COMS W3261: Computer Science Theory
By William Pires and Zachary Thayer
and based on previous material from Rahul Gosain, Benjamin Kuykendall

Final Review

1 Key Terms

Set: a group of objects, which we call its elements. These "objects" can be almost anything: symbols, numbers, strings, or sets themselves. We have multiple ways to define sets:

Enumerate the items of a finite set: *e.g.* $\{1\}$, $\{1, 2, 3\}$, $\{\texttt{a}, \texttt{b}, \texttt{c}\}$.
Use an ellipsis to continue a pattern infinitely: *e.g.* $\{1, 2, 3, \dots\}$, $\{1, 11, 111, \dots\}$.
Give a condition for membership: *e.g.* $\{n \mid n = m^2 \text{ for some } m \in \mathbb{N}\}$.

We have symbols for members and subsets:

Membership: say $x \in S$ (or "x is in S") if x is an element of S.
Subset: say $T \subseteq S$ (or "T is a subset of S") if every element in T is also in S.
Powerset: let $\mathcal{P}(X)$ (or "the set of all subsets of X") be $\mathcal{P}(X) = \{S \mid S \subseteq X\}$.
Note that $\varnothing \in \mathcal{P}(X)$ and $X \in \mathcal{P}(X)$.

We have a few basic operations on sets. *e.g.*

Union: $A \cup B = \{x \mid x \in A \text{ or } x \in B\}$.
Intersection: $A \cap B = \{x \mid x \in A \text{ and } x \in B\}$.
Difference: $A \backslash B = \{x \mid x \in A \text{ and } x \notin B\}$.

Alphabet: a non-empty finite set of symbols, often denoted Σ. *e.g.*

$\{\texttt{0}, \texttt{1}\}$, $\{\texttt{0}, \texttt{1}, \dots, \texttt{9}\}$, $\{\texttt{a}, \texttt{b}, \texttt{c}\}$, $\{\texttt{a}, \texttt{b}, \texttt{c}, \dots, \texttt{z}\}$.

String: a finite sequence of symbols from the alphabet. *e.g.*

ε (the empty string, which has no symbols)
`1`, `11`, `111`, `1996`, `benjamin`

Encodings: finite mathematical objects can be encoded in a string over any language, *e.g.*

Integers (using their base-n representations)
Lists and sets (by separating the objects by commas)
Graphs (by listing the vertices and edges)
Functions between finite sets (by listing the pairs $(x, f(x))$)

Further, since all of our models of computation are tuples of sets and functions, we can encode DFAs, NFAs, PDAs, and TMs. If x is an object, denote the encoding by $\langle x \rangle$.

In general, not all strings will be valid encodings. For example, if we are encoding pairs of integers in the format $x\texttt{\#}y$, then `123#321` is a valid encoding, but `##` is not.

Language: a set of strings from the same alphabet. *e.g.*

$\varnothing$ (the set of no strings)
Σ^* (the set of all strings over Σ)
$A = \{w \in \{\mathtt{0},\mathtt{1}\}^* \mid w \text{ contains a } \mathtt{1}\}$
$G_C = \{\langle G\rangle \mid G \text{ is a cyclic graph}\}$
$M_{\mathtt{0110}} = \{\langle M\rangle \mid M \text{ is a TM that accepts } \mathtt{0110}\}$

Complement of language: the complement $\overline{L}$ is the set of strings in Σ^* but not in L. Or in set theory notation $\overline{L} := \Sigma^* \backslash L$. *e.g.*

$\overline{\varnothing} = \Sigma^*$
$\overline{\Sigma^*} = \varnothing$
$\overline{A} = \{\mathtt{0}^n \mid n \geq 0\}$
$\overline{G_C} = \{\langle G\rangle \mid G \text{ is not a graph or } G \text{ is an acyclic graph}\}$
$\overline{M_{\mathtt{0110}}} = \{\langle M\rangle \mid M \text{ is not a TM or } M \text{ is a TM that does not accept } \mathtt{0110}\}$

Turing Machine: we will formally define a Turing machine later; for now, consider the three possible results when you run a Turing machine M on a string w:

After some number of steps, the machine enters the accept state: say "M accepts w."
After some number of steps, the machine enters the reject state: say "M rejects w."
The machine reaches neither the accept nor the reject state: say "M loops on w."

We say "M halts on w" if it either accepts or rejects.

Recognizer: "M recognizes L" when

$$w \in L \iff M \text{ accepts } w.$$

The language recognized by M is unique, namely $L(M) = \{w \in \Sigma^* \mid M \text{ accepts } w\}$.

Decider: "M decides L" when both

$$w \in L \implies M \text{ accepts } w \quad \text{and} \quad w \notin L \implies M \text{ rejects } w.$$

Or equivalently:

If M recognizes L and M halts on all inputs.

Again, if M decides any language, that language is $L(M)$.

Computable function: a function $f : \Sigma^* \to \Sigma^*$ is computable when there is some Turing machine (technically an input-output Turing machine) that, on input string x, will contain $f(x)$ on the tape when it halts.

Set of Languages: like any mathematical objects, we can have a set of languages. *e.g.*

$\{A\}$ (the set containing only the language A)
$\{L \mid L \subseteq \{0,1\}\}$ (the four languages $\varnothing$, $\{0\}$, $\{1\}$, and $\{0,1\}$)
$\{L \mid L \subseteq \Sigma^* \text{ and } L \text{ is recognizable}\}$ (or regular, decidable, unrecognizable ...)
$\{L \mid L \subseteq \Sigma^* \text{ and } L \text{ can be decided in polynomial time}\}$ (the class P)
$\{L \mid L \subseteq \Sigma^* \text{ and } L \text{ can be verified in polynomial time}\}$ (the class NP)

It is important to be able to distinguish different types of sets: sets of strings (languages) versus sets of languages. Here are some tricky examples of sets of languages that you should understand. *e.g.*

$\{\varnothing\}$ (the set containing only the language $\varnothing$)
$\varnothing$ (the set containing no languages)
$\{L \mid L \subseteq \Sigma^*\}$ (the set of all languages over Σ^*)
$\{\Sigma^*\}$ (the set containing only the language Σ^*)

2 Set Cardinality

The cardinality of a set is its size, denoted by $|S|$. For a finite set, $|S|$ is simply the number of elements. Infinite sets have cardinalities as well. To show that two sets have the same cardinality, i.e. $|S| = |T|$, we need a bijection f from S to T (recall that f is a bijection iff for each $y \in T$, there is a unique $x \in S$ such that $f(x) = y$.)

Definition 1.

- For two sets S, T we say $|S| = |T|$ if there exists a bijection $f : S \to T$.
- We say $|S| \leq |T|$ if there exists an injective function $f : S \to T$.

We also consider the following important fact. If $S \subseteq T$, then $|S| \leq |T|$.

Definition 2. A set S is countable if S is finite or $|S| = |\mathbb{N}|$. If a set isn't countable, it's uncountable.

Alternatively, a set is countable iff every element has a description as a finite string or if we can give an enumeration of every element in the set. Note that Σ^* is always countable, so we can enumerate every string in Σ^* as $w_1, w_2, \ldots$.

Example 1:

Let $\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$ (the integers) and $\mathbb{Z}_{\geq 0} = \{0, 1, 2, 3, \ldots\}$ (the non-negative integers). We claim that $|\mathbb{Z}| = |\mathbb{Z}_{\geq 0}|$. Follow the above proof template: construct a function $f : \mathbb{Z} \to \mathbb{Z}_{\geq 0}$:

$$f(x) = \begin{cases} 2x & \text{if } x \geq 0 \\ 1 - 2x & \text{if } x < 0. \end{cases}$$

Consider two elements $x, y \in \mathbb{Z}$. If $f(x) = f(y)$ and the value is even, then we know $x = f(x)/2 = f(y)/2 = y$. Otherwise, if it's odd, we know $x = -(f(x) - 1)/2 = -(f(y) - 1)/2 = y$. So conclude $f(x) = f(y)$ iff $x = y$. Thus, f is injective, giving $|\mathbb{Z}| \leq |\mathbb{Z}_{\geq 0}|$.

