

Computer Science Theory

Prof. Tal Malkin

Scribe: Ekene Ezeunala

0. Introduction and Prerequisites (06/09)

COMS W3203 Discrete Mathematics is a prerequisite for this class, but just in case, the following topics from that class are most useful in CS Theory.

Boolean logic

Here we use capitalised letters such as (P, Q, R) to denote logical variables, and use 1 for TRUE and 0 for FALSE. Important operations in Boolean logic include $(\neg)$ = NOT, $(\wedge)$ = AND, $(\vee)$ = OR, $(\oplus)$ = XOR, $(\implies)$ = IMPLIES, $(\iff)$ = EQUIVALENT.

Example 0.1. The following are examples of Boolean operations:

- AND: $0 \wedge 0 = 0, 0 \wedge 1 = 0, 1 \wedge 0 = 0, 1 \wedge 1 = 1.$
- OR: $0 \vee 0 = 0, 0 \vee 1 = 1, 1 \vee 0 = 1, 1 \vee 1 = 1.$
- XOR: $0 \oplus 0 = 0, 0 \oplus 1 = 1, 1 \oplus 0 = 1, 1 \oplus 1 = 0.$
- IMPLIES: $0 \implies 0 = 1, 0 \implies 1 = 1, 1 \implies 0 = 0, 1 \implies 1 = 1.$
- EQUIVALENT: $0 \iff 0 = 1, 0 \iff 1 = 0, 1 \iff 0 = 0, 1 \iff 1 = 1.$
- NOT: $\neg 0 = 1, \neg 1 = 0.$

Example 0.2. The following hold for all logical variables P, Q and R :

- | | |
|--|---|
| • $P \wedge Q \iff \neg(\neg P \vee \neg Q)$ | • $P \oplus Q \iff \neg(P \iff Q)$ |
| • $P \implies Q \iff \neg P \vee Q$ | • $P \wedge (Q \vee R) \iff (P \wedge Q) \vee (P \wedge R)$ |
| • $(P \iff Q) \iff ((P \implies Q) \wedge (Q \implies P))$ | • $P \vee (Q \wedge R) \iff (P \vee Q) \wedge (P \vee R)$ |

Predicates, quantified expressions, and De Morgan's law

A *proposition* is a declarative statement that is either true or false. $1 + 1 = 2$, $0 = 1$, and $4! \leq 3!$ are all propositions. $4 - 3$ is not.

A *predicate* is a statement that is based on variables which when assigned, make the statement true or false. $P(x) = "x \text{ is even}"$ is a predicate, where $P(x)$ is true iff $2 \mid x$ and $x \in \mathbb{Z}$. The symbol $\exists$ is read "exists". The proposition $\exists x P(x)$ is true if there exists a value of x such that the predicate $P(x)$ is true.

Example 0.3. The following are examples of quantified statements. Which of them are true?

- $(\forall x)(x + 0 = x)$
- $(\exists x)(x + 5 = 20)$
- $(\forall x \exists y)(x + y = 0)$
- $(\exists x \forall y)(x + y \geq y)$

Theorem 0.1 (De Morgan’s Law for Quantifiers). *For any predicate $P(x)$, $\neg \forall x : P(x) \iff \exists x : \neg P(x)$, and similarly $\neg \exists x : P(x) \iff \forall x : \neg P(x)$.*

Theorem 1 is another way of saying that, given a universe U of possible values x could take,

$$\neg \bigwedge_{x \in U} P(x) \Leftrightarrow \bigvee_{x \in U} \neg P(x), \quad \text{and} \quad \neg \bigvee_{x \in U} P(x) \Leftrightarrow \bigwedge_{x \in U} \neg P(x).$$

Sets

Sets are, informally, a collection of elements. They are characterised by the elements they contain; for a set A , we can say that x is in A by writing $x \in A$, or that x is not in A by writing $x \notin A$. Sets can be defined both explicitly (by exhaustively listing all elements in the set), or implicitly (by defining a property that all elements in the set satisfy).

Example 0.4. The following are examples of sets:

- $\{1, 2, 3\}$.
- $\{\text{Tired, Turing, Teaches, Terribly}\}$.
- The prime numbers.
- The real numbers.

Note that the ordering in a set is irrelevant: $\{\text{Tired, Turing, Teaches, Terribly}\}$ is exactly the same set as $\{\text{Terribly, Teaches, Tired, Turing}\}$, just written in two different ways. This is in contrast to a sequence, where order matters—the sequence $(\text{Tired, Turing, Teaches, Terribly})$ is not the same as $(\text{Terribly, Teaches, Tired, Turing})$. Also note that a set only consists of distinct elements—each element is either in the set or not—it can’t be in the same set twice (so the set $\{\text{Tired, Tired, Turing, Teaches, Terribly}\}$ is just the set $\{\text{Tired, Turing, Teaches, Terribly}\}$).

Definition 0.2. *The set with no elements is called the empty set, and is written $\emptyset$ or $\varnothing$ or $\{\}$.*

Sets that are not explicitly defined can be described verbally or written in set-builder notation. Set-builder notation follows the format $\{x \mid \text{some condition applied to } x\}$ or $\{x : \text{some condition applied to } x\}$, read as the set of all x such that some condition applies to x .

Example 0.5. Here are some examples of sets constructed using set-builder notation:

- The set of integers greater than 5 can be written as $\{x \in \mathbb{Z} \mid x > 5\}$, or $\{x \in \mathbb{Z} : x > 5\}$, read as “all x in $\mathbb{Z}$ such that x is greater than 5.”
- The prime numbers can be formally defined by $\{x \in \mathbb{N} : \nexists y \in \mathbb{N} : 1 < y < x, y \mid x\}$, where $y \mid x$ means y divides x or x is divisible by y .
- The set $\{x + y \mid x, y \in \mathbb{Z}, x > 2, y < 1\}$. (We can show that this set is the same as $\{n \mid n \in \mathbb{Z}\} = \mathbb{Z}$.)

Example 0.6. The following are true statements about the $\notin$ operation: $0 \in \{0, 1, 2, 3, 4\}$, $\text{dog} \in \{\text{cat, bird, tiger}\}$, $3 \in \{x \mid x < 5\}$.

Set cardinality. The *cardinality* of a set A , written as $|A|$, can be thought of as a measure of how “big” a set is. For finite sets, this is simply the number of elements in the set.

Example 0.7. If $A = \{1, 2, 3\}$, then $|A| = 3$. If $B = \emptyset$, then $|B| = 0$.

Definition 0.3 (Union and intersection). *The union of the two sets A and B , written as $A \cup B$, is the set of all x that is either in A or B . Using set-builder notation, $A \cup B = \{x \mid x \in A \vee x \in B\}$. The intersection of two sets A and B , written as $A \cap B$, is the set of all x that is either in A or B . Using set-builder notation, $A \cap B = \{x \mid x \in A \wedge x \in B\}$.*

Example 0.8. $\{1, 2, 3\} \cup \{2, 3, 4\} = \{1, 2, 3, 4\}$, while $\{1, 2, 3\} \cap \{2, 3, 4\} = \{2, 3\}$.

Definition 0.4. *The Cartesian product of two sets A and B , written as $A \times B$, is the set of pairs (p, q) such that $p \in A$ and $q \in B$. Using set-builder notation, $A \times B = \{(p, q) : p \in A, q \in B\}$. Note that $|A \times B| = |A||B|$.*

Example 0.9. $\{1, 2, 3\} \times \{4, 5\} = \{(1, 4), (1, 5), (2, 4), (2, 5), (3, 4), (3, 5)\}$.

Definition 0.5. *For two sets A and B , we say $A \subseteq B$ (or “ A is a subset of B ”) if, for all $x \in A$, $x \in B$. For all such sets, we can also say $B \supseteq A$ (or “ B is a superset of A ”). Note that $A \subseteq A$ for all sets A .*

Example 0.10. $\{0\} \subseteq \{0, 1\}$.

Definition 0.6. *The powerset of a set A , or $\mathcal{P}(A)$, is the set of all possible subsets of A .*

Example 0.11. $\mathcal{P}(\{1, 2, 3\}) = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{2, 3\}, \{3, 1\}, \{1, 2, 3\}\}$.

Functions

A function $f : A \rightarrow B$ is a subset of $A \times B$ satisfying the following: f is a function precisely when each element $x \in A$ is matched to one value $f(x) \in B$. The set of possible inputs to f is called its domain. The outputs of f come from a set called its range.

A property whose domain is a set of k -tuples $A \times A \times \cdots \times A$ is called a k -ary relation on A . A common case is a 2-ary relation, called a binary relation. A special kind of binary relation, called an equivalence relation, captures the notion of two objects being equal in some feature. A binary relation R is an equivalence relation if R satisfies three conditions:

1. R is *reflexive* if for every x , xRx ;
2. R is *symmetric* if for every x and y , xRy implies yRx ; and
3. R is *transitive* if for every x , y , and z , xRy and yRz implies xRz .

Definition 0.7. *A function is called injective or one-to-one if f maps distinct elements of its domain to distinct elements, i.e. $f(x_1) = f(x_2) \implies x_1 = x_2$.*

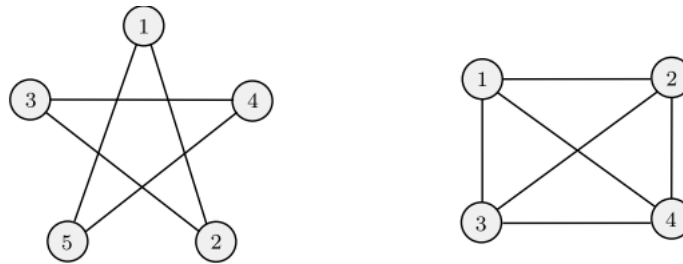
Definition 0.8. A function is called *surjective* or *onto* if f maps an element x to every element y . For every y , there is an x such that $f(x) = y$.

Definition 0.9. A function is called *bijective* if it is both *injective* and *surjective*.

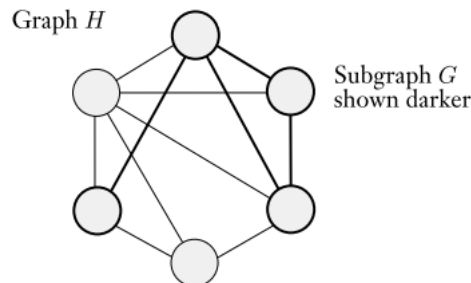
Theorem 0.10. The composition of injective functions is also injective. The composition of surjective functions is surjective. The composition of bijective functions is bijective.

Graphs

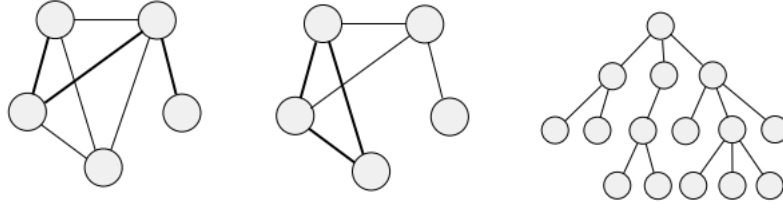
An *undirected graph*, or simply a graph, is a set of points with lines connecting some of the points. The points are called *nodes* or *vertices*, and the lines are called *edges*, as shown in the figure below. The *degree* of a node in a graph is the number of edges associated with the node.



We say that graph G is a subgraph of graph H if the nodes of G are a subset of the nodes of H , and the edges of G are the edges of H on the corresponding nodes. The following figure shows a graph H and a subgraph G .



A *path* in a graph is a sequence of nodes connected by edges. A simple path is a path that doesn't repeat any nodes. A graph is *connected* if every two nodes have a path between them. A path is a *cycle* if it starts and ends in the same node. A *simple cycle* is one that contains at least three nodes and repeats only the first and last nodes. A graph is a *tree* if it is connected and has no simple cycles, as shown below. A tree may contain a specially designated node called the *root*. The nodes of degree 1 in a tree, other than the root, are called the *leaves* of the tree.



The first figure shows a path in a graph, the second shows a cycle in the graph, and the third shows a tree.

Proof Techniques

Proof by construction. One way to prove a statement about the existence of a particular object is by demonstrating how to construct the object. This technique is a *proof by construction*.

Example 0.12. *A graph is k regular if every node in the graph has degree k . Show that for every even $n > 2$, there exists a 3-regular graph with n nodes.*

Let n be an even number greater than 2. Construct graph $G = (V, E)$ with n nodes as follows. The set of nodes of G is $V = \{0, 1, \dots, n-1\}$, and the set of edges of G is the set

$$E = \{\{i, i+1\} \mid \text{for } 0 \leq i \leq n-2\} \cup \{\{n-1, 0\}\} \cup \{\{i, i+n/2\} \mid \text{for } 0 \leq i \leq n/2-1\}.$$

Picture the nodes of this graph written consecutively around the circumference of a circle. In that case, the edges described in the top line of E go between adjacent pairs around the circle. The edges described in the bottom line of E go between nodes on opposite sides of the circle. This mental picture clearly shows that every node in G has degree 3.

Proof by contradiction assumes that the statement to be proved is false, and arrives at a contradiction, thus concluding the statement was true. It is often applied to statements of the form $P \implies Q$. Here is an example.

Example 0.13. *If $3n+2$ is odd, then n is odd.*

Assume for the sake of contradiction that $3n+2$ is odd and n is even. Since n is even, there exists some $k \in \mathbb{Z}$ such that $n = 2k$. Thus $3n+2 = 3(2k)+2 = 2(3k+1)$, which is even. But this would mean that $3n+2$ is both odd and even, which is a contradiction. So it must be true that if $3n+2$ is odd, then n is odd.

Proof by contrapositive. To prove $P \implies Q$ by contradiction, assume $\neg Q$ and derive that this must mean $\neg P$. This proves $\neg Q \implies \neg P$, which is logically equivalent to $P \implies Q$.¹

Example 0.14. *If x^2 is even, then x is even.*

Assume that x is not even, i.e. x is odd. Then $x \cdot x = x^2$ is odd, so x^2 is not even. Therefore if x^2 is even, then x is even.

Proof by induction. To prove that a propositional function $P(n)$ is true for all nonnegative integers n , we complete two steps:

¹Note that this is not the same as a proof by contradiction, which typically asserts that if the statement P is true, then $\neg P \equiv \text{FALSE}$.

1. *Basis step.* Verify that $P(0)$ is true.
2. *Inductive step.* Show that the conditional statement $P(k) \rightarrow P(k+1)$ is true for all positive $k \in \mathbb{Z}$ by assuming that $P(k)$ is true and using this assumption to show that $P(k+1)$ is also true.

Definition 0.11 (The principle of mathematical induction). *The base case $P(0)$ and the induction step together imply that $P(n)$ for any natural number n . In second-order logic, we can write this as*

$$\forall P (P(0) \wedge \forall k (P(k) \rightarrow P(k+1)) \rightarrow \forall n (P(n))) .$$

The above can be generalised *mutatis mutandis* by adjusting the basis case.

Strong induction strengthens the inductive hypothesis to assume $P(1), \dots, P(k)$ (rather than just $P(k)$), and uses this assumption to prove $P(k+1)$. This is very useful when $P(k+1)$ depends on some combination of $P(1), \dots, P(k)$.

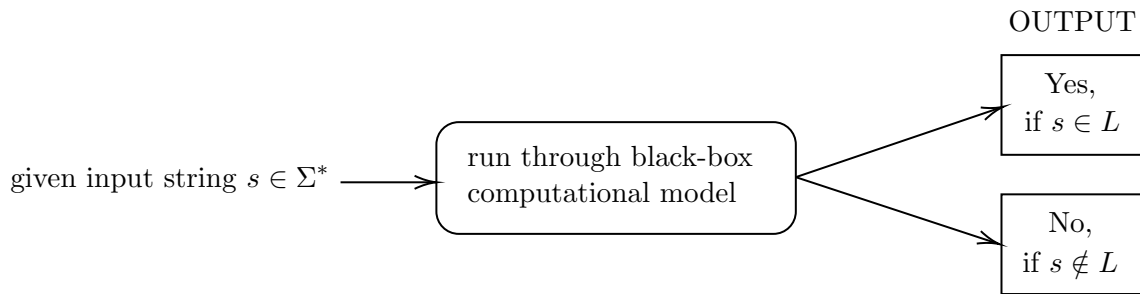
Definition 0.12 (Strong induction). $(P(1) \wedge \forall k ([P(1) \wedge \dots \wedge P(k)] \Rightarrow P(k+1))) \Leftrightarrow \forall n P(n)$.

Strings and Languages

Definitions. The following terms are occasionally useful.

- An *alphabet* Σ is a finite, non-empty set of symbols. Some examples include $\Sigma_1 = \{0, 1\}$, $\Sigma_2 = \{a, b, c\} = \{b, a, c\}$, $\Sigma_3 = \{a, b, \dots, z\}$.
- A *string* over an alphabet Σ is a finite sequence of symbols/characters from Σ . For example, if $\Sigma = \{0, 1\}$, then 00 and 101 are both strings over Σ . Likewise, if $\Sigma_1 = \{a, b, c\}$, then both *abc* and *bac* are both strings over Σ_1 (note that *abc* $\neq$ *bac* even though $\{a, b, c\} = \{b, a, c\}$).
- The *length* of a string s , denoted $|s|$, is the number of characters in the string. Note that $|101| = 3$ and $|00| = 2$. The empty string, denoted ε , is the string of length zero: $|\varepsilon| = 0$.
- The *concatenation* of two strings s and t , denoted $s \circ t$, is st . As an example, $100 \circ 01 = 10001$. Note that $|s \circ t| = |s| + |t|$.
- The set Σ^k is defined as the set of all strings over Σ that have length k . For example, $\{0, 1\}^2 = \{00, 01, 10, 11\}$, and $\{0, 1\}^0 = \{\varepsilon\}$. Note that $\{0, 1\}^2 \cup \{0, 1\}^0 = \{\varepsilon, 00, 01, 10, 11\}$.
- The set Σ^* is defined as the set of all possible strings over Σ , i.e. $\Sigma^* = \bigcup_{k=0}^{\infty} \Sigma^k$. Note that $|\Sigma^*| = \infty$. For example, $\{0, 1\}^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, 001, \dots\}$.
- A language L over an alphabet Σ is a set of strings $L \subseteq \Sigma^*$. All the sets of strings above are languages.

We will often think of the belonging of a string in a language as a yes/no problem:



L is “easy to solve” within some computational model if we can write a program doing this. We will define this formally later.

Example 0.15. A program for the language $L = \Sigma^*$ is easy to decipher: “On any input s , output yes.” The language $L = \emptyset$ is also easy to decipher: “On any input s , output no.” The language $L = \{\varepsilon, 00, 01, 10, 11\}$ is also easy to solve if we can implement the following program:

$L =$ “On input $s \in \{0, 1\}^*$:

1. if $s = \varepsilon$, output yes;
2. else if $s = 00$, output yes;
3. else if $s = 01$, output yes;
4. else if $s = 10$, output yes;
5. else if $s = 11$, output yes;
6. else, output no.”

1. Deterministic Finite Automata (12/09)

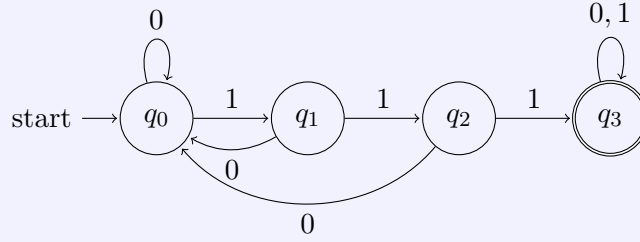
Recap. A language L over an alphabet Σ is a set of strings over Σ , i.e. $L \subseteq \Sigma^*$. Every language L can be thought of as a function $f : \Sigma^* \rightarrow \{0, 1\}$, where

$$f(x) = \begin{cases} 1 & \text{if } x \in L \\ 0 & \text{if } x \notin L. \end{cases}$$

Finite automata, or *finite state machines*, are the simplest computational model, and are good models for computers with an extremely limited amount of memory. Every *deterministic* finite automaton (DFA) has

- finitely many states,
- one *start state*, indicated by an arrow pointing to it from nowhere,
- zero or more *accept states*, indicated by a double circle,
- transition arrows for each alphabet symbol indicating movement between states, and
- for every string $s \in \Sigma^*$ fed into the DFA, the DFA outputs either *accept* or *reject*.

Example 1.1. The following is an example of a DFA. Note that here, $\Sigma = \{0, 1\}$.



When we feed the input string 1110 into the above DFA, the processing proceeds as follows:

1. Start in state q_0 .
2. Read 1. Follow the transition from q_0 to q_1 .
3. Read 1. Follow the transition from q_1 to q_2 .
4. Read 1. Follow the transition from q_2 to q_3 .
5. Read 0. Follow the transition from q_3 and back to q_3 . This is an accept state; accept.

What does the computation of the DFA on the strings 0101101, 111, 1111, and ε look like?

Note that for every (finite) string fed into any DFA, the DFA runs for a finite sequence of steps. We now provide a formal definition of a DFA.

Definition 1.1 (Deterministic finite automaton). A DFA is a 5-tuple $M = (Q, \Sigma, \delta, q_0, F)$, where

1. Q is a finite set called the states of M ,
2. Σ is a finite set called the alphabet,
3. $\delta : Q \times \Sigma \rightarrow Q$ is called the transition function,
4. $q_0 \in Q$ is called the start state, and
5. $F \subseteq Q$ is the set of accepting states.

Example 1.2. The above DFA in Example 1.1 can be written as $M_1 = (Q, \Sigma, \delta, q_0, F)$, where $Q = \{q_0, q_1, q_2, q_3\}$, $\Sigma = \{0, 1\}$, q_0 is the start state, $F = \{q_3\}$, and the transition function δ is defined as

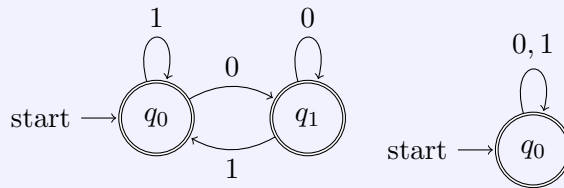
δ	read a 0	read a 1
q_0	q_0	q_1
q_1	q_0	q_2
q_2	q_0	q_3
q_3	q_3	q_3

Definition 1.2 (Accepting computation). For a DFA $M = (Q, \Sigma, \delta, q_0, F)$ and a string $w = w_1 w_2 \cdots w_n \in \Sigma^*$, M accepts w if and only if $\exists r_0, r_1, \dots, r_n \in Q$ such that

1. $r_0 = q_0$ (i.e. we start at the start state)
2. $\delta(r_{i-1}, w_i) = r_i, \forall i \in \{1, 2, \dots, n\}$ (i.e. we follow transitions according to w)
3. $r_n \in F$ (i.e. we end at an accepting state).

Definition 1.3. For a DFA M , we define $L(M) = \{w \in \Sigma^* \mid M \text{ accepts } w\}$. $L(M)$ is called the language recognised by M or simply the language of M .

Example 1.3. The language of the below DFAs is the language of all strings, namely $L(M) = \Sigma^*$.



Example 1.4. The language of the DFA M in Example 1.1 is $L_1(M) = \{w \in \Sigma^* \mid w \text{ has three consecutive 1's}\}$.

Definition 1.4. A language L is regular if there exists a DFA M such that $L(M) = L$.

We have seen that the following languages are regular: Σ^* , $L_1(M)$, etc. Are all languages regular? No. Then how can we characterise a language that is regular?

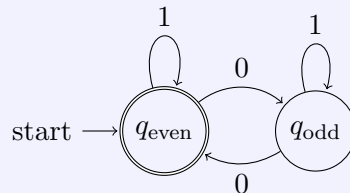
Example 1.5. Consider the language $L = \{w \in \{0,1\}^* \mid w \text{ has an even number of 0's}\}$. Note that $100101101 \in L$, $1111 \in L$, and $\varepsilon \in L$. However, $1011100 \notin L$.

Wrong idea for an algorithm generating L . The following algorithm captures the notion (of an odd number of 0's) that we want:

1. Count how many 0's. Save this to a variable **counter**.
2. If **counter** is even, accept; if **counter** is odd, reject.

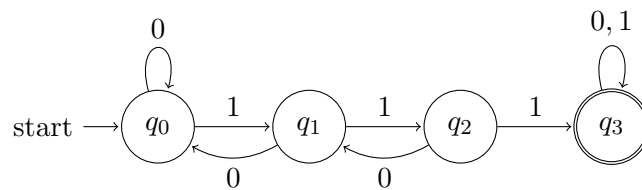
However, intuitively, we cannot implement this algorithm with a DFA, since there is possibly an unbounded number of 0's in the string, but a finite number of states. In general, the number of states that the DFA could possibly have does not scale with the number of zeros that the string could possibly have, as the DFA cannot have infinitely many states.

Right idea for an algorithm generating L . In the following algorithm, each state keeps track of the even or odd number of 0's seen so far. The algorithm cycles through both states:



Challenges

1. Let M_2 represent the following DFA. What is $L(M_2)$?



2. Is the following language regular?

$$L = \{z \in \{0,1\}^* \mid \exists x, y \in \{0,1\}^*, z = xy,$$

x has an even number of 0's, and y has an even number of 1's}.

2. More DFAs, Non-deterministic Finite Automata (14/09)

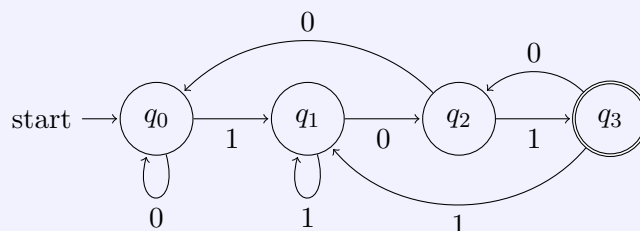
Recap. Reminders from previous class:

- A DFA is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where Q is a finite set of states, Σ is a finite set of alphabet symbols, $\delta : Q \times \Sigma \rightarrow Q$ is transition function ($\delta(q, r) = a$ means $\textcircled{q} \xrightarrow{a} \textcircled{r}$), q_0 a start state, and F a set of accept states.
- We say that M accepts $w = w_1w_2 \cdots w_n$ if there exists an accepting computation for the string, i.e. if there exists a sequence $r_0, r_1, \dots, r_n \in Q$ such that $r_0 = q_0$, $r_i = \delta(r_{i-1}, w_i)$ for all the i in $\{1, \dots, n\}$, and $r_n \in F$.
- The language recognised by a DFA M is the set of all strings that are always accepted by M is $L(M) = \{w \in \Sigma^* \mid M \text{ accepts } w\}$. A language L is regular if there exists some M such that $L = L(M)$, i.e. if there exists some DFA recognising it.

2.1. Demonstrating that a language is regular

To show that a given language is regular, it suffices to construct a DFA that recognises that language.

