CMU 15-251
Great Theoretical Ideas in Computer Science

Fall 2015

Course Notes: Definitions and Proofs[*]

November 27, 2015

[*]Please send comments and corrections to `aada@cs.cmu.edu`.

Acknowledgements

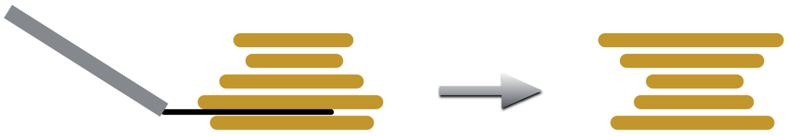
These notes are based on the lectures given by Ariel Procaccia and Anil Ada for the Fall 2015 edition of the course 15-251 “Great Theoretical Ideas in Computer Science” at Carnegie Mellon University. They are also very much related to the Spring 2015 edition of the course and the lectures given by Ryan O’Donnell. The purpose of these notes is to collect the definitions and proofs seen in class. These notes do *not* contain full explanations of all the material covered during lectures.

Thanks to Eric Bae, Teddy Ding, Ellen Kim, Aditya Krishnan, Matthew Salim, Ticha Sethapakdi, Vanessa Siriwalothakul, Natasha Vasthare, Jenny Wang, Ling Xu and Ming Yang for sending valuable comments and corrections.

Contents

1 Pancake Sorting Problem

Definition 1.1 (Pancake numbers)**.**
We are given a stack of n pancakes, each of different size. Our goal is to sort this stack from smallest to largest (largest being on the bottom of the stack). The only thing we are allowed to do is to insert the spatula in between two pancakes (or between the bottom pancake and the plate), and flip over all the pancakes that are on top of the spatula.



We are interested in the maximum number of flips (in terms of n) we would need to sort a stack of n pancakes, where the maximum is over all stacks with n pancakes. In other words, we are interested in

$$P_n = \max_S \min_A \quad \text{number of flips that method } A \text{ takes to sort stack } S.$$

Here, the maximum is over all pancake stacks of size n, and the minimum is over all methods/algorithms for sorting a given stack of pancakes. ■

Notation 1.2. We represent a stack of n pancakes with a permutation of $\{1, 2, \ldots, n\}$. Here, the numbers correspond to how large the pancake is, so 1 represents the smallest pancake and n represents the largest pancake. For example, (5 2 3 4 1) corresponds to a stack of 5 pancakes, where the largest pancake 5 is at the top of the stack, and the smallest pancake 1 is at the bottom.

Proposition 1.3 (Number of flips required for (5 2 3 4 1))**.** *Let X denote the minimum number of flips needed to sort the stack* (5 2 3 4 1)*. Then $X = 4$.*

Proof. To prove $X \leq 4$, we show how to sort (5 2 3 4 1) in 4 flips:

$$(5\ 2\ 3\ 4\ 1) \to (1\ 4\ 3\ 2\ 5) \to (2\ 3\ 4\ 1\ 5) \to (4\ 3\ 2\ 1\ 5) \to (1\ 2\ 3\ 4\ 5).$$

We now prove $X \geq 4$. The proof is by contradiction, so assume that there is a way to sort (5 2 3 4 1) in 3 or less flips.
Observation. Right before a pancake is placed at the bottom of the stack, it must be placed at the top of the stack.
Claim. The first flip must put 5 on the bottom of the stack.
Proof of Claim. Suppose the first flip does not put 5 on the bottom of the stack, so it puts it somewhere in the middle. Then we can show that (5 2 3 4 1) cannot be sorted in 3 or less flips. We know that after 3 flips, 5 must be placed at the bottom of the

stack. The observation above implies that the second flip must send 5 to the top. So in the first two flips, 5 first gets sent from the top to somewhere in the middle, and then it gets flipped back up to the top. In other words, after 2 flips we end up with the original stack (5 2 3 4 1). There is no way to sort (5 2 3 4 1) with the remaining flip, which proves the claim.

So we know that the first flip must be (5 2 3 4 1) $\to$ (1 4 3 2 5). In the remaining two flips, 4 must be placed next to 5. It is clear that 5 should not be touched (i.e., we should not be flipping the whole stack). So we can ignore 5 and and just consider the stack of 4 pancakes (1 4 3 2). We need to put 4 at the bottom of this stack in 2 flips. Again, using the observation stated above, we know that 4 must be first placed at the top of the stack. So the 2 flips must be (1 4 3 2) $\to$ (4 1 3 2) $\to$ (2 3 1 4). The resulting stack is not sorted, which is the desired contradiction. $\square$

Theorem 1.4. *For $n \geq 4$, we have*

$$n \leq P_n \leq 2n - 3.$$

The proof of the theorem follows from the following two lemmas.

Lemma 1.5. *For $n \geq 2$, we have $P_n \leq 2n - 3$.*

Proof. Consider the following algorithm for sorting an arbitrary stack of n pancakes.

- If $n = 1$: do nothing.
- If $n = 2$: sort the pancakes in one flip if they are not already sorted.
- Else (if $n \geq 3$):
  - Bring the largest pancake to the bottom of the stack in 2 flips.
  - Recursively sort the remaining $n - 1$ pancakes.

Clearly,[1] the algorithm correctly sorts a given stack of pancakes. Let $T(n)$ be the number of flips that this algorithm uses to sort a stack of n pancakes. By the definition of P_n, $P_n \leq T(n)$. So we are done once we show $T(n) \leq 2n-3$ for $n \geq 2$. The recursive relation that $T(n)$ satisfies is

$$\begin{aligned} &T(1) = 0, \\ &T(2) \leq 1, \\ &T(n) \leq 2 + T(n-1) \quad \text{for } n \geq 3. \end{aligned}$$

This implies that $T(n) \leq 2n - 3$ for $n \geq 2$, which completes the proof.[2] $\square$

[1]You should be careful using the word "clearly". In this case, it is justified.

[2]To be more complete, you can prove $T(n) \leq 2n - 3$ for $n \geq 2$ with a quick induction. This part is omitted.

Exercise 1.6. Show by induction that the recurrence relation in the above proof solves to $T(n) \leq 2n - 3$ for $n \geq 2$.

Lemma 1.7. *For $n \geq 4$, we have $P_n \geq n$.*

Proof. Given $i, j \in \{1, 2, \dots, n\}$ and a pancake stack, we say that (i, j) form a *bad* pair with respect to that stack if they are adjacent in the stack, and $|i - j| > 1$ (i.e., they are not supposed to be adjacent once the stack is sorted). Observe that if two pancakes are adjacent in a stack, they will remain adjacent if the spatula is never inserted in between them. This means that if (i, j) form a bad pair, then any sorting method that sorts the stack *must* insert the spatula in between i and j at some point. Note that we can also consider the bottom pancake and the plate as a bad pair too. If we never insert the spatula at the bottom of the stack, then the bottom pancake and the plate will remain adjacent. So we extend our definition of a bad pair to include the plate too.

Now we can conclude that a stack with b bad pairs needs at least b flips to be sorted. We finish the proof by showing that for $n \geq 4$, there is a stack of n pancakes containing n bad pairs. We do this by considering two cases: when n is even and when n is odd. When n is even, the following stack has n bad pairs:

$$(2\ 4\ 6\ \cdots\ n-2\ n\ 1\ 3\ 5\ \cdots\ n-1).$$

When n is odd, the following stack has n bad pairs:

$$(1\ 3\ 5\ \cdots\ n-2\ n\ 2\ 4\ 6\ \cdots\ n-1).$$

(Note that the assumption $n \geq 4$ is required so that the pancakes right in the middle of the stacks form a bad pair.) □

Exercise 1.8. Suppose we are allowed to take any contiguous set of pancakes and flip them in place (they need not be on the top of the stack). Let Q_n be the maximum over stacks of size n of the minimum number of flips required to sort that stack, using this new flipping operation. Show that $n/2 \leq Q_n \leq n - 1$ for all $n \geq 2$.

2 Deterministic Finite Automata

Notation 2.1. We let Σ denote a finite and non-empty set of symbols, which we refer to as the *alphabet*. Then, Σ^* denotes the set of all words/strings over the alphabet Σ with finitely many symbols. This includes the string with no symbols. As examples, consider the following:

$$\{1\}^* = \{\epsilon, 1, 11, 111, 1111, 11111, \ldots\},$$
$$\{0,1\}^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, 101, 110, 111, \ldots\}.$$

Above, ϵ denotes the *empty word*, i.e. the word with no symbols.

If u and v are two strings in Σ^*, then we denote by uv the string obtained by concatenating u and v. For example, if $u = 101$ and $v = 001$, then $uv = 101001$. For a word $u \in \Sigma^*$, we denote by u^n the word obtained by concatenating u with itself n times. For example, if $u = 101$ then $u^3 = 101101101$.

For $u \in \Sigma^*$, $|u|$ is called the *length* of u and is defined to be the number of symbols in u.

Exercise 2.2. Let Σ be some arbitrary finite alphabet. Let u and v be two non-empty words with the property that $uv = vu$. Prove that there must be a non-empty word w and numbers $i, j \in \mathbb{N}^+$ such that $u = w^i$ and $v = w^j$.
(Hint: Prove by induction on $|uv|$.)

Definition 2.3 (Language).
Any subset $L \subseteq \Sigma^*$ is called a *language* over the alphabet Σ. ■

Definition 2.4 (Computational problem).
Any function $f : \Sigma^* \to \Sigma^*$ is called a *computational problem.* ■

Definition 2.5 (Decision problem).
Any function $f : \Sigma^* \to \{0,1\}$ is called a *decision problem.* The range of the function is not important as long as it has two elements. Other common choices for the range are $\{\text{Yes}, \text{No}\}$, $\{\text{True}, \text{False}\}$ and $\{\text{Accept}, \text{Reject}\}$. Any $w \in \Sigma^*$ is called an *input* or an *instance* of the decision problem. ■

Remark. There is a one-to-one correspondence between decision problems and languages. Let $f : \Sigma^* \to \{0,1\}$ be some decision problem. Now define $L \subseteq \Sigma^*$ to be the set of all words in Σ^* that f maps to 1. This L is the language corresponding to the decision problem f. Similarly, if you take any language $L \subseteq \Sigma^*$, we can define the corresponding decision problem $f : \Sigma^* \to \{0,1\}$ as $f(w) = 1$ iff $w \in L$. We consider the set of languages and the set of decision problems to be the same set of objects.

Exercise 2.6. Consider the problem of primality testing: given a number in $\mathbb{N}$, output True if the number is prime, and output False otherwise. Express the problem as a decision problem $f : \{0,1\}^* \to \{0,1\}$. What is the language corresponding to this decision problem?

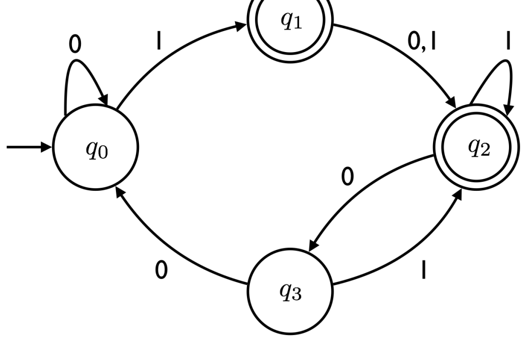
Definition 2.7 (Deterministic Finite Automaton (DFA))**.**
A *deterministic finite automaton* (DFA) M is a 5-tuple

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

where

- Q is a finite set
  (which we refer to as the *set of states*);
- Σ is a finite set
  (which we refer to as the *alphabet*, and each element of the alphabet is called a *symbol* of the alphabet);
- δ is a function of the form $\delta : Q \times \Sigma \to Q$
  (which we refer to as the *transition function*);
- $q_0 \in Q$ is an element of Q
  (which we refer to as the *start state*);
- $F \subseteq Q$ is a subset of Q
  (which we refer to as the *set of accepting states*).

Below is an example of how we draw a DFA:



In this example, $\Sigma = \{0, 1\}$, $Q = \{q_0, q_1, q_2, q_3\}$, $F = \{q_1, q_2\}$. The labeled arrows between the states encode the transition function δ, which can also be represented with a table as below (row $q_i \in Q$ and column $b \in \Sigma$ contains $\delta(q_i, b)$).

	0	1
q_0	q_0	q_1
q_1	q_2	q_2
q_2	q_3	q_2
q_3	q_0	q_2

■

Definition 2.8 (A DFA accepting a string)**.**
Let $w = w_1 w_2 \cdots w_n$ be a string over an alphabet Σ (so $w_i \in \Sigma$ for each $i \in \{1, 2, \ldots, n\}$). Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA. We say that M *accepts* the string w if there exists a sequence of states $r_0, r_1, r_2, \ldots, r_n \in Q$ such that:

- $r_0 = q_0$;
- $\delta(r_{i-1}, w_i) = r_i$ for each $i \in \{1, 2, \ldots, n\}$;
- $r_n \in F$.

Otherwise, we say that M *rejects* the string w. ■

Definition 2.9 (Language recognized/accepted by a DFA)**.**
For a deterministic finite automaton M, we let $L(M)$ denote the set of all strings that M accepts, i.e. $L(M) = \{w \in \Sigma^* : M \text{ accepts } w\}$. We refer to $L(M)$ as the language *recognized* by M (or as the language *accepted* by M).[3] ■

Definition 2.10 (Regular language)**.**
A language $L \subseteq \Sigma^*$ is called *regular* if there is a deterministic finite automaton M such that $L = L(M)$. ■

Theorem 2.11 (A non-regular language)**.**
The language $L = \{0^n 1^n : n \in \mathbb{N}\}$ is **not** *regular.*

Proof. The proof is by contradiction. So let's assume that L is regular. By definition, this means that there is some deterministic finite automaton M that decides L. Let k denote the number of states of M. For $n \in \mathbb{N}$, let r_n denote the state that M reaches after reading 0^n. By the pigeonhole principle[4], we know that there must be a repeat among $r_0, r_1, \ldots, r_k$. In other words, there are indices $i, j \in \{0, 1, \ldots, k\}$ with $i \neq j$ such that $r_i = r_j$. This means that the string 0^i and the string 0^j end up in the same state in M. Therefore $0^i w$ and $0^j w$, *for any* string $w \in \{0,1\}^*$, end up in the same state in M. We'll now reach a contradiction, and conclude the proof, by considering a particular w such that $0^i w$ and $0^j w$ end up in different states. Consider the string $w = 1^i$. Then since M decides L, we know $0^i w = 0^i 1^i$ must end up in an accepting state. On the other hand, since $i \neq j$, $0^j w = 0^j 1^i$ is not in the language, and therefore cannot end up in an accepting state. □

Exercise 2.12. Let $\Sigma = \{a, b, c\}$. Is the language $L = \{a^n b^n c^n : n \in \mathbb{N}\}$ regular?

Exercise 2.13. Let $\Sigma = \{0, 1\}$. Is the following language regular?

$L = \{w \in \Sigma^* : w \text{ contains an equal number of occurrences of 01 and 10 as substrings.}\}$

[3] Here the word "accept" is overloaded since we also use it in the context of a DFA accepting a string. However, this usually does not create any ambiguity.

[4] The *pigeonhole principle* states that if n items are put inside m containers, and $n > m$, then there must be at least one container with more than one item. The name *pigeonhole principle* comes from thinking of the items as pigeons, and the containers as holes.

Exercise 2.14. Is it true that if $L \subseteq \Sigma^*$ is a regular language, then any $L' \subseteq L$ is also a regular language?

Theorem 2.15 (Regular languages are closed under the union operation)**.**
Let Σ be some finite alphabet. If $L_1 \subseteq \Sigma^*$ and $L_2 \subseteq \Sigma^*$ are regular languages, then the language $L_1 \cup L_2$ is also regular.

Proof. Since L_1 and L_2 are regular languages, by definition, there are DFAs $M = (Q, \Sigma, \delta, q_0, F)$ and $M' = (Q', \Sigma, \delta', q'_0, F')$ that decide L_1 and L_2 respectively. To show $L_1 \cup L_2$ is regular, we'll construct a DFA $M'' = (Q'', \Sigma, \delta'', q''_0, F'')$ that decides $L_1 \cup L_2$. The definition of M'' will make use of M and M'. In particular:

- $Q'' = Q \times Q' = \{(q, q') : q \in Q, q' \in Q'\}$,
- δ'' is defined such that for $(q, q') \in Q''$ and $a \in \Sigma$,

$$\delta''((q, q'), a) = (\delta(q, a), \delta'(q', a)),$$

- $q''_0 = (q_0, q'_0)$,
- $F'' = \{(q, q') : q \in F \text{ or } q' \in F'\}$.

This completes the definition of M''. It remains to show that M'' indeed decides the language $L_1 \cup L_2$, i.e. $L(M'') = L_1 \cup L_2$. We'll first argue that $L_1 \cup L_2 \subseteq L(M'')$ and then argue that $L(M'') \subseteq L_1 \cup L_2$. Both inclusions will follow easily from the definition of M'' and the definition of a DFA accepting a string.

$L_1 \cup L_2 \subseteq L(M'')$: Suppose $w \in L_1 \cup L_2$, which means w either belongs to L_1 or it belongs to L_2. Our goal is to show that $w \in L(M'')$. Without loss of generality, assume w belongs to L_1, or in other words, M accepts w. Let n be the length of w. Using Definition 2.8, we know w induces a sequence of states $r_0, r_1, \ldots, r_n \in Q$ such that $r_0 = q_0$, $\delta(r_{i-1}, w_i) = r_i$ for each $i \in \{1, 2, \ldots, n\}$, and $r_n \in F$. This w will also induce a sequence of states of M'', $r'_0, r'_1, \ldots, r'_n \in Q'$ such that $r'_0 = q'_0$, $\delta'(r'_{i-1}, w_i) = r'_i$ for each $i \in \{1, 2, \ldots, n\}$, but r'_n is not necessarily an element of F' (because M' need not accept w). Due to how we have defined M'', when we run w on M'', it will induce the sequence of states $(r_0, r'_0), (r_1, r'_1), \ldots, (r_n, r'_n) \in Q''$ such that $(r_0, r'_0) = (q_0, q'_0)$, $\delta''((r_{i-1}, r'_{i-1}), w_i) = (\delta(r_i, a), \delta'(r'_i, a))$ for each $i \in \{1, 2, \ldots, n\}$. The final state (r_n, r'_n) is such that $r_n \in F$, which implies by the definition of F'' that $(r_n, r'_n) \in F''$. Therefore M'' accepts w, i.e. $w \in L(M'')$. This establishes $L_1 \cup L_2 \subseteq L(M'')$.

$L(M'') \subseteq L_1 \cup L_2$: Suppose that $w \in L(M'')$. Our goal is to show that $w \in L_1 \cup L_2$. We know that w induces a sequence of states of M'', $(r_0, r'_0), (r_1, r'_1), \ldots, (r_n, r'_n) \in Q''$, such that $(r_0, r'_0) = (q_0, q'_0)$, $\delta''((r_{i-1}, r'_{i-1}), w_i) = (\delta(r_i, a), \delta'(r'_i, a))$ for each $i \in \{1, 2, \ldots, n\}$, and $(r_n, r'_n) \in F''$. By the definition of F'', this implies that either $r_n \in F$ or $r'_n \in F'$. Without loss of generality assume $r_n \in F$. If we run w on M, it would induce the sequence of states $r_0, r_1, \ldots, r_n \in Q$ such that $r_0 = q_0$, $\delta(r_{i-1}, w_i) = r_i$ for each $i \in \{1, 2, \ldots, n\}$. And since $r_n \in F$, M would accept w. Therefore $w \in L_1 \cup L_2$. This establishes $L(M'') \subseteq L_1 \cup L_2$, completing the proof. $\square$

Exercise 2.16. The above proof shows that regular languages are closed under the union operation. Suppose we want to show that regular languages are closed under the intersection operation. How can we modify the above proof to show this?

Exercise 2.17. Suppose that $L_1 \subseteq \Sigma_1^*$ and $L_2 \subseteq \Sigma_2^*$ are regular languages, where Σ_1 and Σ_2 are not necessarily the same set. Is it still true that $L_1 \cup L_2$ is regular? Justify your answer.

Exercise 2.18. Let L be a finite language, i.e., it contains a finite number of words. Show that L is regular.

3 Turing Machines

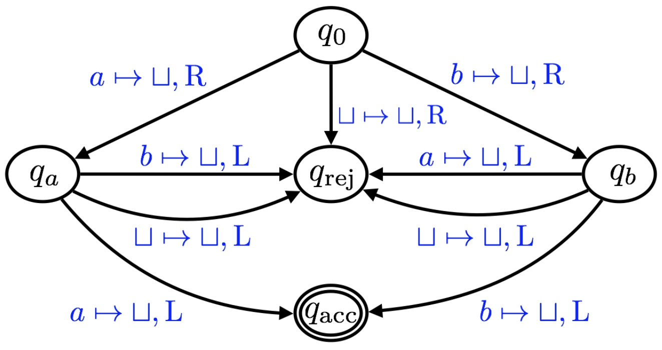
Definition 3.1 (Turing machine)**.**
A Turing machine (TM) M is a 7-tuple

$$M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}}),$$

where

- Q is a finite set
  (which we refer to as the *set of states*);
- Σ is a non-empty finite set that does not contain the *blank symbol* $\sqcup$
  (which we refer to as the *input alphabet*);
- Γ is a finite set such that $\sqcup \in \Gamma$ and $\Sigma \subset \Gamma$
  (which we refer to as the *tape alphabet*);
- δ is a function of the form $\delta : Q \times \Gamma \to Q \times \Gamma \times \{\mathrm{L}, \mathrm{R}\}$
  (which we refer to as the *transition function*);
- $q_0 \in Q$ is an element of Q
  (which we refer to as the *start state*);
- $q_{\text{acc}} \in Q$ is an element of Q
  (which we refer to as the *accepting state*);
- $q_{\text{rej}} \in Q$ is an element of Q such that $q_{\text{rej}} \neq q_{\text{acc}}$
  (which we refer to as the *rejecting state*).

Below is an example of how we draw a TM:


q_0
$a \mapsto \sqcup, \mathrm{R}$
$b \mapsto \sqcup, \mathrm{R}$
$\sqcup \mapsto \sqcup, \mathrm{R}$
q_a
$b \mapsto \sqcup, \mathrm{L}$
q_{rej}
$a \mapsto \sqcup, \mathrm{L}$
q_b
$\sqcup \mapsto \sqcup, \mathrm{L}$
$\sqcup \mapsto \sqcup, \mathrm{L}$
$a \mapsto \sqcup, \mathrm{L}$
q_{acc}
$b \mapsto \sqcup, \mathrm{L}$

In this example, $\Sigma = \{a, b\}$, $\Gamma = \{a, b, \sqcup\}$, $Q = \{q_0, q_a, q_b, q_{\text{acc}}, q_{\text{rej}}\}$. The labeled arrows between the states encode the transition function δ. The above picture is called the *state diagram* of the Turing machine. ■

Remark. We'll consider two Turing machines to be equivalent/same if they are the same machine up to renaming the elements of the sets Q, Σ and Γ.

Exercise 3.2. For each language below, draw a TM that decides the language. You can use any finite tape alphabet Γ containing Σ and $\sqcup$.

(a) $L = \{a^n : n \text{ is a nonnegative integer power of } 2\}$, where $\Sigma = \{a\}$.

(b) $L = \{x \in \{a, b\}^* : x \text{ has the same number of } a\text{'s and } b\text{'s}\}$.

Definition 3.3 (A TM accepting or rejecting a string)**.**
Let M be a Turing machine where Q is the state set, $\sqcup$ is the blank symbol, and Γ is the tape alphabet.[5] To understand how M's computation proceeds we generally need to keep track of three things: (i) the state M is in; (ii) the contents of the tape; (iii) where the tape head is. These three things are collectively known as the "configuration" of the TM. More formally: a *configuration* for M is defined to be a string $uqv \in (\Gamma \cup Q)^*$, where $u, v \in \Gamma^*$ and $q \in Q$. This represents that the tape has contents $\cdots \sqcup\sqcup\sqcup uv \sqcup\sqcup\sqcup \cdots$, the head is pointing at the leftmost symbol of v, and the state is q. We say the configuration is *accepting* if q is M's accept state and that it's *rejecting* if q is M's reject state.
Technicality alert: We also have some technicalities: The string u cannot start with $\sqcup$ and the string v cannot end with $\sqcup$. This is so that the configuration is always unique. Also, if $v = \epsilon$ it means the head is pointing at the $\sqcup$ immediately to the right of u.

Suppose that M reaches a certain configuration α (which is not accepting or rejecting). Knowing just this configuration and M's transition function δ, one can determine the configuration β that M will reach at the next step of the computation. (As an exercise, make this statement precise.) We write

$$\alpha \vdash_M \beta$$

and say that "α yields β (in M)". If it's obvious what M we're talking about, we just write $\alpha \vdash \beta$.

Given an input $x \in \Sigma^*$ we say that $M(x)$ *halts* if there exists a sequence of configurations (called the *computation trace*) $\alpha_0, \alpha_1, \ldots, \alpha_T$ such that:

(i) $\alpha_0 = q_0 x$, where q_0 is M's initial state;

(ii) $\alpha_t \vdash_M \alpha_{t+1}$ for all $t = 0, 1, 2, \ldots, T-1$;

(iii) α_T is either an accepting configuration (in which case we say $M(x)$ *accepts*) or a rejecting configuration (in which case we say $M(x)$ *rejects*).

Otherwise, we say $M(x)$ *loops*. ■

[5] Supernerd note: we will always assume Q and Γ are disjoint sets.

Definition 3.4 (Decider Turing machine)**.**
A Turing machine is called a *decider* if it halts on all inputs. ■

Definition 3.5 (Language accepted/decided by a TM)**.**
Let M be a Turing machine. We denote by $L(M)$ the set of all strings that M accepts. When M is a decider, we say that M *decides* the language $L(M)$. ■

Definition 3.6 (Decidable/Computable language)**.**
A language L is called *decidable* (or *computable*) if $L = L(M)$ for some decider Turing machine M. ■

Notation 3.7. Let M be a Turing machine and Σ a finite alphabet. We denote by $\langle M \rangle \in \Sigma^*$ a string encoding of M.[6]

Technicality Alert: After fixing some encoding scheme, it is clear that every $\langle M \rangle$ for some TM M corresponds to a word $w \in \Sigma^*$. We can also say that every $w \in \Sigma^*$ actually corresponds to $\langle M \rangle$ for some TM M, by assuming that if w does not encode a proper TM, then we'll automatically assume that it corresponds to the simple TM that rejects everything. We'll adopt this convention in order to avoid some uninteresting technicalities in the future.

Notation 3.8. We use the notation $\langle \cdot \rangle$ to denote an encoding of not just Turing machines, but any object we want. For example, if D is a DFA, we can write $\langle D \rangle$ to denote the encoding of D as a string. When we want to encode a tuple of objects, we use the comma sign. For example, if M_1 and M_2 are two Turing machines, we can write $\langle M_1, M_2 \rangle$ to denote the encoding of the tuple (M_1, M_2). As another example, if M is a TM and $x \in \Sigma^*$, we can write $\langle M, x \rangle$ to denote the encoding of the tuple (M, x).

Definition 3.9 (Universal Turing machine)**.**
Let Σ be some finite alphabet. A *universal Turing machine* U is a Turing machine that takes $\langle M, x \rangle$ as input, where M is a TM and x is a word in Σ^*, and has the following description:

```
"On input <M,x>:
    Simulate M on input x (i.e. run M(x)).
    If it accepts, accept.
    If it rejects, reject.
"
```

Note that if $M(x)$ loops forever, then U loops forever as well. ■

[6] As an example, if P is some Python program, we can take $\langle P \rangle$ to be the string that represents the source code of the program. We saw in class that one can view a TM as a piece of code as well. So we can take, for example, $\langle M \rangle$ to be the string that represents that code.

4 Countable and Uncountable Sets

Definition 4.1 (Injection, surjection, and bijection)**.**
Let A and B be two (possibly infinite) sets.