Further, since $\mathbb{Z}_{\geq 0} \subseteq \mathbb{Z}$, we know $|\mathbb{Z}| \geq |\mathbb{Z}_{\geq 0}|$. Conclude that $|\mathbb{Z}| = |\mathbb{Z}_{\geq 0}|$. □

We have some adjectives to describe sets with certain cardinalities: an "inifinite" set is either countably or uncountably infinite, a "countable" set is either finite or countably infinite, and an "uncountable" set is uncountably infinite.

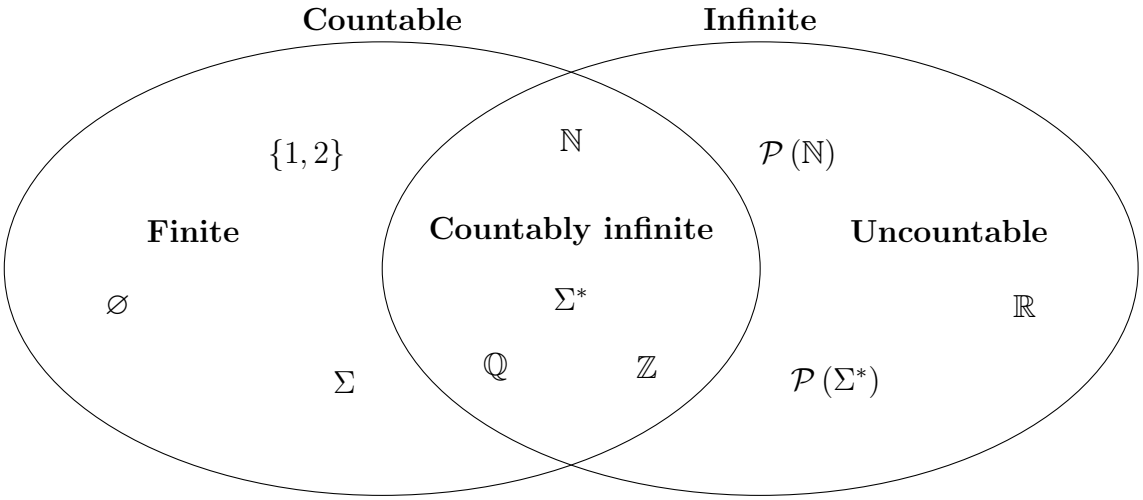


Figure 1: Relationships between different types of cardinalities.

Cantor's diagonalization proves $|\mathbb{N}| < |\mathcal{P}(\mathbb{N})|$. (In fact, for any infinite set X, we know $\mathcal{P}(X)$ is uncountable). In other words, countable sets are smaller than uncountable ones. From this we know that there is no surjective map from $\mathbb{N}$ to $\mathcal{P}(\mathbb{N})$.

2.1 Application to Recognizability

Let $\mathbb{M}$ be the set of all Turing machines. Consider the encoding map $f(M) = \langle M \rangle$. Since the encoding is injective from $\mathbb{M}$ to Σ^*, and Σ^* is countable, this gives that $\mathbb{M}$ is countable. Let us continue to describe sets we know from our study of Turing machines in terms of cardinality:

Σ: finite.
Σ^*: countable.
$L \in \Sigma^*$ (any language): countable.
$\mathbb{M}$ (the set of Turing machines): countably infinite.
$\mathcal{P}(\Sigma^*)$ (the set of all languages): uncountable.

Now consider the function from a Turing machine to the language it recognizes $L : \mathbb{M} \to \mathcal{P}(\Sigma^*)$. Since $\mathcal{P}(\Sigma^*)$ is uncountable, we know that the map cannot be surjective. In other words, there is some language $S \in \mathcal{P}(\Sigma^*)$ such that no Turing machine recognizes S. The same proof technique can be used to show that any uncountable set of languages contains some unrecognizable language.

3 Turing Machines

3.1 Formal Definition

A Turing machine is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$ where

1. Q is a finite set of states,
2. Σ is the input alphabet, $\sqcup \notin \Sigma$,
3. Γ is the tape alphabet, $\sqcup \in \Gamma$ and $\Sigma \subset \Gamma$,
4. $\delta : Q \times \Gamma \to Q \times \Gamma \times \{\text{L}, \text{R}\}$ is the transition function,
5. $q_0 \in Q$ is the start state,
6. $q_{\text{accept}} \in Q$ is the accept state,
7. $q_{\text{reject}} \in Q$ is the reject state, $q_{\text{accept}} \neq q_{\text{reject}}$.

This may seem like a lot to deal with. However, once you understand how the transition function works, each line of this definition should make sense. The machine operates by reading the character γ at the head, computing $\delta(q, \gamma) = (q', \gamma', D)$, writing γ' on the tape at the current position, moving the head left or right depending on D, and entering the state q'. It repeats, stopping only if one of q_{accept} or q_{reject} is reached.

3.2 High Level Descriptions

We use Turing machines and algorithms interchangeably. In class, we showed that Turing machines are able to do many operations involving strings and numbers; in fact, they are much more powerful. Any algorithm you could write in a programming language can be simulated by a Turing machine: the Church-Turing thesis postulates that Turing machines can simulate any reasonable model of computation. Thus, when giving a Turing machine specification, we often use *high level descriptions* (i.e. pseudoscope) to describe a machine.

Examples of permissible syntax and operations include the following:

- Variables: assign variables and use their values, add to or remove from finite lists and sets, check membership in a finite list or set.
- Control flow: use loops, jump conditionally, or have if-then-else statements.
- String operations: move strings, append them to one another, reverse them, find the n^{th} character of a string, find and replace substrings.
- Arithmetic operations: add, subtract, multiply, divide, or exponentiate integers or rational numbers, check if one number divides another, compare or check equality.
- Graphs: check if graph encodings are valid, check if a vertex has neighbors, add or remove edges and vertices, place marks on edges and vertices.
- Turing machines: check if an encoding is valid, run a Turing machine on an input, run a Turing machine on an input for some number of steps, and write a new Turing machine.

The operations on Turing machines can be subtle. Because Turing machines can run forever (and there is no way to check if a Turing machine will run forever, as we proved $\text{HALT}_{\mathsf{TM}}$ is undecidable), we have to account for this in our constructions.

Example 2:

Let M be a Turing machine. We will construct another Turing machine as follows.

$N =$ "On input w,

1. Simulate M on w.
2. If M accepts, reject.
3. If M rejects, accept."

If M decides L, then we know that N decides $\overline{L}$. To prove correctness observe:

$w \notin L \implies M$ rejects $w \implies N$ accepts w.

$w \in L \implies M$ accepts $w \implies N$ rejects w.

However, if M only recognizes L, it is possible that N could loop forever in step 1. For example, consider the case where M never rejects: instead, whenever it encounters an input $w \notin L$, it loops forever. In that case:

$w \notin L \implies M$ loops on $w \implies N$ loops on w.

$w \in L \implies M$ accepts $w \implies N$ rejects w.

Thus, in this case, N accepts no strings; in other words, it recognizes $\varnothing$.

To handle looping, we can use a technique called *dovetailing*, use parallelism, or take advantage of NTMs.

3.3 Example

Consider the language

$$L = \{w \in \{\mathtt{0}, \mathtt{1}\}^* \mid w \text{ contains no } \mathtt{1}\text{'s}\}.$$

We will give three Turing machines, in different formats, which all recognize L. [1]

[1] See the last page of Homework 3 for a reminder of what is meant by a formal level, implementation level, and high level description.

