\langle M \rangle$, and then compute with it. Formally, the statement of the theorem says that, for any machine T that computes some function $t : \Sigma^* \times \Sigma^* \to \Sigma^*$, there exists a machine R that computes a function r such that

$$r(w) = t(\langle R \rangle, w)$$

On any input w to R, R can obtain its own description and pass it to T, which carries out the computation on w provided with the additional information of R's description.

The proof of this theorem depends on the existence of some computable function $q : \Sigma^* \to \Sigma^*$, where $q(w)$ will print out the description of some machine P_w that just prints out w on its tape and halt. Q computes q:

```
$Q =$ "on input $w$,
    1. Construct the following machine:
        $P_w =$ "on input $x$,
            (a) Erase $x$ from tape.
            (b) Write $w$ on the tape.
            (c) Halt."
    2. Write $\langle P_w \rangle$ on the tape and halt."
```

The recursion theorem can be used to formulate diagonalization proofs in a more concise manner. For example, take the following proof of the Halting theorem:

Suppose $HALT_{TM}$ were decidable and recognized by some machine H. Construct the following machine:

```
$R =$ "on input $w$,
    1. Obtain $\langle R \rangle$.
    2. Run $H$ on input $\langle R, w \rangle$.
    3. Accept and halt if $H$ rejects, and continue if $H$ accepts."
```

R loops if H says it will halt, and halts if H says it will loop. By contradiction, H cannot decide $HALT_{TM}$.

5 Time Complexity

Time complexity theory aims to measure and bound the amount of time (or the number of time steps) required to solve a given problem on an input of size n. Conventionally, definitions for time complexity are written in terms of the number of steps required for a *single tape* Turing machine to run on an input of size n, though, for the complexity classes we care about exploring, this choice of model (over, say, a multi-tape Turing machine) does not end up making an enormous difference.

5.1 TIME

Suppose M is a deterministic, single tape Turing machine decider. We say that the *running time* or *time complexity* of M is some function $f : \mathbb{N} \to \mathbb{N}$ if M uses a maximum of $f(n)$ time steps to decide any input of length n.

Asymptotically, we say that M runs in $O(t(n))$ time if $f(n) \in O(t(n))$ for some $t : \mathbb{N} \to \mathbb{R}^+$. Formally, this means that there exists a positive integer c and n_0 such that

$$\forall n \geq n_0,\ f(n) \leq c \cdot t(n)$$

$t(n)$ is the asymptotic upper bound of $f(n)$, and the notation above is called big-O notation. There is a related notion of small-o notation: for functions $f(n)$ and $g(n)$, $f(n)$ is said to be in $o(g(n))$ if

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0,\ f(n) < c \cdot g(n)$$

Big-O notation effectively signals a "$\leq$" comparison between functions, whereas small-o notation signals a strict "$<$" comparison.

We define the time complexity class $TIME(t(n))$ to be the set of all languages A that can be decided by an $O(t(n))$ time Turing machine. A language is in $TIME(t(n))$ if there *exists* some $O(t(n))$ time Turing machine that decides it.

5.1.1 Multitape Conversion

As mentioned previously, the constraint of defining runtimes in terms of single tape machines does not significantly change the power of what can be analyzed. The procedure of every $t(n)$ time multitape Turing machine can be run on an equivalent single tape Turing machine in $O(t^2(n))$ time.

Let M be a k-tape Turing machine that runs in $t(n)$ time. We can construct a single tape Turing machine S that runs with equivalent outputs as M in $O(t^2(n))$ time.

Recall that the single tape equivalent S that simulates a k-tape Turing machine M runs by storing the contents of each of the k tapes consecutively, on a single tape, with an expansion to the tape alphabet to keep track of the boundaries between tapes and the locations of the k different heads. For each step of M, the head of S scans over its entire tape to first record the current state of the simulated M. S then performs a second pass over its tape to carry out the subsequent operations, updating its tape and simulated head positions.

We can bound the size of S's tape by first considering that the maximum length of each of M's tapes must be at most $t(n)$ (as each tape will have size $t(n)$ if its corresponding head keeps taking rightward steps). As k is a constant, the length of S's tape is therefore bounded by $O(t(n))$.

S performs roughly two passes over its tape per time step of M, so it takes $O(t(n))$ time per step. As M runs for $t(n)$ steps, then, the total operation of S simulating M therefore takes $t(n) \cdot O(t(n)) = O(t^2(n))$ time.

5.2 NTIME

Consider N, a nondeterministic Turing machine decider that halts on all computational branches on any given input. We say that N runs in time $f(n)$ if N uses a maximum of $f(n)$ steps for every branch on any input of length n.