- A function $f : A \to B$ is called *injective* if for any $a, a' \in A$ such that $a \neq a'$, we have $f(a) \neq f(a')$. We write $A \hookrightarrow B$ if there exists an injective function from A to B.
- A function $f : A \to B$ is called *surjective* if for all $b \in B$, there exists an $a \in A$ such that $f(a) = b$. We write $A \twoheadrightarrow B$ if there exists a surjective function from A to B.
- A function $f : A \to B$ is called *bijective* if it is both injective and surjective. We write $A \leftrightarrow B$ if there exists a bijective function from A to B.

■

Theorem 4.2. *Let A, B and C be three (possibly infinite) sets. Then,*

- *$A \hookrightarrow B$ if and only if $B \twoheadrightarrow A$,*
- *$A \leftrightarrow B$ if and only if $A \hookrightarrow B$ and $B \hookrightarrow A$,*
- *if $A \hookrightarrow B$ and $B \hookrightarrow C$, then $A \hookrightarrow C$.*

Notation 4.3. Let A and B be two (possibly infinite) sets.

- We write $|A| = |B|$ if $A \leftrightarrow B$.
- We write $|A| \leq |B|$ if $A \hookrightarrow B$, or equivalently, if $B \twoheadrightarrow A$.[7]
- We write $|A| < |B|$ if it is not the case that $|A| \geq |B|$.[8]

Definition 4.4 (Countable and uncountable sets)**.**

- A set A is called *countable* if $|A| \leq |\mathbb{N}|$.
- A set A is called *countably infinite* if it is countable and infinite.
- A set A is called *uncountable* if it is not countable, i.e. $|A| > |\mathbb{N}|$.

■

Theorem 4.5. *A set A is countably infinite if and only if $|A| = |\mathbb{N}|$.*

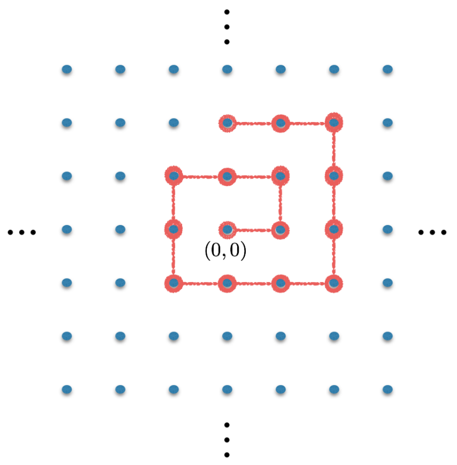
Exercise 4.6. Can you prove the above theorem?

[7] Even though not explicitly stated, $|B| \geq |A|$ has the same meaning as $|A| \leq |B|$.
[8] Similar to above, $|B| > |A|$ has the same meaning as $|A| < |B|$.

4.1 Countable sets

Proposition 4.7. *The set $\mathbb{Z} \times \mathbb{Z}$ is countable.*

Proof. Consider the plot of $\mathbb{Z} \times \mathbb{Z}$ on a 2-dimensional grid. Starting at $(0, 0)$ we can list all the elements of $\mathbb{Z} \times \mathbb{Z}$ using a spiral shape, as shown below.



(The picture shows only a small part of the spiral.) This gives us a way to list all the elements of $\mathbb{Z} \times \mathbb{Z}$ such that every element eventually appears in the list. This implies that there is a surjective function f from $\mathbb{N}$ to $\mathbb{Z} \times \mathbb{Z}$: $f(i)$ is defined to be the i'th element in the list. Since there is a surjection from $\mathbb{N}$ to $\mathbb{Z} \times \mathbb{Z}$, $|\mathbb{Z} \times \mathbb{Z}| \leq |\mathbb{N}|$, and $\mathbb{Z} \times \mathbb{Z}$ is countable.[9] $\square$

Proposition 4.8. *The set of rational numbers $\mathbb{Q}$ is countable.*

Proof. This follows easily from the previous proposition. Every element of $\mathbb{Q}$ can be written as a fraction a/b where $a, b \in \mathbb{Z}$. In other words, there is a surjection from $\mathbb{Z} \times \mathbb{Z}$ to $\mathbb{Q}$ that maps (a, b) to a/b (if $b = 0$, map (a, b) to say 0). This shows that $|\mathbb{Q}| \leq |\mathbb{Z} \times \mathbb{Z}|$. Since $\mathbb{Z} \times \mathbb{Z}$ is countable, i.e. $|\mathbb{Z} \times \mathbb{Z}| \leq |\mathbb{N}|$, $\mathbb{Q}$ is also countable, i.e. $|\mathbb{Q}| \leq |\mathbb{N}|$. $\square$

Proposition 4.9. *Let Σ be a finite set. Then Σ^* is countable.*

Proof. Recall that Σ^* denotes the set of all words/strings over the alphabet Σ with finitely many symbols. For each $n = 0, 1, 2, \ldots$, let Σ^n denote the set of words in Σ^* that have length exactly n. Note that Σ^n is a finite set for each n, and Σ^* is a union of these sets: $\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \cdots$. This gives us a way to list the elements of Σ^* so that any element of Σ^* eventually appears in the list. First list the elements of Σ^0, then list the elements of Σ^1, then list the elements of Σ^2, and so on. This way of listing the elements gives us a surjective function f from $\mathbb{N}$ to Σ^*: $f(i)$ is defined to be the i'th element in the list. Since there is a surjection from $\mathbb{N}$ to Σ^*, $|\Sigma^*| \leq |\mathbb{N}|$, and Σ^* is countable. $\square$

[9]Note that it is not a requirement that we give an explicit formula for $f(i)$. In fact, sometimes in such proofs, an explicit formula may not exist. This does not make the proof any less rigorous.

Exercise 4.10. Is the above proposition true if we allow Σ to be any countable set? In other words, is it true that if Σ is countable, then Σ^* is countable?

Proposition 4.11. *The set of all Turing machines* $\{M : M \text{ is a TM}\}$ *is countable.*

Proof. Let $T = \{M : M \text{ is a TM}\}$. Then $|T| \leq |\Sigma^*|$ for some finite set Σ because the mapping $M \mapsto \langle M \rangle$, where $\langle M \rangle \in \Sigma^*$, is an injective map. Since $|T| \leq |\Sigma^*|$, and we already know Σ^* is countable from Proposition 4.9, the result follows. □

Proposition 4.12. *Let* $\mathbb{Q}[x]$ *denote the set of all polynomials in one variable with rational coefficients.* $\mathbb{Q}[x]$ *is countable.*

Proof. We prove this using the fact that Σ^* is countable for any finite set Σ (Proposition 4.9). Let

$$\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, -, /, \hat{}\}.$$

Then observe that every element of $\mathbb{Q}[x]$ can be written as a string over this alphabet. For example,

$$2x\hat{}3 - 1/34x\hat{}2 + 99/100x\hat{}1 + 22/7.$$

This implies that there is a surjective map from Σ^* to $\mathbb{Q}[x]$. And therefore $|\mathbb{Q}[x]| \leq |\Sigma^*|$. Since Σ^* is countable, $\mathbb{Q}[x]$ is also countable. □

Exercise 4.13. Show that $\mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}$ is countable.

4.2 Uncountable sets

Definition 4.14 (Power set)**.**
Let A be any set. The set of all subsets of A is called the *power set* of A, and is denoted by $\mathcal{P}(A)$. ■

Theorem 4.15 (Cantor's Theorem)**.** *For any non-empty set* A, $|\mathcal{P}(A)| > |A|$.

Proof. The proof that we present here is called the *diagonalization argument*. The proof is by contradiction. So assume that $|\mathcal{P}(A)| \leq |A|$. By definition, this means that there is a surjective function from A to $\mathcal{P}(A)$. Let f be such a surjection. So for any $S \in \mathcal{P}(A)$, there exists an $s \in A$ such that $f(s) = S$. Now consider the set $S = \{a \in A : a \notin f(a)\}$. Since S is a subset of A, $S \in \mathcal{P}(A)$. So there is an $s \in A$ such that $f(s) = S$. But then if $s \notin S$, by the definition of S, s is in $f(s) = S$, which is a contradiction. If $s \in S$, then by the definition of S, s is not in $f(s) = S$, which is also a contradiction. So either way, we get a contradiction as desired. □

Corollary 4.16. *The set* $\mathcal{P}(\mathbb{N})$ *is uncountable.*

Corollary 4.17. *Let* Σ *be a finite set with* $|\Sigma| > 0$. *Then* $\mathcal{P}(\Sigma^*)$ *is uncountable.*

Proof. For Σ with $|\Sigma| > 0$, Σ^* is a countably infinite set (Proposition 4.9). So by Theorem 4.5, we know $|\Sigma^*| = |\mathbb{N}|$. Theorem 4.15 implies that $|\Sigma^*| < |\mathcal{P}(\Sigma^*)|$. So we have $|\mathbb{N}| = |\Sigma^*| < |\mathcal{P}(\Sigma^*)|$, which shows, by the definition of uncountable sets, that $\mathcal{P}(\Sigma^*)$ is uncountable. □

Notation 4.18. Let Σ be some finite alphabet. We denote by Σ^∞ the set of all *infinite* length words over the alphabet Σ.

Theorem 4.19. *The set* $\{0,1\}^\infty$ *is uncountable.*

Proof. One can prove this result simply by observing that $\{0,1\}^\infty \leftrightarrow \mathcal{P}(\mathbb{N})$, and using Corollary 4.16. Here, we will give a direct proof using a diagonalization argument. The proof is by contradiction, so assume that $\{0,1\}^\infty$ is countable. By definition, this means that $|\{0,1\}^\infty| \leq |\mathbb{N}|$, i.e. there is a surjective map f from $\mathbb{N}$ to $\{0,1\}^\infty$. Consider the table in which the i'th row corresponds to $f(i)$. Below is an example.

$f(1)$	$\boxed{0}$	0	0	0	0	$\cdots$
$f(2)$	1	$\boxed{1}$	1	1	1	$\cdots$
$f(3)$	0	1	$\boxed{0}$	1	0	$\cdots$
$f(4)$	1	0	1	$\boxed{0}$	1	$\cdots$
$f(5)$	0	0	1	1	$\boxed{0}$	$\cdots$
$\vdots$			$\vdots$			

(The elements in the diagonal are highlighted.) Using f, we construct an element a of $\{0,1\}^\infty$ as follows. If the i'th symbol of $f(i)$ is 1, then the i'th symbol of a is defined to be 0. And if the i'th symbol of $f(i)$ is 0, then the i'th symbol of a is defined to be 1. Notice that the i'th symbol of $f(i)$, for $i = 1, 2, 3, \ldots$ corresponds to the diagonal elements in the above table. So we are creating this element a of $\{0,1\}^\infty$ by taking the diagonal elements, and flipping their value.

Now notice that the way a is constructed implies that it cannot appear as a row in this table. This is because a differs from $f(1)$ in the first symbol, it differs from $f(2)$ in the second symbol, it differs from $f(3)$ in the third symbol, and so on. So it differs from every row of the table and hence cannot appear as a row in the table. This leads to the desired contradiction because f is a surjective function, which means every element of $\{0,1\}^\infty$, including a, *must* appear in the table. □

Exercise 4.20. Show that the following sets are uncountable.

(a) $\{a_1a_2a_3\ldots \in \{0,1\}^\infty : \forall n \geq 1,$ the string $a_1 \ldots a_n$ contains more 1's than 0's.$\}$

(b) The set of all bijections $f : \mathbb{N} \to \mathbb{N}$.

5 Undecidable Languages

Theorem 5.1. *Fix some alphabet Σ ($|\Sigma| > 0$). There are languages $L \subseteq \Sigma^*$ that are not decidable.*

Proof. To prove the result, we simply observe that the set of all languages is uncountable whereas the set of decidable languages is countable. First, consider the set of all languages. Since a language L is defined to be a subset of Σ^*, the set of all languages is $\mathcal{P}(\Sigma^*)$. By Corollary 4.17, we know that this set is uncountable. Now consider the set of all decidable languages, which we'll denote by D. Let T be the set of all TMs. By Proposition 4.11, we know that T is countable. Furthermore, the mapping $M \mapsto L(M)$ can be viewed as a surjection from T to D (if M is not a decider, just map it to $\emptyset$). So $|D| \leq |T|$. Since T is countable, this shows D is countable and completes the proof.[10] □

Definition 5.2 (Halting problem)**.**
The *halting problem* is defined as the decision problem corresponding to the language HALT $= \{\langle M, x\rangle : M$ is a TM which halts on input $x\}$. ■

Theorem 5.3 (Turing's Theorem)**.** *The language* HALT *is undecidable.*

Proof. The proof is by contradiction, so assume that HALT is decidable. By definition, this means that there is a decider TM, call it M_{HALT}, that decides HALT. We construct a new TM, which we'll call M_{TURING}, that uses M_{HALT} as a subroutine. The description of M_{TURING} is as follows:

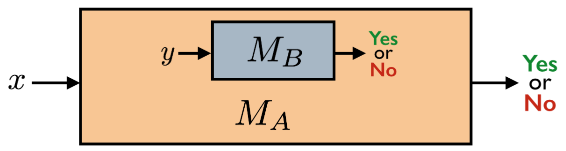
```
"On input <M>:
    run M_HALT(<M,M>).
    if it accepts, go into an infinite loop.
    if it rejects, accept.
"
```

We get the desired contradiction once we consider what happens when we feed M_{TURING} as input to itself, i.e. when we run $M_{\text{TURING}}(\langle M_{\text{TURING}}\rangle)$.

If $M_{\text{HALT}}(\langle M_{\text{TURING}}, M_{\text{TURING}}\rangle)$ accepts, then $M_{\text{TURING}}(\langle M_{\text{TURING}}\rangle)$ is supposed to halt by the definition of M_{HALT}. However, from the description of M_{TURING} above, we see that it goes into an infinite loop. This is a contradiction. The other option is that $M_{\text{HALT}}(\langle M_{\text{TURING}}, M_{\text{TURING}}\rangle)$ rejects. Then $M_{\text{TURING}}(\langle M_{\text{TURING}}\rangle)$ is supposed to lead to an infinite loop. But from the description of M_{TURING} above, we see that it accepts, and therefore halts. This is a contradiction as well. □

[10] This kind of an argument is called *non-constructive* because it does not present an explicit undecidable language. A *constructive* argument would prove the undecidability of an explicit language, as in Theorem 5.3.

Definition 5.4 (Turing reduction)**.**
Fix some alphabet Σ. Let A and B be two languages. We say that A *reduces* to B, written $A \leq_T B$, if it is possible to decide A assuming a decider TM for B exists. In other words, if a decider TM M_B for B exists, then one can construct a decider TM M_A for A which uses M_B as a subroutine.



■

Remark. Observe that if $A \leq_T B$ and B is decidable, then A is also decidable. Equivalently, taking the contrapositive, if $A \leq_T B$ and A is undecidable, then B is also undecidable. So when $A \leq_T B$, we think of B as being at least as hard as A with respect to decidability.

Exercise 5.5.

(a) Let A, B and C be languages. Show that if $A \leq_T B$ and $B \leq_T C$, then $A \leq_T C$.

(b) Give a counter-example for the following claim: if $A \leq_T B$ then $B \leq_T A$.

Definition 5.6. We define the following languages:

$$\text{ACCEPTS} = \{\langle M, x\rangle : M \text{ is a TM that accepts the input } x\},$$

$$\text{EMPTY} = \{\langle M\rangle : M \text{ is a TM with } L(M) = \emptyset\},$$

$$\text{EQ} = \{\langle M_1, M_2\rangle : M_1 \text{ and } M_2 \text{ are TMs with } L(M_1) = L(M_2)\}.$$

■

Theorem 5.7. *The language* ACCEPTS *is undecidable.*

Proof. Since HALT is undecidable (Theorem 5.3), by the definition of a Turing reduction, it suffices to show HALT $\leq_T$ ACCEPTS. To show the reduction, we assume that the language ACCEPTS is decidable, and then show how to decide HALT. Let M_{ACCEPTS} be a decider for ACCEPTS. Here is our decider for HALT:

```
 1.   "On input <M,x>:
 2.        run M_ACCEPTS(<M,x>).
 3.        if it accepts, accept.
 4.
 5.        Construct TM M' by flipping the accept and reject states of M.
 6.
 7.        run M_ACCEPTS(<M',x>).
 8.        if it accepts, accept.
 9.        if it rejects, reject.
10.   "
```

We now argue that this machine indeed decides HALT. To do this, we'll show that no matter what input is given to our machine, it always gives the correct answer.

First let's assume we get any input $\langle M, x\rangle$ such that $\langle M, x\rangle \in$ HALT. In this case our machine is supposed to accept. Since $M(x)$ halts, we know that $M(x)$ either ends up in the accepting state, or it ends up in the rejecting state. If it ends up in the accepting state, then $M_{\text{ACCEPTS}}(\langle M, x\rangle)$ accepts (on line 2 of our machine's description), and so our program accepts and gives the correct answer. If on the other hand, $M(x)$ ends up in the rejecting state, then $M'(x)$ ends up in the accepting state. Therefore $M_{\text{ACCEPTS}}(\langle M', x\rangle)$ accepts (on line 7 of our machine's description), and so our program accepts and gives the correct answer.

Now let's assume we get any input $\langle M, x\rangle$ such that $\langle M, x\rangle \notin$ HALT. In this case our machine is supposed to reject. Since $M(x)$ does not halt, it never reaches the accepting or the rejecting state. By the construction of M', this also implies that $M'(x)$ never reaches the accepting or the rejecting state. Therefore first $M_{\text{ACCEPTS}}(\langle M, x\rangle)$ (on line 2 of our machine's description) will reject. And then $M_{\text{ACCEPTS}}(\langle M', x\rangle)$ (on line 7 of our machine's description) will reject. Thus our program will reject as well, and give the correct answer.

We have shown that no matter what the input is, our machine gives the correct answer and decides HALT. This completes the proof. □

Theorem 5.8. *The language* EMPTY *is undecidable.*

Proof. Since ACCEPTS is undecidable (Theorem 5.7), it suffices to show ACCEPTS $\leq_T$ EMPTY. So let's assume that M_{EMPTY} is a decider for the language EMPTY. We construct a TM that decides ACCEPTS as follows.

```
 1.   "On input <M,x>:
 2.       Construct the following TM M'.
 3.       "On input y:
 4.           if y != x, reject.
 5.           run M(y).
 6.           if it accepts, accept.
 7.           if it rejects, reject.
 8.       "
 9.
10.       run M_EMPTY(<M'>).
11.       if it accepts, reject.
12.       if it rejects, accept.
13.   "
```

We now argue that this machine indeed decides ACCEPTS. To do this, we'll show that no matter what input is given to our machine, it always gives the correct answer.

First let's assume we get an input $\langle M, x\rangle$ such that $\langle M, x\rangle \in$ ACCEPTS, i.e. $x \in L(M)$. Then observe that $L(M') = \{x\}$, because M' first eliminates/rejects every string not equal to x, and then behaves exactly the same as M. So since $x \in L(M)$, $L(M') = \{x\}$. When we run $M_{\text{EMPTY}}(\langle M'\rangle)$ on line 10, it rejects, and so our machine accepts and gives the correct answer.

Now assume that we get an input $\langle M, x\rangle$ such that $\langle M, x\rangle \notin$ ACCEPTS, i.e. $x \notin L(M)$. Then M' does not accept any string, so $L(M') = \emptyset$. When we run $M_{\text{EMPTY}}(\langle M'\rangle)$ on line 10, it accepts, and so our machine rejects and gives the correct answer.

Our machine always gives the correct answer, so we are done. □

Theorem 5.9. *The language* EQ *is undecidable.*

Proof. Since EMPTY is undecidable (Theorem 5.8), it suffices to show that EMPTY $\leq_T$ EQ. So let's assume that M_{EQ} is a decider for the language EQ. We construct a TM that decides EMPTY as follows.

```
"On input <M>:
    Construct TM M' that rejects every input.

    run M_EQ(<M, M'>).
    if it accepts, accept.
    if it rejects, reject.
"
```

It is not difficult to see that this machine indeed decides EMPTY. Notice that $L(M') = \emptyset$. So when we run $M_{\text{EQ}}(\langle M, M' \rangle)$ on line 4, we are deciding whether $L(M) = L(M')$, i.e. whether $L(M) = \emptyset$. $\square$

Theorem 5.10. HALT $\leq_T$ EMPTY.

Proof. This can be considered as an alternative proof of Theorem 5.8. Assume that M_{EMPTY} is a decider for the language EMPTY. We construct a TM that decides HALT as follows.

```
 1.   "On input <M,x>:
 2.       Construct the following TM M'.
 3.       "On input y:
 4.           run M(x).
 5.           ignore the output and accept.
 6.       "
 7.
 8.       run M_EMPTY(<M'>).
 9.       if it accepts, reject.
10.       if it rejects, accept.
11.   "
```

We now argue that this machine indeed decides HALT. First consider an input $\langle M, x \rangle$ such that $\langle M, x \rangle \in$ HALT. Then $L(M') = \Sigma^*$ since in this case M' accepts every string. So when we run $M_{\text{EMPTY}}(\langle M' \rangle)$ on line 8, it rejects, and our machine accepts and gives the correct answer.

Now consider an input $\langle M, x \rangle$ such that $\langle M, x \rangle \notin$ HALT. Then notice that whatever input is given to M', it gets stuck in an infinite loop when it runs $M(x)$. Therefore $L(M') = \emptyset$. So when we run $M_{\text{EMPTY}}(\langle M' \rangle)$ on line 8, it accepts, and our machine rejects and gives the correct answer. $\square$

Theorem 5.11. HALT $\leq_T$ EQ.

Proof. This can be considered as an alternative proof of Theorem 5.9. Assume that M_{EQ} is a decider for the language EQ. We construct a TM that decides HALT as follows.

```
"On input <M,x>:
    Construct TM M’ that rejects every input.

    Construct the following TM M’’.
    "On input y:
        run M(x).
        ignore the output and accept.
    "

    run M_EQ(<M’, M’’>).
    if it accepts, reject.
    if it rejects, accept.
"
```

We now argue that this machine indeed decides HALT. Notice that no matter what the input is, $L(M') = \emptyset$. Let's first consider an input $\langle M, x\rangle$ such that $\langle M, x\rangle \in$ HALT. Then M'' accepts every input, so $L(M') = \Sigma^*$. In this case, $M_{\text{EQ}}(\langle M', M''\rangle)$ rejects, and so our machine accepts as desired. Next, consider an input $\langle M, x\rangle$ such that $\langle M, x\rangle \notin$ HALT. Then whatever input is given to M'', it gets stuck in an infinite loop when it runs $M(x)$. So $L(M'') = \emptyset$. In this case $M_{\text{EQ}}(\langle M', M''\rangle)$ accepts, and so our machine rejects, as it should. Thus we have a correct decider for HALT. □

Exercise 5.12. Show that the following languages are undecidable.

(a) REGULAR $= \{\langle M\rangle : M$ is a TM such that $L(M)$ is regular.$\}$

(b) FINITE $= \{\langle M\rangle : M$ is a TM that accepts finitely many strings.$\}$

6 Time Complexity

Definition 6.1 (Worst-case running time of an algorithm)**.** [11]
The *worst-case running time* of an algorithm A is a function $T_A : \mathbb{N} \to \mathbb{N}$ defined by

$$T_A(n) = \max_{\substack{\text{instances/inputs } I \\ \text{of size } n}} \text{number of steps } A \text{ takes on input } I.$$

We drop the subscript A and just write $T(n)$ when A is clear from the context. ■

Definition 6.2 (Big-Oh)**.**
For $f : \mathbb{R}^+ \to \mathbb{R}^+$ and $g : \mathbb{R}^+ \to \mathbb{R}^+$, we write $f(n) = O(g(n))$ if there exists constants $C > 0$ and $n_0 > 0$ such that for all $n \geq n_0$,

$$f(n) \leq Cg(n).$$

In this case, we say that $f(n)$ *is big-oh of* $g(n)$. ■

Definition 6.3 (Big-Omega)**.**
For $f : \mathbb{R}^+ \to \mathbb{R}^+$ and $g : \mathbb{R}^+ \to \mathbb{R}^+$, we write $f(n) = \Omega(g(n))$ if there exists constants $c > 0$ and $n_0 > 0$ such that for all $n \geq n_0$,

$$f(n) \geq cg(n).$$

In this case, we say that $f(n)$ *is big-omega of* $g(n)$. ■

Definition 6.4 (Theta)**.**
For $f : \mathbb{R}^+ \to \mathbb{R}^+$ and $g : \mathbb{R}^+ \to \mathbb{R}^+$, we write $f(n) = \Theta(g(n))$ if

$$f(n) = O(g(n)) \qquad \text{and} \qquad f(n) = \Omega(g(n)).$$

This is equivalent to saying that there exists constants $c, C, n_0 > 0$ such that for all $n \geq n_0$,

$$cg(n) \leq f(n) \leq Cg(n).$$

In this case, we say that $f(n)$ *is theta of* $g(n)$.[12] ■

Proposition 6.5. *For any constant $b > 1$,*

$$\log_b n = \Theta(\log n).$$

[11]To make this definition more concrete, one should make explicit the specific computational model being considered. If we leave the exact model unspecified, then we'll work with our intuitive notion of an algorithm and our intuitive understanding of what constitutes a *step* in the algorithm.

[12]The reason we don't call it big-theta is that there is no separate notion of little-theta, whereas little-oh $o(\cdot)$ and little-omega $\omega(\cdot)$ have meanings separate from big-oh and big-omega. We don't cover little-oh and little-omega in this course.

Proof. It is well known that $\log_b n = \frac{\log_a n}{\log_a b}$. In particular $\log_b n = \frac{\log_2 n}{\log_2 b}$. Then taking $c = C = \frac{1}{\log_2 b}$ and $n_0 = 1$, we see that $c \log_2 n \leq \log_b n \leq C \log_2 n$ for all $n \geq n_0$. Therefore $\log_b n = \Theta(\log_2 n)$. □

Remark. Since the base of a logarithm only changes the value of the log function by a constant factor, it is usually not relevant in big-oh, big-omega or theta notation. So most of the time, when you see a log function present inside $O(\cdot)$, $\Omega(\cdot)$, or $\Theta(\cdot)$, the base will be ignored. E.g. instead of writing $\ln n = \Theta(\log_2 n)$, we actually write $\ln n = \Theta(\log n)$. That being said, if the log appears in the exponent, the base matters. For example, $n^{\log_2 5}$ is asymptotically different from $n^{\log_3 5}$.

Proposition 6.6.

$$\log_2 n! = \Theta(n \log n).$$

Proof. We first show $\log_2 n! = O(n \log n)$. Observe that

$$n! \quad = \quad n \cdot (n-1) \cdot (n-2) \cdot \cdots \cdot 2 \cdot 1 \quad \leq \quad \underbrace{n \cdot n \cdot n \cdot \cdots \cdot n}_{n \text{ times}} \quad = \quad n^n.$$

Taking the log of both sides gives us $\log_2 n! \leq \log_2 n^n = n \log_2 n$ (here, we are using the fact that $\log a^b = b \log a$). Therefore taking $n_0 = C = 1$ satisfies the definition of big-oh, and $\log_2 n! = O(n \log n)$.

Now we show $\log_2 n! = \Omega(n \log n)$. Assume without loss of generality that n is even. In the definition of $n!$, we'll use the first $n/2$ terms in the product to lower bound it:

$$n! \quad = \quad n \cdot (n-1) \cdot (n-2) \cdot \cdots \cdot 2 \cdot 1 \quad \geq \quad \underbrace{\frac{n}{2} \cdot \frac{n}{2} \cdot \cdots \cdot \frac{n}{2}}_{n/2 \text{ times}} \quad = \quad \left(\frac{n}{2}\right)^{\frac{n}{2}}.$$

Taking the log of both sides gives us $\log_2 n! \geq \frac{n}{2} \log_2 \frac{n}{2}$.

Claim: For $n \geq 4$, $\frac{n}{2} \log_2 \frac{n}{2} \geq \frac{n}{4} \log_2 n$.