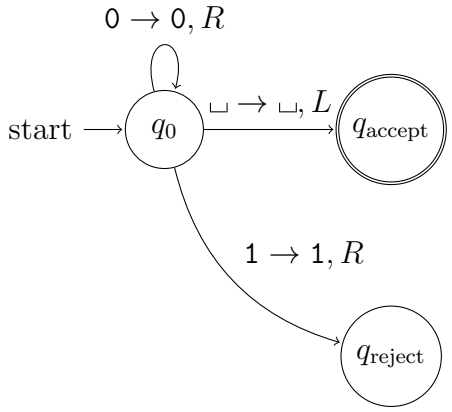
3.3.1 Formal description

$$M = (\{q_0, q_{\text{accept}}, q_{\text{reject}}\}, \{\texttt{0}, \texttt{1}\}, \{\texttt{0}, \texttt{1}, \sqcup\}, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$$

where δ is defined by

$$\delta(q_0, \texttt{0}) = (q_0, \texttt{0}, R),\ \delta(q_0, \texttt{1}) = (q_{\text{reject}}, \texttt{1}, R),\ \delta(q_0, \sqcup) = (q_{\text{accept}}, \sqcup, L)$$

Or, the same TM written as a state diagram:



3.3.2 Implementation level description

Check if the tape is empty, accept if so.
Repeat the following:
    Check if the current character is a 1, reject if so.
    Check if the current character is an empty space, accept if so.
    Else move the tape head right.

3.3.3 High level description

$M =$ "On input $x \in \{\texttt{0}, \texttt{1}\}^*$, check if x contains any 1's. If so, reject, else accept"

3.4 Non-Deterministic Turing Machines

Non-deterministic Turing Machines can have multiple possible moves from a given configuration (similar to NFAs). Like NFAs, they accept an input if and only if there exists a sequence that leads to the accept state. Nondeterminism can be thought of as letting the algorithm "guess", we then need to verify that the an input is in the language if and only if some lucky guess(es) can make the NTM accepts. That is, if a string should be in the input, then some sequence of guesses makes us accept (similar to how then some certificate exists in the polytime verifier model), and if the input is not in the language, no guesses can make us accept (similar to how no certificate exists).

Example We give an NTM to decide the following language (same as handout 8):
REACHABILITY $= \{\langle G, s, t\rangle \mid G$ is a graph, s, t are vertices, there exists a path in G from s to $t\}$

Our NTM N operates as follows:
On input $w = \langle G, s, t \rangle$, where G is a graph and s, t are vertices in G, do the following:

1. Set the current vertex to s.
2. Mark the current vertex as visited.
3. If the set of vertices connected to the current vertex by an edge are all marked:
4. Reject.
5. Else:
6. Nondeterministically set the current vertex to be an unmarked vertex in that set.
7. If the current vertex is t, accept w. Otherwise return to step 2.

Why this works: Suppose that $w \in$REACHABILITY: then by definition of REACHABILITY there exists a path, or sequence of vertices connected by edges from s to t. Thus, there exists a sequence of lucky choices N can make to reach t from s, so N accepts w.
Suppose that $w \notin$ REACHABILITY: then there exists no such guessable path from s to t, so all possible guesses make N reject w.

Finally, the TM can return to line 2 at most once per vertex in G. So the TM must always halts and can't run forever.

4 Decidability, Recognizability, Reductions and Rice's

4.1 Recognizability

Recall the following definition:

Definition 3. A language L is recognizable if there exists a (determnistic) Turing machine M such that:

$$x \in L \iff M \text{ accepts } x.$$

That is, M accepts every string in L and rejects or **or loops infinitely** on strings that are not in L. A language is unrecognizable if it's not recognizable.

Recognizable languages are not closed under complement. For instance, we know $\mathrm{A}_{\mathsf{TM}} := \{\langle M, x\rangle \mid M \text{ is a TM that accepts } x\}$ is recognizable, but $\overline{\mathrm{A}_{\mathsf{TM}}}$ isn't.

However, note that for recognizable languages one can equivalently use a Non-determinisitc TM.

Definition 4. A language L is recognizable if there exists a non-deterministic Turing machine M such that:

- If $x \in L$, there is some branch of nondeterminism on which M accepts x.
- If $x \notin L$, for all branches of nondeterminism M doesn't accept x.

In particular, the TM is allowed to loop on some non-deterministic branches.

4.2 Decidability

Decidability is a lot like recognizability, except we impose an extra condition on the Turing machine: it must *reject* all strings that are not in L.

Definition 5. A language L is decidable if there exists a (determnistic) Turing machine M such that:

$$\text{If } x \in L \text{ then } M \text{ accepts } x \text{ and if } x \notin L \text{ then } M \text{ rejects } x.$$

That is, a decider always gives us back an answer in finite time for all possible inputs.

We also have the following theorem:

Theorem 1. A language L is decidable if and only if L and $\overline{L}$ are both recognizable.

Decidable languages are closed under complement. That is, if A is decidable, then $\overline{A}$ is decidable. We can also use a NTM

Definition 6. A language L is decidable if there exists a non-deterministic Turing machine M such that:

- If $x \in L$, there is some branch of nondeterminism on which M accepts x.
- If $x \notin L$, for all branches of nondeterminism M rejects x.

You need to be careful and make sure that no branch of nondeterminism can go on forever. Here's an example. Consider the following NTM:

Algorithm 1 A decider for $\mathsf{A_{TM}}$?

Input: $\langle M, x\rangle$ where M is a TM

1: Non-deterministically pick $k \geq 0$
2: Run M on x for k steps.
3: **if** M accepted x **then**
4: Accept $\langle M, x\rangle$.
5: **else**
6: Reject $\langle M, x\rangle$.
7: **end if**

Here, clearly if M doesn't accept x, no matter what k we pick, the NTM rejects. On the other hand if M accepts x, picking k large enough means M will accept on line 3, so we accept. But $\mathsf{A_{TM}}$ isn't decidable, so something must be wrong. In particular consider the line "Non-deterministically pick $k \geq 0$", this is a shortand that should really read:

1: $k = 0$
2: **while** True **do**
3: Choose one: "Exit the while loop" or "increase k by 1".
4: **end while**

And observe that here, we can clearly see the NTM might never halt on some branch, where it keeps increasing k forever.

As a rule of thumb: Using an NTM is great when asked to give a recognizer, but maybe not the best when asked to give a decider. If you want to use an NTM to give a decider, make sure that what you "non-deterministically guess" comes from a finite set (so that "the guess can never go on forever").

4.3 Reductions

In general, a reduction is a way to order problems based on how hard they are. Formally, we say $A \leq B$ or "A is reducible to B". Here are three English-language interpretations of that statement that give good intution about reductions:

$$\begin{aligned} A \leq B &\iff \text{“the problem of solving } A \text{ reduces to the problem of solving } B\text{”} \\ &\iff \text{“if we can solve } B\text{, then we can solve } A\text{”} \\ &\iff \text{“}A \text{ is easier than [or just as easy as] } B\text{”}. \end{aligned}$$

Reduction come in multiple flavors: we studied $\leq_{\mathsf{T}}$, $\leq_{\mathsf{m}}$, and $\leq_{\mathsf{p}}$.

Definition 7. Let A, B be two languages:

Turing reduction. We say A is Turing-Reducible to B, denoted $A \leq_{\mathsf{T}} B$, if given a TM M that decides B, there exists a decider N for A

$$M \text{ decides } B \implies N \text{ decides } A.$$

Mapping reduction We say A is Mapping-Reducible to B, denoted $A \leq_{\mathsf{m}} B$, when there exists some computable function f is such that

$$x \in A \iff f(x) \in B.$$

We know that if $A \leq_m B$, then $A \leq_T B$. Thus wherever you might need a Turing reduction, you could use a mapping reduction instead. However, the converse does not hold: in some cases there is a Turing reduction from A to B, but no mapping reduction from A to B. In the next section, we will go over how both types of reductions can be used to solve problems. Before that, however, we'll recall the following theorems:

Theorem 2.

- $A \leq_T B \iff \overline{A} \leq_T B \iff A \leq_T \overline{B} \iff \overline{A} \leq_T \overline{B}$
- $A \leq_M B \iff \overline{A} \leq_M \overline{B}$

We also have the following important theorems:

Theorem 3.

- $A \leq_T B$ and B is decidable, then A is decidable.
- $A \leq_M B$ and B is recognizable, then A is recognizable.

Corollary 4.

- $A \leq_T B$ and A is not decidable, then B is not decidable.
- $A \leq_M B$ and A is not recognizable, then B is not recognizable.

Tip: When doing a mapping reduction $A \leq_m B$, you can ignore inputs that are bad encoding (i.e. not TM, DNA, CNF etc...) by first saying "If x is bad encoding, we just map it to some $y \notin B$". And then it's ok to ignore bad encodings in the proof that $x \in A \iff f(x) \in B$.

