COMS W3261: Computer Science Theory
Omer Mustel, Alex Lindenbaum (with huge credits to Rahul Gosain and Benjamin Kuykendall, 2018 TAs)

Handout 11: Final Review

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^*)

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 $f(x) = y$.) For finite sets, it should be clear why a bijection says the sets must have the same number of elements. For infinite sets, you can take this to be a definition.

Proof Template 1: to show $|S| \leq |T|$

1. Construct a function f from S to T.
2. Check that f maps distinct elements of S to distinct elements of T.
3. Conclude that f is an injective function, so $|S| \leq |T|$.

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 proof template above: 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 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 $|\mathbb{Z}| = |\mathbb{Z}_{\geq 0}|$. □

This same proof template can be used in two important ways. Consider a set S. Then

To prove S is countable, show that $|S| \leq |\mathbb{N}|$ (or some other countable set).
To prove S is uncountable, show that $|S| \geq |\mathcal{P}(\mathbb{N})|$ (or some other uncountable set).

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.

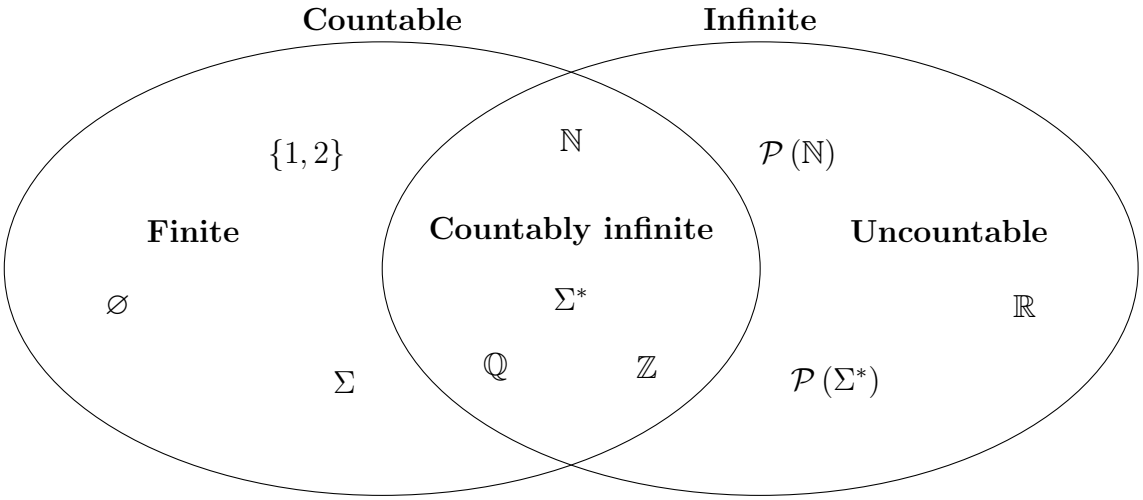


Figure 1: Relationships between different types of cardinalities.

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 $\mathcal{P}(\mathbb{N})$.

Application to Recognizability

Let $\mathbb{M}$ be the set of all Turing machines. Consider the encoding map $f(M) = \langle M \rangle$. Since 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^*: countably infinite.
$L \in \Sigma^*$ (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 the map cannot be surjective. In other words, there is some language $S \in \mathcal{P}(\Sigma^*)$ such that no Turing machine recognizes S. This same proof technique can be used to show that any uncountable set of languages contains some unrecognizable language.

Turing Machines

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. But 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 state q'. It repeats, stopping only if one of q_{accept} or q_{reject} is reached.

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. psuedocode) to describe a machine.

Examples of permissible syntax and operations include:

- 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 replacing 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, 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 N decides $\overline{L}$. To prove correctness:

$w \notin L \implies M$ 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 a 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 $\varnothing$.

Another place we have to account for this effect is in the recognizer for A_{TM}. There we either use a technique called *dovetailing* to avoid looping, or take advantage of NTMs. See more details in lecture 18 and handout 8.

Example

Consider the language

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

We will give four Turing machines, in different formats, that all recognize L. This is intended to show you the wide range of formats that are acceptable for describing algorithms.

