CSC236 Notes

https://github.com/ICPRplshelp

December 12, 2022

Contents

1 Introduction

Computer science is more than programming, the same way that match is way more than calculators. Math is not about calculators; it is just a tool you use. Chemistry is way more than test tubes. Programming is what you learn to learn about the computer science.

Programming is more than trial and error. This course is about reasoning about al-

gorithms. The process that you use to prove that an algorithm is correct is extremely helpful when you are designing code. If you have these at the back of your mind, you can build it up and know that it is correct as you go.

We can reason about code:

- Is this code correct?
- Is this code efficient?

We'll use:

- Induction
- Algorithm correctness
- Algorithm runtime analysis
- Recursive algorithms, such as divide and conquer
- Formal language theory

2 Induction

2.1 Principle of simple induction

Feel free to use n for your quantifier, even though many prefer the usage of the variable k .

What is the goal of simple induction?

- Prove something in a very specific form: that a particular statement is true for every natural number: $\forall n \in \mathbb{N}, P(n)$
  - Naturals include 0 in the context of CS.
  - $P(n)$ is a predicate: P: $\mathbb{N} \to \{\text{True}, \text{False}\}$
- What do we do?
  - Prove a base case: $P(0)$
  - Prove an inductive step: $\forall n \in \mathbb{N}, P(n) \Rightarrow P(n+1)$
    - Let $n \in \mathbb{N}$. Assume $P(n)$. We want to prove $P(n+1)$.

– Conclude $\forall n \in \mathbb{N}, P(n)$.

Why does that make sense? Because of the principle of simple induction:

$$(P(0) \wedge \forall k \in \mathbb{N}, P(k) \Rightarrow P(k+1)) \Rightarrow \forall n \in \mathbb{N}, P(n)$$

We may use it. We do not need to prove this – we can take the PSI as a given.

One simple example:

Prove $12^n - 1$ is divisible by 11, $\forall n \in \mathbb{N}$ (this defines $P(n) = 11 \mid 12^n - 1$).

a is divisible by b if $\exists q \in \mathbb{Z}, a = bq$.

Proof. Base case: $P(n) = 0$.

$$12^0 - 1 = 0$$

0 is divisible by any number other than 0, so it is divisible by 11.

Inductive step: $P(n) \Rightarrow P(n+1)$. Assume $11 \mid 12^n - 1$. We want to show that $11 \mid 12^{n+1} - 1$. This means that we know that

$$\exists q \in \mathbb{Z}, 12^n - 1 = 11q$$

We want to show that $\exists r \in \mathbb{Z}, 12^{n+1} - 1 = 11r$. Choose $r = 12q + 1$. Our work starts here:

$$\begin{aligned} 12\,(12^n - 1) &= 132q \\ 12^{n+1} - 12 &= 132q \\ 12^{n+1} - 1 &= 132q + 11 \\ 12^{n+1} - 1 &= 11(12q + 1) \\ 12^{n+1} - 1 &= 11r \end{aligned}$$

■

Second example:

Suppose you only have 3-cent and 7-cent stamps. What postage amounts can you pay for exactly? (We can pay 12, 13, 14 exactly, and that is enough intuition – just add 3 when we can)

What to prove: We can pay exactly for any amount of 12 cents or more.

Formally? $\forall n \in \mathbb{N}, n \geq 12 \Rightarrow \exists a,\ b \in \mathbb{N}, n = 3a + 7b$. What is my predicate?

$$P(n) = n \geq 12 \Rightarrow \exists a,\ b \in \mathbb{N}, n = 3a + 7b$$

Can we prove this by induction? We can, but it is going to be messy:

Attempt 1: If we use simple induction: Base case: $0 \geq 12 \Rightarrow \exists \cdots$

- This statement is vacuously true.

Inductive step: Let $k \in \mathbb{N}$. Assume $P(k)$, prove $P(k+1)$. The statements we're working is messy, and the proof gets complicated, as we have extra assumptions to have. Assuming that $k \geq 12$ does not mean that you may assume that the k in $P(k+1)$ is greater or equal than 12. That could cause issues.

Attempt 2: Modify simple induction a bit, where the base case is higher:

$$P(n) : \exists a,\ b \in \mathbb{N}, n = 3a + 7b$$

Base case: Prove $P(12)$ (we already know that k being 0 to 11 is vacuously true).

Inductive step: Let $k \in \mathbb{N}$ with $k \geq 12$. Assume $P(k)$. Prove $P(k+1)$.

What is the problem? Both attempt 1 and attempt 2 have a problem. Attempt 1 is messy in terms of the notation and formalism. Attempt 2 focus on the core thing we care about, but it relies on this idea is that to prove something with 15, we must use 14. We could modify simple induction to have three cases, but if we try to do this with other problems, it could get extremely messy.

2.2 Complete induction

Principle of complete induction:

$$(P(0) \wedge \forall k \in \mathbb{N},\ (\forall i \in \mathbb{N} \leq k,\ P(i)) \Rightarrow P(k+1)) \Rightarrow \forall n \in \mathbb{N},\ P(n)$$

In the course notes, it is written like this (no base case explicitly being mentioned):

$$(\forall k,\ (\forall i < k,\ P(i)) \Rightarrow P(k)) \Rightarrow \forall n,\ P(n)$$

2.3 Induction pitfalls

Proposition 1 (Same age)**.** *Everyone in this class has the same age.*

$$P(n): \textit{ In every set of } n \textit{ students, everyone has the same age.}$$

Proof. Base case: Prove $P(1)$. In every set that has one student, everyone here has the same age. So let S be an arbitrary set with one student. Does every student in S have the same age? Yes, as there is only one student.

Induction step: Let $k \geq 1 \in \mathbb{N}$. Assume $P(k)$ (give me any set of k students, and everyone in that set has the same age). Prove $P(k+1)$.

Let S be an arbitrary set with $k+1$ students. Symbolically: $S = \{s_1,\ s_2,\ \ldots,\ s_k,\ s_{k+1}\}$.

Consider $S_1 = S - \{S_{k+1}\} = \{S_1,\ \ldots,\ S_k\}$. By the induction hypothesis, everyone in S_1 has the same age.

Consider $S_2 = S - \{S_1\} = \{S_2,\ \ldots,\ S_k,\ S_{k+1}\}$. By the induction hypothesis, everyone in S_2 has the same age.

So age $(s_1) =$ age $(s_2) = \ldots =$ age $(s_k) =$ age $(s_{k+1}) \Rightarrow P(k+1)$ ■

This proof proved something incorrectly. Is there something we did wrong?

- There are no issues with the predicate.
- There are no issues with the base case. This statement is true or false – expand what "everyone" means.

About the inductive step:

- The "what to prove" is correct.
- Using the induction hypothesis more. Then once is okay.

The problem:

- We assumed that student 2 to student k exists. However, $s_2 \ldots s_k$ can only exist if and only if $k \geq 2$. We never proved $P(2)$. This breaks down as the argument I used on the proof depends on knowing that $k \geq 2$ and I don't know that – I only know that $P(1)$ is true, and that makes the whole proof fall apart.

If I proved $P(2)$, then I can safely assume that s_2 exists.

Moral of the story – be careful – the details do matter. In other words, you need to make sure that k is arbitrary. You can't rely on the fact that any $P(k')$ is true unless you've proven it first, with that value of k' itself.

Tip: start from $k+1$ and work back.

2.4 Principle of complete induction, generalized

Where $b,\ e \in \mathbb{N}, b \leq e$ (begin, end) – these are the base cases. What if you want to prove something not starting at 0, but rather starting at b, and you need multiple base cases? Our goal is to reach this conclusion:

$$\forall n \geq b,\ P(n)$$

What do we do?

$$(P(b) \wedge \ldots \wedge P(e) \wedge \forall k > e,\ P(b) \wedge \ldots \wedge P(k-1) \Rightarrow P(k))$$

$$\Rightarrow \forall n \geq b,\ P(n)$$

What do I want to show from this? I can directly read a proof template.

2.5 Proof template for complete induction

It is there in the principle of complete induction. All parts come directly from the principle of complete induction, left to right.

Base: Prove $P(b) \wedge \ldots \wedge P(e)$. Prove each one.

1. Inductive step: Prove an implication with a universal quantifier: Let $k > e$.
2. Assume the hypothesis: $P(b) \wedge P(b+1) \wedge \ldots \wedge P(k-1)$
3. Under those conditions, prove $P(k)$.
4. Conclude that $\forall n \geq b,\ P(n)$.

2.5.1 Complete induction without base case

Prove that every integer has a prime factorization. Spelt out as a predicate:

$$P(n) : n \text{ has a prime factorization}$$

At least except $P(1)$. We still have cases somewhere in our proof that do not explicitly rely on other values, but it gets rolled into the proof in a different way. I'm not going to complete write down what prime factorization means.

$$P(n) : n = p_1 \cdot p_2 \cdots p_k \text{ for some primes } p_1,\ \ldots,\ p_k,\ k \geq 1$$

Here, I want to treat numbers that are prime have their own prime factorization. Why not use simple induction? Here's the problem. Simple induction lets you assume something about k and prove $k+1$. Here's an example: 83 is a prime number, and it is thus its own prime factorization: 83. $84 = 2 \cdot 2 \cdot 3 \cdot 7$. $85 = 5 \cdot 17$. $86 = \ldots$. Can't easily go from k to $k+1$, but if a number, there is a way that it can depend on smaller number.

Using complete induction with no base case: Start with the inductive step.

Proof. Let $k \geq 2$. Assume $P(2),\ P(3),\ \ldots,\ P(k-1)$ have a prime factorization. Prove that k has a prime factorization.

Given an arbitrary k, either k is prime or not.

Case 1: If k is prime, then $k = k$ (it is its own prime factorization).

Case 2: If k is not prime, it has factors. $k = a \cdot b$ for some $a,\ b$ with $2 \leq a,\ b < k$. But then, by the inductive hypothesis, both a has a prime factorization: $p_1,\ p_2,\ \ldots,\ p_l$, and p has a prime factorization: $q_1,\ q_2,\ \ldots,\ q_m$. ■

There are infinitely many base cases in these proofs, which happen to be all the prime numbers (which are true by default – read case 1).

2.6 Examples with complete induction

If I'm going to break a chocolate bar (I can only break a bar entirely – I can only cut across the entire width or height of a bar). If I want to break this up into individual squares, how many breaks along an entire side of the bar do I need, to split it up completely?

The answer is $n-1$, where n is the number of squares in the bar. This is true regardless of the shape of the bar.

I will prove: $\forall n \geq 1 \in \mathbb{N}$, every chocolate bar with n squares require $n-1$ breaks to split into individual pieces.

Proof. Base case: $n = 1$. There is only one chocolate bar that looks like this, and I wouldn't need to break it apart. $1-1=0$ breaks.

Inductive step: Let $k > 1$. Assume every chocolate bar (needs to be a rectangle) with j requires $j-1$ breaks to split, for $1 \leq j < k$. I want to prove that every chocolate bar with k squares will require $k-1$ breaks to split.

Let C be an arbitrary chocolate bar with k squares.

Make one break that gives me pieces A and B. They contain everything that was in C (I can always do this regardless of the shape of C, as $k > 1$).

- A contains a squares, for some $a \in [1,\ k-1]$.
- B contains b squares, for some $b \in [1,\ k-1]$.
- $k = a+b$

By the induction hypothesis, we can split A using $a-1$ breaks and split b using $b-1$ breaks. The total amount of breaks is $1+(a-1)+(b-1) = a+b-1 = k-1$ ■

2.6.1 Why use complete induction?

1. Because we can always use it
2. If you try to go from k to $k+1$, there's no way to do it.

2.6.2 How would I know if I don't need more base cases?

If your range of values for your sub-problems relies in something more than the base cases I have derived, then I'll need to go back.

3 The principle of well ordering

This and induction are all equally as strong in terms of proof techniques.

The principle of well-ordering (PWO/WOP):

*Every **non-empty** subset of $\mathbb{N}$ contains a smallest element.*

A few notes:

- Applies to finite and infinite subsets of $\mathbb{N}$. Every one of them have some smallest natural number.
- This is a property that is unique to the natural numbers. It is not true for $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$
- This is tied to the structure of $\mathbb{N}$

How do I use this to prove statements?

3.1 Using PWO in Substitution of The Principle of Simple Induction

The principle of well ordering says that every non-empty subset of $\mathbb{N}$ has a smallest element. How do we prove a predicate for all $\mathbb{N}$?

For example, we want to prove $\forall n \in \mathbb{N}$,$P(n)$ holds. Here's how to do it.

1. Assume the statement I want to prove is false: $\exists n \in \mathbb{N}, \neg P(n)$.
2. Define a set $A = \{m \in \mathbb{N} : \neg P(n)\}$. By our assumption in (1), $A \neq \varnothing$.
3. Can it contain 0? Tell me otherwise. It doesn't.
4. By PWO, A has a smallest element $m > 0$. Then, $\neg P(m)$ holds. Also, $P(m-1)$ holds.
5. Show that $P(m-1) \Rightarrow P(m)$ holds. You'll get a contradiction, as $\neg P(m)$ and $P(m)$ holds. End of proof.

The principle of well ordering is simple induction, but in a different way. In fact, the principle of simple induction can be proved in this way.

3.2 Proofs using the principle of well ordering

Prove $\forall n \in \mathbb{N}$, $\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$.

Using PWO, and you'll likely always try to get a contradiction:

Proof. For a contradiction, assume the negation:

$$\exists n_0 \in \mathbb{N}, \sum_{i=0}^{n_0} 2^i \neq 2^{n_0+1} - 1$$

Consider $S = \left\{k \in \mathbb{N} \mid \sum_{i=0}^{k} 2^i \neq 2^{k+1} - 1\right\}$.

By assumption, $n_0 \in S \neq \varnothing$ (the set above isn't empty) and contains at least n_0.

By PWO, S has a smallest element m.

- $m \in S \Rightarrow \sum_{i=0}^{m} 2^i \neq 2^{m+1} - 1$
- $0 \notin S,\ 1 \notin S,\ \ldots,\ m-1 \notin S$

What could m be? Could $m = 0$? **No**, because $\sum_{i=0}^{0} 2^i = 2^0 = 1 = 2 - 1 = 2^{0+1} - 1$

So $m > 0$. So $m - 1 \geq 0$, so $m - 1 \notin S$.

$$\Rightarrow \sum_{i=0}^{m-1} 2^i = 2^{m-1+1} = 2^m - 1$$

$$= 2^m - 1 + 2^m$$
$$= 2^{m+1} - 1$$

This is a contradiction, as $2^{m+1} - 1 \neq 2^m - 1 + 2^m$.

■

3.3 No contradiction

Example: Prove $\forall m \in \mathbb{N},\ \forall n \in \mathbb{Z}^+,\ \exists q,\ r \in \mathbb{N},\ (r < n) \wedge (m = qn + r)$. How do I prove this using PWO?

Proof. Let $m_0 \in \mathbb{N}$, $n_0 \in \mathbb{Z}^+$ (start with picking an arbitrary value).

- The moment I say this, m_0 and n_0 can't change – I don't know their exact value.

I want to prove the existence of something. The statement that I'm trying to prove is an existential, not a universal. PWO will give me the existential of a property.

Insight:

- I need a set of natural numbers, and I need to verify that it is non-empty.
- PWO is going to show me that one of $q,\ r$ exists. Because it is r that I care to make as small as possible, I'll use PWO to find r.

Define $R = \{r \in \mathbb{N} \mid \exists q \in \mathbb{N}, m_0 = qn_0 + r\}$.

In cases where $m_0 = 7,\ n_0 = 2,\ R = \{7, 5, 3, 1\}$, but we only chose what m_0 and n_0. Now, how do I know $R \neq \varnothing$?

Claim: $R \neq \varnothing$, because $m_0 \in R$. m_0 is always an element as if we choose $q = 0$, $m_0 = 0 \cdot n_0 + m_0$ (in the set builder notation, $q = 0,\ r = m_0$, hence $m_0 \in R$. From this point on, I can use PWO. I never said anything about m_0 being the lowest value in R.

By PWO, R contains a smallest element. In other words, we can assert:

$$\exists r_0 \in R,\ \forall r' \in \mathbb{N} < r_0,\ r' \notin R$$

I have proven that r_0 exists. Because $r_0 \in R$, $\exists q \in \mathbb{N}$, $m_0 = qn_0 + r_0$. That is the definition of the set R.