4.4 Rice's Theorem

Note that many of the undecidable languages we have learned about fit a common pattern. That is, they are languages of the form $\{\langle M \rangle \mid M \text{is a TM and } L(M) \text{ satisfies...}\}$. We call such languages "properties of recognizable languages." Formally,

Definition 8. P is a property of recognizable languages if $P \subseteq \{\langle M \rangle \mid M \text{is a TM}\}$ and if $L(M_1) = L(M_2)$ then $\langle M_1 \rangle \in P \iff \langle M_2 \rangle \in P$.

Definition 9. A property of recognizable languages P is not trivial if $P \neq \emptyset$ and $P \neq \{\langle M \rangle \mid M \text{is a TM}\}$.

In particular, we have the following very convenient theorems to tell if a language of this form is decidable / recognizable.

Theorem 5 (Rice's Theorem)**.** Let P be a non-trivial property of recognizable languages. Then P is not decidable.

The following stronger version (not required) tells us about the recognizability of properties.

Theorem 6 (Refined Rice's Theorem)**.** Let P be a non-trivial property of recognizable languages. Let $M_\emptyset$ be a TM such that $L(M_\emptyset) = \emptyset$. Then:

- If $\langle M_\emptyset \rangle \in P$, then P is not recognizable.
- If $\langle M_\emptyset \rangle \in P$, then $\overline{P}$ is not recognizable

5 Proof Techniques

In this section, you'll see three techniques you should consider when approaching a problem: direct demonstrations, reductions or Rice's theorem. Not all will work on every problem, but the proof templates below should help you get the hang of it. If you follow one of these templates, ensure you address all the bullet points in the template in your proof. Sometimes, a point is pretty trivial, but it's still important to mention it.

5.1 Proving Recognizability

To prove L is recognizable, you can give a recognizer for L and prove its correctness.

Proof Template 1: to show L is recognizable via a recognizer

1. Write the pseudo-code of a Turing Machine M that recognizes L.
2. Assuming $x \in L$, prove that M accepts x.
3. Assuming $x \notin L$, prove that M does not accept x.
4. Conclude that since M recognizes L, L is recognizable.

To prove a language is recognizable, it might be easier to use an NTM (see HW3 for examples). In this case you need to do as follows:

Proof Template 2: to show L is recognizable via a nondeterministic recognizer

1. Write a description of an non-deterministic Turing machine M that recognizes L.
2. Assuming $x \in L$, prove that there is some branch of nondetermnism where M accepts x.
3. Assuming $x \notin L$, prove that for every branch of nondeterminism M does not accept x.
4. Conclude that since M recognizes L, L is recognizable.

Example 3:

Recall the language

$$\mathrm{E_{TM}} = \{\langle M \rangle \mid M \text{ is a TM and } L(M) = \varnothing\}.$$

We will show the complement $\overline{\mathrm{E_{TM}}}$ is recognizable by constructing a recognizer.

M_N = "On input x

1. Check that $x = \langle M \rangle$, an encoding of a TM. If not, accept.
2. Let $i = 1$.
3. Run M on every string of length at most i for at most i steps each. If any simulation sees M accept, then accept.
4. Increment i and return to step 2."

Now we will prove correctness:

Assume $x \in \overline{E}_{TM}$. Then either $x \neq \langle M \rangle$ or $x = \langle M \rangle$ where M is a TM with $L(M) \neq \varnothing$. In the first case, then x is accepted in Step 1. In the second, then there is some $w \in L(M)$ that is accepted in some number of steps c. So, if w is not already accepted, then it will definitely be accepted by M_N in step 2 when $i = \max(c, |w|)$. In either case M_N will accept x.

Assume $x \notin \overline{E}_{TM}$. Then $x = \langle M \rangle$ where M is a TM with $L(M) = \varnothing$. This means that x will pass the check in Step 1, and every simulation of M in Step 3 will not result in an accept. So, M_N will run indefinitely and will not accept x.

Conclude that since M_N is a recognizer for $\overline{\mathrm{E_{TM}}}$, that $\overline{\mathrm{E_{TM}}}$ is recognizable. □

This first approach is often easier, but following Theorem 3 you can also use a reduction:

Proof Template 3: to show L is recognizable via mapping reduction

1. Pick a recognizable language B.
2. Write an algorithm that computes a function f. [a]
3. Explain why the algorithm always halts and outputs $f(x)$.
4. Assuming $x \in L$, prove that $f(x) \in B$.
5. Assuming $x \notin L$, prove that $f(x) \notin B$.
6. Conclude that since $L \leq_{\mathsf{m}} B$ and B is recognizable, L must be too.

[a] If f is very simple (such as $f(x) = \langle M, x \rangle$ where M is a fixed TM) you do not have to write down an algorithm.

5.2 Decidability

We can use the same proof strategy we used above and supply the decider explicitly, or we can use a reduction (again following Theorem 3).

Proof Template 4: to show L is decidable via a decider

1. Write the pseudo-code of a Turing Machine M that is a decider for L.
2. Assuming $x \in L$, prove that M accepts x.
3. Assuming $x \notin L$, prove that M rejects x.
4. Conclude that since M is a decider for L, L is decidable.

Example 4:

Recall the language

$$\text{CNF-SAT} = \{\langle\phi\rangle : \phi \text{ is a CNF Boolean formula with a satisfying assignment}\}.$$

We claim that CNF-SAT is decidable. We will define a Turing machine.

M_{CNF} ="On input $\langle\phi\rangle$ where ϕ is CNF. [a]

1. Iterate through all 2^n possible truth assignments to the variables in ϕ and evaluate ϕ with each. If one makes ϕ evaluate to True, accept.
2. After iterating through all assignments (without finding a satisfying assignment), then reject.

Now we will prove correctness:

Assume $\langle\phi\rangle \in$ CNF-SAT. Then ϕ is a Boolean formula in CNF form with a satisfying assignment. Then, $\langle\phi\rangle \in$ CNF-SAT will be accepted in Step 2 since there is some satisfying assignment that will be checked and cause ϕ to evaluate to True. So, M_{CNF} will accept $\langle\phi\rangle \in$ CNF-SAT.

Assume $\langle\phi\rangle \notin$ CNF-SAT [b]. Then ϕ is a Boolean formula in CNF form with no satisfying assignment. So, every assignment tested in Step 2 will not evaluate to True (since there is no satisfying assignment). So, M_{CNF} will reach Step 3 and reject $\langle\phi\rangle \in$ CNF-SAT $\notin$ CNF-SAT.

Conclude that since M_{CNF} is a decider for CNF-SAT, that CNF-SAT is decidable. □

[a] This is shorthand for "On input x: If x isn't of the form $\langle\phi\rangle$ where ϕ is a CNF then reject x.

[b] Here, we ignored cases where the input to the algorithm isn't of the form $\langle\phi\rangle$. This is because such inputs are automatically rejected by the algorithm. On the final it's ok to "ignore" bad encodings in your proof *if your algorithm automatically rejects bad encodings like we did here.*

Proof Template 5: to show L is decidable via Turing reduction

1. Pick some decidable language B and assume that you have a decider M_B for it.
2. Give the pseudo-code of a decider M_L for L using M_B.
3. Assuming $x \in L$, prove that M_L accepts x.
4. Assuming $x \notin L$, prove that M_L rejects x.
5. Conclude that since $L \leq_{\mathsf{T}} B$ and B is decidable, L must be too.

Finally, one can use Theorem 1 to prove that a language is decidable.

Proof Template 6: to show L is decidable via recognizability of L and $\overline{L}$

1. Prove the language is recognizable using aforementioned techniques.
2. Prove the complement of the language is recognizable as well.
3. Conclude the language is decidable.

To prove a language is decidable, you could also use an NTM. BUT as a rule of thumb, it's probably easier to give a determinisitc TM.

5.3 Undecidability

A language L is undecidable if it's impossible to construct a Turing machine that is a decider for L. Since we want to make a claim about *all* Turing machines, we can't check all the options individually. Most of the time, the best strategy is to follow Corollary 4, and use a reduction to prove that L is "at least as difficult" as some undecidable language.

