Lectures

William Bernardoni

January 11, 2022

Contents

1 Induction and Well Ordering

As a general note, we'll be working with $\mathbb{N}$, natural numbers, for this lecture. Some people take 0 as a natural number (most of the time I do too), but for the purpose of this lecture 1 is the smallest natural number.

Everything we go over works whether or not you take 0 as the smallest number or 1, if you take 0 to be then in all of the places where we reference 1 as the "first step" or smallest number, just take 0.

1.1 What is Induction?

How would we prove a statement like: $\forall n \in \mathbb{N}$, $3^n - 1$ *is divisible by 2?*

We could try plugging in numbers – $n = 3$ gives: 26 is divisible by 2, which is a true statement. But this is not a very time efficient way to prove this statement – there's an infinite number of n's to try and we unfortunately have a finite amount of time.

We could try doing a straight proof – for $3^n - 1$ to be divisible by 2 then we would need to find some $k \in \mathbb{N}$ such that $2 * k = 3^n - 1$, our knowledge of arithmetic gives us $k = \frac{3^n - 1}{2}$, and if we can show that is a natural number we are done. However we have just entered into a loop, because to show that $\frac{3^n - 1}{2}$ is a natural number we would need to prove $3^n - 1$ is divisible by 2. [1]

To prove one of these "infinite" statements we use **induction**. Intuitively, induction is like climbing a staircase. If we want to climb a staircase then we need to know: **(1)** How to go from a step to the next, and **(2)** How to get *on* the first step.

The idea being that if I want to show that I can get to the *fifth* step on a staircase, then **(1)** gives us that it suffices to get to the *fourth* step and then we know how to step up by one, **(1)** again gives that for us to get to the *fourth* step we need to know how to get to the *third*, then by **(1)** the second, and by **(1)** the first – and by **(2)** we know how to get on the first step, so we have generated a proof that we can get to the fifth step.

Induction is in a way a program to generate proofs. It is a recursive proof. If $P(n-1)$ implies $P(n)$ then we get:

$$P(n) \impliedby P(n-1) \impliedby P(n-2) \impliedby ... \impliedby P(3) \impliedby P(2) \impliedby P(1)$$

So to show $P(n)$ is true, once we know $P(n-1)$ implies $P(n)$ we just need to show that the statement holds for 1.

1.2 Induction Formally

To put this in the language of formal logic. Let us have a map P that takes as input a natural number, and has as an output a formal statement. So for each

[1] There is a straightforward proof by contradiction of this particular theorem – let $3^n - 1$ not be divisible by 2, then it is odd and so 3^n is even. However $2 \not| 3^n$ as we know the prime factorization of 3^n, and so we have a contradiction.

$n \in \mathbb{N}$, $P(n)$ is a statment (i.e. $3^n - 1$ is divisible by 2).

If $P(n-1) \implies P(n)$ and $P(1)$ is true, then $P(n)$ is true for all $n \in \mathbb{N}$.[2]

To prove this we need the Well Ordering Principle, which is discussed in section 1.7. The Well Ordering Principle states that for any subset X of the natural numbers, there is a unique element $x \in X$ such that x is the smallest element. Induction is equivalent to the Well Ordering Principle, and either Induction or the Well Ordering Principle are taken as an axiom of the natural numbers.

Well Ordering $\implies$ *Induction.* We prove using the contrapositive. Assume there is some n such that $P(n)$ is not true, then either $P(n-1) \not\implies P(n)$ or $P(1)$ is not true.

Let X be the set of $n \in \mathbb{N}$ such that $P(n)$ is false. By the well ordering principle there exists a unique smallest element x in X. If $x = 1$ then we are done, as then $P(1)$ is not true.

If $x \neq 1$ then $x - 1$ is a natural number, and $x - 1$ is not in X, as then x would not be the smallest element in X. Thus $P(x-1)$ is true and $P(x)$ is false, so $P(x-1) \not\implies P(x)$.

Thus the contrapositive holds, and so our statement holds. □

We now see that induction not only *intuitively* works but it also *formally* works.

1.3 Proof Template for Induction

Proofs by induction tend to follow a pretty specific outline: To prove the statement

For all $n \in \mathbb{N}, P(n)$ holds

1. **Establish the Base Case:**
   Show that $P(1)$ holds

2. **Complete the Inductive Step:**
   Assuming that $P(n)$ holds prove $P(n+1)$

The assumption that $P(n)$ holds is called the **induction hypothesis**.
Let's use this template to prove that 2 divides $3^n - 1$:
In this case $P(n) =$"2 divides $3^n - 1$"

1. **Establish the Base Case:**
   Show that $P(1) =$"2 divides $3^1 - 1 = 2$ holds.

   We note that $2 * 1 = 2$, and so 2 divides 2. Thus we have established our base case.

[2] As n is arbitrary, note that we could also write $P(n) \implies P(n+1)$

2. **Complete the Inductive Step:**
   Assuming that $P(n)$ holds prove $P(n+1)$

   We assume here that 2 divides $3^n - 1$ and want to show that 2 divides $3^{n+1} - 1$.

   Note $3^{n+1} - 1 = 3 * 3^n - 1 = 3(3^n - 1) + 2$.

   We know from our *induction hypothesis* that 2 divides $3^n - 1$, so there is some $k \in \mathbb{N}$ such that $2 * k = 3^n - 1$. So we can write:

$$3^{n+1} - 1 = 3(3^n - 1) + 2 = 3(2k) + 2$$

   Which can again be rewritten:

$$3(2k) + 2 = 2(3k + 1)$$

   So $3^{n+1} - 1 = 2(3k + 1)$. As $3k + 1 \in \mathbb{N}$ we get that 2 divides $3^{n+1} - 1$.

   So if 2 divides $3^n - 1$ then 2 divides $3^{n+1} - 1$

With that we've finished our proof. By induction we get that for all $n \in \mathbb{N}$, 2 divides $3^n - 1$.

1.4 Strong Induction

Take a look back at the "generated proof" from our induction:

$$P(n) \impliedby P(n-1) \impliedby P(n-2) \impliedby ... \impliedby P(3) \impliedby P(2) \impliedby P(1)$$

The reason this "generated proof" works is that we have a *finite length*[3] path from something true: $P(1)$ to what we want to be true: $P(n)$

We notice though that before we prove $P(n-1)$ is true in this generated proof, we have proven $P(n-2)$ and $P(n-3)$, and in fact every k between $n-1$ and 1. So by the time we are taking the step to go from $P(n-1)$ to $P(n)$ we know that $P(k)$ for $1 \leq k < n$ is true.