Similarly, we define the complexity class $NTIME(t(n))$ to be the set of all languages A that can be decided in $O(t(n))$ time by some nondeterministic Turing machine.

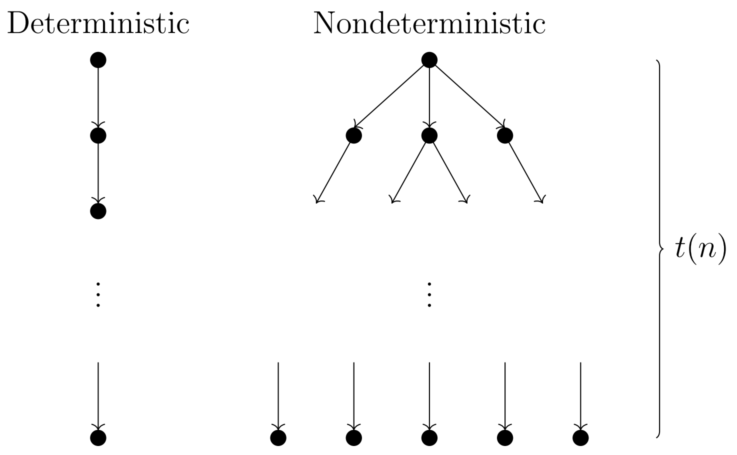
Every $t(n)$ time *nondeterministic* Turing machine N has an equivalent $2^{O(t(n))}$ time *deterministic* 3-tape Turing machine M.

M effectively simulates all possible branches of N by performing a breadth-first search through its computation tree. M's tapes contain the input string, the simulation contents, and the address/index of the branch currently being simulated, respectively.

We bound the number of choices for nondeterminism at each step by some number b, so that the total number of leaves of the tree (i.e. the number of branches that must be simulated) is at most $O(b^{t(n)})$. Additionally, each branch takes $O(t(n))$ time to simulate, so M takes $O(t(n)b^{t(n)}) \in 2^{O(t(n))}$ time total.

Converting M to a single tape Turing machine only squares this bound, which remains in $2^{O(t(n))}$.

The distinction between deterministic and nondeterministic time classes is shown in the diagram below.



5.3 P

The class P contains all languages that are decidable on a single tape Turing machine in *deterministic* polynoimal time.

$$P = \bigcup_k TIME(n^k)$$

We also note that multi-tape Turing machines are *polynomially equivalent* to single tape Turing machines; simulating a multi-tape Turing machine only requires a polynomial (more specifically, quadratic) increase in time bounds, so any language that is decidable in polynomial time on a multi-tape Turing machine is also decidable in polynomial time on a single tape Turing machine. The languages in P are invariant for all models of computation that are polynomially equivalent to a single tape Turing machine (which turns out to broadly encompass any model of computation undertaken by a real computer).

5.3.1 Dynamic Programming

The easiest way to prove that a problem A is in P is to simply provide a poly-time algorithm that solves A. In many simple cases, this boils down to tracking the number of nested iterations the algorithm has to process its input.

However, with more complex problems, smarter strategies may be needed to fit within a polynomial time bound — in most cases, this requires recognizing that some repeated section of computation is *redundant*, and does not need to be recomputed again and again. Instead, we can store and reference these sections in constant time, pruning out a significant amount of work. This is the main motivation behind *dynamic programming*, a framework for algorithm design that recursively divides a large problem into subproblems, and then *memoizes* (stores) past computation in these subproblems to be reused in solving the larger problem. This is helpful, as many larger problems decompose into the same atomic subproblems; not having to repeat computation saves a lot of time.

5.4 NP

Similarly to P, the class NP is the set of languages decidable in polynomial time by a nondeterministic Turing machine.

$$NP = \bigcup_k NTIME(n^k)$$

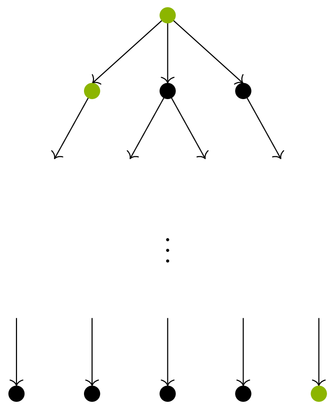
Languages in NP additionally have the property of *polynomial verifiability.* For $A \in NP$, given a proposed "certificate" of membership for an input x, we can easily check if the certificate does, indeed, prove that $x \in A$. For example, a certificate for determining if a graph G has a Hamiltonian path would be to provide the path itself; the polynomial time verifier would take this path and check that (a) it is a valid path, (b) it contains all vertices in G, and (c) it does not repeat vertices.