The proof of the claim is a simple exercise which we omit. Using the claim, we know that for $n \geq 4$, $\log_2 n! \geq \frac{n}{4} \log_2 n$. Therefore, taking $n_0 = 4$ and $c = 1/4$ satisfies the definition of big-omega, and $\log_2 n! = \Omega(n \log n)$. □

Exercise 6.7. Show that $n!^2$ is $\Omega(n^n)$ but $n!^2$ is not $O(n^n)$.

Definition 6.8 (Names for common growth rates)**.**

Constant time:	$T(n) = O(1)$.
Logarithmic time:	$T(n) = O(\log n)$.
Linear time:	$T(n) = O(n)$.
Quadratic time:	$T(n) = O(n^2)$.
Polynomial time:	$T(n) = O(n^k)$ for some constant $k > 0$.
Exponential time:	$T(n) = O(2^{n^k})$ for some constant $k > 0$.

■

Exercise 6.9. Consider the following computational problem: Given as input a positive integer N, output $N!$. Show that this problem cannot be computed in polynomial-time, i.e. $O(n^k)$ time for some constant k, where n denotes the number of bits in the binary representation of the input.

7 Cake Cutting

Definition 7.1 (Cake cutting problem)**.**
We refer to the interval $[0,1] \subset \mathbb{R}$ as the *cake*, and the set $N = \{1, 2, \ldots, n\}$ as the set of *players*. A *piece of cake* is any set $X \subseteq [0,1]$ which is a finite union of disjoint intervals. Let $\mathcal{X}$ denote the set of all possible pieces of cake. Each player $i \in N$ has a valuation function $V_i : \mathcal{X} \to \mathbb{R}$ that satisfies the following 4 properties.

- **Normalized:** $V_i([0,1]) = 1$.
- **Non-negative:** For any $X \in \mathcal{X}$, $V_i(X) \geq 0$.
- **Additive:** For $X, Y \in \mathcal{X}$ with $X \cap Y = \emptyset$, $V_i(X \cup Y) = V_i(X) + V_i(Y)$.
- **Divisible:** For every interval $I \subseteq [0,1]$ and $0 \leq \lambda \leq 1$, there exists a subinterval $I' \subseteq I$ such that $V_i(I') = \lambda V_i(I)$.

The goal is to find an *allocation* $A_1, A_2, \ldots, A_n$, where each A_i is a piece of cake allocated to player i. The allocation is assumed to be a partition of the cake $[0,1]$, i.e., the A_i's are disjoint and their union is $[0,1]$. There are 2 properties desired about the allocation:

- **Proportionality:** For all $i \in N$, $V_i(A_i) \geq 1/n$.
- **Envy-Freeness:** For all $i, j \in N$, $V_i(A_i) \geq V_i(A_j)$.

■

Proposition 7.2 (Envy-freeness implies proportionality)**.** *If an allocation is envy-free, then it is proportional.*

Proof. Let's assume we have an allocation $A_1, \ldots, A_n$ that is envy-free. Recall that the A_i's form a partition of $[0,1]$. Fix some player i. Notice that the additivity and normality properties imply that $\sum_{j \in N} V_i(A_j) = 1$. Therefore, there must be $k \in N$ such that $V_i(A_k) \geq 1/n$ (otherwise the sum could not be 1). The envy-freeness property implies that $V_i(A_i) \geq V_i(A_k)$, and so $V_i(A_i) \geq 1/n$. This is true for all players i, so the allocation must be proportional. □

Definition 7.3 (The Robertson-Webb model for measuring time complexity)**.**
We measure the time complexity of an algorithm that solves the cake cutting problem using the Robertson-Webb model. In this model, the input size is considered to be the number of players n. There is a referee who is allowed to make two types of queries to the players:

- $\text{Eval}_i(x, y)$, which returns $V_i([x, y])$,
- $\text{Cut}_i(x, \alpha)$, which returns y such that $V_i([x, y]) = \alpha$.
  (If no such y exists, it returns "None".)

The referee follows an *algorithm/strategy* and chooses the queries that she wants to make. Afterwards, she must decide on an allocation $A_1, A_2, \ldots, A_n$. The *time complexity* of the algorithm is the number of queries she makes for n players and the worst possible V_i's. So

$$T(n) = \max_{(V_1,\ldots,V_n)} \text{ number of queries when the valuations are } (V_1, \ldots, V_n).$$

■

Proposition 7.4 (Cut and Choose algorithm for 2 players)**.** *When $n = 2$, there is always an allocation that is proportional and envy-free.*

Proof. The idea is easy to describe in the following way. The first player marks a point y in the cake so that $V_1([0, y]) = V_1([y, 1]) = 1/2$ (this can be done because of the divisibility property). Then player 2 chooses the piece (among $[0, y]$ and $[y, 1]$) that he values more. The remaining piece is what player 1 gets. In the Robertson-Webb model, this algorithm corresponds to the following. The referee first queries $\text{Cut}_1(0, 1/2)$. Say this returns the value y. Then the referee queries $\text{Eval}_2(0, y)$ and $\text{Eval}_2(y, 1)$.[13] Whichever gives the larger value, referee assigns that piece to player 2. The remaining piece is assigned to player 1.

This algorithm is envy-free: From player 1's perspective, both players get a piece of the cake of value 1/2. Therefore $V_1(A_1) \geq V_1(A_2)$ is satisfied. From player 2's perspective, since he gets to choose the piece of larger value to him, $V_2(A_2) \geq V_2(A_1)$ is satisfied.

It is also pretty clear that the algorithm is proportional since each player gets a piece of value at least 1/2. □

Theorem 7.5 (Dubins-Spanier algorithm)**.** *There is an algorithm of time complexity $\Theta(n^2)$ that produces an allocation for the cake cutting problem that satisfies the proportionality property.*

Proof. The algorithm is as follows. The referee first makes n queries: $\text{Cut}_i(0, 1/n)$ for all i. She computes the minimum among these values, which we'll denote by y. Let's assume j is the player that corresponds to the minimum value. Then the referee assigns $A_j = [0, y]$. So player j gets a piece that she values at $1/n$. After this, we remove player j, and repeat the process on the remaining cake. So then the referee makes $n-1$ queries, $\text{Cut}_i(y, 1/n)$ for $i \neq j$, figures out the player corresponding to the minimum value, and assigns her the corresponding piece of the cake, which she values at $1/n$. This repeats until there is one player left. The last player gets the piece that is left.

We have to show that the algorithm's time complexity is $\Theta(n^2)$ and that it produces a proportional allocation. First we show that the allocation is proportional. Notice that if the queries that the referee makes never return "None", then at each iteration, until one player is left, the player j who is removed is assigned A_j such that $V_j(A_j) = 1/n$. So it suffices to argue that:

[13] In fact, just querying $\text{Eval}_2(0, y)$ is enough.

(i) the queries never return "None",

(ii) the last player, call it ℓ, gets A_ℓ such that $V_\ell(A_\ell) \geq 1/n$.

To show (i), assume we have just completed iteration k, where $k \in \{1, 2, \dots, n-1\}$. Let j be one of the players who has not been removed yet. Observe that all the pieces that have been removed so far have value at most $1/n$ to player j. So the cake remaining after iteration k has value at least $1 - (k/n) \geq 1/n$ for player j. This argument holds for any $k \in \{1, 2, \dots, n-1\}$ and any player j that remains after iteration k. So the queries never return "None". Part (ii) actually follows from the same argument. The cake remaining after iteration $n-1$ has value at least $1 - (n-1)/n = 1/n$ for the last player. This completes the proof that the allocation is proportional.

Now we show that the time complexity is $\Theta(n^2)$. To do this, we'll first argue that the number of queries is $O(n^2)$, and then argue that it is $\Omega(n^2)$. Note that the algorithm has n iterations, and at iteration i, it makes $n + 1 - i$ queries. There is one exception, which is the last iteration when only one player is left. In that case, we don't make any queries. So the total number of queries is

$$n + (n-1) + (n-2) + \cdots + 2.$$

We can upper bound this as follows:

$$n + (n-1) + (n-2) + \cdots + 2 \;\leq\; \underbrace{n + n + \cdots + n}_{n \text{ times}} \;=\; n^2.$$

This implies that the number of queries is $O(n^2)$. We can also lower bound the number of queries by lower bounding the first $n/2$ terms in the sum by $n/2$:

$$n + (n-1) + (n-2) + \cdots + 2 \;\geq\; \underbrace{\frac{n}{2} + \frac{n}{2} + \cdots + \frac{n}{2}}_{n/2 \text{ times}} \;=\; \frac{n^2}{4}.$$

This implies that the number of queries is $\Omega(n^2)$. Hence, the number of queries is $\Theta(n^2)$. $\square$

Exercise 7.6. Design a cake cutting algorithm for a set of players $N = \{1, \dots, n\}$ that finds an allocation A with the property that there exists a permutation $\pi_A : N \to N$ such that for all $i \in N$, $V_i(A_i) \geq \frac{1}{2^{\pi(i)}}$. In words, there is an order on the players such that the first player has value at least $1/2$ for her piece, the second player has value at least $1/4$, and so on. The complexity of your algorithm in the Robertson-Webb model should be $O(n^2)$.

Theorem 7.7 (Even-Paz algorithm)**.** *Assume n is a power of 2, i.e., $n = 2^t$ for some $t \in \mathbb{N}$. There is an algorithm of time complexity $\Theta(n \log n)$ that produces an allocation for the cake cutting problem that satisfies the proportionality property.*

Proof. Our algorithm will be recursive, so we give some flexibility for the input by allowing it to consist of an interval $[y, z] \subseteq [0, 1]$ and a subset of players $S \subseteq \{1, 2, \ldots, n\}$. Our algorithm's name is EP, and we would initially call it with the input $([0, 1], \{1, 2, \ldots, n\})$. Below is the description of EP. A verbal explanation of what the algorithm does follows its description.

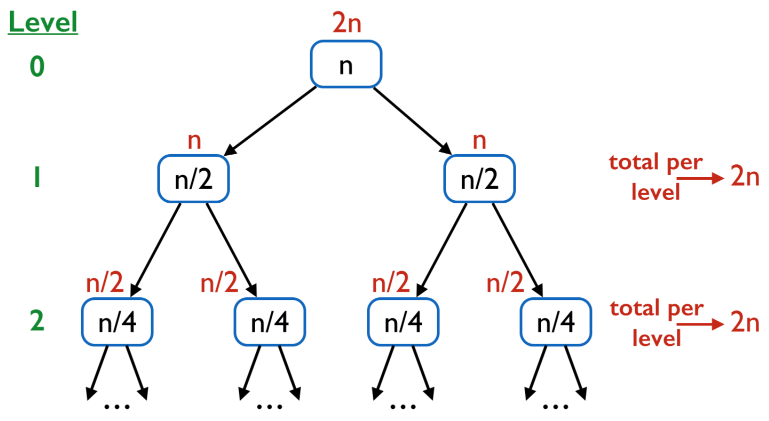
```
1.  EP on input $([x, y], S)$ where $|S| = k$:
2.
3.      If $S = \{i\}$ for some $i$, then assign $[x, y]$ to $A_i$.
4.
5.      Else:
6.          For $i \in S$, let $z_i = \text{Cut}_i(x, \text{Eval}_i(x, y)/2)$.
7.          Sort the $z_i$ so that $z_{i_1} \le z_{i_2} \le \cdots \le z_{i_k}$. Let $z^* = z_{i_{k/2}}$.
8.          Run EP$([x, z^*], \{i_1, \ldots, i_{k/2}\})$.
9.          Run EP$([z^*, y], \{i_{k/2+1}, \ldots, i_k\})$.
```

The base case of the algorithm is when there is only one player. In this case we give the whole piece $[x, y]$ to that player. Otherwise, each player i makes a mark z_i such that $V_i([x, z_i]) = \frac{1}{2}V_i([x, y])$. Let z^* denote the $n/2$ mark from the left. We first recurse on $[x, z^*]$ and the left $n/2$ players, and then we recurse on $[z^*, y]$ and the right $n/2$ players.

We have to show that the algorithm's time complexity is $\Theta(n \log n)$ and that it produces a proportional allocation. First we show that the time complexity $T(n)$ is $\Theta(n \log n)$. Observe that the recursive relation that $T(n)$ satisfies is

$$T(1) = 0, \qquad T(n) = 2n + 2T(n/2) \quad \text{for } n > 1.$$

The base case corresponds to line 3 of the algorithm, and in this case, we don't make any queries. In $T(n) = 2n+2T(n/2)$, the $2n$ comes from line 6 where we make 2 queries for each player. The $2T(n/2)$ comes from the two recursive calls on lines 8 and 9. To solve the recursion, i.e., to figure out the formula for $T(n)$, we draw the associated *recursion tree.*



The root (top) of the tree corresponds to the input $S = \{1, 2, \ldots, n\}$ and is therefore labeled with an n. This branches off into two nodes, one corresponding to each recursive call. These nodes are labeled with $n/2$ since they correspond to recursive calls in which $|S| = n/2$. Those nodes further branch off into two nodes, and so on, until at the very bottom, we end up with nodes corresponding to inputs S with $|S| = 1$. The number of queries made for each node of the tree is provided with a label on top of the node. For example, at the root (top), we make $2n$ queries before we do our recursive calls. This is why we put a $2n$ on top of that node. Similarly, every other node can be labeled. We can divide the nodes of the tree into *levels* according to how far a node is from the root. So the root corresponds to level 0, the nodes it branches off to correspond to level 1, and so on. Observe that level j has exactly 2^j nodes. The nodes that are at level j make $2n/2^j$ queries. Therefore, the total number of queries made for level j is $2n$. The only exception is the last level. The nodes at the last level correspond to the base case and don't make any queries. In total, there are exactly $1 + \log_2 n$ levels (since we are counting the root as well). Thus, the total number of queries, and hence the time complexity, is exactly $2n \log_2 n$, which is $\Theta(n \log n)$.

We now prove that the allocation obtained by the algorithm is proportional. Observe that when we make the recursive call on $[x, z^*]$ and the left $n/2$ players, all these players value $[x, z^*]$ at least at $1/2$. Similarly, when we make the recursive call on $[z^*, y]$ and the right $n/2$ players, all these players value $[z^*, y]$ at least at $1/2$. This property is preserved at each level of the recursion in the following way. At level ℓ of the recursion, the players are divided into groups of size $n/2^\ell$. If each player values the corresponding interval at least at $1/2^\ell$, then at level $\ell + 1$, the players will value the interval that they are "assigned to" at least at $1/2^{\ell+1}$. In particular, when $\ell = \log_2 n$, each group is a singleton, and each player gets assigned a piece of cake that she values at least at $1/2^{\log_2 n} = 1/n$. This shows that the allocation is proportional. $\square$

Theorem 7.8 (Edmonds-Pruhs, 2006)**.** *Any algorithm that produces an allocation satisfying the proportionality property must have time complexity $\Omega(n \log n)$.*

8 Boolean Circuits

Notation 8.1. For a finite alphabet Σ, Σ^n denotes all words in Σ^* with length (i.e. number of symbols) exactly n. Let $f : \{0,1\}^* \to \{0,1\}$ be a decision problem. We denote by $f^n : \{0,1\}^n \to \{0,1\}$ the restriction of f to words of length n. So f can be thought of as a collection of functions $(f^0, f^1, f^2, \ldots)$. We'll write $f = (f^0, f^1, f^2, \ldots)$ as a shorthand for this.

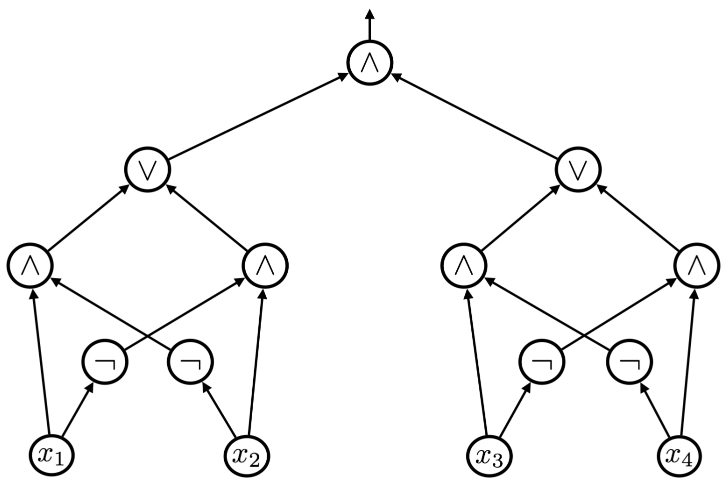
Notation 8.2. We denote by $\neg$ the unary NOT operation, by $\wedge$ the binary AND operation, and by $\vee$ the binary OR operation. In particular, we can write the truth tables of these operations as follows:

x	$\neg x$
0	1
1	0

x	y	$x \wedge y$
0	0	0
0	1	0
1	0	0
1	1	1

x	y	$x \vee y$
0	0	0
0	1	1
1	0	1
1	1	1

Definition 8.3 (Boolean circuit).
A Boolean circuit with n-input variables is a finite collection of 4 types of *gates*: AND gates, OR gates, NOT gates, and input gates. There are n input gates, one corresponding to each input variable. The AND gate corresponds to the binary AND operation $\wedge$, the OR gate corresponds to the binary OR operation $\vee$, and the NOT gate corresponds to the unary NOT operation $\neg$. The gates are linked together as follows. Each binary gate is connected to two other gates, which are considered to be the inputs for the gate. A NOT gate is connected to one other gate, which is considered to be the input for the NOT gate. One of the gates in the circuit is labeled as the *output gate*. The circuit is not allowed to have any cycles in the following sense: there cannot be a sequence of gates $g_1, g_2, \ldots, g_k, g_{k+1}$ such that $g_1 = g_{k+1}$ and for each $i \in \{1, 2, \ldots, k\}$, g_i is an input to g_{i+1}. Below is an example of how we draw a circuit ($n = 4$).



For each 0/1 assignment to the input variables, the Boolean circuit produces a one-bit output. The output of the circuit is the output of the gate that is labeled as the *output gate* (in the picture, it is the one at the very top with an arrow that links to nothing). The output is calculated naturally using the truth tables of the operations corresponding to the gates. The input-output behavior of the circuit defines a function $f : \{0,1\}^n \to \{0,1\}$ and in this case, we say that the circuit *computes* this function. The circuit in the above example computes the parity of the input bits, i.e., it computes $(x_1 + x_2 + x_3 + x_4) \bmod 2$. ■

Exercise 8.4. Draw a circuit that computes the following functions.

(a) The parity function $\mathrm{PAR} : \{0,1\}^2 \to \{0,1\}$ on 2 variables, which is defined as $\mathrm{PAR}(x_1, x_2) = 1$ iff $x_1 + x_2$ is odd.

(b) The majority function $\mathrm{MAJ} : \{0,1\}^3 \to \{0,1\}$ on 3 variables, which is defined as $\mathrm{MAJ}(x_1, x_2, x_3) = 1$ iff $x_1 + x_2 + x_3 \geq 2$.

Exercise 8.5. Define a NAND gate as $\mathrm{NAND}(x, y) = \neg(x \wedge y)$. Show that any circuit with AND, OR and NOT gates can be converted into an equivalent circuit (i.e. a circuit computing the same function) that uses *only* NAND gates. The size of this circuit should be at most a constant times the size of the original circuit.

Definition 8.6 (Circuit family)**.**
A *circuit family* C is a collection of circuits, $(C_0, C_1, C_2, \ldots)$, such that each C_n is a circuit with n input gates. ■

Definition 8.7 (A circuit family deciding/computing a decision problem)**.**
Let $f : \{0,1\}^* \to \{0,1\}$ be a decision problem and let $f^n : \{0,1\}^n \to \{0,1\}$ be the restriction of f to words of length n. We say that a circuit family $C = (C_0, C_1, C_2, \ldots)$ *decides/computes* f if C_n computes f^n for every n. ■

Definition 8.8 (Circuit size and complexity)**.**
The size of a circuit is defined to be the number of gates in the circuit. The size of a circuit family $C = (C_0, C_1, C_2, \ldots)$ is a function $S(\cdot)$ such that $S(n)$ is the size of C_n. The *circuit complexity* of a decision problem $f = (f^0, f^1, f^2, \ldots)$ is the size of the minimal circuit family that decides f. In other words, the circuit complexity of f is defined to be a function $S(\cdot)$ such that $S(n)$ is the minimum size of a circuit computing f^n. ■

Exercise 8.9. Let $L \subseteq \{0,1\}^*$ be the set of words which contain an odd number of 1's. Show that the circuit complexity of L is $\Theta(n)$.

Theorem 8.10. *Any language $L \subseteq \{0,1\}^*$ can be computed by a circuit family of size $O(2^n)$.*

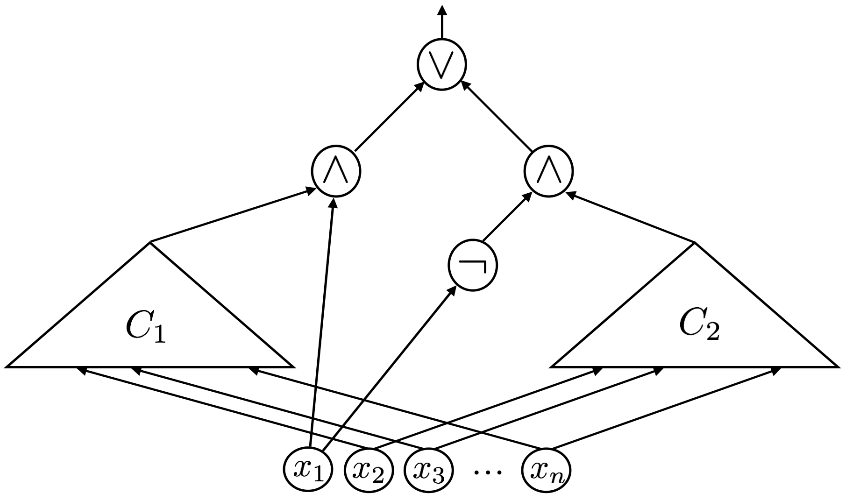
Proof. Let

$$S(n) = \max_{f:\{0,1\}^n \to \{0,1\}} \text{size of the smallest circuit computing } f.$$

Observe that the theorem follows once we show that $S(n) = O(2^n)$. Take any function $f : \{0,1\}^n \to \{0,1\}$. Notice that we can write

$$f(x_1, x_2, \ldots, x_n) = (x_1 \wedge f(1, x_2, \ldots, x_n)) \vee (\neg x_1 \wedge f(0, x_2, \ldots, x_n)).$$

Let C_1 be the smallest size circuit that computes $f(1, x_2, \ldots, x_n)$ and let C_2 be the smallest size circuit that computes $f(0, x_2, \ldots, x_n)$. We can then construct a circuit for $f(x_1, x_2, \ldots, x_n)$ as shown in the picture below.



The circuits C_1 and C_2 compute functions on $n-1$ variables, so their size is bounded by $S(n-1)$ each. Then the size of the above circuit is bounded above by $2S(n-1)+5$. So we can conclude that $S(n) \le 2S(n-1)+5$. In the base case, when there is just one variable, there are four different functions: $f(x) = x$, which requires only 1 gate, $f(x) = \neg x$, which requires only 2 gates, $f(x) = 1$, which requires only 3 gates, and $f(x) = 0$, which requires only 3 gates. Therefore $S(1) \le 3$. It is then easy to solve the recurrence and verify that $S(n) = O(2^n)$ (we omit this part of the proof). □

Proposition 8.11. *The set of all functions of the form $f : \{0,1\}^n \to \{0,1\}$ has size 2^{2^n}.*

Proof. A function of the form $f : \{0,1\}^n \to \{0,1\}$ has 2^n possible inputs. For each input, we have 2 choices for the output, either 0 or 1. Therefore we have 2^{2^n} different functions. □

Theorem 8.12. *There exists a language $L \subseteq \{0,1\}^*$ such that any circuit family computing L must have size at least $2^n/4n$.*

Proof. Observe that the theorem follows once we show that there is some function $f : \{0,1\}^n \to \{0,1\}$ which cannot be computed by a circuit of size less than $2^n/4n$. We'll do this by showing that the number of circuits of size less than $2^n/4n$ is strictly less than the total number of functions $f : \{0,1\}^n \to \{0,1\}$. Since one circuit computes one function, this implies that there are not enough circuits of size less than $2^n/4n$ to compute every possible function. So there exists at least one function which cannot be computed by a circuit of size less than $2^n/4n$.

From the previous proposition, we know that the total number of functions $f : \{0,1\}^n \to \{0,1\}$ is 2^{2^n}. In the next lemma (Lemma 8.13), we show that the number of possible circuits of size at most s is less than or equal to $2^{4s \log s}$. It is an easy exercise (which we leave to the reader) to confirm that for $s \le 2^n/4n$, $2^{4s \log s} < 2^{2^n}$. In other words, for $s \le 2^n/4n$, there are more functions than circuits, and the result follows. □

Lemma 8.13. *The number of possible circuits of size at most s is less than or equal to $2^{4s \log s}$.*

Proof. Let A be the set of circuits of size at most s, and let $B = \{0,1\}^{4s \log s}$. Recall that $|A| \le |B|$ if and only if there is a surjection from B to A. Since $|B| = 2^{4s \log s}$, we are done once we show there is a surjection from B to A. To show that there is a surjection, we show how to encode a circuit of size at most s with a binary string of length $4s \log s$. The encoding is as follows. Number the gates of the circuit $1, 2, \ldots, s$. Note that it takes $\log_2 s$ bits to write down the number of a gate. We'll assume that the first gate corresponds to the output gate. For each gate of the circuit, write down in binary:

(i) type of the gate (input, OR, AND, NOT),

(ii) from which gates the inputs are coming from.

Once we know (i) and (ii) for every gate, we have all the information to reconstruct the circuit. Note that (i) takes 2 bits to specify, and (ii) takes $2 \log s$ bits. Since we do this for each gate in the circuit, the total number of bits is $s(2 + 2\log s)$, which can be upper bounded by $4s \log s$. □

Theorem 8.14. *Let $L \subseteq \{0,1\}^*$ be a language which can be decided in $O(T(n))$ time. Then L can be computed by a circuit family of size $O(T(n)^2)$.*

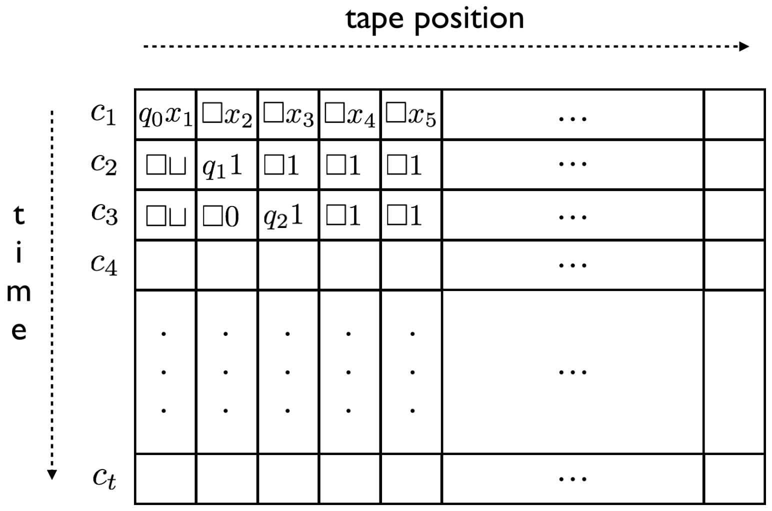
Proof Sketch. [14] Warning: This is only a sketch of the proof. You will not be responsible for this proof sketch.

Let L be decided by a TM M in $O(T(n))$ time. For simplicity, we will assume that M's tape is infinite in one direction (to the right). In fact, often the tape of a TM is defined to be one-way infinite rather than two-way infinite.