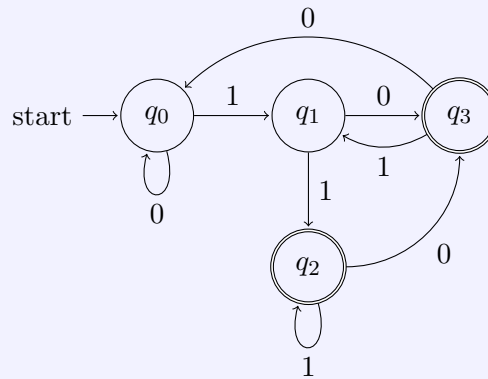
Example 2.1. We claim that the language $L_1 = \{w \in \{0,1\}^* \mid w \text{ ends in } 101\}$ is regular. Here is a DFA that recognises this language:



Let us go through this DFA. By the time we transition from state q_0 to state q_1 , we have at

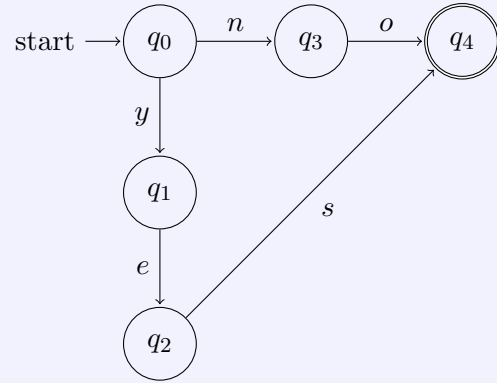
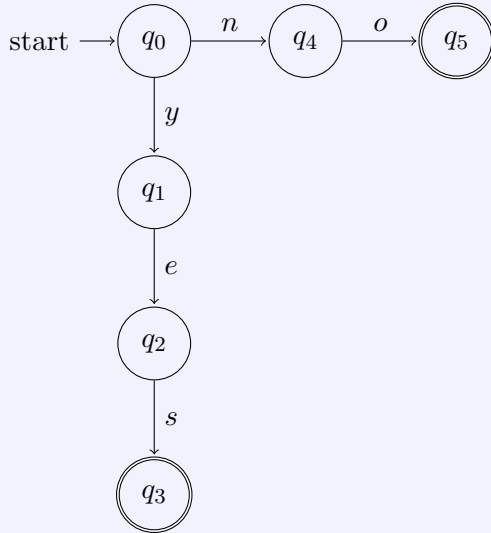
least seen the string 0^n1 , where $n \geq 0$ is an integer. By the time we transition from state q_1 to state q_2 , we have at least seen the string 0^n1^k0 where $n \geq 0$, and $k \geq 1$. Now if the final character of the string is a 1, then we move straight into the accept state q_3 , and we are done. If we see a 0, then we should move back to the start state and keep checking the string (this string's last three characters are 110 instead). If after initially entering q_3 we see a zero, then we should essentially skip it (by moving back to q_2) and continuing on the rest of the string.

Example 2.2. We claim the language $L_2 = \{w \in \{0,1\}^* \mid w\text{'s second to last character is } 1\}$ is regular. Here is a DFA that recognises this language:



Let us interpret this DFA like before. If we are still in q_0 at any snapshot of the DFA's process sequence, then it's either the case that the last two characters are 00, or that the string so far is either 0 or ε . If we are in q_1 , then it is guaranteed that the last two characters are 01, or that the string so far is 1. If we are in q_3 and the string is not exhausted, we just cycle through until either (a) we exhaust the string and accept or reject, (b) we cycle back to q_0 if we see more than one zero consecutively (because then the penultimate character is definitely not 1). Similar thing with q_2 . Thus this DFA recognises L_2 .

Example 2.3. Let the finite alphabet $\Sigma = \{a, b, \dots, z\}$. Then the language $L_3 = \{\text{yes}, \text{no}\}$ is regular. Here is a DFA that recognises L_3 :



Note that the DFA on the right is only a more efficient^a version of the one on the left.

^aA DFA M_2 is more efficient than M_1 if M_2 performs all the operations of M_1 with fewer steps.

Convention. Any missing transitions go to a (missing) q_{bad} state. So in the example above, there is an extra transition from q_0 to q_{bad} for whenever we see $\Sigma \setminus \{y, n\}$, an extra transition from q_1 to q_{bad} for whenever we see $\Sigma \setminus \{e\}$, et cetera.

Theorem 2.1. *Every finite language is regular.*

Proof sketch. If L is finite, then by the well-ordering principle, there exists a longest string in L , say of length m . Now, we can construct a DFA with a path of (finitely many) states for every possible string s such that $|s| \leq m$, plus one bad state q_{bad} . Now mark the transitions and accepting states as appropriate. We can construct a DFA for L , and so L is regular. ■

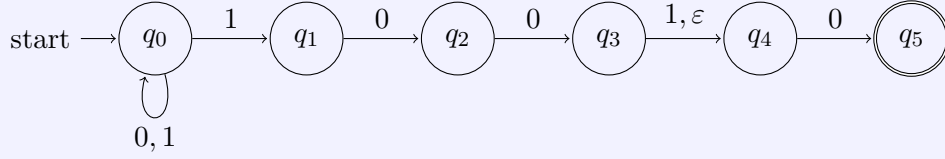
The converse of Theorem 2.1—that every finite language is regular—is *not* true. In fact, L_1 and L_2 above are infinite and regular.

2.2. Non-deterministic finite automata

When a machine is in a given state and reads the next input symbol, we immediately know what the next state will be—that is, transitions are determined. We call this *deterministic* computation. In a *nondeterministic* machine, several choices exist for the next state at any point.

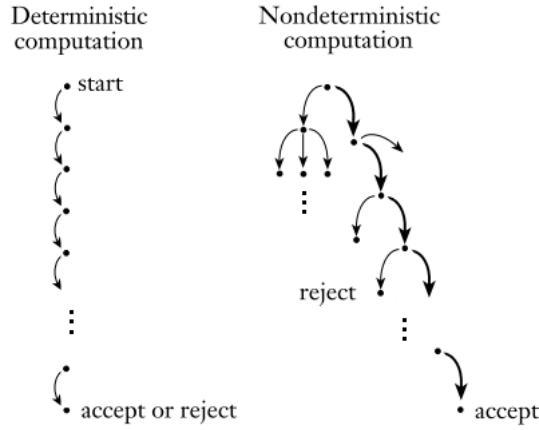
So we can informally describe a non-deterministic finite automaton (NFA) as a finite automaton that acts like a DFA, but can have zero or more transitions for every state and character (including ε -transitions). A string is accepted by an NFA if *there exists some* accepting computation on the string.

Example 2.4. The following is an example of a non-deterministic finite automaton.



Some possible movements of the transition function include $\delta(q_0, 1) = \{q_0, q_1\}$, $\delta(q_2, 0) = \{q_3\}$, $\delta(q_2, 1) = \emptyset$. The string 1010010 is accepted by this NFA, by the following computation: $q_0 q_0 q_0 q_1 q_2 q_3 q_4 q_5$.

There are two views on how an NFA computes on a string, or how nondeterministic computations work in general. The “token view” is that an NFA computes on a string by trying all possible computations at each stage in parallel by “forking” into several children, and accepting if at least one of them succeeds.



The other view, which aligns closer to what we imagine a nondeterministic computation is, is that the NFA proceeds like a computation tree of possibilities. When there is more than one option, the NFA “guesses” the best path that leads to an accept state.

Example 2.5. The language of the NFA N in Example 2.4 is the language defined by $L(N) = \{w \in \{0, 1\}^* \mid w \text{ ends in } 10010 \text{ or } 1000\}$.

Definition 2.2 (Non-deterministic finite automaton). A DFA is a 5-tuple $M = (Q, \Sigma, \delta, q_0, F)$, where

1. Q is a finite set called the states of M ,
2. Σ is a finite set called the alphabet,
3. $\delta : Q \times \Sigma_\epsilon \rightarrow \mathcal{P}(Q)$ is called the transition function,
4. $q_0 \in Q$ is called the start state, and
5. $F \subseteq Q$ is the set of accepting states,

where $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$ and the set of all subsets of Q is $\mathcal{P}(Q) = \{S \mid S \subseteq Q\}$.

Definition 2.3. An NFA N accepts a string w if w can be written as $w = w_1w_2\cdots w_n$, where $w_i \in \Sigma_\varepsilon$ and there is a sequence of states $r_0, r_1, \dots, r_n$ such that

1. $r_0 = q_0$ (i.e. we start at the start state),
2. for all $i \in \{1, 2, \dots, n\}$, $r_i \in \delta(r_{i-1}, w)$ (i.e. we nondeterministically “guess” the “best” state to take based on transitions according to w)
3. $r_n \in F$ (i.e. we end at an accepting state).

Definition 2.4. For an NFA N , $L(N) = \{w \in \Sigma^* \mid N \text{ accepts } w\}$ is the language recognised by N .

Challenge

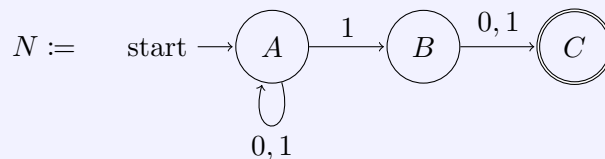
The language

$$L = \{z \in \{0, 1\}^* \mid \exists x, y \in \{0, 1\}^*, z = xy, \\ x \text{ has an even number of 0's, and } y \text{ has an even number of 1's}\}.$$

is regular. Can you build an NFA for this language? If we didn't know that this language was regular but are able to construct an NFA for the language, does that immediately tell us that the language is regular?

3. More on NFAs (19/09)

Example 3.1. Let $L_1 = \{w \in \{0,1\}^* \mid \text{the penultimate character in } w \text{ is } 1\}$. Then the NFA N below recognises N .



Note that we could also describe N using the canonical form described in the definition of NFAs:

- $Q = \{A, B, C\}$, is the set of all states
- $\Sigma = \{0, 1\}$ is the finite alphabet
- δ is given by the table below

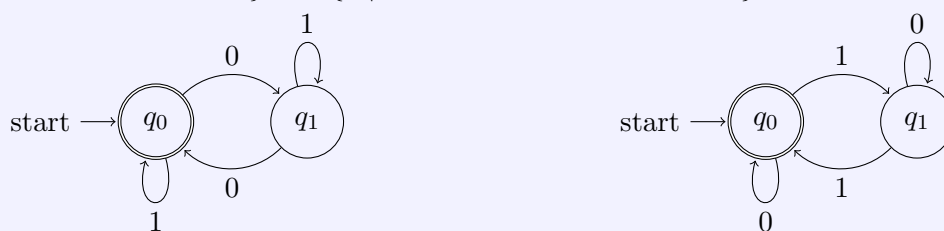
δ	0	1	ε
A	$\{A\}$	$\{A, B\}$	$\emptyset$
B	$\{C\}$	$\{C\}$	$\emptyset$
C	$\emptyset$	$\emptyset$	$\emptyset$

- $q_0 = A$
- $F = \{C\}$.

Example 3.2. Recall the language from last class:

$$L_2 = \{z \in \{0,1\}^* \mid \exists x, y \in \{0,1\}^*, z = xy, \\ x \text{ has an even number of 0's, and } y \text{ has an even number of 1's}\}.$$

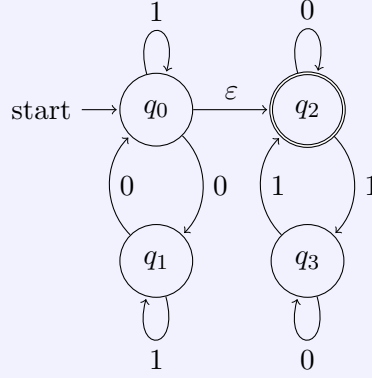
To construct an NFA for L_2 , we first construct DFAs for both of the languages $\{x \mid x \text{ has an even number of 0's}\}$ and $\{y \mid y \text{ has an even number of 1's}\}$:



We can use both of these to construct an NFA for L_2 , shown below. In particular, we create an ε -transition from the first DFA to the second DFA. The function of this ε -transition is to help us develop the concatenation $xy \circ \varepsilon$, in the case that this is necessary.

Suppose the input string $z = 111$. Ignoring parsings that lead to q_{bad} , we can start by parsing this string as $\varepsilon 111$, so that we are (nondeterministically) at states $\{A, C\}$ after reading the first character. When we read the first 1, we are now nondeterministically at $\{A, C, D\}$. When we read the last two 1's, we are nondeterministically at $\{A, C, D\}$ (draw

the computation tree to confirm). Now $C \in F$, so 111 is accepted.



Theorem 3.1. *A language L is regular if and only if some nondeterministic finite automaton recognises it.*

Proof. ($\implies$) If a language is regular, then there is a DFA for L . If there is a DFA for L , then there is an NFA for L , because a DFA is only a special case of an NFA.

($\impliedby$) If there is an NFA for L , we can use this NFA to construct a DFA D for L by means of the *subset construction*, outlined below:

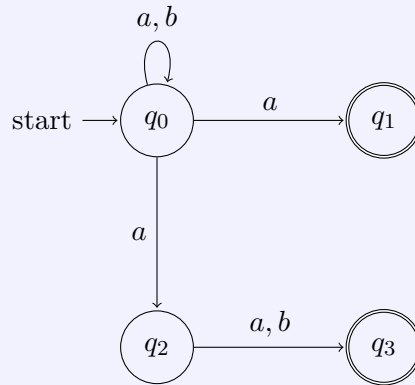
PROCEDURE: The Subset Construction.

Let $N = (Q_N, \Sigma_N, \delta_N, q_{0N}, F_N)$ be an NFA recognising L . Now construct a DFA D defined by $D = (Q_D, \Sigma_D, \delta_D, q_{0D}, F_D)$ as follows:

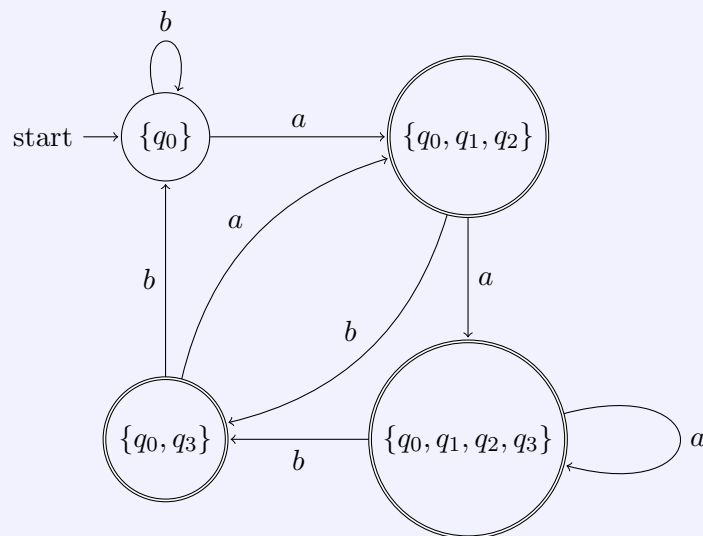
- $Q_D = \mathcal{P}(Q_N)$, i.e. each state in D corresponds to each subset of states in N ,
 - $\Sigma_D = \Sigma_N$, i.e. the alphabet remains invariant,
 - $q_{0D} = E(\{q_{0N}\})$ where $E(S)$ is the set of all states in S , or reachable from a set in S via ε -transitions,
 - $F_D = \{S \subseteq Q_N \mid S \cap F_N \neq \emptyset\}$, i.e. any subset of states in Q_N that includes one accepting state in N is an accepting state in D ,
 - $\delta_D(S, a) = \{q' \in Q_N \mid q' \in E(\delta_N(q, a)) \text{ for some } q \in S\}$, i.e. we start from a state in S and take all possible a -transitions plus any ε -reachable states.
-

We can show that this D recognises the same language as N . Thus the proof is done. ■

Example 3.3. Consider the language $L(N) = \{w \in \{a, b\}^* \mid w \text{ ends in } a \text{ or } ab\}$. The NFA N below recognises this language:



We can apply the subset construction via the above procedure to construct an equivalent DFA for N :



4. Operations on Languages (21/09)

Recap. So far, we have seen the following:

- L is regular $\iff \exists$ DFA recognising $L \iff \exists$ NFA recognising L .
- The subset construction for transferring an NFA to a DFA. This construction incurs an exponential blowup in the number of states (since $\mathcal{P}(Q) = 2^{|Q|}$). As it turns out, this exponential overhead is inherent (see Section D in Appendix).

4.1. Language operations

For languages L, L_1, L_2 over some fixed finite alphabet Σ , we have that the

- *complement* of L is $\overline{L} = \{w \in \Sigma^* \mid w \notin L\} = \Sigma^* \setminus L$,

- *union* of L_1 and L_2 is $L_1 \cup L_2 = \{w \in \Sigma^* \mid w \in L_1 \text{ or } w \in L_2\}$,
- *intersection* of L_1 and L_2 is $L_1 \cap L_2 = \{w \in \Sigma^* \mid w \in L_1 \text{ and } w \in L_2\}$,
- *concatenation* of L_1 and L_2 is $L_1 \circ L_2 = \{w \in \Sigma^* \mid \exists x_1 \in L_1, \exists x_2 \in L_2 \text{ such that } w = x_1 x_2\}$,
- *power* of L is the operation that outputs the set of all strings that can be parsed as a concatenation $x_1 x_2 \cdots x_k$ of k strings in L (with $x_1, x_2, \dots, x_k \in L$). Note that $L^0 = \{\varepsilon\}$, $L^{i+1} = L^i \circ L$ for all $i \geq 0$, and $L^k = \underbrace{L \circ L \circ \dots \circ L}_{k \text{ times}}$,
- *Kleene star* of L is the operation defined by

$$L^* = \bigcup_{k=0}^{\infty} L^k = \{\varepsilon\} \cup L \cup L^2 \cup \dots = \{w \in \Sigma^* \mid \exists k \geq 1, x_1, \dots, x_k \in L, w = x_1 \cdots x_k\} \cup \{\varepsilon\}.$$

Note that L^* contains all the strings that can be parsed as a concatenation of zero or more strings from L .²

Example 4.1. Let $\Sigma = \{0, 1\}$, and the languages $L_1 = \{\varepsilon, 001, 1\}$, $L_2 = \{11, 1\}$, $L_3 = \{111\}$, and $L_4 = \{111, \varepsilon\}$. Then:

- $L_1 \cup L_2 = \{\varepsilon, 001, 1, 11\}$ and $L_1 \cap L_2 = \{1\}$,
- $L_1 \circ L_2 = \{11, 1, 00111, 0011, 111\}$ (note that $11001 \in L_2 \circ L_1$, but $11001 \notin L_1 \circ L_2$ ^a)
- $L_1^2 = L_1 \circ L_1 = \{111111\}$, and $L_2^2 = L_2 \circ L_2 = \{111, 111111, \varepsilon\}$. Note that $L_1^2 \neq L_2^2$.

^aorder matters for concatenation!

Theorem 4.1. *The class of regular languages is closed under the operations above. That is:*

- if L is regular, then $\bar{L}$ is regular,
- if L_1, L_2 are regular, then $L_1 \cap L_2, L_1 \cup L_2$ and $L_1 \circ L_2$ are all regular,
- if L is regular, then L^k is regular $\forall k \geq 0$,
- if L is regular, then L^* is regular.

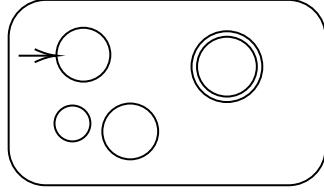
Proof. This will be a long proof. We separate the different parts of the proof into separate cases:

For complement. Given that L is regular, L has a DFA D recognising it. We claim that the DFA formed by switching the accepting and nonaccepting states of D recognises $\bar{L}$, and so L is regular. Let $D = (Q, \Sigma, \delta, q_0, F)$ be a DFA recognising L . Now construct the DFA $D' = (Q, \Sigma, \delta, q_0, Q \setminus F)$. We need to prove that $x \in \bar{L} \iff D'$ accepts x . This is true, because $x \in \bar{L} \iff x \notin L \iff$ the computation of D on x ends in a state in $Q \setminus F \iff$ the computation of D' on x ends in a state in $Q \setminus F \iff D'$ accepts x .³

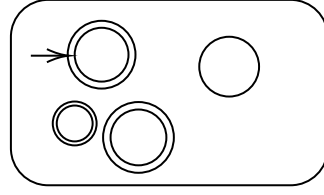
²The empty string ε is itself called the “concatenation of zero strings from L ”.

³Note that the same proof does not work if D was an NFA. Can you see why?

$D :$



$D' :$

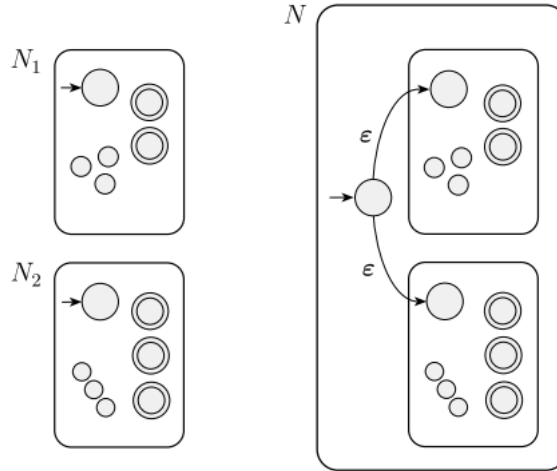


For union. If L_1 and L_2 are regular, then we can construct NFAs N_1 and N_2 recognising both these languages. The goal here is to construct an NFA recognising $L_1 \cup L_2$ using an additional state and ε -transitions from this state to both the start states of N_1 and N_2 .

Let $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ be an NFA recognising L_1 , and $N_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ be an NFA recognising L_2 . We now construct an NFA N recognising $L_1 \cup L_2$. This NFA is constructed from both N_1 and N_2 as $N = (\{q_0\} \cup Q_1 \cup Q_2, \Sigma, \delta_N, q_0, F_1 \cup F_2)$, where δ_N is defined as, for any $q \in \{q_0\} \cup Q_1 \cup Q_2$ and any $a \in \Sigma_\varepsilon$,

$$\delta(q, a) = \begin{cases} \delta_1(q, a) & q \in Q_1 \\ \delta_2(q, a) & q \in Q_2 \\ \{q_1, q_2\} & q = q_0 \text{ and } a = \varepsilon \\ \emptyset & q = q_0 \text{ and } a \neq \varepsilon \end{cases}$$

This NFA recognises $L_1 \cup L_2$ because $w \in L_1 \cup L_2 \iff w \in L_1 \text{ or } w \in L_2 \iff \exists$ an accepting computation of N_1 on w or $\exists$ an accepting computation of N_2 on $w \iff \exists$ accepting computation of N on w .



For intersection. We can present two proofs here:

Proof 1. By De Morgan's law, $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$ and the fact that we already proved the closure of regular languages under complement and union,

$$\left. \begin{array}{l} L_1 \text{ regular} \implies \overline{L_1} \text{ regular} \\ L_2 \text{ regular} \implies \overline{L_2} \text{ regular} \end{array} \right\} \implies \overline{L_1 \cup L_2} \text{ regular}$$

By closure under complement, we then have that $\overline{\overline{L_1} \cup \overline{L_2}} = L_1 \cap L_2$ is regular.

Proof 2. Otherwise we can directly construct a DFA/NFA for the language $L_1 \cap L_2$. The idea here is to run both machines for L_1 and L_2 in parallel, and accept if *both* machines end in an accepting state.

Let $D_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ be a DFA for L_1 , and $D_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ be a DFA for L_2 . We construct a DFA for $L_1 \cap L_2$ as follows: $D = (Q, \Sigma, \delta, q_0, F)$, where $Q = Q_1 \times Q_2 = \{(r_1, r_2) \mid r_1 \in Q_1, r_2 \in Q_2\}$, $q_0 = (q_1, q_2)$, $F = F_1 \times F_2 = \{(r_1, r_2) \mid r_1 \in F_1, r_2 \in F_2\}$, and $\delta((r_1, r_2), a) = (\delta_1(r_1, a), \delta_2(r_2, a))$ for all $(r_1, r_2) \in Q_1 \times Q_2$ and for all $a \in \Sigma$.

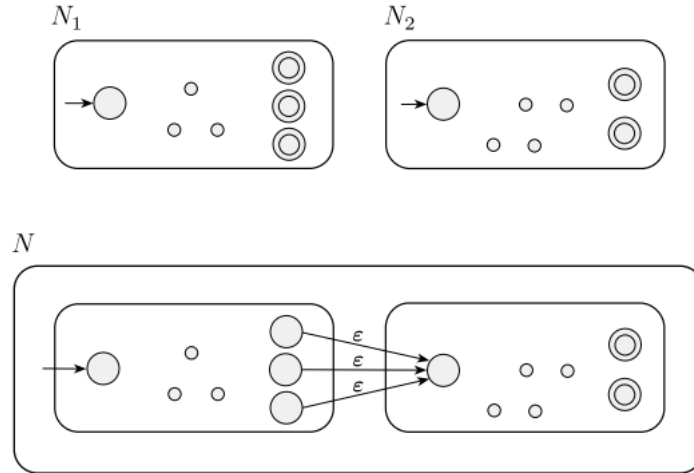
This DFA recognises $L_1 \cap L_2$ because $w \in L_1 \cap L_2 \iff D$ is in state (r_1, r_2) after reading $w \iff D$ is in state r_1 after reading w and D_2 is in state r_2 after reading w . Thus from our definition of F , w is accepted by $D \iff w$ is accepted by both D_1 and D_2 .⁴

For concatenation. If L_1 and L_2 are regular, then we can construct NFAs N_1 and N_2 recognising both these languages respectively. The goal here is to build an NFA for $L_1 \circ L_2$, as follows: Add ε -transitions from the accepting states of N_1 to the start state of N_2 . Then make the start state of N to be N_1 's start state, and the new accept state of N to be N_2 's accept states.

Let $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ and $N_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$. We construct an NFA recognising $L_1 \circ L_2$ as follows: $N = (Q_1 \cup Q_2, \Sigma, \delta, q_1, F_2)$, where δ is defined as follows

$$\delta(q, a) = \begin{cases} \delta_1(q, a) & q \in Q_1 \text{ and } q \in F_1 \\ \delta_1(q, a) & q \in F_2 \text{ and } a \neq \varepsilon \\ \delta_1(q, a) \cup \{q_2\} & q \in F_1 \text{ and } a = \varepsilon \\ \delta_2(q, a) & q \in Q_2 \end{cases}$$

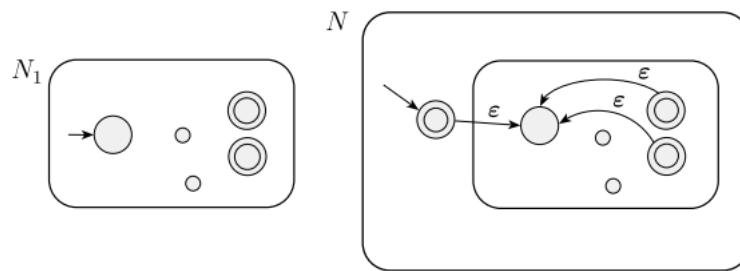
As an exercise, prove that N below recognises $L_1 \circ L_2$.