Formally, a verifier for a language A is an algorithm V for which

$$A = \{w \mid V \text{ accepts } \langle w, c\rangle \text{ for some string } c\}$$

where c is the certificate, the "extra information," that proves w's membership in A. V is a polynomial-time verifier if it runs in polynomial time in the length of w.

Polynomial verifiability follows necessarily from the definition of NP. For $A \in NP$, decided by a nondeterministic Turing machine N, some x is in A if there is at least one computational branch in the execution of N on which N accepts. The certificate for x, then, is the sequence of nondeterministic *choices* made in the execution of N on the accepting branch.



5.4.1 Polynomial Time Reduction

A function $f : \Sigma^* \rightarrow \Sigma^*$ is polynomial time computable if there exists a Turing machine M that can compute it in polynomial time; M can take any w as input and halt with $f(w)$ written on its tape in a polynomial number of time steps in the size of w.

A language A is polynomial time (mapping) reducible to another language B if there exists a polynomial time computable function f for which

$$w \in A \iff f(w) \in B$$

This is written as

$$A \leq_P B$$

A polynomial time reduction is simply a special case of mapping reduction, so the same properties apply; we've just added an additional computation time restraint on f.

Like mapping reducibility, polynomial time reducibility implies a certain "hardness" ordering. If $A \leq_P B$, and B is in P, then A must also be in P; given a polynomial time decider M for B, we can construct a polynomial time decider for N as follows.

```
N = "On $w$,
    1. Compute $f(w)$.
    2. Run $M$ on $f(w)$ and do as $M$ does."
```

Since both stages are computable in polynomial time, the overall process is computable in polynomial time.

Conversely, if A is not in P, then B cannot also be in P; if B were in P, this would mean that A must also be computable in polynomial time, but this conclusion leads to a contradiction.

5.4.2 NP-Completeness

A problem B is said to be NP-complete if it is both in NP and is NP-hard.

NP-hardness indicates that a certain problem is provably more difficult than every other problem in the class NP. Formally, B is said to be NP-hard if,

$$\forall A \in NP,\ A \leq_P B$$

Therefore, if any NP-complete B is found to be in P, this would imply that all problems in NP would also be in P.

Furthermore, if $B \leq_P C$, then C is also NP-hard, as reductions can be composed in polynomial time.

5.4.3 Cook-Levin Theorem

A boolean formula ϕ consists of variables $x_1, x_2, \cdots$ that are composed together with the boolean operations (NOT, AND, and OR). For example,

$$\phi = (x_1 \vee x_2) \wedge (x_1 \vee \overline{x_3} \vee x_4)$$

is a boolean formula. Each x_1 or $\overline{x_1}$ is called a *literal.* We say that ϕ is satisfiable if there exists an assignment to its variables that makes the formula evaluate to true (for example, a satisfying assignment to the formula above is $(x_1, x_2, x_3, x_4) = (T, T, F, F)$).

$$SAT = \{\langle \phi \rangle \mid \phi \text{ is a satisfiable Boolean formula}\}$$

We can furthermore impose more structure on ϕ. ϕ is in 3-cnf (conjunctive normal form) if it is strictly composed of *clauses* of length 3 that are ANDed together. A *clause* is an expression consisting of literals that are ORed together; specifically, in 3-cnf, clauses consist of exactly 3 literals.

$$3SAT = \{\langle \phi \rangle \mid \phi \text{ is a satisfiable Boolean formula in 3-cnf}\}$$

The Cook-Levin theorem states that SAT and, furthermore, $3SAT$, are NP-complete. Demonstrating that any other problem A is NP-complete usually requires demonstrating the reduction

$$3SAT \leq_P A$$

3-cnf provides a nice structure to ϕ in terms of demonstrating its satisfiability, as the length of each clause is bounded by a constant, and demonstrating that ϕ is satisfiable only requires showing that at least one literal per clause in ϕ evaluates to T. This gives rise to a convenient strategy for exhibiting reductions from $3SAT$ — given some ϕ, we can construct "variable gadgets" and "clause gadgets" in the specific context of A, in which variable gadgets represent assignments to each of the variables in the corresponding ϕ, and clause gadgets correspond to the truth value of each clause given the exhibited assignments.

6 Space Complexity

In addition to time, another resource that Turing machines use and require is *space*, or the amount of working tape they need to decide a particular input.

6.1 SPACE and NSPACE

The *space complexity* of a Turing decider M is some function $f : \mathbb{N} \to \mathbb{N}$, where $f(n)$ is the maximum number of tape cells M uses on any input of size n. Similar to the $TIME$ and $NTIME$ classes, $SPACE(f(n))$ is the set of languages decided by a deterministic Turing machine in $O(f(n))$ space, and $NSPACE(f(n))$ is the set of languages decided by a nondeterministic Turing machine in $O(f(n))$ space.