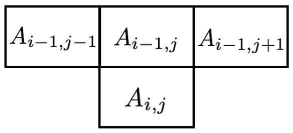
[14]The diagrams in this proof are redrawings of the ones in https://lucatrevisan.wordpress.com/2010/04/25/cs254-lecture-3-boolean-circuits/.

Fix the input length n. We want to design a circuit on n input bits such that for the inputs accepted by M, the circuit will output 1, and for the inputs rejected by M, the circuit will output 0. The size of our circuit will be $O(T(n)^2)$.

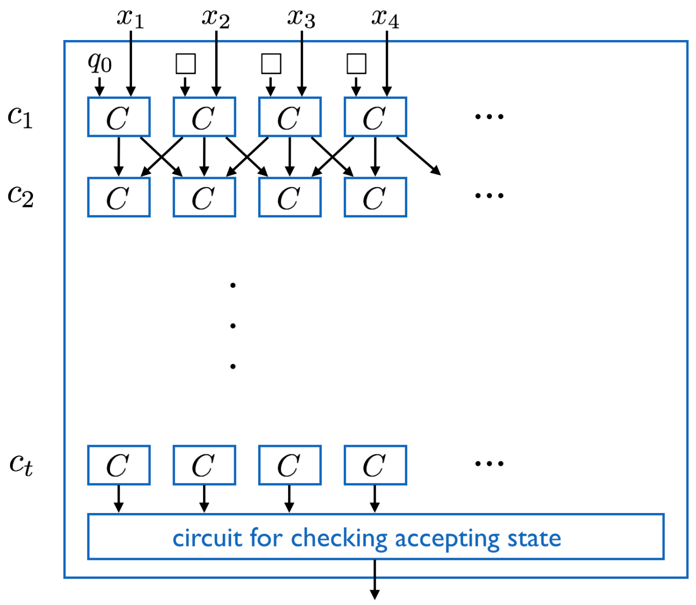
Let's denote the input by $x_1, x_2, \ldots, x_n$. We know that when M runs on this input, it goes through configurations (see Definition 3.3) $c_1, c_2, \ldots, c_t$, where each configuration c_i is of the form uqv for $u, v \in \Gamma^*$, $q \in Q$. Here the number of steps M takes is t so $t = O(T(n))$. Consider a $t \times t$ table where row i corresponds to c_i.



Each cell contains two pieces of information: (i) a state name or NONE (if no state is given), (ii) a symbol from Γ. Let's call this table A. In the table above, NONE is represented with a box $\square$, and the input is assumed to be the all-1 string. Observe that the contents of a cell of the table $A_{i,j}$ are determined by the contents of $A_{i-1,j-1}$, $A_{i-1,j}$ and $A_{i-1,j+1}$.



The transition function of M governs this transformation. Assume each cell encodes k bits of information. Note that k is a constant because $|Q|$ and $|\Gamma|$ are constant. So the transition function of the form $\{0,1\}^{3k} \to \{0,1\}^k$, which determines the contents of a cell based on the contents of the three cells above it, can be implemented by a circuit of constant size (we can allow our circuit to have multiple output gates, one for each output bit of the function). Let's call this circuit C. Now we can build a circuit that computes the answer given by M as shown in the picture below.



The size of the circuit is at most ct^2 for some constant c. $\square$

Definition 8.15 (Complexity class P)**.**
We denote by P the set of all languages that can be decided in polynomial-time, i.e., in time $O(n^k)$ for some constant $k > 0$. ■

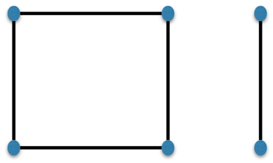
Corollary 8.16. *If $L \in \mathsf{P}$, then L can be computed by a circuit family of polynomial size. Equivalently, if L cannot be computed by a circuit family of polynomial size, then $L \notin \mathsf{P}$.*

9 Graphs I: The Basics

Definition 9.1 (Undirected graph)**.**
An *undirected graph*[15] G is a pair (V, E), where

- V is a finite set called the set of *vertices* (or *nodes*),
- E is a finite set called the set of *edges*, and every element of E is of the form $\{u, v\}$ for distinct $u, v \in V$.

Usually, the size of V is denoted by n and the size of E is denoted by m. Below is an example a graph with $n = 6$ and $m = 5$. We usually draw the vertices as dots, and edges as lines connecting two dots.



We'll assume that a graph is represented/encoded using a matrix called the *adjacency matrix*.[16] Suppose $v_1, v_2, \ldots, v_n$ is some arbitrary ordering of the vertices. In the adjacency matrix representation, a graph is represented by a $n \times n$ matrix A such that

$$A[i,j] = \begin{cases} 1 & \text{if } \{v_i, v_j\} \in E, \\ 0 & \text{otherwise.} \end{cases}$$

■

Exercise 9.2. In an n-vertex graph, what is the maximum possible value for the number of edges in terms of n?

Definition 9.3 (Neighborhood of a vertex)**.**
Let $G = (V, E)$ be a graph, and $e = \{u, v\} \in E$ be an edge in the graph. In this case, we say that u and v are *neighbors* or *adjacent*. We also say that u and v are *incident* to e. For $v \in V$, we define the *neighborhood* of v, denoted $N(v)$, as the set of all neighbors of v, i.e. $N(v) = \{u : \{v, u\} \in E\}$. The size of the neighborhood, $|N(v)|$, is called the *degree* of v, and is denoted by $\deg(v)$. ■

Definition 9.4 (d-regular graphs)**.**
A graph $G = (V, E)$ is called *d-regular* if every vertex $v \in V$ satisfies $\deg(v) = d$. A 1-regular graph is called a *perfect matching*. ■

[15]Often the word "undirected" is omitted.

[16]This is not always the best representation of a graph. In particular, it is wasteful if the graph has very few edges. For such graphs, it can be preferable to use the *adjacency list* representation. In the adjacency list representation, you are given an array of size n and the i'th entry of the array contains a pointer to a linked list of vertex i's neighbors.

Theorem 9.5. *Let $G = (V, E)$ be a graph. Then*

$$\sum_{v \in V} \deg(v) = 2m.$$

Proof. For each vertex $v \in V$, put a "token" on all the edges it is incident to. Every vertex v is incident to $\deg(v)$ edges, so the total number of tokens put is $\sum_{v \in V} \deg(v)$. Now observe that each edge $\{u, v\}$ in the graph will get two tokens, one from vertex u and one from vertex v. So the total number of tokens put is $2m$. Therefore it must be that $\sum_{v \in V} \deg(v) = 2m$. [17] □

Definition 9.6 (Paths and cycles)**.**
Let $G = (V, E)$ be a graph. A *path* of length k in G is a sequence of <u>distinct</u> vertices

$$v_0, v_1, \ldots, v_k$$

such that $\{v_{i-1}, v_i\} \in E$ for all $i \in \{1, 2, \ldots, k\}$. In this case, we say that the path is from vertex v_0 to vertex v_k.
A *cycle* of length k (also known as a k-cycle) in G is a sequence of vertices

$$v_0, v_1, \ldots, v_{k-1}, v_0$$

such that $v_0, v_1, \ldots, v_{k-1}$ is a path, and $\{v_0, v_{k-1}\} \in E$. In other words, a cycle is just a "closed" path. The starting vertex in the cycle is not important. So for example,

$$v_1, v_2, \ldots, v_{k-1}, v_0, v_1$$

would be considered the same cycle. Also, if we list the vertices in reverse order, we consider it to be the same cycle. For example,

$$v_0, v_{k-1}, v_{k-2} \ldots, v_1, v_0$$

represents the same cycle as before.
A graph that contains no cycles is called *acyclic*. ■

Definition 9.7 (Connected graph, connected component)**.**
Let $G = (V, E)$ be a graph. We say that two vertices in G are *connected* if there is a path between those two vertices. We say that G is *connected* if every pair of vertices in G is connected.
A subset $S \subseteq V$ is called a *connected component* of G if G restricted to S, i.e. the graph $G' = (S, E' = \{\{u, v\} \in E : u, v \in S\})$, is a connected graph, and S is disconnected from the rest of the graph (i.e. $\{u, v\} \notin E$ when $u \in S$ and $v \notin S$). Note that a connected graph is a graph with only one connected component. ■

[17] This kind of an argument is called a *double counting argument*. We count a certain set of objects in two different ways to prove an identity.

Theorem 9.8. *Let $G = (V, E)$ be a connected graph with n vertices and m edges. Then $m \geq n - 1$ and*

$$m = n - 1 \iff G \text{ is acyclic.}$$

Proof. Take G and remove all its edges. This graph consists of isolated vertices and therefore contains n connected components. Let's now imagine a process in which we put back the edges of G one by one. The order in which we do this does not matter. At the end of this process, we must end up with just one connected component since G is connected. When we put back an edge, there are two options. Either

(i) we connect two different connected components by putting an edge between two vertices that are not already connected, or

(ii) we put an edge between two vertices that are already connected, and therefore create a cycle.

Observe that if (i) happens, then the number of connected components goes down by 1. If (ii) happens, the number of connected components remains the same. So every time we put back an edge, the number of connected components in the graph can go down by at most 1. Since we start with n connected components and end with 1 connected component, (i) must happen at least $n - 1$ times, and hence $m \geq n - 1$. We now prove $m = n - 1 \iff G$ is acyclic.

$m = n - 1 \implies G$ is acyclic: If $m = n - 1$, then (i) must have happened at each step since otherwise, we could not have ended up with one connected component. Note that (i) cannot create a cycle, so in this case, our original graph must be acyclic.

G is acyclic $\implies m = n - 1$: To prove this direction (using the contrapositive), assume $m > n - 1$. We know that (i) can happen at most $n - 1$ times. So in at least one of the steps, (ii) must happen. This implies G contains a cycle. □

Exercise 9.9. Show that if a graph is acyclic and satisfies $m = n - 1$, then it is connected.

Definition 9.10 (Tree, leaf of a tree)**.**
A connected acyclic graph is called a *tree.*[18] A vertex of degree 1 in a tree is called a *leaf.* ■

Exercise 9.11. Let T be a tree with at least 2 vertices. Show that T must have at least 2 leaves.

Definition 9.12 (Hamiltonian cycle)**.**
Let $G = (V, E)$ be a graph. A *Hamiltonian cycle* in G is a cycle that visits every vertex $v \in V$ exactly once. In other words, a Hamiltonian cycle is a cycle of length n. ■

[18]In fact, a graph having 2 of the following 3 properties is a tree: (i) connected, (ii) $m = n - 1$, (iii) acyclic.

Theorem 9.13 (Ore's Theorem)**.** *Let $G = (V, E)$ be a graph on $n \geq 3$ vertices. Suppose $\deg(u) + \deg(v) \geq n$ for every pair of non-adjacent vertices $u, v \in V$. Then G contains a Hamiltonian cycle.*

Proof. Consider a sequence of vertices

$$v_1, v_2, \ldots, v_n, v_{n+1}$$

where $v_{n+1} = v_1$, and the number of i such that $\{v_i, v_{i+1}\} \in E$ is maximized. That is, there is no other way of ordering the vertices so that more pairs of consecutive vertices form an edge. Note that G has an Hamiltonian cycle if and only if every consecutive pair of vertices in this sequence forms an edge.

The proof is by contradiction, so assume G does not have a Hamiltonian cycle. Without loss of generality, assume the first pair of vertices don't form an edge, i.e. $\{v_1, v_2\} \notin E$. We will derive a contradiction by constructing another sequence of vertices that contains more edges and thus contradicting the maximality of the original sequence.

We define the set S to be the set of *successors* of $N(v_1)$. That is, if $N(v_1) = \{v_{i_1}, v_{i_2}, \ldots, v_{i_k}\}$, then $S = \{v_{i_1+1}, v_{i_2+1}, \ldots, v_{i_k+1}\}$. We know the following are true:

(a) $\deg(v_1) + \deg(v_2) \geq n$ since $\{v_1, v_2\} \notin E$,

(b) $|S| = |N(v_1)|$,

(c) $v_2 \notin S$ since the *predecessor* of v_2 is v_1, but $v_1 \notin N(v_1)$.

These observations allow us to show that:

$$\begin{aligned}
\deg(v_2) &\geq n - \deg(v_1) && \text{(using (a))} \\
&= |V| - |N(v_1)| \\
&= |V| - |S| && \text{(using (b))} \\
&> |V \backslash (S \cup \{v_2\})|. && \text{(using (c))}
\end{aligned}$$

So $\deg(v_2) > |V \backslash (S \cup \{v_2\})|$, which implies $N(v_2)$ must intersect with $S \cup \{v_2\}$. The intersection cannot be v_2, so let's assume it is v_j for some $j \notin \{1, 2\}$. So $v_j \in N(v_2) \cap S$. Thus,

- since $v_j \in N(v_2)$, we know $\{v_j, v_2\} \in E$,
- since $v_j \in S$, we know $\{v_{j-1}, v_1\} \in E$.

Now consider the sequence:

$$v_2, v_3, \ldots, v_{j-1}, v_1, v_n, v_{n-1}, \ldots, v_j, v_2.$$

Let's compare this sequence with our original sequence. The portions $v_2, v_3, \ldots, v_{j-1}$ and $v_1, v_n, v_{n-1}, \ldots, v_j$ are shared by both sequences (even though the latter is in reverse order). The above sequence has the edges $\{v_{j-1}, v_1\} \in E$ and $\{v_j, v_2\} \in E$ that the original sequence does not have. The original sequence has $\{v_1, v_2\} \notin E$ and $\{v_{j-1}, v_j\}$ (which may or may not be in E) that the above sequence does not have. So as desired, the above sequence has more edges than the original sequence, which contradicts the maximality of the original sequence. $\square$

10 Graphs II

10.1 Depth-first search

Definition 10.1 (Depth-first search (DFS) algorithm)**.**
The *depth-first search* algorithm, denoted DFS, is the following algorithm.

```
DFS on input $G = (V, E)$ and $u \in V$:
  • Mark $u$ as "visited".
  • For each $v \in N(u)$:
      – If $v$ is not marked "visited", then run DFS($G$, $v$).
```

■

Exercise 10.2. Show that the running time of DFS is $O(m)$ where m is the number of edges of the input graph.

Remark. Note that DFS(G, u) visits all the vertices in the connected component that u is a part of. If we want to traverse all the vertices in the graph, and the graph has multiple connected components, then we can do:

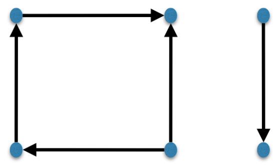
```
  • For each vertex $v$ that is not marked as "visited":
      – Run DFS($G$, $v$).
```

The running time of this algorithm is $O(n + m)$.

Definition 10.3 (Directed graph)**.**
A *directed graph* G is a pair (V, A), where

- V is a finite set called the set of *vertices* (or *nodes*),
- A is a finite set called the set of *directed edges* (or *arcs*), and every element of A is a tuple (u, v) for $u, v \in V$. If $(u, v) \in A$, we say that there is a directed edge from u to v. Note that $(u, v) \neq (v, u)$ unless $u = v$.

Below is an example of how we draw a directed graph:

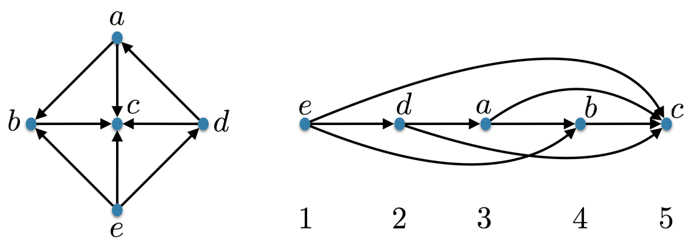


■

Definition 10.4 (Neighborhood, out-degree, in-degree, sink, source).
Let $G = (V, A)$ be a directed graph. For $u \in V$, we define the neighborhood of u, $N(u)$, as the set $\{v \in V : (u, v) \in A\}$. The *out-degree* of u, denoted $\deg_{\text{out}}(u)$, is $|N(u)|$. The *in-degree* of u, denoted $\deg_{\text{in}}(u)$, is the size of the set $\{v \in V : (v, u) \in A\}$. A vertex with out-degree 0 is called a *sink*. A vertex with in-degree 0 is called a *source*. ■

Remark. The notions of *paths* and *cycles* naturally extend to directed graphs. For example, we say that there is a path from u to v if there is a sequence of distinct vertices $u = v_0, v_1, \dots, v_k = v$ such that $(v_{i-1}, v_i) \in A$ for all $i \in \{1, 2, \dots, k\}$. Also note that the DFS algorithm can be applied to directed graphs as well.

Definition 10.5 (Topological order of a directed graph).
A *topological order* of an n-vertex directed graph $G = (V, A)$ is a bijection $f : V \to \{1, 2, \dots, n\}$ such that if $(u, v) \in A$, then $f(u) < f(v)$. Below is an example:



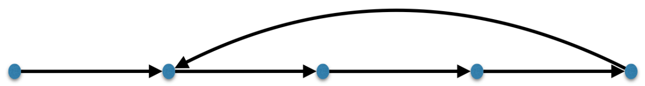
Here, $f(e) = 1$, $f(d) = 2$, $f(a) = 3$, $f(b) = 4$, and $f(c) = 5$. ■

Exercise 10.6. Show that if a directed graph has a cycle, then it does not have a topological order.

Definition 10.7 (Topological sorting problem).
In the *topological sorting problem*, the input is a directed acyclic graph, and the output is a topological order of the graph. ■

Lemma 10.8. *If a directed graph is acyclic, then it has a sink vertex.*

Proof. By contrapositive: If a directed graph has no sink vertices, then it means that every vertex has an outgoing edge. Start with any vertex, and follow an outgoing edge to arrive at a new vertex. Repeat this process. At some point, you have to visit a vertex that you have visited before. This forms a cycle.



□

Proposition 10.9 (Topological sort - naïve algorithm)**.** *The following algorithm solves the topological sorting problem in polynomial time.*

```
On input an $n$-vertex directed acyclic graph $G = (V, A)$:
  • Let $p = n$.
  • While $p \geq 1$:
      – Find a sink vertex $v$ and remove it from the graph $G$.
      – Let $f(v) = p$.
      – Let $p = p - 1$.
  • Output $f$.
```

Exercise 10.10. Show the algorithm correctly solves the topological sorting problem, i.e., show that for $(u, v) \in A$, $f(u) < f(v)$. What is the running time of this algorithm?

Theorem 10.11 (Topological sort via DFS)**.** *There is a $O(n+m)$-time algorithm that solves the topological sorting problem.*

Proof. The algorithm is a slight variation of DFS.

```
On input an $n$-vertex directed acyclic graph $G = (V, A)$:
  • Let $p = n$
  • For each vertex $v$ that is not marked as "visited":
      – Run DFS($G$, $v$).

    DFS on input $G = (V, A)$ and $v \in V$:
      – Mark $v$ as "visited".
      – For each $u \in N(v)$:
          ∗ If $u$ is not marked "visited", then run DFS($G$, $u$).
      – Let $f(v) = p$.
      – Let $p = p - 1$.

  • Output $f$.
```

The running time is the same as DFS. To show the correctness of the algorithm, all we need to show is that for $(u, v) \in A$, $f(u) < f(v)$. There are two cases to consider.

- *Case 1:* u is visited before v. In this case observe that DFS(G, v) will finish before DFS(G, u). Therefore $f(v)$ will be assigned a value before $f(u)$, and so $f(u) < f(v)$.
- *Case 2:* v is visited before u. Notice that we cannot visit u from DFS(G, v) because that would imply that there is a cycle. Therefore DFS(G, u) is called after DFS(G, v) is completed. As before, $f(v)$ will be assigned a value before $f(u)$, and so $f(u) < f(v)$.

□

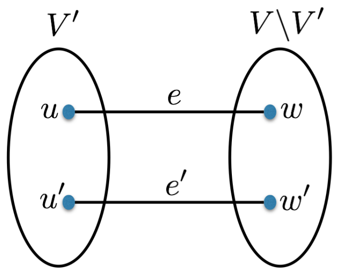
10.2 Minimum spanning tree

Definition 10.12 (Minimum spanning tree (MST) problem)**.**
In the *minimum spanning tree problem*, the input is a connected undirected graph $G = (V, E)$ together with a *cost* function $c : E \to \mathbb{R}^+$. The output is a a subset of the edges of minimum total cost such that, in the graph restricted to these edges, all the vertices of G are connected.[19] For convenience, we'll assume that the edges have unique edge costs, i.e. $e \neq e' \implies c(e) \neq c(e')$. ■

Exercise 10.13. With unique edge costs, show that the minimum spanning tree is unique.

Theorem 10.14 (MST cut property)**.** *Suppose we are given an instance of the MST problem. For any $V' \subseteq V$, let $e = \{u, w\}$ be the cheapest edge with the property that $u \in V'$ and $w \in V \backslash V'$. Then e must be in the minimum spanning tree.*

Proof. Let T be the minimum spanning tree. The proof is by contradiction, so assume that $e = \{u, w\}$ is not in T. Since T spans the whole graph, there must a path from u to w in T. Let $e' = \{u', w'\}$ be the first edge on this path such that $u' \in V'$ and $w' \in V \backslash V'$. Let $T_{e-e'} = (T \backslash \{e'\}) \cup \{e\}$. If $T_{e-e'}$ is a spanning tree, then we reach a contradiction because $T_{e-e'}$ has lower cost than T (since $c(e) < c(e')$).



[19]Obviously this subset of edges would not contain a cycle since if it did, we could remove any edge on the cycle, preserve the connectivity property, and obtain a cheaper set. Therefore, this set forms a tree.

$T_{e-e'}$ is a spanning tree: Clearly $T_{e-e'}$ has $n-1$ edges (since T has $n-1$ edges). So if we can show that $T_{e-e'}$ is connected, this would imply that $T_{e-e'}$ is a tree and touches every vertex of the graph, i.e., $T_{e-e'}$ is a spanning tree. Consider any two vertices $s, t \in V$. There is a unique path from s to t in T. If this path does not use the edge $e' = \{u', w'\}$, then the same path exists in $T_{e-e'}$, so s and t are connected in $T_{e-e'}$. If the path does use $e' = \{u', w'\}$, then instead of taking the edge $\{u', w'\}$, we can take the following path: take the path from u' to u, then take the edge $e = \{u, w\}$, then take the path from w to w'. So replacing $\{u', w'\}$ with this path allows us to construct a sequence of vertices starting from s and ending at t, such that each consecutive pair of vertices is an edge. Therefore s and t are connected. □

Theorem 10.15 (Jarník-Prim algorithm for MST)**.** *There is an algorithm that solves the MST problem in $O(n^2)$ time, where n is the number of vertices of the input graph.*

Proof. We first present the algorithm which is due to Jarník and Prim. Given an undirected graph $G = (V, E)$ and a cost function $c : E \to \mathbb{R}^+$:

```
• Let $V' = \{u\}$ for some arbitrary $u \in V$, and let $E' = \emptyset$.
• While $V' \neq V$ do:
    – Let $\{u, v\}$ be the minimum cost edge such that $u \in V'$ but $v \notin V'$.
    – Add $\{u, v\}$ to $E'$.
    – Add $v$ to $V'$.
• Output $E'$.
```

By the MST cut property (Theorem 10.14), the algorithm always adds an edge that must be in the MST. The number of iterations is $n-1$, so all the edges of the MST are added to E'. Therefore the algorithm correctly outputs the unique MST.

The running time of the algorithm is $O(n^2)$ because there are $O(n)$ iterations, and the body of the loop can be done in $O(n)$ time. □

Exercise 10.16. Suppose an instance of the Minimum Spanning Tree problem is allowed to have negative costs for the edges. Explain whether the Jarník-Prim algorithm would work in this case as well.

Exercise 10.17. Consider the problem of computing the maximum spanning tree, i.e., a spanning tree that maximizes the sum of the edge costs. Explain whether the Jarník-Prim algorithm solves this problem if we modify it so that at each iteration, the algorithm chooses the edge between V' and $V \backslash V'$ with the maximum cost.

Exercise 10.18. Consider the following algorithm for the MST problem (which is known as Kruskal's algorithm). Start with MST being the empty set. Go through all the edges of the graph one by one from the cheapest to the most expensive. Add the edge to the MST if it does not create a cycle. Show that this algorithm correctly outputs the MST.

11 Graphs III

11.1 Maximum matching

Definition 11.1 (Matching – maximum, maximal, perfect)**.**
A *matching* in a graph $G = (V, E)$ is a subset of the edges that do not share an endpoint. A *maximum matching* in G is a matching with the maximum number of edges among all possible matchings. A *maximal matching* is a matching with the property that if we add any other edge to the matching, it is no longer a matching. Note that a maximal matching is not necessarily a maximum matching, but a maximum matching is always a maximal matching. A *perfect matching* is a matching that covers all the vertices of the graph. ■

Exercise 11.2. Let n be even, and let G be the complete graph on n vertices. How many different perfect matchings does G contain?

Definition 11.3 (Maximum matching problem)**.**
In the *maximum matching problem* the input is an undirected graph $G = (V, E)$ and the output is a maximum matching in G. ■

Definition 11.4 (Augmenting path)**.**
Let $G = (V, E)$ be a graph and let $M \subseteq E$ be a matching in G. An *augmenting path* in G with respect to M is a path such that

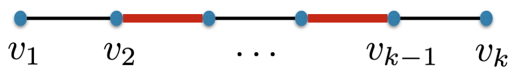
(i) the edges in the path alternate between being in M and not in M
(a single edge which is not in M satisfies this property),

(ii) the first and last vertices in the path are not a part of the matching M.

Note that an augmenting path does not need to contain all the edges in M. ■

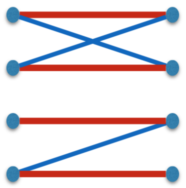
Theorem 11.5. *Let $G = (V, E)$ be a graph. A matching $M \subseteq E$ is maximum if and only if there is no augmenting path in G with respect to M.*

Proof. First, suppose there is an augmenting path in G with respect to M. Then we want to show that M is not maximum. Let the augmenting path be $v_1, v_2, \ldots, v_k$:



The highlighted edges represent edges in M. By the definition of an augmenting path, we know that v_1 and v_k are not matched by M. On this path, the number of edges in the matching is one less than the number of edges not in the matching. To see that M is not a maximum matching, observe that we can obtain a bigger matching by flipping the matched and unmatched edges on the augmenting path. In other words, if an edge on the path is in the matching, we remove it from the matching, and if an edge on the path is not in the matching, we put it in the matching. This gives us a matching with one more edge than M, so M is not maximum.

We now prove the other direction. In particular, we want to show that if M is not a maximum matching, then we can find an augmenting path in G with respect to M. Let M^* denote a maximum matching in G. Since M is not maximum, we know that $|M| < |M^*|$. We define the set S to be the set of edges contained in M^* or M, but not both. That is, $S = (M^* \cup M) \backslash (M^* \cap M)$. If we color the edges in M with blue, and the edges in M^* with red, then S consists of edges that are colored either blue or red, but not both. Below is an example:



(Horizontal edges correspond to the red edges. The rest is blue.) Our goal is to find an augmenting path with respect to M in S. Observe that each vertex that is a part of S has degree 1 or 2 because it can be incident to at most one edge in M and at most one edge in M^*. If the degree was more than 2, M and M^* would not be matchings. We make two claims:

(i) Because every vertex has degree 1 or 2, S consists of disjoint paths and cycles.

(ii) The edges in these paths and cycles alternate between blue and red.

Given the first claim, the second claim is clearly true. The proof of the first claim is omitted and is left as an exercise for the reader.

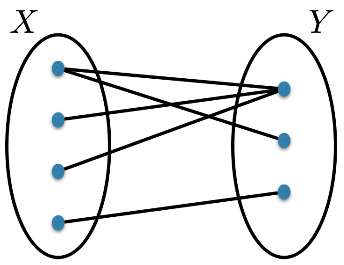
Since M^* is a bigger matching than M, we know that S has more red edges than blue edges. Observe that the cycles in S must have even length, because otherwise the edges cannot alternate between blue and red. Therefore the cycles have an equal number of red and blue edges. This implies that there must be a path in S with more red edges than blue edges. In particular, this path starts and ends with a red edge. This path is an augmenting path with respect to M, since it is clearly alternating between edges in M and edges not in M, and the endpoints are unmatched with respect to M. So using the assumption that M is not maximum, we were able to find an augmenting path with respect to M. This completes the proof. $\square$

Exercise 11.6. Let $G = (V, E)$ be a graph such that all vertices have degree 1 or 2. Then prove that G consists of disjoint paths and cycles.