Proof Template 7: to show L is undecidable via Turing reduction

1. Assume that you have a decider M_L for L.
2. Pick an undecidable language U.
3. Use M_L to give the psuedo-code of a decider for M_U for U.
4. Assuming $x \in U$, prove that M_U accepts x.
5. Assuming $x \notin U$, prove that M_U rejects x.
6. Conclude that since $U \leq_{\mathsf{T}} L$ and U is undecidable, L must be undecidable as well.

Example 5:

We want to show that E_{TM} is undecidable; we already know that A_{TM} is undecidable. For reference, here are the definitions of the two languages:

$$\mathrm{E}_{\mathsf{TM}} = \{\langle M \rangle \mid M \text{ is a TM and } L(M) = \varnothing\}.$$
$$\mathrm{A}_{\mathsf{TM}} = \{\langle M \rangle, x \mid M \text{ is a TM and } M \text{ accepts } \langle x \rangle\}.$$

In order to carry out the reduction, assume that there exists a decider M_E for E_{TM}. We could use M_E to create a decider for A_{TM} as follows.

$M_A =$"On input $\langle M, x \rangle$ where M is a TM,

1. Creates an encoding of the following Turing machine:

   $M' =$"On input z, run M on x and output the same."
2. Runs M_E on input $\langle M' \rangle$ and output the opposite."

Now we will prove correctness:

Assume $\langle M, x \rangle \in \mathrm{A}_{\mathsf{TM}}$. Then M accepts x. This means that the constructed M' will accept every input z, namely $L(M') = \Sigma^*$. So, M_E will reject when run on $\langle M' \rangle$ (since it is a decider for E_{TM}), and M_A will therefore accept x.

Assume $\langle M, x \rangle \notin \mathrm{A}_{\mathsf{TM}}$.[a]. Since M doesn't accept x, the constructed M' doesn't accept any string z. So $L(M') = \emptyset$. So, M_E will reject when run on $\langle M' \rangle$ (since it is a decider for E_{TM}), and M_A will therefore reject $\langle M, x \rangle$.

Conclude that since $\mathrm{A}_{\mathsf{TM}} \leq_{\mathsf{T}} \mathrm{E}_{\mathsf{TM}}$ and A_{TM} is undecidable, that E_{TM} is undecidable too. □

[a] Again, we ignored inputs that are bad encodings, since M_A automatically rejects them.

Finally, to show L is not decidable, you can use Rice's theorem.

Proof Template 8: to show L is undecidable using Rice's theorem

1. Prove that L is a language property:
   (a) Check that L consists of strings of the form $\langle M \rangle$ where M is a TM.
   (b) Check that for any two TMs M_1, M_2 such that $L(M_1) = L(M_2)$ it holds that $M_1 \in L \iff M_2 \in L$.
2. Prove that L is non-trivial:
   (a) Show that there exists a TM M such that $\langle M \rangle \in L$.
   (b) Show that there exists a TM M' such that $\langle M' \rangle \notin L$
3. Conclude that by Rice's theorem L is not decidable.

Example 6: E_{TM} is not decidable

Recall the language

$$\mathrm{E}_{\mathsf{TM}} = \{\langle M \rangle \mid M \text{ is a TM and } L(M) = \varnothing\}.$$

We first show E_{TM} is a property of recognizable language. Clearly $\mathrm{E}_{\mathsf{TM}} \subseteq \{\langle M \rangle \mid M \text{ is a TM}\}$. Now let M_1, M_2 be TMs such that $L(M_1) = L(M_2)$. If $\langle M_1 \rangle \in \mathrm{E}_{\mathsf{TM}}$ then $L(M_1) = \emptyset = L(M_2)$ so $\langle M_2 \rangle \in \mathrm{E}_{\mathsf{TM}}$. Similarly, if $\langle M_2 \rangle \in \mathrm{E}_{\mathsf{TM}}$, we can see $\langle M_1 \rangle \in \mathrm{E}_{\mathsf{TM}}$. So $\langle M_1 \rangle \in \mathrm{E}_{\mathsf{TM}} \iff \langle M_2 \rangle \in \mathrm{E}_{\mathsf{TM}}$.
Now we prove E_{TM} is a non trivial property. Considet the TMs M', M.

- M'= On input x:
  1. Accept x.

  $L(M') = \Sigma^*$ so $M' \notin \mathrm{E}_{\mathsf{TM}}$.
- M= On input x:
  1. Reject x.

  $L(M) = \emptyset$ so $M \in \mathrm{E}_{\mathsf{TM}}$.

So L is a non-trivial property of regular languages. So, by Rice's theorem, L is undecidable.

5.4 Unrecognizability

A language L is unrecognizable if it's impossible to construct a Turing machine that is a recognizer for L. As in the previous section, we're going to want to follow Corollary 4 and use **a mapping reduction** to make a claim about all possible Turing machines. The reduction strategy looks very similar. Note that for unrecognizability, you must use a mapping reduction. Turing reductions can only sow undecidability.

Proof Template 9: to show L is unrecognizable via mapping reduction

1. Pick an unrecognizable language U.
2. Write an algorithm that computes a function f.
3. Explain why the algorithm always halts and outputs $f(x)$.
4. Assuming $x \in L$, prove that $f(x) \in U$.
5. Assuming $x \notin L$, prove that $f(x) \notin U$.
6. Conclude that since $L \leq_{\mathsf{m}} U$ and U is recognizable, L must be recognizable too.

A small note about mapping reductions: When giving a mapping reduction $A \leq_m B$, it's cumbersome to deal with "wrong encoding" which are obviously not in A . Note that if you

want to show $A \leq_m B$, as long as $B \neq \Sigma^*$, we can fix some string $y \notin B$. And then if x has the wrong format for A (for example, $x \neq \langle M \rangle$), we can just fix $f(x) = y$.

As such, when doing mapping reductions, you can say "We ignore wrong encoding by mapping them to a fix string $y \notin B$". And then when giving f and the proof of correctness you can ignore wrong encodings.

We could use Rice's theorem for to prove undecidability, and we can use refined Rice's theorem in a similar way to prove unrecognizability:

Proof Template 10: to show L is unrecognizable via Refined Rice's Theorem

1. Prove that L is a language property:
   (a) Check that L consists of strings of the form $\langle M \rangle$ where M is a TM.
   (b) Check that for any two TMs M_1, M_2 such that $L(M_1) = L(M_2)$ it holds that $M_1 \in L \iff M_2 \in L$.
2. Prove that L is non-trivial:
   (a) Show that there exists a TM M such that $\langle M \rangle \in L$.
   (b) Show that there exists a TM M' such that $\langle M' \rangle \notin L$
3. Show that $\langle M_\emptyset \rangle \in L$. ($M_\emptyset$ is a TM with language equal to $\emptyset$)
4. Conclude that by Refined Rice's theorem L is not recognizable.

The last method you should keep in mind relies on Theorem 1.

Proof Template 11: to show L is unrecognizable by Theorem 1

1. Prove that L isn't decidable.
2. Prove that $\overline{L}$ is recognizable.
3. Conclude that since L is undecidable, but $\overline{L}$ is not recognizable then $\overline{L}$

5.5 Some notes about looping

Whenever you run a TM M, make sure to ask yourself "what if M loops and I get stuck on this line". Here are some important examples:

- When writing an algorithm you should never say something along the lines of :
  1. Run M on x
  2. If M loops then do X.

  This is because if M loops, you're stuck on line 1 running M forever. So X is just dead code (it can never get executed).

  Furthermore, the following would be a wrong algorithm:

– On input $\langle M, x\rangle$ where M is a TM:

– If M doesn't halt on x then X.

Any line of the form "If M loops/doesn't halt on x" isn't actually implementable and thus it's wrong to use something like that in your algorithms [2]!

- Using parallelism: Say you have a recognizer M_1 for L_1 and a recognizer M_2 for L_2. You want to make a recognzier for $L_1 \cup L_2$.