However take another look at section 1.3, in particular focusing on our induction step:

$$\text{Assuming that } P(n-1) \text{ holds prove } P(n)$$

That's a far *weaker* assumption than we are allowed to make. The process of induction doesn't just give us $P(n-1)$ to work with by the time that we are proving $P(n+1)$, it gives us everything less than n to work with.

We can change the induction hypothesis to use this *stronger* fact, and create something called **strong induction**. For strong induction the base case is the same – you prove that $P(1)$ holds. But for the induction step, you assume that $P(k)$ holds for $1 \leq k < n$ and show that implies $P(n)$ holds.

As a proof template: To prove the statement

$$\text{For all } n \in \mathbb{N}, P(n) \text{ holds}$$

[3]There are systems of logic where the *finite length* requirement is not needed. Those types of logic are called Infinitary Logic

1. **Establish the Base Case:**
   Show that $P(1)$ holds

2. **Complete the Inductive Step:**
   Assuming that $P(k)$ holds for $1 \leq k < n$ prove $P(n)$

Strong induction is logically equivalent to "regular" induction, but can make proofs somewhat simpler.

Let's do an example of a proof that is easier using strong induction.

For all $1 < n \in \mathbb{N}$: n can be written as the product of prime numbers.

1. **Establish the Base Case:**
   For this theorem our base case is *not* that 1 can be written as the product of prime numbers, as we are looking at $n > 1$. Instead we want to consider the smallest number that is not 1 in the natural numbers – 2.

   2 is prime itself, so writing 2 as 2 is writing it as the product of prime numbers. Thus our base case holds.

2. **Complete the Inductive Step:**
   We now assume that for all $2 \leq k < n$, k can be written as the product of prime numbers.

   n either has a divisor less than it that is not 1, or it doesn't. So we can break into cases:

   (a) n **has no divisors less than it that aren't** 1**.** So for all $k < n$, k does not divide n. This tells us that n is prime, as if n was not prime then $n = a * b$ where $n > a, b > 1$.

   Thus n is prime, and so writing n by itself is writing it as the product of prime numbers.

   (b) n **has some divisor less than** n **that is not** 1. Let that divisor be b, so we can write $n = ab$ for some $a \in \mathbb{N}$ with $2 \leq b < n$. We note then that $2 \leq a < n$, as $k \neq 1$ so $a \neq n$, and $b < n$ so $a > 1$.

   Thus by our induction hypothesis as $2 \leq a, b < n$ both a and b can be written as the product of prime numbers. n can then be written as the product of those two sequences of primes.

   Thus n can be written as the product of prime numbers.

And thus we have completed our induction.

There are two important things to note here:

1. Our base case isn't always $n = 1$. In the previous theorem we weren't looking at the number 1, so it didn't make sense to start with 1.

2. We sometimes don't even *use* $n-1$ to prove n. For our above proof, if you consider $n = 12$, what we did was note that 3 divides 12 and so 12 can be written as $3 * 4$. We then noted that 3 and 4 are both less than 12, and so

they each have their own prime decomposition (3 and $2 * 2$ respectively), and so to build the decomposition of 12 we just multiply those together: $3 * 2 * 2$.

Nowhere in the proof did we use that 11 had a prime decomposition.

Strong induction lets us "jump" around, which can be extremely useful.

1.5 Some Example Inductions

For all n, $\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$:[4]

Proof.
Base Case: $\sum_{i=0}^{0} i = 0 = \frac{0(1)}{2}$.

Inductive Step: Let $\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$

$$\sum_{i=0}^{n+1} i = n + 1 + \sum_{i=0}^{n} \tag{1}$$

$$= n + 1 + \frac{n(n+1)}{2} \tag{2}$$

$$= \frac{2n + 2 + n^2 + n}{2} \tag{3}$$

$$= \frac{n^2 + 3n + 2}{2} \tag{4}$$

$$= \frac{(n+1)(n+2)}{2} \tag{5}$$

$$- \frac{(n+1)((n+1)+1)}{2} \tag{6}$$

So by induction $\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$ □

If we have as many 4 oz and 5 oz weights as we want, we can create a weight of k oz for any $k \geq 12$ (with k a natural number)

Proof.
Base Case: $k = 12$ – $12 = 4 + 4 + 4$.[5]

Induction Step: Assume we can form k oz using 4 and 5 oz weights.
We break this into cases

1. If we can form k oz using at least one 4 oz weight then we can simply replace the 4 oz weight with a 5 oz weight, and we have a weight of $k + 1$ oz.

[4] This identity is commonly attributed to Gauss. The story goes that he annoyed his teacher and was told to go to a corner and add all of the numbers from 1 to 100 before he could talk again, after a minute he returned to his chair with the answer. When his teacher asked him how he added them so fast he said that to find the sum he just had to do $\frac{100*101}{2}$

[5] Note here that our base case was not $k = 0$ but $k = 12$

2. If k is formed using only 5 oz weights, then we can see that $k = 15 + 5n$ for some $0 \leq n$ (n a whole number). $k + 1 = 16 + 5n = 4(3) + 5(n)$, so we can form $k + 1$ oz using three four oz weights and a number of 5 oz weights.

□

For $n \geq 1$:

$$\sum_{i=1}^{n} \frac{1}{i(i+1)} = \frac{n}{n+1}$$

Proof.
Base Case: $n = 1$, $\frac{1}{1(2)} = \frac{1}{2}$
Inductive Step: Let it hold for n.

$$\sum_{i=1}^{n+1} \frac{1}{i(i+1)} = \frac{1}{(n+1)(n+2)} + \sum_{i=1}^{n} \frac{1}{i(i+1)} \tag{7}$$
$$= \frac{1}{(n+1)(n+2)} + \frac{n}{n+1} \tag{8}$$
$$= \frac{1 + n(n+2)}{(n+1)(n+2)} \tag{9}$$
$$= \frac{n^2 + 2n + 1}{(n+1)(n+2)} \tag{10}$$
$$= \frac{(n+1)^2}{(n+1)(n+2)} \tag{11}$$
$$= \frac{n+1}{n+2} \tag{12}$$

□

$n! > 2^n$ **for** $n \geq 4$

Proof.
Base Case: $n = 4$, $4! = 24$, $2^4 = 16$.
Inductive Step Let it be true for n.

$$(n+1)! = n!(n+1) \tag{13}$$
$$> 2^n(n+1) \tag{14}$$
$$> 2^n * 2 \tag{15}$$
$$= 2^{n+1} \tag{16}$$

With step 15 holding as $n + 1 \geq 4 > 2$ □

Let the "Tribbonaci" sequence T_n, be $T_1 = T_2 = T_3 = 1$ and $T_n = T_{n-1} + T_{n-2} + T_{n-3}$ for $n \geq 4$. Show that $T_n < 2^n$ for all $n \in \mathbb{N}$

Proof.
Base Case: We can see that $T_1 = 1 < 2$, $T_2 = 1 < 4$, $T_3 = 1 < 8$.