Now I need to show that $r_0 < n_0$. Is it? We know that the set contains no element smaller than r_0. This must be shown with a proof by contradiction. If $r_0 \geq n_0$, then we would have the following:

$$\begin{aligned} &= q_0 n_0 + n_0 - n_0 + r_0 \\ &= (q+1)n_0 + (r_0 - n_0) \end{aligned}$$

$$= q' n_0 + r_0'$$

As $r_0 \geq n_0$, $r' \in \mathbb{R}$ and $r' = r_0 - n_0 < r_0$, as $n_0 > 0$. This is a contradiction because r_0 was said to be the smallest element of R. ■

4 Structural induction

Recursion and induction. All these principles and techniques help us prove fact about the natural numbers, the positive integers, or the sets like them. If we can prove with one of them, we can prove it with the other.

When you do induction, these are the steps:

- Prove $P(0)$
- Prove $\forall k \in \mathbb{N}$, $P(k) \Rightarrow P(k+1)$
- Conclude $\forall n \in \mathbb{N}$, $P(n)$.

Well ordering is tied to an essential property of the structure of $\mathbb{N}$. The reason why induction works – that it is a reasonable approach to prove al things about natural numbers, has to do with the structure of natural numbers.

4.1 Defining sets of numbers recursively

If I had to formally write down the formal definition of $\mathbb{N}$, it might look like this:

- $0 \in \mathbb{N}$ (0 is a natural number, because that is the first one, we put in) – tied to proving the base case
- $\forall k \in \mathbb{N}$, $k+1 \in \mathbb{N}$ – tied to prove $\forall k \in \mathbb{N}$, $P(k) \Rightarrow P(k+1)$
- Nothing else goes in $\mathbb{N}$, which is tied to the conclusion: $\forall n \in \mathbb{N}$, $P(n)$. This asserts I can't sneak in any additional values or rules.

If I have a set-in mind, and my set contains 0, and whenever some number is in the set, one more than one number is in the set, what set am I thinking of? Maybe $\mathbb{N}$? Or maybe $\mathbb{Z}^{\geq -5}$? If I didn't put in the "nothing else," it wouldn't capture the natural numbers exactly.

So, what is structural induction about? If regular induction was just induction on the structure of the natural numbers, if I have other sets of numbers that I can define recursively, I can do induction on those numbers too? Yes. I can do induction on $\mathbb{Z}$. All I need is a recursive definition of the integers.

Defining the integers:

- $0 \in \mathbb{Z}$
- $\forall k \in \mathbb{Z},\ (k+1 \in \mathbb{Z}) \wedge (k-1 \in \mathbb{Z})$
- Nothing else (asserts that I can't sneak in additional values or rules)

These three points in the recursive definition capture the integers, exactly. Whatever integer you want, positive or negative, I can get to it following one of the points from above using any previous value.

Maybe not as elegant, but I do get every value. Want to do induction on $\mathbb{Z}$? Sure! Just do the following:

- Prove $P(0)$
- Prove $\forall k \in \mathbb{Z}$, $P(k) \Rightarrow P(k+1) \wedge P(k-1)$
- Conclude $\forall n \in \mathbb{Z}$, $P(n)$

This justifies some of the things we are doing. Prove something $\forall n \in \mathbb{N} \geq 3$? I can define $\mathbb{N}^{\geq 3}$ if I want to. What if I want to get every power of two?

- Start with 1 in the set.
- Given any power of two, how do I get a power of two? Just multiply it by two.
- You have a recursive definition of all powers of two.

If you do a proof that has this structure, then you can prove properties for all powers of 2.

4.2 Recursively defined sets that are not just numbers

The set below is just algebraic expressions.

Example: Define E as the smallest set, such that:

- Base elements: $x_i \in E,\ \forall i \in \mathbb{N}$
- Recursion: $\forall e_1,\ e_2 \in E$ (pick two variables), $(e_1 + e_2) \in E$ and $(e_1 \times e_2) \in E$
- *I don't need to write "nothing else" – by me saying "smallest set," that covers it.*

Example:

$$
\begin{aligned}
&(x_3 + (x_2 \times x_2)) \in E \\
&(((x_{27} + x_4) \cdot x_2) + (x_2 + (x_3 \cdot x_4)))\,,\ e_1 \\
&= ((x_{27} + x_4) \cdot x_2)\,,\ \ldots
\end{aligned}
$$

This set is defined recursively, so if I want to prove anything about that set, I can prove it using structural induction.

4.3 Using structural induction

Conjecture 1. *I want to prove that $\forall e \in E$ (for every expression in this set – see the previous section for how E is defined), the number of variables is always one more than the number of operators. In other words, numvar$(e) = 1 +$ numop(e).*

The predicate $P(e) : numvar(e) = 1 + \text{numop}(e)$

Proof outline:

1. Base case: Prove $P(x_i) \forall i \in \mathbb{N}$. In this expression, we only have one variable: x_i, and only no operators. So, we only really started with one variable.

2. Inductive step: I know every other possible element is $(e_1 + e_2)$ and $(e_1 \cdot e_2)$. Let $e_1,\ e_2 \in E$ (they are arbitrary).

    a. If there are any conditions in the recursion, assume the same kinds of conditions. Assume what I want to prove, as this is induction: Assume $P(e_1)$ and $P(e_2)$. *How do I know? I don't. I'm assuming it. What if it was true? That's the point of an implication proof. I'm engaging in hypothetical thinking.*

    b. I want to prove $P((e_1 + e_2))$ and $P((e_1 \cdot e_2))$.

To do the proof, I need to understand: e_1 and e_2: how many variables are there, and how many operators are there? Well,

$$\text{numvar}((e_1 + e_2)) = \text{numvar}(e_1) + \text{numvar}(e_2)$$

And

$$\text{numop}((e_1 + e_2)) = 1 + \text{numop}(e_1) + \text{numop}(e_2)$$

For the bottom one, we can see the additional $+$ symbol, hence why it is like that. We can continue from here.

Proof. Base case: Pick an expression with one variable. The number of operators is zero.

Inductive step: Assume all expressions with 1 to $k-1$ variables include have one less operator than variables. Prove it for k.

An expression has k variables. If we remove an arbitrary variable, we also must remove an operator. The moment we do that, we have an expression with $k-1$ variables, which has $k-2$ operators. So now we know. ■

4.4 Connection between structural induction and simple/-complete induction

Any kind of finite object that can be defined recursively (we have a recursive definition with base cases and a finite number of rules, and we can apply this rule a finite number of times to generate new objects), if you want to construct new elements, all you have were:

- The elements that were put in directly

- Then you could combine variables using the operators (check the previous example)
- And so on, as I apply more operators

Any kind of recursion will always have that feature. Some elements are given directly, and everything else is gained through some operator and some constructor.

The reason why structural induction and simple induction are facets of the same idea is that anything that can be proven with structural induction can be proven with simple induction. When proving with simple induction, we can look at how many times you have "constructed" something.

For example:

Elements $e \in E$ are either:

- Base (x_i), 0 constructor
- One constructor $(e_1 + e_2)$ or $(e_1 \cdot e_2)$ which I apply to previous, smaller elements.
  - They are smaller in that the number of constructors in e_1 and e_2 are always smaller than $(e_1 + e_2)$ and $e_1 \cdot e_2$.
- To prove something for all elements in that set, use structural induction.
- Or you could fake structural induction by doing induction on the number of constructions that generates the element. For example, prove it using one construction, two, three, and onwards.
  - This works better with complete induction. It's easier that way.

4.5 Formatting a proof by structural induction

Base case: Prove $P(n)$ for $n =$ the base case, which is likely mentioned in the recursive data structure.

Inductive hypothesis: Suppose $a_0,\ a_1,\ \ldots \in S$ and assume $P(a_0),\ P(a_1),\ \ldots$ hold.

Inductive step: You may assume the existence of variables or anything that was used to build $a_0,\ a_1 \ldots$. **You need to prove that for each way to construct new elements in S that it satisfies the predicate.** For example, if I can form a new element in S by taking $x,\ y \in S,\ x+y$, then I need to prove $P(x+y)$, whilst assuming that $\exists x_0,\ x_1,\ x = x_0 + x_1$.

4.6 PSI, PCI, and PWO

These three things are equivalent. Why?

- PSI: $(P(0) \wedge \forall k,\ P(k) \Rightarrow P(k+1)) \Rightarrow \forall n,\ P(n)$
- PCI: $(\forall k,\ (\forall j < k,\ P(j)) \Rightarrow P(k)) \Rightarrow \forall n,\ P(n)$
- PWO: $\forall S \subseteq \mathbb{N},\ S \neq \varnothing \Rightarrow S$ has a minimum element.

Firstly, we'll show that PWO $\Rightarrow$ PSI, PSI $\Rightarrow$ PCI, and PCI $\Rightarrow$ PWO. If I can do that, then I can conclude that PWO $\Leftrightarrow$ PCI $\Leftrightarrow$ PWO.

Proof that PWO $\Rightarrow$ PSI:

Proof. Assume PWO. WTP: PSI

- Assume $P(0) \wedge \forall k,\ P(k) \Rightarrow P(k+1)$.
- Prove $\forall n,\ P(n)$, without assuming PSI. Attempt to get a contradiction.

For a contradiction, assume $\exists n,\ \neg P(n)$. (This doesn't invalidate what we assumed before that)

Let $S = \{n \in \mathbb{N} \mid \neg P(n)\} \neq \varnothing$.

So, by PWO, S has a minimum element n_0.

- $n_0 \in S$, so $\neg P(n_0)$.
- $\Rightarrow n_0 > 0$ (Because $P(0)$ is true, from our assumption.)
- $\Rightarrow n_0 - 1 \in \mathbb{N}$, so $n_0 - 1 \notin S$, so $P(n_0 - 1)$ is true.
- Then, $P(n_0 - 1) \Rightarrow P(n_0)$, as this is true by assumption for $k = n_0 - 1$, so $P(n_0)$. Contradiction.

■

A contradiction occurs when $P(n)$ is true and false at the same time. I proved PSI itself using well-ordering. This uses the same structure, the same template when using PWO to prove things that could normally be proven with simple induction.

Proof that PSI $\Rightarrow$ PCI:

Proof. Assume PSI:

$$(P(0) \wedge \forall k,\ P(k) \Rightarrow P(k+1)) \Rightarrow \forall n,\ P(n)$$

Assume

$$\forall k,\ (\forall j < k,\ P(j)) \Rightarrow P(k)$$

WTP: $\forall n,\ P(n)$. This is required to complete the entire proof of this question.

Let $P'(n) = (\forall k \leq n,\ P(k))$

Prove $\forall n,\ P'(n)$ by PSI. Proving this helps us prove $P(n)$.

Base case: $P'(0)$. Then, we need to show that $\forall k \leq 0,\ P(k)$, which simply requires us to show that $P(0)$ is true. Firstly, $\forall j < 0,\ P(j)$ is vacuously true, so then we know that $P(0)$ is true.

Inductive step: Let $m \in \mathbb{N}$ and assume $P'(m)$. We want to prove that $P'(m+1)$. By our assumption,

$$(\forall j < m+1,\ P(m+1)) \Rightarrow P(m+1)$$

Also,

$$P'(m) = (\forall k \leq m,\ P(m)) = (\forall k < m+1,\ P(k))$$

Because we know that this is true, then we can conclude that $P(m+1)$ is true. Because we assumed $P'(m)$, which is $\forall k \leq m,\ P(k)$, and that $P(m+1)$ holds, we can conclude

$$\forall k \leq m+1,\ P(k)$$

and thus, we can conclude $P'(m+1)$.

By the principle of simple induction, we can conclude $\forall n,\ P'(n)$, meaning $\forall n,\ \forall k \leq n,\ P(k)$. This also means that $\forall n,\ P(n)$ holds, which completes the proof. ■

5 Iterative Algorithm analysis

How do we reason about algorithms?

- We can talk about complexity, in terms of running time or space in terms of memory. For example, how many elements in an array do I need to carry out this calculation?
  - Runtime functions, worst-case, bounds (upper/lower bounds, $\mathcal{O}$, Ω, Θ)
- Algorithm correctness

What is meant by algorithm correctness? Intuitively, we think of algorithm that takes in inputs or produces outputs, or sometimes for some algorithms, its behavior. Correctness is when we want to prove that when we have valid inputs, we get the expected outputs.

For all valid inputs, the output/behavior is as expected.

5.1 Preconditions

How do we specify for an input to be valid? By using **preconditions.** Preconditions is any predicate that holds before an algorithm executes or runs.

In general, any predicate. This means I can say any condition I want in my precondition. A precondition limiting an input to be one value is a valid precondition, but that isn't very practical. So, in practice, you want your preconditions to not be limited. The more conditions on your input, the more restrictive is it.

When you have the option for preconditions, say as little as you want, and don't introduce any assumptions that aren't necessary. Make preconditions as weak as possible. Weak, meaning as few conditions as possible.

5.2 Postconditions

To specify expected behavior or output. It is any predicate that holds after an algorithm executes, and we want postconditions to be strong.

5.3 Proving correctness of an algorithm

Preconditions: $x \in \mathbb{R}$, $y \in \mathbb{N}$, $(x \neq 0 \vee y \neq 0)$

Post condition: returns x^y, or 1 if $x = y = 0$. The loop postcondition is $z = x^y$ and $m = y$.

How do we prove correctness? It's always relative to the preconditions and post conditions. What we need to prove:

For all inputs that satisfy the preconditions, if the algorithm is run, then two things happen:

function POW(*x, y*)
    $z = 1$
    $m = 0$ ▷ Loop precondition: $x \in \mathbb{R}, y \in \mathbb{N}, z = 1, m = 0$)
    while $m < y$: **do** ▷ Loop invariant: $z = x^m \wedge m \leq y$
        $z = m \cdot x$
        $m = m + 1$
    end while
    return z
end function

1. Make sure the algorithm eventually stops (no infinite while loop, termination)
2. When it stops, the postcondition holds (partial correctness)

If we want to prove that precondition and execution $\Rightarrow$ termination and partial correctness of the algorithm, for this algorithm:

- Before the loop: $x \in \mathbb{R} \wedge y \in \mathbb{N} \wedge m = 0$
- After the loop: $\overbrace{z = x^y}^{\text{partial correctness}} \wedge \overbrace{m = y}^{\text{stopping}}$.

Loop invariants:

- **Any** predicate that holds just before the loop condition is evaluated after every iteration of the loop, and before the first one.

The loop invariant is evaluated **right before the while loop condition is evaluated.**

We want our loop invariant to be:

- An actual invariant, and that we can prove that it is invariant. For example, $z = x^y$ is *NOT* a loop invariant (because it's not true every time I would evaluate the loop invariant).
- We want it to be useful: we want it to imply the loop postcondition.

For example, $m \geq 0$. This is an invariant – it is true as $m \geq 0$ is true at every iteration of the algorithm, but it is not very useful.

Question: How can we tell what is true about z and m ? That is the most difficult – coming up with an algorithm given an algorithm? How are the variables changing?

The index means **how many iterations that have already happened before the one we're observing starts.**

Iteration	0	1	...	k
m_k	0	1		k
z_k	1	x		x^k

If I want to formally prove this, it means induction – this is whenever a $\cdots$ appears in the table.

Once I've seen a pattern, can I write down a statement that captures that pattern? And how do we prove this? In my proof, I will have the added complication that I want to talk about the values of z before and after each iteration. How do I talk about z during one point and a different point of the loop?

5.4 Notation in loop invariants

z_k means the variable at a very particular point during the execution of the code. $z_k =$ value of variable z at the end of k complete iterations.

5.4.1 When do we check loop invariants

The point at the execution where we take this snapshot is the same point, we claim our loop invariant is going to be true. *Just about the point where you'll look at the loop condition.*

5.5 What's the loop invariant?

What is true about z, m, x, y? This is the loop invariant:

$$z = x^m$$

Do not use the variable k in the loop invariant as k (no. of iterations) is not a variable in the code. Do not use it as a subscript as well, as they restrict the loop invariant to a specific number of iterations.

You can return to using subscripts when you are proving loop invariants.

Your loop invariant can build up to the postcondition. It is always more than a specific condition must hold which makes it easier to guarantee correctness.

5.6 Proving the loop correctness

With the loop invariant, what do we want to do next:

WTP:

1. The loop invariant is invariant
2. (Loop precondition holds and **the loop runs and stops**) $\Rightarrow$ loop postconditions are true
3. Loop preconditions and loop runs $\Rightarrow$ loop stops

If I can prove all 3, then I've proven that the algorithm is correct.

5.6.1 Showing that precondition holds, and loop runs and stops implies postconditions are true

Proof. Assume loop precondition holds and the loop runs and stops (we're at the point where the loop just stopped). Prove that the loop postconditions are true.

What do we know at this point?