  The following is wrong:

  1. On input x:
  2. Run M_1 on x.
  3. Run M_2 on x.
  4. Accept if one M_1, M_2 accepted. Otherwise, reject.

  In particular if M_1 loops on x and M_2 accept x, the above loops on $x \in L_1 \cup L_2$. That's why you should proceed as follow:

  1. On input x:
  2. For $i = 1$ to ∞:
  3. Run M_1 on x for i steps.
  4. Run M_2 on x for i steps.
  5. If M_1 or M_2 accepted, accept.

  Equivalently

  1. On input x:
  2. Run M_1 and M_2 in **parallel** on x.
  3. Accept whenever one M_1, M_2 accepts.
  4. If both M_1, M_2 rejected, reject.

- Using dovetailing: We let $\Sigma^* = \{w_1, w_2, \ldots\}$ [3].Say you want to build a recognizer for

$$L = \{\langle M\rangle \mid M \text{ is a TM and } L(M) \neq \emptyset\}$$

  The following would be wrong.

  1. On input $\langle M\rangle$ where M is a TM:
  2. For $i = 1$ to ∞:
  3. Run M on w_i.
  4. If it accepts w_i accept.

[2]Recall that $\overline{\text{HALT}_{\text{TM}}}$ isn't recognizable, there's no algorithm that can check if a TM doesn't halt on some string.

[3]If you want to go through all strings in Σ^* in an algorithm like we do here, please write that you fix $w_1, w_2, \ldots$ to be an enumeration of Σ^*.

The above fails, because if M loops on some w_i, you get stuck on line 3 and never get to run M on w_{i+1}, w_{i+2} etc... For instance if M accepts w_2 but loops on w_1, the above wouldn't accept $\langle M \rangle$.

Hence, we must use dovetailing. Here are the two common ways to do this:

1. On input $\langle M \rangle$ where M is a TM:
2. For $i = 1$ to ∞:
3. Run M on all strings w of length $\leq i$ for i steps.
4. If it accepted some string w, accept.

When using dovetailing, you need to argue that at some point your algorithm accepts. In particular, there is a point where i is so large that you'd accept. Here is how to argue this:

Say $\langle M \rangle \in L$. Then $L(M) \neq \emptyset$, so M accepts some string w in k steps. When $i \geq \max(|w|, k)$, we have that we'll run M on w (since $|w| \leq i$, and we'll run M for enough steps that it accepts w. So the algorithm accepts $\langle M \rangle$.

Another way to describe dovetailing is as follow:

1. On input $\langle M \rangle$ where M is a TM:
2. For $i = 1$ to ∞:
3. For $j = 1$ to i:
4. Run M on w_j for i steps.
5. If M accepted w_j, accept.

Here the proof that the algorithm accepts $\langle M \rangle \in L$ should look as follow: Let $\langle M \rangle \in L$, then $L(M) \neq \emptyset$, so M accepts some string w_t in k steps. When $i \geq \max(t, k)$, we'll set $j = t$ and have that we'll run M on w_t. Since $i \geq k$, we'll run M for enough steps that it accepts w_t. So the algorithm accepts $\langle M \rangle$.

Finally, note that we can show the above is recognizable using a simple NTM:

1. On input $\langle M \rangle$ where M is a TM:
2. Non deterministically pick $x \in \Sigma^*$
3. Run M on x.
4. If M accepts, accept.
5. If M rejects, rejects.

6 Complexity

Up to this point, we have studied *computability*, that is, which languages are decidable, recognizable, or neither: roughly, whether a problem can be solved by a computer. However, this tells us nothing about how efficiently it can be solved. Now we turn our attention to *complexity*: the study of how much resources are needed to solve a problem. Efficiency is always measured relative to the length of the input (i.e. the number of characters), which we denote by n.

Definition 10. Let M be a deterministic Turing machine that halts on every input. We say M has running time $t(n)$ if for every input x of length n, M halts on input x within at most $t(n)$ steps.

Definition 11. Let M be a non-deterministic TM that halts on every input. We say M has running time $t(n)$ if for every input x of length n, **every possible computation** of M on x terminates within at most $t(n)$ steps.

In the above, the number of **steps** of M on an input x is the number of transitions M takes before it halts on x.

Definition 12.

$\mathsf{TIME}(f(n))$: the class of languages decidable by Turing machines that run in time $O(f(n))$.

$\mathsf{NTIME}(f(n))$: the class of languages decidable by non-deterministic Turing machines that run in time $O(f(n))$.

Definition 13. We say $t : \mathbb{N} \to \mathbb{N}$ is polynomial if there exists some constant k such that $f(n) = O(n^k)$.

Good to know: If $f : \mathbb{N} \to \mathbb{N}$ and $g : \mathbb{N} \to \mathbb{N}$ are both polynomials then the following are also polynomials:

- $f + g$, where $(f + g)(x) = f(x) + g(x)$.
- $f * g$, where $(f * g)(x) = f(x) * g(x)$.
- $f \circ g$, where $(f \circ g)(x) = f(g(x))$.

In particular if you have a step in your algorithm that takes polynomial time $O(n^{k_1})$ which is repeated polynomially many time $O(n^{k_2})$ (if you have a for loop, for instance) then the total running time is $O(n^{k_1+k_2}) = O(n^k)$.

Definition 14. P: the class of languages decidable by Turing machines that run in time $O(n^k)$ for some constant k. In other words

$$\mathsf{P} = \bigcup_{k=1}^{\infty} \mathsf{TIME}(n^k).$$

Definition 15 (Verifier)**.** A verifier V for a language L, is a **deterministic** algorithm such that V takes as as input x and some string c and

$$x \in L \leftrightarrow \exists\, c \text{ such that } V(x, c) \text{ accepts.}$$

V is said to be a polytime verifier if it runs in time $O(|x|^k)$ for some constant k.

Definition 16. NP: the class of languages decidable by non-deterministic Turing machines that run in time $O(n^k)$ for some constant k. In other words

$$\mathsf{NP} = \bigcup_{k=1}^{\infty} \mathsf{NTIME}(n^k).$$

Alternatively, NP is the class of languages with polytime verifiers (as defined above).

In this class, we don't care exactly what the exact running time is, as long it's polynomial in the input size. In particular, the speed of each step in your algorithm might depend on the model of computation you're thinking of. So, you need to argue every step of your algorithm takes polynomial time, and each for/while loop is run polynomially many times.

6.1 Polytime mapping reduction

Definition 17 (Poly-time mapping reducible)**.** A language A is polynomial-time (mapping) reducible to language B (written A $\leq_{\mathsf{P}}$ B) if there is a polynomial-time computable function $f : \Sigma^* \to \Sigma^*$ such that :

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

In particular, we must have $|f(x)|$ is polynomial in $|x|$.

Since $|f(x)|$ is polynomial in $|x|$, running a polytime algorithm M on $f(x)$ takes time polynomial in $|x|$. As such we have the following theorem:

Theorem 7.

- If $A \leq_{\mathsf{P}} B$ and $B \in \mathsf{P}$, then $A \in \mathsf{P}$.
- If $A \leq_{\mathsf{P}} B$ and $B \in \mathsf{NP}$, then $A \in \mathsf{NP}$.

Definition 18 (NP-Hardness)**.** A language B is NP hard, if **for all** languages $A \in \mathsf{NP}$ we have that there is a polynomial time reduction from A to B (A $\leq_{\mathsf{P}}$ B).

Another important definition is that of NP completeness :

Definition 19 (NP-Complete)**.** A language B is NP complete iff $B \in \mathsf{NP}$ **and** B is NP-hard.

Many Problems are known to be NP-Complete (See table at the end). The most important example is SAT := $\{\langle\phi\rangle \mid \phi$ a satisfiable boolean formula$\}$. The Cook-Levin theorem says that for any language $L \in \mathsf{NP}$, we have that there exists a polytime function computable function f such that $f(x) = \langle\phi\rangle$ where $\langle\phi\rangle$ is a boolean formula. Furthermore, $\langle\phi\rangle$ is satisfiable iff $x \in L$.

This last theorem explains how one can show new problems are NP-hard.

Theorem 8. If $A \leq_{\mathsf{P}} B$ and A is NP-hard, then B is NP-hard.

6.2 P versus NP

We would like to think of NP-hard problems as "harder" than P problems. However, we do not know if this is true or not! This is the famous P versus NP problem. Either way, we know that:

Theorem 9.

$$\mathsf{P} \subseteq \mathsf{NP}.$$

But some properties are dependent on the answer to this open problem.