Inductive Step: Let n be given, and let $T_k < 2^k$ for all $k < n$.

$$\begin{aligned} T_n &= T_{n-1} + T_{n-2} + T_{n-3} && (17)\\ &< 2^{n-1} + 2^{n-2} + 2^{n-3} && (18)\\ &= 2^n(\frac{1}{2} + \frac{1}{4} + \frac{1}{8}) && (19)\\ &= 2^n * \frac{7}{8} && (20)\\ &< 2^n && (21) \end{aligned}$$

With step 18 coming from our induction hypothesis. □

Take a moment to examine the above proof, in particular why we needed three base cases[6] and why we needed strong induction[7].

1.6 A Common Inductive Danger

Consider the following proof:

Any group of horses will be the same color.

Proof.
Base Case: If we have a group of one horse, the whole group is the color of that one horse, and so is the same color.

Inductive Step: Let all groups of n horses be the same color. Consider a group of $n + 1$ horses.

Pick a horse in this group and move it away from the group, the remaining n horses are then the same color by our inductive hypothesis. Reintroduce that horse to the group and move a different horse away, the remaining n horses are the same color.

Note that the "middle" horses who were not taken away are the same color in both groups, and so our two taken away horses are the same color.

Thus all horses are the same color. □

All horses are not in fact the same color in reality, so have we found some fatal flaw in math? Have we broken the system?

Nope!

Take a moment to try and find the flaw in the proof.

None of the steps taken are incorrect, but the proof is incorrect. The base case is true – all groups of 1 horse are the same color, and the inductive step is true – the "middle" horses force the whole group to have the same color.

[6] For $n = 1, 2, 3$ we do not have that $T_n = T_{n-1} + T_{n-2} + T_{n-3}$, so our inductive step proof does not apply.

[7] We used that $T_{n-1} < 2^{n-1}$, $T_{n-2} < 2^{n-2}$, and $T_{n-3} < 2^{n-3}$ not just $T_{n-1} < 2^{n-1}$ – which is why we needed strong induction and not regular induction.

But consider $n = 2$, *there are no "middle" horses.* Both of our groups of 1 horses are completely distinct, so our inductive step doesn't prove $P(2)$. To go back to our staircase metaphor, we can't skip a stair in the staircase if we are only taking one step up at a time.

If this held for 2 horses then the proof would work for every number of horse, and so every group of horses would be the same color.

You need to make sure your inductive step works for *every* case not in the base case.

1.7 The Well Ordering Principle and Infinite Descent

There is a dual notion and proof technique to induction called **Infinite Descent**. Infinite Descent uses a key feature of the natural numbers called the *Well Ordering Principle.*

The **Well Ordering Principle** is the following statement: For any nonempty subset of the natural numbers there is a *least* element.

Well Ordering is logically equivalent to induction, and can be proven by using the successor function:

Proof of the Well Ordering Principle. Let s be the successor function (i.e. $s(0) = 1$, $s(n) = n + 1$)

Let X be a nonempty subset of $\mathbb{N}$, by definition for each $x \in X$ either $x = s(x - 1)$ or $x = 0$. As X is nonempty there is some $n \in X$. By definition $n = s^n(0)$ for some $n \in \mathbb{N}$. Consider the set $N = \{0, 1, 2, ..., n\}$. $N \cap X \neq \emptyset$ as $n \in N$ and $n \in X$, and as N is a finite set, $X \cap N$ is a finite set.

$X \cap N$ then has a least element (as each finite set must have a least and a maximal element). We note that if $x \in X \cap N$ is the least element of $X \cap N$ then $x \leq k$ for all $k \leq n$ and $k \in X$. We note that if $k \in X$ and $k \notin X \cap N$ then $k > n$, so $x \leq n < k$. So $x \leq k$ for all $k \in X$.

Thus X has a least element. □

As $\mathbb{N}$ be definition is the set with some initial element (0 or 1 depending on the context) and the image of the successor function when iterated any finite number of times, we get that $\mathbb{N}$ has the well ordering principle.

We can now use the Well Ordering Principle to establish a dual technique to induction – Infinite Descent.

Let P be a map between natural numbers and logical statements – i.e. for each n, $P(n)$ is some statement.

If $P(n)$ does not hold for all $n \in N$ then we can examine $X = \{n \text{ such that } P(n) \text{ does not hold}\}$. By the well ordering principle X has a least element, x. So there has to be some *least* element that $P(n)$ fails on if it fails to hold in general. So if we want to show that P holds for all n, we can just show that there is *no* smallest counterexample.

How would we do this?

Assume that $P(x)$ is the "smallest" false statement, if you can show that implies that $P(x - k)$ for some $k > 0$ is false then P must be true for all n.

Why? Because $x - k$ is smaller than x, so we have found a "smaller" false statement than $P(x)$ – but our assumption was that $P(x)$ was the "smallest" false statement, so we have a contradiction.

This is an example of a proof by contradiction, and so it can be a pretty unintuitive type of proof. It is inherently non-constructive, we aren't showing that P is *true*, we are showing that it can't be false.

The best way to get a grasp on Infinite Descent is by doing an example!

$\sqrt{2}$ is irrational

Another way of phrasing this, $\sqrt{2}$ is irrational is the same as saying $P(n)$ is the proposition that "there is *no* expression $\frac{n}{q}$ for any $q \in \mathbb{N}$ such that $\frac{n}{q} = \sqrt{2}$".

To do infinite descent we just need to look at the counterexamples: n such that there *is* an expression $\frac{n}{q}$ with $q \in \mathbb{N}$ such that $\frac{n}{q} = \sqrt{2}$, and show that if one exists we can find a smaller counterexample. Then we know there is no least counterexample, and so there cannot be *any* counterexample.

Proof. Let $\sqrt{2}$ be rational, then $\sqrt{2} = \frac{p}{q}$ where p, q are natural numbers and $\frac{p}{q}$ is in simplest form – i.e. we cannot write $\sqrt{2}$ using a smaller numerator.