1. If the loop has stopped, it must be because the loop condition evaluated to false. This means $m \geq y$ (inverse of loop condition)
2. $m \leq y$, from loop invariant
3. From the loop invariant, $z = x^m$. Can we conclude that $z = x^y$ and $m = y$?

How do we show that $m = y$? We know this because we can see that m increments by 1 per loop, but that is not captured in the loop invariant. In fact, if the value of the variable changes in more unpredictable ways, then we can't automatically conclude cases like $m = y$ – it could overshoot. Simply add $m \leq y$ to the loop invariant.

So, if we know $m \geq y \wedge m \leq y$, $m = y$. Then:

$$m = y$$

And

$$z = x^m = x^y$$

We have proven the loop postcondition. ■

5.6.2 Proving the loop invariant

The loop invariant is $z = x^m \wedge m \leq y$. We need to show this is true after every iteration. This means we need to prove LI_k by induction on k. $LI_k =$ the loop invariant where I put the subscript k on every variable. $LI_k = z_k = x^{m_k} \wedge m_k \leq y$ (Don't put the subscript k on variables that don't change).

However, k is finite as the loop stops. Yet, we're proving $\forall k \in \mathbb{N}$. What is the best way to do so?

- NOTE: When $k >$ no. iterations, we're using the convention that $z_k,\ m_k$ remain unchanged (look at $z,\ m$). After the loop ends, the value of k keeps increasing, but $z_k,\ m_k$ will remain the same after the loop ends.

Proof. Base case. WTP: $LI_0 = z_0 = x^{m_0} \wedge m_0 \leq y$.

- $z_0 = 1$, by the loop precondition
- $x^0 = x^{m_0} = 1$.
- $m_0 = 0 \leq y$.

Inductive step: Let $k \in \mathbb{N}$. Assume LI_k. Assume there is one more iteration. Then I need to show LI_{k+1}: $z_{k+1} = x^{m_{k+1}}$ and $m_{k+1} \leq y$.

Showing $z_{k+1} = x^{m_{k+1}}$:

$$z_{k+1} = z_k \cdot x \overset{\text{by } LI_k}{=} x^{m_k} \cdot x = x^{m_k+1}$$

Showing $m_{k+1} \leq y$ assuming the loop condition is true when the invariant is checked:

$$\begin{aligned} m_k < y \overset{\text{by loop condition}}{\quad} &\Leftrightarrow m_k \leq y-1 \\ &\wedge m_{k+1} = m_k + 1 \\ &\Rightarrow m_k + 1 \leq y \Leftrightarrow m_k \leq y-1 \\ \text{loop condition} &\Rightarrow m_k < y \Leftrightarrow m_k \leq y-1 \\ &m_k + 1 \leq y \Leftrightarrow m_k \leq y-1 \\ &m_{k+1} = m_k + 1 \leq y \end{aligned}$$

$$m_k \in \mathbb{N} \Rightarrow m_{k+1} = m_k + 1 \in \mathbb{N}$$

By induction, LI holds. ■

5.6.3 Proving the loop stops

The technique we're going to use here:

- Technique: find a loop "measure". It's an expression E with two properties:
  - $E_k \in \mathbb{N},\ \forall k \in \mathbb{N}$ (an expression that involves the program variables, with the property that this value is a **natural number** property – if there's an edge case that makes it -1, just add 1 to it)
  - $E_{k+1} < E_k,\ (\forall k$ with at least $k+1$ iterations$)$

If we can find an expression like this, it gives us a bound on how many iterations there are. This is a quantity that is always an integer (but nonnegative), keeps getting smaller, but can't become smaller than some element (by PWO). This means:

By PWO, $E_0,\ E_1,\ E_2,\ \ldots$ has a smallest element (it is always *strictly* decreasing).

For this element, $E = y - m$. This is a bound on how many iterations remain. The initial value for E should be larger or equal to the number of iterations the program will have (as an upper bound).

To prove that a loop stops:

- Find a loop measure E_k.
- Show that it is strictly decreasing: $E_{k+1} < E_k$
- By PWO, $E_{\ldots}$ has a finite number of elements, which is enough to conclude that the loop stops.

6 Iterative algorithm correctness

You can do the three in any order as long as you do all three, but there may be dependencies. If you come up with a loop invariant, you need to be extra sure that it works if you don'

- We came up with a loop invariant (LI) to describe something that's true for every iteration for the loop, including before the loop starts and right as the loop ends (loop condition not met only for the first time).
  - We proved this by induction on the number of iterations.
  - The loop invariant is the hardest and requires the most creativity.

- Termination: Precondition and the loop executes $\Rightarrow$ the loop stops.
  - The loop invariant can be used
  - Set up a loop measure: a strictly decreasing sequence of $\mathbb{N}$.
- Partial correctness: assume the precondition and the algorithm runs and stops $\Rightarrow$ postcondition. Partial correctness applies for the entire algorithm, not the loop.
  - Use the loop invariant

What we really care about is overall correctness. The overall conclusion from all of this is:

- Precondition and execution $\Rightarrow$ postcondition

When we're focused on reaching the postcondition, we don't worry about whether the loop will stop. We just say, what if the loop stops? Then what can I say? How do I use what I know to reach the postcondition?

Suppose we have a counting loop:

```
function CountingLoop()
    for x = a, a+1, ..., b:   do                    ▷ LI: TBA
        the loop body...
    end for
end function
```

Showing that the loop stops is easy, as the algorithm will only go through a limited number of things. How will I show this loop is correct?

6.1 Approach 1: Converting a for loop to a while loop

So, turn the loop into the following:

Apart from scoping issues only present in programming languages, this while loop is equivalent to the for loop. There are some advantages when converting the for loop to a while loop:

- x exists outside the loop
- LI_0 true when $x = a$

```
function ForWhileTemplate()
    x = a                                                    ▷ LI: TBA
    while x ≤ b do
        the loop body...
        x = x + 1
    end while
end function
```

- LI true when $x = b+1$ (even though for the for loop, there's no such point)

This simplifies the loop invariant so that we can use the two conventions.

About the loop stopping: If we're saying that we have a for loop, we'll treat it as a while loop. Our loop measure will simply be $b+1-x$. We need to make sure that this value is always non-negative. However, **it is perfectly fine to conclude that a for loop will always stop.** *State it, though, as the loop stopping is an important component of correctness.*

6.2 Approach 2: Unwrap the for loop

The for loop is equivalent to:

```
function ForLoopExample()
    x = a                                                    ▷ (LI_{x=a}: TBA)
    ...body...
    x = a + 1                                                ▷ (LI_{x=a+1}: TBA)
    ...body...
    ⋮
    x = b                                                    ▷ (LI_{x=b}: TBA)
    ...body...
    x = b + 1                                                ▷ (LI: TBA)
end function
```

The behavior of the algorithm, if it is captured by the loop invariant, mimics exactly what happens by the while loop and is the pattern that we implicitly understand to be the for loop. Thus, we can meaningfully talk about x in the loop invariant before the loop executes.

What about the last loop invariant? Instead of using the subscript k to talk about the values of my variables after some amount if iterations, an alternate convention is to **use the program variable x directly to talk about which iteration you are currently working on. Use x to count iterations instead of k . This means:**

- Base case: Prove $LI_{x=a}$
- Inductive step: Let $k \in \{a,\ a+1\ ,\ \ldots,\ b\}$.
  - Assume $LI_{x=k}$ is true. Prove that after the body executes, $I_{x=k+1}$ is true ($I_{x=k}$ means the value of x after these many iterations.

6.3 A for loop example

Say we have this loop:

- for x in L:

We can easily turn it into:

- for i in range(len(L)):
  - Body of the loop, with $L[i]$ in place of x.

The selection sort algorithm.

---

function SELECTSORT(A)

▷ Precondition: TBA

```
    for i = 0, 1, ..., len(A) − 2 do
        m = i
        for j = i + 1, i + 2, ..., len(A) − 1 do
            if A[j] < A[m] then
                A[j], A[m] = A[m], A[j]
            end if
        end for
    end for
end function
```

Here's the complication: Nested for loops. For nested loops, a loop invariant will still exist for the inner loop, but the inner loop invariant will be a bit more complicated: **set an inner loop invariant for an arbitrary iteration of an outer loop. Prove it first.**

A good inner loop invariant would be: ILI: m is the index of the smallest element in $A[I:J]$.

Then, set up an outer loop invariant: Example: $A[0:i]$ is sorted and all elements in $A[0:i]$ is $\leq$ to $A[i:len(A)]$. To prove this, use the inner loop invariant to support the outer loop invariant.

Postcondition: A is sorted. We may amend the method to use while loops:

```
function SELECTSORT(A)
    i = 0
    while i ≤ len(A) − 2 do
        m = i
        j = i + 1
        while j < len(A) do
            if A[j] < A[m] then
                A[j], A[m] = A[m], A[j]
            end if
            j = j + 1
        end while
        m = m + 1
    end while
end function
```

The inner loop invariants are:

- $j \leq len(A)$
- m is the index of the smallest element in $A[I : J]$

The inner loop postcondition is that m is the index of the smallest element in $A[I : len(A)]$

6.4 Selection sort (full, correct)

Slice notation works exactly how it does in Python.

The loop has:

```
function SELECTSORT(A)
    for i = 0, 1, ..., len(A) − 2 do
        m = i
        for j = i + 1, i + 2, ..., len(A) − 1 do
            if A[j] < A[m] then
                m = j
            end if
            A[j], A[m] = A[m], A[j]
        end for
    end for
end function
```

The preconditions: A is a list and elements are comparable.

Convert the for loop to a while loop: the value of i when looking at the loop invariant is the value that has been set up for the next iteration. What I have done is always up to and including $i-1$ when I state my loop invariant. At some arbitrary iteration of the outer loop, what have we accomplished?

Outer loop invariant: $A[0:i]$ is sorted, meaning $A[0] \leq A[i] \leq \ldots \leq A[i-1]$, and these are the smallest elements overall in the entire list (all elements in $A[0:i] \leq$ all elements in $A[i:]$):

$$\forall x \in A[0:i],\ \forall y \in A\left[i : len(A)\right],\ x \leq y$$

This statement is vacuously true if either side is empty.

For the inner loop, we said what is true at each step. In some iteration of the outer loop, assume that j is some value.

Inner loop invariant: m is the index of the smallest element in $A[i:j]$.

Method postcondition: A has been sorted.

The proof? We want to prove that this **entire** algorithm is correct. There are three things to show:

- The loop invariant holds
- Preconditions, and the loop runs and stops implies partial correctness
- The loop eventually stops

The loop invariants are not directly part of the statement of correctness; they are a tool to prove partial correctness.

You don't need to express a postcondition for an inner loop – justification in later steps is acceptable. Can every nested loop be written in this style? Maybe not. It's never wrong to set up an intermediate precondition and postcondition for a loop. It may not be always necessary.

The easiest thing to prove is that the loop eventually stops. The loop eventually stops because it is a for loop. This justification is sufficient.

Then, **partial correctness**. When proving partial correctness, we say:

- Assume we have an input A satisfying the preconditions
- The algorithm runs

- The algorithm stops

What do we know?

- The equivalent while loop condition failed
  - Then, we know the loop iteration variable $i = len(A) - 1$
- The outer loop invariant is true: $A[0 : i] = A[0 :]$ is sorted
- Also, from the outer loop invariant, $A[0 : i] \leq A[i : len(A)]$. We know that $i = len(A) - 1$, so then we can assert $A[0 : len(A) - 1] \leq A[len(A) - 1 : len(A)]$, but $A[len(A) - 1 : len(A)] = A[len(A) - 1]$, so we can conclude that A is fully sorted as $A[: i]$ is sorted. Thus, the postcondition holds.

We only relied on the outer loop invariant for partial correctness. What we know is true is the outer loop invariant. The inner loop invariant is helpful for showing the outer loop invariant. Most of the time, you shouldn't rely on the inner loop invariant to prove the correctness of the entire method.

I didn't prove the outer loop invariant yet, so I need to prove it.

6.5 What is going on in the inner loop invariant proof?

The inner loop invariant: m is the index of the smallest element in $A[i : j]$. Assume that I'm in some arbitrary iteration of the outer loop. I know from the outer loop invariant is that $A[0 : i]$ is sorted. I want to prove the inner loop invariant.

Proof. **Base case.** WTP: m_0 is the index of the smallest element in $A[i : j_0]$. Well, $j_0 = i + 1$, so $A[i : j_0] = A[i]$, which is the smallest element.

Inductive step. Show that at some arbitrary point in our loop, if m_k is the smallest element in $A[i : j_k]$ and the loop iterates at least $k + 1$ times, then m_{k+1} is the smallest element in $A[i : j_{k+1}]$.

- If $A[j_k] > A[m_k]$, do nothing. Still, in $A[i : j_{k+1}]$, $A[m]$ is the smallest element.
- Otherwise, $A[m_{k+1}]$ will be the smallest element, as $A[m_{k+1}] = A[j_k]$, as $A[j_k]$ is now the smallest element in $A[i : j_{k+1}]$.

■

6.6 The outer loop invariant

If I don't need the inner loop invariant to prove the outer loop invariant, it means the inner loop isn't doing anything useful. The outer loop invariant is as follows:

Outer loop invariant: $A[0:i]$ is sorted, meaning $A[0] \leq A[i] \leq \ldots \leq A[i-1]$, and these are the smallest elements overall in the entire list (all elements in $A[0:i] \leq$ all elements in $A[i:]$):

Proof. **Base case.** $A[0:i_0] = []$, which is sorted. And $A[0:i_0] = [\text{EMPTY}] \leq A[i_0 : len(A)]$ is vacuously true.

Inductive step. In A_k, (the list itself changes during the algorithm, thus we subscript A). The inner loop invariant tells me that $A_k[m] \leq A_k[i_k : len(A)]$. So A_{k+1} is constructed by swapping index m and i_k. Then, $A[: i_k + 1] = A[: i_{k+1}]$ is sorted. ■

Think of the step that brings you from the current iteration to the next.

6.7 Do loop measures need to reach zero?

The loop measure does not **have to reach zero** when the loop stops. A good loop measure can still be equal to some positive integer by time the loop stops. The only two things that matter is that:

- LM is always non-negative (feel free to add 9999 to the LM)
- It gets strictly smaller after every iteration

7 Correctness of recursive algorithms

An overview for recursive correctness:

We have a

- Base case
- Recursive case

Which is all dependent on the input size.

On every recursive algorithm proof: WTP: By complete induction on input size n:

> **For every input** of size n that satisfies the preconditions, ALGO(input) stops **and** postcondition holds.

The structure for our proof goes as follows:

- Base case: determined by the method
- Inductive step: follow the recursive structure of the code.

When you first attempted to write a recursive function, you realized that every recursive call's input size gets smaller. As the input size can always be represented as a natural number, that can't go on infinitely. There's a finite chain of calls until I reach a base case. That's why recursion works, and the same structure with induction applies. Hence, that's why recursive correctness is easier than iterative correctness.

7.1 Binary search

```
function RecBinSearch(x, A, b, e)
    if e == b + 1 then
        if x ≤ A[b] then
            return b
        else
            return e
        end if
    else
        m = ⌊(b + e)/2⌋
        if x ≤ A[m − 1] then
            return RecBinSearch(x, A, b, m)
        else
            return RecBinSearch(x, A, m, e)
        end if
    end if
end function
```

Preconditions:

- A is a list of comparable elements
- x is comparable
- $b,\ e \in \mathbb{N}$ and $0 \leq b < e \leq len(A)$ (prevents empty lists)
- $A[b:e]$ is sorted

Postcondition: $RecBinSearch(x,\ A,\ b,\ e)$ returns $p \in \mathbb{N}$ such that:

1. $b \leq p \leq e$
2. $p > b \Rightarrow x > A[p-1]$
    a. Element before if it is there is less than x
3. $p < e \Rightarrow x \leq A[p]$
    a. Element at p, if it is there, is $\geq x$.

What does this method do? *Return the index of x if it is in the list. Otherwise, return the index where it should be inserted.* Then, everything before p will be less than $A[p]$ and everything after p will be greater or equal than $A[p]$.

7.2 Proving recursive correctness

The key insight in doing recursive correctness, is that we will use induction on the size of the input. If we have a well-written recursive algorithm, that is the way the algorithm is structured. It is already written such that it recursively calls inputs that are smaller. The structure of the proof will mirror the structure of the elements.

What we need to show: for all inputs $(x,\ A,\ b,\ e)$ of size $n = e - b$ that satisfies the preconditions, the call $RecBinSearch(x,\ A,\ b,\ e)$ stops (won't result in a stack overflow) and returns an index p that satisfies the postcondition.

Complete induction is needed. It will always work, and it has the advantage, and not all recursive algorithms call itself with a size of $n-1$.