6.1.1 Connection to time

We can make a few connections between time and space bounds.

$$TIME(f(n)) \subseteq SPACE(f(n))$$

This is proven by realizing that the "most wasteful" $f(n)$ time Turing machine would simply keep increasing the size of its working tape by one cell for each time step, by simply moving its head rightward. Therefore, any $f(n)$ time algorithm uses at most $f(n)$ space.

$$SPACE(f(n)) \subseteq TIME(2^{O(f(n))})$$

Any $f(n)$ space Turing machine has $|\Gamma|^{f(n)} \cdot |Q| \cdot f(n) \in 2^{O(f(n))}$ distinct configurations that it could take at any point in its computation. Therefore, there are at most $2^{O(f(n))}$ configurations the machine could process before either halting and accepting/rejecting, or recognizing that it has repeated a configuration and must be looping.

6.1.2 Savitch's Theorem

Space is generally considered to be a more powerful resource than time, because space can be reused where time cannot. In fact, simulating a nondeterministic Turing machine with a deterministic one only requires a quadratic (polynomial) increase in space needed. Savitch's theorem states that, for $f : \mathbb{N} \to \mathbb{R}^+$,

$$NSPACE(f(n)) \subseteq SPACE(f^2(n))$$

This is compared to the exponential increase in time required to do the same simulation. The idea behind proving this theorem rests on solving the *yieldability problem*: given an NTM N, can we determine if N can go from configuration c_1 to configuration c_2 within t steps (does $c_1 \xrightarrow{t} c_2$)? We give the following procedure:

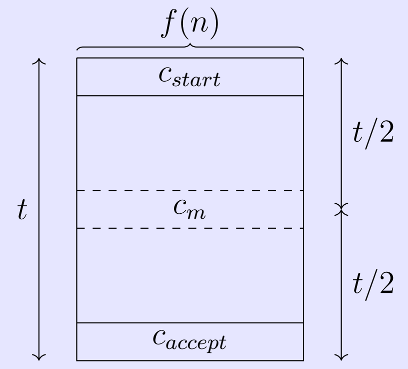
For configurations c_1, c_2, and nonnegative integer t:

1. if $t \leq 1$, test if $c_1 \to c_2$ directly, by comparing each symbol in c_1 to c_2 and determining if they are either equivalent or if it is possible to reach c_2 from c_1 within one step according to N.

2. Otherwise, we use the following recursive process: for every possible intermediate configuration c_m, test if $c_1 \xrightarrow{t/2} c_m$ and $c_m \xrightarrow{t/2} c_2$.

3. If no such c_m exists, reject.

Calling this algorithm on c_{start} and c_{accept} of N with $t = 2^{O(f(n))}$ will deterministically decide the language of N. Additionally, since the recursion halves the search space for each c_m, the maximum recursion depth is $\log t = O(f(n))$; each iteration will need to store a maximum of $O(f(n))$ configurations. Additionally, each configuration requires a maximum of $f(n)$ space to be written down,by virtue of N being in $NSPACE(f(n))$. This simulation therefore requires a maximum of $O(f^2(n))$ space at any given step.



6.2 PSPACE and NPSPACE

PSPACE is the class of languages that can be computed in polynomial space.

$$PSPACE = \bigcup_k SPACE(n^k)$$

Similarly,

$$NPSPACE = \bigcup_k NSPACE(n^k)$$

By Savitch's theorem, the PSPACE and NPSPACE are actually equivalent.

$$PSPACE = NPSPACE$$

We note that

$$P \subseteq PSPACE,\ NP \subseteq NPSPACE = PSPACE$$

In addition, if

$$EXPTIME = TIME(2^{\text{poly}(n)})$$

then

$$PSPACE \subseteq EXPTIME$$

so the hierarchy becomes

$$P \subseteq NP \subseteq PSPACE = NPSPACE \subseteq EXPTIME$$

6.2.1 PSPACE-Completeness

A language B is PSPACE-complete if it is both in PSPACE and is PSPACE-hard, meaning that

$$\forall A \in PSPACE,\ A \leq_P B$$

A *quantified boolean formula* adds the quantifiers $\forall$ and $\exists$ to a boolean formula ϕ. The statement $\forall x[\phi]$ claims that ϕ is satisfied for every possible assignment of x, whereas $\exists x[\phi]$ claims that ϕ is satisfied for at least one assignment to x. A *true* quantified boolean formula is a quantified boolean formula that evaluates to true — it satisfies all of its quantifiers.

$$TQBF = \{\langle\phi\rangle \mid \phi \text{ is a true quantified boolean formula}\}$$

$TQBF$ is PSPACE-complete. In addition, $TQBF$ is NP-hard and coNP-hard — this is because SAT is a special instance of $TQBF$, with all $\exists$ quantifiers.