If $\mathsf{P} = \mathsf{NP}$, then every problem[4] in P is NP-complete.

If $\mathsf{P} \neq \mathsf{NP}$, then no NP-hard problem is in P.

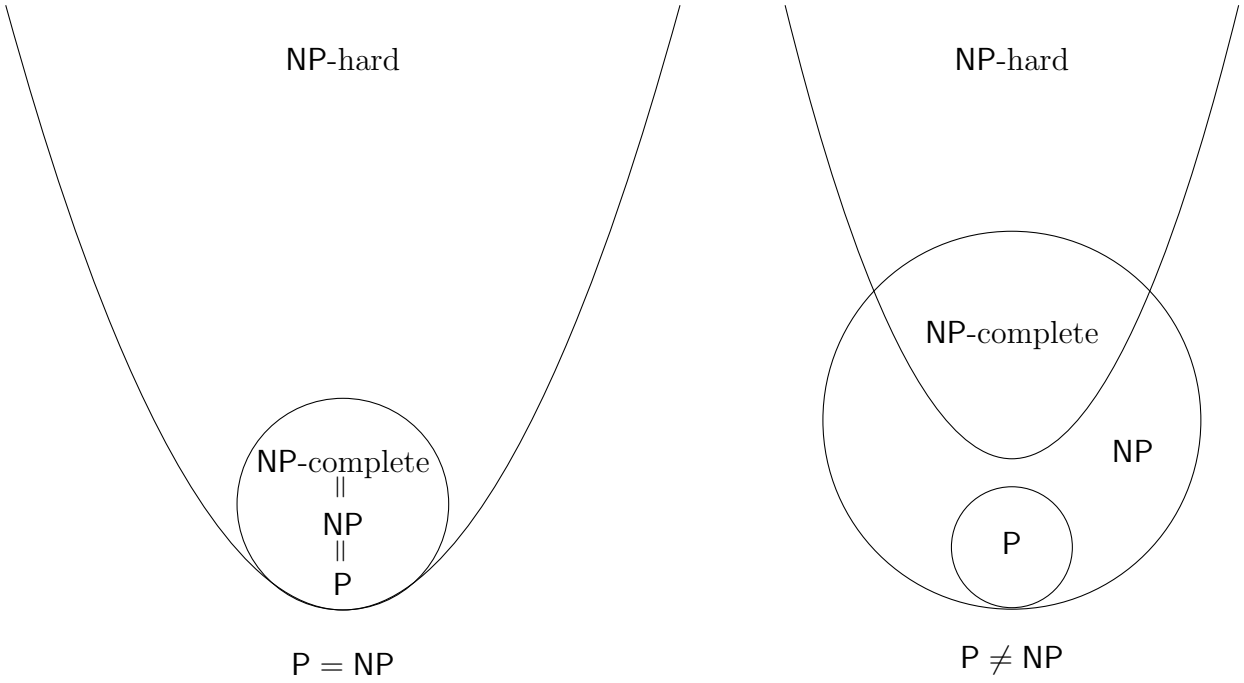


Figure 2: Venn diagrams of inclusions between classes depending on $\mathsf{P} \stackrel{?}{=} \mathsf{NP}$.

[4]other than $\varnothing$ and Σ^*. This technical restriction is by virtue of how we defined mapping reductions.

7 Proof Techniques

7.1 Problems in P

Proof Template 12: to show L is in P using polytime algorithm

1. Give pseudo-Code of a deterministic decider for L
2. Show that if $x \in L$, your algorithm accepts.
3. Show that if $x \notin L$, your algorithm rejects.
4. Show that M runs in polynomial time, namely in time $O(n^k)$ for some constant k. (No need to prove specifically what k is).

Another (probably less practical) way is to use mapping reductions.

Proof Template 13: to show L is in P using a poly-time reduction

1. Pick some $S \in \mathsf{P}$.
2. Write an algorithm that computes function f.
3. Show that given x as input, $f(x)$ can be computed in polynomial time, namely in time $O(n^k)$ for some constant k.
4. Assuming $x \in L$, prove that $f(x) \in S$.
5. Assuming $x \notin L$, prove that $f(x) \notin S$.
6. Conclude that since $L \leq_{\mathsf{p}} S$ and $S \in \mathsf{P}$, that $L \in \mathsf{P}$.

Small note: If your algorithm takes as input number a N. Your algorithm should run in time $O(\log^k(N))$ for some k to be polynomial time ! This is because if N is given in binary, its represented using $O(\log(N))$ bits.

7.2 Problems in NP

Proof Template 14: to show L is in NP using a poly-time reduction

1. Pick some $S \in \mathsf{NP}$.
2. Write an algorithm that computes function f.
3. Show that given x as input, $f(x)$ can be computed in polynomial time, namely in time $O(n^k)$ for some constant k.
4. Assuming $x \in L$, prove that $f(x) \in S$.
5. Assuming $x \notin L$, prove that $f(x) \notin S$.
6. Conclude that since $L \leq_{\mathsf{p}} S$ and $S \in \mathsf{NP}$, that $L \in \mathsf{NP}$.

Proof Template 15: to show L is in NP using a verifier

1. Give pseudo-Code of a deterministic verifier V for L. Here V takes as input x and c (the certificate).[a]
2. Show that if $x \in L$, there is some c such that V accepts (x, c).
3. Show that if $x \notin L$, for all c, V rejects (x, c).
4. Show that V runs in polynomial time, namely, in time $O(n^k)$ for some constant k. (No need to prove specifically what k is).

[a] You can choose what kind of certificate you want your algorithm to use, as long as it is of length polynomial in $|x|$.

Example 7:

Recall a *simple cycle* is a cycle with no repeated vertices. Define the following problem:

$$\textsc{CycleSize} = \{\langle G, k\rangle \mid G \text{ is a graph with a simple cycle of size } k\}$$

Claim that $\textsc{CycleSize} \in \mathsf{NP}$. To show this, we will construct a verifier.

$V_C =$ "On input x, c,

1. Checks that $x = \langle G, k\rangle$ an encoding of a graph G along with integer k. If not, reject.
2. Checks that c is a list of k distinct vertices of G. If not, reject. [a]
3. For each $i = \{1, \dots k\}$, check that (c_i, c_{i+1}) is an edge in G. If not, reject.
4. Accept.

First we will analyze runtime: claim V_C operates in polynomial time in the size of $n = |x|$ because encoding validation (steps $1 + 2$) can be done in polynomial time and step 3 consists of at most $|V|$ edge lookups, where $|V| \leq n$, so it is a polynomial time.

Now we will prove correctness:

If $x \in \textsc{CycleSize}$, then $x = \langle G, k\rangle$ where G is a graph with a simple cycle of size k. Let c be a list of the vertices in G's simple cycle of size k. When x, c are given to V_C, c will therefore pass the checks in steps 1, 2, and 3. So, V_C will accept x, c.

If $x \notin \textsc{CycleSize}$, then either $x \neq \langle G, k\rangle$ or $x = \langle G, k\rangle$ where G is a graph with no simple cycle of size k. In the first case, V_C will always reject in step 1. In the second, no matter what c is input to M_L it will not pass steps 2 and 3 (doing so would mean that c is a simple cycle of length k in G). So, V_C will reject x.

Conclude V_C is a polynomial-time verifier for $\textsc{CycleSize}$and therefore $\textsc{CycleSize} \in \mathsf{NP}$. □

[a] You could remove line $1, 2$ and say instead "On input $\langle G, k\rangle, c$ where G is a graph, k and integer and c a list of k distinct vertices from G." If you do this, you can ignore the case $x \neq \langle G, x\rangle$ in the proof of correctness.

Proof Template 16: to show L is in NP via an NTM

1. Give pseudo-Code of a non-deterministic decider M for L. Here M takes as input x only.
2. Show that if $x \in L$, there is some non-deterministic choices that makes M accept.
3. Show that if $x \notin L$, M always reject x, no matter what the non-deterministic steps are.
4. Show that M runs in polynomial time, namely in time $O(n^k)$ for some constant k. (No need to prove specifically what k is).