For Kleene star. Here, we provide only a proof idea; the actual proof is left as an exercise. Suppose that there is a regular language L and an NFA N_1 recognising this language. Then we can construct an NFA N recognising L^* by adding additional ε -transitions from the accept states of N_1 back to the start state to ensure that N has the option of jumping back to the start state to try to read a new piece of string accepted by N_1 . To account for the fact that this construction may allow extra

⁴These proofs implies that if $L_1, L_2, \dots, L_k$ are all regular languages, then $L_1 \cap L_2 \cap \dots \cap L_k$ is regular for any k . We can prove this by induction. However, we cannot use induction to prove the case for Kleene star, i.e. to go to ∞ , as induction only works to go from 2 to any finite number.

undesired strings and does not yet accept ε , we add a new start state which is also accepting and an extra ε -transition from this new start state to the previous start state of N_1 . Altogether, we have something like:



As an exercise, write this construction formally and prove that it works. ■

Challenges

1. Given an NFA for L , how would we construct an NFA for $\overline{L}$?
2. When we constructed an NFA N for the union case from the smaller NFAs N_1 and N_2 , we first added an extra state and then added ε -transitions from this state to both the start states of N_1 and N_2 . Suppose instead we selected N_1 and then added an ε -transition from the start state of N_1 to the start state of N_2 . Would this work?⁵ Why?
3. Look at the DFA constructed in the second proof that regular languages are closed under intersection. We can use the same technique to complete the construction proof for the union case by an identical construction, but writing $F = (F_1 \times Q_2) \cup (Q_1 \times F_2) = \{(r_1, r_2) \mid r_1 \in F_1 \text{ or } r_2 \in F_2\}$. Can you see why this is the case?
4. Look again at the DFA constructed in the second proof that regular languages are closed under intersection. Adapt the proof to the case where you are given NFAs instead of DFAs, i.e. build a new NFA directly.
5. In the proof for the Kleene star case, we mentioned that without the new accepting start state and the ε -transition to the previous start state, we may construct a machine that accepts strings not in L^* . Can you find some such strings?

5. Regular expressions (27/09 and 29/09)

Definition 5.1. The operations $\cup$, $\circ$, and $*$ are called the regular operations.

Definition 5.2 (Regular expression). We say that R is a regular expression over some alphabet Σ if R is

1. a for some $a \in \Sigma$,
2. ε ,

⁵No.

3. $\emptyset$

4. $(R_1 \cup R_2)$, where R_1 and R_2 are regular expressions,

5. $(R_1 \circ R_2)$, where R_1 and R_2 are regular expressions, or

6. (R_1^*) , where R_1 is a regular expression.⁶

Thus, a regular expression is a string using symbols from $\Sigma \cup \{ (,), \cup, \circ, *, \varepsilon, \emptyset \}$.

Regular expressions generate regular languages:

Definition 5.3. For a regular expression R , there is an associated language $L(R)$ defined as follows:

1. $L(a) = \{a\}$,
2. $L(\varepsilon) = \{\varepsilon\}$,
3. $L(\emptyset) = \{\} = \emptyset$,
4. $L(R_1 \cup R_2) = L(R_1) \cup L(R_2)$,
5. $L(R_1 \circ R_2) = L(R_1) \circ L(R_2)$,
6. $L(R^*) = (L(R))^*$.

Note. The order of precedence of regular operations is $*$, $\circ$, $\cup$.

Example 5.1. We have $L((a \cup b)^*) = \{a, b\}^*$, i.e. all strings over $\{a, b\}$.

Example 5.2. We have $L(aa^*) = \{a^k \mid k \geq 1\}$. Note the difference between this and the language $L((aa)^*) = \{a^k \mid k \text{ is even}\}$.

Example 5.3. $L(\emptyset^*) = \{\varepsilon\}$.

Example 5.4. $L((a \cup b)^* a (a \cup b) (a \cup b)) = \{w \in \{a, b\}^* \mid \text{third to last character in } w \text{ is } a\}$.

Definition 5.4 (Equivalence of regular expressions). We say $R_1 \equiv R_2$ if $L(R_1) = L(R_2)$.

Note. For all regular expressions R ,

- $\emptyset \circ R \equiv R \circ \emptyset \equiv \emptyset$, that is, $\emptyset$ is an *annihilator* for concatenation,
- $\emptyset \cup R \equiv R \cup \emptyset \equiv R$, that is, $\emptyset$ is the *identity* for union,
- $\varepsilon \circ R \equiv R \circ \varepsilon \equiv R$, that is, ε is the *identity* for concatenation.

Example 5.5. $L(\emptyset \circ (abb)^*) = \{\} \circ \{\varepsilon, abb, abbabb, \dots\} = \emptyset$.

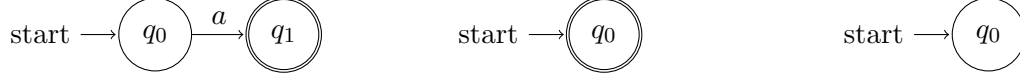
Theorem 5.5. A language L is generated by a regular expression if and only if L is regular.

⁶The first three items here can be thought of as “base” cases, and the last three can be thought of as “recursive”.

Proof. We prove this theorem in two lemmas covering both directions of the implication. ■

Lemma 5.6. *If L is generated by a regular expression R , then L is regular, i.e. L is recognised by some DFA/NFA.*

Proof. We first cover the three base cases $R = a$, $R = \varepsilon$, $R = \emptyset$.



Each of these regular expressions are recognised by these NFAs, and so the lemma is proved for them. For the cases $R = R_1 \cup R_2$, $R = R_1 \circ R_2$, and $R = R_1^*$, we can simply adopt the NFA constructions given when we proved closure under these operations. The proof is done. ■

Lemma 5.7. *If a language L is regular, then L is generated by some regular expression R .*

Plan for proof. We will define a generalised NFA (a GNFA), and show the following. An NFA reduces to a GNFA, which in turn reduces to a GNFA with one fewer states. Then we continue this reduction until we have a GNFA with only one start state, one accept state, and no other states. The transition from the start state to the accept state of this GNFA is then the regular expression R .

Definition 5.8 (Generalised nondeterministic finite automaton, GNFA). *A GNFA is a 5-tuple $(Q, \Sigma, \delta, q_{\text{start}}, q_{\text{accept}})$ where*

- Q is the finite set of states,
- Σ is the input alphabet, $\delta : (Q \setminus \{q_{\text{accept}}\}) \times (Q \setminus \{q_{\text{start}}\}) \rightarrow \mathcal{R}$ is the transition function,
- q_{start} is the start state, and
- q_{accept} is the accept state.

A string w is accepted by a GNFA if there exists a path P from the start state to the accept state such that concatenating a regular expression along the P gives a regular expression R such that $w \in L(R)$.

Transformation from NFA to regular expression via GNFA.

Step 1. Given an NFA, transform it to an equivalent GNFA via the following procedure:

- add a new state q_{start} , with ε -transition to the old start state,
- add a new accept state q_{accept} , which will be the only accepting state, and add ε -transitions from all old accept states to q_{accept} ,
- if a transition was labelled with multiple symbols (e.g. a, b, d), transform to a single transition labelled as the union of these symbols (i.e. $a \cup b \cup d$)

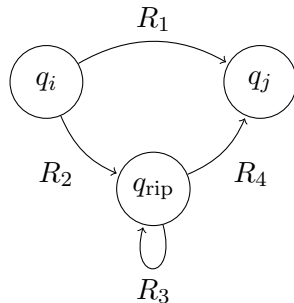


- all other missing transitions are labelled with $\emptyset$.

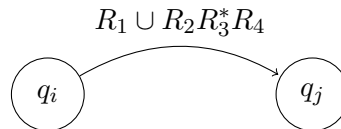
Step 2. Given a GNFA with more than 1 intermediate states (i.e. states other than q_{start} and q_{accept}), remove one of the intermediate states, and update all transitions as follows.

- say the state to be removed is q_{rip} ; for all pairs q_i, q_j with $q_i \neq q_{\text{rip}}$ and $q_j \neq q_{\text{rip}}$, update the transition from q_i to q_j as follows:

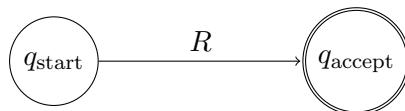
from:



to:



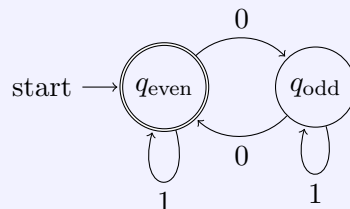
- keep ripping states until the GNFA has only two states, namely q_{start} and q_{accept} .



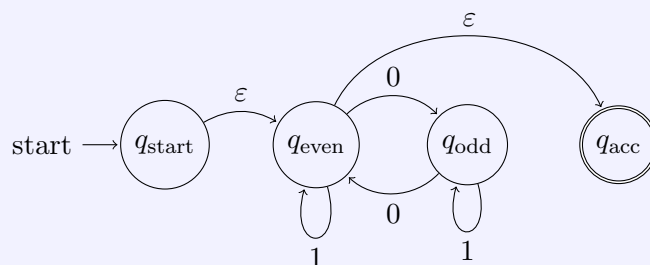
Proof of Lemma 5.7. We claim that $L(R)$ is the exact language recognised by the original NFA. The proof is left as an exercise. ■

Proof of Theorem 5.5. The proof of this theorem then follows from the above two lemmas. ■

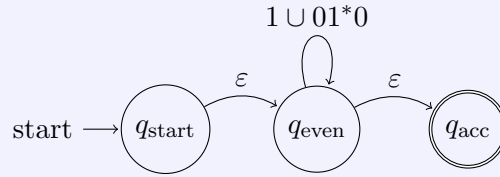
Example 5.6. Consider the language $\{w \in \{0,1\}^* \mid w \text{ has an even number of 0's}\}$. The NFA for this language is



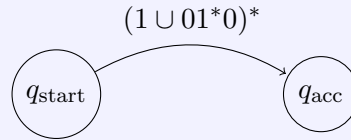
Now we convert this to a GNFA (excluding $\emptyset$ -transitions):



Now we rip/remove q_{odd} :



Now consider the path $q_{\text{start}} \rightarrow q_{\text{even}}$ as $\varepsilon \cup \emptyset 1^*0 \equiv \varepsilon$, the path $q_{\text{even}} \rightarrow q_{\text{even}}$ as $1 \cup 01^*0$, and the path $q_{\text{even}} \rightarrow q_{\text{acc}}$ as $\varepsilon \cup \emptyset 1^*0 \equiv \varepsilon$. We can rip/remove q_{even} :



Now we consider the path from $q_{\text{start}} \rightarrow q_{\text{acc}}$, as $\emptyset \cup \varepsilon \circ (1 \cup 01^*0)^* \circ \varepsilon \equiv (1 \cup 01^*0)^*$. This final regular expression, $(1 \cup 01^*0)^*$, generates the original language.

Challenges

1. Show that the language $L_{2023} = \{0^n 1^n \mid 0 \leq n \leq 2023\}$ is regular.
2. Write a regular expression generating the set of binary strings which represent in binary a number that is an integer multiple of 3 (leading zeros are allowed). For example, 00, 11, and 110 are in the language (they correspond to 0, 3, and 6 respectively), but 001 and 101 are not (they correspond to 1 and 5 respectively).
3. Let L be the language consisting of all strings of a 's and b 's with an equal number of occurrences of ab and ba as substrings. (The string $aabbbbaa$ is in L , because there is exactly one occurrence of each of the substrings ab and ba .) Is L a regular language? Prove your answer.

6. Non-regular languages (03/10)

Recap. So far we have seen that, for some DFA D , NFA N , and regular expression R :

$$L \text{ is regular} \iff L \text{ is recognised by } D \iff L \text{ is recognised by } N \iff L \text{ is generated by } R.$$

Now to prove L is regular, we only need exhibit some DFA/NFA/regular expression for L , or use closure properties (e.g. show that $L = L_1 \cap L_2$ for some regular L_1, L_2).

Consider the language $L_1 = \{a^n b^n \mid n \geq 0\}$. If we wanted to find a DFA that recognises L_1 , this DFA must keep track of identical sequences of a and b symbols for it to accept $\varepsilon, ab, aabb, aaabbb, \dots$. Suppose such a DFA existed. How many states would we then assign to keep track of the a 's? How many to keep track of the b 's? What if the input had more characters?

We cannot create a DFA for L_1 , so L_1 is not a regular language. How can we prove that some language is not regular?

Plan/overview. We first prove that all regular languages satisfy some “property”, the *pumping lemma*. To show that a given language L is not regular, it is then sufficient to show that L does not satisfy this property.

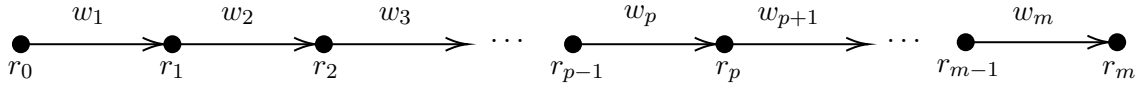
Theorem 6.1 (Pumping lemma for regular languages). *For any regular language L , there exists a number p (called the pumping length or the pumping number) such that for all $w \in L$ and $|w| \geq p$, there exists a way to parse w as $w = xyz$ such that*

- $|xy| \leq p$, i.e. the substring y is within the first p characters,
- $|y| > 0$, i.e. the substring y consists of at least one character,
- $\forall i \geq 0, xy^iz \in L$, i.e. y is “pumpable”.^a

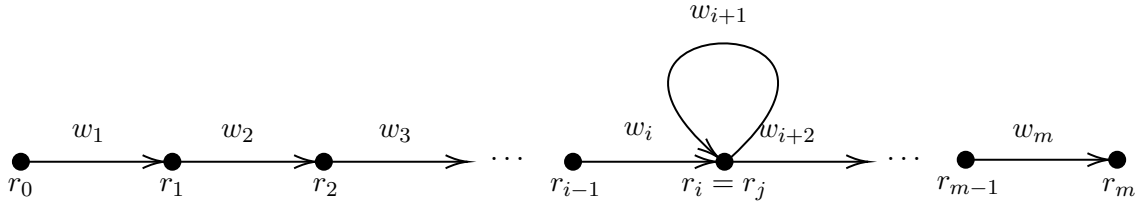
^aThis in turn means that the (distinct, since $|y| > 0$) strings $xz \in L$, $xyz \in L$, $xyyz \in L$, ...

Proof. Let L be a regular language. Let D be a DFA recognising L , and take p as the number of states in D .

Now consider any $w \in L$ with $|w| = m \geq p$, and write $w = w_1w_2 \cdots w_m$ where all the $w_i \in \Sigma$. Let $r_0, r_1, \dots, r_m$ be the sequence of states that D enters when processing w . Then the computation of D on w proceeds as follows:



Thus, $r_{i+1} = \delta(r_i, s_i)$ for $1 \leq i \leq n$. This sequence has length $n + 1$, so w passes through $p + 1$ states. There are only p distinct states, so by the pigeonhole principle, there is a repetition within these $p + 1$ states $r_0, \dots, r_p$, i.e. $r_i = r_j$ for $0 \leq i < j \leq p$:



Now, choose $x = w_1 \cdots w_i$, $y = w_{i+1} \cdots w_j$, and $z = w_{j+1} \cdots w_m$.

As x takes D from r_0 to r_i , y takes D from r_i to r_{j+1} , and z takes D from r_{j+1} to r_m , which is an accept state, D must accept xy^iz for $i \geq 0$. We know that $i < j$, so $|y| \geq 1$; and $j \leq p$, so $|xy| \leq p$. Additionally, $\forall i, xy^iz \in L$, because the computation of D on xy^iz ends in the accepting state r_m . Thus all the conditions of the pumping lemma are satisfied.⁷ ■

Using the pumping lemma. To prove that a language L is not regular, we just show that L does not satisfy the pumping lemma. This means that we show that L satisfies the *negation* of the pumping lemma, i.e. $\forall p$, there exists some $w \in L$ with $|w| \geq p$ such that for all the ways to parse w as $w = xyz$ where $|xy| \leq p$ and $|y| > 0$, there always exists some $i \geq 0$ such that $xy^iz \notin L$.

⁷Note that if all strings in L are shorter than p characters, then the conditions of the pumping lemma hold vacuously.

Example 6.1. Let us now prove that the language $L_1 = \{a^n b^n \mid n \geq 0\}$ is not regular using the pumping lemma.

Assume towards contradiction that L_1 is regular. Let p be the pumping number, and choose $w = a^p b^p$. Now $w \in L_1$, and $|w| = 2p \geq p$. For every possible way to parse $w = a^p b^p$ where $|y| > 0$ and $|xy| \leq p$, it must be that $y = a^k$ for some $k > 0$:

$$w = \underbrace{aa \overbrace{aa \cdots aa}^{y=a^k} aa}_{p \text{ times}} \underbrace{bbbb \cdots bbbb}_{p \text{ times}}.$$

Now choose^a $i = 0$. Then $w = xy^i z = xy^0 z = xz = a^{p-k} b^p \notin L$. This is a contradiction, hence L_1 is not regular.

^aNote that any other choice $i \neq 1$ would work in this example. Can you see why?

Example 6.2. Let us prove that the language of palindromes (i.e. the language described by $L_2 = \{w \in \{a, b\}^* \mid w = w^R\}$ where w^R is the reverse of w) is not regular. We do this in two attempts.

First attempt. Choose $w = a^p$. Then $w \in L_2$, and $|w| = p \geq p$. Now we pump: we can parse w as $x = \varepsilon$, $y = a$, and $z = a^{p-1}$. Indeed $xy^i z \in L_2$ for all i . But this does not lead to a contradiction, so this attempt fails.

Second attempt. Choose^a $w = a^p b^p a^p$. Now $w \in L_2$, and $|w| = 3p \geq p$. For every possible way to parse $w = a^p b^p a^p = xyz$ where $|y| > 0$ and $|xy| \leq p$, we must have $x = a^m$, $y = a^k$, and $z = a^{p-k-m}$ for some $k > 0$:

$$w = \underbrace{a \overbrace{a \cdots a}^{x=a^m} \overbrace{a \cdots a}^{y=a^k} a}_{p \text{ times}} \underbrace{b^p a^p}_{z=a^{p-k-m}}.$$

Now choose $i = 2$. Then $w = xy^i z = xy^2 z = a^{p+1+k} b^p a^p \notin L_2$. Contradiction— L_2 is not regular.

^aOne other choice for the pumping string is $w = a^p b a^p$; try to complete the rest of the exercise from here.

Challenges

Recall the earlier discussion about the closure properties of regular languages. Prove or disprove the following:

1. L is regular $\iff \bar{L}$ is regular.
2. L_1 and L_2 are regular $\iff L_1 \cup L_2$ is regular.
3. L_1 and L_2 are regular $\iff L_1 \cap L_2$ is regular.
4. L_1 and L_2 are regular $\iff L_1 \circ L_2$ is regular.
5. L is regular $\iff L^*$ is regular.

7. More on non-regular languages (05/10)

Recap. Here's what to keep in mind when you want to prove whether a given language is regular or not. Remember that the pumping lemma is a condition all regular languages satisfy. If a given language L is regular, then it satisfies this condition. If L does not satisfy this condition, then it must not⁸ be regular.

7.1. More examples

Example 7.1. We want to prove claim that the language of unary strings of prime length, defined by $L_{\text{prime}} = \{1^n \mid n \text{ is prime}\} = \{11, 111, 11111, \dots\}$ is not regular.

Assume towards contradiction that L_{prime} is regular, and let p be the pumping number. Choose $w = 1^q$ where q is some prime number such that $q \geq p$ (such a q must always exist since there are infinitely many primes). For $w \in L_{\text{prime}}$ then, $|w| = q \geq p$. For any parsing $w = 1^q = xyz$ where $|xy| \leq p$ and $|y| > 0$, it must be that $y = 1^k$ for $0 < k \leq p$. Now we want to choose $i \geq 0$ such that $xy^iz \notin L_{\text{prime}}$. Choose $i = q + 1$, so that

$$xy^iz = 1^{q+qk} = 1^{q(1+k)} \notin L_{\text{prime}}.$$

Since $k > 0$ and $1 + k > 1$, the number $q(1 + k)$ is not prime. Contradiction.

Example 7.2. Let $\#_y(x)$ represent the number of (characters) y 's in a given string x . Consider the language $L = \{w \in \{a, b\}^* \mid \#_a(w) = \#_b(w)\}$. We claim that this language is non-regular.

To prove this claim, assume for the sake of contradiction that L was regular, and consider the language L' defined below:

$$L' = L \cap L(a^*b^*).$$

Now L is regular (from our assumption). Also $L(a^*b^*) = \{a^ib^j \mid i \geq 0, j \geq 0\}$ is regular, since it is generated by a regular expression (namely a^*b^*). Thus since L and $L(a^*b^*)$ are both regular, and regular languages are closed under intersection, L' must be regular. But the intersection $L' = \{a^n b^n \mid n \geq 0\}$, which we already proved (6.1) to be non-regular. This is a contradiction— L must be non-regular.

There is a third technique called the Myhill-Nerode theorem.⁹ used to tell whether a language is regular.

7.2. Other facts about the regular languages

We have now seen three equivalent models: DFAs, NFAs, and regular expressions. The following statements about their equivalence are true:

⁸Actually, in this case, not all non-regular languages satisfy the pumping lemma (see Challenges). But this characterisation is helpful just as it is, for this course.

⁹This technique is covered in the Appendix.

- **The conversion of a DFA to an NFA** is simple, and takes linear $\mathcal{O}(n)$ time on an n -state DFA. All we need do is modify the table representing δ for the DFA by putting set-brackets around states, and if the output is an ε -NFA, adding an extra column for ε . Since we treat the number of input symbols as a constant, copying and processing the table takes $\mathcal{O}(n)$ time.
- **The conversion of an NFA to a DFA** forces an exponential blowup in the number of states. Showing this is more involved—we do so in the Appendix.
- **The conversion of a regular expression to an ε -NFA** takes linear time. First we need to parse the expression efficiently in $\mathcal{O}(n)$, yielding an expression tree with one node for each symbol of the regular expression. Once we have an expression tree for the regular expression, we work up the tree, building the ε -NFA for each node. With a bit of thought we can show that the number of states and transitions of the resulting ε -NFA are both $\mathcal{O}(n)$. Moreover, creating each node of the parse tree takes constant work time. (As an aside, we can eliminate ε -transitions from an n -state ε -NFA to make an ordinary NFA in $\mathcal{O}(n^3)$ time, keeping the number of states constant.)
- **The conversion of an DFA/NFA to a regular expression** yields a regular expression with length exponentially long in the number of states, since if each regular expression is built from k expressions of the previous round in the subset construction, we can k -tuple the size of the regular expressions constructed at each of the n rounds (here n is the number of states in a DFA).

Challenges

1. We mentioned in passing that there are languages that are not regular but that satisfy the pumping lemma. One such language is

$$L = \{ab^n c^n \mid n \geq 0\} \cup \{a^k w \mid k \neq 2 \text{ and } w \in \Sigma^*\}.$$

Prove that L satisfies the pumping lemma for a given pumping number. Now prove that L is non-regular (you can use the Myhill-Nerode theorem (see Appendix) or a number of techniques to do this).

Can you provide instances of other non-regular languages that satisfy the pumping lemma?

2. We already showed that L_{prime} is not regular. Is the language L_{prime}^* also not regular?

8. Context-free languages (03/10 and 05/10)

Definition 8.1 (Context-free grammar, CFG). A *context-free grammar* is a 4-tuple (V, Σ, R, S) , where

- V is a finite set of variables,
- Σ is a finite alphabet,
- R is a finite set of productions or rules that represent the recursive definition of a language, where each rule is of the form $A \rightarrow w$ where $A \in V$ and $w \in (V \cup \Sigma)^*$,
- $S \in V$ is the start variable.

Example 8.1. Consider the context-free grammar G_1 defined by the following rules: $V = \{S\}$, $\Sigma = \{0, 1\}$, and R is defined by

$$R: \quad \begin{aligned} S &\rightarrow \varepsilon \\ S &\rightarrow 0S1. \end{aligned}$$

Note that R can be written more compactly as $S \rightarrow \varepsilon \mid 0S1$.

Definition 8.2. For a CFG $G = (V, \Sigma, R, S)$, if $u, v \in (V \cup \Sigma)^*$ and there is a rule $A \rightarrow w$ in R , then we can write $uAv \Rightarrow uwv$ (read as uAv “yields” uwv via the w -substitution).

Definition 8.3. For a CFG $G = (V, \Sigma, R, S)$, if $u, v \in (V \cup \Sigma)^*$, we say that u derives v , that is, $u \xRightarrow{*} v$, if $u \Rightarrow u_1 \Rightarrow \dots \Rightarrow u_n \Rightarrow w$ for some $u_1, \dots, u_n$.

Definition 8.4 (Context-free language). For a CFG $G = (V, \Sigma, R, S)$, the language generated by G is given by $L(G) = \{v \in \Sigma^* \mid S \xRightarrow{*} v\}$. A language L is a context-free language (CFL) if there exists a CFG G that generates L , i.e. $L(G) = L$.

Example 8.2. In $G_1 = (\{S\}, \{0, 1\}, S \rightarrow \varepsilon \mid 0S1, S)$ above, $S \xRightarrow{*} \varepsilon$, since $S \Rightarrow \varepsilon$. Additionally,

$$S \Rightarrow 0S1 \Rightarrow 00S11 \Rightarrow 000S111 \Rightarrow 0000S1111 \Rightarrow 0000\varepsilon1111 \Rightarrow 00001111.$$

More generally, $S \xRightarrow{*} 0^n 1^n$ for all $n \geq 0$, and these are in fact the only strings in $\{0, 1\}^*$ that can be derived from S . Thus, $L(G_1) = \{0^n 1^n \mid n \geq 0\}$.

Example 8.3. Consider the CFG $G_2 = (\{S\}, \{a, b\}, R, S)$, where R is defined by

$$R: \quad \begin{aligned} S &\rightarrow \varepsilon \\ S &\rightarrow aSa \mid bSb. \end{aligned}$$

Some example strings generated by this CFG include ε (since $S \Rightarrow \varepsilon$), aa (since $S \Rightarrow aSa \Rightarrow aa$), bb (since $S \Rightarrow bSb \Rightarrow bb$), and $aabb$ (since $S \Rightarrow aSa \Rightarrow aaSaa \Rightarrow aabSbaa \Rightarrow aabb$). Now, $L(G_2) = \{w \in \{a, b\}^* \mid w = w^R \text{ and } |w| \text{ is even}\}$.

Suppose we had this exact same CFG as G_3 with R redefined by

$$R: \quad \begin{aligned} S &\rightarrow \varepsilon \mid a \mid b \\ S &\rightarrow aSa \mid bSb. \end{aligned}$$

This context-free grammar G_3 generates the language of all palindromes in the alphabet $\{a, b\}$, i.e. $L(G_3) = \{w \in \{a, b\}^* \mid w = w^R\}$

Context-free grammars are a more powerful method of describing languages than finite automata and regular languages—in fact, both languages above cannot be generated by a finite automaton (and hence are not regular), but they can both be generated by a context-free grammar. CFGs can

describe certain features that have a recursive structure, which makes them useful in a variety of applications.