High level verbal description

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

Formal description

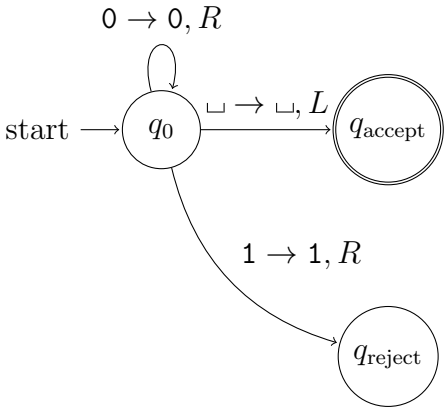
This high level description corresponds to the following TM (described in a formal level):

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

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:



Real world programming language

```
public boolean isInLanguage(String x) {
    for (int i = 0; i < x.length(); i++) {
        if (x.charAt(i) !='0') {
            return false;
        }
    }
    return true;
}
```

Pseudocode (unnatural algorithm)

The above algorithms were straight-forward and natural. However, note that there are (infinitely many) other algorithms that recognize the same language. Here is one of them.

M = "On input $x \in \{\texttt{0}, \texttt{1}\}^*$,

1. If x starts with 1, loop forever.
2. Check if x contains any 1's, and if so reject.
3. Calculate the 7^{th} digit of π and if it is even accept, else reject"

Although you would never write such a convoluted algorithm yourself, you should remember that recognizers do not need to be reasonable: if you are given a recognizer in the problem, it could very well be doing "silly" operations like this under the hood.

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 $\leq_{\mathsf{p}}$.

Turing reduction say $A \leq_{\mathsf{T}} B$ (or "A is Turing-reducible to B") when there is some algorithm N, which has access to an oracle M and

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

Mapping reduction say $A \leq_{\mathsf{m}} B$ (or "A is mapping-reducible to B") when some computable function f is such that

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

We know that given a mapping reduction, there is also a Turing reduction. 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, but no mapping reduction. In the next section, we will go over how both types of reductions can be used to solve problems.

Proof Techniques

In this section, you'll see that there are three techniques that you should consider when you approach a problem: direct demonstrations, reductions or Rice's theorem. Not all will work on every problems, but the proof templates below should help you get the hang of it.

We note that in all examples we explicitly check whether the input is a valid encoding, and deal with it appropriately. However, you may use the convention we often used in class of omitting this check (assuming that it's implicitly there, since it's easy to deal with).

Recognizability

Recall that a language L is recognizable if we can construct a Turing machine that accepts only all the strings in L. That is, it accepts every string in L and rejects or loops infinitely on all strings that are not in L. There are two ways you can prove a language is recognizable.

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

1. Write a description 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.

Example 3:

Recall the language

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

We will show the complement $\overline{\mathrm{E}_{\mathsf{TM}}}$ 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 $\overline{\mathrm{E}_{\mathsf{TM}}}$, that $\overline{\mathrm{E}_{\mathsf{TM}}}$ is recognizable. □

A second approach to showing recognizability is by showing a mapping reduction to another language you already know is recognizable.

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

1. Pick a recognizable language B. Write an algorithm that computes f.
2. Assuming $x \in L$, prove that $f(x) \in B$.
3. Assuming $x \notin L$, prove that $f(x) \notin B$.
4. Conclude that since $L \leq_{\mathsf{m}} B$ and B is recognizable, L must be too.

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. That is, a decider always gives us back an answer in finite time for all possible inputs. We can use the same proof strategy we used above and supply the decider explicitly, or we can use a reduction.

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

1. Write a description 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

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

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

M_{SAT} ="On input x

1. Check that $x = \langle \phi \rangle$, an encoding of a Boolean formula of some number of variables n. If not, reject.
2. 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.
3. After iterating through all assignments (without finding a satisfying assignment), then reject.

Now we will prove correctness:

Assume $x \in$ SAT. Then $x = \langle \phi \rangle$ where ϕ is a Boolean formula with a satisfying assignment. This means that x will pass the check in Step 1. Then, x 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_{SAT} will accept x.

Assume $x \notin$ SAT. Then either $x \neq \langle \phi \rangle$ for any Boolean formula ϕ or $x = \langle \phi \rangle$ where ϕ is a Boolean formula with no satisfying assignment. In the first case, x will be rejected in Step 1. In the second, every assignment tested in Step 2 will not evaluate to True (since there is no satisfying assignment). So, M_{SAT} will reach Step 3 and reject x.

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

Note that a proof similar to this one can be given to show that in fact every langauge in NP is decidable.

You can also show decidability by using a Turing reduction, utilizing a decider for another langauge you already know how to decide:

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

1. Pick some decidable language B and assume you have a decider M_B for it.
2. Demonstrate that you can use M_B to create a decider M_L for L.
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.