6.2.2 Formula games

We can simulate "gameplay" with quantified boolean formulas by treating a particular formula *as* a game. For a boolean formula ϕ with variables $x_1, x_2, \cdots$, the game is represented as

$$\phi_t = \exists x_1 \forall x_2 \exists x_3 \cdots [\phi]$$

We say that the two players in the formula game are $\exists$ and $\forall$, which can take turns choosing assignments for the variables they are each bound to. Further, we say that $\exists$ wins if ϕ_t is true, and $\forall$ wins if ϕ_t is false. We say that a particular player has a *winning strategy* if there is a way for that player to win, given that both parties play "optimally." Determining whether $\exists$ has a winning strategy is equivalent to asking if $\phi_t \in TQBF$. If we formulate this as a language,

$$FORMULA - GAME = \{\langle \phi \rangle \mid \exists \text{ has a winning strategy for } \phi\}$$

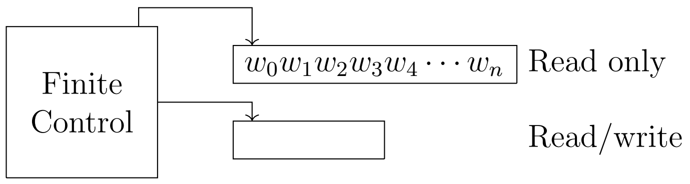
and $FORMULA - GAME$ is PSPACE-complete.

6.3 L and NL

L is the class of languages that can be deterministically computed in logarithmic space, and NL is the class of languages that can be nondeterministically computed in logarithmic space.

$$L = SPACE(\log n)$$
$$NL = NPSPACE(\log n)$$

Slight changes have to be made to our typical Turing machine model to make sense of log-space algorithms. We note that, with only logarithmic space, there is not even enough space to write the input down; we therefore separate our tape into a *read-only* input tape and a *read/write* work tape.



Otherwise, the mechanics of deterministic/nondeterministic computation remain the same.

6.3.1 Log-space Reduction

Polynomial time reductions no longer have the desired effect with respect to the languages in L, as every language in L is computable in polynomial time. Instead, we define a log-space reduction, using a machine called a *log-space transducer.*

A log-space transducer M is a Turing machine with three tapes: one input tape (read-only), one work tape (read/write), and one output tape (write-only). The size of the work tape is bounded by $O(\log n)$. M computes a log-space computable function $f : \Sigma^* \to \Sigma^*$; given an input w written on its read-only tape, it will compute and halt with $f(w)$ written on its write-only tape.

A language A is log-space reducible to a language B ($A \leq_L B$) if there exists a log-space computable f such that $w \in A \Leftrightarrow f(w) \in B$.

6.3.2 NL-Completeness

A language B is NL-complete if B is in NL and B is NL-hard, so

$$\forall A \in NL,\ A \leq_L B$$

Let

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

$PATH$ is an NL-complete language, and $\overline{PATH}$ is coNL-complete. Additionally, it can be shown that $\overline{PATH}$ can be computed in nondeterministic log space. Therefore, $\overline{PATH} \in NL$, so all problems in $coNL$ are additionally in NL, and therefore, $NL = coNL$.

7 Intractability

7.1 Space Hierarchy Theorem

A function $f(n)$ is *space constructible* if mapping the unary representation of n to the (at least) binary representation of $f(n)$ is computable using $O(f(n))$ space. For example, $f(n) = n^2$ is space-constructible; given 1^n (which already uses $O(n)$ space), a Turing machine can first obtain n in binary ($O(\log n)$ space) and then use any standard multiplication algorithm to obtain n^2 in binary, which will ultimately end up using less than $O(n^2)$ space.

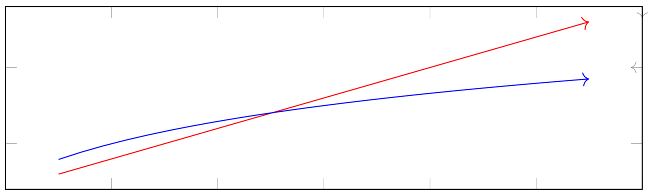
The space hierarchy theorem states that, given a space-constructible function f, there exists a language A that is computable in $O(f(n))$ but not $o(f(n))$ space. Essentially, there is a strict boundary between $SPACE(f(n))$ and $SPACE(g(n))$, where $g(n) \in o(f(n))$.

The proof of this theorem takes a diagonalization approach; we construct a machine D that decides A in $O(f(n))$ space, such that A differs from all languages computable in $o(f(n))$ space by some machine M. By the typical diagonalization argument, the place in which A differs from $L(M)$ will be the description of M itself.