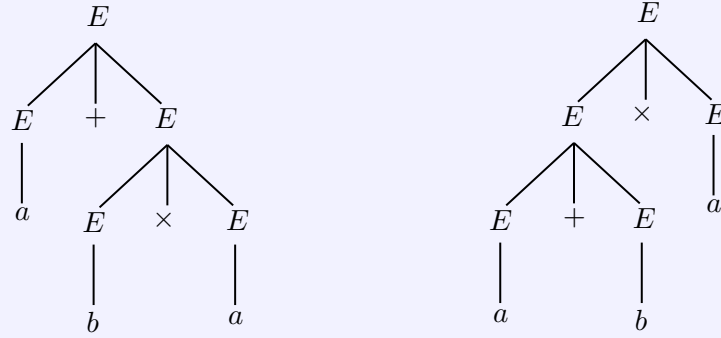
Definition 8.5 (Derivations, sentential form). *The sequence of substitutions to obtain a string is called a derivation. Derivations are represented pictorially using derivation trees, which are also called parse trees.*

If $G = (V, \Sigma, R, S)$ is a CFG, then any string $w \in (V \cup \Sigma)^*$ such that $S \xRightarrow{*} w$ is a sentential form.

Example 8.4. Consider the CFG $G_{\text{expr}} = (\{E\}, \{a, b, +, -, \times, (,)\}, R, E)$, where R is defined by the rule

$$R: \quad E \rightarrow E + E \mid E \times E \mid (E) \mid a \mid b.$$

Note that $a + b \times a \in L(G_{\text{expr}})$, because $E \Rightarrow E + E \Rightarrow E + E \times E \Rightarrow a + E \times E \Rightarrow a + b \times E \Rightarrow a + b \times a$. A different derivation with the same result is $E \Rightarrow E \times E \Rightarrow E \times a \Rightarrow E + E \times a \Rightarrow a + E \times a \Rightarrow a + b \times a$. The corresponding derivation trees for each derivation, respectively, are



Definition 8.6 (Leftmost derivation). *A leftmost derivation is a derivation where at each step we replace the leftmost variable.*

Claim. There is a one-to-one correspondence between a derivation tree and a leftmost derivation, i.e. every tree corresponds to a single leftmost derivation, and every leftmost derivation corresponds to a single tree.

Check that in Example 8.4, the derivations we saw were not leftmost. The corresponding leftmost derivations are $E \Rightarrow E + E \Rightarrow a + E \Rightarrow a + E \times E \Rightarrow a + b \times E \Rightarrow a + b \times a$, and $E \Rightarrow E \times E \Rightarrow E + E \times E \Rightarrow a + E \times E \Rightarrow a + b \times E \Rightarrow a + b \times a$.

Definition 8.7 (Ambiguous CFG). *A CFG G is ambiguous if there exists an $w \in L(G)$ such that w has at least two different derivation trees (equivalently, if w has two or more different leftmost derivations). The string w is said to be derived ambiguously.*

Remark. The CFG G_{expr} is ambiguous.

Ambiguity often comes up in the context of natural languages, eg:

Alice ate bread with cheese.
Alice ate bread with Bob.

Ambiguity also comes up in programming languages, eg

```
if x == 0 then if y == 1:
    then print "hi" else print "bye"
```

It is important to remove ambiguity whenever we can, for obvious reasons. Sometimes, we can replace a CFG with an equivalent CFG—i.e. a CFG generating the same language—that is not ambiguous. Different programs, for example, employ different techniques to eliminate ambiguity whilst preserving the power of the language itself.

Example 8.5. Define $G_4 = (\{E, F, G\}, \{a, b, +, \times, (,)\}, R, E)$, where the productions R are defined by

$$\begin{aligned} R: \quad E &\rightarrow E + F \mid F \\ F &\rightarrow F \times G \mid G \\ G &\rightarrow (E) \mid a \mid b \end{aligned}$$

Claim. G_4 is unambiguous, and $L(G_4) = L(G_{\text{expr}})$.

Proof. Omitted, but the proof can be completed by simply arguing that there is exactly one derivation for every string in $L(G_4)$, or by induction over the length of a string $w \in L(G_4)$, or by other methods.

Definition 8.8. A context-free language L is *inherently ambiguous* if every CFG that generates L is ambiguous.

Example 8.6. Let $\Sigma = \{a, b, c, d\}$. Define the language L of all strings in $L(a^*b^*c^*d^*)$ such that either the number of a 's is equal to the number of b 's and the number of c 's is equal to the number of d 's or the number of a 's is equal to the number of d 's and the number of b 's is equal to the number of c 's as

$$L = \{a^n b^n c^m d^m \mid n, m \geq 0\} \cup \{a^n b^m c^m d^n \mid n, m \geq 0\}.$$

Claim 1. L is a context-free language.

Proof 1. This is left as an exercise—just find a CFG that generates L .

Claim 2. L is an inherently ambiguous language.

Proof 2. Omitted, but the idea for the proof goes like this. We can show that a string of the form $a^n b^n c^n d^n \in L$ will have at least some number of distinct derivation trees for any CFG G that generates L .

Example 8.7. Let the CFG $G = (\{S\}, \{a, b\}, R, S)$, where R is given by

$$S \rightarrow SS \mid aSb \mid bSa \mid \varepsilon.$$

An initial guess for the language generated by G is

$$L(G) \stackrel{?}{=} \{w \in \{a, b\}^* \mid \text{the number of } a\text{'s and } b\text{'s are both even}\}.$$

However, this is not true, because $S \rightarrow aSb \rightarrow ab$, but $ab \notin L(G)$. Instead, we can show that

$$L(G) = \{w \in \{a, b\}^* \mid \text{the number of } a\text{'s and } b\text{'s are equal}\}.$$

Proof idea. We need to show implication in both directions:

- First, we show that if $S \xRightarrow{*} w$, that is, if $w \in L(G)$, then w has the same number of a 's as b 's. We can show that this is true for any $v \in (V \cup \Sigma)^*$ where $S \xRightarrow{*} v$ by induction on the number of steps.
- Next, we show that if the number of a 's in w is equal to the number of b 's in w , then $S \xRightarrow{*} w$. We can do this by strong induction on $|w|$ coupled with some case analysis.

Example 8.8. Consider all valid regular expressions over $\Sigma = \{a, b\}$. That is, take the alphabet as $\{a, b, \varepsilon, \emptyset, (,), \cup, \circ, *\}$, and the language as

$$\text{REGEXP} := \{w \in \{a, b, \varepsilon, \emptyset, (,), \cup, \circ, *\}^* \mid w \text{ is a valid regular expression}\}.$$

Note that $(a \cup b)^*b \in \text{REGEXP}$, but $((a \circ b) \notin \text{REGEXP}$.

Claim 8.9. REGEXP is not regular, but it is context-free.

Proof. To show REGEXP is context-free, we just need to give a CFG that generates REGEXP. One such CFG is $G_{\text{REGEXP}} = (\{S\}, \{a, b, \varepsilon, \emptyset, (,), \cup, \circ, *\}, R, S)$, where R is defined by

$$\begin{aligned} R: \quad S &\rightarrow a \mid b \mid \varepsilon \mid \emptyset \\ S &\rightarrow (S \cup S) \mid (S \circ S) \mid (S^*) \end{aligned}$$

If we allow a more liberal definition, omitting parentheses/concatenation, we may set

$$\begin{aligned} R: \quad S &\rightarrow a \mid b \mid \varepsilon \mid \emptyset \\ S &\rightarrow S \cup S \mid S \circ S \mid S^* \mid SS \mid (S) \end{aligned}$$

To show REGEXP is not regular, we will use the pumping lemma. Assume it is regular, and let p be the pumping number. Now choose the string w as

$$w = \underbrace{(\cdots (((a^*)^*)^* \cdots))^*}_{p \text{ times}},$$

so that $|w| = 2p + p + 1 \geq p$, and $w \in \text{REGEXP}$. However, for any way to parse w as $w = xyz$ with $|xy| \leq p$ and $y \neq \varepsilon$, y is within the first p '('s, that is, $y = ({}^k$ for some k . Thus when we pump, e.g. $i = 2$ times (or any $i \neq 1$ for that matter), $w = xy^iz$ has unbalanced parentheses, so that $xy^iz \notin \text{REGEXP}$. Contradiction. ■

Definition 8.10. A context-free grammar is in Chomsky normal form (CNF) if every rule takes the form $A \rightarrow BC \mid a$, where a is any terminal and A, B , and C are any variables—except that B and C may not be the start variable. In addition, we permit the rule $S \rightarrow \varepsilon$, where S is the start variable.

Claim 8.11. *Any context-free language is generated by a context-free grammar in CNF.*

Proof idea. We can convert any grammar G into CNF by doing the following: adding a new start variable, eliminating all ε -rules of the form $A \rightarrow \varepsilon$, eliminating all unit rules of the form $A \rightarrow B$, and patching up the grammar to ensure it generates the same language. Of course, we have to convert the remaining rules to the proper form.

Challenges

1. Consider the CFG G defined by the rules R given by

$$S \rightarrow aS \mid Sb \mid a \mid b$$

Prove (perhaps by induction on the string length) that no string in $L(G)$ has ba as a substring. Can you describe $L(G)$ informally using this result?

2. Consider the CFG G defined by the rules R given by

$$S \rightarrow aSbS \mid bSaS \mid \varepsilon$$

Show that $L(G)$ is the set of all strings with an equal number of a 's and b 's. Is this CFG “better” or “worse” than the CFG in Example 8.7?

3. Construct a CNF grammar for REGEXP.

9. More on CFLs (12/10)

Theorem 9.1. *The class of context-free languages is closed under the regular operations.*

We'll prove this theorem in separate lemmas. As a side remark, context-free languages are not closed under intersection and complement—we'll see this later.

Lemma 9.2. *The class of context-free languages is closed under union.*

Proof. Given two CFGs $G_1 = (V_1, \Sigma_1, R_1, S_1)$ and $G_2 = (V_2, \Sigma_2, R_2, S_2)$, we want to construct a G such that $L(G) = L(G_1) \cup L(G_2)$. Assume WLOG that V_1 and V_2 are disjoint, i.e. $V_1 \cap V_2 = \emptyset$. Let $G = (V, \Sigma, R, S)$, where

- $S \notin V_1, S \notin V_2$, i.e. a new start variable,
- $V = V_1 \cup V_2 \cup \{S\}$,
- $\Sigma = \Sigma_1 \cup \Sigma_2$,
- $R = R_1 \cup R_2 \cup \{S \rightarrow S_1 \mid S_2\}$.

We can prove that with this construction, for all $w \in \Sigma^*$, $S \xRightarrow{*} w$ if and only if $S_1 \xRightarrow{*} w$ and $S_2 \xRightarrow{*} w$. The proof is left as an exercise. ■

Lemma 9.3. *The class of context-free languages is closed under concatenation.*

Proof. Given two CFGs $G_1 = (V_1, \Sigma_1, R_1, S_1)$ and $G_2 = (V_2, \Sigma_2, R_2, S_2)$, we want to construct a G such that $L(G) = L(G_1) \cup L(G_2)$ as before, where $G = (V, \Sigma, R, S)$ and

- $S \notin V_1, S \notin V_2$, i.e. a new start variable,
- $V = V_1 \cup V_2 \cup \{S\}$,
- $\Sigma = \Sigma_1 \cup \Sigma_2$,
- $R = R_1 \cup R_2 \cup \{S \rightarrow S_1 S_2\}$.

We can show that $L(G) = L(G_1) \circ L(G_2)$ by showing that $\forall w \in \Sigma^*, S \xRightarrow{*} w$ if and only if $\exists w_1, w_2$ such that $w = w_1 \circ w_2$, and $S_1 \xRightarrow{*} w_1, S_2 \xRightarrow{*} w_2$. The proof is again left as an exercise. ■

Lemma 9.4. *The class of context-free languages is closed under Kleene star.*

Proof. Given $G_1 = (V_1, \Sigma_1, R_1, S_1)$, we want to construct a G such that $L(G) = (L(G_1))^*$. Let $G = (V, \Sigma, R, S)$, where

- $S \notin V_1$, is a new start variable,
- $V = V_1 \cup \{S\}$,
- $\Sigma = \Sigma_1$,
- $R = R_1 \cup \{S \rightarrow \varepsilon \mid S S_1\}$.

We can show that $L(G) = (L(G_1))^*$ by showing that $\forall w \in \Sigma^*, S \xRightarrow{*} w$ if and only if $w = \varepsilon$, or $\exists w_1 \dots, w_n$ such that $w = w_1 \circ \dots \circ w_n$ and $S_1 \xRightarrow{*} w_1, \dots, S_1 \xRightarrow{*} w_n$. ■

Theorem 9.5. *If L is a regular language, then L is context-free language.*

Proof idea. If L is regular, then L is generated by some regular expression R . We'll show that every regular expression R has an equivalent CFG G (i.e. one that generates the exact same language) using two proofs, first strong induction on the number of characters in R ¹⁰, and then by constructing a CFG for the DFA recognising L .

Proof 1. Base case, $|R| = 1$. The only possible options are $R = \varepsilon, \emptyset, a$, where $a \in \Sigma$. Then the CFGs generating these regular expressions are:

- $G = (\{S\}, \Sigma, \{S \rightarrow \varepsilon\}, S)$; note that $L(G) = \varepsilon = L(\varepsilon)$.
- $G = (\{S\}, \Sigma, \{S \rightarrow S\}, S)$; note that $L(G) = \emptyset = L(\emptyset)$.
- $G = (\{S\}, \Sigma, \{S \rightarrow a\}, S)$; note that $L(G) = a = L(a)$.

¹⁰Note the difference between the number of characters in R , $|R|$, and the number of strings in the language generated by R , $|L(R)|$. In fact, $|R| = |L(R)| = 1$ only when $R = \varepsilon$, $|L(R)| = 0$ and $|R| = 1$ when $R = \emptyset$, and $|L(R)| = \infty$ otherwise. Can you see why?

Inductive case, $|R| = n \geq 2$. Assume that Theorem 9.5 is true for all regular expressions R such that $|R| < n$; we want to show that it is true for $|R| = n$. We proceed by casework, given regular expressions R_1, R_2 :

- *Union, $R = (R_1 \cup R_2)$.* Note that $|R_1| < |R|$ and $|R_2| < |R|$, so that by the induction hypothesis, $\exists G_1, G_2$ such that $L(G_1) = L(R_1)$ and $L(G_2) = L(R_2)$. Using Lemma 9.2, there must exist G such that $L(G) = L(G_1) \cup L(G_2) = L(R_1 \cup R_2)$.
- *Concatenation, $R = (R_1 \circ R_2)$.* Note that $|R_1| < |R|$ and $|R_2| < |R|$, so that by the induction hypothesis, $\exists G_1, G_2$ such that $L(G_1) = L(R_1)$ and $L(G_2) = L(R_2)$. Using Lemma 9.3, there must exist G such that $L(G) = L(G_1) \circ L(G_2) = L(R_1 \circ R_2)$.
- *Kleene star, $R = (R_1)^*$.* Note that $|R_1| < |R|$, so that by the induction hypothesis, $\exists G_1$ such that $L(G_1) = L(R_1)$. By Lemma 9.4, there must exist G such that $L(G) = (L(G_1))^* = L(R_1^*)$.

The proof is now complete. ■

Proof 2. Let L be a regular language. Then there exists a DFA $D = (Q, \Sigma, \delta, q_0, F)$ such that $L(D) = L$. Build a context-free grammar $G = (V, \Sigma, R, S)$ as follows. Set $V = \{R_i \mid q_i \in Q\}$, i.e. G has a variable for every state of D . Now, for every transition $\delta(q_i, a) = q_j$, add a rule $R_i \rightarrow aR_j$. For every accepting state $q_i \in F$, add a rule $R_i \rightarrow \varepsilon$. Finally, make the start variable $S = R_0$. The proof follows immediately. ■

Definition 9.6 (Right-linear CFG). *A context-free grammar $G = (V, \Sigma, R, S)$ is right-linear if all rules in R are of the form $A \rightarrow \varepsilon$ or $A \rightarrow aB$, where $A, B \in V$ and $a \in \Sigma$.*

Theorem 9.7. *L is regular if and only if L is generated by a right-linear CFG.*

Proof. Omitted, but can be completed by showing that every right-linear CFG has a corresponding DFA and that every DFA has a corresponding right-linear grammar. ■

Corollary 9.8. *If L is regular, then L is context-free.*

There are languages that are not context-free—these languages are called *context-sensitive* languages. Try to think of a few examples of context-sensitive languages.

10. Non-CFLs, Tandem Pumping Lemma (17/10)

Theorem 10.1 (Tandem pumping lemma). *If L is a context-free language, then there exists a constant p , called the pumping length, such that $\forall w \in L$ with $|w| \geq p$, there exists a way to parse w as $w = uvxyz$ such that*

- $|vxy| \leq p$,
- $|vy| > 0$,
- for all $i = 0, 1, 2, \dots$, $uv^i xy^i z \in L$.

Proof. Omitted, but uses ideas about the derivation tree. ■

Using the tandem pumping lemma. To prove that a language L is not context-free, we just show that L does not satisfy the tandem pumping lemma. This means that we show that L satisfies the *negation* of the tandem pumping lemma, i.e. $\forall p$, there exists some $w \in L$ with $|w| \geq p$ such that for all the ways to parse w as $w = uvxyz$ where $|vxy| \leq p$ and $y > 0$, there always exists some $i \geq 0$ such that $uv^i xy^i z \notin L$.

Example 10.1. We claim that the language $L_1 = \{a^n b^n c^n \mid n \geq 0\}$ is not context-free. We can prove this via the tandem pumping lemma.

Proof. Assume L_1 is context free, and let p be the pumping length. Now choose $w = a^p b^p c^p$. Note that $w \in L_1$, and $|w| = 3p \geq p$. Consider now any parsing of w into $w = uvxyz$ where $|vxy| \leq p$ and $|vy| > 0$. We proceed by casework:

- Either of v or y contains two or more distinct types of symbols a, b, c . Now choose $i = 2$. In this case, $uv^2 xy^2 z$ is not of the form $a^n b^n c^n$, so $uv^2 xy^2 z \notin L$.
- Neither of v or y contains two or more distinct types of symbols. In this case, vy may contain at most two distinct types of symbols a, b, c . Choose $i = 0$; the resulting string $uv^0 xy^0 z$ will decrease the count of one or two types of symbols, but not the third. Thus $uv^0 xy^0 z \notin L$.

L_1 does not obey the tandem pumping lemma, so L_1 is not context-free. ■

Example 10.2. We claim that the language $L_2 = \{ww \mid w \in \{a, b\}^*\}$ is not context-free. We can show that this is true by the tandem pumping lemma. We have three attempts:
Attempt 1. Choose $w = a^p a^p \in L_2$, so that $|w| = 2p \geq p$. However, we fail to find a contradiction for every i , because there always exists a parsing that works:

$$\overbrace{\underbrace{aa}_v \underbrace{a \cdots a}_{p \text{ times}} \underbrace{aaa \cdots aa}_{p \text{ times}}}^z.$$

We can simply choose $u = \varepsilon$, $v = aa$, $x = \varepsilon$, $y = \varepsilon$, $z = a^{2p-2}$.

Attempt 2. Choose $w = a^p b a^p b \in L_2$, so that $|w| = 2p + 2 \geq p$. However, we still don't find a contradiction for every i , because there always exists a parsing that can be pumped:

$$\underbrace{aaa \cdots}_{u} \underbrace{aa}_v \underbrace{b}_x \underbrace{aa}_y \underbrace{\cdots ab}_z.$$

We can simply choose $u = a^{p-2}$, $v = a^2$, $x = b$, $y = a^2$, $z = a^{p-2}b$.

Attempt 3. Choose $w = a^p b^p a^p b^p \in L_2$, so that $|w| = 4p \geq p$. For any parsing of $w = uvxyz$ where $|vxy| \leq p$ and $|vy| > 0$. Then it is either the case that vxy is within the first $2p$ characters, vxy is within the middle $2p$ characters, or vxy is within the final $2p$ characters. Now choose $i = 2$, and consider the string $uv^2 xy^2 z$. In all of these cases, the number of a 's in the first half of the string is not the same as the number of a 's in the second half of the

string or the number of b 's in the first half of the string is not the same as the number of b 's in the second half of the string. Thus $uvvxyyz \notin L_2$, and the tandem pumping lemma does not hold. Contradiction— L_2 is not a context-free language.

Theorem 10.2. *The class of context-free languages is not closed under intersection.*

Proof. We will show the existence of context-free languages L_1 and L_2 such that $L_1 \cap L_2$ is not context-free. Take $L_1 = \{a^i b^i c^k \mid i, k \geq 0\} = \{a^i b^j c^k \mid i = j\}$. This language is context-free, because

$$L_1 = \underbrace{\{a^i b^i \mid i \geq 0\}}_{\text{this is a CFL (see 8.2)}} \circ \underbrace{L(c^*)}_{\text{this is regular, so also CFL}},$$

and CFLs are closed under $\circ$. Similarly, take $L_2 = \{a^i b^k c^k \mid i, k \geq 0\} = \{a^i b^j c^k \mid j = k\}$. L_2 is context-free, because

$$L_2 = \underbrace{L(a^*)}_{\text{this is regular, so also CFL}} \circ \underbrace{\{b^k c^k \mid k \geq 0\}}_{\text{this is a CFL (see 8.2)}},$$

and CFLs are closed under $\circ$. However,

$$L_1 \cap L_2 = \{a^n b^n c^n \mid n \geq 0\}$$

is not context-free, as we saw in Example 10.1. ■

Theorem 10.3. *The class of regular languages is not closed under complement.*

Proof. Assume towards contradiction that context-free languages were closed under complement. Then for any L_1, L_2 , by De Morgan's law,

$$L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}.$$

Since context-free languages are closed under union, if they are also closed under complement, then they would be closed under intersection. This is a contradiction—they are not closed under complement. ■

Challenges

1. For the alphabet $\Sigma = \{a, b, c, d\}$, define the language

$$L_1 = \{cw \mid w \in \{a, b\}^*, w = w^R\} \cup \{dw \mid w \in \{a, b\}^*\}.$$

Prove that L_1 is context-free.

2. Consider the following alternative direct proof for Theorem 10.3.

Let $A = \{a, b\}^* \setminus \{ww \mid w \in \{a, b\}^*\}$. We can show that A is a context-free language. However, the complement of this language $\overline{A} = \{ww \mid w \in \{a, b\}^*\}$, which is not context-free (see Example 10.2). Counterexample.

Show that A is a context-free language.

3. There is a stronger version of the tandem pumping lemma known as *Ogden's lemma*. It differs from the pumping lemma we proved by allowing us to focus on any p “distinguished” positions of a string w and guaranteeing that the strings to be pumped have between 1 and p distinguished positions. The advantage of this ability is that a language L may have strings consisting of two parts, one of which can be pumped without producing strings not in L , while the other does produce strings not in L when pumped. The formal statement of the lemma is: If L is a CFL, then there is a constant p such that if w is any string of length at least p in L , in which we select p positions to be distinguished, then we can write $w = uvxyz$, such that (1.) vxy has at most p distinguished positions. (2.) vy has a least one distinguished position. (3.) For all i , $uv^i xy^i z$ is in L .

Prove Ogden's lemma.

11. Pushdown Automata (10/17)

Pushdown automata are to context-free languages what NFAs are to regular languages, with the extra criterion that pushdown automata are equipped with a stack data structure. As we recall, a typical NFA computation step is the following:

- read input symbol
- switch the automata state
- move to the next input symbol.

However, a pushdown automaton, equipped with both the input alphabet and a stack alphabet, computes as follows:

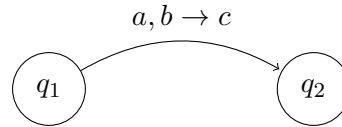
- read input symbol
- pop the top of the stack and read
- switch the automata state
- push to the top of the stack
- move to the next input symbol.

Definition 11.1 (Pushdown automaton, PDA). A PDA is a 6-tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ where

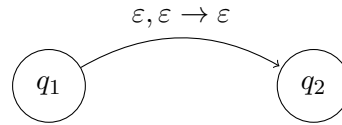
- Q is a finite set of states,
- Σ is the finite alphabet of input symbols,
- Γ is the finite stack alphabet,

- $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \rightarrow \mathcal{P}(Q \times \Gamma_\varepsilon)$ is the transition function, where $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$, $\Gamma_\varepsilon = \Gamma \cup \{\varepsilon\}$, and $\mathcal{P}(X)$ is the set of all subsets of X ,
- $q_0 \in Q$ is the start state,
- $F \subseteq Q$ is the set of accepting states.

The state diagram for the operation “read a from input, pop b from stack, push c to stack” is described by



Note that if there is no a in the input or no b in the stack (or both), we cannot take this transition. Note also that any of a, b , or c can be ε . For example we could have



which reads as “read nothing, pop nothing, push nothing.”

Definition 11.2. Given a PDA M , the computation of M starts with the start state and an empty stack, and processing begins at the start of the input. A string $w \in \Sigma^*$ is accepted if there exists a computation path on w that ends in an accepting state.

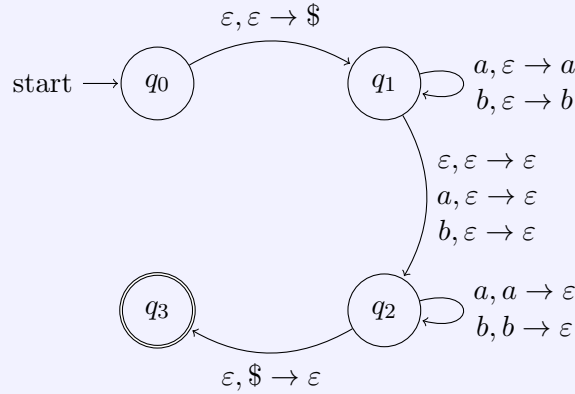
Theorem 11.3. A language L is context-free if and only if there exists a pushdown automaton recognising L .

Proof. Omitted. ■

Example 11.1. Consider the language of all palindromes, given by

$$\text{PALI} = \{w \in \{a, b\}^* \mid w = w^R\}.$$

We claim that PALI is context-free, and justify this claim by constructing a pushdown automaton recognising PALI. Here, we set $\Sigma = \{a, b\}$ and use a special symbol $\$$ in the stack alphabet to let us know when we have reached the bottom of the stack. Thus the stack alphabet is $\Gamma = \{a, b, \$\}$.



Informal argument for why this PDA works. If $w \in \text{PALI}$, here is an accepting computation:

- Start with the transition $q_0 \rightarrow q_1$, and push $\$$ to the stack.
- Take $q_1 \rightarrow q_1$ transition for $\lfloor |w|/2 \rfloor$ steps. That is, push the first half of w to the stack, one symbol at a time.
- Take $q_1 \rightarrow q_2$ transition. If $|w|$ is odd, read one character (the middle one).
- Take the $q_2 \rightarrow q_2$ transition for $\lfloor |w|/2 \rfloor$ steps, and then take the $q_2 \rightarrow q_3$ transition.