Proof. By complete induction on n.

Base case. $n = 1$. Let $(x,\ A,\ b,\ e)$ be an arbitrary input that satisfies the precondition with $e - b = n = 1$.

When I run the algorithm, if $e - b = 1$ then $e = b + 1$, so the if condition of the algorithm is true, and the if-block executes.

- This means $RecBinSearch(x,\ A,\ b,\ e)$ stops, as there is no loop or recursive call anymore in this branch (if-block).
- Returns b if $x \leq A[b]$. If that was the case:
    - Postconditions 1, 2, and 3 hold as $p = b < e$ and $x \leq A[p = b]$.
- Returns $p = e = b + 1$ if $x > A[b]$. If that was the case:

– Postconditions 1, 2, and 3 hold as $b+1 > b$ and $x > A[p-1 = b]$.

Inductive step. Let $k > 1$. Assume for all inputs $(x',\ A',\ b',\ e')$ of size $< k$ that satisfy the precondition, $RecBinSearch\,(x',\ A',\ b',\ e')$ returns index p that satisfies the postconditions.

Let $(x,\ A,\ b,\ e)$ be an input of size $e-b=k$ and assume that the preconditions are satisfied. Consider $RecBinSearch(x,\ A,\ b,\ e)$. We note that $e-b=k>1 \Rightarrow e>b+1$, so the algorithm executes the else branch.

$m=\left\lfloor \frac{b+e}{2} \right\rfloor$ and makes one recursive call depending on whether $x \leq A[m-1]$ or $x > A[m-1]$. There are two cases to check:

Case 1: $x \leq A[m-1]$. Then, we call $RecBinSearch\left(x,\ A,\ b,\ m=\left\lfloor \frac{b+e}{2} \right\rfloor\right)$. Firstly, is the input size smaller than what I started with? $e > b+1 \Rightarrow b < \left\lfloor \frac{b+e}{2} \right\rfloor < e$ (mini-proof: $b < \left\lfloor \frac{2b+2}{2} \right\rfloor = \lfloor b+1 \rfloor < e$) – because this is true, then we know that the recursive call is an input size of $m-b<e-b$, so by inductive hypothesis, it stops, and returns p such that:

- $b \leq p \leq m$
- $p > b \Rightarrow x > A[p-1]$
- $p < m \Rightarrow x \leq A[p]$

The conditions we want to prove are:

1. $b \leq p \leq e$
    a. Proof of it: $m < e$, so this is satisfied.
2. $p > b \Rightarrow x > A[p-1]$
    a. True by IH
3. $p < e \Rightarrow x \leq A[p]$
    a. If $p < m$, then $x \leq A[p]$ by IH.
    b. If $p = m$, then $x \overset{\text{by case 1}}{\leq} A[m-1] \leq \overset{\text{by precondition}}{A[m=p]}$

Case 2: This will look very similar to case 1. ■

7.3 Design based on correctness

Maybe proving isn't the thing you'll do all the time. You'll spend more of your time writing code than proving correctness. Here's an example of a design for **merge sort:**

Preconditions:

- A is a list of comparable elements
- More. . . ?

If we don't have enough preconditions, we can always add more as we go.

Postconditions:

- A has been rearranged that $A[0] \leq A[i] \leq \ldots \leq A\,[len(A)-1]$

The high level for Mergesort:

- Split the list in half
- Sort them separately and recursively
- Merge them back

---

function MERGESORT(A)
    if len$(A) > 1$: **then** ▷ (otherwise A is already sorted)
        $F = A\left[: \left\lfloor \frac{\text{len}(A)}{2} \right\rfloor\right]$ ▷ (Split A into two halves)
        $S = A\left[\left\lfloor \frac{\text{len}(A)}{2} \right\rfloor :\right]$ ▷ (Recursively sort each half)
        MergeSort(F) ▷ Combine and merge them back. F MUST be smaller than A. But len$(A) > 1 \Rightarrow 0 < \left\lfloor \frac{\text{len}(A)}{2} \right\rfloor <$ len(A), so we're safe
        MergeSort(S)
        Merge$(A,\ F,\ S)$ ▷ (Replace the indices A with F and S. Assume this method works)
    end if
end function

But how does the Merge method work?

Preconditions for Merge:

- F and S are sorted
- $\text{len}(A) = \text{len}(F) + \text{len}(S)$

Postconditions for Merge:

- A contains everything that was in F, S, but sorted

We have F as a list, and S as a list. We have A as a list.

Loop invariants:

- $F[i:]$ and $S[j:]$ have already been merged into A
- $F[i]$ and $S[j]$ are the possible candidates to be merged next.

---

function MERGE(*A, F, S*)
    $i = 0,\ j = 0$
    while $i < \text{len}(F)$ or $j < \text{len}(S)$ **do**
        if $F[i] \leq S[j]$ **then**
            $A[i+j] = F[i]$
            $i = i+1$
        else
            $A[i+j] = j+1$
            $j = j+1$
        end if
    end while
    if $i = \text{len}(F)$ **then**
        $A[i+j:] = S[j:]$
    else
        $A[i+j:] = F[i:]$
    end if
end function

Loop invariants:

- $i,\ k \in \mathbb{N}$ and $0 \leq i \leq \text{len}(F)$ and $0 \leq j \leq \text{len}(S)$
- $F[:i]$ and $S[:j]$ has been merged into $A[:i+j]$ in fully sorted order
- All elements in $A[:i+j]$ are $\leq$ than all elements in $F[i:]$ and $S[j:]$.

8 Algorithm Complexity and Running Time

When we measure running time, what are we measuring? When we say running, it is a function of the input size. It measures steps. A step is not defined precisely; they are approximations. The idea of a step is any block of code that takes constant time to execute regardless of the input size.

In practice, we work with the worst-case running time. We know, from an algorithm, what information do we get:

- We have an input size
- I want a function using that

The problem with that, is think of linear search. Running time isn't always going to be steady if we fix the input size. If we know the size of the input, there's still a lot of possible inputs with the same input size (such as every permutation of a list with a fixed number of elements). But a function gives one value for each input.

The worst case uses the largest possible running time given input size.

The bounds come in, as in practice, there's often no exact expression for runtime. Even if there were, it wouldn't be that helpful. What we're counting is arbitrary anyways. We know that it will be in a constant factor of the elementary operations performed by the computer. But we don't know what the constant means.

Hence, we look for bounds.

- Upper bound: $WC(n) \in \mathcal{O}(U(n))$
- Lower bound: $WC(n) \in \Omega(L(n))$
- Tight bound: $WC(n) \in \Theta(B(n))$

An upper bound on the worst case is something larger than the largest of running times. A lower bound on the worst case doesn't have to be smaller than the running time for every input – it just must be less than or equal than the slowest. I want it as close to the worst case.

8.1 To Solve Recurrence Relations

Look at this (pre: $n \in \mathbb{N}$, post: returns $n!$):

```
function FACT(n)
    if n ≤ 1 then
        return 1
    end if
    return n · Fact(n − 1)
end function
```

The recurrence for $T(n)$ (note that T means the runtime function):

$$T(n) = \begin{cases} 1 & n = 0,\ 1 \\ 1 + T(n-1) & t > 1 \end{cases}$$

IMPORTANT: You need to get this step – the recurrence – correctly. Once we have the recurrence, we'll work purely with the recurrence and completely forgot about the algorithm. If you get it wrong, the rest of your work will give you a completely useless answer.

What's a closed form of T?

- You could start plugging in values and see if a pattern emerges
- Doesn't work for non-trivial recurrence relations
- We need a different approach – there's still going to be some pattern matching.

8.2 Repeated substitution method

Instead of trying to spot a pattern in concrete numbers, can we spot a pattern in the general value of T?

- Assume we start with a large value of n.
- What is $T(n)$ if n is large? Well, it's not the base case, so it's $T(n) = 1 + \overbrace{T(n-1)}^{\text{I want to get rid of this}}$. Not closed form, though.
- But n is large, so $n-1$ is large. Large enough that $T(n-1)$ doesn't hit the base case right away:
- $T(n) = 1 + (1 + T(n-2)) = 2 + \overbrace{T(n-2)}^{\text{still not gone}}$
- Do this again:

- $T(n) = 2 + (1 + T(n-3)) = 3 + T(n-3)$ (come on...)
- Can we see a pattern? What is the process? Imagine that I do this a bunch of times. After k substitutions, what can I expect $T(n)$ to be?
- $T(n) = k + T(n-k)$

$$\begin{aligned} T(n) &= 1 + T(n-1) \\ &= 1 + (1 + T(n-2)) = 2 + T(n-2) \\ &= 2 + (1 + T(n-3)) = 3 + T(n-3) \\ &\vdots \\ &= k + T(n-k) \end{aligned}$$

Note: this is just a guess! Why? There's a $\vdots$ present in the expansion, which is a sign for you to do **induction.**

I still haven't gotten rid of T on the right-hand side. Of course, $T(n-k)$ will disappear when the argument to T is 0 or 1 – meaning when $n - k \leq 1$. If this is true, what values must k have for this to be true?

$$k : k \geq n - 1$$

If $k \geq n-1$, what if we substitute $k = n-1$? (What is a value of k at the **base case**?)

$$\begin{aligned} T(n) &= n - 1 + T(n - n + 1) \\ &= n - 1 + T(1) = n \end{aligned}$$

We now have high confidence that $T(n) = n$. We have high confidence in our guess, but we just came up with a *guess*.

Conclusion. We expect $T(n) = n$. (Maybe not for $T(0)$, so not necessarily exact). Prove it using induction on $n \geq 1$. That proof will be very straightforward. It doesn't matter how you came up with your guess; all that matters is how you come up with it. Hence, you don't need to be precise when making your guess. When making the guess in binary search, we're just going to take the $\mathcal{O}$ of it. Precision is a bit of an illusion – you don't have to be precise.

```
function RBS(x, A, b, e)
    if e == b + 1 then
        if x ≤ A[b] then
            return b
        else
            return e
        end if
    else
        m = ⌊(b+e)/z⌋
        if x ≤ A[m − 1] then
            return RBS(x, A, b, m)
        else
            return RBS(x, A, m, e)
        end if
    end if
end function
```

8.3 Recursive Binary Search

Recall recursive binary search (assume $A[b:e]$ is sorted):

The insight: Recursive expression for $T(n)$:

$$T(n) = \begin{cases} 1 & n = 1 \\ \overbrace{1}^{\text{outside rec calls}} + \max\left(\overbrace{T\left(\left\lceil \frac{n}{2} \right\rceil\right)}^{\text{first recursive call}}, \overbrace{T\left(\left\lfloor \frac{n}{2} \right\rfloor\right)}^{\text{second recursive call}}\right) & n > 1 \end{cases}$$

8.3.1 Solving The Recurrence Relation

If n is large, then:

$$T(n) = 1 + \max\left(T\left(\left\lceil \frac{n}{2} \right\rceil\right), T\left(\left\lfloor \frac{n}{2} \right\rfloor\right)\right)$$

$$T\left(\left\lceil \frac{n}{2} \right\rceil\right) = 1 + \max\left(T\left(\left\lceil \frac{\left\lceil \frac{n}{2} \right\rceil}{2} \right\rceil\right), T\left(\left\lfloor \frac{\left\lceil \frac{n}{2} \right\rceil}{2} \right\rfloor\right)\right)$$

Here's a workaround: We can take advantage of **this fact (note: there are some assumptions using this, so it doesn't always work):**

$$\max\left(T\left(\left\lceil\frac{n}{2}\right\rceil\right), T\left(\left\lfloor\frac{n}{2}\right\rfloor\right)\right) \leq T\left(\left\lceil\frac{n}{2}\right\rceil\right)$$

That lets me get rid of using the max function... but this only works if T is **non-decreasing**.

We've got two options:

1. We prove that T is non-decreasing. Use your MAT137 skills: $\forall n,\ \forall m \leq n,\ T(m) \leq T(n)$
2. **Simplify!** Ignore floors, ceilings, ± 1 s, and so on. **These can be safely ignored, most of the time, with no ill effect.**
    a. You can only simplify in the recursive call: you can't simplify the 1 on the left in $1 + T_{\text{amended}}\left(\frac{n}{2}\right)$.

This is a guess, so don't worry about being exact.

$$T_{\text{amended}}(n) = \begin{cases} 1 & n = 1 \\ 1 + T_{\text{amended}}\left(\frac{n}{2}\right) & n > 1 \end{cases}$$

The answer I get from doing repeated substitution here is the starting point. If I can prove the recurrence relation here, it means I didn't simplify too far.

To prevent clutter, let $T' = T_{\text{amended}}$. Let's find a **pattern**:

$$\begin{aligned} T'(n) &= 1 + T'\left(\frac{n}{2}\right) \\ &= 1 + \left(1 + T'\left(\frac{n}{4}\right)\right) = 2 + T'\left(\frac{n}{4}\right) \\ &= 2 + \left(1 + T'\left(\frac{n}{8}\right)\right) = 3 + T'\left(\frac{n}{8}\right) \\ &\vdots \\ &= k + T'\left(\frac{n}{2^k}\right) \\ &= k + 1 + T'\left(\frac{n}{2^{k+1}}\right) \text{ looks like an I.S.} \end{aligned}$$

Remember that this is just a **guess**. When do I reach a base case? When the argument to T' is equal to 1.

$$\begin{aligned}\frac{n}{2^k} &= 1 \text{?solve for } k\\ n &= 2^k\\ \Leftrightarrow k &= \log_2(n)\end{aligned}$$

Then, I get:

$$\begin{aligned}T'(n) &= \log_2(n) + T'\left(\frac{n}{2^{\log_2(n)}}\right)\\ \approx \log_2(n) + T'\left(\frac{n}{n}\right) &= \log_2(n) + T'(1)\\ &\approx \log_2(n) + 1\end{aligned}$$

Still a guess!! Not exactly equal, but it should be close. I don't know how far I've gotten off with my implications, and I hope I'm not more than a constant factor off. All of this gives us a guess, and now we're going to prove it.

Here's the true recurrence again: WTP

$$T(n) \in \Theta(\log_2(n))$$

Start with proving $T(n) \in \Omega(\log_2(n))$. Here's some rough work:

Imagine the inductive step. We're trying to prove that $\exists c \in \mathbb{R}^{\geq 0},\ T(n) \geq c \cdot \log_2(n)$, the inductive hypothesis:

$$\begin{aligned}T(n) &= 1 + \max\left(T\left(\left\lceil\frac{n}{2}\right\rceil\right),\ T(\left\lfloor\frac{n}{2}\right\rfloor\right)\\ &\geq 1 + \max\left(\overset{\text{by the IH}}{c \cdot \log_2\left(\left\lceil\frac{n}{2}\right\rceil\right),\ c \cdot \log_2\left(\left\lfloor\frac{n}{2}\right\rfloor\right)}\right)\\ &= 1 + \max\left(c \cdot \log_2\left(\left\lceil\frac{n}{2}\right\rceil\right)\right) \text{ as log is non-decreasing}\\ \log_2\left(\left\lceil\frac{n}{2}\right\rceil\right) \geq \log_2\left(\frac{n}{2}\right) &\geq \log_2\left(\left\lfloor\frac{n}{2}\right\rfloor\right)\\ \cdots &\geq 1 + c \cdot \log_2\left(\frac{n}{2}\right)\\ &= 1 + c \cdot \log_2(n) - c \geq c \cdot \log(n)\end{aligned}$$

Solve for c and do the proof with that value of c. Oh, and $c \cdot \log_2(n) \geq c$.

8.4 Proving the recurrence, for real

$$T(n) = \begin{cases} 1 & n = 1 \\ 1 + \max\left(T\left(\left\lceil \frac{n}{2} \right\rceil\right), T\left(\left\lfloor \frac{n}{2} \right\rfloor\right)\right) & n > 1 \end{cases}$$

Prove that $T(n) \in \Theta(\lg(n))$.

The Ω proof:

Proof that $\forall n \geq B,\ T(n) \geq c \cdot \lg(n)$

Proof. Choose $B = 1$ (which will always be the base case). Choose $c = 1$. Base case. $T(1) = 1$. Is $T(1) \geq \lg(1) = 0$? YES.

Note: we chose the constants at the end, and I would've erased them afterwards. That is, c . For $B = 1$, I just guessed. Feel free to cross them out once you choose them to be friendly numbers (a.k.a. 1).