D = "On w with length n,

1. Compute the space-constructible $f(n)$, and mark off this amount of tape on the current workspace. If, at any point in the computation on w, D attempts to move past this marker, then reject.
2. If w is not the form of some $\langle M\rangle 10^*$, for some Turing machine description $\langle M\rangle$, reject.
3. For a maximum of $2^{f(n)}$ steps, simulate M on w (i.e. its own description with the appended 10^*) while keeping a counter for the number of steps used thus far.
4. If M fails to reach a conclusion in $2^{f(n)}$ steps, it must be looping; reject.
5. If M halts and accepts, reject. If M halts and rejects, accept."

The goal of D is to disagree with any M that can run on itself within $o(f(n))$ time. We append "10^*" to $\langle M\rangle$, as the relation $o(f(n))$ describes *asymptotic* behavior — that is, we require a sufficiently large input size n to guarantee that $f(n) > g(n)$, if $g(n) \in o(f(n))$. For example, $f(n)$ and $g(n)$ could be related as below:



Though $g(n)$ is asymptotically strictly less than $f(n)$, $g(n)$ starts *larger than* $f(n)$. Large input sizes are needed to confirm that $g(n) \in o(f(n))$.

Because D certainly runs in $f(n)$ space (as it rejects any computation that tries to go beyond that) but disagrees with every machine that uses strictly less than $f(n)$ space, $L(D) \in SPACE(f(n))$, and this bound is tight. Further, for any $1 \leq \epsilon_1 < \epsilon_2$,

$$SPACE(n^{\epsilon_1}) \subsetneq SPACE(n^{\epsilon_2})$$

This theorem allows us to separate space-related hierarchy classes.

7.2 Time Hierarchy Theorem

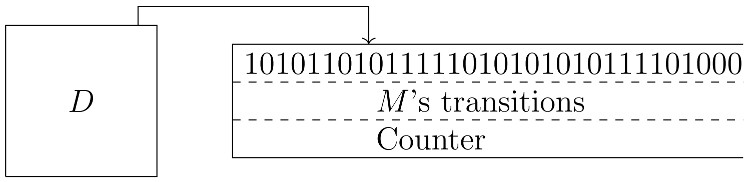
Similar to space-constructibility, a function f is *time-constructible* if the function mapping the unary input n (1^n) to the binary output $f(n)$ is computable in $O(f(n))$ time.

The time hierarchy theorem states that, for any time-constructible function $t : \mathbb{N} \to \mathbb{N}$, there exists a language A that is decidable in $O(t(n))$ time but not $o(t(n)/\log t(n))$ time. Compared to the space hierarchy theorem, this separation is not quite as tight; the proof is very similar, however, and the log factor comes from needing to use time steps to count the total amount of time taken by the constructed D that recognizes A.

D = "On w with length n,

1. Compute $t(n)$ and store the value $t(n)/\log t(n)$ in a counter variable. This counter decrements before each subsequent step taken by D. If the counter ever reaches 0, reject.
2. If $w \neq \langle M \rangle 10^*$ for some Turing machine description $\langle M \rangle$, reject.
3. Simulate M on w.
4. If M ever accepts, then reject; if M rejects, then accept."

We can envision the operation of D as running on three *tracks* of computation in parallel: one for storing M's simulated tape, one for storing the instructions for M for quick access, and one for keeping track of the counter.



Note that all three tracks are simulated on *one tape*, achieved by expanding D's tape alphabet or interleaving the contents of the three tracks on one tape. Moving M's transition function with each step is a constant time overhead, since the size of M's transition function is just a constant dependent on M, and is independent of any input size to M. The length of the counter is $\log(t(n)/\log(t(n))) \in O(\log t(n))$, so the cost of moving and updating the counter is about a $\log t(n)$ overhead per step. The language of D is decided in $O(t(n))$ time, as it runs a maximum of $t(n)/\log t(n)$ operations, each with $O(\log t(n))$ overhead. However, D disagrees with any M that can be simulated within $t(n)/\log t(n)$ time, so it cannot be decided in $o(t(n)/\log t(n))$ time. Therefore, for any $g(n) \in o(t(n))$

$$L(D) \in TIME(t(n)),\ L(D) \notin TIME(g(n))$$

For any $1 \leq \epsilon_1 < \epsilon_2$,

$$TIME(n^{\epsilon_1}) \subsetneq TIME(n^{\epsilon_2})$$

7.3 EXPSPACE

EXPSPACE is the class of all languages decidable in exponential space, $SPACE(2^{\text{poly}n})$. We introduce the exponentiation operation to regular expressions, such that for a regular expression R,

$$R^k = R \uparrow k = \underbrace{R \circ R \circ \cdots \circ R}_{k \text{ times}}$$

In addition, we define the problem

$$EQ_{REX\uparrow} = \{\langle Q, R\rangle \mid Q,\ R \text{ are equivalent}\}$$

for regular expressions with exponentiation Q and R. $EQ_{REX\uparrow}$ is EXPSPACE-complete.