Since $w \in \text{PALI}$, the symbols in the second half of w match the symbols in the stack, and reach $\$$ exactly when we finished reading w . Thus this PDA accepts w .

Conversely, every accepting computation path on this PDA must start with q_0 and reach q_3 (i.e. the only accepting states). This means the PDA must go from $q_0 \rightarrow q_1$ (pushing $\$$ to the stack), then $q_1 \rightarrow q_1$ for zero or more times (pushing the input symbols to the stack), then $q_1 \rightarrow q_2$ (either consuming a symbol from the input or not). Now, we cannot take the $q_2 \rightarrow q_3$ transition unless we see the $\$$ symbol on the stack. Thus we must take the transition $q_2 \rightarrow q_2$ the same number of times as we took the transition $q_1 \rightarrow q_1$, and we can only take these transitions if there is a match in the input and the stack. This only succeeds if $w \in \text{PALI}$.

Theorem 11.4 (CFL-to-PDA equivalence). *A language L is a context-free language (that is, if there is a context-free grammar generating L) if and only if L has a pushdown automaton recognising it.*

11.1. Discussing these models in context

Think about the models we have seen so far. In terms of which languages the models recognise and what problems they can solve (and strictly not in terms of efficiency), which models are “stronger” than the other?

What if we had an NFA equipped with a queue data structure? What if we had an NFA with two stacks? It’s intuitively clear that this model is richer than the context-free languages, since they can recognise, for example, the language $\{a^n b^n c^n \mid n \geq 0\}$. Is an NFA with three stacks more

powerful than an NFA with two stacks? These are all questions worth thinking about.

As it turns out, all of the above models are just as strong as C programs or as Turing machines, et cetera. In particular,

Regular languages $\subsetneq$ Deterministic CFLs $\subsetneq$ Non-inherently ambiguous CFLs $\subsetneq$ CFLs
 $\subsetneq$ Context-sensitive languages $\subsetneq$ Turing machine computable languages $\subsetneq$ All languages.

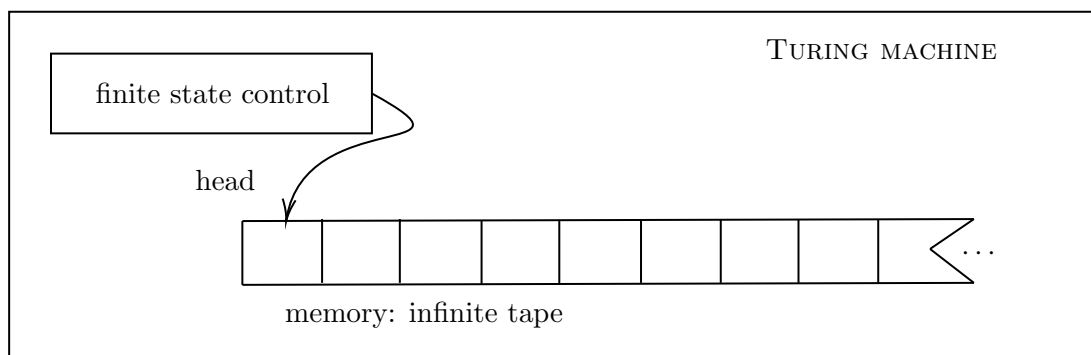
Challenge

For any language L , let $\text{SUFFIX}(L) = \{v \mid uv \in L \text{ for some string } u\}$. Show that the class of context-free languages is closed under the SUFFIX operation.

12. Turing Machines (20/10 and 26/10)

We now move to discussing a more powerful model of computation that can be adopted by general purpose computers. So far, we have seen DFAs/NFAs (both of which are good models for devices with a small amount of memory) and pushdown automata (which is a good model for devices that have unlimited memory that can only be used in the LIFO manner due to the stack data structure). The problem is that, like we have seen with non-regular languages and non-context-free languages, there are simple tasks that these models cannot execute.

Now, all of these previous models did not have unbounded memory, could not go back on the characters of an input, and couldn't write to memory. A *Turing machine* will have all of those capabilities. Informally, the Turing machine is structured as follows:

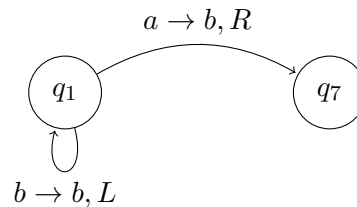


A computation of a Turing machine on an input $w \in \Sigma^*$ goes as follows. The starting configuration consists of the finite state control set to the start state q_0 , the head pointing to the first cell of the tape, and the tape containing the input string w followed by infinitely many blank symbols which we will write as $\sqcup$. In each step, depending on the current state and the current symbol on the tape, we switch to a state, write on the tape (and by so doing, overwriting the current symbol on the tape), and move the head right or left¹¹. There are two special states: the

¹¹If the head pointing to the first cell is instructed to move left, it stays on the first cell.

accepting state q_{acc} and the rejecting state q_{rej} . As soon as we enter either of these states, the Turing machine halts¹², returning accept or reject.

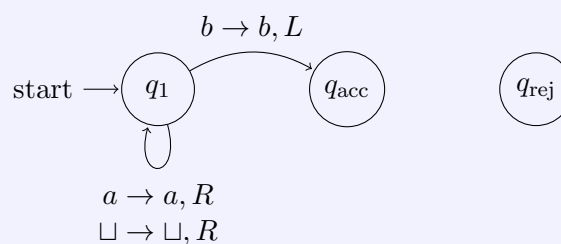
Turing machines also have their associated state diagrams. The state diagram for the operation “when see b on tape, write b , move left and stay in q_1 ; when see a on tape, write b in that cell of tape, move right, switch to q_7 ” is



Differences between a Turing machine and a DFA.

- We can go left and right on the input, not just right as is the case with a DFA.
- We have unbounded memory on the tape, as opposed to the DFA with finitely many states and deterministic transitions.
- We can read and write to input, instead of only reading as is the case with DFAs. The Turing machine immediately halts upon entering q_{acc} or q_{rej} , while the DFA halts only after processing all the characters in the input string.

Example 12.1. What is the set of strings accepted by the Turing machine below?



With a bit of thought, we find that it is $L = \{w \in \{a, b\}^* \mid w \text{ contains } b\}$. To see why, think about the computation of this Turing machine on strings that contain b versus strings that don't.

We often don't explicitly indicate q_{rej} since it is not reachable from the start state.

Definition 12.1 (Turing machine, TM, standard model). A Turing machine is a 7-tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{acc}, q_{rej})$ where:

- Q is the finite set of states,
- Σ is the finite alphabet,
- Γ is the finite alphabet called the “tape alphabet”; note that we must have $\Sigma \subseteq \Gamma$, $\sqcup \in \Gamma$, and $\sqcup \in \Sigma$,

¹²This is one crucial distinction between DFAs/NFAs and Turing machines.

- the start, accepting, and rejecting states are respectively $q_0, q_{acc}, q_{rej} \in Q$,
- the transition function is $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$.

Definition 12.2. A configuration for a TM M represents the status of computation at a certain point. A configuration takes the form uqv where $u, v \in \Gamma^*$ and $q \in Q$. This means that the state on the finite state machine is q , the content of the tape is uv followed by infinitely many blanks, and the head is pointing to the first character after u (i.e. the first character of v , or $\sqcup$ if $v = \varepsilon$).

Definition 12.3. For two configurations c_1, c_2 , we say $c_1 \vdash c_2$ if the application of δ on c_1 gives c_2 .

Example 12.2. Say configuration $c_1 = aq_7bab$ and $\delta(q_7, b) = (q_0, a, R)$. Then $c_1 \vdash c_2$ where $c_2 = aaq_0ab$, since $aq_7bab \vdash aaq_0ab$ via the application of δ .

Definition 12.4. For a TM $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{acc}, q_{rej})$ and a string $w \in \Sigma^*$:

- The initial configuration of M on w is q_0w (i.e. the state is q_0 , the head is pointing to the first cell containing the first character of w , and the tape has w followed by infinitely many blanks).
- An accepting configuration is $uq_{acc}v$ for any $u, v \in \Gamma^*$.
- A rejecting configuration is $uq_{rej}v$ for any $u, v \in \Gamma^*$.

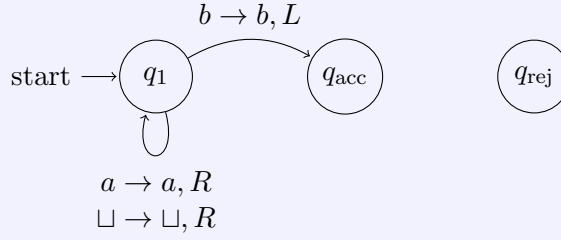
Definition 12.5. M accepts w (or w is accepted by M) if there exists a sequence of configurations $c_1, \dots, c_n$ such that

- $c_1 = q_0w$ (i.e. start with the initial configuration)
- $c_i \vdash c_{i+1}$ for all $i \in \{1, n-1\}$ (i.e. take steps according to the dictates of δ)
- $c_n = uq_{acc}v$ for some $u, v \in \Gamma^*$ (i.e. end with an accepting configuration).

Definition 12.6. The language recognised by a TM M is $L(M) = \{w \in \Sigma^* \mid M \text{ accepts } w\}$.

Definition 12.7. A language L is TM-recognisable if there exists a TM M that recognises L ; that is, $L(M) = L$. We say that L is recursively-enumerable, i.e. $L \in \text{RE}$.

Example 12.3. The Turing machine M_1 from the previous example says: “keep moving right; if you see a b , then accept.”



We already found that $L(M_1) = \{w \in \{a, b\}^* \mid w \text{ contains } b\}$. Here are the computations of M_1 on a few strings:

- $ababb$: $q_0ababb \vdash aq_0babb \vdash q_{acc}ababb$. Thus $ababb$ is accepted by M_1 .
- $bbbb$: $q_0bbbb \vdash q_{acc}bbbb$. Thus $bbbb$ is accepted by M_1 .
- aa : $q_0aa \vdash aq_0a \vdash aaq_0 \vdash aa \sqcup q_0 \vdash aa \sqcup \sqcup q_0 \vdash \dots$. Thus aa runs forever on M_1 .

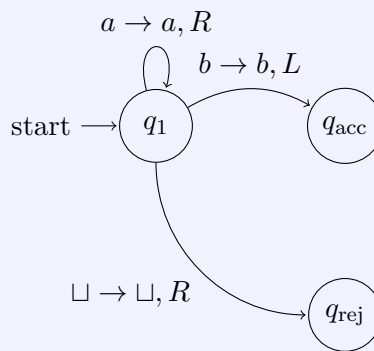
Definition 12.8. For any TM M and string $w \in \Sigma^*$, there are three possible outcomes:

- M accepts w implies $w \in L(M)$.
- M rejects w implies $w \notin L(M)$.
- M runs forever on w implies $w \notin L(M)$.

Definition 12.9. A Turing machine that halts (that is, exclusively either accepts or rejects) on all inputs is called a decider.

Definition 12.10. A language L is TM-decidable if there exists a TM M that recognises L , and moreover, M is a decider (we say that L is a recursive language).

Example 12.4. The Turing machine M_1 in Example 12.3 is not a decider, but the language $\{w \in \{0, 1\}^* \mid w \text{ contains } b\}$ is a decidable language^a. Here is a decider M_2 for it:



This Turing machine simply implements the instructions “keep moving right, if you see a b , accept; if you see a $\sqcup$, reject.” We can check that the language of both these machines is $L(M_2) = L(M_1) = \{w \in \{0, 1\}^* \mid w \text{ contains } b\}$. Here are some select computations of the machine:

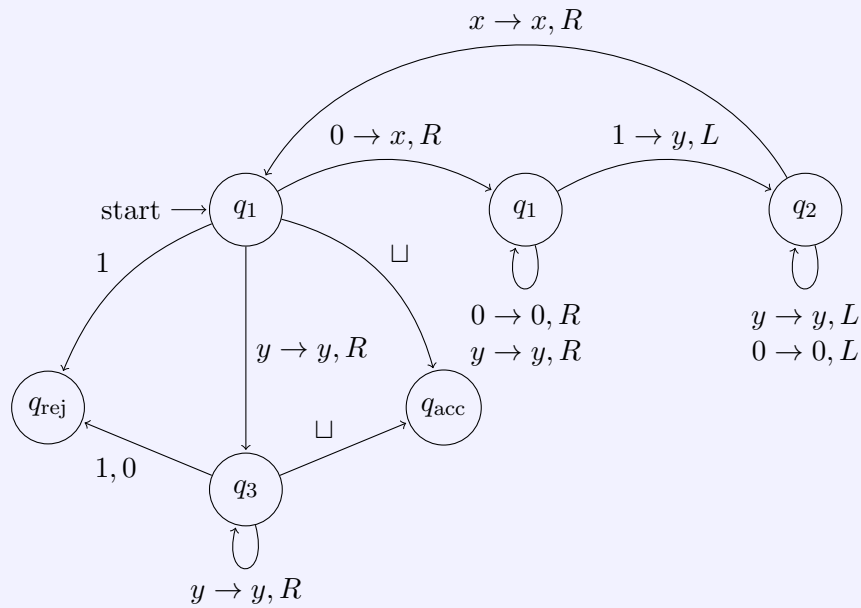
- aa : $q_0aa \vdash aq_0a \vdash aaq_0 \vdash aa \sqcup q_{\text{rej}}$. Thus aa is rejected by M_2 .
- ε : $q_0\varepsilon \vdash q_0 \vdash \sqcup q_{\text{rej}}$. Thus ε is rejected by M_2 .
- aba : $q_0aba \vdash aq_0ba \vdash aq_{\text{acc}}ba$. Thus aba is accepted by M_2 .

^aIn fact it is also regular—can you construct a DFA/NFA for it?

Example 12.5. Consider the language $L_3 = \{0^n1^n \mid n \geq 0\}$. This language is TM-decidable; we will show a TM-decider for L_3 .

Idea: We want to check each 0 against each 1. Moving back and forth, (1) mark the leftmost 0 with x , (2) go to the leftmost 1 and mark it with y , (3) go back to the left half of the string, repeat.

The following TM M_3 decides L_3 :



Fact. If a language is TM-decidable, then it is also TM-recognisable. TM-recognisable languages are not necessarily TM-decidable.

Challenges

1. It is true that all regular languages are also TM-decidable. Can you prove this? How would you construct a decider for an arbitrary regular language?
2. Consider the language $\text{POWER}_2 = \{1^{2^n} \mid n \geq 0\}$. This language is TM-decidable. How would you construct a decider M_{POWER_2} for POWER_2 ?

13. Input-output TMs, Equivalent models of TMs (31/10)

Recap. We have seen TM-decidable and TM-recognisable languages:

1. A language L is *TM-recognisable* if $\exists$ TM M such that M recognises L . That is, $\forall x \in \Sigma^*$:
 - $x \in L \implies M$ accepts x .
 - $x \notin L \implies M$ rejects x or M runs forever on x .
2. A language L is *TM-decidable* if $\exists$ TM M such that M recognises L , and M is a decider (that is, M halts on all inputs). That is, $\forall x \in \Sigma^*$:
 - $x \in L \implies M$ accepts x .
 - $x \notin L \implies M$ rejects x .

For the rest of class, we will use high-level descriptions of TM algorithms (think along the lines of macros/functions/subroutines). You should be convinced that each algorithmic step is implementable with a low-level TM.

Definition 13.1 (Input-output TM). *An input-output Turing machine is a 6-tuple specified by $(Q, \Sigma, \Gamma, \delta, q_{start}, q_{halt})$ starts with the input w written on the tape (that is, the TM has configuration $q_{start}w$). When the TM moves to state q_{halt} , computation stops, and the string written on the tape (up to the infinitely many $\sqcup$'s) is the output.*

Definition 13.2. *A function $f : \Sigma^* \rightarrow \Sigma^*$ is TM-computable if there exists an input-output TM M such that $\forall x \in \Sigma^*$, M on input x halts with $f(x)$ written on the tape.*

Example 13.1. The following functions are computable:

1. “On input n represented in unary, output $n + 1$ represented in unary.”
2. “On input n represented in binary, output $n + 1$ represented in binary.”
3. “On input w , insert a symbol $\#$ at the beginning and output $\#w$.”
4. “On an input of two strings $w_1\#w_2$ separated by the $\#$ symbol, copy the first string to the end of the second string.”
5. “On input $1^m\#1^n$, output 1^{m+n} .”
6. “On input $1^m\#1^n$, output $1^{m \cdot n}$.”
7. “On input $w_1\#w_2$ with $w_1, w_2 \in \{0, 1\}^*$, output $w_1 \cdot w_2$.”

13.1. Different models of Turing machines

Definition 13.3 (Equivalence). *We say that Turing machines M_1, M_2 (possibly in different models) are equivalent if they do the same thing on every input x . That is, $\forall x \in \Sigma^*$, M_1 accepts $x \iff M_2$ accepts x , M_1 rejects $x \iff M_2$ rejects x , and M_1 runs forever on $x \iff M_2$ runs forever on x .*

TM with stay-put transition. In this model of the Turing machine, the transition function is altered to $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R, S\}$. So the head can make state transitions despite remaining fixed on the current cell in the tape.

Theorem 13.4. *The stay-put TM is equivalent to the standard TM.*

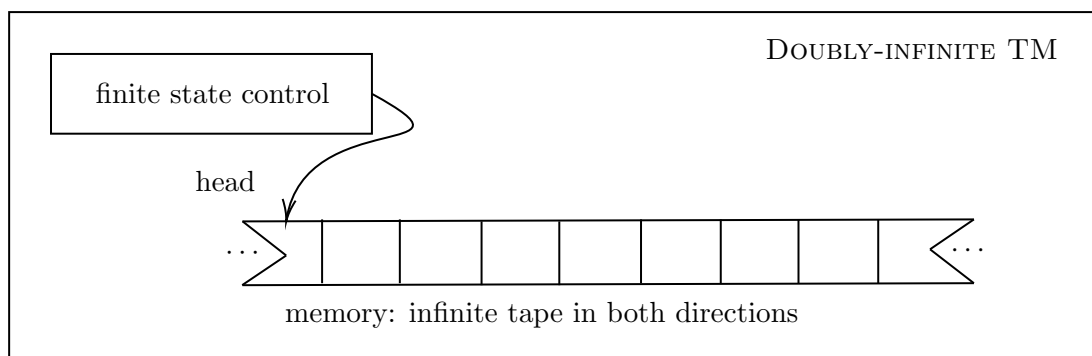
Proof. We need to prove equivalence in both directions.

1. *Given a standard TM, we can simulate it with the stay-put TM.* This is obvious, because the standard TM is just a special case of the stay-put TM without the instruction to stay-put.
2. *Given a stay-put TM, we can simulate it with a standard TM.* This is true, because we can keep all the characteristics of the TM exactly the same, except that when δ in the stay-put TM asks to stay-put, δ in the standard TM moves right, then moves left. We can implement this with extra states in the TM:



The proof is completed—stay-put TMs are equivalent to standard TMs. ■

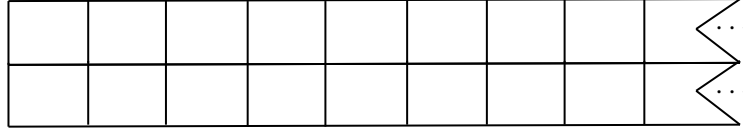
TM with doubly-infinite tape. This variant of Turing machine has cells of blanks extending in both directions, and starts with the head pointing to the first input character.



Theorem 13.5. *The doubly-infinite TM is equivalent to the standard TM.*

Proof. Once again, we prove equivalence in both directions:

1. *Given a standard TM, we can implement it with a doubly-infinite TM.* This is true, because we can keep all the characteristics of the TM exactly the same, except that we stay put when trying to move left from the first position. We can implement this by initially inserting a symbol $\#$ to the left of the input; once we see $\#$, then we have reached the “left-most end” of the tape.
2. *Given a doubly-infinite TM, we can implement it with a standard TM.* Imagine stacking both halves of the doubly-infinite TM one on top of another:



The new input alphabet is Σ' , Γ' , and δ' . Next, we do the following:

- Keep the alphabet the same, i.e. $\Sigma' = \Sigma$,
- Expand the tape alphabet to Γ' allowing two symbols per cell: $\Gamma' = \Gamma^2 \cup \Sigma \cup \{\sqcup\}$. For example, if $\Gamma = \{0, 1, \sqcup, \#\}$, then Γ' will have $4^2 = 16$ symbols, corresponding to

0	0	1	1	0	$\sqcup$	$\sqcup$	0	#	...
0	1	0	1	$\sqcup$	0	$\sqcup$	#	0	...

- Encode via state whether we are in the left/top or right/bottom position.
- Adjust the transitions δ accordingly.

Now the proof is complete. ■

TM with multiple tapes. A multitape Turing machine is like an ordinary Turing machine with several tapes, with a transition function equipped for reading, writing, and moving the heads on some or all of the tapes simultaneously. This transition function is defined as $\delta : Q \times \Gamma^k \rightarrow Q \times \Gamma^k \times \{L, R\}^k$, where k is the number of tapes. To specify transitions, we write

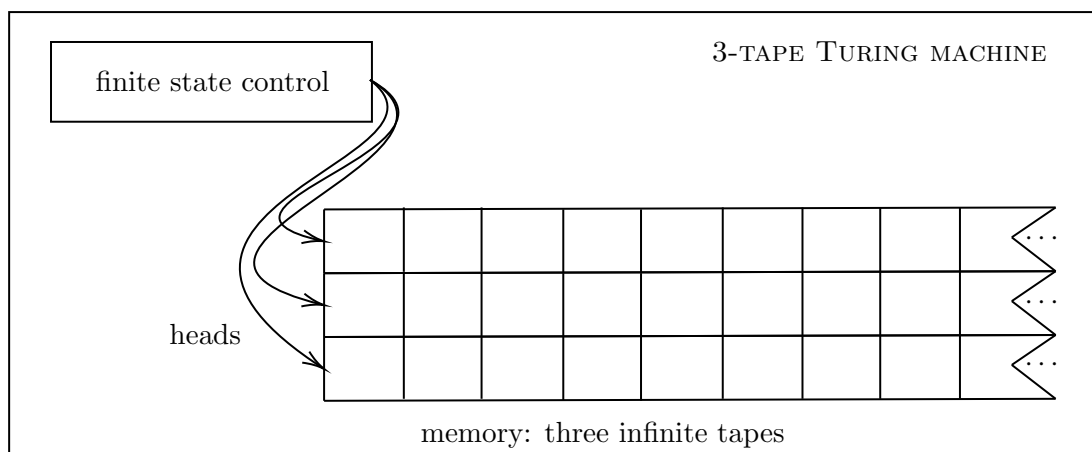
$$\delta(q_i, x_1, \dots, x_k) = (q_j, y_1, \dots, y_k, L, R, \dots, L)$$

to mean that if the multitape TM is in state q_i and heads $1, \dots, k$ are respectively reading symbols $x_1, \dots, x_k$, the machine goes to state q_j , writes symbols $y_1, \dots, y_k$, and directs each head to move left, right, or do nothing.

At the beginning of the computation, the input w is on the first tape (followed by blanks); all other tapes have blanks, and the head points to the first cell on each tape.

It may appear on first glance that multitape Turing machines are more powerful than the standard Turing machine with a single tape. However, as we will show, both models are equivalent to each other in capability.

Recall that two machines are equivalent if they recognise the same language.

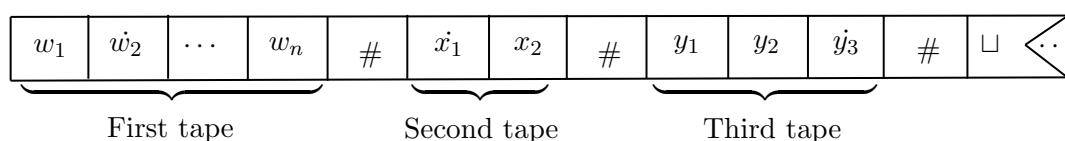


Theorem 13.6. *The multitape TM is equivalent to the standard TM.*

Proof. We once again prove equivalence in both directions. The first case is obvious and uninteresting: we can simulate a single state standard TM with a k -tape TM by ignoring all the other tapes. The reverse is more interesting. Given a k -tape TM, we can simulate it using a standard TM via the following procedure:

1. Bring the tape to the correct format:
 - Keep the content of all tapes on one tape, separated by #.
 - Keep a special “marked” version of the symbol that each head is pointing to.
2. For each single step of a k -tape TM, execute the following:
 - Scan input from left to right, record (via state) what each reading head points to.
 - Go back left and update the tape based on the transition function of the multitape TM.
 - Take care of edge cases. For example, if one of the heads needs to move right but encounters #, we need to shift the rest of the string on the tape one spot to the right and insert $\sqcup$ before continuing.

An example implementation is the following:



The proof is completed. ■

Theorem 13.7 clarifies that the quadratic overhead in running time of this construction is inherent.

Theorem 13.7. *The time taken by the single tape standard TM M to simulate n of the k -tape TM M_k is $\mathcal{O}(n^2)$.*

Proof. After n moves of M , the tape head markers cannot have separated by more than $2n$ steps. Thus, if M_k starts at the leftmost marker, it has to move no more than $2n$ cells right, to find all the head markers. It can then make an excursion leftward, changing the contents of the simulated tapes of M , and moving head markers left or right according to δ . Doing so requires no more than $2n$ moves left, plus at most $2k$ moves to reverse direction and write a marker in the cell to the right (in the case that a tape head of M moves right).

Thus the number of moves by M_k needed to simulate one of the first n moves is no more than $4n + 2k$. k is a constant independent of the number of moves simulated, so the number of moves is actually $\mathcal{O}(n)$. Simulating n moves requires less than n times this amount, or $\mathcal{O}(n^2)$. ■

Challenges

1. In Example 13.1, we gave examples of some computable functions. Can you describe the Turing machines that compute these functions?
2. A k -head TM has k heads reading the cells of one tape. A move of this TM depends on the state and on the symbol scanned by each head. In one move, the TM can change state, write a new symbol on the cell scanned by each head, and can move each head left, right, or keep it stationary. Since several heads may be scanning the same cell, we assume the heads are numbered 1 through k , and the symbol written by the highest numbered head scanning a given cell is the one that actually gets written there.

Prove that the k -head TM is equivalent to the standard TM.

3. A two-dimensional TM has the usual finite state control but a tape that is a two-dimensional grid of cells, infinite in all directions. The input is placed on one row of the grid, with the head at the left end of the input and the control in the start state, as usual. Acceptance is by entering a final state, also as usual.

Prove that the two-dimensional TM is equivalent to the standard TM.

14. Non-deterministic Turing machines (02/11)

Quiz 5 common error. Many people got the following problem incorrectly.

Below is a buggy “proof” that the class of TM-recognisable languages is closed under complement. However, the statement is false. Identify the line with the error in the proof.