We note than that:

$$\left(\frac{p}{q}\right)^2 = 2 \tag{22}$$

$$p^2 = 2q^2 \tag{23}$$

So 2 divides p^2. As 2 is a prime number, if 2 divides p^2 then it divides p.[8] As 2 divides p then we can write $p = 2r$ for some $r \in \mathbb{N}$ with $r < p$

Thus we get:

$$4r^2 = 2q^2 \tag{24}$$

$$2r^2 = q^2 \tag{25}$$

So 2 divides q^2, and so 2 divides q – so we can write $q = 2s$ for some $s \in \mathbb{N}$.

Thus:

$$\sqrt{2} = \frac{p}{q} = \frac{2r}{2s} = \frac{r}{s}$$

However $r < p$, and our assumption was that p was the smallest possible numerator, so we have a contradiction. As we have a contradiction one of our assumptions must be incorrect, but we only made one assumption - that $\sqrt{2}$ was rational.

Thus $\sqrt{2}$ is irrational. □

[8] If a prime n divides $a * b$ then n divides a or n divides b – this is one way of defining prime numbers, and is also called Euclid's Lemma

1.8 Exercises

1. **Divisibility I:** Prove that for all n

$$11^n - 6$$

Is divisible by 5.

2. **Divisibility II:** Prove that $n^2 + n$ is even for all n.

3. **Divisibility III:** Show that for all integers $n \geq 1$:

$$5^k + 1$$

Is an even number.

4. **Sequences I:** Show for all integers $n \geq 1$:

$$\sum_{i=1}^{n} i^2 = 1^2 + 2^2 + ... + n^2 = \frac{n(n+1)(2n+1)}{6}$$

5. **Sequences II:** Show for all integers $n \geq 1$:

$$\sum_{i=1}^{n} i * (i+1) = 1 * 2 + 2 * 3 + ... + n * (n+1) = \frac{n(n+1)(n+2)}{3}$$

6. **Fibbonaci Numbers:** Let

$$\varphi = \frac{1+\sqrt{5}}{2} \qquad \psi = \frac{1-\sqrt{5}}{2}$$

And let F_n be the n-th Fibbonaci number ($F_{n+2} = F_{n+1} + F_n$. Use induction to show:

$$F_n = \frac{\varphi^n - \psi^n}{\varphi - \psi}$$

Hint: You need two base cases for this, $n = 0$ and $n = 1$.

7. **Irrationality of $\sqrt{k}$:** Let k be a positive integer such that $\sqrt{k}$ is not an integer. Prove that $\sqrt{k}$ is irrational.
Hint: Remember how we showed $\sqrt{2}$ was irrational.

8. **DeMorgan's Law:** DeMorgan's Law states that for sets A, B:

$$\overline{(A \cup B)} = \overline{A} \cap \overline{B}$$

We can use induction to generalize this though. Prove that if $A_1, ..., A_n$ are sets then:

$$\overline{(A_1 \cup A_2 \cup ... \cup A_n)} = \overline{A}_1 \cap \overline{A}_2 \cap ... \cap \overline{A}_n$$

Hint: $A_1 \cup ... \cup A_{n-1}$ is a set.

9. **Commutativity of Addition in $\mathbb{N}$:** Using induction prove that for any $a, b \in \mathbb{N}$:

$$a + b = b + a$$

Hint: Induct on a and then induct on b. In addition $a + 0 = 0 + a$ for all a, and in general $a + S(b) = S(a + b)$ where S is the successor function.

10. **Proof of Correctness:** Consider the usual algorithm for bubble sort. Prove that at the n'th pass through the list, the top n elements are sorted – thus a list of length n takes at most n passes through the list to sort, and use this to justify why bubble sort is an $O(n^2)$ algorithm.[9]

[9] This "proves the correctness" of bubble sort. These proofs are very important to show that the algorithms we develop do what we want them to.

2 Graph Theory

2.1 Graphs

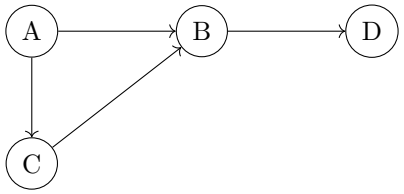
A **Graph** is a collection of *vertexes/nodes* and *edges* between the nodes.

Formally a graph G is a tuple, (V, E) where V is some set and $E \subseteq V \times V$.

If $(a, b) \in E$ that means that there is an edge going **from** a **to** b.

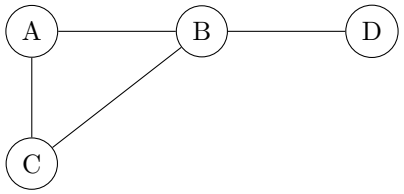
In general, $(a, b) \in E$ does not mean that $(b, a) \in E$.

A graph can be **directed**:



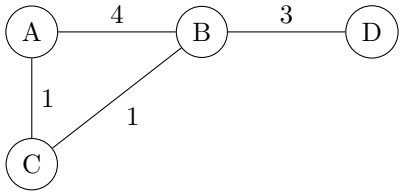
Meaning that you can only "move" along the edges in one direction (in the above graph you can move from A to C but not from C to A)

Or a graph can be **undirected**:



Meaning that if $(a, b) \in E$ then $(b, a) \in E$ – if you can move from A to C then you can move from C to A.

Oftentimes we also deal with **weighted graphs**, a weighted graph is a graph (V, E) along with a function $w : E \to \mathbb{R}$, where $w(a, b) \geq 0$ for all $(a, b) \in E$. A weighted graph can be directed or undirected.



The weight of an edge ($w(a, b)$) is the "distance" or "cost" of moving along that edge.

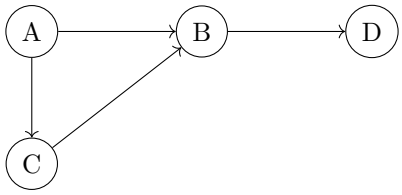
Weighted graphs come up in many natural contexts in Computer Science.

A natural question might be: "How could we represent graphs in a computer?"

The two most common ways are via a linked-list analogue, by having a list of vertices, each with a list of outbound edges.

The other is via what is called an **adjacency matrix**, a matrix with a 1 in the i, j'th entry if the edge (i, j) exists.

Consider the directed graph below:



We could write this as a list:

```
graph = {
    "A" : ["B", "C"],
    "B" : ["D"],
    "C" : ["B"],
    "D" : []
};
```

Or as a matrix:

$$\begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

As a note, it is the most common convention to consider that $(a, a) \in E$ for all $a \in V$ – every vertex has an edge to itself.