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 use a reduction to prove that L is "at least as difficult" as a language.

Proof Template 6: 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. Demonstrate that you can use M_L to create a decider 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 E_{TM} is undecidable; we already know that A_{TM} is undecidable via the diagonalization proof in class. For reference, here are the definitions of the two languages:

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

In order to carry out the reduction, assume 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 x,

1. Check if $x = \langle M, w \rangle$, an encoding of a TM M and string w. If not, reject.
2. Create an encoding of the following Turing machine:

   $M' =$"On input z, run M on w and output the same."
3. Run M_E on input $\langle M' \rangle$. If M_E accepts, then reject and vice versa."

Now we will prove correctness:

Assume $x \in A_{TM}$. Then $x = \langle M, w \rangle$ where M is a TM that accepts when run on input w. This means that the constructed M' will accept on all inputs. 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 $x \notin A_{TM}$. Then either $x \neq \langle M, w \rangle$ or $x = \langle M, w \rangle$ where M is a TM that does not accept w. In the first case, x is rejected in Step 1. In the second, the constructed M' accepts no strings. Namely, $L(M') = \emptyset$. So, M_E will accept when run on $\langle M' \rangle$ (since it is a decider for E_{TM}), and M_A will therefore reject x.

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

An alternative strategy is to use Rice's Theorem. Define a *property of recognizable languages* to be any set of TM encodings $P \subseteq \{\langle M \rangle : M \text{ is a TM}\}$, where any two TMs that recognizae the same language are both in P or both not in P (namely, for any pair of TMs M_1, M_2 with $L(M_1) = L(M_2)$ it holds that $\langle M_1 \rangle \in P \iff \langle M_2 \rangle \in P$).

Theorem 1. (Rice's Theorem Let P be a property of recognizable languages, such that P is non-trivial (namely there exists at least one TM M with $\langle M \rangle \in P$ and at least one TM M' with $\langle M' \rangle \notin P$). Then P is not decidable.

Proof Template 7: to show L is undecidable using Rice's theorem

1. Argue that L is a property of recognizable languages, namely $L \subseteq \{\langle M \rangle : M \text{ is a TM}\}$ (a set of TMs).
2. Demonstrate one TM whose encoding is in L.
3. Demonstrate one TM whose encoding is not in L.
4. Conclude that by Rice's theorem, L is undecidable.

Example 6: E_{TM} is not decidable

Recall the language

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

E_{TM} is clearly a property of recognizable langauges, from its definition.

To show that E_{TM} is non-empty, note that $\langle M_\varnothing \rangle \in \mathrm{E}_{\mathsf{TM}}$ where $M_\varnothing$ is the TM that immediately rejects all inputs.

To show that E_{TM} is not all TM encodings, note that $\langle M_{\text{all}} \rangle \in \mathrm{E}_{\mathsf{TM}}$ where M_{all} is the TM that immediately accepts all inputs.

Conclude that E_{TM} is a non-trivial property of recognizable languages, so by Rice's theorem E_{TM} must be undecidable.

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 use a mapping reduction to make a claim about all possible Turing machines. The reduction strategy looks very similar.

Proof Template 8: to show L is unrecognizable via mapping reduction

1. Pick an unrecognizable language U. Write an algorithm that computes f.
2. Assuming $x \in U$, prove that $f(x) \in L$.
3. Assuming $x \notin U$, prove that $f(x) \notin L$.
4. Conclude that since $U \leq_{\mathsf{m}} L$ and U is unrecognizable, L must be unrecognizable too.

We can also prove unrecognizability by relying on the theorem that a language L is decidable if and only if both L and its complement $\overline{L}$ are recognizable. The following strategy works only for languages that are undecidable, but their complement is recognizable.

Proof Template 9: To show L is unrecognizable by analyzing L and $\overline{L}$

1. Prove that L is undecidable (e.g., using Rice's theorem).
2. Prove that $\overline{L}$ is recognizable.
3. Conclude that L is therefore un-recognizable (since it it were, then $L, \overline{L}$ would both be recognizable, so L would be decidable).

Finally, We could use refined Rice's theorem for to prove unrecognizability, in a similar way to the use of Rice's theorem to prove undecidability. The condition of refined Rice's theorem is the same as the one in the standard Rice's theorem (and the proof is the same too), but the conclusion is stronger. This is bonus material (in notes of Lecture 21) which is not required for class, but you are welcome to use it.

Theorem 2. (Refined Rice's Theorem) Let P be a property of recognizable languages, such that P is non-trivial. Let $\langle M_\varnothing \rangle$ be the TM that immediately rejects all inputs.