Definition 11.7 (Bipartite graph)**.**
A graph $G = (V, E)$ is called *bipartite* if there is a partition[20] of V into sets X and Y such that all the edges in E have one endpoint in X and the other in Y. Sometimes

[20]Recall that a *partition* of V into X and Y means that X and Y are disjoint and $X \cup Y = V$.

the bipartition is given explicitly and the graph is denoted by $G = (X, Y, E)$. Below is an example of a bipartite graph.



■

Exercise 11.8.

(a) Show that a graph is bipartite if and only if it contains no odd-length cycles.

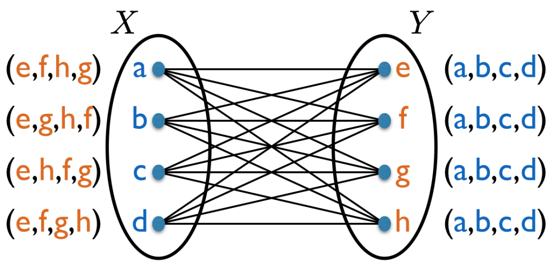
(b) Give a polynomial-time algorithm that determines whether an input graph is bipartite or not.

Exercise 11.9. Given as input a bipartite graph $G = (X, Y, E)$, find a polynomial-time algorithm that outputs a maximum matching in G.
(Hint: Use Theorem 11.5.)

11.2 Stable matching

Definition 11.10 (Complete graph)**.**
A graph $G = (V, E)$ is called *complete* if E contains all the possible edges, i.e., $\{u, v\} \in E$ for any distinct $u, v \in V$. A bipartite graph $G = (X, Y, E)$ is called complete if E contains all the possible edges between X vertices and Y vertices. ■

Definition 11.11 (Stable matching problem)**.**
An instance of the *stable matching problem* is a complete bipartite graph $G = (X, Y, E)$ with $|X| = |Y|$, and a *preference list* for each node of the graph. A preference list for a node in X is an ordering of the Y vertices, and a preference list for a node in Y is an ordering of the X vertices. Below is an example of an instance of the stable matching problem:



The output of the stable matching problem is a *stable matching*, which is defined as a matching that satisfies two properties:

(i) The matching is a perfect matching.

(ii) The matching does not contain any *unstable* pairs. A pair of vertices (x, y) where $x \in X$ and $y \in Y$ is called *unstable* if they are not matched to each other, but they both prefer each other to the partners they are matched to.

■

Theorem 11.12 (Gale-Shapley proposal algorithm)**.** *There is a polynomial time algorithm which, given an instance of the stable matching problem, always returns a stable matching.*

Proof. We first describe the algorithm (which is called the Gale-Shapley algorithm). For the sake of clear exposition, we refer to the elements of X as men, and the elements of Y as women.

```
• While there is a man $m$ who is not matched:
    – Let $w$ be the highest ranked woman on $m$'s preference list
      to whom $m$ has not "proposed" yet.
    – Let $m$ "propose" to $w$.
    – If $w$ is unmatched or $w$ prefers $m$ over her current partner,
      match $m$ and $w$.
      (The previous partner of $w$, if there was any, is now unmatched.)
```

The theorem will follow once we show the following 3 things:

(a) the number of iterations in the algorithm is at most n^2, where $n = |X| = |Y|$,

(b) the algorithm always outputs a perfect matching,

(c) this matching contains no unstable pairs.

Part (a) implies that the algorithm is polynomial time. Parts (b) and (c) imply that the matching returned by the algorithm is a stable matching.

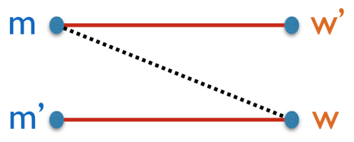
Proof of (a): Notice that the number of iterations in the algorithm is equal to the total number of proposals made. No man proposes to a woman more than once, so each man makes at most n proposals. There are n men in total, so the total number of proposals is at most n^2.

Proof of (b): The proof is by contradiction, so suppose the algorithm does not output a perfect matching. This means that some man, call it m, is not matched to any woman. The proof can be broken down as follows:

$$\begin{aligned} m \text{ is not matched at the end} &\implies \text{all women must be matched at the end} \\ &\implies \text{all men must be matched at the end.} \end{aligned}$$

This obviously leads to the desired contradiction. The second implication is quite simple: since there are an equal number of men and women, the only way all the women can be matched at the end is if all the men are matched. To show the first implication, notice that since m is not matched at the end, he got rejected by all the women he proposed to. Either he got rejected because the woman preferred her current partner, or he got rejected by a woman that he was already matched with. Either way, all the women that m proposed to must have been matched to someone at some point in the algorithm. But once a woman is matched, she never goes back to being unmatched. So at the end of the algorithm, all the women must be matched.

Proof of (c): We first make a crucial observation. As the algorithm proceeds, a man can only go down in his preference list, and a woman can only go up in her preference list. Now consider any pair (m, w) where $m \in X$, $w \in Y$, and m and w are not matched by the algorithm. We want to show that this pair is not unstable. Let w' be the woman that m is matched to, and let m' be the man that w is matched to.



There are two cases to consider:

(i) m proposed to w at some point in the algorithm,

(ii) m never proposed to w.

If (i) happened, then w must have rejected m at some point, which implies w must prefer m' over m (recall w can only go up in her preference list). This implies w does not prefer m over her current partner, and so (m, w) is not unstable. If (ii) happened, then w' must be higher on the preference list of m than w (recall m can only go down in his preference list). This implies m does not prefer w over his current partner, and so (m, w) is not unstable. So in either case, (m, w) is stable, and we are done. □

Definition 11.13 (Best and worst valid partners)**.**
Consider an instance of the stable matching problem. We say that $m \in X$ is a *valid partner* of $w \in Y$ (or w is a valid partner of m) if there is some stable matching in which m and w are matched. For $u \in X \cup Y$, we define the *best valid partner* of u, denoted $\text{best}(u)$, to be the highest ranked valid partner of u. Similarly, we define the *worst valid partner* of u, denoted $\text{worst}(u)$, to be the lowest ranked valid partner of u. ■

Theorem 11.14. *The Gale-Shapley algorithm always matches a male $m \in X$ with its best valid partner, i.e., it returns $\{(m, \text{best}(m)) : m \in X\}$. It also always matches a female $w \in Y$ with its worst valid partner, i.e., it returns $\{(\text{worst}(w), w) : w \in Y\}$.*

Remark. Note that it is not a priori clear at all that $\{(m, \text{best}(m)) : m \in X\}$ would be a matching, not to mention a stable matching.

Exercise 11.15. Give a polynomial time algorithm that determines if a given instance of the stable matching problem has a unique solution or not.
(Hint: Use Theorem 11.14)

Exercise 11.16. Consider the following variant of the stable matching problem. The input is a complete graph on n vertices, where n is even. Each vertex has a preference list over every other vertex in the graph. The goal is to find a stable matching. Give an example to show that a stable matching does not always exist.

12 Polynomial-time Reductions

We start by defining several important and well-known decision problems.

Definition 12.1 (k-Coloring problem).
Let $G = (V, E)$ be a graph. Let $k \in \mathbb{N}^+$. A *k-coloring* of V is just a map $\chi : V \to C$ where C is a set of cardinality k. (Usually the elements of C are called *colors*. If $k = 3$ then $C = \{\text{red}, \text{green}, \text{blue}\}$ is a popular choice. If k is large, we often just call the "colors" $1, 2, \dots, k$.) A k-coloring is said to be *legal* (or *valid*) for G if every edge in E is *bichromatic*, meaning that its two endpoints have different colors. (I.e., for all $\{u, v\} \in E$ it is required that $\chi(u) \neq \chi(v)$.) Finally, we say that G is *k-colorable* if it has a legal k-coloring. Note that a graph is 2-colorable if and only if it is bipartite.

In the *k-coloring problem*, the input is an undirected graph $G = (V, E)$, and the output is True if and only if the graph is k-colorable. We denote this problem by kCOL. ■

Definition 12.2 (Clique problem).
Let $G = (V, E)$ be an undirected graph. A subset of the vertices is called a *clique* if there is an edge between any two vertices in the subset. We say that G contains a k-clique if there is a subset of the vertices of size k that forms a clique.

In the *clique problem*, the input is an undirected graph $G = (V, E)$ and a number $k \in \mathbb{N}^+$, and the output is True if and only if the graph contains a k-clique. We denote this problem by CLIQUE. ■

Definition 12.3 (Independent set problem).
Let $G = (V, E)$ be an undirected graph. A subset of the vertices is called an *independent set* if there is no edge between any two vertices in the subset. We say that G contains an independent set of size k if there is a subset of the vertices of size k that forms an independent set.

In the *independent set problem*, the input is an undirected graph $G = (V, E)$ and a number $k \in \mathbb{N}^+$, and the output is True if and only if the graph contains an independent set of size at least k. We denote this problem by IS. ■

Definition 12.4 (Circuit satisfiability problem).
In the *circuit satisfiability problem*, the input is a Boolean circuit, and the output is True if and only if there is an assignment to the input gates that makes the circuit output 1. We denote this problem by CIRCUIT-SAT. ■

Definition 12.5 (Boolean satisfiability problem).
Let $x_1, \dots, x_n$ be Boolean variables, i.e., variables that can be assigned True or False. A *literal* refers to a Boolean variable or its negation. A *clause* is an "OR" of literals. For example, $x_1 \vee \neg x_3 \vee x_4$ is a clause. A Boolean formula in *conjunctive normal form* (CNF) is an "AND" of clauses. For example,

$$(x_1 \vee \neg x_3 \vee x_4) \wedge (x_2 \vee x_2 \vee x_4) \wedge (x_1 \vee \neg x_1 \vee \neg x_5)$$

is a CNF formula. We say that a Boolean formula is *satisfiable* if there is a truth assignment to the Boolean variables that makes the formula evaluate to True.

In the CNF *satisfiability problem*, the input is a CNF formula, and the output is True if and only if the formula is satisfiable. We denote this problem by SAT. In a variation of SAT, we restrict the input formula such that every clause has exactly 3 literals. This variation of the problem is denoted by 3SAT. ■

Exercise 12.6. Find exponential-time algorithms to solve the problems listed above.

Definition 12.7 (Polynomial-time Turing reduction (Cook reduction))**.**
Fix some alphabet Σ. Let A and B be two languages. We say that A *polynomial-time reduces* to B, written $A \leq_T^P B$, if it is possible to decide A in polynomial time assuming a polynomial-time decider for B exists. Polynomial-time reductions are also known as *Cook reductions*, named after Stephen Cook. ■

Remark. Observe that if $A \leq_T^P B$ and $B \in \mathsf{P}$, then $A \in \mathsf{P}$. Equivalently, taking the contrapositive, if $A \leq_T^P B$ and $A \notin \mathsf{P}$, then $B \notin \mathsf{P}$. So when $A \leq_T^P B$, we think of B as being at least as hard as A with respect to polynomial-time decidability.

Exercise 12.8. Show that if $A \leq_T^P B$ and $B \leq_T^P C$, then $A \leq_T^P C$.

Theorem 12.9. CLIQUE $\leq_T^P$ IS.

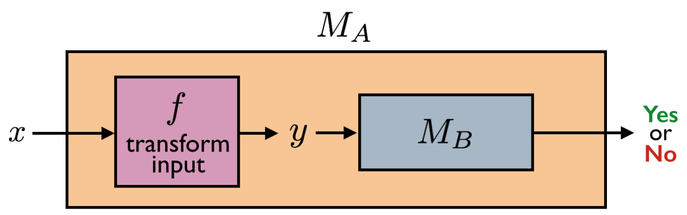
Proof. Suppose M_{IS} is a polynomial-time decider for IS. Our goal is to show that there is a polynomial-time decider for the CLIQUE problem. Before we describe the decider for CLIQUE, we make a definition. Let $G = (V, E)$ be a graph. The *complement* of G is the graph $G^* = (V, E^*)$ such that $E^* = \{\{u, v\} : \{u, v\} \notin E\}$. The decider for CLIQUE is as follows.

```
"On input <G,k>:
    Construct G*.
    Run M_IS(<G*,k>).
    If it accepts, accept.
    If it rejects, reject.
"
```

First of all, this machine runs in polynomial time because the construction of G^* takes polynomial time and M_{IS} runs in polynomial time. To see that the decider works correctly, it suffices to show that $\langle G, k\rangle \in$ CLIQUE if and only if $\langle G^*, k\rangle \in$ IS. To see that this is true, just observe that if S is a clique in G of size k, then $\{u, v\} \in E$ for all distinct $u, v \in S$, which means $\{u, v\} \notin E^*$ for all distinct $u, v \in S$. So S is an independent set of size k in G^*. Similarly, any independent set in G^* of size k corresponds to a clique of size k in G. This implies that $\langle G, k\rangle \in$ CLIQUE if and only if $\langle G^*, k\rangle \in$ IS, as desired. □

Exercise 12.10. How can you modify the above reduction to show that IS $\leq_T^P$ CLIQUE?

Definition 12.11 (Many-one reduction (Karp reduction))**.**
Let A and B be two languages. Suppose that there is a polynomial-time algorithm (also called a polynomial-time transformation) f that maps instances of A to instances of B such that $x \in A$ if and only if $f(x) \in B$. Then we say that there is a polynomial-time *many-one reduction* (or a *Karp reduction*, named after Richard Karp) from A to B. A many-one reduction is a Turing reduction, however not all Turing reductions are a many-one reduction.



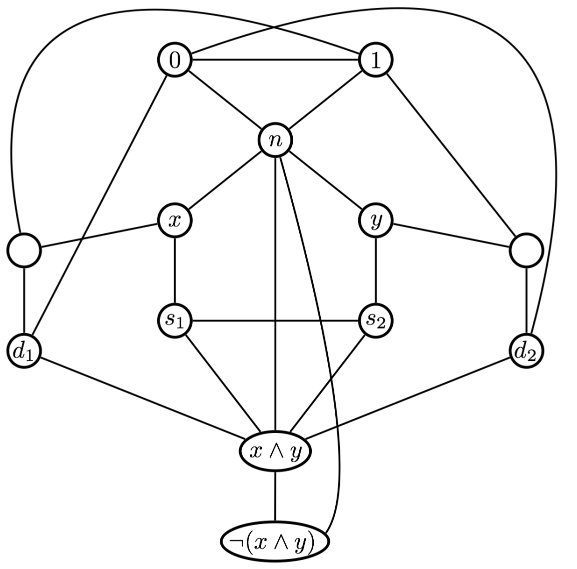
∎

Remark. The reduction presented in the proof of Theorem 12.9 is a Karp reduction. In fact, almost all of the reductions that we present will be Karp reductions.

Theorem 12.12. CIRCUIT-SAT $\leq_T^P$ 3COL.

Proof. To prove the theorem, we will present a Karp reduction from CIRCUIT-SAT to 3COL. In particular, given a CIRCUIT-SAT instance C, we will construct a 3COL instance G such that C is a satisfiable Boolean circuit if and only if G is 3-colorable. Furthermore, the construction will be done in polynomial time.

First, using Exercise 8.5, we know that any Boolean circuit with AND, OR, and NOT gates can be converted into an equivalent circuit that only has NAND gates (in addition to the input gates). This transformation can easily be done in polynomial time. So without loss of generality, we assume that our circuit C is a circuit with NAND and input gates. We construct G by converting each NAND gate into a set of 13 vertices and 23 edges. The construction is given below.



The vertices labeled with x and y correspond to the inputs of the NAND gate. The vertex labeled with $\neg(x \wedge y)$ corresponds to the output of the gate. We construct such a graph for each NAND gate of the circuit, however, we make sure that if, say, gate g_1 is an input to gate g_2, then the vertex corresponding to the output of g_1 coincides with (is the same as) the vertex corresponding to one of the inputs of g_2. Furthermore, the vertices labeled with 0, 1 and n are the same for each gate. In other words, in the whole graph, there is only one vertex labeled with 0, one vertex labeled with 1, and one vertex labeled with n. Lastly, we put an edge between the vertex corresponding to the output vertex of the output gate and the vertex labeled with 0. This completes the construction of the graph G. Before we prove that the reduction is correct, we make some preliminary observations.

Let's call the 3 colors we use to color the graph 0, 1 and n (we think of n as "none"). Any valid coloring of G must assign different colors to 3 vertices that form a triangle (e.g. vertices labeled with 0, 1 and n). If G is 3-colorable, we can assume without loss generality that the vertex labeled 0 is colored with the color 0, the vertex labeled 1 is colored with color 1, and the vertex labeled n is colored with the color n. This is without loss of generality because if there is a valid coloring of G, any permutation of the colors corresponds to a valid coloring as well. Therefore we can permute the colors so that the labels of those vertices coincide with the colors they are colored with.

Notice that since the vertices corresponding to the inputs of a gate (i.e. the x and y vertices) are connected to vertex n, they will be assigned the colors 0 or 1. Let's consider two cases:

- If x and y are assigned the same color (i.e. either they are both 0 or they are both 1), the vertex labeled with $x \wedge y$ will have to be colored with that same color. That is, the vertex labeled with $x \wedge y$ must get the color corresponding to the evaluation of $x \wedge y$. To see this, just notice that the vertices labeled s_1 and

s_2 must be colored with the two colors that x and y are not colored with. This forces the vertex $x \wedge y$ to be colored with the same color as x and y.

- If x and y are assigned different colors (i.e. one is colored with 0 and the other with 1), the vertex labeled with $x \wedge y$ will have to be colored with 0. That is, as in the first case, the vertex labeled with $x \wedge y$ must get the color corresponding to the evaluation of $x \wedge y$. To see this, just notice that one of the vertices labeled d_1 or d_2 must be colored with 1. This forces the vertex $x \wedge y$ to be colored with 0 since it is already connected to vertex n.

In either case, the color of the vertex $x \wedge y$ must correspond to the evaluation of $x \wedge y$. It is then easy to see that the color of the vertex $\neg(x \wedge y)$ must correspond to the evaluation of $\neg(x \wedge y)$.

We are now ready to argue that circuit C is satisfiable if and only if G is 3-colorable. Let's first assume that the circuit we have is satisfiable. We want to show that the graph G we constructed is 3-colorable. Since the circuit is satisfiable, there is a 0/1 assignment to the input variables that makes the circuit evaluate to 1. We claim that we can use this 0/1 assignment to validly color the vertices of G. We start by coloring each vertex that corresponds to an input variable: In the satisfying truth assignment, if an input variable is set to 0, we color the corresponding vertex with the color 0, and if an input variable is set to 1, we color the corresponding vertex with the color 1. As we have argued earlier, a vertex that corresponds to the output of a gate (the vertex at the very bottom of the picture above) is forced to be colored with the color that corresponds to the value that the gate outputs. It is easy to see that the other vertices, i.e., the ones labeled s_1, s_2, d_1, d_2 and the unlabeled vertices can be assigned valid colors. Once we color the vertices in this manner, the vertices corresponding to the inputs and output of a gate will be consistently colored with the values that it takes as input and the value it outputs. Recall that in the construction of G, we connected the output vertex of the output gate with the vertex labeled with 0, which forces it to be assigned the color 1. We know this will indeed happen since the 0/1 assignment we started with makes the circuit output 1. This shows that we can obtain a valid 3-coloring of the graph G.

The other direction is very similar. Assume that the constructed graph G has a valid 3-coloring. As we have argued before, we can assume without loss of generality that the vertices labeled 0, 1, and n are assigned the colors 0, 1, and n respectively. We know that the vertices corresponding to the inputs of a gate must be assigned the colors 0 or 1 (since they are connected to the vertex labeled n). Again, as argued before, given the colors of the input vertices of a gate, the output vertex of the gate is forced to be colored with the value that the gate would output in the circuit. The fact that we can 3-color the graph means that the output vertex of the output gate is colored with 1 (since it is connected to vertex 0 and vertex n by construction). This implies that the colors of the vertices corresponding to the input variables form a 0/1 assignment that makes the circuit output a 1, i.e. the circuit is satisfiable.

To finish the proof, we must argue that the construction of graph G, given circuit C, can be done in polynomial time. This is easy to see since for each gate of the circuit, we create at most a constant number of vertices and a constant number of edges. So if the circuit has s gates, the construction can be done in $O(s)$ steps. $\square$

Definition 12.13 ($\mathcal{C}$-hard, $\mathcal{C}$-complete)**.**
Let $\mathcal{C}$ be a set of languages.

- We say that L is $\mathcal{C}$-hard (with respect to polynomial-time reductions) if for all languages $K \in \mathcal{C}$, $K \leq_T^P L$.
(With respect to polynomial time decidability, a $\mathcal{C}$-hard language is at least as "hard" as any language in $\mathcal{C}$.)

- We say that L is $\mathcal{C}$-complete if L is $\mathcal{C}$-hard and $L \in \mathcal{C}$.
(A $\mathcal{C}$-complete language represents the "hardest" language in $\mathcal{C}$ with respect to polynomial time decidability.)

$\blacksquare$

Remark. Notice that if L is $\mathcal{C}$-hard and $L \in \mathsf{P}$, then $\mathcal{C} \subseteq \mathsf{P}$.

13 Non-Deterministic Polynomial Time

13.1 The complexity class NP

Definition 13.1 (Non-deterministic polynomial time, complexity class NP).
Fix some alphabet Σ. We say that a language L can be decided in non-deterministic polynomial time if there exists

(i) a polynomial time decider TM V that takes two strings as input, and

(ii) a constant k,

such that for all $x \in \Sigma^*$:

- if $x \in L$, then there exists $u \in \Sigma^*$ with $|u| \leq |x|^k$ such that $V(x, u)$ accepts,
- if $x \notin L$, then for all $u \in \Sigma^*$, $V(x, u)$ rejects.

If $x \in L$, a u that makes $V(x, u)$ accept is called a *proof* (or *certificate*) of x being in L. The TM V is called a *verifier*.

We denote by NP the set of all languages which can be decided in non-deterministic polynomial time. ■

Proposition 13.2. 3COL $\in$ NP.

Proof. To show 3COL is in NP, we need to show that there is a polynomial-time verifier TM V with the properties stated in Definition 13.1. Recall that an instance of the 3COL problem is an undirected graph G. The description of V is as follows.

```
"On input <G,u>:
    If u is not a valid encoding of a 3 coloring of the vertices, reject.
    If there is an edge {v,w} where v and w have same color, reject.
    Else, accept.
"
```

This machine is polynomial-time: To check whether u is a valid encoding of a 3-coloring of the vertices takes polynomial time (you just need to check that you are given $|V|$ colors, each being one of 3 colors). To check that it is indeed a valid 3-coloring is polynomial time as well (you just need to go through every edge once). We now verify that V satisfies the two conditions stated in Definition 13.1. If $\langle G \rangle$ is in the language, that means there must be some valid 3-coloring of the vertices. When u is this valid 3-coloring, $|u| = O(|V|)$, and the verifier accepts. On the other hand, if $\langle G \rangle$ is not in the language, this means that there is no valid 3-coloring of G, so no matter what u is given to the verifier, it rejects. This shows 3COL is in NP. □

Proposition 13.3. CIRCUIT-SAT $\in$ NP.

Proof. To show CIRCUIT-SAT is in NP, we need to show that there is a polynomial-time verifier V with the properties stated in Definition 13.1. Recall that an instance of the CIRCUIT-SAT problem is a circuit C. The description of V is as follows.

```
"On input <C,u>:
    If u doesn't correspond to a 0/1 assignment to input gates, reject.
    Given the 0/1 assignment to input gates,
        compute the output of the circuit.
    If the output is 1, accept.
    Else, reject.
"
```

Let n be the total number of gates in C. This machine is polynomial-time since checking that u is a valid 0/1-assignment to the input-gates takes at most $O(n)$ steps, and computing the output of the circuit takes at most $O(n)$ steps (it takes constant number of steps to compute each gate). We now check that V satisfies the two conditions stated in Definition 13.1. If $\langle C\rangle$ is in the language, that means there must be some 0/1-assignment to the input gates that makes the circuit output 1. When u is this 0/1-assignment, then $|u| = O(n)$, and the verifier accepts. On the other hand, if $\langle C\rangle$ is not in the language, this means that there is no 0/1-assignment to the input gates that makes the circuit output 1, so no matter what u is given to the verifier, it rejects. This shows CIRCUIT-SAT is in NP. □

Exercise 13.4. Show that CLIQUE $\in \mathsf{NP}$.

Exercise 13.5. Show that IS $\in \mathsf{NP}$.

Proposition 13.6. $\mathsf{P} \subseteq \mathsf{NP}$.

Proof. Given a language $L \in \mathsf{P}$, we want to show that $L \in \mathsf{NP}$. Since L is in P, we know that there is a polynomial-time decider M that decides L. To show that $L \in \mathsf{NP}$, we need to describe a polynomial-time verifier V that have the properties described in Definition 13.1. The description of V is as follows.

```
"On input <x,u>:
    run M(x).
    if it accepts, accept.
    if it rejects, reject.
"
```

First, note that since M is a polynomial time decider, the line "run $M(x)$" takes polynomial time, and so V is polynomial-time. We now check that V satisfies the two conditions stated in Definition 13.1. If $x \in L$, then $M(x)$ accepts, so for any u, $V(x,u)$ accepts. For example, $V(x,\epsilon)$ accepts, and clearly $|\epsilon| = 0 \leq |x|$. If $x \notin L$, then $M(x)$ rejects, so no matter what u is, $V(x,u)$ rejects, as desired. This shows that $L \in \mathsf{NP}$. □

Definition 13.7 (Complexity class EXP)**.**
We denote by EXP the set of all languages that can be decided in at most exponential-time, i.e., in time $O(2^{n^k})$ for some constant $k > 0$. ■

Exercise 13.8. Show that $\mathsf{NP} \subseteq \mathsf{EXP}$.

13.2 NP-complete problems

Theorem 13.9 (Cook 1971, Levin 1973)**.** CIRCUIT-SAT *is* NP-complete.

Proof Sketch. Warning: This is only a proof sketch. You will not be responsible for this proof.

To prove this theorem, we have to show that CIRCUIT-SAT is in NP and that it is NP-hard. We have already shown that CIRCUIT-SAT is in NP in Proposition 13.3. So it remains to show that CIRCUIT-SAT is NP-hard, i.e., $L \leq_T^P$ CIRCUIT-SAT for every language $L \in \mathsf{NP}$.

Let M_{CS} denote a polynomial-time decider for CIRCUIT-SAT. Let L be an arbitrary language in NP. Our task is to show that we can decide L in polynomial time using M_{CS} as a subroutine. Since L is in NP, there is a polynomial-time verifier TM V that satisfies the conditions stated in Definition 13.1. Note that V is just a good old decider TM. In the Boolean Circuits section, we have seen that every TM can be efficiently simulated by a Boolean circuit family (Theorem 8.14). So using that theorem, we know that there exists a polynomial size circuit family that simulates V. In order to decide L, we'll use this fact and feed M_{CS} an appropriate circuit. Then the output of M_{CS} will be the output of our decider for L.

In more detail, the polynomial-time decider for L will work as follows. Let x be the input. By Theorem 8.14, there exists a polynomial size circuit C_V that simulates V (note that the circuits take x-variables and u-variables as input). Furthermore, this circuit can be constructed in polynomial time (details omitted). Let $C_{V,x}$ denote C_V with the x-variables fixed to the specific input x we are given (so the input gates of $C_{V,x}$ correspond to the u-variables only). Then observe that $x \in L$ if and only if $C_{V,x}$ is a Yes instance of CIRCUIT-SAT (i.e. $C_{V,x}$ is satisfiable). So to decide whether $x \in L$, we simply feed $C_{V,x}$ into M_{CS}, the polynomial-time decider for CIRCUIT-SAT, and give the same answer as $M_{CS}(\langle C_{V,x} \rangle)$. □

Remark. To show that a language L is NP-hard, by the transitivity of polynomial-time reductions (Exercise 12.8), it suffices to show that $K \leq_T^P L$ for some language K which is known to be NP-hard. In particular, using Theorem 13.9, it suffices to show that CIRCUIT-SAT $\leq_T^P L$.