Adjacency matrices are often easier to work with computationally, but many graphs can be **sparse**, meaning that almost all entries in the matrix are zero. For this reason most libraries have a *sparse matrix* that you can use to represent graphs with.

A few more important definitions:

The **degree** of a node is the number of inbound edges to that node.

A **path** is a sequence of edges (a_i, b_i) in a graph such that $b_i = a_{i+1}$.

A graph is **connected** if there is a path between any pair of nodes.

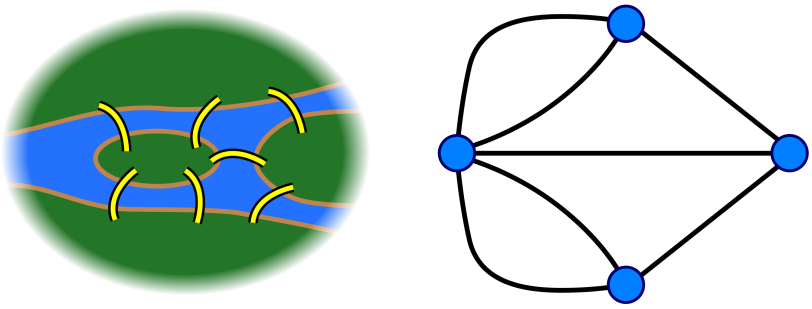
2.2 Paths - Hamiltonian and Eulerian

The city of Königsberg in Prussia is set on a forked river with two central islands. Crossing this river are seven bridges.

The question that was raised was the following: "Is it possible to find a path through Königsberg which crosses each bridge exactly once and ends where you started?"

Take a moment to examine this question and see if you can solve it.

Leonhard Euler answered this question in 1736 by creating the first theorem in Graph Theory. What Euler had noticed was that the landmasses themselves



(a) Layout of the Seven Bridges (b) Represented as a Graph

Figure 1: Courtesy of Wikimedia: (a), (b)

do not matter. Instead we can represent the bridges as a graph, with a node per landmass and edges for each bridge[10]

We can formulate the problem as a problem about *circuits* in graphs: A **circuit**[11] is a path which starts and ends at the same node.

The question then becomes the **Eulerian Circuit Problem**: For a given graph (V, E) does there exist a circuit which visits each edge exactly once. Such a circuit is called a *Eulerian Circuit*

Take a moment to try and see if you can find a necessary condition for a graph to have a Eulerian Circuit.

Euler noticed a particular feature of these paths. For every step of the path except the first and last step, we enter a node and then leave it – if our path is a circuit then our first step leaves the initial node and our last step enters it. So it is necessary that each node has an even number of edges. It turns out that it is not only necessary but sufficient.

Which become the first theorem of graph theory:

Theorem 2.1 (Euler's Theorem)**.** *A connected graph (V, E) has a Eulerian circuit if and only if each node has even degree.*

This observation can be generalized to Eulerian paths:

Theorem 2.2. *A connected graph (V, E) has a Eulerian path if and only if exactly zero or two nodes have odd degree.*

Which means we can solve the Eulerian cycle problem in $O(|E|)$ time! To figure out if a graph has a Eulerian cycle we just need to find the degree of each node, which we can do by simply iterating over the edges.

[10] This is technically a multi-graph rather than a graph as we allow multiple edges between two nodes. The definition of a graph given allows at most one edge between two nodes, a **multi**-graph allows **multi**ple.

[11] These are also commonly called cycles or tours.

Applying this to the Seven Bridges problem we can see that there is in fact *no* such circuit, as the graph corresponding to the Seven Bridges problem has three vertices of odd degree – it doesn't even have a Eulerian path.[12]

There's a natural next question. We have found a necessary and sufficient condition for the existence of a circuit which visits each *edge* once. What about a circuit which visits each *node* once.

A **Hamiltonian Circuit** is a circuit which visits each node exactly once. A graph that contains a Hamiltonian circuit is called a **Hamiltonian Graph**.

It would be tempting to thing that since we could solve the Eulerian cycle problem in $O(|E|)$ time that there should be a similarly quick algorithm to solve the Hamiltonian cycle problem, however this is not the case. The Hamiltonian cycle problem is *NP-Complete*[13]– so if we had an efficient algorithm to solve it then we would have shown that $P = NP$.

2.3 Flow and Maximum Flow

A common problem in computer science involves what are called *flow networks.* A **flow network** is a weighted graph (V, E) with two special nodes, $s, t \in V$ the *source* and the *sink* respectively.

The weights on the network are called the *capacities* of the edges.

A **flow** is a map $f : E \to R$ such that:

- The flow of an edge cannot exceed its capacity: $f(u, v) \leq c(u, v)$
- The sum of the flows entering a node must equal the sum of flows exiting that node, except for the source and sink:

$$\forall v \in V \setminus \{s, t\} : \qquad \sum_{u:(u,v)\in E} f(u, v) = \sum_{u:(v,u)\in E} f(v, u)$$

The **value of a flow** is the amount of flow going from source to sink:

$$|f| = \sum_{v:(s,v)\in E} f(s, v)$$

The **max flow problem** is the question of finding a flow which maximizes $|f|$.

Flow problems tell you the maximum amount of traffic that an architecture can support in bringing customers from point A to B.

In solving this problem, computer scientists noticed that there was a relation to another common problem, the *Min-cut* problem:

Theorem 2.3 (Max-flow Min-cut)**.** *In a flow network the maximum flow is equal to the total weight of the edges in a minimum cut.*

[12] Due to bombing in World War 2 which destroyed two bridges, it is now possible to find a Eulerian *path* (but still not cycle) through the bridges of Königsberg.

[13] See section 3.3 for details on what $NP - Complete$ means.

The result of this theorem is that the problem of finding a minimum cut is the same as the problem of finding maximum flow. Both min-cut and max-flow are common problems to find in computer science. As min-cuts are easier to verify, this theorem gave a method to approach this problem and lead to the **Ford-Fulkerson algorithm**:

```
Inputs: A Network G = (V,E) with flow capacity c, source s and sink t.
Output: A flow f from s to t of maximum value
    1. f(u,v) = 0 for all edges (u,v)
    2. While there is a path p from s to g in G such
    that c(u,v) > f(u,v) for all edges (u,v) in the path:
        i. Let cp = min{c(u,v) - f(u,v) : (u,v) in path p}
        ii. For each edge (u,v) in path p:
            a. f(u,v) = f(u,v) + cp
            b. f(v,u) = f(v,u) - cp
```

Ford-Fulkerson has complexity $O(|E||f|)$ where $|f|$ is the max flow, and is guaranteed to terminate as long as all capacities in the graph are rational.