1. Let L be a TM-recognisable language. Then there exists a TM M recognising it, namely that $L(M) = L$. We take M' to be identical to M except that we flip the roles of the accepting and rejecting states.
2. Since M is deterministic, for every string w there is a single unique computation of M on w .
3. Therefore the computation of M' on w is identical to the computation of M , except that if M accepts M' rejects, and if M rejects M' accepts.
4. Now for any string w , $w \in L \implies M \text{ accepts } w \implies M' \text{ rejects } w \implies w \notin L(M')$.

5. On the other hand, $w \notin L \implies M \text{ rejects } w \implies M' \text{ accepts } w \implies w \in L(M')$.
6. From 4 and 5 above, $w \in L \iff w \notin L(M')$, which means $L(M')$ is the complement of L , and thus the complement of L is also TM-recognisable.

This proof is wrong, because M is a recogniser and not a decider! We should instead have, in line five, $w \notin L \implies M \text{ rejects } w$ or M runs forever on w . Thus it is either the case that M' accepts w and $w \in L(M')$, or M' runs forever on w and $w \notin L(M')$.

Here is a counterexample for the construction, from the first question on the quiz:

M : “If the input starts with 0, accept; else, run forever.”

Obviously $L(M) = L(0\Sigma^*)$. If we flip the accept and reject states of M , we obtain

M' : “If the input starts with 0, reject; else, run forever.”

Now $L(M') = \emptyset$. But this is not a complement of $L(M)$ above! For some more intuition for why the statement is false, consider M' again:

M' = “On input w :

1. if M accepts w , reject
2. else [if M rejects w or runs forever], accept.”

But this is not implementable, since we cannot check whether M runs forever on w —we would need ∞ -time!

If we replace the word “recognisable” with “decidable”, we can get a correct theorem and proof:

Theorem 14.1. *The class of TM-decidable languages is closed under complement.*

Proof. Let L be TM-decidable. Then there exists M such that $L(M) = L$. Define M' to be the same as M , except flip the states q_{acc} and q_{rej} . Then for any string $w \in \Sigma^*$, there is exactly one computation of M' on w , which is the same except that if M accepts w , M' rejects w , and if M rejects w , M' accepts w . Thus:

- $w \in L \implies M \text{ accepts } w \implies M' \text{ rejects } w \implies w \notin L(M')$.
- $w \notin L \implies M \text{ rejects } w \implies M' \text{ accepts } w \implies w \in L(M')$.

M' decides $\bar{L}$, so $\bar{L}$ is TM-decidable. ■

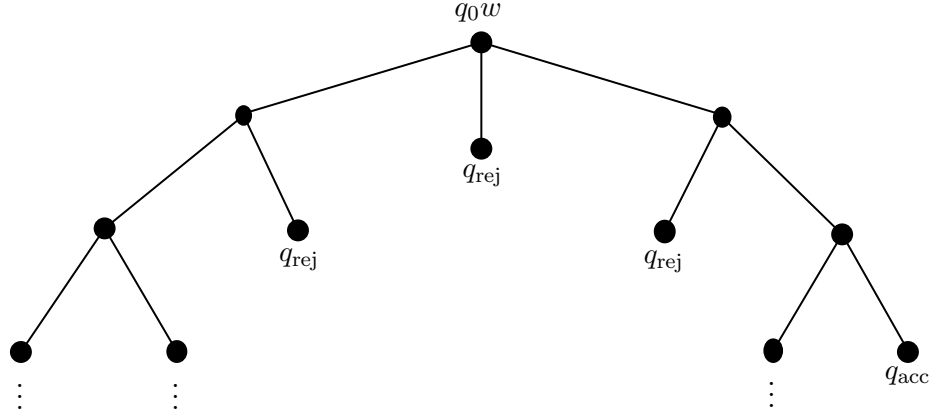
14.1. Non-deterministic TMs

Definition 14.2 (Non-deterministic Turing machine, NTM). *The machine $N = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})$ is a non-deterministic Turing machine where every other component retains the same meaning as the standard (deterministic) TM except that $\delta : Q \times \Gamma \rightarrow \mathcal{P}(Q \times \Gamma \times \{L, R\})$.*

Definition 14.3. *N accepts w if there exists an accepting computation on w . We say that the language of N is $L(N) = \{w \in \Sigma^* \mid N \text{ accepts } w\}$*

Note that while the (deterministic) Turing machine M has one computation on any input w (see 12.2), the non-deterministic Turing machine N has a computation tree of possible outcomes, where each node is a configuration uqv .

An NTM accepts on a string w if any node of the computation tree is in the state q_{acc} . For example, the NTM that computes on w according to the following computation tree accepts.



Theorem 14.4. *These models are equivalent; that is (1) given a TM M , there exists an equivalent NTM N ; (2) given an NTM N , there exists an equivalent TM M .*

Proof. The first part is uninteresting, obviously we immediately know that given a TM M , there exists an equivalent NTM N , since TMs are a special case of NTMs.

The second part is more interesting. Given an NTM N , we will construct a TM M such that if there exists an accepting computation branch of N on w , M accepts w ; else, if all computations of N on w reject, M rejects w ; else (i.e. there is no accepting branch of N on w , and there is at least one infinite branch) M runs forever on w . Thus we would have $w \in L(N) \iff M$ accepts w .

The idea is to construct M as follows:

M = “On input w ,

1. explore entire computation tree of N on w in breadth-first fashion¹³.
2. if N would ever switch to q_{acc} , M accepts.”

Implementation. We use a three-tape TM. The first tape T_1 is read-only, and has the input string on it. The second tape T_2 has the configuration in the current node in the tree. The third tape T_3 tracks which non-deterministic choice to try next. We repeat the following until termination:

1. Copy tape T_1 onto tape T_2 , erasing anything that was previously there.
2. Simulate N on tape T_2 by running N via the application of δ :
 - Each time there is a non-deterministic choice, look at tape T_3 to decide which choice to take.
 - If the choice is valid, move right on tape T_3 to the next position, and continue simulating N .
 - Continue until one of the following happens:

¹³Note that this is important, because exploring in depth-first fashion can get us stuck in an infinite branch in the case that the NTM is not a decider.

- If N accepted, accept (and halt).
 - If the choice on tape T_3 is not valid, or tape T_3 ran out, or N rejected, go to step (3).
3. To indicate the next node to explore, replace the string on tape T_3 with the next string determined by the following procedure: If b is the maximum branching factor¹⁴, then tape T_3 initially contains one symbol, and it is incremented from 0 to $b - 1$ for strings of a given length. For example, if $b = 3$, then tape T_3 content is 0, 1, 2, 00, 01, 02, $\dots$, 22, 000, 001, $\dots$
 4. Return to step (1), and restart the process.

This construction is sufficient, and so the proof is done. ■

The following theorem shows that this construction has exponential overhead in running time. Whether or not this exponential overhead is inherent is unknown, and related to the biggest open problem (the P vs NP problem).

Theorem 14.5. *Let $t : \mathbb{N} \rightarrow \mathbb{N}$, where $t(n) \geq n$. Then for any $t(n)$ -time nondeterministic TM N , there is an equivalent $2^{\mathcal{O}(t(n))}$ -time deterministic single tape TM M .*

Proof. Let $w \in \{0, 1\}^*$, and let B_w be the BFS tree created when searching the computation tree of N on input w . On any input length n , every branch of B_w has length at most $t(n)$. Let b be the maximum branching factor, so that each node in B_w has at most b children. Every tree with at most b children and depth at most t has at most $b^t = (2^{\log b})^t = 2^{t \log b}$ nodes in total. For sufficiently large n , the time required to search this tree is at most $2^{\mathcal{O}(t(n))}$. ■

15. Decidability (09/11)

Recap. We have now seen several models equivalent to (or weaker than) a TM. In fact, there are many more, including NFAs with two stacks, NFAs with one queue, general grammars, the Lambda calculus¹⁵, random access machines, your favourite programming language, etc.

Theorem 15.1 (Church-Turing Thesis). *Every reasonable model of computation (or notion of “algorithm”) can be simulated by a Turing machine.*

To prove L is recognisable, describe an algorithm M , and prove that:

$$\begin{aligned} x \in L &\implies M \text{ accepts } x \\ x \notin L &\implies M \text{ rejects } x \text{ or } M \text{ runs forever on } x \end{aligned}$$

To prove L is decidable, describe an algorithm M , and prove that:

$$\begin{aligned} x \in L &\implies M \text{ accepts } x \\ x \notin L &\implies M \text{ rejects } x \end{aligned}$$

There are three levels of detail we use in describing a TM (i.e. its algorithm).

¹⁴That gives the maximum number of choices for δ .

¹⁵This was Alonzo Church’s model of computation.

- For *formal level* descriptions, we must use the standard definition of TM given earlier (single tape, deterministic), and provide all transitions in detail (using either a table/list) or a state transition diagram. The only exception is that omitted transitions are assumed to go to q_{rej} .
- For *implementation level* descriptions, we are also allowed to use one of the TM variants we proved to be equivalent (e.g. a multitape TM or a non-deterministic TM), and we can use low-level subroutines like the ones we say in class (for example, we can say “insert a blank and shift the rest of the tape content one space to the right” or “copy the content of the first tape onto the second tape”).
- For *high level* descriptions, we do not need to discuss the tapes and head, but we must make sure that each step in the algorithm is clear and well-defined (we must be convinced that with enough time and patience, that step could be translated to implementation level and then to formal level; it may help to think about whether we are convinced that we can write an implementation of the step in our favourite programming language).

Theorem 15.2. *A language L is decidable if and only if both L and $\bar{L}$ are both recognisable.*

Lemma 15.3. *If L is decidable, then both L and $\bar{L}$ are recognisable.*

Proof. Assume that L is decidable, so that there is a TM M that is a decider for L . Then we immediately know that L is recognisable, since a decider is a special case of a recogniser. $\bar{L}$ is also recognisable: take M , and construct an M' by flipping the accept and reject states of M . Since M was a *decider* for L , then by Theorem 14.1, M' is a decider for $\bar{L}$, and hence also a recogniser. ■

Lemma 15.4. *If both L and $\bar{L}$ are recognisable, then L is decidable.*

Proof. Assume that L and $\bar{L}$ are recognisable, that is, $\exists M_1, M_2$ TMs such that M_1 recognises L , and M_2 recognises $\bar{L}$. We want to construct a decider M for L .

The wrong idea for this construction of M would be the following:

$M =$ “On input x :

1. run M_1 on x , if it accepts, accept
2. run M_2 on x , if it accepts, reject.”

This is bad, because M is *not* a decider, and hence may get stuck in an infinite loop in step 1 and never reach step 2. The correct construction is the following:

$M =$ “On input x :

1. run M_1 on input x and M_2 on input x in parallel
2. if M_1 accepts x , accept
3. if M_2 accepts x , reject.”

Note that $x \in L \iff M_1 \text{ accepts } x$, since M_1 recognises L . Additionally, $x \notin L \iff x \in \bar{L} \iff M_2 \text{ accepts } x$, since M_2 recognises $\bar{L}$. Thus for any x , exactly one of M_1, M_2 will eventually accept x , and so M will halt on every input.

An alternative construction of a decider for L is the following.

$M =$ “On input x :

1. for $i = 1, 2, 3, \dots$:
2. run M_1 on x for i steps
3. if it accepts, accept
4. run M_2 on x for i steps
5. if it accepts, reject.”

We can prove M decides L : $x \in L \implies M_1$ accepts x after some i steps (while M_2 never accepts x , since $x \notin L$) $\implies M$ rejects x . Likewise, $x \in \bar{L} \implies M_2$ accepts x after some i steps (while M_1 never accepts x , since $x \notin L$) $\implies M$ rejects x . We once again have our decider. ■

Every TM or algorithm can be encoded as a string. We won’t be specific about the details of the encoding (those details are left for the Appendix), but we will assume that there is an agreed-upon encoding that is easy to check for validity. We will write $\langle M \rangle$ to denote the encoding of a TM M , $\langle D \rangle$ to denote the encoding of a DFA D , and $\langle X \rangle$ to denote the encoding of an object X . We will often construct algorithms that take such an encoding as input, and check something about it.

15.1. Examples of decidable languages

1. (Acceptance problem for DFAs). $A_{\text{DFA}} = \{\langle D, w \rangle \mid D \text{ is a DFA, } w \text{ a string, and } D \text{ accepts } w\}$ is a decidable language.

Proof. Here is a decider for A_{DFA} :

$M_{A_{\text{DFA}}} =$ “On input x :

1. ¹⁶check that x is a valid encoding $\langle D, w \rangle$
2. if not, reject
3. run D on w
4. if D accepts, accept
5. if D rejects, reject.”

Check that $x \in A_{\text{DFA}} \implies x = \langle D, w \rangle$ where D is a DFA, w a string, and D accepts $w \implies M_{A_{\text{DFA}}}$ accepts x (in step 4). On the other hand, $x \notin A_{\text{DFA}} \implies$ either x is not a valid encoding or it is a valid encoding but D rejects $w \implies M_{A_{\text{DFA}}}$ rejects x (in step 1) or $M_{A_{\text{DFA}}}$ rejects x (in step 2) $\implies M_{A_{\text{DFA}}}$ rejects x . The proof is complete. ■

2. (Acceptance problem for NFAs.) $A_{\text{NFA}} = \{\langle N, w \rangle \mid N \text{ is a NFA, } w \text{ a string, and } N \text{ accepts } w\}$ is a decidable language.

Proof. Here is a decider for A_{NFA} :

$M_{A_{\text{NFA}}} =$ “On input $\langle N, w \rangle$:

1. convert N to an equivalent DFA D
2. check via $M_{A_{\text{DFA}}}$ if D accepts w
3. if so, accept
4. else, reject.”

¹⁶We conventionally skip this step moving forward; assume we do have a valid encoding

We know that there is a deterministic algorithm (with an inherent exponential blowup in the number of states) for converting an NFA to an equivalent DFA, so step (1) takes finite time. Furthermore, N and D are equivalent, so $M_{A_{\text{NFA}}}$ accepts $\langle N, w \rangle$ if and only if D accepts w if and only if N accepts w . Thus $M_{A_{\text{NFA}}}$ is a decider for A_{NFA} . ■

3. (Emptiness problem for DFAs.) $E_{\text{DFA}} = \{\langle D \rangle \mid D \text{ is a DFA, and } L(D) = \emptyset\}$ is decidable.

Proof. Here is a decider for E_{DFA} :

$M_{E_{\text{DFA}}} =$ “On input $\langle D \rangle$ where D is a DFA:

1. if D has no accepting states, accept
2. mark all accepting states
3. repeat until no new state is added:
4. mark all states that have a transition into a marked state
5. if the start state is marked, reject
6. else, accept.”

Exercise: verify that this decides E_{DFA} , using the fact that the loop in steps 3-4 is only executed finitely many times, since D has finitely many states. ■

16. More decidable languages, A_{TM} and NE_{TM}

Review of quiz 6 common error. Many people got the following problem on quiz 6 incorrectly:

What is the language recognised by the following TM?

$T =$ “On input $\langle M \rangle$ where M is a TM:

1. run M on ε , if it accepts, accept
2. run M on 101, if it accepts, accept
3. reject.”

The language is $L(T) = \{\langle M \rangle \mid M \text{ is a TM and } M \text{ accepts } \varepsilon \text{ or } M \text{ rejects } \varepsilon \text{ and accepts } 101\}$.

- Note that:
 - If M accepts ε and runs forever on 101, T accepts $\langle M \rangle$, and $\langle M \rangle \in L(T)$.
 - If M accepts 101 and runs forever on ε , then T runs forever on $\langle M \rangle$, and $\langle M \rangle \notin L(T)$.
 - On input 101, the algorithm realises that 101 is probably not a valid encoding of a TM, so T rejects immediately.
 - Also note that $L(T) \neq \{\langle M \rangle \mid M \text{ is a TM and } M \text{ accepts } \varepsilon \text{ or } M \text{ accepts } 101\}$ —this would be the case if T was running M on ε and 101 in parallel.
 - Thinking big picture: we cannot have a step “run M on 101, if it runs forever, reject/accept/run forever.” We cannot test whether or not a program runs forever!
-

16.1. More decidable languages

1. (Equivalence for DFAs.) $EQ_{\text{DFA}} = \{\langle D_1, D_2 \rangle \mid D_1, D_2 \text{ are DFAs and } L(D_1) = L(D_2)\}$ is decidable.

Proof. We will not give an explicit construction of a machine M here. Instead, we will note that (as we can show),

$$L(D_1) = L(D_2) \iff (L(D_1) \cap \overline{L(D_2)}) \cup (L(D_2) \cap \overline{L(D_1)}) = \emptyset.$$

Thus, in a similar fashion to A_{NFA} , we can build a new DFA D' accepting $L(D_1) \oplus L(D_2)$, and we use the algorithm $M_{E_{\text{DFA}}}$ from last class to check if this language is empty. ■

2. (Acceptance for CFGs.) $A_{\text{CFG}} = \{\langle G, w \rangle \mid G \text{ is a CFG, and } w \in L(G)\}$ is decidable.

Proof. We will construct a decider $M_{A_{\text{CFG}}}$ for A_{CFG} as follows:

$M_{A_{\text{CFG}}} =$ “On input $\langle G, w \rangle$:

1. Convert G to G' in CNF using the procedure outlined in Claim 8.11
2. G' has some finite number of rules k . Thus, there are $k^{2|w|-1}$ possible derivations of length $2|w| - 1$. Enumerate all of these rule sequences, trying all of them and discarding invalid sequences.
3. if any sequence produces w , accept
4. reject.”

Now, $\langle G, w \rangle \in A_{\text{CFG}}$ if and only if G can generate w , which is true if and only if G' can generate w in $2|w| - 1$ steps. Thus, if $\langle G, w \rangle \in A_{\text{CFG}}$ then some rule sequence will generate w , and $M_{A_{\text{CFG}}}$ will accept. Otherwise no such rule sequence will generate w , and M will reach step 4 and reject. ■

3. (Emptiness for CFGs.) $E_{\text{CFG}} = \{\langle G \rangle \mid G \text{ is a CFG, and } L(G) = \emptyset\}$ is decidable.

Proof. We will not give an explicit decider here, but will instead sketch out the construction. We use a similar marking approach to E_{DFA} , beginning with all the terminals marked, and then marking A if $A \rightarrow x$ is a rule for some terminal x , or if $A \rightarrow BC$ is a rule for marked variables BC . When finished, reject if S is marked and accept otherwise. ■

16.2. $A_{\text{TM}} \in \text{RE}$

We define $A_{\text{TM}} = \{\langle M, w \rangle \mid M \text{ is a TM, and } M \text{ accepts } w\}$.

Theorem 16.1. A_{TM} is recognisable.

Proof. We construct the following algorithm:

- $U =$ “On input $\langle M, w \rangle$ where M is a TM:
1. run M on w
 2. if M accepts, accept
 3. if M rejects, reject.”

We now prove that U recognises A_{TM} . If $x \in A_{\text{TM}}$, then $x = \langle M, w \rangle$ where M is a TM and M accepts w . Thus, U accepts x .

If $x \notin A_{\text{TM}}$, then either:

- (a) We have a bad encoding, i.e. $x \neq \langle M, w \rangle \implies U$ rejects $x \implies U$ does not accept x .
- (b) $x = \langle M, w \rangle$ and M rejects $w \implies U$ rejects $x \implies U$ does not accept x .
- (c) $x = \langle M, w \rangle$ and M runs forever on $w \implies U$ runs forever on $x \implies U$ does not accept x .

We have our recogniser—we call U the *universal Turing machine*. ■

Questions. Does A_{TM} contain $\langle U, \langle U \rangle \rangle$? What does U do on input $\langle U, \langle U \rangle \rangle$?

16.3. $NE_{\text{TM}} \in \text{RE}$

We define $NE_{\text{TM}} = \{\langle M \rangle \mid M \text{ is a TM, and } L(M) \neq \emptyset\}$.

Theorem 16.2. NE_{TM} is recognisable.

Wrong proof. Construct the following recogniser for NE_{TM} :

$M_{NE_{\text{TM}}} =$ “On input $\langle M \rangle$ where M is a TM:

1. for all strings $s \in \Sigma^*$:
2. run M on s ,
3. if M accepts, accept
4. if M rejects, continue.”

This is algorithmically implementable, since we can go over all $s \in \Sigma^*$ in lexicographic order by length. However, it doesn’t recognise NE_{TM} : if $\langle M \rangle$ is a TM that accepts some string s , but runs forever on an earlier string s' , then $\langle M \rangle \in NE_{\text{TM}}$, but the above runs forever on $\langle M \rangle$. ■

(Correct) Proof. We can use a procedure called *dovetailing* to construct a correct recogniser for NE_{TM} that terminates in finite time:

$R =$ “On input $\langle M \rangle$ where M is a TM:

1. for $i = 1, 2, 3, \dots$:
2. for all $s \in \Sigma^*$ where $|s| \leq i$:
3. run M on s for i steps
4. if M accepts, accept.”

We claim that R recognises NE_{TM} , because:

$$\begin{aligned} \langle M \rangle \in NE_{\text{TM}} &\implies \text{there exists a string } s \text{ that is accepted by } M \text{ after some } k \text{ steps} \\ &\implies \text{when the loop is in } i = \max(|s|, k) \text{ iteration, it will be accepted} \\ &\implies R \text{ accepts } \langle M \rangle. \end{aligned}$$

On the other hand,

$$\begin{aligned} x \notin NE_{\text{TM}} &\implies \text{either } x \neq \langle M \rangle \text{ (in which case } x \text{ is rejected) or } x = \langle M \rangle \\ &\implies R \text{ runs forever on } x \text{ since } L(M) = \emptyset \\ &\implies R \text{ does not accept } x. \end{aligned}$$

Ding, ding, ding—we have our recogniser! ■

17. Counting languages (16/11)

Definition 17.1. Two sets A, B must have the same cardinality—that is, $|A| = |B|$ —if there is a bijection $f : S \rightarrow T$. If no such bijection exists, we say that S and T have different cardinalities, and write $|S| \neq |T|$.

Definition 17.2. A set S is countable if S is finite or $|S| = |\mathbb{N}|$. We say that S is uncountable otherwise. Examples of countable sets include the set of all integers $\mathbb{Z}$, and the set of all rational numbers $\mathbb{Q}$.

Example 17.1. Let $\mathbb{Z} = \dots, -1, 0, 1, \dots$ be the set of all integers, and let $\mathbb{N} = 1, 2, \dots$ be the set of all naturals. Then $|\mathbb{Z}| = |\mathbb{N}|$; consider the function $f : \mathbb{N} \rightarrow \mathbb{Z}$ given by

$$f(n) = \begin{cases} n/2 & \text{if } n \text{ is even} \\ (1-n)/2 & \text{if } n \text{ is odd.} \end{cases}$$

Theorem 17.3. Any countable union of countable sets is countable—that is, if $\{E_i\}_{i=1}^{\infty}$ are countable sets, then $S = \bigcup_{n=1}^{\infty} E_n$ is also countable.

Theorem 17.4. Any finite product of countable sets is countable, that is, if all the sets $E_1, \dots, E_n$ are countable sets, then

$$E_1 \times E_2 \times \dots \times E_n = \{(a_1, \dots, a_n) \mid a_i \in E_i\}$$

is countable.

Example 17.2. The rational numbers $\mathbb{Q}$ are countable. To prove this, we can consider every element of $\mathbb{Q}$ as a pair (p, q) written as p/q , where $p, q \in \mathbb{Z}$ and $q \neq 0$. Of course, there is some overlap: we say that $p/q = p'/q'$ if $pq' = p'q$. The upshot is that we can map every element of $\mathbb{Q}$ to a pair $(p, q) \subseteq \mathbb{Z} \times \mathbb{Z}$ injectively, e.g. by taking p, q coprime with $q > 0$. By Theorem 17.4, since $\mathbb{Z}$ is countable, $\mathbb{Z} \times \mathbb{Z}$ is also countable. Also, since we just exhibited an injection $\mathbb{Q} \rightarrow \mathbb{Z} \times \mathbb{Z}$, we know that $\mathbb{Q}$ is in bijection with a subset of a countable set, hence $\mathbb{Q}$ is countable.

Example 17.3. The set $\{(x_1, x_2, \dots) \mid x_i \in \{0, 1\}\} = \{0, 1\}^{\mathbb{N}}$ of all infinite binary sequences is uncountable. To prove this, suppose that this set were countable, so that we can list its elements $\{x_1, x_2, \dots\}$ where each $x_i \in \{0, 1\}^{\mathbb{N}}$ in lexicographic order. Then construct the infinite sequence $y = (y_1, y_2, \dots) \in \{0, 1\}^{\mathbb{N}}$ where each y_i is the opposite of the i th entry of x_i . Then y is not equal to any x_i because their i th entries are different. Since we assumed that the list $\{x_1, x_2, \dots\}$ contained all elements of $\{0, 1\}^{\mathbb{N}}$, that means $y \notin \{0, 1\}^{\mathbb{N}}$, but this is a contradiction. Hence $\{0, 1\}^{\mathbb{N}}$ is uncountable.

Fact 17.5. If $A \subseteq B$, then obviously $|A| \leq |B|$.

Corollary 17.6. If $A \subseteq B$ and B is countable, then A is countable.

Theorem 17.7. For any finite alphabet Σ , Σ^* is countable.

Proof. Notice for every $n = 0, 1, \dots$ that Σ^n is finite; for any string $s \in \Sigma^n$ there are $|\Sigma|$ choices for each character of s , so that $|\Sigma^n| = |\Sigma|^n$ is finite. Now recall that $\Sigma^* = \bigcup_{n=0}^{\infty} \Sigma^n$ is a countable union of finite sets, so Σ^* is countable as a result. ■

Corollary 17.8. Any specific language $L \subseteq \Sigma^*$ is countable.

In particular, the language $\{\langle M \rangle \mid M \text{ is a TM}\}$ is countable, thus the set of TMs $\{M \mid M \text{ is a TM}\}$. Consequently, the set of TM-recognisable languages $\{L \mid L \text{ is TM-recognisable}\}$ is countable.

As we will see, the set of all languages $\{L \mid L \subseteq \Sigma^*\}$ is not countable.

17.1. Diagonalisation

One famous method of proof is called proof by diagonalisation. The idea behind this technique is to abstractly describe a tabular collection of objects, and to manipulate the “diagonal” of this table to create a new object that is not in the table.

Theorem 17.9. The reals in $(0, 1)$ are uncountable.

Proof (Cantor’s). Assume towards contradiction that we can write the set $\{r_1, r_2, r_3, \dots\}$ in such a way that includes all the reals in $(0, 1)$. Now consider the number $r = 0.d_1d_2d_3\dots$, where $\forall i, d_i \in \{1, \dots, 8\}$ such that $d_i \neq i$ th digit of r_i . Now $r \neq r_i$ for any i (that is, it is not on the set), because r, r_i differ in at least their i th digit. But r is a real number in $(0, 1)$, so it should be in the set. Contradiction. ■

Theorem 17.10. The set of all languages, i.e. $\mathcal{P}(\Sigma^*) = \{L \mid L \subseteq \Sigma^*\}$, is not countable.