Let $n > B$. Assume:

IH: $T(B) \geq c \cdot \lg(B),\ \ldots,\ T(n-1) \geq c \cdot \lg(n-1)$

Then $T(n) = 1 + \max\left(T\left(\left\lceil \frac{n}{2} \right\rceil\right), T\left(\left\lfloor \frac{n}{2} \right\rfloor\right)\right)$

$$\begin{aligned} &\geq 1 + max(c \cdot \lg\left(\left\lceil \frac{n}{2} \right\rceil\right), c \cdot \lg\left(\left\lfloor \frac{n}{2} \right\rfloor\right) \\ &= 1 + c \cdot \lg\left(\left\lceil \frac{n}{2} \right\rceil\right) \end{aligned}$$

(Note that $\lg\left(\left\lceil \frac{n}{2} \right\rceil\right)$ is non-decreasing).

We'll use the fact that $\left\lceil \frac{n}{2} \right\rceil \geq \frac{n}{2}$, so:

$$\begin{aligned} 1 + c \cdot \lg\left(\left\lceil \frac{n}{2} \right\rceil\right) \geq 1 + c \cdot \lg\left(\frac{n}{2}\right) &= 1 + c\left(\lg n - \lg 2\right) \\ &= 1 + c\lg(n) - c \\ &1 + \lg(n) - 1 \\ &= \lg(n) \end{aligned}$$

We can rewrite our rough work and just get erase all c s.

■

$\mathcal{O}$ proof: Proof that $\forall n \geq TBA,\ T(n) \leq c \cdot \lg(n)$.

Proof. Inductive step. Let $n > 4 = 4$. Assume $T(4) \leq 4 \cdot \lg(c) \ldots T(n-1) \leq c \cdot \lg(n-1)$. Then, $T(n) = 1 + \max\left(T\left(\left\lceil \frac{n}{2} \right\rceil\right),\ T\left(\left\lfloor \frac{n}{2} \right\rfloor\right)\right)$. Which is:

$$
\begin{aligned}
&\leq 1 + \max\left(c \cdot \lg\left(\left\lceil \frac{n}{2} \right\rceil\right),\ c \cdot \lg\left(\left\lfloor \frac{n}{2} \right\rfloor\right)\right) \\
&= 1 + c \cdot \lg\left(\left\lceil \frac{n}{2} \right\rceil\right) \quad \text{as log is non-decreasing} \\
&\leq 1 + c \cdot \lg\left(\frac{n+1}{2}\right) \\
&= 1 + c \cdot \lg(n+1) - c
\end{aligned}
$$

I want $\leq c \cdot \lg(n)$. Issue? I have $\lg(n+1)$. Stuck? Adjust the proof to $T(n) \leq \lg(n-1)$.

Inductive hypothesis is like before. I'm going to skip rewriting everything, so I'm going to continue where I left off where I rewrote everything: (HERE I START PLACING THE $+2$ s)

$$
\begin{aligned}
T(n) &\leq 1 + \lg\left(\left\lceil \frac{n}{2} \right\rceil - 1\right) \quad +2 \\
&\leq 1 + \lg\left(\frac{n+1}{2} - 1\right) \quad +2 \\
&= 1 + \lg\left(\frac{n+1-2}{2}\right) \quad +2 \\
&= 1 + \lg\left(\frac{n-1}{2}\right) \quad +2 \\
&= 1 + \lg(n-1) - 1 \quad +2 \\
&= \lg(n-1) \quad +2
\end{aligned}
$$

Am I done? We still have the base case.

$$T(1) = 1 \overset{?}{\leq} \lg(1-1)$$

Issue? $\lg(0)$ is undefined. This doesn't work. Maybe start the comparison with a larger n?

$$T(2) = 1 + \max\left(T(1),\ T(1)\right) = 1 + 1 = 2$$

So, $T(2) = 2 \leq \lg(2-1) + 2$.

What about $T(3)$?

$$T(3) = 3 \overset{?}{\leq} \lg(3-1)+2 = \lg(2)+2$$

If we keep going, it works out fine (I'm not doing a proof by induction).

Now, I want to make sure in my inductive step, I want to make sure that $\frac{n}{2} \geq 2$ in my inductive step.

And now, in my IH, I'll set $n \geq 4$ so that $\left\lfloor \frac{n}{2} \right\rfloor \geq 2$. ■

Now, the Θ. What does that give us? Overall, $T(n) \in \Omega(\lg(n))$ and $T(n) \in \mathcal{O}(\lg(n-1)+2)$. The two things look different, but do they really look different? We know that in $\mathcal{O}$ and Ω expressions, we are allowed to drop all lower terms. $\lg(n-1) \leq \lg(n)$, so $T(n) \in \mathcal{O}(\lg(n))$.

We can therefore conclude that $T(n) \in \Theta(\lg(n))$.

TL: DR – take advantage of $\mathcal{O}$. For $\mathcal{O}$, can we prove it for something a bit smaller? It still captures what is larger.

8.5 Analyzing Merge Sort

---

function MERGESORT(A)
  if $n > 1$ **then**
    $F = A\left[: \left\lfloor \frac{\text{len}(A)}{2} \right\rfloor \right]$ ▷ $\Theta(n)$ due to Python's slicing
    $S = A\left[\left\lfloor \frac{\text{len}(A)}{2} \right\rfloor : \right]$ ▷ $\Theta(n)$ due to Python's slicing
    MergeSort(F) ▷ $T\left(\left\lfloor \frac{n}{2} \right\rfloor\right)$
    MergeSort(S) ▷ $T\left(\left\lceil \frac{n}{2} \right\rceil\right)$
    Merge(F, S, A) ▷ $\Theta(n)$ due to linear number of iterations
  end if
end function

If we let $T(n)$ be the worst-case running time on an input of size len(A).

So:

$$T(n) = \begin{cases} 1 & n \leq 1 \\ T\left(\left\lceil \frac{n}{2} \right\rceil\right) + T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + \overset{\text{simple enough}}{n} & n \geq 2 \end{cases}$$

If we do repeated substitution, we first simplify the recurrence. How simple can it get? All we need to do is get rid of the floors and ceilings.

$$T'(n) = \begin{cases} 1 & n \leq 1 \\ 2T\left(\frac{n}{2}\right) + n & n \geq 2 \end{cases}$$

Then let's say that n is some large value.

$$\begin{aligned} T'(n) &= 2T\left(\frac{n}{2}\right) + n \\ &= 2\left(2T\left(\frac{n}{4}\right) + \frac{n}{2}\right) + n = 4T'\left(\frac{n}{4}\right) + 2n \\ &= 4\left(2T\left(\frac{n}{8}\right) + \frac{n}{4}\right) + 2n = 8T\left(\frac{n}{8}\right) + 3n \end{aligned}$$

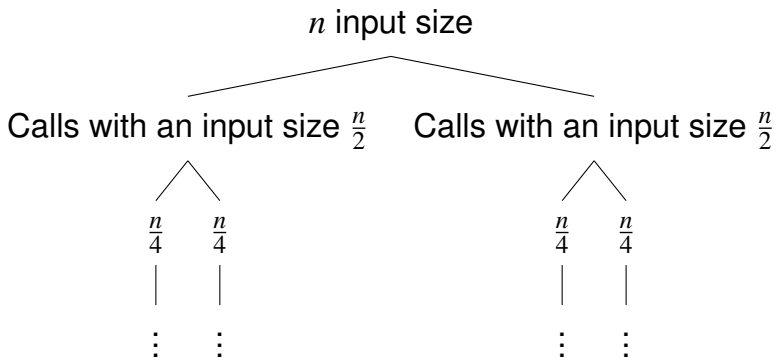
After k substitutions:

$$= 2^k T'\left(\frac{n}{2^k}\right) + k \cdot n$$

The base is reached when $\frac{n}{2^k} = 1 \Leftrightarrow 2^k = n \Leftrightarrow k = \lg(n)$:

$$\begin{aligned} &nT'\left(\frac{n}{n}\right) + \lg(n) \cdot n \\ &= n\left(\lg(n) + 1\right) \end{aligned}$$

There's our guess. Once we have the guess, we must go back and use that to prove the Θ bound. Here's another faster technique: use a recursion tree. Draw a tree of all the recursive calls but imagine that you have all of them.



Then, figure out, at each level of the tree: how many nodes?

- 1 node has n work per node, which is n total

- 2 nodes have $\frac{n}{2}$ work per node, which is n total
- 4 nodes have $\frac{n}{4}$ work per node, which is n total
- And so on... What's the total work done across all the recursive calls? $\lg(n)$.

8.6 About Trees

Mergesort ends up having this simplified recurrence:

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

And we ended up seeing a recursion tree. But what is going on? I want to track what happens during the entire recursion and get an overall point of view on the total work done. We'll draw a tree with one node for each recursive call. What we'll track is the input size.

Recursion tree:

- One node for each recursive call
- Write input size
- On the side, track the work done outside recursive calls (no. nodes, work done per node outside recursive calls, assuming the recursive call is constant time – anything part of a T term)

For the tree above:

Table 2: For the Mergesort tree

No. of nodes	Work per node	Total work
1	n	n
2	$\frac{n}{2}$	n
4	$\frac{n}{4}$	n
8	$\frac{n}{8}$	n
n	1	n

We're not trying to account for the work done by the recursive calls; we'll see them elsewhere in the trees. We are only tallying up the work done outside the recursive calls, otherwise you will end up double counting. The only thing missing here, is when does this bottom out?

This bottoms out with a whole bunch of nodes of size 1. These are the last calls I'm going to make, but how many of those will I have?

Now, I'll sum up the table: the total work is $n \cdot$ tree height. There are k levels $\Rightarrow$ input size $\frac{n}{2^k}$ and when is $\frac{n}{2^k} = 1$? When $k = \lg(n)$. If I plug that back into my total, I get $T(n) = n \cdot \lg(n)$.

It seems a lot more eyeballing, but just like repeated substitution, the answer this gives us is still just a **guess.** This whole technique is rough work, and the purpose of this is to give you a good guess, and then you need to plug things into the actual recurrence to see if you can prove your guess.

9 Master Theorem For Recurrences

A divide and conquer algorithm is a recursive algorithm that has three phases. Sometimes they exist, sometimes they don't:

- **Divide**: split the input. The sizes of the new inputs are always a *fraction* of the existing size.
- **Recurse:** Solve **each** sub-problem recursively
- **Conquer:** Combine the solutions

We can see them for Mergesort very easily:

- Divide: split the list
- Recurse: recursively sort the halves
- Conquer: merge them together

Algorithms that follow this paradigm can have a general recurrence for their running times. It has a recurrence that follows the same pattern.

For any divide and conquer algorithm like that, the runtime satisfies

$$T(n) = \begin{cases} 1 & n \leq B \\ aT\left(\frac{n}{b}\right) + \underbrace{n^k}_{\text{work of divide and conquer steps}} & n > B \end{cases}$$

Where $k \in \mathbb{R}^{\geq 0}$. Also, $b \in \mathbb{R}$, $b > 1$ (fraction of input size one recursive call in), and $a \in \mathbb{Z}^+$ (number of recursive calls per call).

Issue is that Mergesort doesn't split exactly, so note that we're using simplified recurrence. This recurrence would correspond to something like $T(n) = a_1 T_1\left(\left\lceil \frac{n}{b} \right\rceil\right) + a_2 T\left(\left\lfloor \frac{n}{b} \right\rfloor\right) + n^k$.

If you have an algorithm that fits this pattern, then the solution is:

$$T(n) \in \begin{cases} \Theta\left(n^k\right) & k > \log_b(a) \Leftrightarrow b^k > a \\ \Theta\left(n^k \lg(n)\right) & k = \log_b(a) \Leftrightarrow b^k = a \\ \Theta\left(n^{\log_b(a)}\right) & k < \log_b(a) \Leftrightarrow b^k < a \end{cases}$$

No repeated substitution or proof required. Plug things in and you've got your answer.

9.1 Some Examples

For binary search:

$$T(n) = T\left(\frac{n}{2}\right) + 1$$

Then, $\overset{\text{recursive calls per call}}{a = 1}$, $b = 2$, $k = 0$. Then:

$$T(n) \in \Theta\left(n^k \lg(n)\right) = \Theta\left(\lg(n)\right)$$

What about Mergesort? We had:

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right) + n \\ a = 2,\ b &= 2,\ k = 1 \\ \Rightarrow T(n) \in \Theta\left(n^k \lg(n)\right) &= \Theta\left(n \lg(n)\right) \end{aligned}$$

9.2 Intuition

If we have a recurrence that we can apply the master theorem to:

If we start with size n, our tree looks like:

- n
  - $\frac{n}{b} \cdots \frac{n}{b}$, a nodes of that
    - $\frac{n}{b^2} \cdots \frac{n}{b^2}$, a^2 nodes of that

In terms of the work done per node:

No. of nodes	Work per node	Total work
1	n^k	n^k
a	$\frac{n^k}{b^k}$	$n^k \left(\frac{a}{b^k}\right)$
a^2	$\frac{n^k}{\left(b^2\right)^k}$	$n^k \left(\frac{a}{b^k}\right)^2$

Figure 9B: Table of a divide and conquer

The height? $\frac{n}{b^i} \leq 1 \Leftrightarrow i = \log_b(n)$. The number of nodes on the last level is $\approx a^{\log_b(n)}$. We know that $a^{\log_b(n)} = n^{\log_b(a)}$, always (this is an identity).

9.3 To come up with Divide and Conquer Algorithms

How do we carry out basic arithmetic operations with large values? Most of the basic operations are straightforward. Multiplication gets a bit difficult. We're working with two large binary numbers of the same length:

$$X = (x_{n-1},\ x_{n-2},\ \ldots,\ x_1,\ x_0)_2$$
$$Y = (y_{n-1},\ y_{n-2},\ \ldots,\ y_1,\ y_0)_2$$

Example values: $X = 22 = (10110)_2,\ Y = 7 = (00111)_2$.

The grade school's algorithm is $\Theta\left(n^2\right)$, assuming every operation performed on a digit is constant time.

9.4 The Divide and conquer algorithm

How are we going to split the bits up? If we don't know what to do, split it in half. Split X and Y into two halves (assume that n is even. If not, add extra 0s):

$$X_1 = x_{n-1} \ldots x_{\frac{n}{2}},\ X_0 = x_{\frac{n}{2}-1} \ldots x_0$$
$$Y = y_{n-1} \ldots y_{\frac{n}{2}},\ Y_0 = y_{\frac{n}{2}-1} \ldots y_0$$

In my example:

$$
\begin{aligned}
X &= \underbrace{010}_{X_1}110 \\
Y &= 000111 \\
X_1 \cdot 2^3 + X_0 &= 010000 + 11 = 010110 = X \\
X &= 2^{\frac{n}{2}} X_1 + X_0 \\
Y &= 2^{\frac{n}{2}} Y_1 + Y_0 \\
\Rightarrow XY &= \left(2^{\frac{n}{2}} X_1 + X_0\right)\left(2^{\frac{n}{2}} Y_1 + Y_0\right) \\
&= 2^n X_1 Y_1 + 2^{\frac{n}{2}} X_1 Y_0 + 2^{\frac{n}{2}} X_0 Y_1 + X_0 Y_0
\end{aligned}
$$

There. That's the recursive algorithm. Let's write it down: Note that n is the length of X and Y.

---

function MULT(*X, Y, n*)
  if $n == 1$ **then**
    return $X_0 \times Y_0$ ▷ 1-bit product; bult-in
  else
    Split X and Y into $X_1,\ X_0,\ Y_1,\ Y_0$
    $P_1 = Mult\left(X_1,\ Y_1,\ \left\lceil \frac{n}{2} \right\rceil\right)$ ▷ Ceiling to prevent issues
    $P_2 = Mult\left(X_1,\ Y_0,\ \left\lceil \frac{n}{2} \right\rceil\right)$
    $P_3 = Mult\left(X_0,\ Y_1,\ \left\lceil \frac{n}{2} \right\rceil\right)$
    $P_4 = Mult\left(X_0,\ Y_0,\ \left\lceil \frac{n}{2} \right\rceil\right)$
    return $2^{2\left\lceil \frac{n}{2} \right\rceil} P_1 + 2^{\left\lceil \frac{n}{2} \right\rceil} P_2 + 2^{\left\lceil \frac{n}{2} \right\rceil} P_3 + P_4$
  end if
end function

What is the runtime?

$$
T(n) = \begin{cases} 1 & n = 1 \\ 4T\left(\left\lceil \frac{n}{2} \right\rceil\right) + \underset{\text{split, shift, add}}{n} & n > 1 \end{cases}
$$

Using the master theorem: $a = 4,\ b = 2,\ k = 1$. The master theorem tells us: $b^k = 2 < a$. So, this tells us that

$$
T(n) \in \Theta\left(n^{\log_b(a)}\right) = \Theta\left(n^2\right)
$$

Still too slow? What is $(X_1 + X_0)(Y_1 + Y_0)$? It is: Maybe:

function MULT(*X, Y, n*)
  if $n == 1$ **then**
    return $X_0 \times Y_0$ ▷ 1-bit product; bult-in
  else
    Split X and Y into $X_1,\ X_0,\ Y_1,\ Y_0$
    $P_1 = Mult\left(X_1,\ Y_1,\ \left\lceil \frac{n}{2} \right\rceil\right)$ ▷ Ceiling to prevent issues
    $P_2 = Mult\left(X_1 + X_0,\ Y_1 + Y_0,\ \left\lceil \frac{n}{2} \right\rceil\right)$
    $P_4 = Mult\left(X_0,\ Y_0, \left\lceil \frac{n}{2} \right\rceil\right)$
    return $2^{2\left\lceil \frac{n}{2} \right\rceil} P_1 + 2^{\left\lceil \frac{n}{2} \right\rceil}(P_2 - P_1 - P_4) + P_4$
  end if
end function

The recurrence is:

$$T'(n) = \begin{cases} 1 & n = 1 \\ 3T\left(\frac{n}{2}\right) + n & n > 1 \end{cases}$$

Using the master theorem, the runtime is:

$$T'(n) \in \Theta\left(n^{\log_2(3)}\right)$$

This is Karatsuba's algorithm.