Theorem 13.10. 3COL *is* NP-complete.

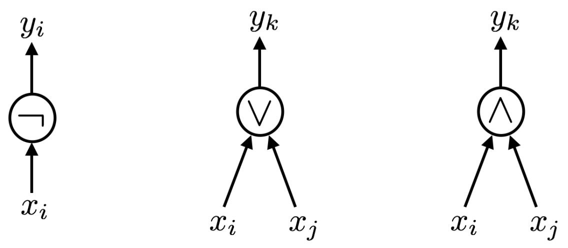
Proof. We have already done all the work to prove that 3COL is NP-complete. First of all, in Proposition 13.2, we have shown that 3COL $\in \mathsf{NP}$. To show that 3COL is NP-hard, by the transitivity of reductions, it suffices to show that CIRCUIT-SAT $\leq_T^P$ 3COL, which we have done in Theorem 12.12. □

Theorem 13.11. 3SAT *is* NP-complete.

Proof Sketch. To show that 3SAT is NP-complete, we have to show that it is in NP and it is NP-hard. To see that it is in NP, we can take the *proof string* u to be a satisfying truth assignment to the variables. If the input formula is indeed satisfiable, then once such a proof string is given, it is easy to check in polynomial time that u is indeed a satisfying assignment by evaluating the truth value of each clause one by one, and checking that every one of them evaluates to True. If the input formula is not satisfiable, then no matter what u is given, it will never be the case that u makes each clause evaluate to True. So the verifier will reject as desired.

To show that 3SAT is NP-hard, by the transitivity of reductions, it suffices to show that CIRCUIT-SAT $\leq_T^P$ 3SAT. Our reduction will be a Karp reduction. Given an instance of CIRCUIT-SAT, i.e. a Boolean circuit C, we will construct an instance of 3SAT, i.e. a Boolean CNF formula φ in which every clause has exactly 3 literals. The reduction will be polynomial-time and will be such that C is a Yes instance of CIRCUIT-SAT (i.e. C is satisfiable) if and only if φ is a Yes instance of 3SAT (i.e. φ is satisfiable).

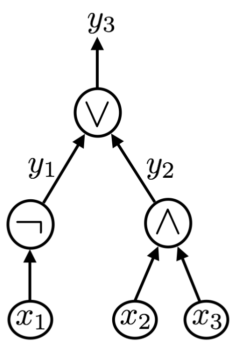
The construction is as follows. A circuit C has three types of gates (excluding the input-gates): NOT, OR, AND.



We will convert each such gate of the circuit C into a 3SAT formula. It is easy to verify that

$$\begin{aligned} y_i = \neg x_i &\iff (x_i \vee y_i) \wedge (\neg x_i \vee \neg y_i), \\ y_k = x_i \vee x_j &\iff (y_k \vee \neg x_i) \wedge (y_k \vee \neg x_j) \wedge (\neg y_k \vee x_i \vee x_j), \\ y_k = x_i \wedge x_j &\iff (\neg y_k \vee x_i) \wedge (\neg y_k \vee x_j) \wedge (y_k \vee \neg x_i \vee \neg x_j). \end{aligned}$$

So the behavior of each gate can be represented using a Boolean formula. As an example, consider the circuit below.



In this case, we would let

$$\begin{aligned}
\text{Clause}_1 &= (x_1 \vee y_1) \wedge (\neg x_1 \vee \neg y_1) \\
\text{Clause}_2 &= (\neg y_2 \vee x_2) \wedge (\neg y_2 \vee x_3) \wedge (y_2 \vee \neg x_2 \vee \neg x_3) \\
\text{Clause}_3 &= (y_3 \vee \neg y_1) \wedge (y_3 \vee \neg y_2) \wedge (\neg y_3 \vee y_1 \vee y_2),
\end{aligned}$$

and the Boolean formula equivalent to the circuit would be

$$\varphi = \text{Clause}_1 \wedge \text{Clause}_2 \wedge \text{Clause}_3 \wedge y_3.$$

This is not quite a 3SAT formula since each clause does not necessarily have exactly 3 literals. However, each clause has at most 3 literals, and every clause in the formula can be converted into a clause with exactly 3 literals by duplicating a literal in the clause if necessary.

This completes the description of how we construct a 3SAT formula from a Boolean circuit. We leave it as an exercise to the reader to verify that C is satisfiable if and only if φ is satisfiable, and that the reduction can be carried out in polynomial time. $\square$

Theorem 13.12. CLIQUE *is* NP-complete*.*

Proof. To show that CLIQUE is NP-complete, we have to show that it is in NP and it is NP-hard. Exercise 13.4 asks you to show that CLIQUE is in NP, so we will show that CLIQUE is NP-hard by presenting a reduction from 3SAT to CLIQUE.

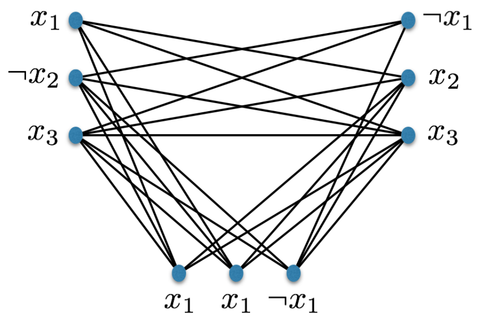
Our reduction will be a Karp reduction. Given an instance of 3SAT, i.e. a Boolean formula φ, we will construct an instance of CLIQUE, $\langle G, k\rangle$ where G is a graph and k is a number, such that φ is a Yes instance of 3SAT (i.e. φ is satisfiable) if and only if $\langle G, k\rangle$ is a Yes instance of CLIQUE (i.e. G contains a k-clique). Furthermore, this construction will be done in polynomial time.

Let

$$\varphi = (a_1 \vee b_1 \vee c_1) \wedge (a_2 \vee b_2 \vee c_2) \wedge \cdots (a_m \vee b_m \vee c_m),$$

where each a_i, b_i and c_i is a literal, be an arbitrary 3SAT formula. Notice that φ is satisfiable if and only if there is a truth assignment to the variables so that each clause has at least one literal set to True. From this formula, we build a graph G as follows. For each clause, we create 3 vertices corresponding to the literals of that clause. So in total the graph has $3m$ vertices. We now specify which vertices are connected to each other with an edge. We do *not* put an edge between two vertices if they correspond to the same clause. We do *not* put an edge between x_i and $\neg x_i$ for any i. Every other pair of vertices is connected with an edge. This completes the construction of the graph. We still need to specify k. We set $k = m$.

As an example, if $\varphi = (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_1 \vee \neg x_1)$, then the corresponding graph is as follows:



We now prove that φ is satisfiable if and only if G has a clique of size m. If φ is satisfiable, then there is an assignment to the variables such that in each clause, there is at least one literal set to True. We claim that the vertices that correspond to these literals form a clique of size m in G. It is clear that the number of such vertices is m. To see that they form a clique, notice that the only way two of these vertices do not share an edge is if they correspond to x_i and $\neg x_i$ for some i. But a satisfying assignment cannot assign True to both x_i and $\neg x_i$.

For the other direction, suppose that the constructed graph G has a clique K of size m. Since there are no edges between two literals if they are in the same clause, there must be exactly one vertex from each clause in K. We claim that we can set the literals corresponding to these vertices to True and therefore show that φ is satisfiable. This claim is easy to see. The only way we could not simultaneously set these literals to True is if two of these literals correspond to x_i and $\neg x_i$ for some i. But there is no edge between such literals, so they cannot both be in the same clique.

This completes the correctness of the reduction. We still have to argue that it can be done in polynomial time. This is rather straightforward. Creating the vertex set of G is clearly polynomial-time since there is just one vertex for each literal of the Boolean formula. Similarly, the edges can be easily added in polynomial time as there are at most $O(m^2)$ many of them. □

Theorem 13.13. IS *is* NP-complete.

Proof. To show that IS is NP-complete, we have to show that it is in NP and it is NP-hard. Exercise 13.4 asks you to show that IS is in NP, so we show that IS is NP-hard. By Theorem 13.12, we know that CLIQUE is NP-hard, and by Theorem 12.9 we know that CLIQUE $\leq_T^P$ IS. By the transitivity of reductions, we conclude that IS is also NP-hard. □

Remark. The collection of reductions that we have shown can be represented as follows:

$$\begin{tikzcd}
& \text{Every } L \in \mathsf{NP} \arrow[d, "\text{Cook-Levin Theorem}"] & \\
& \text{CIRCUIT-SAT} \arrow[dl] \arrow[dr] & \\
\text{3SAT} \arrow[d] & & \text{3COL} \\
\text{CLIQUE} \arrow[d] & & \\
\text{IS} & &
\end{tikzcd}$$

Definition 13.14 (Vertex-cover problem)**.**
Let $G = (V, E)$ be an undirected graph. A subset S of the vertices is called a *vertex cover* if for every edge of the graph, there is a vertex in S that is incident to that edge. In other words, every edge of the graph has at least one end-point in S.

In the *vertex-cover problem*, the input is an undirected graph $G = (V, E)$ and a number $k \in \mathbb{N}^+$, and the output is True if and only if the graph contains a vertex cover of size at most k. We denote this problem by VERTEX-COVER. ■

Exercise 13.15. Show that VERTEX-COVER is NP-complete.

Exercise 13.16. Let 2SAT denote the restriction of SAT in which every clause has exactly 2 literals. Show that 2SAT $\in \mathsf{P}$.

14 Computational Social Choice Theory

Definition 14.1 (Election, voters, alternatives, preference profile, voting rule)**.**
An *election* is specified by 4 objects:

- **Voters:** a set of n *voters* $N = \{1, 2, \dots, n\}$;
- **Alternatives:** a set of m *alternatives* denoted by A;
- **Preference profile:** for each voter, a ranking over the alternatives from rank 1 to rank m;
- **Voting rule:** a function that maps a preference profile to an alternative.

The output of the voting rule is called the *winner* of the election. ■

Definition 14.2 (Pairwise election)**.**
In a *pairwise election* $m = 2$ and an alternative x wins if the majority of voters prefer x over the other alternative. ■

Definition 14.3 (Condorcet winner)**.**
We say that an alternative is a *Condorcet winner* if it beats every other alternative in a pairwise election. ■

Definition 14.4 (Various voting rules)**.**
The following are definitions of various voting rules. [21]

- **Plurality:** Each voter awards one point to their top-ranked alternative. The alternative with the most points is declared the winner.
- **Borda count:** Each voter awards $m - k$ points to their k'th ranked alternative. The alternative with the most points is declared the winner.
- **Plurality with runoff:** There are 2 rounds. In the first round, a plurality rule is applied to identify the top two alternatives. In the second round, a pairwise election is done to determine the winner.
- **Single transferable vote (STV):** There are $m - 1$ rounds. In each round a plurality rule is applied to identify and eliminate the alternative with the lowest points. The alternative that survives every round is selected as the winner.
- **Copeland:** An alternative's score is the number of alternatives it would beat in a pairwise election. The winner of the election is the alternative with the highest score.

[21] We'll assume that ties are broken deterministically according to some order on the alternatives.

- **Dodgson:** Given a preference profile, define the *Dodgson score* of an alternative x as the number of swaps between adjacent alternatives needed in the preference profile in order to make x a Condorcet winner. In Dodgson voting rule, the winner is an alternative with the minimum Dodgson score.

■

Theorem 14.5 (Bartholdi-Tovey-Trick 1989)**.** *Consider the following computational problem. Given as input an election, an alternative x in the election, and a number k, the output is True if and only if the Dodgson score of x is at most k. This problem is* NP*-complete.*

Definition 14.6 (Types of voting rules)**.**
We call a voting rule

- **majority consistent** if given a preference profile such that a majority of the voters rank an alternative x first, then x is the winner of the election;
- **Condorcet consistent** if given a preference profile such that there is an alternative x that beats every other alternative in a pairwise election, then x is the winner of the election;
- **onto** if for any alternative, there is a preference profile such that the alternative wins;
- **dictatorial** if there is a voter v such that no matter what the preference profile is, the winner is v's most preferred alternative;
- **constant** if no matter what the preference profile is, the same alternative is the winner.

The first 3 are considered to be desirable types of voting rules, whereas the last 2 are considered undesirable. ■

Exercise 14.7. Determine whether plurality and Borda count voting rules are Condorcet consistent or not.

Exercise 14.8. What is the relationship between majority consistency and Condorcet consistency, i.e., does one imply the other?

Definition 14.9 (Manipulation, strategy-proof (SP) voting rule)**.**
Consider an election in which alternative x wins. We say that a voter can *manipulate* the voting rule of the election if by changing his preference list, he changes the winner of the election to an alternative y that the voter ranks higher than x. A voting rule is called *strategy-proof* if no voter can manipulate the voting rule. ■

Exercise 14.10. What is the largest value of m for which plurality voting rule is strategy-proof?

Exercise 14.11. Are constant and dictatorial voting rules strategy-proof?

Theorem 14.12 (Gibbard-Satterthwaite)**.** *If $m \geq 3$ then any voting rule that is strategy-proof and onto is dictatorial. Equivalently, any voting rule that is onto and nondictatorial is manipulable.*

Definition 14.13 (r-Manipulation problem)**.**
Let r be some voting rule. In the *r-Manipulation problem*, the input is an election, a voter (called the *manipulator*), and an alternative (called the *preferred candidate*). The output is True if there exists a ranking over the alternatives for the manipulator that makes the preferred candidate a *unique winner* of the election. ■

Remark. Below is a greedy algorithm that can be used to solve the r-Manipulation problem, however, it is not always guaranteed to give the correct answer. The algorithm works by trying to build a ranking of the alternatives for the manipulator, starting with the highest rank and moving down to the lowest rank one by one.

```
Given as input an election, a manipulator $m$, and a preferred candidate $p$:
  • Rank $p$ in the first place for $m$.
  • While there are unranked alternatives:
      – If there is an alternative that can be placed in the next spot
        without preventing $p$ from winning, place this alternative.
      – Otherwise, output False.
  • Output True.
```

Theorem 14.14. *The greedy algorithm above is a polynomial-time algorithm that correctly solves the r-Manipulation problem for*

$$r \in \{\textit{plurality, Borda count, plurality with runoff, Copeland}\}.$$

Theorem 14.15 (Bartholdi-Orlin 1991)**.** *The r-Manipulation problem is* NP-complete *for r being the single transferable voting (STV) rule.*

15 Approximation Algorithms

Definition 15.1 (Optimization problem)**.**
A *minimization optimization problem* is a function $f : \Sigma^* \times \Sigma^* \to \mathbb{R}^{\geq 0} \cup \{\text{no}\}$. If $f(x, y) = \alpha \in \mathbb{R}^{\geq 0}$, we say that y is a *solution* to x with value α. If $f(x, y) = \text{no}$, then y is not a solution to x. We let $\text{OPT}_f(x)$ denote the minimum $f(x, y)$ among all solutions y to x.[22] We drop the subscript f, and just write $\text{OPT}(x)$, when f is clear from the context.

In a *maximization optimization problem*, $\text{OPT}_f(x)$ is defined using a maximum.

We say that an optimization problem is computable if there is an algorithm such that given as input $x \in \Sigma^*$, it produces as output a solution y to x such that $f(x, y) = \text{OPT}(x)$. We often describe an optimization problem by describing the input and a corresponding output (i.e. a solution y such that $f(x, y) = \text{OPT}(x)$). ■

Definition 15.2 (Optimization version of the Vertex-cover problem)**.**
The decision version of the vertex-cover problem was defined in Definition 13.14. In the optimization version, we are given as input an undirected graph $G = (V, E)$, and the output is a vertex cover of minimum size. We refer to the optimization version of the problem as MIN-VC.

Using the notation in Definition 15.1, the corresponding function f is defined as follows. Let $x = \langle G \rangle$ for some graph G. If y represents a vertex cover in G, then $f(x, y)$ is defined to be the size of the set that y represents. Otherwise $f(x, y) = \text{no}$. ■

Remark. Each decision problem that we have defined in the beginning of Section 12 has a natural optimization version.

Remark. The complexity class NP is a set of decision problems. Similarly, the set of NP-hard problems is a set of decision problems. Given an optimization problem f, suppose it is the case that if f can be computed in polynomial time, then every language in NP can be decided in polynomial time. In this case, we will abuse the definition of NP-hard and say that f is NP-hard.

Definition 15.3 (Approximation algorithm)**.**

- Let f be a minimization optimization problem and let $\alpha > 1$ be some parameter. We say that an algorithm A is an α-approximation algorithm for f if for all instances x, $f(x, A(x)) \leq \alpha \cdot \text{OPT}(x)$.
- Let f be a maximization optimization problem and let $0 < \beta < 1$ be some parameter. We say that an algorithm A is a β-approximation algorithm for f if for all instances x, $f(x, A(x)) \geq \beta \cdot \text{OPT}(x)$.

■

[22] There are a few technicalities. We will assume that f is such that every x has at least one solution y, and that the minimum always exists.

Remark. Note that when showing that a certain minimization problem has an α-approximation algorithm, you need to first present an algorithm, and then argue that for any input, the value of the output produced by the algorithm is within a factor α of the optimum. For the latter, it is usually hard to know exactly what the optimum value would be. So a good strategy is to find a convenient lower bound on the optimum, and then argue that the output of the algorithm is within a factor α of this lower bound. For example, for the MIN-VC problem, we will use Lemma 15.4 below to say that the optimum (the size of the minimum size vertex cover) is at least the size of a matching.

The same principle of course applies to maximization problems as well. For maximization problems, we want to find a convenient upper bound on the optimum.

Lemma 15.4. *Given a graph $G = (V, E)$, let $M \subseteq E$ be a matching in G, and let $S \subset V$ be a vertex cover in G. Then, $|S| \geq |M|$.*

Proof. Observe that in a vertex cover, one vertex cannot be incident to more than one edge of a matching. Therefore, a vertex cover must have at least $|M|$ vertices in order to touch every edge of M. $\square$

Theorem 15.5. *There is a polynomial-time 2-approximation algorithm for the optimization problem* MIN-VC.

Proof. We start by presenting the algorithm:

```
On input an undirected graph $G = (V, E)$:
  • Let $M = \emptyset$.
  • For each edge $e \in E$ do:
      – If $M \cup \{e\}$ is a matching, let $M = M \cup \{e\}$.
  • Let $S$ be the set of vertices incident to an edge in $M$.
  • Output $S$.
```

We need to argue that the algorithm runs in polynomial time and that it is a 2-approximation algorithm. It is easy to see that the running-time is polynomial. We have a loop that repeats $|E|$ times, and in each iteration, we do at most $O(|E|)$ steps. So the total cost of the loop is $O(|E|^2)$. The construction of S takes $O(|V|)$ steps, so in total, the algorithm runs in polynomial time.

Now we argue that the algorithm is a 2-approximation algorithm. To do this, we need to argue that

(i) S is indeed a valid vertex-cover,

(ii) if S^* is a vertex cover of minimum size, then $|S| \leq 2|S^*|$.

For (i), notice that the M constructed by the algorithm is a maximal matching, i.e., there is no edge $e \in E$ such that $M \cup \{e\}$ is a matching. This implies that the set S is indeed a valid vertex-cover, i.e., it touches every edge in the graph. For (ii), a convenient lower bound on $|S^*|$ is given by Lemma 15.4: for any matching M, $|S^*| \geq |M|$. Observe that $|S| = 2|M|$. Putting the two together, we get $|S| \leq 2|S^*|$ as desired. □

Definition 15.6 (Max-cut problem)**.**
Let $G = (V, E)$ be a graph. Given a 2-coloring of the vertices, we say that an edge $e = \{u, v\}$ is *cut* if u and v are colored differently. In the *max-cut problem*, the input is a graph G, and the output is a 2-coloring of the vertices that maximizes the number of cut edges. We denote this problem by MAX-CUT. ■

Theorem 15.7. *There is a polynomial-time $\frac{1}{2}$-approximation algorithm for the optimization problem* MAX-CUT.

Proof. Here is the algorithm:

```
On input an undirected graph $G = (V, E)$:
  • Color every vertex with the same color. Let $c = 0$.
    ($c$ stores the number of cut edges.)
  • Repeat:
      – If there is a vertex such that changing its color increases
        the number of cut edges, change the color of that vertex.
        Update $c$.
      – Else, output the current coloring of the vertices.
```

We first argue that the algorithm runs in polynomial time. Note that the maximum number of cut edges possible is $|E|$. Therefore the loop repeats at most $|E|$ times. In each iteration, the number of steps we need to take is at most $O(|V|^2)$ since we can just go through every vertex once, and for each one of them, we can check all the edges incident to it. So in total, the number of steps is polynomial in the input length.

We now show that the algorithm is a $\frac{1}{2}$-approximation algorithm. It is clear that the algorithm returns a valid coloring of the vertices. Therefore, if c is the number of cut edges returned by the algorithm, all we need to show is that $c \geq \frac{1}{2}\text{OPT}(\langle G \rangle)$. We will use the trivial upper bound of m (the total number of edges) on $\text{OPT}(\langle G \rangle)$, i.e. $\text{OPT}(\langle G \rangle) \leq m$. So our goal will be to show that $c \geq \frac{1}{2}m$.

Observe that in the coloring that the algorithm returns, for each $v \in V$, at least $\deg(v)/2$ edges incident to v are cut edges. To see this, notice that if there was a vertex such that this was not true, then we could change the color of the vertex to obtain a solution that has strictly more cut edges, so our algorithm would have changed the color of this vertex. From Theorem 9.5, we know that when we count the number of edges of a graph by adding up the degrees of all the vertices, we count every edge exactly twice, i.e. $2m = \sum_v \deg(v)$. In a similar way we can count the number of cut edges, which implies $2c \geq \sum_v \deg(v)/2$. The RHS of this inequality is equal to m, so we have $c \geq \frac{1}{2}m$, as desired. □

Definition 15.8 (Traveling salesperson problem (TSP))**.**
In the *Traveling salesperson problem*, the input is a connected graph $G = (V, E)$ together with edge costs $c : E \to \mathbb{N}$. The output is a Hamiltonian cycle that minimizes the total cost of the edges in the cycle, if one exists.

A popular variation of this problem is called *Metric-TSP*. In this version of the problem, instead of outputting a Hamiltonian cycle of minimum cost, we output a "tour" that starts and ends at the same vertex and visits every vertex of the graph at least once (so the tour is allowed to visit a vertex more than once). In other words, the output is a sequence of vertices $v_{i_1}, v_{i_2}, \ldots, v_{i_k}, v_{i_1}$ such that the vertices are not necessarily unique, all the vertices of the graph appear in the sequence, any two consecutive vertices in the sequence form an edge, and the total cost of the edges is minimized. ■

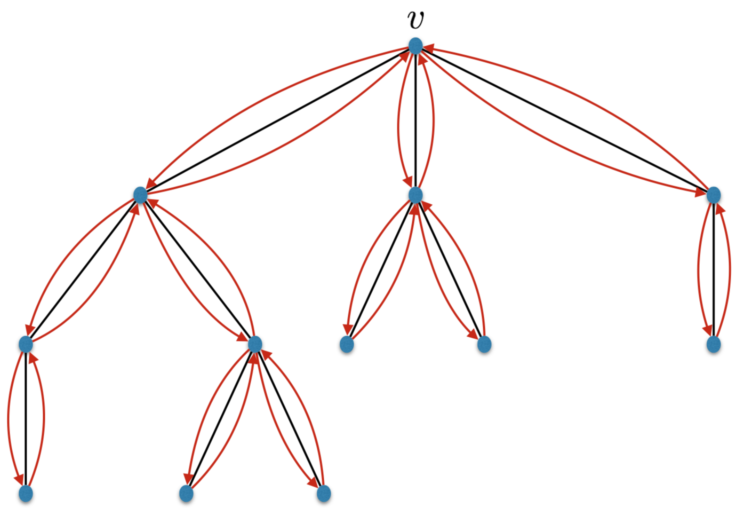
Theorem 15.9. *There is a polynomial-time* 2*-approximation algorithm for the optimization problem* Metric-TSP*.*

Proof. The algorithm first computes a minimum spanning tree, and then does a depth-first search on the tree starting from an arbitrary vertex. More precisely:

```
On input a connected graph $G = (V, E)$ together with edge costs $c : E \to \mathbb{N}$:
  • Compute a MST $T$ of $G$.
  • Let $v$ be an arbitrary vertex in $V$.
  • Let $S$ be an empty sequence.
  • Run DFS($T$, $v$).
      DFS on input $G = (V, E)$ and $v \in V$:
        – Mark $v$ as "visited".
        – Add $v$ to $S$.
        – For each $u \in N(v)$:
            ∗ If $u$ is not marked "visited", then run DFS($G$, $u$).
            ∗ Add $v$ to $S$.
  • Output $S$.
```

This is clearly a polynomial-time algorithm since computing a minimum spanning tree (Theorem 10.15) and doing a depth-first search (Exercise 10.2) both take polynomial time.

To see that the algorithm outputs a valid tour, note that it visits every vertex (since T is a spanning tree), and it starts and ends at the same vertex v.



Let $c(S)$ denote the total cost of the tour that the algorithm outputs. Let S^* be a optimal solution so that $c(S^*) = \text{OPT}(\langle G, c\rangle)$. Our goal is to show that $c(S) \leq 2c(S^*)$. The graph induced by S^* is a connected graph on all of the vertices. Let T^* be a spanning tree within this induced graph. It is clear that $c(S^*) \geq c(T^*)$ and this will be the convenient lower bound we use on the optimum. In other words, we'll show $c(S) \leq 2c(T^*)$. Clearly $c(S) = 2c(T)$ since the tour uses every edge of T exactly twice. Furthermore, since T is a *minimum* spanning tree, $c(T) \leq c(T^*)$. Putting these together, we have $c(S) \leq 2c(T^*)$, as desired.

$\square$

16 Online Algorithms

Definition 16.1 (Online problem, online algorithm, c-competitiveness)**.**
An *online problem* is a collection of 3 objects:

- a set R called *requests*;
- a set A called *actions*;
- a function $\text{cost} : \mathcal{I} \to \mathbb{R}^+ \cup \{\infty\}$ where $\mathcal{I} = \bigcup_{n\geq 1} R^n \times A^n$.

For a request sequence $r \in R^n$, we let $\text{OPT}(r) = \min_{a\in A^n} \text{cost}(r, a)$.[23]

An *online algorithm* ALG is determined by a function $f : \mathcal{R} \to A$ where $\mathcal{R} = \bigcup_{n\geq 1} R^n$. Given as input some request sequence $r \in R^n$, the algorithm produces the action sequence $f(r_1), f(r_1r_2), \ldots, f(r_1r_2 \ldots r_n)$, which we'll denote by $\text{ALG}(r)$. The cost of the output is $\text{cost}(r, \text{ALG}(r))$.

Let $c > 1$. We say that ALG is a *c-competitive algorithm* if $\text{cost}(r, \text{ALG}(r)) \leq c \cdot \text{OPT}(r)$ for every r. ■

Definition 16.2 (Ski rental problem)**.**
In the *ski rental online problem*, you are on a ski trip, and each sunny day, you have to decide whether to rent skis for \$1 or buy skis for \$B dollars (which you can then use for the remaining of the trip). The trip ends when there is a non-sunny day. The total cost of the trip is the number of days you rent skis plus B if you end up buying skis. The goal is to minimize the cost of the trip.

Using the notation in Definition 16.1, we can define the ski rental problem as follows. Let $R = \{\text{go skiing}\}$ and $A = \{\text{rent, buy, use bought skis}\}$. Let (r, a) be an input to the problem which contains i "rent" actions and j "buy" actions. Then we define $\text{cost}(r, a) = i + Bj$. If "use bought skis" appears before any "buy" action, we'll assume the cost is ∞.