Proof. Assume towards contradiction that it was, that is, that we can write $\mathcal{P}(\Sigma^*) = \{L_1, L_2, \dots\}$. Let $\Sigma^* = \{s_1, s_2, \dots\}$, and define the diagonal language $L_d = \{s_i \in \Sigma^* \mid s_i \notin L_i\}$. This language is not on the list, since for any i , $L \neq L_i$, as exactly one of them contains s_i and the other one doesn't. Contradiction. ■

Corollary 17.11. *There exists uncountably many languages that are not TM-recognisable.*

Proof. We already stated that there are countably many TM-recognisable languages, and just proved that there are uncountably many languages. The corollary follows immediately. ■

Challenge

Prove by diagonalisation that the power set of *any* infinite countable set is not countable.

18. Undecidable & unrecognisable languages; Reductions (21/11)

Theorem 18.1. $A_{TM} = \{\langle M, w \rangle \mid M \text{ is a TM, and } M \text{ accepts } w\}$ is not decidable.

Proof. We complete the proof by diagonalisation. Assume towards contradiction that A_{TM} is decidable. That is, there exists a TM ACC which is a decider for A_{TM} :

ACC = “On any input $\langle M, w \rangle$:

1. if M accepts w , accept
2. otherwise, reject.”

The plan is to construct an input $\langle M, w \rangle$ for which the construction ACC is not possible, leading to a contradiction. The idea is similar in spirit to the “barber paradox”, and the input will be $\langle M, w \rangle = \langle B, \langle B \rangle \rangle$. Consider the following algorithm B :

B = “On input $\langle M \rangle$ where M is a TM:

1. run ACC($\langle M, \langle M \rangle \rangle$)
2. if it accepts, reject
3. if it rejects, accept.”

Now what happens when we run B on $\langle B \rangle$?

- If B accepts $\langle B \rangle$, then ACC($\langle B, \langle B \rangle \rangle$) accepts. Thus B rejects $\langle B \rangle$ —contradiction.
- If B rejects $\langle B \rangle$, then ACC($\langle B, \langle B \rangle \rangle$) rejects. Thus B accepts $\langle B \rangle$ —contradiction.
- It is not possible for B to run forever on $\langle B \rangle$, since ACC is a decider, and hence B is a decider.

We get a contradiction in all three cases, so no such decider ACC for A_{TM} can exist. ■

Theorem 18.2. $\overline{A_{TM}}$ is not recognisable.

Proof. We have already seen (Theorem 15.2) that a language L is decidable if and only if L and $\overline{L}$ are both recognisable. We have also seen (Theorems 16.1 and 18.1) that A_{TM} is recognisable but not decidable. It immediately follows that $\overline{A_{\text{TM}}}$ is not recognisable. (If it were, then A_{TM} and $\overline{A_{\text{TM}}}$ would both be recognisable, so A_{TM} would be decidable—contradiction. ■

Corollary 18.3. *The class of recognisable languages is not closed under complement.*

Proof. A_{TM} is recognisable, but $\overline{A_{\text{TM}}}$ is not. ■

Theorem 18.4 (Halting problem). $\text{HALT}_{\text{TM}} = \{\langle M, w \rangle \mid M \text{ is a TM and } M \text{ halts on } w\}$ is undecidable.

Proof 1. Assume towards contradiction that HALT_{TM} is decidable, i.e. that there exists a TM A that is a decider for HALT_{TM} . Now consider a TM B defined as follows: it takes an input $\langle M \rangle$, runs A on input $\langle M, \langle M \rangle \rangle$, and halts if and only if A rejects. Consider what happens when B receives $\langle B \rangle$ as input. It runs A on $\langle B, \langle B \rangle \rangle$. There are two cases:

- A accepts. This means B on input $\langle B \rangle$ halts. But by the definition of B , if A accepts we will enter an infinite loop, and so B on input $\langle B \rangle$ does not halt after all.
- A rejects. This means B on input $\langle B \rangle$ doesn't halt. But by the definition of B , if A rejects we will terminate, and so B on input $\langle B \rangle$ halts after all.

In both cases, we derive a contradiction. Thus HALT_{TM} is undecidable. ■

Proof 2. The plan for this proof is to use a technique called the reduction. In particular, we will show that if HALT_{TM} is decidable, then so is A_{TM} (thus reaching a contradiction).

Assume towards contradiction that HALT_{TM} is decidable. So there exists a TM R that decides HALT_{TM} described by

- $R =$ “On input $\langle M, w \rangle$:
1. if M accepts w or M rejects w
 2. accept
 3. if M runs forever on w
 4. reject.”

We will use R to construct a decider S for A_{TM} :

- $S =$ “On input $\langle M, w \rangle$ where M is a TM:
1. run $R(\langle M, w \rangle)$
 2. if it rejects, reject
 3. if it accepts,
 4. run M on w
 5. if it accepts, accept
 6. if it rejects, reject.”

Note that S is a decider, since R is a decider, and line 4 is executed only if M halts on w (otherwise we reject earlier in line 2).

Furthermore, $\langle M, w \rangle \in A_{\text{TM}} \implies M \text{ accepts } w \implies R \text{ accepts (line 2)} \implies S \text{ accepts (line 5)}$. On the other hand, $\langle M, w \rangle \notin A_{\text{TM}} \implies$ either (a) M rejects w or (b) M runs forever on $w \implies$ (a) R accepts (line 1) and S rejects (line 6) or (b) R rejects (line 1) and S rejects (line 2). Finally, if $x \notin \langle M, w \rangle$ (i.e. bad encoding), x is rejected in line 0 and $x \notin A_{\text{TM}}$.

We have just proved that if HALT_{TM} was decidable, then A_{TM} is decidable. But we saw in Theorem 18.1 that A_{TM} is undecidable. Contradiction— HALT_{TM} is undecidable. ■

18.1. Reductions

Definition 18.5 (Turing-reducibility). *We say a language A is Turing-reducible to a language B , denoted $A \leq_T B$, if given a decider/subroutine/oracle deciding B , there exists a decider for A .*

Note. We proved above that $A_{\text{TM}} \leq_T \text{HALT}_{\text{TM}}$. That is, given a decider R for HALT_{TM} , we constructed a decider S for A_{TM} .

Theorem 18.6. *If $A \leq_T B$ and B is decidable, then A is decidable. Equivalently, if $A \leq_T B$ and A is undecidable, then B is also undecidable.*

Proof. Follows immediately from the definition of Turing-reducibility. ■

Answer to student question. The following statements are true—note the difference between the languages!

- The language $\{\langle M \rangle \mid M \text{ is a TM and } L(M) = \emptyset\}$ is undecidable.
- The language $\{\langle M \rangle \mid M \text{ is a TM that has no transitions going into an accept state}\}$ is decidable.

Challenge

Prove that the language $\{\langle M, w \rangle \mid M \text{ rejects } w\}$ is undecidable.

19. More undecidability, Rice's theorem (28/11)

Recap. We have now learned one or two things about Turing machines:

- A TM is an “algorithm” which we can think of as computing a partial function from an alphabet to a decision, i.e. a TM computes $f : \Sigma^* \rightarrow \{\text{accept, reject}\}$.
- The language of a TM is $L(M) = \{w \in \Sigma^* \mid M \text{ accepts } w\}$.
- How to prove that a language is not decidable?
 - by diagonalisation (similar in spirit to the “barber paradox”)
 - by (Turing) reductions: to show L is undecidable, it is enough to show $A \leq_T L$ where A is undecidable.

- A typical proof that $A \leq_T B$ looks like this¹⁷:

Suppose $\mathfrak{D}$ is a decider for B . Now, construct a decider for A thus:

$A =$ “On input [A -type input]:

- $\vdots$
- $\vdots$
1. create [B -type input]
2. run $\mathfrak{D}$ on [B -type input]
3. if $\mathfrak{D}$ accepts, accept/reject
4. if $\mathfrak{D}$ rejects, accept/reject.”

Then prove that $x \in A \implies A$ accepts, and $x \notin A \implies A$ rejects.

We have now seen a number of undecidable languages: A_{TM} , HALT_{TM} , E_{TM} , EQ_{TM} , and REG_{TM} . We now add one more tool to help us in proving the undecidability of a language.

Theorem 19.1 (Rice’s theorem, informal). *Any nontrivial languages of the form described by $\{\langle M \rangle \mid M \text{ is a TM and } L(M) \text{ satisfies “something”}\}$ is undecidable.*

This is not very formal, but we can formalise this using the definition of a *property*.

Definition 19.2. *P is a property of recognisable languages if $P \subseteq \{\langle M \rangle \mid M \text{ is a TM}\}$, and for any TMs M_1, M_2 if $L(M_1) = L(M_2)$, then $\langle M_1 \rangle \in P \iff \langle M_2 \rangle \in P$.*

Definition 19.3. *We say that property P is trivial if $P = \emptyset$ or $P = \{\langle M \rangle \mid M \text{ is a TM}\}$. We say that P is nontrivial otherwise, that is, if $\exists \text{ TM } M \text{ such that } \langle M \rangle \in P \text{ and } \exists \text{ TM } M' \text{ such that } \langle M' \rangle \notin P$.*

Theorem 19.4 (Rice’s theorem, formal). *Let P be a nontrivial property of recognisable languages. Then P is not decidable.*

Proof idea. The idea is to show that $A_{\text{TM}} \leq_T P$. That is, we assume that $\mathfrak{D}$ is a decider for P , and use it to construct a decider for A_{TM} . So since A_{TM} is undecidable, we conclude that P is also undecidable.

Proof. Let $M_\emptyset$ be some TM such that $L(M_\emptyset) = \emptyset$. For example, we could have one such TM as $M_\emptyset$: “On input x , reject.”

Assume $\langle M_\emptyset \rangle \in P$ without loss of generality. (If $\langle M_\emptyset \rangle \notin P$, then simply continue the proof with $P' = \{\langle M \rangle \mid M \text{ is a TM and } \langle M \rangle \notin P\}$, and note that P is decidable $\iff P'$ is decidable.)

Since P is nontrivial, there exists a TM M_{out} , $\langle M_{\text{out}} \rangle \notin P$. Since P is a property of recognisable languages, we know that for any TM M :

- if $L(M) = \emptyset$, $\langle M \rangle \in P$

¹⁷In general, this could be more complicated, e.g. we could call $\mathfrak{D}$ more than once, or do more things. Or it could be simpler.

- if $L(M) = L(M_{\text{out}})$, $\langle M \rangle \notin P$.

Now here is a decider R for A_{TM} , assuming we have a decider $\mathfrak{D}$ for P :

$R =$ “On input $\langle M, w \rangle$ where M is a TM, w a string:

1. construct $\langle M' \rangle$ where M' is the following TM:
2. $M' =$ “On input x :
 1. run M_{out} on x
 2. if M_{out} accepts,
 3. run M on w
 4. if it accepts, accept
 5. reject.”
3. run $\mathfrak{D}(\langle M' \rangle)$
4. if it accepts, reject
5. if it rejects, accept.”

Now, R is a decider for A_{TM} , because:

$$\begin{aligned} \langle M, w \rangle \in A_{\text{TM}} &\implies M \text{ accepts } w \implies L(M') = L(M_{\text{out}}) \implies \langle M' \rangle \notin P \implies \mathfrak{D} \text{ rejects} \\ &\implies R \text{ accepts.} \end{aligned}$$

$$\begin{aligned} \langle M, w \rangle \notin A_{\text{TM}} &\implies \text{either } M \text{ rejects } w \text{ or } M \text{ runs forever on } w \implies L(M') = \emptyset \\ &\implies \langle M' \rangle \in P \implies \mathfrak{D} \text{ accepts} \implies R \text{ rejects.} \end{aligned}$$

We just proved $A_{\text{TM}} \leq_T P$. But A_{TM} is undecidable, so P cannot be decidable. ■

Example 19.1. We claim that

$$P = \{ \langle M \rangle \mid M \text{ is a TM, and every string accepted by } M \text{ starts with } 0 \}$$

is not decidable. Note that P is a property of recognisable languages, since for any M_1, M_2 , if $L(M_1) = L(M_2)$, then w is accepted by M_1 if and only if w is accepted by M_2 . Thus $\langle M_1 \rangle \in P \iff \langle M_2 \rangle \in P$.

Also, P is nontrivial. There exists M such that $\langle M \rangle \in P$, for example

- $M =$ “On input x :
1. if $x = 001$, accept
 2. else, reject.”

Note that $L(M) = \{001\}$. Also, there exists M' such that $\langle M' \rangle \notin P$, for example

- $M' =$ “On input x :
1. accept.”

Here $L(M') = \Sigma^*$. Thus we can apply Rice’s theorem— P is immediately undecidable.

When can’t we apply Rice’s theorem? Although Rice’s theorem is widely applicable, there are a number of situations where it fails, eg:

1. when the input is not a TM property. For example, we cannot apply Rice's to DFAs $\langle D \rangle$, graphs $\langle G \rangle$, pairs of TMs $\langle M_1, M_2 \rangle$, etc.
2. When it is a TM property but depends on an implementation. For example, we cannot apply Rice's to $\{\langle M \rangle \mid M \text{ is a TM, } L(M) = \emptyset, \text{ and } M \text{ has at least 17 states}\}$, nor can we apply it to $\{\langle M \rangle \mid M \text{ is a TM that always moves right}\}$.

Note. If $A \leq_T B$ and A is not recognisable, can we conclude that B is not recognisable? *Not necessarily!* However, we will define a special type of reduction, called a mapping reduction $A \leq_m B$ that can be used in this way.

The proof of Rice's theorem above showed that $A_{TM} \leq_T P$. If you look closely, the proof can actually be rephrased as showing that

$$\begin{aligned} \overline{A_{TM}} &\leq_m P \text{ assuming that } \langle M_\emptyset \rangle \in P \\ \overline{A_{TM}} &\leq_m \overline{P} \text{ assuming that } \langle M_\emptyset \rangle \notin P. \end{aligned}$$

Thus the same proof in fact shows that

Theorem 19.5 (Refined Rice's theorem). *Let P be a nontrivial property of recognisable languages. Then,*

$$\begin{aligned} \langle M_\emptyset \rangle \in P &\implies P \text{ is not recognisable, and} \\ \langle M_\emptyset \rangle \notin P &\implies \overline{P} \text{ is not recognisable,} \end{aligned}$$

where $M_\emptyset$ is a TM such that $L(M_\emptyset) = \emptyset$.

Corollary 19.6. *The following languages are not recognisable. The proof is left as an exercise.*

- $E_{TM} = \{\langle M \rangle \mid M \text{ is a TM and } L(M) = \emptyset\}$.
- $REG_{TM} = \{\langle M \rangle \mid M \text{ is a TM and } L(M) \text{ is regular}\}$.

20. Mapping reductions, computation histories (30/11)

20.1. Mapping reductions

Definition 20.1 (Mapping-reducibility). *A language A is mapping-reducible to a language B , written $A \leq_m B$ if there exists a computable function $f : \Sigma^* \rightarrow \Sigma^*$ such that $x \in A \iff f(x) \in B$ for all strings x .*

Example 20.1. Let

$$\begin{aligned} A &= \{n \mid n \text{ is a nonnegative even integer}\} \\ B &= \{m \mid m \text{ is a nonnegative odd integer}\}. \end{aligned}$$

Note that $A \leq_m B$ via the function f defined as

$f =$ "On input x represented in binary:

1. output $x + 1$.”

f is computable— $x \in A \iff f(x) \in B$.

For the following theorem, we define $\text{ALL}_{\text{TM}} = \{\langle M \rangle \mid M \text{ is a TM and } L(M) = \Sigma^*\}$.

Theorem 20.2. $A_{\text{TM}} \leq_m \text{ALL}_{\text{TM}}$.

Proof. Consider the following f :

$f =$ “On input $\langle M, w \rangle$:

1. construct $\langle M' \rangle$ where $M' =$ “On input x , run M on w and output same.”

2. output $\langle M' \rangle$.”

f is computable: it is easy to modify $\langle M, w \rangle$ into $\langle M' \rangle$.

$$\begin{aligned} \langle M, w \rangle \in A_{\text{TM}} &\implies M \text{ accepts } w \implies M' \text{ accepts every input } x \text{ (i.e. } L(M) = \Sigma^*) \\ &\implies \langle M' \rangle \in \text{ALL}_{\text{TM}}. \end{aligned}$$

$$\begin{aligned} \langle M, w \rangle \notin A_{\text{TM}} &\implies M \text{ does not accept } w \implies M' \text{ does not accept any input } x \text{ (i.e. } L(M) = \emptyset) \\ &\implies \langle M' \rangle \notin \text{ALL}_{\text{TM}}. \end{aligned}$$

The proof is done. ■

Comments:

- Technically, f also needs to address inputs that are not correctly formed (that is, $x \neq \langle M, w \rangle$). We will ignore it, but it is typically easy to check (e.g. here, we could have $f(x) = \varepsilon$ or any other string not in ALL_{TM} , when $x \neq \langle M, w \rangle$)
- There are many other f ’s possible. Additionally, there is an alternative reduction showing $A_{\text{TM}} \leq_m \text{ALL}_{\text{TM}}$. Check that $f'(\langle M, w \rangle) = \langle M' \rangle$ where

$M' =$ “On input x :

 1. if $x \neq \varepsilon$, accept
 2. else
 3. run M on w and output same.”

Theorem 20.3. If $A \leq_m B$, then $A \leq_T B$. That is, we can view mapping reductions as a special case of Turing reductions.

Proof. Assume $A \leq_m B$ via f , and that $\mathfrak{D}$ is a decider for B . Now we construct a decider R for A .

$R =$ “On input x :

1. compute $y = f(x)$
2. run $\mathfrak{D}(y)$, and output same.”

R is implementable, since f is computable and we are given $\mathfrak{D}$ a decider.

$$x \in A \implies y = f(x) \in B \implies \mathfrak{D}(y) \text{ accepts} \implies R \text{ accepts}$$

$$x \notin A \implies y = f(x) \notin B \implies \mathfrak{D}(y) \text{ rejects} \implies R \text{ rejects.}$$

The proof is done. ■

Theorem 20.4. *The following are true:*

1. $A \leq_T B \iff \overline{A} \leq_T B \iff A \leq_T \overline{B} \iff \overline{A} \leq_T \overline{B}.$
2. $A \leq_m B \iff \overline{A} \leq_m \overline{B}.$

Proof. The proof of the first part is left as an exercise, and uses the fact that L is decidable iff $\overline{L}$ is decidable. We present the proof of the second part. Suppose $A \leq_m B$ via f . The same f is also a mapping reduction from $\overline{A}$ to $\overline{B}$:

$$x \in \overline{A} \iff x \notin A \iff f(x) \notin B \iff f(x) \in \overline{B}.$$

Now, we already proved $A \leq_m B \implies \overline{A} \leq_m \overline{B}$. Now, we have that if the mapping given by $\overline{A} \leq_m \overline{B} \implies \overline{(\overline{A})} \leq_m \overline{(\overline{B})} \implies A \leq_m B$. ■

Theorem 20.5. *If $A \leq_m B$ and B is recognisable, then A is recognisable. Equivalently, if $A \leq_m B$ and A is not recognisable, then B is not recognisable.*

Proof. Assume $A \leq_m B$ via a mapping f , and B is recognised by some TM T . Now construct a recogniser R for A , as follows:

- $R =$ “On input x :
1. compute $y = f(x)$
 2. run $T(y)$, and output same.”

Note that R recognises A , since

$$x \in A \implies y = f(x) \in B \implies T(y) \text{ accepts} \implies R \text{ accepts,}$$

$$x \notin A \implies y = f(x) \notin B \implies T(y) \text{ rejects or runs forever} \implies R \text{ rejects or runs forever.}$$

The proof is done. ■

Theorem 20.6. $\text{ALL}_{\text{TM}} = \{\langle M \rangle \mid M \text{ is a TM, and } L(M) = \Sigma^*\}$ is neither recognisable nor co-recognisable.

Plan: Recall that we proved $\overline{\text{ALL}_{\text{TM}}}$ is not recognisable. Now we want to show:

1. $\overline{\text{ALL}_{\text{TM}}} \leq_m \text{ALL}_{\text{TM}}$, which implies that ALL_{TM} is not recognisable;

2. $\overline{A_{\text{TM}}} \leq_m \overline{\text{ALL}_{\text{TM}}}$, which implies that $\overline{\text{ALL}_{\text{TM}}}$ is not recognisable.

Proof. We start with the second part, and attempt to show that $\overline{A_{\text{TM}}} \leq_m \overline{\text{ALL}_{\text{TM}}}$. But we are immediately done, since by Theorem 20.4 this is equivalent to $A_{\text{TM}} \leq_m \text{ALL}_{\text{TM}}$, which we already showed to be true in Theorem 20.2.

Now we turn to the first part. We want to prove $\overline{A_{\text{TM}}} \leq_m \text{ALL}_{\text{TM}}$, or equivalently, $A_{\text{TM}} \leq_m \overline{\text{ALL}_{\text{TM}}}$. Now we need to construct a computable function f such that $f(\langle M, w \rangle) = \langle M' \rangle$, such that M accepts w if and only if M' does not accept all possible strings.

- $f =$ “On input $\langle M, w \rangle$:
1. construct $\langle M' \rangle$ where M' is described below:
 2. $M' =$ “On input x :
 1. run M on w for $|x|$ steps
 2. if within these steps M accepted, reject
 3. else if M rejected w , or hasn't halted yet, accept.”
 3. output $\langle M' \rangle$.”

f is computable:

$$\begin{aligned} \langle M, w \rangle \in A_{\text{TM}} &\implies M \text{ accepts } w \text{ within some number of steps } k \\ &\implies M' \text{ rejects all input of length exceeding } k \implies \langle M' \rangle \notin \text{ALL}_{\text{TM}}. \end{aligned}$$

$$\begin{aligned} \langle M, w \rangle \notin A_{\text{TM}} &\implies M \text{ rejects } w \text{ or runs forever on } w \\ &\implies M' \text{ accepts all inputs} \implies \langle M' \rangle \in \text{ALL}_{\text{TM}}. \end{aligned}$$

The proof is done. ■

20.2. Computation histories

So far, all of our reductions have been about languages concerning decision problems about TMs. Computation histories allow us to prove undecidability of languages L even for L not involving general TMs, but still using the reduction from A_{TM} .

Definition 20.7 (Computation histories). *Let M be a TM and w an input string. An accepting computation history for M on w is a sequence of configurations $C_1, C_2, \dots, C_k$, for $k \geq 1$ so that the following properties hold:*

1. C_1 is the starting configuration of M on w ,
2. each C_i yields C_{i+1} legally according to the rules of M ,
3. C_k is an accepting configuration.

A rejecting configuration for M on w is defined similarly, except that C_k is a rejecting configuration.

Note. The following are true:

- Computation histories are always finite sequences.

- If M runs forever on w , then there is no accepting or rejecting configuration for M on w .
- Deterministic TMs have at most one computation history on a given input. Nondeterministic TMs may have multiple computation histories on a given input, corresponding to various computation branches.

Theorem 20.8. $\text{ALL}_{\text{CFG}} = \{\langle G \rangle \mid G \text{ is a CFG and } L(G) = \Sigma^*\}$ is undecidable.

Plan. We proceed by contradiction. Assuming that ALL_{CFG} is decidable, we can devise the following decision procedure for A_{TM} :

- For a TM M and input string w , construct a CFG G that generates all strings iff M does not accept w .
- If M does accept w , then G does not generate some particular string. This will correspond to the accepting computation history for M on w .

Proof. Suppose that a TM R decides ALL_{CFG} . Now construct a TM S that decides A_{TM} as follows:

- $S =$ “On input $\langle M, w \rangle$ where M is a TM, w a string:
1. construct a CFG G from M and w
 2. run R on input $\langle G \rangle$
 3. if R rejects, accept
 4. if R accepts, reject.”

If R accepts $\langle G \rangle$, then $L(G) = \Sigma^*$. Thus M has no accepting computation on w , and M does not accept w . Consequently S rejects $\langle M, w \rangle$.

If R rejects $\langle G \rangle$, then $L(G) \neq \Sigma^*$. Since the only string that G cannot generate is an accepting computation history for M on w , we have that M accepts w . Consequently S accepts $\langle M, w \rangle$.

This is a contradiction, so R cannot exist, therefore ALL_{CFG} is undecidable. ■

Challenges

1. Using a mapping reduction from the language ALL_{CFG} , show that the language given by $\text{EQ}_{\text{CFG}} = \{\langle G, H \rangle \mid G \text{ and } H \text{ are CFGS and } L(G) = L(H)\}$ is undecidable.
2. Consider the following language:

$$L_1 = \{\langle M \rangle \mid M \text{ is a TM that halts and accepts with '42' on the tape on at least one input}\}.$$

Show that L_1 is undecidable.

3. Consider the language

$$\text{EQ}_{\text{TM}} = \{\langle M_1, M_2, w \rangle \mid M_1, M_2 \text{ are both Turing machines, } w \text{ is a string, and } M_1, M_2 \text{ either both accept or both reject on } w\}.$$

Show that this language is undecidable by reducing from the halting problem.

21. (Brief) Intro. to Complexity, I (05/12)

Definition 21.1 (Running time). Given a TM M , the running time of M is a function $f : \mathbb{N} \rightarrow \mathbb{N}$ where $f(n)$ is the maximum number of steps that M takes on any input of length n . If $f(n)$ is the running time of M , we say that M is an $f(n)$ -time TM.

Definition 21.2 (Big-O). Let f and g be functions $f, g : \mathbb{N} \rightarrow \mathbb{R}^+$. We say that $f(n) = \mathcal{O}(g(n))$ if there exists constants c, n_0 such that $\forall n \geq n_0$, we have $f(n) \leq c \cdot g(n)$.

Definition 21.3 (Small-O). Let f and g be functions $f, g : \mathbb{N} \rightarrow \mathbb{R}^+$. We say that $f(n) = o(g(n))$ if for any real $c > 0$, a number n_0 exists where $f(n) < cg(n)$ for all $n \geq n_0$.

Definition 21.4 (Time complexity class). Let $t : \mathbb{N} \rightarrow \mathbb{R}^+$ be a function. We define the (deterministic) time complexity class $\text{TIME}(t(n))$ to be the collection of all languages that are decidable by an $\mathcal{O}(t(n))$ -time TM.

Note. If $t(n) = \mathcal{O}(t'(n))$, then $\text{TIME}(t(n)) = \text{TIME}(t'(n))$.

Example 21.1. If $a_0, a_1, \dots, a_k$ are constants, then $a_0 + a_1n + \dots + a_kn^k = \mathcal{O}(n^k)$.

Example 21.2. $a \log_b(n) = \mathcal{O}(\log_2 n)$.

21.1. P

Definition 21.5. The class of poly-time computable languages is

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

That is, a language $L \in \mathbf{P}$ iff there exists a constant k and TM M that decides L and runs in time $\mathcal{O}(n^k)$. Moving forward, we will identify $\mathbf{P}$ with the class of “efficiently computable” languages.

Note that the definition of $\text{TIME}(t(n))$ is sensitive to details of the model of computation (eg. TM, multitape TM, nondeterministic TM, etc). However, the definition of $\mathbf{P}$ is not sensitive to these details.

Theorem 21.6 (Strong Church-Turing thesis). Every reasonable notion of algorithm or model of computation can be simulated by a TM with polynomial overhead.