7.4 Oracles and Relativization

An oracle for a language A is a black box device that is said to output the correct answer to any instance of A in $O(1)$ computation steps. An oracle Turing machine M^A is a modified Turing machine that can query an oracle for A at any point in its computation, and use the output of A to decide its initial input. The class P^A is the class of all languages that are decidable in deterministic polynomial time, given access to an oracle for A; similarly, the class NP^A is the class of all languages that are decidable in nondeterministic polynomial time, given an oracle for A.

The goal of conceptualizing an oracle for A is essentially to ask: supposing that I am able to solve A, what *further* capabilities does this give my computation? The oracle for A is describes computing *relative to* the problem of solving A.

For example, we note that $NP \subseteq P^{SAT}$; recall that any problem in NP is reducible to SAT in polynomial time. Once the reduction is computed on any problem instance, it can be queried to the SAT oracle, which provides an answer in one time step. Furthermore, $coNP \subseteq P^{SAT}$.

There exists an oracle A and B such that

1. $P^A \neq NP^A$
2. $P^B = NP^B$

This result demonstrates that solving $P \overset{?}{=} NP$ cannot simply be done using a *simulation* argument (like diagonalization), as oracles are simulation-invariant. If P were provably either the same or different from NP based on an argument about simulation, then adding the capabilities of an oracle should not change anything; however, as the above result notes, this is not the case, and therefore, solving $P \overset{?}{=} NP$ requires a proof at a more fundamental level.

8 Probabilistic Computation

A probabilistic algorithm is an algorithm that uses some sort of random process to decide a language. In most cases, this will boil down to a "coin flip," or a 50/50 guess, the result of which will determine the direction that an algorithm takes in its computation.

A *probabilistic* Turing machine M is similar to a nondeterministic Turing machine, except in that each configuration may branch into two nondeterministic choices, and each step will use a "coin flip" to determine which branch to pursue. Suppose that a branch b of M's computation chooses between nondeterministic paths k times (i.e., M performs k coin flips on b). We define the probability of going down branch b to be

$$\Pr[b] = 2^{-k}$$

We furthermore define

$$\Pr[M \text{ accepts } w] = \sum_{b_+} \Pr[b_+]$$

where b_+ is an accepting branch for M on w.

$$\Pr[M \text{ rejects } w] = 1 - \Pr[M \text{ accepts } w]$$

We say that M decides a language A with error probability $\epsilon \in [0, 1/2)$ if

1. $w \in A \longrightarrow \Pr[M \text{ accepts } w] \geq 1 - \epsilon$
2. $w \notin A \longrightarrow \Pr[M \text{ rejects } w] \geq 1 - \epsilon$

In other words, M has a ϵ-probability of being *wrong* on w. ϵ may be a function of the input size n — $\epsilon = 2^{-n}$, for example, indicates an exponentially decreasing error probability.

8.1 BPP

BPP is the class of languages that are decided by probabilistic polynomial time Turing machines with $\epsilon = 1/3$. We say that M runs in probabilistic polynomial time if every possible branch of M will halt within polynomial time.

8.1.1 Amplification Lemma

The error probability bound $\epsilon = 1/3$ is an arbitrary definition; in fact, it turns out that every probabilistic Turing machine M_1 with an error bound $\epsilon_1 < 1/2$ has an *equivalent* probabilistic Turing machine M_2 with error bound ϵ_2 for all $0 < \epsilon_2 < 1/2$. A stronger statement is that $\epsilon_2 = 2^{-p(n)}$ for any polynomial $p(n)$.

Given a probabilistic Turing machine M_1 with an error probability $\epsilon_1 < 1/2$ and a polynomial $p(n)$, we can construct an M_2 as follows:

```
$M_2$ = "On $x$,
    1. Calculate and compute some constant $k$.
    2. Run $2k$ independent simulations of $M_1$ on $x$.
    3. If most of the runs ($> k$) of $M_1$ accept $x$, accept. Otherwise,
       reject."
```

Suppose S is the sequence of length $2k$ of outputs of M_1. Suppose S has c correct results and w incorrect results, so that $c + w = 2k$. If $c \leq w$, then M_2 is *incorrect* on x, and the corresponding S is a "bad sequence". Let P_S be the probability that we obtain a bad sequence, and we sum over all P_S to bound the error probability of M_2. Since there are at most 2^{2k} sequences (and therefore, 2^{2k} *bad sequences*),

$$\Pr[M_2 \text{ is incorrect }] = \sum_{\text{bad } S} P_s \leq 2^{2k}\epsilon_1^k(1-\epsilon_1)^k$$

The above error upper bound decreases exponentially in k. If we choose $k \geq t/\alpha$, for $\alpha = -\log_2(4\epsilon_1(1-\epsilon_1))$, then we achieve an error bound for M_2 of $2^{-p(n)}$.