2.4 Exercises

1. **Graph Counting I:** A **complete** graph is a graph where between any two nodes a, b there is an edge. How many edges are in an undirected complete graph with n nodes?

2. **Graph Counting II:** What is the minimal number of bits required to encode the adjacency matrix of any directed, unweighted graph with n nodes?

3. **Graphs and Paths I:** Write an algorithm that when given a graph with a countably infinite number of vertices will output a shortest path between two nodes.

   Hint: Depth first search will not work. Why?

4. **Graphs and Paths II:** Write an algorithm that when given a graph (V, E) outputs a Eulerian circuit. An optimal algorithm is $O(|E|)$. Prove the correctness of your algorithm.

5. **A Foray into the Tropical I:** The *tropical semiring* is the usual set of real numbers $\mathbb{R}$ along with ∞, except its addition $\oplus$, and multiplication $\otimes$ are different:

   $$a \oplus b = \min(a, b)$$

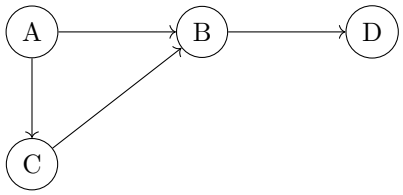
   and

   $$a \otimes b = a + b$$

   So $3 \oplus 5 \otimes 2 = 3 \oplus 7 = 3$

We can use these operations to define tropical matrix multiplication – which works the same as regular matrix multiplication but we use the tropical operations.

We can use tropical matrix multiplication to solve the shortest path problem. We can turn a graph:



Into a tropical matrix:

$$\begin{bmatrix} 0 & 1 & 1 & \infty \\ \infty & 0 & \infty & 1 \\ \infty & 1 & 0 & \infty \\ \infty & \infty & \infty & 1 \end{bmatrix}$$

By placing ∞ in the entries with no corresponding edges, 0 in the self loops, and 1 (or the weight of the edge) on the edges.

(a) Show that if A is the tropical matrix representing a graph, then A^n (using tropical matrix multiplication) has in entry i, j the length of the shortest path between i and j that uses at most n steps, and that $\lim_{n\to\infty} A^n$ has as entries the shortest paths between any two nodes.

(b) How can you extend this to a graph with arbitrary weights?

(c) How could we compute what $\lim_{n\to\infty} A^n$ is?

6. **A Foray into the Tropical II:** The assignment problem is the following: You have a set of tasks T and a set of workers W and a cost to assigning a worker i to a task t: $c(i, t)$. Each worker can be given at most a single task. What is the minimum cost possible to complete all tasks?

   This can be represented as a graph problem where you have nodes for each task and worker, and an edge between each worker and task with weight $c(i, t)$. The problem becomes the problem of picking a subgraph of this graph that looks like a bunch of pairings of workers and tasks which has minimal total weight.

   This can also be solved tropically. You can set up a matrix:

$$\begin{bmatrix} c(1,1) & c(1,2) & c(1,3) & c(1,4) \\ c(2,1) & c(2,2) & c(2,3) & c(2,4) \\ c(3,1) & c(3,2) & c(3,3) & c(3,4) \\ c(4,1) & c(4,2) & c(4,3) & c(4,4) \end{bmatrix}$$

If you take the tropical *permanent* of this matrix, it gives you the minimum cost possible.

The *permanent* is the determinant if you do not negate anything.

Explain why this works and use this fact to find an efficient algorithm to find the tropical permanent of a matrix

7. **Max Flow:** The Ford-Fulkerson algorithm is not the most efficient algorithm to find the max flow. Find an improvement to Ford-Fulkerson to make its runtime polynomial in V and E

   Hint: Consider how you can choose the path in step 2

   Hint 2: Modifying that step can create the *Edmonds-Karp algorithm.* Further improvements get you *Dinic's algorithm.*

3 Automata and Computation Theory

In the early to mid 1900s math underwent what is now called the "Foundational crisis", a series of theorems began to shake the belief in a single mathematical system that could prove or disprove any statement. The limits of math were beginning to be found.

Prior to the foundational crisis, if you asked a mathematician "What problems *can* we solve?" their answer would be "Everything.", but in the midst and afterwards that became an open question.

What problems can we actually solve?

As that question was (and is) explored the emphasis shifted slightly:

What problems can *we* actually solve?

As some problems may have theoretical algorithms that can solve them, but that's not terribly useful if it would take the lifetime of the universe to be able to approach the problem.

3.1 Chomsky's Hierarchy of Languages

In order to answer these questions we needed a mathematical model of what an algorithm was, and what we *could do*. Such a model is called a **model of computation** or an **automata**.

Automata are usually used as *deciders* or *recognizers*. They will take in an input and output if it has some property or not. The **language** of an automata is the set of inputs that it *accepts* or says that it has the desired property.

Automata may also be used as *transformers*, where they take in an input string and output a different string, but for the purpose of this lecture we will be examining deciders.

We can break automata into wide types of automata: finite state machines are one such class, and called finite automata, and turing machines are another. We can then examine these types of automata and ask "What possible languages[14] can these automata express?"

Different classes of automata have different classes of languages that they can describe.

In 1956 the linguist Noam Chomsky, in the process of formalizing syntax and semantics as structural features of language, described a hierarchy of languages, each with a corresponding automata that recognized them. While there are other "hierarchies of language", and other classes of automata and corresponding languages, Chomsky's is the canonical starting point.

While it may seem intuitive that there would be such a hierarchy, or that not all classes of automata can recognize the same classes of languages, mathematically it is not straightforward to show that something *can't* do something. so I would like to walk through an instance of how you would do that.

[14] A language in general just being a set of strings, usually in binary, so $\{0, 10, 111\}$ would be a finite language, $\{0^n 1^{2^n} : n \in \mathbb{N}\}$ would be an infinite language or even $\emptyset$ would be a language.