Theorem 21.6 is more controversial/less universally believed compared to be true for all models we’ve seen so far and for all programming languages, with the exception of nondeterministic TMs and quantum computers.

21.2. Some problems in P

We will think about languages and problems as meaning the same thing.

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

Theorem 21.7. $\text{PATH} \in \text{P}$.

Proof sketch. Run breath-first search on the directed graph. Note in particular that there is an exponential number of simple paths from s to t in the worst case ($(n-2)!$ paths in the complete directed graph with n vertices), but we do not examine them all in a breadth-first search. The BFS takes a shortcut to zeroing on one particular path in poly-time. ■

2. Define $\text{REL-PRIME} = \{\langle x, y \rangle \mid x \text{ and } y \text{ are relatively prime}\}$.

Theorem 21.8. $\text{REL-PRIME} \in \text{P}$.

Proof. The Euclidean algorithm E is as follows:

$E =$ “On input $\langle x, y \rangle$, where x and y are natural numbers in binary:

1. until $y = 0$, do:
2. assign $x \leftarrow x \bmod y$
3. swap x and y
4. output x .”

The following algorithm R solves REL-PRIME , using E as a subroutine.

$R =$ “On input $\langle x, y \rangle$, where x and y are natural numbers in binary:

1. run $E(\langle x, y \rangle)$
2. if the result is 1, accept
3. else, reject.”

Each iteration of the loop of E halves the value of x (can you see why?). Therefore, at most $\log x \leq |\langle x, y \rangle|$ iterations of the loop execute, and each iteration requires a constant number of poly-time computable arithmetic operations, whence R operates in poly-time. ■

3. Define $\text{PRIMES} = \{\text{bin}(n) \mid n \text{ is prime}\}$, where $\text{bin}(n)$ is the binary representation of the number n .

Theorem 21.9. $\text{PRIMES} \in \text{P}$.

Proof. See AKS algorithm. ■

4. Every $\text{CFL} \in \text{P}$. Every regular language $\in \text{P}$.

21.3. NP

Definition 21.10. A nondeterministic TM runs in time $t(n)$ if for every input x of length n , every possible computation of N on x terminates with at most $t(n)$ steps.

Definition 21.11 (Time complexity class). Let $t : \mathbb{N} \rightarrow \mathbb{R}^+$ be a function. We define the non-deterministic time complexity class $\text{NTIME}(t(n))$ to be the collection of all languages that are decidable by an $\mathcal{O}(t(n))$ -time TM.

Definition 21.12. The class of non-deterministic poly-time computable languages is

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

21.4. Some problems in NP

1. Define $\text{COMPOSITE} = \{\langle N \rangle \mid N \text{ is a (binary representation) of a composite integer}\}$.

Theorem 21.13. $\text{COMPOSITE} \in \text{NP}$.

Proof. Here is an NTM that decides COMPOSITE in poly-time:

$\mathfrak{N}$ = “On input $\langle N \rangle$:

1. non-deterministically choose $1 < d < N$
2. check if $d \mid N$
3. if so, output yes
4. else, output no.”

We can check that $\mathfrak{N}$ indeed runs in poly-time (exercise). ■

2. Define $\text{CLIQUE} = \{\langle G, k \rangle \mid G \text{ is an undirected graph with } k\text{-clique}\}$.

Theorem 21.14. $\text{CLIQUE} \in \text{NP}$.

Proof. The following algorithm *verifies* CLIQUE in poly-time:

$\mathfrak{D}$ = “On input $\langle \langle G, k \rangle, C \rangle$:

1. test whether C is a set of k nodes from G
2. test whether every pair of nodes in C is connected by an edge in G
3. accept if and only if both tests pass.”

The proof is done. ■

We will associate NP with the class of “efficiently verifiable” problems.

22. (Brief) Intro. to Complexity, II (07/12)

Definition 22.1 (Verifier). A verifier for a language A is a TM V such that V gets as input a pair (x, c) and outputs accept/reject where $A = \{x \mid V(x, c) \text{ accepts for some } c\}$.

A poly-time verifier for a language A is a verifier V that runs in time $\mathcal{O}(|x|^k)$ for some k . A language A is poly-time verifiable if there exists a poly-time verifier for A .

Theorem 22.2. $\text{NP} = \{L \mid L \text{ is poly-time verifiable}\}$.

Proof. Omitted. ■

Thus, to prove $L \in \text{NP}$, we can either

- show a nondeterministic TM that runs in poly-time and decides L , or
- show a poly-time verifier for L .

Example 22.1. Define $\text{HAM-CYCLE} = \left\{ \langle G \rangle \mid \begin{array}{l} G \text{ is an undirected graph and} \\ G \text{ has a Hamiltonian cycle} \end{array} \right\}$.

We claim that $\text{HAM-CYCLE} \in \text{NP}$. Here is a poly-time verifier for it:

$V =$ “On input $\langle G, c \rangle$:

1. Check that $\langle G \rangle$ is a valid encoding of an undirected graph $G = (V, E)$.
2. Check that c constitutes a Hamiltonian cycle:
3. Set $n = |V|$, the number of vertices
4. Check that $c = (p_1, \dots, p_n)$ is a permutation on $\{1, \dots, n\}$
5. Check that $(p_i, p_{i+1}) \in E$ and $(p_n, p_1) \in E$ for all $i = 1, \dots, n-1$
6. If all checks succeed, accept
7. Else, reject.”

We can prove that V runs in poly-time in $|G|$. Furthermore, $G \in \text{HAM-CYCLE} \iff \exists c$ such that $V(x, c)$ accepts.

Alternatively, we could have constructed a non-deterministic TM:

$N =$ “On input $\langle G \rangle$, where $G = (V, E)$ is an undirected graph:

1. Let $n = |V|$
2. Non-deterministically choose $(p_1, \dots, p_n)$ a permutation on $\{1, \dots, n\}$
3. Check if this constitute a Hamiltonian cycle
4. If so, accept. Else, reject.”

This N decides HAM-CYCLE in non-deterministic poly-time. (HAM-CYCLE also belongs to a class of languages called **NP-complete** languages, which we will see later.)

Example 22.2. Define $\text{SUBSET-SUM} = \left\{ \langle S, t \rangle \mid \begin{array}{l} S = \{x_1, \dots, x_k\} \text{ is a collection of} \\ \text{integers and } \exists S' \subseteq S \text{ with } t = \sum_{y \in S'} y \end{array} \right\}$.

For example, $\langle \{4, 4, 11, 16, 21, 27\}, 29 \rangle \in \text{SUBSET-SUM}$ because we know $4 + 4 + 21 = 29$, but $\langle \{4, 11, 16\}, 13 \rangle \notin \text{SUBSET-SUM}$. We claim that $\text{SUBSET-SUM} \in \text{NP}$. Here is a poly-time verifier for SUBSET-SUM :

$\mathcal{V} =$ “On input $\langle \langle S, t \rangle, S' \rangle$:

1. Test whether S' is a collection of integers that sum to t .
2. Test whether S contains all the integers in S' , i.e. whether $S' \subseteq S$.
3. Accept if and only if both tests pass.”

We can check that V runs in polynomial time—left as an exercise.

Note that the complements of $\overline{\text{CLIQUE}}$ and $\overline{\text{SUBSET-SUM}}$, are not obviously members of **NP** (and are believed not to be). The class **coNP** is the class of languages whose complements are in **NP**, so $\overline{\text{CLIQUE}}, \overline{\text{SUBSET-SUM}} \in \text{coNP}$. It is not known whether $\text{coNP} = \text{NP}$, but this is believed to be

false. Proving this is at least as difficult as proving $P \neq NP$, since if $P = coP$ (which is true) and $coNP \neq NP$, then $P \neq NP$.

22.1. P vs NP

In summary:

- P is the class of languages for which membership can be *decided* quickly.
- NP is the class of languages for which membership can be *verified* quickly.

Theorem 22.3. $P \subseteq NP$.

Proof. Omitted, but clear from the definitions. ■

Conjecture 22.3. $NP \subseteq P$.¹⁸

The best-known method for solving NP problems in general is a brute-force search over all possible witnesses, giving each as input to the verifier. Since we require that each witness for a string of length n has length at least $p(n)$ for some polynomial p , there are at most $2^{p(n)}$ potential witnesses, which implies that every NP problem can be solved in *exponential* time by searching through all possible witnesses; that is,

$$NP \subseteq EXP = \bigcup_{k=1}^{\infty} TIME\left(2^{O(n^k)}\right).$$

We don't know if $NP \neq EXP$. It is known that $P \subseteq NP \subseteq EXP$, and since $P \subsetneq EXP$ (we can prove), it is known that at least one of the inclusions $P \subseteq NP$ or $NP \subseteq EXP$ is proper, although we don't know which one. It is suspected that they both are; i.e. that $P \subsetneq NP \subsetneq EXP$.

We distinguish EXP from *linear-exponential time*:

$$E = \bigcup_{k=1}^{\infty} TIME\left(2^{O(kn)}\right).$$

Note that $E \subsetneq EXP$ (we can prove). It is not known whether $NP \subseteq E$ or whether $E \subseteq NP$, although oddly enough, it is known that $NP \neq E$. Probably, neither class contains the other.

22.2. NP-Completeness

Definition 22.4. A language A is *poly-time reducible* to a language B , written as $A \leq_p B$, if there is a poly-time computable function $f : \Sigma^* \rightarrow \Sigma^*$ such that $x \in A \iff f(x) \in B$.

Definition 22.5 (NP-Hardness). A language L is *NP-hard* if $\forall A \in NP, A \leq_p L$.

¹⁸This is the biggest open problem in CS.

Definition 22.6 (NP-Completeness). *A language L is NP-complete if (1) $L \in \text{NP}$, (2) L is NP-hard.*

Corollary 22.7. *If $P \neq \text{NP}$, then no NP-complete problem is in P . Equivalently, if there is an NP-complete problem in P , then $P = \text{NP}$.*

Thus, to prove L is NP-complete, it suffices to:

- prove $L \in \text{NP}$,
- prove $A \leq_p L$ for some A that is already known to be NP-hard.

Definition 22.8 (Boolean formula). *A Boolean formula is an expression involving Boolean variables and the three operations $\vee$, $\wedge$, and $\neg$.*

Example 22.4. $\phi_1 = (x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee \overline{x_3} \vee \overline{x_1}) \wedge (x_3 \vee x_4)$ is a Boolean formula. In fact it is in conjunctive normal form (CNF).

Definition 22.9. *A Boolean formula is satisfiable if some assignment w of 0's and 1's to its variables causes the formula to have the value 1.*

Example 22.5. Recall the formula in Example 22.4. ϕ_1 is satisfiable because assigning $x_1 = 1, x_2 = 1, x_3 = 0$ and $x_4 = 1$ causes ϕ_1 to evaluate to 1, written as $\phi_1(1101) = 1$.

Example 22.6. $\phi_2 = (\overline{x} \vee y) \wedge (z \wedge \overline{z}) \wedge (y \vee x)$ is not satisfiable.

We define the satisfiability problem as

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

Theorem 22.10. $\text{SAT} \in \text{NP}$.

Proof. Easy—left as exercise. ■

Theorem 22.11 (Cook–Levin Theorem). *SAT is NP-complete.*

Very terse proof sketch. Given any $A \in \text{NP}$, there is a nondeterministic Turing machine N deciding A in nondeterministic polynomial time. We can show a poly-computable function f such that $f(x) = \phi_{N,x}$, where $\phi_{N,x}$ is a Boolean formula capturing an accepting computation of N on x . This can be expressed by a poly-sized formula $\phi_{N,x}$ such that $\phi_{N,x}$ is satisfiable if and only if there exists an accepting computation of N on x . Thus $A \leq_p \text{SAT}$ for all $A \in \text{NP}$, so that SAT is NP-hard. With this information and with Theorem 22.10, we have that Sat is NP-complete. ■

22.3. Challenge

Show that SUDOKU is NP-complete, where

$\text{SUDOKU} := \{ \langle P \rangle \mid P \text{ is a } n^2 \times n^2 \text{ grid partially filled with integers from 1 to } n^2$
such that there exists a solution}.

Appendix

A. Minimisation of DFAs

DFAs can be efficiently minimised in poly-time. That is, for each DFA, we can find an equivalent DFA that has as few states as any DFA accepting the same language. Additionally, up to the permutations of the constituent states, this minimum-state DFA is unique for a given language. An informal algorithm for doing this is as follows:

1. First, eliminate any state that cannot be reached from the start state.
2. Then, partition the remaining states into blocks, so that all states in the same block are equivalent, and no pair of states from different blocks are equivalent. Theorem A.2 below shows that we can always make such a partition.

To precisely describe the formal procedure for minimising a DFA, we state the following two theorems without proof, the second being a consequence of the first.

Theorem A.1. *The equivalence of states is transitive. That is, if in some DFA $D = (Q, \Sigma, \delta, q_0, F)$ we find that states q_i and q_j are equivalent, and we also find that states q_j and q_k are equivalent, then it must be that states q_i and q_k are also equivalent.*

Theorem A.2. *If we create for each state q of a DFA a block consisting of q and all the states equivalent to q , then the different blocks of states form a partition of the set of states. That is, each state is in exactly one block. All members of a block are equivalent, and no pair of states chosen from different blocks are equivalent.*

Now we can succinctly state the algorithm for minimising a DFA $D = (Q, \Sigma, \delta, q_0, F)$. First we describe the table-filling algorithm, used for computing all pairs of distinguishable states of a DFA.

ALGORITHM: The Table-Filling Algorithm.

- *Input:* A DFA $D = (Q, \Sigma, \delta, q_0, F)$.
- *Output:* a table T of all pairs of distinguishable states.
- *Method:*
 - for all states p and q do
 - if p is final and q is non-final
 - add $\{p, q\}$ to T
 - for all states p and q do
 - for all input symbols a do
 - if $\delta(p, a)$ and $\delta(q, a)$ are in T then
 - add $\{p, q\}$ to T
 - until no more pairs can be added to T

If two states p and q are not distinguishable by the table-filling algorithm, then p and q are

equivalent.

We can use this algorithm as a subroutine to minimise the number of states in a DFA via the following algorithm.

ALGORITHM: The DFA Minimisation Algorithm.

- *Input*: a DFA $D = (Q_D, \Sigma, \delta_D, q_D, F_D)$.
- *Output*: an equivalent minimum-state DFA $D^* = (Q_{D^*}, \Sigma, \delta_{D^*}, q_{D^*}, F_{D^*})$
- *Method*: Up to the number of states in D , do the following:
 1. Eliminate any state that cannot be reached from the start state.
 2. Compute the sets of all equivalent states.
 3. Partition the states into blocks so that
 - all states in the same block are equivalent and
 - no pair of states from different blocks are equivalent.
 4. Construct the minimum-state DFA D^* as follows:
 - a) Q_{D^*} is the set of blocks of equivalent states.
 - b) If R and S are blocks containing the states p and q of D , respectively, then $\delta_{D^*}(R, a) = S$ if $\delta_D(p, a) = q$.
 - c) q_{D^*} is the block containing q_D .
 - d) A state S is in F_{D^*} if S contains a state in F_D .

Note that we can also minimise regular expressions—i.e. given a regular expression of length n , go through all regular expressions of length $1, 2, 3, \dots, n$, and check equivalence until we find the shortest equivalent one—but we don't know how to do it efficiently. This is a PSPACE-complete problem, believed to be hard.

B. The Pumping Lemma as an Adversarial Game

A theorem whose statement involves several alternations of “for-all” and “there-exists” quantifiers can be thought of as a game between two players. The pumping lemma is one such theorem, so we may design a game implementing the pumping lemma.

1. Player 1 picks the language L to be proved non-regular.
2. Player 2 picks p , the pumping length, but doesn't reveal to player 1 what p is; player 1 must devise a play for all possible p 's.
3. Player 1 picks w , which may depend on p and which must have length at least n .
4. Player 2 divides w into x, y , and z , obeying the constraints that are stipulated in the pumping lemma: $y \neq \varepsilon$ and $|xy| \leq p$. Again, player 2 does not have to tell player 1 what x, y , and z are, although they must obey the constraints.

5. Player 1 “wins” by picking i , which may be a function of p, x, y , and z , such that $xy^iz \notin L$.

It is an instructive exercise to show what exactly goes wrong in this adversarial game when L is a regular language.

C. The Myhill-Nerode Theorem

The Myhill-Nerode theorem is a fundamental result that provides a characterisation of regular languages, and hence can be used to prove whether or not a language L is regular. The key concept to the Myhill-Nerode theorem is the distinguishing extension:

Definition C.1 (Distinguishing extension). *Let $L \subseteq \Sigma^*$ be any language over the alphabet Σ . For $x, y \in \Sigma^*$, we call $z \in \Sigma^*$ a distinguishing extension of x and y if exactly one of xz and yz are in L (where xz is the concatenation of x and z). If such a z exists, we say that x and y are distinguishable by L ; otherwise we say x and y are indistinguishable by L .*

Proposition C.2. *Let $L \subseteq \Sigma^*$ be a language. Define $\sim_L$ to be the relation on Σ^* where $x \sim_L y$ if and only if x and y are indistinguishable by L . Then $\sim_L$ is reflexive, symmetric, and transitive, i.e. $\sim_L$ is an equivalence relation on Σ^* .*

Recall that an equivalence relation $\sim$ on a set S induces *equivalence classes* which partition S into subsets of elements related to each other by $\sim$.

Remark 1. If L is regular and recognised by a DFA D , then x and y are in the same equivalence class of $\sim_L$ iff the state of D after reading x and the state of D after reading y are equivalent. Thus if D has no states which are equivalent to each other, then x and y are in the same equivalence class iff the state of D after reading x is the same as the state of D after reading y and it can be shown that D has the least amount of states possible. So let L be regular and let D be its minimal DFA with states $\{q_0, q_1, \dots, q_n\}$. We can then systematically define the equivalence classes of $\sim_L$ to be the sets $\langle i \rangle = \{w \in \Sigma^* \mid D \text{ is in state } q_i \text{ after reading } w\}$. This realisation leads to the following remark as a corollary.

Remark 2. If $x \in L$ and $y \notin L$, then x and y are distinguishable by L : we can take the distinguishing extension to be the empty string ε . Thus the equivalence classes will contain strings which are either all in L or all not in L .

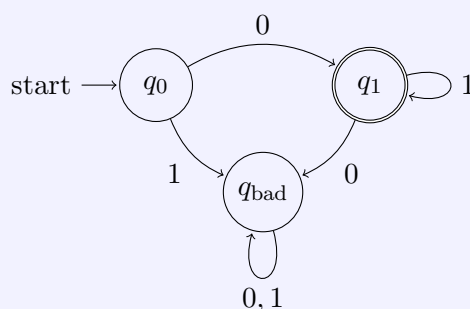
We can now state the theorem.

Theorem C.3 (Myhill-Nerode). *Let $L \subseteq \Sigma^*$ be a language. Then L is regular if and only if the number of equivalence classes of $\sim_L$ is finite. Furthermore, if L is regular, then the number of equivalence classes of $\sim_L$ is also the number of states in the minimal DFA.^a*

^aCan you see how this relates to the algorithm for determining the minimum number of states in a DFA?

Let us see three examples.

Example C.1. Let $L = L(01^*)$. What are the equivalence classes? First notice that ε is distinguishable from all other strings in $\{0, 1\}^*$, so it's in its own equivalence class. Indeed, if w is a non-empty binary string then $w0$ is not in L but $\varepsilon 0 = 0$ is. Another equivalence class consists of strings which are not in the right “format”, since then no matter what extension z is added, the string would not be in the language. This would include strings which start with 1 and strings which contain more than one 0. A third equivalence class consists of the strings in L .^a These are the only three equivalence classes, so L is regular and has a minimal DFA with 3 states. Consider the following minimal DFA:



How equivalence classes relate to states. Each equivalence class simply consists of the strings which cause the DFA to be in the same state. Indeed, the initial state has no arrows going into it, and thus correspond to the string ε . The state q_{bad} refers to the strings with “improper format,” which the DFA sees as the strings starting with 1 or that have more than one 0. Finally, the state q_1 refers to the strings in the language. Indeed, the fact that L itself is contained entirely in one equivalence class corresponds to the fact that the minimal DFA has only one accepting state.

^aCan you see why?

Example C.2. Let $L = \{w \in \{0, 1\}^* \mid w \text{ is a palindrome}\}$. We will prove that this language is not regular by showing that it has infinitely many equivalence classes. Again, we can do this by coming up with an example of infinitely many strings such that no two are equivalent. We consider all strings of the form a^k for $k \geq 0$, and show that no two of them are equivalent. Take $x = a^k$, $y = a^j$ for any $k \neq j$. Choosing the extension $z = ba^k$ gives $xz = a^kba^k \in L$, and $yz = a^jba^k \notin L$. Therefore, all such pairs a^k, a^j are distinguishable, and we have infinitely many equivalence classes, so L is not regular.^a

^aNote that there are many other possible infinite sets of strings we could use where every pair is distinguishable. In fact, in this example one could prove that every string is in its own equivalence class, namely that no two strings are equivalent.

D. DFA can be exponentially larger than equivalent NFA

One interesting fact is that the transformation from an NFA to an equivalent DFA via the subset construction necessitates an exponential blow up in the number of states. We can show this by exemplifying a specific language for which we prove that the best possible NFA is exponentially

smaller than the best possible DFA.

Definition D.1. For arbitrary $n \in \mathbb{N}$, define the language

$$L_n = \{w \in \{0,1\}^* \mid \text{the } n\text{th to last symbol in } w \text{ is } 1\}.$$

Claim D.2. For all n , there is an NFA with $n + 1$ states recognising L_n .

Proof. For any fixed n , define the NFA $N_n = (\{q_0, \dots, q_n\}, \{0,1\}, \delta, q_0, \{q_n\})$, where δ is defined as follows:

$$\begin{aligned} \delta(q_0, 0) &= \{q_0\} \\ \delta(q_0, 1) &= \{q_0, q_1\} \\ \delta(q_i, a) &= \{q_{i+1}\} & \forall i \in \{1, \dots, n-1\}, a \in \{0,1\} \\ \delta(q_n, a) &= \emptyset & \forall a \in \{0,1\} \end{aligned}$$

It is not hard to verify that N_n recognises the language L_n . ■

Theorem D.3. For any $n \in \mathbb{N}$, any DFA recognising the language L_n must have at least 2^n states.

Proof. Assume towards contradiction that there is a DFA D with fewer than 2^n states recognising the language L_n .

Consider the set of all n -bit strings. There are 2^n different n -bit strings, but fewer than 2^n states in D , so by the pigeonhole principle, there must be two distinct strings $w, w' \in \{0,1\}^n$ such that the computation of D on w and w' ends in the same state. Note that this also means that the computation of D on wz and on $w'z$ ends in the same state, for any string z . Write $w = w_1 \dots w_n$ and $w' = w'_1 \dots w'_n$ as the bit representation of the strings. Since $w \neq w'$, they must differ in at least one bit location. Take some $i \in \{1, \dots, n\}$ such that $w_i \neq w'_i$, which means that one of w_i, w'_i is 0 and the other one is 1. Append the string $z = 0^{i-1}$ to each of w, w' . The n th to last bit in $w_1 \dots w_n 0^{i-1}$ is w_i , and the n th to last bit in $w'_1 \dots w'_n 0^{i-1}$ is w'_i , one of which is 0 and one of which is 1. Thus, one of these strings is in L_n and one is not in L_n .

However, the computation of the DFA D on both these strings ends in the same state, which, whether it is an accepting state or not, gives the wrong output on one of these strings. That is, one of these two strings $w0^{i-1}$ or $w'0^{i-1}$ is a counterexample contradicting the fact that D recognises L_n . ■

This exponential blow up in the number of states we have in our subset construction is inherent:

Corollary D.4. An algorithm that can transform any NFA with m states to an equivalent DFA with $f(m)$ states, must have $f(m) = \Omega(2^m)$.

E. The Tandem Pumping Lemma as an Adversarial Game

Like the earlier pumping lemma for regular languages, we can construct an adversarial version for the CFLs thus:

1. We pick a language L that we want to show is not a context-free language.
2. Our “adversary” gets to pick p , which we do not know; we must plan for any possible p .
3. We get to pick w , and may use p as a parameter when we do so.
4. Our adversary breaks w into $uvxyz$, subject to the constraints $|vxy| \leq p$ and $vy \neq \varepsilon$.
5. We “win” the game, if we can, by picking i and showing that $uv^i xy^i z$ is not in L .

F. Encodings

As we know, Turing machines are given their input as the starting content of the tape when the machine begins execution. However, many familiar algorithms do not deal with “strings”: we have trees, graphs, queues, and many other abstract mathematical structures. In order to develop Turing machines which work with these structures, we need to have a method for turning them into strings in a standardised way. The process to do this will differ for each type of structure, so we will not discuss all of them here.

Most interestingly, we can develop an encoding method for Turing machines themselves, which allows Turing machines to reason about Turing machines. This shouldn’t be too shocking, since every time we write out a formal description of a Turing machine we are indeed converting it to a string. We will discuss one such method of encoding in the next few paragraphs, but know that many different encodings are possible.

Recall that the formal definition of a Turing machine M is a tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})$, with the meanings of all terms retained from Definition 12.1. To make the encoding easier, we will make some assumptions about the structure of our TM.

Claim. Any arbitrary TM can be converted to one satisfying the following constraints: the state set $Q = \{0, 1, \dots, n\}$, the start state $q_0 = 0$, the accept state $q_{\text{acc}} = 1$, the reject state $q_{\text{rej}} = 2$, the input alphabet $\Sigma = \{0, 1\}$, and the tape alphabet $\Gamma = \Sigma \cup \{0, 1, \dots, m\}$ where $\sqcup = 2$ (i.e. 2 is our blank symbol).

We assume further that the machine M is deterministic, since we have already seen the equivalence of deterministic and nondeterministic Turing machines. Thus the only information that we must encode is the value of n, m and the transitions of δ .

Suppose that $\delta(q_i, a) = (q_j, b, X)$. To encode this transition, we write $0^{i+1}10^{a+1}10^{j+1}10^{b+1}1x$, where $x = 0$ if $X = L$ and $x = 00$ if $X = R$. For example, if we had the transition $\delta(4, 0) = (3, 1, L)$, then the encoding of this transition would be 00000101000010010. Now, for every transition specified by δ , we encode that transition and append them all together, separated by 11. Thus, if δ were the function $\delta(0, 1) = (1, 1, R)$, $\delta(0, 0) = (0, 0, R)$, and $\delta(0, 2) = (0, 2, R)$, we would have the encoding of δ as:

01001001001001101010101001101000101000100

Finally, we encode n as a string of n zeros 0^n , and append this with the encoding for δ , separated by three 1's. The same is done for m . Thus the full encoding of the above machine would be

00011100011101001001001001101010101001101000101000100

This process of encoding can be applied to any (finite) mathematical structure. Thus, we can talk about the encodings of DFAs, NFAs, PDAs, graphs, tuples, etc. In general, we use the notation $\langle X \rangle$ to denote the encoding of the object X .