8.2 Arithmetization

Arithmetization is a technique that transforms a boolean formula $\phi(x_1, \cdots, x_n)$ into a polynomial $p(x_1, \cdots, x_n)$ that is *equivalent to* the truth value of ϕ when $(x_1, \cdots, x_n) \in \{0,1\}^n$. Implicitly, we are encoding '1' to mean 'true', and '0' to mean 'false'.

$$\begin{aligned}(x_1 \vee x_2) &\mapsto x_1 + x_2 - x_1x_2 \\ (x_1 \wedge x_2) &\mapsto x_1 \cdot x_2 \\ \overline{x} &\mapsto (1-x)\end{aligned}$$

Arithmetization is often used to reduce the complexity of testing the equality or inequality of boolean expressions, by making an argument about the equivalence of their corresponding polynomials. This is based on the Schwartz-Zippel lemma:

Let p be a polynomial $p(x_1, x_2, \cdots, x_n)$ defined on the finite field $\mathbb{F}_q$ for some prime q. Let $(r_1, r_2, \cdots, r_n)$ be numbers selected uniforly at random from $\mathbb{F}_q$. Then,

$$\Pr[P(r_1, r_2, \cdots, r_n) = 0] \leq \frac{nd}{q}$$

where d is the maximum degree of any variable in the polynomial.

For two polynomials p_1 and p_2, $p_1 - p_2 = 0$ everywhere iff $p_1 = p_2$; otherwise, they agree on very few places. The above lemma allows us to bound the probability of sampling places on which p_1 and p_2 agree.

8.3 IP Systems

Interactive proof systems are the probabilistic analogue to the class NP, just like BPP is the probabilistic analogue to the class P. We can recontextualize the property of polynomial time verifiability for problems in NP in terms of two parties "interacting" with each other: the Prover (P) and the Verifier (V). For an input w in an IP system deciding the language A, we conceptualize P as a party with infinite computational resources, attempting to *convince* a computationally limited V of w's membership in A. They do so by exchanging "messages"; for example,if P tells V that some $w \in A$, then V may modify the problem slightly to ask P a "trick question" that can only be answered deterministically if P was actually telling the truth.

More specifically, for an IP model, V is a polynomial time probabilistic Turing machine, so the calculations that V can make must be in the class BPP. In addition, the number of exchanges between P and V must be polynomial in the input size.

For example, take the language

$$NONISO = \{\langle G, H\rangle \mid G, H \text{ are nonisomorphic graphs}\}$$

V can first query P to ask if G and H are not isomorphic. If P replies that they are isomorphic, P can furthermore provide V with proof of this in the form of a certificate (the vertex correspondence between graphs), which V can verify in polynomial time and reject accordingly. If P replies that they aren't isomorphic, then V can:

1. Randomly choose between G and H, and reorder the chosen graph to produce graph F, keeping record of which graph was reordered to produce it.
2. Send F to P, and ask P which graph F is the rearrangement of.
3. P is only guaranteed to answer V correctly if P could truly tell between G and H. Otherwise, it has a $1/2$ chance of providing V with the correct answer.

Over many iterations of the above procedure, V can discern whether P is truly solving the problem.

Formally, a language A is in IP if some polynomial time computable V exists such that

$$w \in A \longrightarrow \Pr[V \leftrightarrow P \text{ accepts } w] \geq \frac{2}{3}$$
$$w \notin A \longrightarrow \Pr[V \leftrightarrow P \text{ accepts } w] < \frac{1}{3}$$

IP is like the class NP but with an added capability of probabilistic verification. IP is like the class BPP but with added “interactivity.”

It turns out that the problem

$$\#SAT = \{\langle \phi, k \rangle \mid \phi \text{ has exactly } k \text{ satisfying assignments}\}$$

is in IP, the proof of which depends on arithmetization and can be extended to show that $TQBF \in$ IP. Therefore, PSPACE $\in$ IP. We can furthermore prove that IP $\in$ PSPACE by simulating V with a PSPACE machine and defining

$$\Pr[V \text{ accepts } w] = \max_P \Pr[V \leftrightarrow P \text{ accepts } w]$$

such that the PSPACE machine that simulates V is an upper bound on the probability of $V \leftrightarrow P$ accepting w for any P.

Therefore, IP = PSPACE.