10 Formal Language Theory

In this section of the course, we use the word language in a very specific way. We mean something very formal. For us:

Definition 10.1 (Language)**.** A language is any set of strings. It can be empty, finite, or infinite.

Examples: $\varnothing$, $\{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$, $\{+\}$.

Definition 10.2 (String)**.** Any finite or empty sequence of characters.

Because strings are sequences, we could write them in the way we write sequences $(\mathsf{a}, \mathsf{b}, \mathsf{a}, \mathsf{a})$ or use commas: `"abaa"`, but for me, I'll just write them without the commas. The issue is that if I don't use commas, what happens if I end up typing a Unicode character that looks like two letters squished together? Just don't allow them.

Definition 10.3 (Alphabet)**.** Any non-empty, finite set of characters or symbols with the requirement: all characters are distinguishable and unique. *Technically speaking, if we want to allow ambiguous characters, then we better just use UTF-8 encoding codes, and when we write our strings, we need to make them unambiguous.*

I don't want the alphabet I have to have any sort of ambiguity. No matter what sequence of characters I write, there should be no ambiguity in the sequence of characters I write.

How do we represent an empty string? Without parentheses, commas, or quotation marks? By **convention**, we're going to pick a fixed symbol ε `varepsilon` to represent the empty string (NOT ε). Some textbooks use Λ, but we're not using that.

To end with a question: What can we do with languages? When we write a program, we're generating a file – I'm just writing one long string. Every piece of information? We write it down. That's how we represent and process information. That's the motivation – what we learn about properties of languages will apply to everything we do.

What's the difference between $\varnothing$ and $\{\varepsilon\}$?

10.1 Language Operators

Notation. For a fixed but arbitrary alphabet Σ (example: set of everything in UTF-8):

- $\Sigma^* = \{\text{all strings over } \Sigma\}$
  - This is a language. If all the keys on my keyboard were an alphabet, this set would contain all possible combinations of me mashing my keyboard
- $\Sigma^n = \{\text{all strings of length } n\}$

For example, $\{\text{a, b}\}^*$ is every possible string I can write with the letters a, and b. This set is infinite.

On the other hand, $\{\text{a, b}\}^2 = \{\text{aa, ab, ba, bb}\}$. As a note, $|s| = \text{len}(s)$, where s is a string.

Set operations: for all $L_1,\ L_2 \subseteq \Sigma^*$ (things we can do with languages)

- Union: $L_1 \cup L_2$
- Intersection: $L_1 \cap L_2$
- Complement: $\overline{L} = L^c = \Sigma^* \setminus L$

- Concatenation:
  - With strings: $\forall s_1,\ s_2 \in \Sigma^*$ (Let $s_1,\ s_2$ be arbitrary here)
    - $s_1 \cdot s_2$ (Or $s_1 s_2$) contains all the characters of s_1 followed by s_2. In Python, that's like adding two strings.
    - For example: `ab`$\cdot$ `bb`$=$ `abb`
    - And $\varepsilon\cdot$ `ba`$=$ `ba`
    - **Concatenation is not always commutative!** (Not even the magnitude as sets can't have duplicates)
  - We can extend this to languages:
    - $L_1 \cdot L_2 = \{s \cdot t\ : s \in L_1 \wedge t \in L_2\}$
    - For example: $\{\text{a},\ \text{bb}\} \cdot \{\text{aa},\ \text{b}\} =$ `{aaa, ab, bbaa, bbb}`
    - The concatenation of two languages can give us strings that aren't in either.
    - It's **not quite** the cartesian product.
- Exponentiation:
  - $\forall s \in \Sigma^*,\ \forall n \in \mathbb{N},\ s^n = \underbrace{s \cdot s \cdots s}_{n \text{ times}} = \begin{cases} \varepsilon & n = 0 \\ s \cdot s^{n-1} & n > 0 \end{cases}$
  - $\forall L \subseteq \Sigma^*,\ \forall n \in \mathbb{N},\ L^n = \underbrace{L \cdots L}_{n} = \begin{cases} \{\varepsilon\} & n = 0 \\ L \cdot L^{n-1} & n > 0 \end{cases}$
    - $L^0 = \{\varnothing\}$
- Kleene star (or just star):
  - $\forall L \subseteq \Sigma^*,\ L^* = \bigcup_{k=0}^{\infty} L^k = L^0 \cup L^1 \cup L^2 \cup \cdots$, a.k.a.
    - $\varepsilon \in L^*$, always.
  - L^* = every string of the form $s_1 \cdot s_2 \cdots s_k$ for some $k \in \mathbb{N}$ and $s_1,\ \ldots,\ s_k \in L$.
  - Here's an example:
    - $\{\text{a},\ \text{bb}\}^* = \underbrace{\{\varepsilon\}}_{k=0} \cup \underbrace{\{\text{a},\ \text{bb}\}}_{k=1} \cup \underbrace{\{\text{aa},\ \text{abb},\ \text{bba},\ \text{bbbb}\}}_{k=2} \cdots$

Now, with the set of the empty string and the empty set:

- $\{\varepsilon\} \cdot \{\text{a},\ \text{aa}\} = \{\text{a},\ \ \text{aa}\}$
- $\varnothing \cdot \{\text{a},\ \text{aa}\} = \varnothing$

  - Oops, I can't pick anything from the empty set. Look at the set builder notation to see what is going on.

- $\varnothing^* = \{\varepsilon\}$, by definition. I *can* pick nothing at all from the empty set.

By the way, Σ^* is the same star as the Kleene star – look at what we said Σ^* to be.

10.2 Regular Expressions

This is the most basic forms of regular expressions we can come up with. The idea here, is, we want to come up with some formal way of describing some types of languages.

Below we have a **recursive definition.** Recall **structural induction.** For any alphabet Σ:

The **basic** regular expressions are:

- $\varnothing$
- ε
- a, for each $a \in \Sigma$

Recursively, for all regular expressions $R_1,\ R_2$:

- Star: $\left(R_1^{\circledast}\right)$
- Concatenation: $(R_1 \cdot R_2)$
- Union: $(R_1 + R_2)$

For example, if $\Sigma = \{\mathrm{a},\ \mathrm{b}\}$, the following are regular expressions:

- a (This is a regular expression! Sometimes, the star will be put in a circle.)
- $(\mathrm{a}^{\circledast})$
- b
- $\left(\mathrm{b}^{\circledast}\right)$
- $\left((\mathrm{a}^{\circledast}) + \left(\mathrm{b}^{\circledast}\right)\right)$

Each regular expression R denotes a language:

$$\mathscr{L}(R) \subseteq \Sigma^*$$

This means:

$$\begin{aligned}
\mathscr{L}(\varnothing) &= \varnothing \\
\mathscr{L}(\varepsilon) &= \{\varepsilon\} \\
\mathscr{L}(a) &= \{a\} \\
\mathscr{L}\left(R_1^{\circledast}\right) &= \mathscr{L}\left(R_1^{\circledast}\right) \\
\mathscr{L}\left(R_1 \cdot R_2\right) &= \mathscr{L}\left(R_1\right) \cdot \mathscr{L}\left(R_2\right) \\
\mathscr{L}\left(R_1 + R_2\right) &= \mathscr{L}\left(R_1\right) \cup \mathscr{L}\left(R_2\right)
\end{aligned}$$

Say I start with $\mathscr{L}\left(\mathsf{a}^{\circledast} + \mathsf{b}^{\circledast}\right)$ Do I mean $\mathscr{L}\left(\mathsf{a} + \mathsf{b}^{\circledast}\right)$?

Firstly:

$$\begin{aligned}
&\mathscr{L}\left(a^{\circledast} + b^{\circledast}\right) = \mathscr{L}\left(a^{\circledast}\right) \cup \mathscr{L}\left(b^{\circledast}\right) \\
&= \{\varepsilon,\ \mathsf{a},\ \mathsf{aa},\ \mathsf{aaa},\ \ldots\} \cup \{\varepsilon,\ \mathsf{b},\ \mathsf{bb},\ \mathsf{bbb},\ \ldots\}
\end{aligned}$$

Which is `/(a*)|(b*)/`

In contrast:

$$\mathscr{L}\left((\mathsf{a} + \mathsf{b})^{\circledast}\right) = \mathscr{L}\left(\mathsf{a} + \mathsf{b}\right)^{\circledast} = \{\mathsf{a},\ \mathsf{b}\}^*$$

Note that $\mathscr{L}\left(\mathsf{a}^{\circledast} \cdot \mathsf{b}^{\circledast}\right) =$ anything that matches the regex `/a*b*/`.

Can we write a regular expression for any strings of 0s and 1s with at least one 1? A.K.A. `/[01]*1[01]*/`. How do we write this?

$$(0+1)^{\circledast} \cdot 1 \cdot (0+1)^{\circledast}$$

This is not the only correct answer. If I start looking at the characters from the start, at some point I will hit the first 1. The only thing I can have in front, is, maybe, an arbitrary number of zeroes. Then, any combination of 0s and 1s. Meaning:

$$0^{\circledast} \cdot 1 \cdot (0+1)^{\circledast}$$

What does it mean for regular expressions to be equivalent?

10.3 Equivalence of Regular Expressions

Prove that $0^{\circledast}1(0+1)^{\circledast} \equiv (0+1)^{\circledast}10^{\circledast}$

- First show $\mathscr{L}\left(0^{\circledast}1(0+1)^{\circledast}\right) \subseteq \mathscr{L}\left((0+1)^{\circledast}10^{\circledast}\right)$
- Second, show $\mathscr{L}\left((0+1)^{\circledast}10^{\circledast}\right) \subseteq \mathscr{L}\left(0^{\circledast}1(0+1)\right)$

I'll only show the first one: $\mathscr{L}\left(0^{\circledast}1(0+1)^{\circledast}\right) \subseteq \mathscr{L}\left((0+1)^{\circledast}10^{\circledast}\right)$

Proof.
- Let $s \in \mathscr{L}\left(0^{\circledast}1(0+1)^{\circledast}\right)$ (be an arbitrary string in the language of this regular expression).
- Then, $s = u \cdot v \cdot w$
  - Any string in this regex can be written with the three components: $u \in \mathscr{L}\left(0^{\circledast}\right),\ v \in \mathscr{L}(1),\ w \in \mathscr{L}\left((0+1)^{\circledast}\right)$
    - $u = 0^m$, for some $m \in \mathbb{N}$ (including $m = 0$).
    - $v = 1$, always.
    - w is any string in $\{0,\ 1\}^*$.
- I want to show that I can write s to match the pattern of the RHS.
- Then, $s = 0^m \cdot 1 \cdot$ anything
  - Either w has no 1, or it has at least one 1.
  - **Case 1.** w has no 1. Then, $w = 0^n$, for some $n \in \mathbb{N}$.
    - Then, $s = 0^m \cdot 1 \cdot 0^n$
    - $\Rightarrow \underbrace{\mathscr{L}\left((0+1)^{\circledast}\right)}_{0^m \in \uparrow}, \underbrace{\mathscr{L}(1)}_{1 \in \uparrow}, \underbrace{\mathscr{L}\left(0^{\circledast}\right)}_{0^n \in \uparrow}$
      - Note: below the underbrace, is a string that is a subset of that language.
  - **Case 2.** w contains a *last (final)* 1 (does not mean the last char of w is 1).
    - Then, $w = w_1 \cdot 1 \cdot 0^n$, for some $n \in \mathbb{N}$, where $w_1 \in \mathscr{L}\left((0+1)^{\circledast}\right)$
    - If I can write w in this way, then:
    - $s = 0^m \cdot 1 \cdot w_1 \cdot 1 \cdot 0^n$
    - $\Rightarrow \underbrace{\mathscr{L}\left((0+1)^{\circledast}\right)}_{0^m \cdot 1 \cdot w_1 \in \uparrow}, \underbrace{\mathscr{L}(1)}_{1 \in \uparrow}, \underbrace{\mathscr{L}\left(0^{\circledast}\right)}_{0^n \in \uparrow}$
    - No additional justification needed.

■

The second direction is left as an exercise to the reader. Well, I am the reader, so I'm going to do it anyway.

$$\mathscr{L}\left((0+1)^{\circledast}10^{\circledast}\right) \subseteq \mathscr{L}\left(0^{\circledast}1(0+1)^{\circledast}\right)$$

Proof.

- Let $s \in \mathscr{L}\left((0+1)^{\circledast}10^{\circledast}\right)$.
- Then, $s = u \cdot v \cdot w$, as any string in this regex can be written with these components:
  - $u \in \mathscr{L}\left((0+1)^{\circledast}\right)$
  - $v \in \mathscr{L}(1)$
  - $w \in \mathscr{L}\left(0^{\circledast}\right)$
- About u:
  - **Case 1.** u has no 1 at all. Then, $u \in \mathscr{L}\left(0^{\circledast}\right)$
    - So, we can represent u as 0^m for any m
    - $v = 1$ and $w \in \mathscr{L}\left(0^{\circledast}\right) \subseteq \mathscr{L}\left((0+1)^{\circledast}\right)$
    - $\Rightarrow s = 0^m \cdot 1 \cdot 0^n$
    - $\Rightarrow \mathscr{L}\left(0^{\circledast}\right),\ \mathscr{L}(1),\ \mathscr{L}\left((0+1)^{\circledast}\right)$
  - **Case 2.** u has at least one 1 character. Then, $u \in \mathscr{L}\left(0^{\circledast}10^{\circledast}\right)$, which is $\subseteq \mathscr{L}\left(0^{\circledast}1(0+1)^{\circledast}\right)$. I mean, we can rewrite u as $0^m \cdot 1 \cdot u_1$, where $u_1 \in \mathscr{L}\left((0+1)^{\circledast}\right)$
    - Also, $v \in \mathscr{L}\left((0+1)^{\circledast}\right)$ and $w \in \mathscr{L}\left((0+1)^{\circledast}\right)$
    - So, we have the string $0^m \cdot 1 \cdot u_1 \cdot v \cdot w$
    - $u_1 \cdot v \cdot w \in \mathscr{L}\left(\left((0+1)^{\circledast}\right)^3\right) \subseteq \mathscr{L}\left((0+1)^{\circledast}\right)$
    - $\Rightarrow \mathscr{L}\left(0^{\circledast}\right),\ \mathscr{L}(1),\ \mathscr{L}\left((0+1)^{\circledast}\right)$

■

10.4 The Easiest Regular Expression Question

$$R \neq \varnothing,\ RS \equiv RT \Rightarrow S \equiv T$$

The easiest counter example involves using *s and. Look for $S \equiv T$ but if you tag on an extra component, it makes it equivalent. Pick S and T generate different strings but adding R smooths it all out.

Let $R = (a+\varepsilon)$, $S = a^{\circledast}$, $T = (aa)^{\circledast}$. Then:

$$RS = (a+\varepsilon)a^{\circledast} = a^{\circledast},\ RT = (a+\varepsilon)(aa)^{\circledast} = a^{\circledast}$$

But $a^{\circledast}$ and $(aa)^{\circledast}$ are not equivalent.

11 Deterministic Finite-State Automata (DFSA/DFA)

Regular expressions describe a pattern all in one chunk, but they give no information about how we can tell if a string fits a pattern.

Your DFA must accept every possible string it could get.

A DFA tells us how we keep track of things, more like an actual procedure/program. It doesn't give the pattern directly, as if it were an executable script (maybe I am wrong?).

For example, A vending machine.