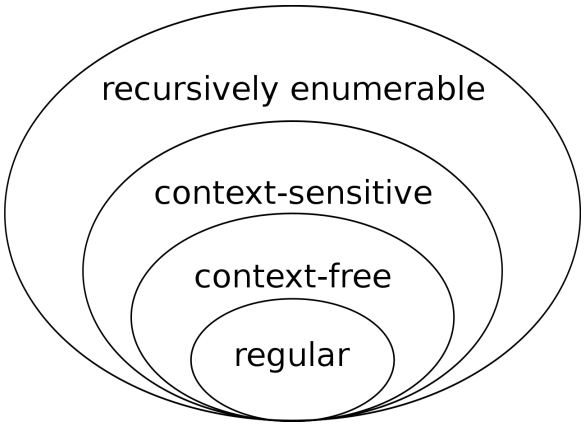


Figure 2: Chomsky's Hierarchy of Languages
Courtesy of Wikimedia: Here

Class of Language	**Type of Automata**	**Example Language**
Recursively Enumerable	Turing Machine	HALT
Context Sensitive	Linearly Bounded Automata	$\{a^n b^n c^n : n \in \mathbb{N}\}$
Context Free	Pushdown Automata	{w : w is a pallindrome}
Regular	Finite Automata	$\{w : w$ has an even number of 0's$\}$

Figure 3: Classes of languages with corresponding automata

We will show that the language of pallindromes is outside of the class of languages that Finite Automata can accept.

First, what is a *Finite Automata*?

The main type of finite automata is a **Deterministic Finite Automata** or DFA[15]. A DFA is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where:

- Q is a finite set of states.
- Σ is the finite alphabet, a finite set of symbols that the input string is made of.
- δ is the transition function, $\sigma : Q \times \Sigma \to Q$
- q_0 is the start state
- $F \subseteq Q$ is the set of acceptance states.

If $w = w_1 w_2 ... w_n$ is a string with each $w_i \in \Sigma$, the automata accepts w if and only if there is a sequence of states $q_0, q_1, ..., q_n$ with

- q_0 is the start state

[15]There are also **Nondeterministic Finite Automata** (NFA) but their computational power is equivalent to DFAs, so in the interest of simplicity and time we will only cover DFAs.

- $q_{i+1} = \delta(q_i, w_{i+1})$ for $i = 0, ..., n-1$
- $w_n \in F$

We can graphically represent a DFA as a state machine.

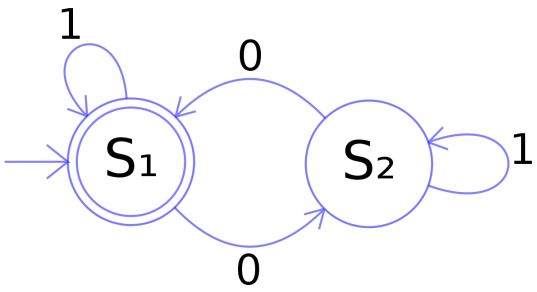


Figure 4: A DFA recognizing $\{w : w$ has an even number of 0's$\}$
Courtesy of Wikimedia: Here

The above rules say that a string is accepted or recognized by a DFA if and only if you can start at the start state, and then follow the corresponding arrows for each character in the string and then end at an accept state.

Although the "weakest" well known automata the class of finite automata is still pretty useful. The languages of regular expressions is the same as the class of languages of finite automata. There is a strong argument to use the "weakest" possible model of computation to solve a problem, as while analyzing Turing Machines is often an impossible task, analyzing finite automata is far easier.

With that being said, let's show a limit in this model of computation. We will prove that the language:

$$\{w : w \text{ is a pallindrome}\}$$

Cannot be expressed by a DFA.

First we will prove a general result and then use it to prove that the language of pallindromes is not expressible by DFAs.

Theorem 3.1 (Pumping Lemma)**.** *Let L be a regular language. There exists a number $p \geq 1$ (called the **pumping number**) depending only on L such that for each string in w of length greater than or equal to p, w can be expressed as xyz where x, y, z are strings such that:*

- $|y| \geq 1$
- $|xy| \leq p$
- $xy^n z \in L$

Proof. If L is a regular language then there exists a DFA $(Q, \Sigma, \delta, q_0, F)$ which recognizes L.

Let $p = |Q| + 1$. We note that if $w \in L$ and $|w| \geq p$, let $q_0, ..., q_n$ be the sequence of states that w passes through.

As $p > |Q|$ then by the pigeonhole principle there must be at least state repeated twice in $q_0, ..., q_p$, let that be q_i and q_j with $i < j$.

Let $x = w_0 w_1 ... w_{i-1}$, $y = w_i w_{i+1} ... w_{j-1}$ and $z = w_j w_{j+1} ... w_n$

We can see that as $0 \leq i < j \leq p$ then $|xy| = |w_0 w_1 ... w_{j-1}| = j - 1 < p$, and $|y| \geq 1$

We note then that the path that the string $xy^n z$ follows in the DFA will be $q_0 ... q_{i-1}(q_i ... q_j)^n q_{j+1} ... q_n$. The string w was accepted, so $q_n \in F$, thus $xy^n z \in L$.

Thus every string of length greater than or equal to p has a proper decomposition. □

We can now use this theorem to show that DFAs cannot recognize the language of palindromes.

Theorem 3.2. *Let $L = \{$Palindromes made of 0s and 1s$\}$. This language cannot be expressible by a DFA.*

Proof. Assume that L is a regular language. Let p be the pumping number of L.

Consider the string $w = 0^p 1^p 0^p$, as $|w| = 3p$ then by the pumping lemma there exist x, y, z such that $w = xyz$, $|y| \geq 1$, and $|xy| \leq p$.

As $|xy| \leq p$ then $xy = 0^j$ for some $0 \leq j \leq p$.

As $|y| \geq 1$ then $y = 0^i$ and $x = 0^{j-i}$ for some $0 < i \leq j \leq p$.

By the pumping lemma $xy^2 z \in L$, however:

$$xy^2 z = 0^{j-i} 0^{2i} 0^{p-j} 1^p 0^p = 0^{p+i} 1^p 0^p$$

As $i > 0$ we can see that $xy^2 z$ is not a palindrome, so $xy^2 z \notin L$ and we have a contradiction.

As our only assumption was that L was a regular language we can then see that L is not a regular language. □

So we've done our first proof about computation! We know that the model of computation given by DFAs cannot recognize palindromes.

3.2 Turing Machines: Decidable vs Recognizable

Computers are currently made according to the Von-Neumann architecture which can be represented by the mathematical model of computation called a Turing Machine based on the work of Alan Turing.

Any algorithm you write can be turned into a Turing Machine.

A class of automata is called **Turing Complete** if it can simulate a Turing machine. Such an automata then likely shares its class of languages with those of Turing machines – the *Recursively Enumerable* class of languages.

Proofs themselves can usually be converted into the form of Turing machines – according to the Church Turing thesis any form of computation that we are able to do with paper and pencil can at most be recursively enumerable.