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

Proof Template 10: to show L is unrecognizable using Refined Rice's theorem

1. Argue that L is a property of recognizable languages, namely $L \subseteq \{\langle M \rangle : M \text{ is a TM}\}$ (a set of TMs).
2. Demonstrate one TM whose encoding is in L.[a]
3. Demonstrate one TM whose encoding is not in L.
4. Demonstrate that $\langle M_\varnothing \rangle \in L$
5. Conclude that by Refined Rice's theorem, L is unrecognizable.

[a] This also follows from step 4 so could be skipped

Complexity

We have a very rich theory of *computability* saying 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. Complexity is the study of how much resources are needed to solve a problem. Efficiency is always measured in terms of the length of the input (i.e. the number of characters), which we denote by n.

$\mathsf{TIME}(f(n))$: 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 $O(f(n))$.

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

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

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 polynomial time verifiers; so $A \in \mathsf{NP}$ when there exists a V, a polynomial time verifier such that

$$x \in A \iff \exists c \text{ such that } V \text{ accepts } \langle x, c \rangle$$

Poly-time mapping reduction: a mapping reduction that runs in time $O(n^k)$ for some constant k, write $A \leq_{\mathsf{p}} B$ or "A is polynomial-time mapping-reducible to B".

NP-Hard: a problem L is NP-hard when for all $S \in \mathsf{NP}$, $S \leq_{\mathsf{p}} L$.

NP-Complete: a problem is NP-complete when it is NP-hard and in NP.

Problems in NP

Proof Template 11: to show L is in NP using a verifier

1. Write a verifier V as an algorithm that takes $\langle x, c \rangle$.
2. Show that V runs in time $O(|x|^k)$ for some k.
3. Assuming $x \in L$, prove that $\exists c$ such that $V(x, c)$ accepts.
4. Assuming $x \notin L$, prove that $\forall c$ that $V(x, c)$ must reject.
5. Conclude that since V is a polynomial-time verifier for L, that $L \in \mathsf{NP}$.

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.
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 $\textsc{CycleSize} \in \mathsf{NP}$. □

Proof Template 12: to show L is in NP via an NTM

1. Write an NTM M.
2. Show that M runs in time $O(n^k)$ for some k.
3. Assuming $x \in L$, prove that some branch of M accepts x.
4. Assuming $x \notin L$, prove that no branch of M accepts x.
5. Conclude that since M decides L in time $O(n^k)$, that $L \in \mathsf{NTIME}(n^k) \subseteq \mathsf{NP}$.

NP Hardness

Proof Template 13: 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.

Below is an example of a reduction proving NP-hardness according to the above template. We note that given how little time we had for complexity this year, you will not be asked to produce a full proof of NP hardness on the exam. But the example is useful since you are expected to understand what a reduction is, the definition of NP-hardness, etc.

We also note that as before, here we take care to deal with bad encodings explicitly, but it is also ok in this class to ignore it and implicitly assume that strings that are not a valid encoding are dealt with implicitly (for the case of standard encodings that are easy to check).

Example 8:

We claim CycleSize is also NP-hard. To show this, we start with the HamCycle problem

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

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.

In order to properly deal with bad encodings, we will need a string that is not in the language. We can construct one explicitly: let D_2 be the graph with 2 vertices and no edges. Clearly $\langle D_2, 2\rangle \notin$ CycleSize.

We propose the following algorithm.

$F =$ "On input x,

1. Check that $x = \langle G\rangle$ for a graph G. If not, output $\langle D_2, 2\rangle$.
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 x is caught in step 1 and the output $f(x) = \langle D_2, 2\rangle \notin$ 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. □

NP Completeness

Proof Template 14: to show L is NP-Complete

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

Example 9:

Claim CycleSize is NP-complete.

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

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 P $\subseteq$ NP. But some properties are dependent on the answer to this open problem.

If P = NP, then every problem[1] in P is NP-complete.

If P = NP, then no NP-hard problem is in P.