- Everything costs $0.30
- It only takes nickels ($0.05), dimes ($0.10), and quarters ($0.25).
- It does not take any change.
- You only get one thing at a time, and each time your credit disappears after it gives you the item from it.

If we want this to be an illustration for some language-related problems, we need to introduce strings somehow.

Let's introduce an alphabet:

- $\Sigma = \{\text{n},\ \text{d},\ \text{q}\}$
- An input is a string $s \in \Sigma^*$
- The string is processed one character at a time
- We determine **states** to track information about the string.
  - Total amount so far, in increments of 5 cents

So, I could lay this out in a table:

Table 4: The table of inputs and the next state

I/S	\$0.00	\$0.05	\$0.10	\$0.15	\$0.20	\$0.25	\$0.30+
n	\$0.05	\$0.10	\$0.15	...			
d	\$0.10	\$0.15	\$0.20	...			
q	\$0.25	\$0.30+	\$0.30+	...			

Input: `dndd`

$$0 \xrightarrow{\text{d}} 10 \xrightarrow{\text{n}} 15 \xrightarrow{\text{d}} 25 \xrightarrow{\text{d}} \text{30+}$$

Formally, this example illustrates everything in a deterministic final state automaton (DFA). A DFA consists of $(Q,\ \Sigma,\ \delta,\ s,\ F)$:

- $Q =$ set of states (finite, non-empty)
- $\Sigma =$ input alphabet (finite, non-empty)
  - $\Sigma \cap Q = \varnothing$ (To prevent look-alike conflicts, Q and Σ are disjoint)
- $\delta : Q \times \Sigma \to Q$ – transition function:
  - `delta(cur_state, input)` $\to$ `next_state` after processing a (`input`) from state q (`cur_state`)
  - In other words, $\forall q \in Q,\ \forall a \in \Sigma,\ \delta(q,\ a) =$ next state after processing a from state q.
- $s \in Q =$ the start/initial state
- $F \subseteq Q =$ accepting/~~final~~ states
  - States signaling "good" strings.
  - For example: $F = \{\$30+\}$
  - I don't use final states, particularly if end anchors are used or something else that might make an "accepting" state "unaccepting".

11.1 State Transition Diagram

Using a table to represent a DFA is fine, but we could do better.

Try to come up with a DFA for strings of the form:

- `+n`
- `-n`
- `+n.m`
- `-n.m`

Where $n,\ m \in \{0,\ 1,\ \ldots,\ 9\}^*$, $n \neq \varepsilon$, $m \neq \varepsilon$. For example:

MATCHING

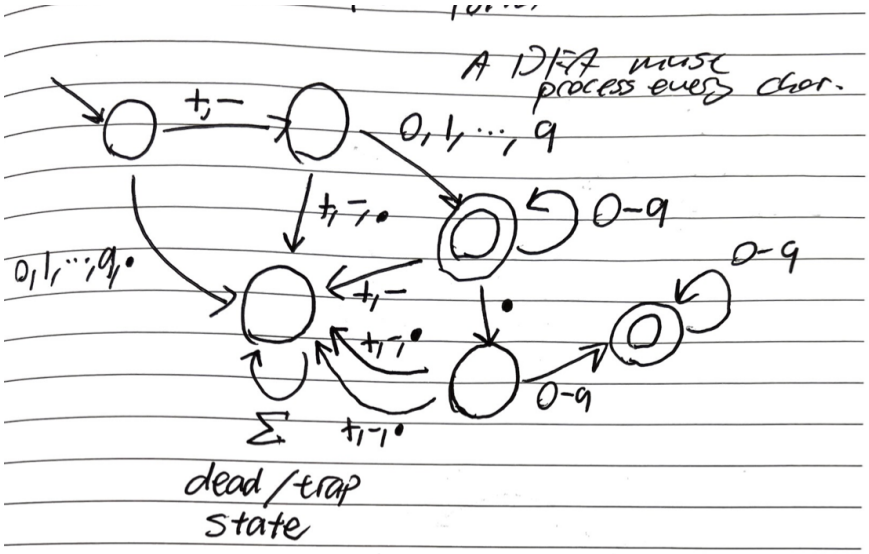
- `+1.2`
- `-3`

DOES NOT MATCH

- `1.2`
- `-3-4++.`
- `+3.`
- `-.5`

The regex is `[\+\-]\d*(\.\d)?`

So, $\Sigma = \{0,\ 1,\ \ldots,\ 9,\ +,\ -,\ .\}$. Your automata contain any characters in this order, even if it makes zero sense. A DFA can't stop until the string ends, so when you're in a reject state, just stay in the reject state.



Every character corresponds to 1 and only 1 transition.

11.2 Writing Proofs about Final Automata

How does this correspond to a language? Give me a DFA, and the language is all strings that are accepted. To do that, here's one more notation:

We know that

- $\delta(q,\ a) =$ state reached from q after processing character a
  - Where $q \in Q,\ a \in \Sigma$
  - The transition function gives us the definition of a DFA.

To prove properties about the DFA, when I start and process this string, where do I end up? Generalize the transition function. There's a standard notation for this:

$$\delta^*(q,\ s) = \text{state reached from } q \text{ after processing the entire string } s$$
$$\text{where } q \in Q,\ s \in \Sigma^*$$

So, δ^* tells us what happens after we process a string.

Note 1. Recursive definition. Strings are either empty or they contain characters. If a string isn't empty, it has a first character (or a last character and a whole string in front). We could define this any way, but say "everything, followed by the last character" for now.

If I'm in state q and I process every character in the empty string, nothing happens, so I'll remain in the same state.

When I process any string that is composed of something followed by the last character, I'll have to process the last character from the state I processed everything before.

$$\forall s \in \Sigma^*,\ \delta^*(q,\ s) = \begin{cases} q & s = \varepsilon \\ \delta\left(\delta^*\left(q,\ s'\right),\ a\right) & s = s'a \text{ for some } a \in \Sigma \end{cases}$$

I'd like to see this as `[PROCESSED RECURSIVELY]a`, where `a` just happens to be the last character. Note that Σ does not contain ε.

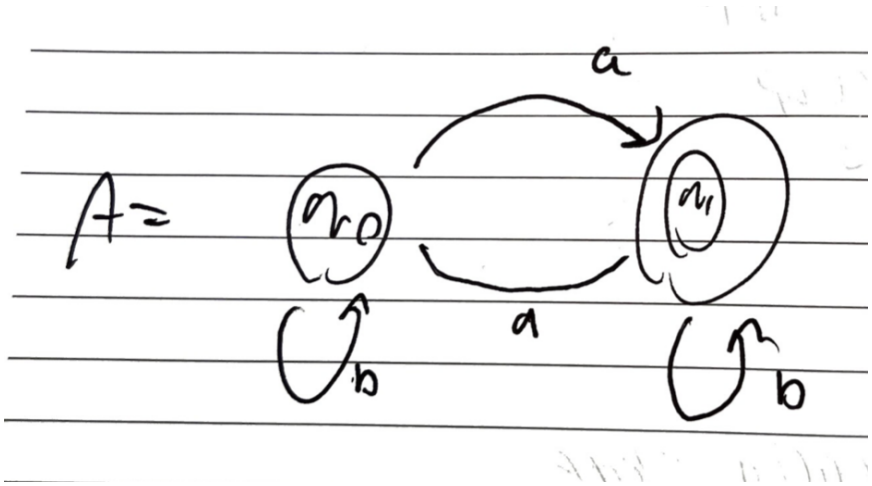
Definition 11.1. For any DFA $A = (Q,\ \Sigma,\ \delta,\ q_0,\ F)$, the language formed by it is every string in the input alphabet **with the following property:** If I process the string, **starting in the initial state**, process string s character by character gives me an accepting state.

$$\mathscr{L}(A) = \{s \in \Sigma^* \ : \delta^*(q_0,\ s) \in F\}$$

Annotating this:

$$\mathscr{L}(A) = \left\{ s \in \Sigma^* \ : \delta^* \left(\begin{array}{c} \text{start} \\ \text{in initial state} \\ \overbrace{q_0} \end{array} , \begin{array}{c} s \\ \text{process } s \\ \text{char by char} \end{array} \right) \in \begin{array}{c} \text{last state} \\ \text{is accepting} \\ F \end{array} \right\}$$

Say we have this DFA:



And we want to show that $\mathscr{L}(A) = \{s \in \{a,\ b\}^*\ :$ no. a's in s are odd$\}$

From a state, where do I end up? What does that mean about the string if I end up in a certain state?

Definition 11.2 (State invariant)**.** A state invariant is a property that is true for all strings... from somewhere. Here's an example:

$$\delta^*(q_0,\ s) = \begin{cases} q_0 & \text{iff no. a's in } s \text{ are even} \\ q_1 & \text{else} \end{cases} \qquad s \in \Sigma^*$$

What these proofs boil down to is a long case-by-case analysis of every single individual transition in the DFA. In your proof, you end up explaining, for every one of these transitions, if you have string that takes you to that state, the new state has that property, and your transition does this... that's too long ($\mathscr{O}(n^2)$ number of steps, which gets overwhelming quick). At least we have only two states for the current DFA we're looking at. You'll never be asked to do that on a test.

11.3 Dead State Convention

A convention we'll use is when we have a dead state, it just crams the diagram. If you have a dead state, **just don't draw it, and assume that it's the default.**

12 Non-Deterministic Final State Automata (NFAs)

Example 12.1. $L = \{s \in \{a,\ b,\ c\}^*\ : s$ ends with babc$\}$

The regex is `/.*babc$/`. What we want to say: just process everything until you get to the last four characters. The issue is that the finite automata doesn't know where the last four characters are.

Hence, introduce **non-deterministic FSAs.**

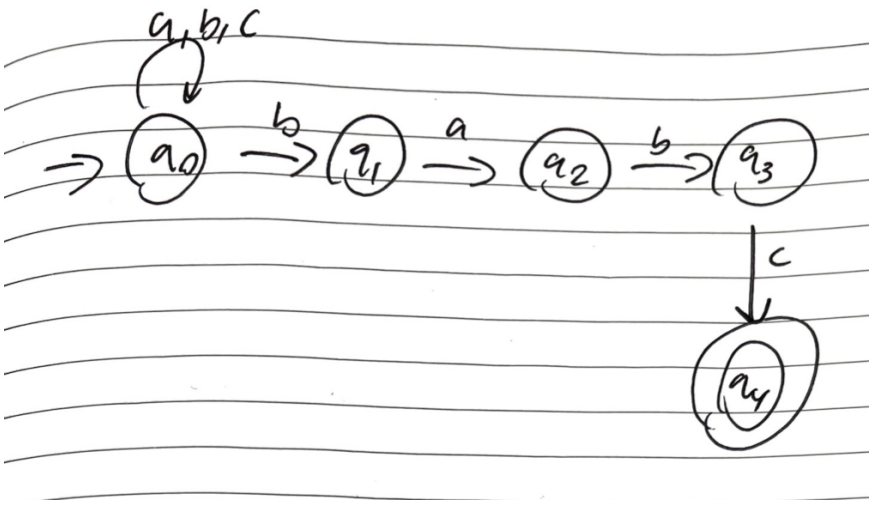


Figure 1: NFA

Intuition. An NFA can be in **any** combination of states at the same time, from **none** to **all**. Of course, if it ends up in no state, there's no way to recover (dead state).

How do I do it. At the end of the string, check if at least one state is accepting. Otherwise, keep processing.

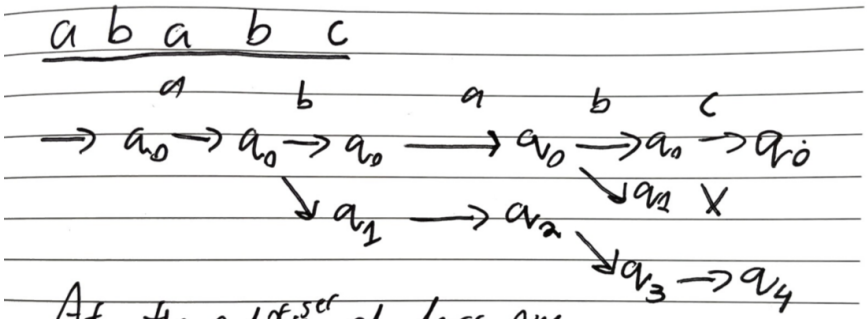


Figure 2: NFA tracing of the above

12.1 Empty String Transitions

How many states can an NFA be at the same time?

- As many

What about initially?

- **Always one,** it's initial state

But that's limiting! **There's a workaround.** Let the ε transition run immediately. When I'm in a certain state and I process a certain character, I may have more than one next state. But what about empty string transitions?

This **does not mean that** ε is an input character! This is a different kind of transition. We might want to draw the arrow differently, like a dashed arrow, but in practice, we never draw dashed arrows. We instead put an epsilon label on the transition. Epsilons are automatic and no time passes. When I hit the tail end of an epsilon transition, it happens in the same tick. If there are multiple epsilon transitions in a row (series or in parallel), they all happen at the same time.

So, whenever you're in the first state of an ε transistion, you're in the second one.

This means that ε-transitions put the NFA explicitly in multiple states.

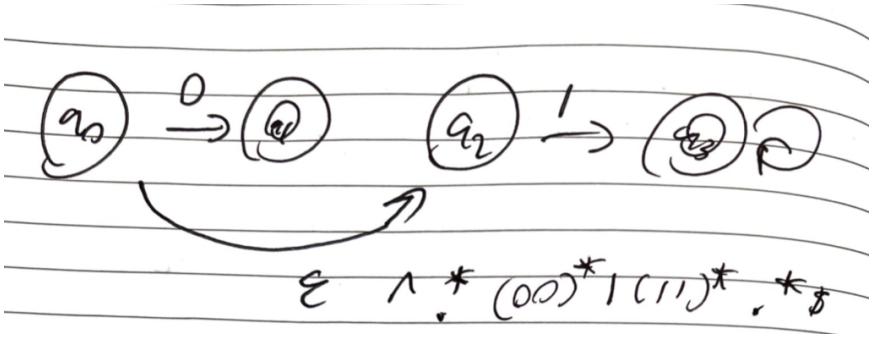


Figure 3: NFA with the ε.

If I trace on the input string 011:

- The NFA starts at the state q_0
  - But there, I can see an ε transition
  - I'm immediately in state 2 and state 1 at the same time.

So:

- $q_0 \to q_1 \to X$ (reject)
- $q_2 \to X$ (accept)

What is the language?

$$\mathscr{L}(A) = \{0\} \cup \left\{ s \in \{0,\ 1\}^* \ : \text{starts with 1} \right\}$$

And that is non-determinism.

13 Regular Languages

If I can use a regex, DFA, or NFA to represent something, I can represent it with all else.

Definition 13.1. Language $L \subseteq \Sigma^*$ is **regular**:

- Iff $L = \mathscr{L}(R)$ for some regex R
- Iff $L = \mathscr{L}(A)$ for some DFA A
- Iff $L = \mathscr{L}(A)$ for some NFA A'

A language is a subset, so it is like tracing a boundary for all strings. Some strings have some property that I care about, others don't, and languages are what I can say about them. There's a pattern I can describe so all strings I want fits the patter and all strings that I don't want don't fit that pattern. I. could draw a DFA and NFA such that I run every possible string through it, it will say yes for all strings in my language, and no for all strings that aren't.

Non-determinism is something you're allowed to do in an NFA, but you're not required to do it.

13.1 NFA to DFA

NFA $\Rightarrow$ DFA:

Given NFA $A = (Q,\ \Sigma,\ \delta,\ s,\ F)$, construct DFA $A' = (Q',\ \Sigma,\ \delta',\ s',\ F')$ such that $\mathscr{L}(A') = \mathscr{L}(A)$.

Insight (subset construction). DFA's states represent subsets of states from the NFA.

To convert an NFA to a DFA, I might need exponentially more states. I can define the same languages, but I may need a lot of work to convert it into a DFA.

By convention, $\{0,\ 1,\ 3\} = \{q_0,\ q_1,\ q_3\}$ for now. This just reduces strain from handwriting.