We'll denote this problem by $\text{SKI}(B)$. ■

Theorem 16.3. *There is a c-competitive algorithm for* $\text{SKI}(B)$ *where $c = (2B - 1)/B$. Furthermore, this c is the best possible, i.e., there is no c'-competitive algorithm for* $\text{SKI}(B)$ *with $c' < c$.*

Proof. Observe that an optimum (offline) solution must be of the form "rent for t days, and then buy on day $t+1$" for some value t. In particular, let s be the number of sunny days. Then if $s < B$, the optimum solution would rent every day and the cost would be s. If $s > B$, the optimum solution would buy on the first day and the cost would be B. If $s = B$, both renting every day or buying the first day results in cost B, therefore both choices are equally good.

Now consider the following online algorithm: rent for $B - 1$ days, and then buy on day B. There are two cases to consider: (i) $s \leq B - 1$, (ii) $s \geq B$. In case (i), the cost

[23] One can also define a maximization version of an online algorithm. We'll restrict ourselves to the minimization version in these notes.

incurred by this algorithm is equal to the optimum cost. In case (ii), the cost incurred by the algorithm is $(B-1)+B$, whereas the optimum solution has cost B. The ratio is $(2B-1)/B$, therefore the algorithm is c-competitive where $c=(2B-1)/B$.

It remains to show that this ratio is best possible. As before, the best online algorithm must be of the form "rent for t days, and then buy on day $t+1$" for some value t. The algorithm may choose a t such that $t \leq B-2$, $t \geq B$, or $t = B-1$. The case of $t=B-1$ is analyzed above, and achieves $c=(2B-1)/B$. We'll argue that in the remaining cases, c cannot be better than 2, which is worse than $(2B-1)/B$.

First consider the case where $t \leq B-2$. Suppose we are given the instance $s=t+1$. Since $s<B$, the optimum (offline) solution has cost $s=t+1$. However, the optimum online solution has cost $t+B \geq t+(t+2) \geq 2t+2$. So the cost ratio is at least 2. Let's now consider the case $t \geq B$. Again, we look at the instance $s=t+1$. The optimum (offline) solution has cost B, whereas the optimum online solution has cost $t+B \geq 2B$. So the cost ratio is at least 2 in this case as well. $\square$

Definition 16.4 (Paging problem)**.**
In the *paging problem*, we have a hard drive that can hold N pages. We'll denote the set of pages as $\{1,2,\ldots,N\}$. We also have a memory which holds k pages. We'll assume that the memory initially contains the pages $1,2,\ldots,k$. The request sequence is a sequence of pages $p_1,p_2,\ldots,p_n$. If a page is already in the memory, we have a hit, otherwise we have a miss. If we have a miss, we have to add the page to the memory and evict a page that was in the memory. The goal is to minimize the number of misses. We'll denote this problem by PAGE(N,k). ■

Exercise 16.5. Express PAGE(N,k) using the notation in Definition 16.1.

Theorem 16.6. *There is a k-competitive algorithm for* PAGE(N,k)*. Furthermore, this is the best possible, i.e., there is no c-competitive algorithm for* PAGE(N,k) *with $c<k$.*

Proof. Consider the following online algorithm for PAGE(N,k): given a page, if it is a miss, we evict from memory a page that is least recently requested. This algorithm is called *least recently used* (LRU). We will first show that LRU is k-competitive.

We ignore all the requests until the first one that is a miss. We divide the request sequence into *phases*. The phases partition the request sequence, and each phase consists of a subsequence of consecutive requests. Phase 1 starts with the first request (which we assume is a miss). Each phase has the property that it is the longest possible subsequence with at most k requests of distinct pages. In other words, within a phase, if we get a request for a $(k+1)$'st distinct page, we have to end that phase and start the next phase. Below is an example with $k=3$.

4 1 2 1	5 3 4 5	1 2 3
Phase 1	Phase 2	Phase 3

Let m be the number of phases, and denote by p^i_j the j'th distinct page in phase i. Note that $p^i_1, p^i_2, \ldots, p^i_k, p^{i+1}_1$ are all distinct pages. So if there are no misses on requests $p^i_2, p^i_3, \ldots, p^i_k$, there will be a miss on p^{i+1}_1. This implies that each phase must contain at least one miss. This also includes the first phase since by definition, p^1_1 is a miss. Therefore, in an optimal solution, the number of misses is at least m. On the other hand, it is not hard to see that LRU misses at most once on each distinct page in a phase, so in total the number of misses is at most km (we leave the verification of this observation as an exercise to the reader). Thus, the ratio between LRU and an optimal solution is at most k, as desired.

We now show that this ratio is the best possible. Fix some online algorithm. Consider an instance in which the request sequence is such that each requested page is in $\{1, 2, \ldots, k, k+1\}$, and each request produces a miss for that algorithm. So for any n, we can make the algorithm miss n times on some request sequence of length n. On the other hand, in an optimal (offline) solution, once there is a miss, we can make sure that there won't be a miss in the next $k-1$ requests. To accomplish this, once there is a miss, the solution evicts the page i with the property that every other page in the memory will be requested before i. Since there is at most one miss in every k requests, the ratio between the cost of the best online algorithm and the best offline algorithm is at least k. □

Exercise 16.7. Consider the following online algorithm for PAGE(N, k): given a page, if it is a miss, we evict from memory a page that has been in the memory for the longest time since the last time it was inserted into memory. This algorithm is called *first in first out* (FIFO). Show that FIFO is a k-competitive online algorithm for PAGE(N, k).

Exercise 16.8. Consider the following online algorithm for PAGE(N, k): given a page, if it is a miss, we evict from memory a page that has been in the memory for the shortest time since the last time it was inserted into memory. This algorithm is called *last in first out* (LIFO). Show that there is no c such that LIFO is c-competitive? If we applied the analysis for LRU to LIFO, where does it break down, i.e., why doesn't it show that LIFO is k-competitive?

Exercise 16.9. Consider the following online algorithm for PAGE(N, k): given a page, if it is a miss, we evict from memory a page that is least frequently requested. This algorithm is called *least frequently used* (LFU). Show that there is no c such that LFU is c-competitive? If we applied the analysis for LRU to LFU, where does it break down, i.e., why doesn't it show that LFU is k-competitive?

Definition 16.10 (List update problem).
In the *list update problem*, we have a list of n items. There are n requests: "access x" for each item x. The cost of accessing x is the position of x in the list. For a request "access x", there are i valid actions $a_1, \ldots, a_i$, where i is the position of x in the list. Action a_j corresponds to moving x to position j in the list. The goal is to minimize the total cost. ■

Exercise 16.11. Here are 3 algorithms for the list update problem.

- *Transpose:* Given the request "access x", move x one position down (if it is not in the first position).
- *Move to front:* Given the request "access x", move x to the first position.
- *Frequency counter:* Keep track of how many times an element is requested. Given the request "access x", move x past elements that were requested less frequently.

One of these algorithms is c-competitive for some constant c. Determine which one it is. (Hint: Find it by eliminating two of the options.)

17 Probability I: The Basics

Definition 17.1 (Finite probability space, sample space, probability distribution)**.**
A *finite probability space* is a tuple $(\Omega, \mathbf{Pr})$, where

- Ω is a non-empty finite set called the *sample space*;
- $\mathbf{Pr} : \Omega \to [0, 1]$ is a function, called the *probability distribution*, with the property that $\sum_{\ell \in \Omega} \mathbf{Pr}[\ell] = 1$.

■

Remark. In this course, we'll usually restrict ourselves to finite sample spaces. In cases where we need an infinite Ω, the above definition generalizes naturally.

Exercise 17.2. How would you model a roll of a single 6-sided die using Definition 17.1? How about roll of two dice? How about roll of a die and a coin toss together?

Definition 17.3 (Uniform distribution)**.**
If a probability distribution $\mathbf{Pr} : \Omega \to [0, 1]$ is such that $\mathbf{Pr}[\ell] = 1/|\Omega|$ for all $\ell \in \Omega$, then we call it a *uniform distribution*. ■

Definition 17.4 (Event)**.**
Let $(\Omega, \mathbf{Pr})$ be a probability space. Any subset $E \subseteq \Omega$ is called an *event*. We abuse notation and write $\mathbf{Pr}[E]$ to denote $\sum_{\ell \in E} \mathbf{Pr}[\ell]$. Using this notation, $\mathbf{Pr}[\emptyset] = 0$ and $\mathbf{Pr}[\Omega] = 1$. We use the notation $\bar{E}$ to denote the event $\Omega \backslash E$. ■

Exercise 17.5. Suppose we roll two 6-sided dice. How many different events are there? Write down the event corresponding to "we roll a double" and determine its probability.

Exercise 17.6. Suppose we roll a 3-sided die and see the number d. We then roll a d-sided die. How many different events are there? Write down the event corresponding to "the second roll is a 2" and determine its probability.

Exercise 17.7. Let A and B be two events. Prove the following.

- If $A \subseteq B$, then $\mathbf{Pr}[A] \leq \mathbf{Pr}[B]$.
- $\mathbf{Pr}[\bar{A}] = 1 - \mathbf{Pr}[A]$.
- $\mathbf{Pr}[A \cup B] = \mathbf{Pr}[A] + \mathbf{Pr}[B] - \mathbf{Pr}[A \cap B]$.

Definition 17.8 (Disjoint events)**.**
We say that two events A and B are *disjoint* if $A \cap B = \emptyset$. ■

Exercise 17.9 (Union bound)**.** Let $A_1, A_2, \ldots, A_n$ be events. Then

$$\mathbf{Pr}[A_1 \cup A_2 \cup \ldots \cup A_n] \leq \mathbf{Pr}[A_1] + \mathbf{Pr}[A_2] + \ldots + \mathbf{Pr}[A_n].$$

We get equality if and only if the A_i's are pairwise disjoint.

Definition 17.10 (Conditional probability)**.**
Let A and B be two events such that $\mathbf{Pr}[B] \neq 0$. The *conditional probability of A given B*, denoted $\mathbf{Pr}[A \mid B]$, is defined as

$$\mathbf{Pr}[A \mid B] = \frac{\mathbf{Pr}[A \cap B]}{\mathbf{Pr}[B]}.$$

■

Exercise 17.11. Suppose we roll a 3-sided die and see the number d. We then roll a d-sided die. We are interested in the probability that the first roll was a 1 given that the second roll was a 1. First express this probability using the notation of conditional probability and then determine what the probability is.

Proposition 17.12 (Chain rule)**.** *Let $n \geq 2$ and let $A_1, A_2, \ldots, A_n$ be events. Then*

$$\mathbf{Pr}[A_1 \cap \ldots \cap A_n] = \mathbf{Pr}[A_1] \cdot \mathbf{Pr}[A_2 \mid A_1] \cdot \mathbf{Pr}[A_3 \mid A_1 \cap A_2] \cdots \mathbf{Pr}[A_n \mid A_1 \cap A_2 \cap \ldots \cap A_{n-1}].$$

Proof. We prove the proposition by induction on n. The base case with two events follows directly from the definition of conditional probability. Let $A = A_n$ and $B = A_1 \cap \ldots \cap A_{n-1}$. Then

$$\begin{aligned}
\mathbf{Pr}[A_1 \cap \ldots \cap A_n] &= \mathbf{Pr}[A \cap B] \\
&= \mathbf{Pr}[B] \cdot \mathbf{Pr}[A \mid B] \\
&= \mathbf{Pr}[A_1 \cap \ldots \cap A_{n-1}] \cdot \mathbf{Pr}[A_n \mid A_1 \cap \ldots \cap A_{n-1}],
\end{aligned}$$

where we used the definition of conditional probability for the second equality. Applying the induction hypothesis to $\mathbf{Pr}[A_1 \cap \ldots \cap A_{n-1}]$ gives the desired result. □

Proposition 17.13 (Bayes' rule)**.** *Let A and B be events. Then,*

$$\mathbf{Pr}[A \mid B] = \frac{\mathbf{Pr}[A] \cdot \mathbf{Pr}[B \mid A]}{\mathbf{Pr}[B]}.$$

Proof. Since by definition $\mathbf{Pr}[B \mid A] = \mathbf{Pr}[A \cap B] / \mathbf{Pr}[A]$, the RHS of the equality above simplifies to $\mathbf{Pr}[A \cap B] / \mathbf{Pr}[B]$. This, by definition, is $\mathbf{Pr}[A \mid B]$. □

Proposition 17.14 (Law of total probability)**.** *Let $A_1, A_2, \ldots, A_n, B$ be events such that the A_i's form a partition of the sample space Ω. Then*

$$\mathbf{Pr}[B] = \mathbf{Pr}[B \cap A_1] + \mathbf{Pr}[B \cap A_2] + \cdots + \mathbf{Pr}[B \cap A_n].$$

Equivalently,

$$\mathbf{Pr}[B] = \mathbf{Pr}[A_1] \cdot \mathbf{Pr}[B \mid A_1] + \mathbf{Pr}[A_2] \cdot \mathbf{Pr}[B \mid A_2] + \cdots + \mathbf{Pr}[A_n] \cdot \mathbf{Pr}[B \mid A_n].$$

Exercise 17.15. Prove the above proposition.

Exercise 17.16. There are 2 bins. Bin 1 contains 6 red balls and 4 blue balls. Bin 2 contains 3 red balls and 7 blue balls. We choose a bin uniformly at random, and then choose one of the balls in that bin uniformly at random. Calculate the probability that the chosen ball is red using Proposition 17.14.

Definition 17.17 (Independent events)**.**

- Let A and B be two events. We say that A and B are *independent* if $\mathbf{Pr}[A \cap B] = \mathbf{Pr}[A] \cdot \mathbf{Pr}[B]$. Note that if $\mathbf{Pr}[B] \neq 0$, then this is equivalent to $\mathbf{Pr}[A \mid B] = \mathbf{Pr}[A]$. If $\mathbf{Pr}[A] \neq 0$, it is also equivalent to $\mathbf{Pr}[B \mid A] = \mathbf{Pr}[B]$.

- Let $A_1, A_2, \ldots, A_n$ be events with non-zero probabilities. We say that $A_1, \ldots, A_n$ are *independent* if for any subset $S \subseteq \{1, 2, \ldots, n\}$,
$$\mathbf{Pr}\left[\bigcap_{i \in S} A_i\right] = \prod_{i \in S} \mathbf{Pr}[A_i].$$

■

18 Probability II: Random Variables

18.1 Basics of random variables

Definition 18.1 (Random variable)**.**
A *random variable* is a function $X : \Omega \to \mathbb{R}$. ■

Remark. Note that a random variable is just a labeling of the elements in Ω with some numbers. One can think of this as a transformation of the original sample space into one that contains only numbers. If the probability distribution $\mathbf{Pr}[\cdot]$ is known, then this allows us, for example, to take a *weighted average* of the elements in Ω, where the weights correspond to the probabilities of the elements (if the distribution is uniform, the weighted average is just the regular average). This is called the *expectation* of the random variable and is formally defined in Definition 18.6.

Notation 18.2. Let X be a random variable and $x \in \mathbb{R}$ be some real value. We use

$$\begin{aligned}
X = x &\quad \text{to denote the event } \{\ell \in \Omega : X(\ell) = x\},\\
X \leq x &\quad \text{to denote the event } \{\ell \in \Omega : X(\ell) \leq x\},\\
X \geq x &\quad \text{to denote the event } \{\ell \in \Omega : X(\ell) \geq x\},\\
X < x &\quad \text{to denote the event } \{\ell \in \Omega : X(\ell) < x\},\\
X > x &\quad \text{to denote the event } \{\ell \in \Omega : X(\ell) > x\}.
\end{aligned}$$

For example, $\mathbf{Pr}[X = x]$ denotes $\mathbf{Pr}[\{\ell \in \Omega : X(\ell) = x\}]$. More generally, for $S \subseteq \mathbb{R}$, we use

$$X \in S \quad \text{to denote the event } \{\ell \in \Omega : X(\ell) \in S\}.$$

Exercise 18.3. Suppose we roll two 6-sided dice. Let X be the random variable that denotes the sum of the numbers we see. Explicitly write down the input-output pairs for the function X. Calculate $\mathbf{Pr}[X \geq 7]$.

Remark. Given some probability space $(\Omega, \mathbf{Pr})$ and a random variable $X : \Omega \to \mathbb{R}$, we often forget about the original sample space and consider the sample space to be the range of X, $\text{range}(X) = \{X(\ell) : \ell \in \Omega\}$.

Definition 18.4 (Probability mass function of a random variable)**.**
Let $X : \Omega \to \mathbb{R}$ be a random variable. The *probability mass function* of X is a function $p_X : \mathbb{R} \to [0, 1]$ such that for any $x \in \mathbb{R}$, $p_X(x) = \mathbf{Pr}[X = x]$. ■

Exercise 18.5. Verify the following:

- $\sum_{x \in \text{range}(X)} p_X(x) = 1$,
- for $S \subseteq \mathbb{R}$, $\mathbf{Pr}[X \in S] = \sum_{x \in S} p_X(x)$.

Remark. Related to the previous remark, we sometimes "define" a random variable by just specifying its probability mass function. In particular we make no mention of the underlying sample space.

Definition 18.6 (Expected value of a random variable)**.**
Let X be a random variable. The *expected value* of X, denoted $\mathbf{E}[X]$, is defined as follows:

$$\mathbf{E}[X] = \sum_{\ell \in \Omega} \mathbf{Pr}[\ell] \cdot X(\ell).$$

Equivalently,

$$\mathbf{E}[X] = \sum_{x \in \text{range}(X)} \mathbf{Pr}[X = x] \cdot x,$$

where $\text{range}(X) = \{X(\ell) : \ell \in \Omega\}$. ■

Exercise 18.7. Prove that the above two expressions for $\mathbf{E}[X]$ are equivalent.

Exercise 18.8. Suppose we roll two 6-sided dice. Let X be the random variable that denotes the sum of the numbers we see. Calculate $\mathbf{E}[X]$.

Proposition 18.9 (Linearity of expectation)**.** *Let X and Y be two random variables, and let $c_1, c_2 \in \mathbb{R}$ be some constants. Then $\mathbf{E}[c_1 X + c_2 Y] = c_1 \mathbf{E}[X] + c_2 \mathbf{E}[Y]$.*

Proof. Define the random variable Z as $Z = c_1 X + c_2 Y$. Then using the definition of expected value, we have

$$\begin{aligned}
\mathbf{E}[c_1 X + c_2 Y] &= \mathbf{E}[Z] \\
&= \sum_{\ell \in \Omega} \mathbf{Pr}[\ell] \cdot Z(\ell) \\
&= \sum_{\ell \in \Omega} \mathbf{Pr}[\ell] \cdot (c_1 X(\ell) + c_2 Y(\ell)) \\
&= \sum_{\ell \in \Omega} \mathbf{Pr}[\ell] \cdot c_1 X(\ell) + \mathbf{Pr}[\ell] \cdot c_2 Y(\ell) \\
&= \left(\sum_{\ell \in \Omega} \mathbf{Pr}[\ell] \cdot c_1 X(\ell) \right) + \left(\sum_{\ell \in \Omega} \mathbf{Pr}[\ell] \cdot c_2 Y(\ell) \right) \\
&= c_1 \left(\sum_{\ell \in \Omega} \mathbf{Pr}[\ell] \cdot X(\ell) \right) + c_2 \left(\sum_{\ell \in \Omega} \mathbf{Pr}[\ell] \cdot Y(\ell) \right) \\
&= c_1 \mathbf{E}[X] + c_2 \mathbf{E}[Y],
\end{aligned}$$

as desired. □

Corollary 18.10. *Let $X_1, X_2, \ldots, X_n$ be random variables, and $c_1, c_2, \ldots, c_n \in \mathbb{R}$ be some constants. Then*

$$\mathbf{E}[c_1X_1 + c_2X_2 + \ldots + c_nX_n] = c_1\,\mathbf{E}[X_1] + c_2\,\mathbf{E}[X_2] + \ldots + c_n\,\mathbf{E}[X_n].$$

In particular, when all the c_i's are 1, we get

$$\mathbf{E}[X_1 + X_2 + \ldots + X_n] = \mathbf{E}[X_1] + \mathbf{E}[X_2] + \ldots + \mathbf{E}[X_n].$$

Exercise 18.11. Suppose we roll three 10-sided dice. Let X be the sum of the three values we see. Calculate $\mathbf{E}[X]$.

Exercise 18.12. Let X and Y be random variables. Is it always true that $\mathbf{E}[XY] = \mathbf{E}[X]\,\mathbf{E}[Y]$?

Definition 18.13 (Indicator random variable)**.**
Let $E \subseteq \Omega$ be some event. The *indicator random variable* with respect to E is denoted by I_E and is defined as

$$I_E(\ell) = \begin{cases} 1 & \text{if } \ell \in E, \\ 0 & \text{otherwise.} \end{cases}$$

■

Proposition 18.14. *Let E be an event. Then $\mathbf{E}[I_E] = \mathbf{Pr}[E]$.*

Proof. By the definition of expected value,

$$\begin{aligned} \mathbf{E}[I_E] &= \mathbf{Pr}[I_E = 1] \cdot 1 + \mathbf{Pr}[I_E = 0] \cdot 0 \\ &= \mathbf{Pr}[I_E = 1] \\ &= \mathbf{Pr}[\{\ell \in \Omega : I_E(\ell) = 1\}] \\ &= \mathbf{Pr}[\{\ell \in \Omega : \ell \in E\}] \\ &= \mathbf{Pr}[E]. \end{aligned}$$

□

Remark. Suppose that you are interested in computing $\mathbf{E}[X]$ for some random variable X. If you can write X as a sum of indicator random variables, i.e., if $X = \sum_j I_{E_j}$ where I_{E_j} are indicator random variables, then by linearity of expectation,

$$\mathbf{E}[X] = \mathbf{E}\left[\sum_j I_{E_j}\right] = \sum_j \mathbf{E}[I_{E_j}].$$

Furthermore, by Proposition 18.14, we know $\mathbf{E}[I_{E_j}] = \mathbf{Pr}[E_j]$. Therefore $\mathbf{E}[X] = \sum_j \mathbf{Pr}[E_j]$. This often provides an extremely convenient way of computing $\mathbf{E}[X]$. **This combination of indicator random variables together with linearity expectation is one of the most useful tricks in probability theory.**

Exercise 18.15. There are n balls and n bins. For each ball, you pick one of the bins uniformly at random and drop the ball in that bin. What is the expected number of balls in bin 1? What is the expected number of empty bins?

Exercise 18.16. Suppose you randomly color the vertices of the complete graph on n vertices one of k colors. What is the expected number of paths of length c (where we assume $c \geq 3$) such that no two adjacent vertices on the path have the same color?

Definition 18.17 (Conditional expectation)**.**
Let X be a random variable and E be an event. The *conditional expectation* of X given the event E, denoted by $\mathbf{E}[X \mid E]$, is defined as

$$\mathbf{E}[X \mid E] = \sum_{x \in \text{range}(X)} x \cdot \mathbf{Pr}[X = x \mid E].$$

■

Proposition 18.18 (Law of total expectation)**.** *Let X be a random variable and $A_1, A_2, \dots, A_n$ be events that partition the sample space Ω. Then*

$$\mathbf{E}[X] = \mathbf{E}[X \mid A_1] \cdot \mathbf{Pr}[A_1] + \mathbf{E}[X \mid A_2] \cdot \mathbf{Pr}[A_2] + \cdots + \mathbf{E}[X \mid A_n] \cdot \mathbf{Pr}[A_n].$$

Exercise 18.19. Prove the above proposition.

Exercise 18.20. We first roll a 4-sided die. If we see the value d, we then roll a d-sided die. Let X be the sum of the two values we see. Calculate $\mathbf{E}[X]$.

Definition 18.21 (Independent random variables)**.**
Two random variables X and Y are *independent* if for all $x, y \in \mathbb{R}$, the events $X = x$ and $Y = y$ are independent. ■

Exercise 18.22. Show that if $X_1, X_2, \dots, X_n$ are independent random variables, then

$$\mathbf{E}[X_1 X_2 \cdots X_n] = \mathbf{E}[X_1] \cdot \mathbf{E}[X_2] \cdots \mathbf{E}[X_n].$$

18.2 Three popular random variables

Definition 18.23 (Bernoulli random variable)**.**
Let $0 < p < 1$ be some parameter. If X is a random variable with probability mass function $p_X(1) = p$ and $p_X(0) = 1 - p$, then we say that X has a *Bernoulli distribution with parameter* p (we also say that X is a Bernoulli random variable). We write $X \sim \text{Bernoulli}(p)$ to denote this. The parameter p is often called the *success* probability. ■

Remark. Note that $\mathbf{E}[X] = p$.

Definition 18.24 (Binomial random variable)**.**
Let $X = X_1 + X_2 + \ldots + X_n$, where the X_i's are independent and for all i, $X_i \sim$ Bernoulli(p). Then we say that X has a *binomial distribution with parameters n and p* (we also say that X is a binomial random variable). We write $X \sim \text{Bin}(n, p)$ to denote this. ■

Remark. Note that a Bernoulli random variable is a special kind of binomial random variable where $n = 1$.

Exercise 18.25. Let X be a random variable with $X \sim \text{Bin}(n, p)$. Determine $\mathbf{E}[X]$ (use linearity of expectation). Also determine X's probability mass function.

Exercise 18.26. We toss a coin 5 times. What is the probability that we see at least 4 heads?

Definition 18.27 (Geometric random variable)**.**
Let X be a random variable with probability mass function p_X such that for $n \in \{1, 2, \ldots\}$, $p_X(n) = (1-p)^{n-1}p$. Then we say that X has a *geometric distribution with parameter p* (we also say that X is a geometric random variable). We write $X \sim \text{Geometric}(p)$ to denote this. ■

Exercise 18.28. Let X be a geometric random variable. Verify that $\sum_{n=1}^{\infty} p_X(n) = 1$.

Exercise 18.29. Suppose we repeatedly flip a coin until we see a heads for the first time. Determine the probability that we flip the coin n times. Determine the expected number of coin flips (hint: use Proposition 18.18).

Exercise 18.30. Let X be a random variable with $X \sim \text{Geometric}(p)$. Determine $\mathbf{E}[X]$.

18.3 Some general tips

Remark. Here are some general tips on probability calculations (this is not meant to be an exhaustive list).

- If you are trying to upper bound $\mathbf{Pr}[A]$, you can try to find B with $A \subseteq B$, and then bound $\mathbf{Pr}[B]$. Note that if an event A implies an event B, then this means $A \subseteq B$. Similarly, if you are trying to lower bound $\mathbf{Pr}[A]$, you can try to find B with $B \subseteq A$, and then bound $\mathbf{Pr}[B]$.
- If you are trying to upper bound $\mathbf{Pr}[A]$, you can try to lower bound $\mathbf{Pr}[\bar{A}]$ since $\mathbf{Pr}[A] = 1 - \mathbf{Pr}[\bar{A}]$. Similarly, if you are trying to lower bound $\mathbf{Pr}[A]$, you can try to upper bound $\mathbf{Pr}[\bar{A}]$.
- In some situations, law of total probability can be very useful in calculating (or bounding) $\mathbf{Pr}[A]$.

- If you need to calculate $\mathbf{Pr}[A_1 \cap \ldots \cap A_n]$, try the chain rule. If the events are independent, then this probability is equal to the product $\mathbf{Pr}[A_1] \cdots \mathbf{Pr}[A_n]$. Note that the event "for all $i \in \{1, \ldots, n\}$, A_i" is the same as $A_1 \cap \ldots \cap A_n$.

- If you need to upper bound $\mathbf{Pr}[A_1 \cup \ldots \cup A_n]$, you can try to use the union bound. Note that the event "there exists an $i \in \{1, \ldots, n\}$ such that A_i" is the same as $A_1 \cup \ldots \cup A_n$.