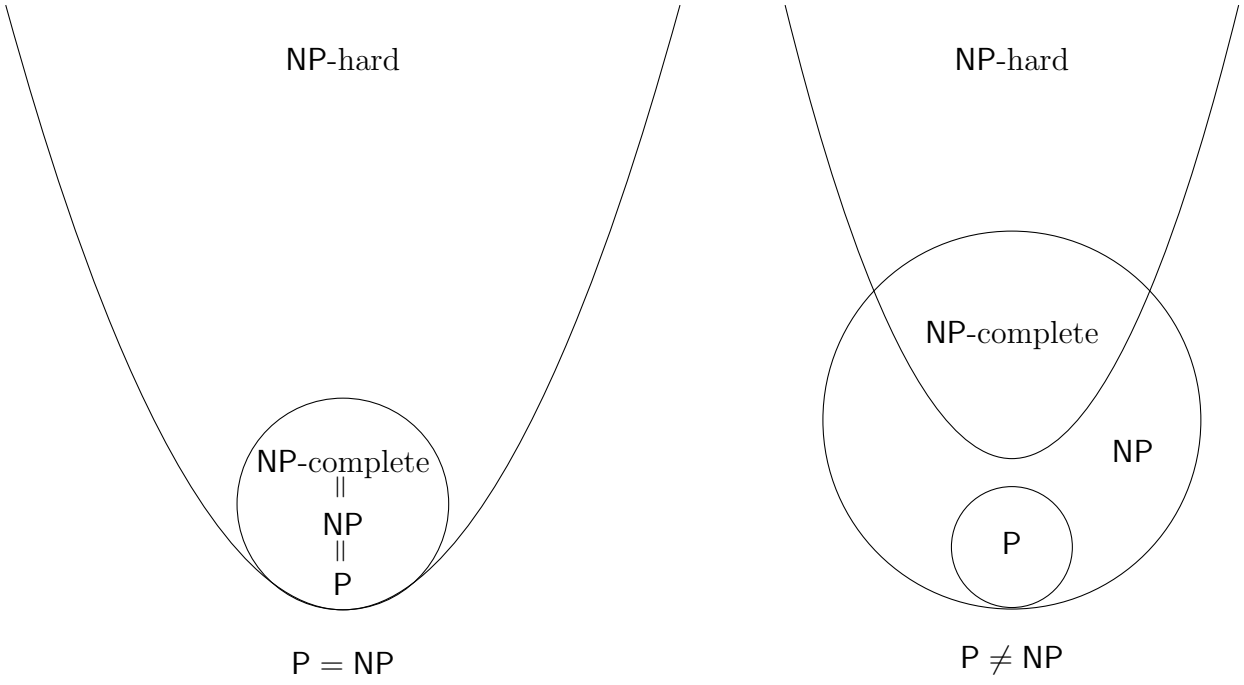


Figure 2: Venn diagrams of inclusions between classes depending on P $\stackrel{?}{=}$ NP.

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

Appendix

Example Languages for Computability Theory

Finding the correct language to reduce to can make your proofs much quicker. Omitted entries were not mentioned in class (you could work them out yourself as practice, but some are hard).

	Decidable	Recognizable	co-Recognizable	Definition
A_{DFA}	Yes	Yes	Yes	$\{\langle D, w\rangle \mid D \text{ is a DFA and } D \text{ accepts } w\}$
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_{CFG}	Yes	Yes	Yes	$\{\langle G, w\rangle \mid G \text{ is a CFG and } G \text{ accepts } w\}$
E_{CFG}	Yes	Yes	Yes	$\{\langle G\rangle \mid G \text{ is a CFG and } L(G) = \varnothing\}$
EQ_{CFG}	No	·	·	$\{\langle G, H\rangle \mid G, H \text{ are CFGs and } L(G) = L(H)\}$
ALL_{CFG}	No	·	·	$\{\langle G\rangle \mid G \text{ is a CFG and } L(G) = \Sigma^*\}$
AMB_{CFG}	No	·	·	$\{\langle G\rangle \mid G \text{ is an ambiguous CFG}\}$
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	$\{\langle M\rangle \mid M \text{ is a TM and } L(M) = \Sigma^*\}$

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 other 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.
HAMCYCLE	·	Yes	Yes	$\{\langle G \rangle \mid G$ a graph with a cycle that visits each node exactly once$\}$ Certificate: such a cycle.
INDSET	·	Yes	Yes	$\{\langle G, k \rangle \mid G$ a graph with a set of k vertices, none of which are adjacent$\}$ Certificate: such a set of vertices.
SAT	·	Yes	Yes	$\{\langle \phi \rangle \mid \phi$ a satisfiable Boolean formula$\}$ 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.
SUDOKU	·	Yes	Yes	$\{P \mid P$ is a $n^2 \times n^2$, partially filled with integers from 1 to n^2, s.t. there ex Certificate: a complete and valid grid.