Turing machines do not need to be able to stop however. Unlike finite automata which will always either accept or reject an input, a Turing machine has another option, it can loop infinitely.

This splits the class of recursively enumerable languages into three important classes:

A recursively enumerable language is **Decidable** if there exists a Turing machine that recognizes it that will always either accept or reject an input in a finite amount of time.

A recursively enumerable language is **Recognizeable** if there exists a Turing machine that will always accept an input if it is in the language in a finite amount of time, and for inputs not in the language it will either halt or loop infinitely.

A recursively enumerable language is **Co-Recognizable** if there exists a Turing machine that will always reject an input if it is not in the language in a finite amount of time, and for inputs in the language it will either halt or loop infinitely.

As sets: **Decideable** = **Recognizable** ∩ **Co-Recognizable**.

Not every problem is decidable. The canonical example is $HALT$ – the Halting Problem: Given a description of a Turing machine T and an input x, $\langle T, x\rangle \in HALT$ if and only if T halts on the input x.

3.3 P vs NP

Let's zoom in on the class of Turing Decideable languages. Inside of this class is a whole large variety of *complexity classes* – classes of languages determined by the speed or space required of Turing machines used to compute them. A list of many of the interesting complexity classes can be found here.

Before we talk about P vs NP we need to briefly go over some terminology and techniques.

A problem[16] A is said to **reduce** to a problem B if we can create an algorithm using an *oracle*[17] for B to solve A. A is said to X**-reduce** to B if that algorithm exists in the complexity class X.

If we have a complexity class X, we can create another complexity class $X - hard$, the class of problems Y such that *every* problem in X X-reduces to Y.

We then create another complexity class $X-complete$, where $X-complete = X \cap X - hard$, or $X - complete$ is the class of problems Y such that Y is in X and *every* problem in X X-reduces to Y.

Now we can talk about the P vs NP problem.

[16] A problem is the same as a language, but when talking about complexity classes we tend to talk about *problems* instead of *languages* – the only difference is that a problem is usually phrased as a question (i.e. does this turing machine halt on this input?) versus languages which are phrased as sets.

[17] An oracle is a hypothetical machine that can instantly solve a problem

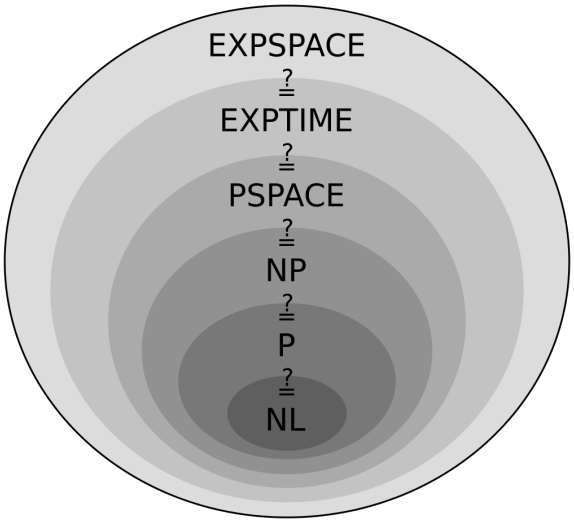


Figure 5: A subset of the complexity hierarchy
Courtesy of Wikimedia: Here

P is the polynomial-time complexity class. A problem is in P if there exists a Turing machine that takes $O(n^k)$ time for any constant k where n is the length of the input. In general we like problems to be in P, as it tends to mean that *eventually* we'll have computers powerful enough to solve it for all interesting cases.

NP is a little harder to describe. There are two equivalent ways to describe it:

1. A problem is in NP if it can be solved in polynomial time by a Nondeterministic Turing Machine.

2. A problem is in NP if for each accepted string there is a proof of why it is accepted which can be verified in polynomial time.

For instance if the problem was: N is divisible by 15, we offer as proof for the number 153482175, the number 10232145, as $15 * 10232145 = 153482175$ we can verify that 153482175 is accepted by our problem.

P vs NP is the open, million dollar worth, question: Is the class of languages NP the *same* as P, or is it different?

We know $P \subseteq NP$, as for each problem in P our proof can just be running our machine.

To prove that $P = NP$ we would just need to take a *single* $NP - complete$ problem and show it is in P. So if you can find a polynomial time algorithm for the Travelling Salesman problem[18], the Hamiltonian Path problem[19], solving

[18] Given a weighted graph, find the most efficient path to visit each node
[19] Given a graph, does it have a Hamiltonian path?

$n \times n$ sudoku puzzles, or many other surprisingly common problems then $P = NP$.

To prove that $P \neq NP$ we would have to show that for any $NP - complete$ problem there *cannot* be a polynomial time algorithm. Unfortunately lower bounds are extremely hard to prove for complexity – bordering on impossible for many problems.

3.4 Exercises

1. **Finite Automata I:** Create a finite automata that recognizes the language: $\{w : w$ has an odd number of 0's$\}$ where strings are expressed in binary.

2. **Finite Automata II:** Create a finite automata that recognizes the language: $\{w : w$ has an even number of 0's and all three character substrings are 001 or 101 $\}$ where strings are expressed in binary.

3. **Finite Automata III:**

   (a) Prove that the class of regular languages is closed under union.

   (b) Prove that the class of regular languages is closed under compliment.

   (c) Prove that if a class of sets is closed under unions and compliments then it is closed under intersection.

4. **Finite Automata IV:** Use the pumping lemma to prove that the language $\{0^n 1^n\}$ is not expressible by a DFA.

5. **Finite Automata V:** Prove the language $\{0^a 1^b | a \neq b\}$ is not expressible by a DFA.

   Hint: Examine problems 3 and 4 and use the fact that $\{0^a 1^b\}$ is expressible by a DFA without the restriction that $a \neq b$.

6. **Turing Computability I:** Prove that $HALT$ is Turing recognizable.

7. **Turing Computability II:** Prove that $HALT$ is not decidable.

   Hint: Assume it is. Then there is a Turing machine that decides it. Use that Turing machine to construct a Turing machine that would cause something akin to the Barber paradox.

8. **Turing Reductions I:** Examine the problem ES: Given a Turing machine M, accept if and only if the language of M is empty.

   Prove that this problem is undecidable by finding a reduction from $HALT$ to ES. (If EQ were decidable and such a reduction existed then $HALT$ would be which we know it is not.)

9. **Turing Reductions II:** Examine the problem EQ: Given two Turing machines M, N accept if and only if the language of M is the same as N

   Prove that this problem is undecidable by finding a reduction from $HALT$ to EQ.