- When trying to calculate $\mathbf{E}[X]$, try:

  (i) directly using the definition of expectation;

  (ii) writing X as a sum of indicator random variables, and then using linearity of expectation;

  (iii) using law of total expectation.

19 Randomized Algorithms

Remark. Informally, we'll say that an algorithm is *randomized* if it has access to a randomness source. In this course, we'll assume that a randomized algorithm is allowed to call RandInt(m), which returns a uniformly random element of $\{1, 2, \ldots, m\}$, and Bernoulli(p), which returns 1 with probability p and returns 0 with probability $1 - p$. We assume that both RandInt and Bernoulli take $O(1)$ time to execute. The notion of a randomized algorithm can be formally defined using *probabilistic Turing machines*, but we will not do so here.

Definition 19.1 (Monte Carlo algorithm)**.**
Let $f : \Sigma^* \to \Sigma^*$ be a computational problem. Let $0 \leq \epsilon < 1$ be some parameter and $T : \mathbb{N} \to \mathbb{N}$ be some function. Suppose A is a randomized algorithm such that

- for all $x \in \Sigma^*$, $\mathbf{Pr}[A(x) \neq f(x)] \leq \epsilon$, where the probability is over the random choices made by A;
- for all $x \in \Sigma^*$, the number of steps $A(x)$ takes is at most $T(|x|)$ no matter what the random choices of A are.

Then we say that A is a $T(n)$-time *Monte Carlo algorithm* that computes f with ϵ probability of error. ■

Definition 19.2 (Las Vegas algorithm)**.**
Let $f : \Sigma^* \to \Sigma^*$ be a computational problem. Let $T : \mathbb{N} \to \mathbb{N}$ be some function. Suppose A is a randomized algorithm such that

- for all $x \in \Sigma^*$, $\mathbf{Pr}[A(x) = f(x)] = 1$, where the probability is over the random choices made by A;
- for all $x \in \Sigma^*$, $\mathbf{E}[\text{number of steps } A(x) \text{ takes}] \leq T(|x|)$.

Then we say that A is a $T(n)$-time *Las Vegas algorithm* that computes f. ■

Remark. One can also define the notions of Monte Carlo algorithms and Las Vegas algorithms that compute optimization problems (Definition 15.1).

Exercise 19.3. Let P be some problem, and suppose you are given a Monte Carlo algorithm A that runs in worst-case $O(T_1(n))$ time and solves P with success probability at least p (i.e., for every input, the algorithm gives the correct answer with probability at least p and takes at most $O(T_1(n))$ steps.). Suppose it is possible to check in $O(T_2(n))$ time whether the output produced by A is correct or not. Show how to convert A into a Las Vegas algorithm that runs in expected time $O((T_1(n) + T_2(n))/p)$.

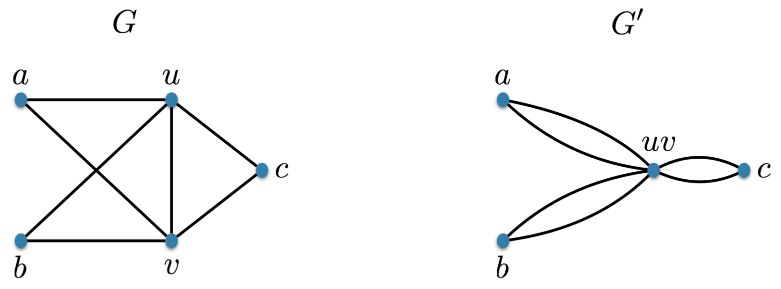
Definition 19.4 (Minimum cut problem)**.**
In the minimum cut problem, the input is an undirected graph G, and the output is a 2-coloring of the vertices such that the number of cut edges is minimized. (See Definition 15.6 for the definition of a cut edge.) Equivalently, we want to output a subset $S \subseteq V$ such that the number of edges between S and $V \backslash S$ is minimized. We denote this problem by MIN-CUT. ■

Definition 19.5 (Multi-graph)**.**
A *multi-graph* $G = (V, E)$ is an undirected graph in which E is allowed to be a multi-set. In other words, a multi-graph can have multiple edges between two vertices.[24] ■

Definition 19.6 (Contraction of two vertices in a graph)**.**
Let $G = (V, E)$ be a multi-graph and let $u, v \in V$ be two vertices in the graph. *Contraction* of u and v produces a new multi-graph $G' = (V', E')$. Informally, in G', we collapse/contract the vertices u and v into one vertex and preserve the edges between these two vertices and the other vertices in the graph. Formally, we remove the vertices u and v, and create a new vertex called uv, i.e. $V' = V \backslash \{u, v\} \cup \{uv\}$. The multi-set of edges E' is defined as follows:

- for each $\{u, w\} \in E$ with $w \neq v$, we add $\{uv, w\}$ to E';
- for each $\{v, w\} \in E$ with $w \neq u$, we add $\{uv, w\}$ to E';
- for each $\{w, w'\} \in E$ with $w, w' \notin \{u, v\}$, we add $\{w, w'\}$ to E'.

Below is an example:



■

Theorem 19.7 (Contraction algorithm)**.** *There is a polynomial-time Monte-Carlo algorithm that solves the* MIN-CUT *problem with error probability at most* $1/e^n$*, where* n *is the number of vertices in the input graph.*

Proof. The algorithm has two phases. The description of the first phase is as follows.

[24]Note that this definition does not allow for self-loops.

```
On input an undirected graph $G = (V, E)$:
  • Repeat until two vertices remain:
      – Select an edge $\{u, v\}$ uniformly at random.
      – Contract $u$ and $v$ to obtain a new graph.
  • Two vertices remain, which corresponds to a partition of $V$ into $V_1$ and $V_2$.
    Output $V_1$.
```

First it is easy to see that this algorithm runs in polynomial time. We leave the details to the reader. Our goal now is to show that the success probability of the first phase, i.e., the probability that the above algorithm outputs a minimum cut, is at least

$$\frac{2}{n(n-1)} \geq \frac{1}{n^2}.$$

In the second phase, we'll boost the success probability to the desired $1 - 1/e^n$. Let $F \subseteq E$ correspond to an optimum solution, i.e., a minimum size set of cut edges. We will show

$$\mathbf{Pr}[\text{algorithm finds } F] \geq \frac{2}{n(n-1)}.$$

Observe that if the algorithm picks an edge in F to contract, its output cannot correspond to F. If the algorithm never contracts an edge in F, then its output corresponds to F. In other words, the algorithm's output corresponds to F if and only if it never contracts an edge of F. Let E_i be the event that at iteration i of the algorithm, an edge in F is contracted. Note that there are $n-2$ iterations in total as we go from n vertices down to 2 vertices, and in each iteration, the number of vertices goes down by exactly 1. Therefore

$$\mathbf{Pr}[\text{algorithm finds } F] = \mathbf{Pr}[\bar{E}_1 \cap \bar{E}_2 \cap \ldots \cap \bar{E}_{n-2}].$$

Using the chain rule (Proposition 17.12), we have

$$\begin{aligned}&\mathbf{Pr}[\bar{E}_1 \cap \bar{E}_2 \cap \ldots \cap \bar{E}_{n-2}] = \\ &\qquad \mathbf{Pr}[\bar{E}_1] \cdot \mathbf{Pr}[\bar{E}_2 \mid \bar{E}_1] \cdot \mathbf{Pr}[\bar{E}_3 \mid \bar{E}_1 \cap \bar{E}_2] \cdots \mathbf{Pr}[\bar{E}_{n-2} \mid \bar{E}_1 \cap \bar{E}_2 \cap \ldots \cap \bar{E}_{n-3}]. \quad (1)\end{aligned}$$

To lower bound the success probability of the algorithm, we'll find a lower bound for each term of the RHS of the above equation. We start with $\mathbf{Pr}[\bar{E}_1]$. It is easy to see that $\mathbf{Pr}[E_1] = |F|/m$. However, it will be more convenient to have a bound on $\mathbf{Pr}[E_1]$ in terms of $|F|$ and n rather than m. Observe that

$$\forall v \in V, \quad |F| \leq \deg(v).$$

If this was not true, i.e. there was some v with $|F| > \deg(v)$, then there would be a cut, namely $S = \{v\}$, with smaller number of cut edges than $|F|$. And this would contradict the minimality of F. Using this observation, we have

$$2m = \sum_{v \in V} \deg(v) \geq |F| \cdot n, \tag{2}$$

or equivalently, $|F| \leq 2m/n$. Therefore,

$$\mathbf{Pr}[E_1] = \frac{|F|}{m} \leq \frac{2}{n},$$

or equivalently, $\mathbf{Pr}[\bar{E}_1] \geq 1 - 2/n$. At this point, going back to Equality (1) above, we can write

$$\begin{aligned}&\mathbf{Pr}[\text{algorithm finds } F] \geq \\ &\qquad \left(1 - \frac{2}{n}\right) \cdot \mathbf{Pr}[\bar{E}_2 \mid \bar{E}_1] \cdot \mathbf{Pr}[\bar{E}_3 \mid \bar{E}_1 \cap \bar{E}_2] \cdots \mathbf{Pr}[\bar{E}_{n-2} \mid \bar{E}_1 \cap \bar{E}_2 \cap \ldots \cap \bar{E}_{n-3}].\end{aligned}$$

We move onto the second term $\mathbf{Pr}[\bar{E}_2 \mid \bar{E}_1]$. Let ℓ_1 be the number of edges remaining after the first iteration of the algorithm. Then

$$\mathbf{Pr}[\bar{E}_2 \mid \bar{E}_1] = 1 - \mathbf{Pr}[E_2 \mid \bar{E}_1] = 1 - \frac{|F|}{\ell_1}.$$

As before, we can argue that, at any iteration in the algorithm, for any $v \in V$, $|F| \leq \deg(v)$. If not, we can find a min-cut with less than $|F|$ edges, which is not possible. Therefore, similar to Inequality (2) above, we have $2\ell_1 \geq |F|(n-1)$. Using this inequality,

$$\mathbf{Pr}[\bar{E}_2 \mid \bar{E}_1] = 1 - \frac{|F|}{\ell_1} \geq 1 - \frac{2|F|}{|F|(n-1)} = 1 - \frac{2}{n-1}.$$

Thus

$$\begin{aligned}&\mathbf{Pr}[\text{algorithm finds } F] \geq \\ &\qquad \left(1 - \frac{2}{n}\right) \cdot \left(1 - \frac{2}{n-1}\right) \cdot \mathbf{Pr}[\bar{E}_3 \mid \bar{E}_1 \cap \bar{E}_2] \cdots \mathbf{Pr}[\bar{E}_{n-2} \mid \bar{E}_1 \cap \bar{E}_2 \cap \ldots \cap \bar{E}_{n-3}].\end{aligned}$$

The same argument allows us to give a lower bound for each term in the product above, so

$$\begin{aligned}&\mathbf{Pr}[\text{algorithm finds } F] \geq \\ &\qquad \left(1 - \frac{2}{n}\right) \cdot \left(1 - \frac{2}{n-1}\right) \cdot \left(1 - \frac{2}{n-2}\right) \cdots \left(1 - \frac{2}{n-(n-3)}\right) = \\ &\qquad \left(\frac{n-2}{n}\right) \cdot \left(\frac{n-3}{n-1}\right) \cdot \left(\frac{n-4}{n-2}\right) \cdots \left(\frac{1}{3}\right).\end{aligned}$$

After cancellations between the numerators and denominators of the fractions, the first two denominators and the last two numerators survive, and the above simplifies to $2/n(n-1)$. So we have reached our goal for the first phase and have shown that

$$\mathbf{Pr}[\text{algorithm finds } F] \geq \frac{2}{n(n-1)} \geq \frac{1}{n^2}.$$

This implies

$$\mathbf{Pr}[\text{algorithm finds a min-cut}] \geq \frac{1}{n^2}.$$

In the second phase of the algorithm, we boost the success probability by repeating the first phase t times using completely new and independent random choices. The output of the second phase is a subset $S \subseteq V$ that gives rise to the least number of cut edges. Our analysis will show that $t = n^3$ is a good choice.

Let A_i be the event that our algorithm does *not* find a min-cut at repetition i. Note that the A_i's are independent since our algorithm uses fresh random bits for each repetition. Also, each A_i has the same probability, i.e. $\mathbf{Pr}[A_i] = \mathbf{Pr}[A_j]$ for all i and j. Therefore

$$\begin{aligned}\mathbf{Pr}[\text{our algorithm fails to find a min-cut}] &= \mathbf{Pr}[A_1 \cap \ldots \cap A_t] \\ &= \mathbf{Pr}[A_1] \cdots \mathbf{Pr}[A_t] \\ &= \mathbf{Pr}[A_1]^t.\end{aligned}$$

From the analysis of the first phase, we know that

$$\mathbf{Pr}[A_1] \leq 1 - \frac{1}{n^2}.$$

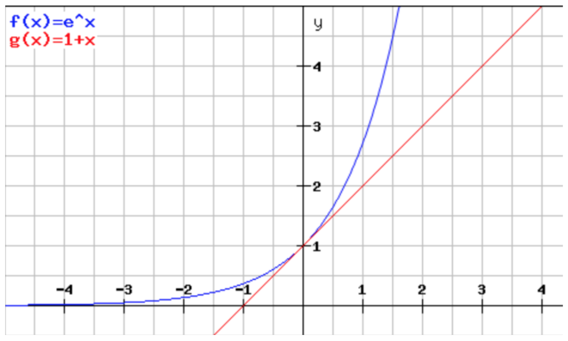
So

$$\mathbf{Pr}[\text{our algorithm fails to find a min-cut}] \leq \left(1 - \frac{1}{n^2}\right)^t.$$

To upper bound this, we'll use an extremely useful inequality:

$$\forall x \in \mathbb{R}, \quad 1 + x \leq e^x.$$

We will not prove this inequality, but we provide a plot of the two functions below.



Notice that the inequality is close to being tight for values of x close to 0. Letting $x = -1/n^2$, we see that

$$\mathbf{Pr}[\text{our algorithm fails to find a min-cut}] \leq (1+x)^t \leq e^{xt} = e^{-t/n^2}.$$

For $t = n^3$, this probability is upper bounded by $1/e^n$, as desired. □

Exercise 19.8. Using the analysis of the randomized minimum cut algorithm seen in class, show that a graph can have at most $n(n-1)/2$ distinct minimum cuts.

Exercise 19.9. Suppose we modify the min-cut algorithm seen in class so that rather than picking an edge uniformly at random, we pick 2 vertices uniformly at random and contract them into a single vertex. True or False: The success probability of the algorithm (excluding the part that boosts the success probability) is $1/n^k$ for some constant k, where n is the number of vertices. Justify your answer.

Exercise 19.10. Let P be a decision problem, and let A be a Monte Carlo algorithm for A such that if x is a YES instance, then A always gives the correct answer, and if x is a NO instance, then A gives the correct answer with probability at least $1/2$. Suppose A runs in worst-case $O(T(n))$ time. Design a new Monte Carlo algorithm A' for P that runs in $O(nT(n))$ time and has error probability at most $1/2^n$.

20 Modular Arithmetic

Definition 20.1 (A divides B)**.**
Let $A, B \in \mathbb{Z}$. We say that A *divides* B (or A is a *divisor* of B), denoted $A|B$, if there is a number $C \in \mathbb{Z}$ such that $B = AC$. ■

Definition 20.2 (Prime number)**.**
Let $P \in \mathbb{N}$. We say that P is a *prime number* if $P \geq 2$ and the only divisors of P are 1 and P. ■

Definition 20.3 (Congruence modulo N)**.**
We denote by $A \bmod N$ the remainder you get when you divide A by N. Note that $A \bmod N \in \{0, 1, 2, \cdots, N-1\}$. We say that A and B are congruent modulo N, denoted $A \equiv_N B$ (or $A \equiv B \bmod N$), if $A \bmod N = B \bmod N$. ■

Exercise 20.4. Show that $A \equiv_N B$ if and only if $N|(A - B)$.

Notation 20.5. We write $\gcd(A, B)$ to denote the greatest common divisor of A and B. Note that for any A, $\gcd(A, 1) = 1$ and $\gcd(A, 0) = A$.

Definition 20.6 (Relatively prime)**.**
We say that A and B are relatively prime if $\gcd(A, B) = 1$. ■

20.1 Basic modular operations

20.1.1 Addition and subtraction

Notation 20.7. We let $\mathbb{Z}_N$ denote the set $\{0, 1, 2, \ldots, N-1\}$.

Definition 20.8 (Addition in $\mathbb{Z}_N$)**.**
For $A, B \in \mathbb{Z}_N$, we define the *addition* of A and B, denoted $A +_N B$, as $(A + B) \bmod N$. (When N is clear from the context, we drop the subscript N from $+_N$ and write $+$.) For example, for $N = 5$, we can represent the addition operation in $\mathbb{Z}_5$ using the following table.

+	0	1	2	3	4
0	0	1	2	3	4
1	1	2	3	4	0
2	2	3	4	0	1
3	3	4	0	1	2
4	4	0	1	2	3

In $\mathbb{Z}_N$, the element 0 is called the *additive identity*. It has the property that for any $A \in \mathbb{Z}_N$, $A + 0 = 0 + A = A$. ■

Exercise 20.9. Show that if $A \equiv_N B$ and $A' \equiv_N B'$, then $A + A' \equiv_N B + B'$.

Exercise 20.10. Show that for any $A, B \in \mathbb{Z}$,

$$(A + B) \bmod N = (A \bmod N) +_N (B \bmod N).$$

Definition 20.11 (Additive inverse)**.**
Let $A \in \mathbb{Z}_N$. The *additive inverse* of A, denoted $-A$, is defined to be an element in $\mathbb{Z}_N$ such that $A + (-A) = 0$. ■

Exercise 20.12. Show that every element of $\mathbb{Z}_N$ has a unique additive inverse.

Definition 20.13 (Subtraction in $\mathbb{Z}_N$)**.**
For $A, B \in \mathbb{Z}_N$, we define the *subtraction* of B from A, denoted $A - B$, as $A + (-B)$. ■

Exercise 20.14. Show that in the addition table of $\mathbb{Z}_N$, every row and column is a permutation of the elements $\mathbb{Z}_N$.

20.1.2 Multiplication and division

Definition 20.15 (Multiplication in $\mathbb{Z}_N$)**.**
For $A, B \in \mathbb{Z}_N$, we define the *multiplication* of A and B, denoted $A \cdot_N B$, as $AB \bmod N$. (When N is clear from the context, we drop the subscript N from $\cdot_N$ and write $\cdot$.) ■

Exercise 20.16. Show that if $A \equiv_N B$ and $A' \equiv_N B'$, then $AA' \equiv_N BB'$.

Exercise 20.17. Show that for any $A, B \in \mathbb{Z}$,

$$AB \bmod N = (A \bmod N) \cdot_N (B \bmod N).$$

Definition 20.18 (Multiplicative inverse)**.**
Let $A \in \mathbb{Z}_N$. The *multiplicative inverse* of A, denoted A^{-1}, is defined to be an element in $\mathbb{Z}_N$ such that $A \cdot A^{-1} = 1$. ■

Proposition 20.19. *Let $A, N \in \mathbb{N}$. The multiplicative inverse of A in $\mathbb{Z}_N$ exists if and only if* $\gcd(A, N) = 1$.

Definition 20.20 (Division in $\mathbb{Z}_N$)**.**
Let $A, B \in \mathbb{Z}_N$, where B has a multiplicative inverse B^{-1}. Then we define the *division* of A by B, denoted A/B, as $A \cdot B^{-1}$. ■

Notation 20.21. We let $\mathbb{Z}_N^*$ denote the set $\{A \in \mathbb{Z}_N : \gcd(A, N) = 1\}$. In other words, $\mathbb{Z}_N^*$ is the set of all elements of $\mathbb{Z}_N$ that have a multiplicative inverse.

Exercise 20.22. Show that $\mathbb{Z}_N^*$ is *closed* under multiplication, i.e., $A, B \in \mathbb{Z}_N^* \implies A \cdot B \in \mathbb{Z}_N^*$.

Remark. Similar to an addition table for $\mathbb{Z}_N$, one can consider a multiplication table for $\mathbb{Z}_N^*$. For example, $\mathbb{Z}_8^* = \{1, 3, 5, 7\}$, and the multiplication table is as below:

•	1	3	5	7
1	1	3	5	7
3	3	1	7	5
5	5	7	1	3
7	7	5	3	1

Exercise 20.23. Show that in the multiplication table of $\mathbb{Z}_N^*$, every row and column is a permutation of the elements $\mathbb{Z}_N^*$.

Definition 20.24 (Euler totient function)**.**
The Euler totient function $\varphi : \mathbb{N} \to \mathbb{N}$ is defined as $\varphi(N) = |\mathbb{Z}_N^*|$. By convention, $\varphi(0) = 0$. ■

Exercise 20.25. Show that for P a prime number, $\varphi(P) = P - 1$. Also show that for P and Q distinct prime numbers, $\varphi(PQ) = (P-1)(Q-1)$.

20.1.3 Exponentiation

Definition 20.26 (Exponentiation in $\mathbb{Z}_N$)**.**
Let $A \in \mathbb{Z}_N$ and $E \in \mathbb{Z}$. We write A^E to denote

$$\underbrace{A \cdot A \cdot \cdots \cdot A}_{E \text{ times}}.$$

■

Theorem 20.27 (Euler's Theorem)**.** *For any $A \in \mathbb{Z}_N^*$, $A^{\varphi(N)} = 1$. Equivalently, for any $A, N \in \mathbb{Z}$ with* $\gcd(A, N) = 1$, $A^{\varphi(N)} \equiv_N 1$.

Proof. Take an arbitrary $A \in \mathbb{Z}_N^*$. Let $B_1, B_2, \ldots, B_k$ be the elements of $\mathbb{Z}_N^*$, where $k = \varphi(N)$. By Exercise 20.23, $\{AB_1, AB_2, \ldots, AB_k\} = \mathbb{Z}_N^*$. The product of all the elements in the first set can be written as $(AB_1)(AB_2)\cdots(AB_k)$. This must be equal to the product $B_1 B_2 \cdots B_k$, i.e.

$$(AB_1)(AB_2)\cdots(AB_k) = B_1 B_2 \cdots B_k.$$

Dividing both sides by $B_1 B_2 \cdots B_k$ (i.e. multiplying both sides by the inverse of $B_1 B_2 \cdots B_k$), we get $A^k = 1$, as desired. □

Remark. When N is a prime number, then Euler's Theorem is known as Fermat's Little Theorem.

Exercise 20.28. Let $A \in \mathbb{Z}_N^*$ and $E \in \mathbb{Z}$. Show that $A^E \equiv_N A^{E \bmod \varphi(N)}$. Show that this is not always true if $A \in \mathbb{Z}_N \setminus \mathbb{Z}_N^*$.

Remark. What the previous exercise implies is that if we are exponentiating an element $A \in \mathbb{Z}_N^*$, then we can effectively think of the exponent as living in the set $\mathbb{Z}_{\varphi(N)}$.

Definition 20.29 (Generator)**.**
Let $A \in \mathbb{Z}_N^*$. We say that A is a *generator* if

$$\{A^E : E \in \mathbb{Z}_{\varphi(N)}\} = \mathbb{Z}_N^*.$$

■

Theorem 20.30. *If N is a prime number, then $\mathbb{Z}_N^*$ contains a generator.*

20.2 Computational complexity of basic modular operations

In this section, we will look at the computational complexities of doing the basic modular operations discussed in the previous section. We will use the fact that addition, subtraction, multiplication and division operations can be computed efficiently in $\mathbb{Z}$. Note that $A \bmod N$ is easy to compute by dividing A by N and seeing what the remainder is.

20.2.1 Addition and subtraction

In order to compute $A + B$ in $\mathbb{Z}_N$, we can simply add A and B in $\mathbb{Z}$ and then take the sum modulo N. To compute $A - B$, we can do $A + (N - B)$ in $\mathbb{Z}$ and then take the result modulo N.

20.2.2 Multiplication and division

In order to compute $A \cdot B$ in $\mathbb{Z}_N$, we can multiply A and B in $\mathbb{Z}$ and then take the product modulo N. To compute $A/B = AB^{-1}$, we first need to figure out whether B has a multiplicative inverse. Recall that B^{-1} exists if and only if B and N are relatively prime, i.e. $\gcd(B, N) = 1$. The following algorithm, known as *Euclid's Algorithm*, efficiently computes the greatest common divisor of two numbers.

```
gcd($A, B$):
    • If $B = 0$, then return $A$.
    • Else return gcd($B, A \bmod B$).
```

Exercise 20.31. Show that if $A \geq B$, $\gcd(A, B) = \gcd(A-B, B)$. Use this to show that Euclid's Algorithm correctly computes the greatest common divisor of two numbers.

Exercise 20.32. Suppose A and B can be represented with at most n bits each. Give an upper bound on the number of recursive calls Euclid's Algorithm makes in terms of n.

Using Euclid's Algorithm, we can check if $\gcd(B, N) = 1$ and determine if B has a multiplicative inverse. It turns out that a slight modification of Euclid's Algorithm also allows us to compute B^{-1} if it exists. In order to show this, we first need a definition.

Definition 20.33 (Miix)**.**
Let $A, B, C \in \mathbb{N}$. We say that C is a *miix* of A and B if

$$C = kA + \ell B$$

for some $k, \ell \in \mathbb{Z}$. ■

Exercise 20.34. Let $A, B, C \in \mathbb{N}$. Show that if C is a miix of A and B then C is a multiple of $\gcd(A, B)$.

Exercise 20.35. Let $A, B, C \in \mathbb{N}$. Show that if C is any multiple of $\gcd(A, B)$, then C is a miix of A and B.
Hint: Show how to modify Euclid's algorithm so that it outputs k and ℓ such that $\gcd(A, B) = kA + \ell B$.

Suppose B has a multiplicative inverse modulo N, i.e. $\gcd(B, N) = 1$. Then by the previous exercise, we can obtain k and ℓ such that $1 = kB + \ell N$. If we take this equation modulo N, we get that $kB \equiv_N 1$. Therefore k is the multiplicative inverse of B.

To sum up, if we want to compute $A/B = A \cdot B^{-1}$, we can first compute B^{-1} and then compute $A \cdot B^{-1}$.

Exercise 20.36. Prove Proposition 20.19 using the previous two exercises.

20.2.3 Exponentiation

Given $A \in \mathbb{Z}_N$ and $E \in \mathbb{Z}$, we can compute $A^E \bmod N$ efficiently. Assume that A, E and N can be represented using at most n bits each. The algorithm below is known as *fast modular exponentiation.* To understand how the algorithm works, see the example following the algorithm.

FME(A, E, N):

- Repeatedly square A to obtain
  $A^2 \bmod N$, $A^4 \bmod N$, $A^8 \bmod N$, ..., $A^{2^n} \bmod N$.
- Multiply together (modulo N) the powers of A so that the product is A^E.
  To figure out which powers to multiply, look at the binary representation of E.

Consider the example of computing $2337^{53} \bmod 100$. The first step of the algorithm computes

$$\begin{aligned} 2337^2 &\bmod 100 \\ 2337^4 &\bmod 100 \\ 2337^8 &\bmod 100 \\ 2337^{16} &\bmod 100 \\ 2337^{32} &\bmod 100 \end{aligned}$$

by squaring 2337 modulo 100 5 times. The binary representation of 53 is 110101. This implies that

$$53 = 1 + 4 + 16 + 32.$$

Therefore to calculate $2337^{53} \bmod 100$, the second step of the algorithm does:

$$(2337 \bmod 100) \cdot (2337^4 \bmod 100) \cdot (2337^{16} \bmod 100) \cdot (2337^{32} \bmod 100).$$

Exercise 20.37. Suppose A, E and N are integers that can be represented using at most n bits. Give an upper bound on the running time of the above algorithm in terms of n.

21 Cryptography

There are no technical definitions or proofs for this section. Make sure you know how the following work:

- One-time pad secret-key cryptographic system,
- Diffie-Hellman secret-key exchange protocol,
- RSA public-key cryptographic system.