Example 8:

For any $L \subseteq \Sigma^*$, define

$$\mathsf{AddOne}(L) = \{w \in \Sigma^* \mid w = xay, xy \in L, a \in \Sigma\}.$$

We will show that NP is closed under AddOne. Take any $L \in \mathsf{NP}$, let M be a polytime NTM that decides L. We propose the following NTM M' for $\mathsf{AddOne}(L)$

M' = "On input w,

1. Nondeterministically partition $w = xay$ where $a \in \Sigma$
2. Run M on xy.
3. If M accepted, accept w.
4. If M rejected, reject w.

First, we claim that M' runs in polynomial time. This is because partitioning w runs in time $O(|w|)$, and then we run M on xy. Since M is a polytime decider for L, M runs in time $O(|xy|^k) = O(|w|^k)$ for some k. So M' runs in time polynomial in $|w|$.

Assume $w \in \mathsf{AddOne}(L)$. Then we can wite $w = xay$ such that $xy \in L, a \in \Sigma$. So when we non-deterministically pick this partition of w, we will run M on $xy \in L$. Since M decides L, there is some non-deterministic branch on which M will accept xy, which means M' will accept w. So there is some non-deterministic choices that make M' accept w.

Assume $w \notin \mathsf{AddOne}(L)$. Then we no matter how we split $w = xay$, we never $xy \in L$. So no matter what partition we pick on line 1, M will always reject xy (no matter what non-deterministic choices M makes). So M' always rejects w.
Since we've given a NTM that runs in polynomial time and decides $\mathsf{AddOne}(L)$, we can conclude that if $L \in \mathsf{NP}$, $\mathsf{AddOne}(L) \in \mathsf{NP}$.

7.3 NP Hardness

Proof Template 17: to show L is NP-Hard using a poly-time reduction

1. Pick some H that is NP-Hard. Write an algorithm that computes f.
2. Check that your algorithm runs in time $O(n^k)$ for some k.
3. Assuming $x \in H$, prove that $f(x) \in L$.
4. Assuming $x \notin H$, prove that $f(x) \notin L$.
5. Conclude that since $H \leq_{\mathsf{p}} L$ and H is NP-hard, that L is NP-hard.

Example 9:

We claim CycleSize is also NP-hard. To show this, we will start with the HamCycle problem, which we already know is NP-hard:

$$\{\langle G \rangle \mid G \text{ a graph with a cycle that visits each node exactly once}\}.$$

Will will show HamCycle $\leq_{\mathsf{p}}$ CycleSize. The idea is that a Hamiltonian cycle is just a simple cycle that contains all of the nodes.

Let y be some string that isn't in CycleSize. We propose the following algorithm.

$F =$ "On input x,

1. Check that $x = \langle G \rangle$ for a graph G. If not, output y.
2. Output $\langle G, |V(G)| \rangle$."

First, we claim F runs in polynomial time. Check the validity of the encoding in Step 1 is certainly polynomial time. Then Step 2 takes time $O(n)$ to count the number of vertices in G.

Next, let f be the function that F computes; we will prove correctness.

Assume $x \in$ HamCycle. Then $x = \langle G \rangle$ where G is a graph with a Hamiltonian cycle. Because G is a graph, F will pass the check in step 1. Then, because a Hamiltonian cycle for G is a simple cycle of size $|V(G)|$, we have $f(x) = \langle G, |V(G)| \rangle \in$ CycleSize.

Assume $x \notin$ HamCycle. Then either $x \neq \langle G \rangle$ or $x = \langle G \rangle$ where G is a graph with no Hamiltonian cycle. In the first case, then $f(x) = y \notin$ CycleSize. In the second case, G is a graph, so F will pass the check in Step 1. Then since a simple cycle of size $|V(G)|$ is a Hamiltonian cycle, we have $f(x) = \langle G, |V(G)| \rangle \notin$ CycleSize (if it were, then G would have a Hamiltonian cycle).

Conclude that since HamCycle $\leq_{\mathsf{p}}$ CycleSize and HamCycle is NP-hard, that CycleSize is NP-hard. □

A small note about mapping reductions: When giving a polytime mapping reduction

$A \leq_{\mathsf{P}} B$ it's cumbersome to deal with "wrong encoding" which are obviously not in A, as we did in the above. Note that if you want to show $A \leq_{\mathsf{P}} B$, as long as $B \neq \Sigma^*$, we can fix some string $y \notin B$. And then if x has the wrong format for A (for example, $x \neq \langle G \rangle$), we can just fix $f(x) = y$.

As such, when doing mapping reductions, you can say "We ignore wrong encoding by mapping them to a fix string $y \notin B$". And then when giving f and the proof of correctness you can ignore wrong encodings (see HW5, 3d for an example).

7.4 NP Completeness

Proof Template 18: to show L is NP-Complete

1. Prove that L is in NP.
2. Prove that L is NP-Hard.

Example 10:

Claim CycleSize is NP-complete.

This follows from our proofs above that CycleSize $\in$ NP and CycleSize is NP-hard. □

8 Appendix

8.1 Example Languages for Computability Theory

Finding the correct language to reduce to can make your proofs much quicker. Check that any language which is not decidable, is not both recognizable and co-recognizable, recall this is because of the theorem L and $\overline{L}$ are recognizable $\iff$ L is decidable.

	Decidable	Recognizable	co-Recognizable	Definition
A_{DFA}	Yes	Yes	Yes	$\{\langle D, w\rangle \mid D \text{ is a DFA and } D \text{ accepts } w\}$
A_{NFA}	Yes	Yes	Yes	$\{\langle D, w\rangle \mid N \text{ is a NFA and } N \text{ accepts } w\}$
E_{DFA}	Yes	Yes	Yes	$\{\langle D\rangle \mid D \text{ is a DFA and } L(D) = \varnothing\}$
EQ_{DFA}	Yes	Yes	Yes	$\{\langle D, E\rangle \mid D, E \text{ are DFAs and } L(D) = L(E)\}$
A_{TM}	No	Yes	No	$\{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ accepts } w\}$
E_{TM}	No	No	Yes	$\{\langle M\rangle \mid M \text{ is a TM and } L(M) = \varnothing\}$
EQ_{TM}	No	No	No	$\{\langle M, N\rangle \mid M, N \text{ are TMs and } L(M) = L(N)\}$
$HALT_{\mathsf{TM}}$	No	Yes	No	$\{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ halts on } w\}$
REG_{TM}	No	No	No	$\{\langle M\rangle \mid M \text{ is a TM and } L(M) \text{ is a regular language}\}$
ALL_{TM}	No	No	No	$\{\langle M\rangle \mid M \text{ is a TM and } L(M) = \Sigma^*\}$

8.2 Example Languages for Complexity Theory

Omitted entries are unknowns, dependent on the solution to the P versus NP problem. The certificate listed is a reminder of how we constructed a verifier for each language. There are always other possible certificates or verifiers.

	In P	In NP	NP-Hard	
COMPOSITE	Yes	Yes	·	$\{\langle x\rangle \mid x$ a composite number$\}$ Certificate: a non-trivial divisor of x.
PRIME	Yes	Yes	·	$\{\langle x\rangle \mid x$ is a prime number$\}$ Primes in P via complicated proof from [AKS]
HAMCYCLE	·	Yes	Yes	$\{\langle G\rangle \mid G$ a graph with a cycle that visits each node exactly once$\}$ Certificate: such a cycle.
HAMPATH	·	Yes	Yes	$\{\langle G, s, t\rangle \mid G$ a graph with a path from s to t that visits each node exactly once$\}$ Certificate: such a path from s to t.
SAT	·	Yes	Yes	$\{\langle \phi\rangle \mid \phi$ a satisfiable Boolean formula$\}$ Certificate: a satisfing assignment.
CNF-SAT	·	Yes	Yes	$\{\langle \phi\rangle \mid \phi$ a satisfiable Boolean formula that is a CNF$\}$ Certificate: a satisfing assignment.
3SAT	·	Yes	Yes	$\{\langle \phi\rangle \mid \phi$ a satisfiable CNF where each clause has 3 literals$\}$ Certificate: a satisfing assignment.