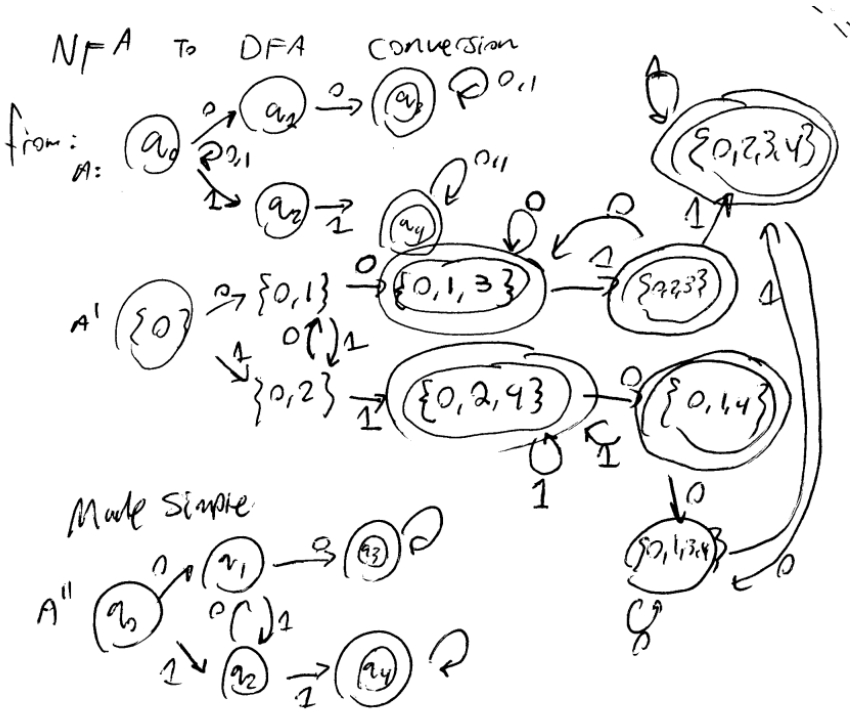


Figure 4: NFA to DFA conversion

Here's a way I'd like to picture NFA to DFA conversions. Here's a picture of determining one transition: let's say $\{0,\ 2\} \xrightarrow{1}$.

On the NFA, put your fingers over q_0 and q_2. Move your fingers in the direction of the arrow of $\xrightarrow{1}$ (if there isn't, get rid of that finger because it's a dead state). Where your fingers end up represent the set of states you should end up writing. You may need to add fingers if new states come up.

13.2 DFA to RE

Intuition. Remove all non-accepting states except the initial state. We're going to do the subset elimination construction. Look at every accepting state in my DFA. Each accepting state represents a certain pattern that strings can have to be accepted. For each of these accepting states, remove everything but the accepting state and the initial state.

We'll do it in such a way that looks like a transition, but is not a transition diagram, as we will start changing the labels of the transitions. The labels will eventually look like more complicated regular expressions.

At the end, we will end up with a singular regular expression.

Remove one at a time, until you've removed all of them.

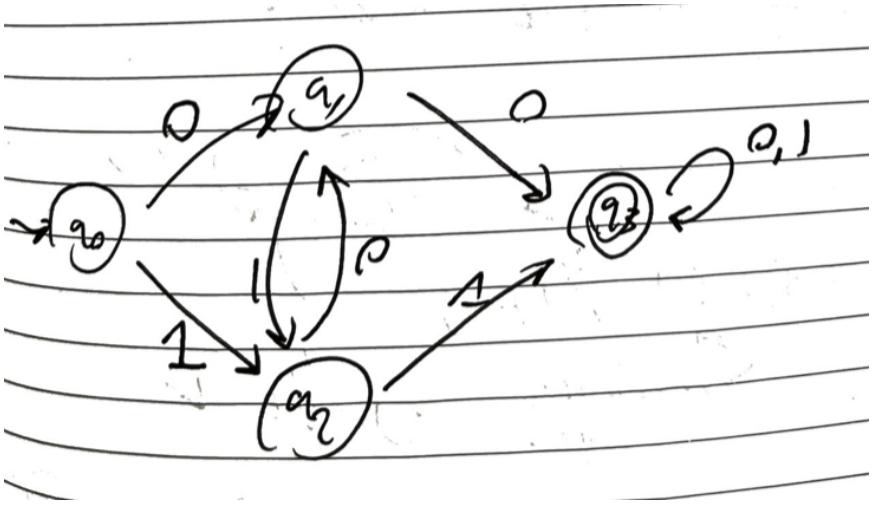


Figure 5: DFA to be converted to a regex

Here, we remove q_1, as seen below:

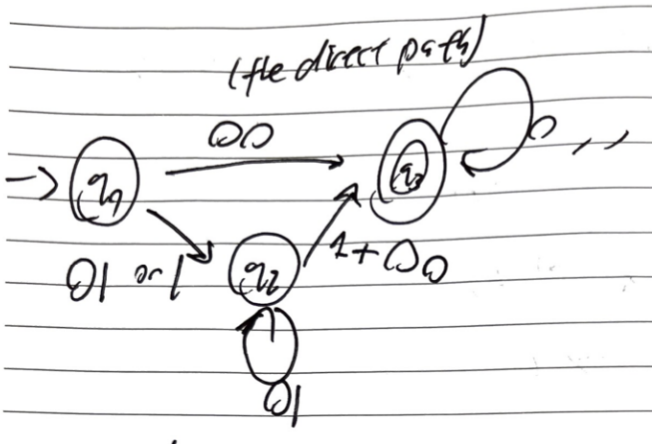


Figure 6: DFA to regex, step 2

We no longer have a transition diagram. We instead have an intermediate, which contains the same information in our transition diagram. We then remove q_2:

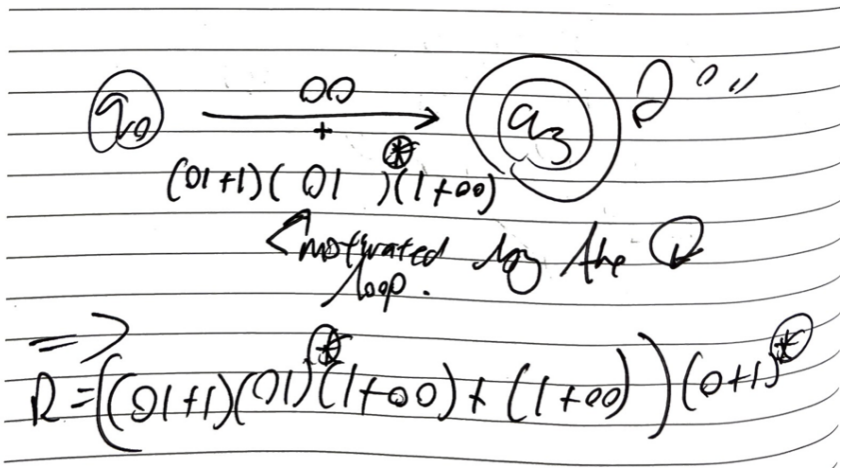


Figure 7: DFA now fully converted to a regex

And there's our regex. A loop back is a sign that a(n) $\circledast$ is used.

You'll always get something that works, but not always something that is easy.

13.3 Regex to NFA

If I have an arbitrary regular expression, how do I construct an NFA from it?

- Well, we can do our construction recursively, as regular expressions are built up recursively.

I'm going to show that any regular expression can be written as an NFA.

Here are the basic regular expressions:

- $\varnothing$
  - NFAs composed of this accept nothing
  - $\mathscr{L}(\rightarrow \circ)$
- ε
  - NFAs where its initial state is accepting, and nothing else. Any other character moves it to a dead state
  - $\mathscr{L}(\rightarrow \circledcirc)$
- a

- Initial state is non-accepting. Only accepts one a, and that's it.
- $\mathscr{L}\left(\rightarrow \circ \xrightarrow{a} \circledcirc\right)$

Here's the recursive constructions:

Let $R_1,\ R_2$ be arbitrary regular expressions. Assume A_1 and A_2 exist such that $\mathscr{L}(A_1) = \mathscr{L}(R_1)$ and $\mathscr{L}(A_2) = \mathscr{L}(R_2)$.

- $R_1 + R_2$:
  - Use double epsilons and a new starting state. Hence, I immediately start at the start of both A_1 and A_2 at the start time.
  - $\mathscr{L}\left(\rightarrow \circ \overset{\varepsilon}{\underset{\varepsilon}{\rightrightarrows}} \begin{matrix} A_1 \\ A_2 \end{matrix}\right)$
- $R_1 \cdot R_2$:
  - Strings that match $R_1 \cdot R_2$ can be split into two substrings: one that matches R_1 immediately followed by the rest matching R_2
  - All accepting states in A_1 are no longer accepting states, and instead I add an ε transition from all formerly accepting states in A_1 to the start of A_2.
  - $\mathscr{L}\left(\rightarrow \circ \rightarrow (\circ \rightarrow \overset{A_1}{\cdots} \rightarrow \mathscr{O}) \xrightarrow{\varepsilon} A_2\right)$
    - Notation only here: $\mathscr{O}$ is a former accepting state, which is to be removed in this process for A_1 (but don't touch A_2).
- $R_1^{\circledast}$:
  - $\mathscr{L}\left(\rightarrow \circledcirc \xrightarrow{\varepsilon} (\circ \rightarrow \overset{A_1}{\cdots} \rightarrow \mathscr{O}) \xrightarrow{\varepsilon} \cdots\right)$

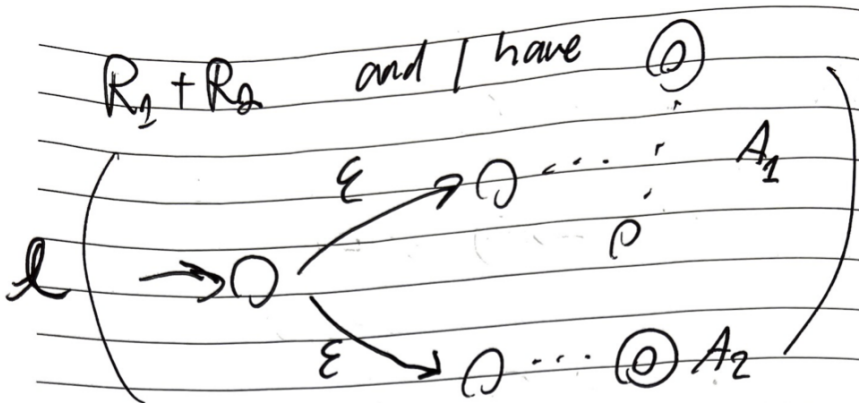


Figure 8: DFA from adding regular expressions

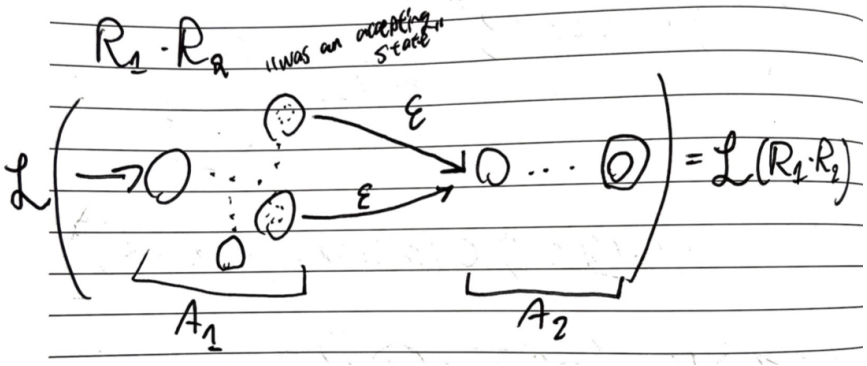


Figure 9: DFA from concatenating regular expressions

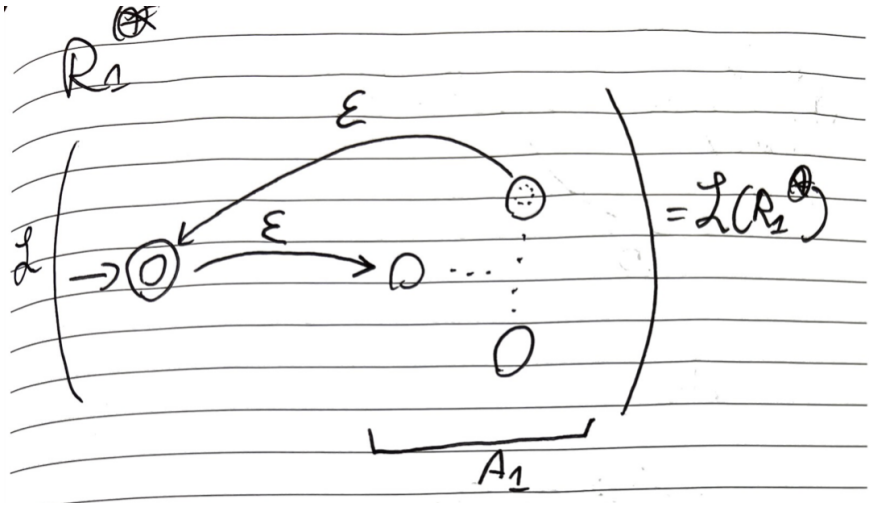


Figure 10: DFA from the $\circledast$ operator

14 Closure

Recall. If a language is regular, there exists a regex, DFA, and NFA that represents it.

What kinds of operations can I do with regular languages that give me new languages that are still regular?

14.1 Union

Example 14.1. Let $L_1,\ L_2$ be regular. Is $L_1 \cup L_2$ regular? Yes, and here's the proof:

Proof. Assume $\exists R_1,\ R_2$ s.t. $\begin{matrix} L_1 = \mathscr{L}(R_1) \\ L_2 = \mathscr{L}(R_2) \end{matrix}$. Then, $\mathscr{L}(R_1 + R_2) = L_1 \cup L_2$.

■

14.2 Complement / Negation

Example 14.2. Let L be regular. Is $\overline{L}$ regular? Note that $\overline{L} = \Sigma^* - L$.

No, regular expressions don't help. Using a contradiction is tricky here.

Proof. Assume $\exists$ DFA $\underset{Q,\ \Sigma,\ \delta,\ s,\ F}{A}$ s.t. $L = \mathscr{L}(A)$. Draw the DFA and include all dead states. Replace rejecting states with accepting states and accepting states with rejecting states **(swap them)**. And you're done.

Symbolically:

Let $A' = \left(Q,\ \Sigma,\ \delta,\ s,\ \overset{\text{only different}}{Q - F}\right)$. That is exactly what I want: $\mathscr{L}(A') = \overline{\mathscr{L}(A)} = \overline{L}$.

Extra symbolically:

$$\forall w \in \Sigma^*,\ w \in L \Leftrightarrow \quad \underbrace{\delta^*(s,\ w)}_{\substack{\text{last state of } a \\ \text{after processing } w \\ \text{starting at } s}} \quad \in F$$

Then, negate both sides. The iff will still be true.

$$\forall w \in \Sigma^*,\ w \notin L \Leftrightarrow \underbrace{\delta^*(s,\ w)}_{\substack{\text{last state of } a \\ \text{after processing } w \\ \text{starting at } s}} \notin F \Leftrightarrow \delta^*(s,\ w) \in Q - F$$

■

When making an argument about finite automata, start with a DFA, and switch to an NFA if arguing about it is easier.

14.3 Product Construction

Example 14.3. $L_1,\ L_2$ is regular $\Rightarrow L_1 \cap L_2$ is regular. Is this statement true?

Introducing **product construction.** We'll merge two DFAs together.

Let us declare:

$$\begin{matrix} L_1 = \{s \text{ contains aaa}\} \\ L_2 = \{s \text{ has odd b chars}\} \end{matrix} \subseteq \{a,\ b\}^*$$

What to do to get $A_1 \times A_2$:

1. Track all pairs of states by writing them down.
2. Trace through A_1 and A_2 simultaneously.
3. The accepting state is when both A_1 and A_2 are at an accepting state.

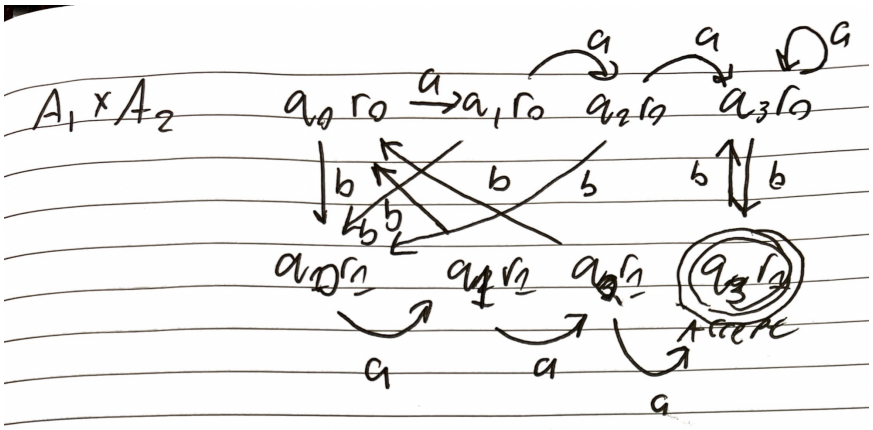


Figure 11: Product construction

14.4 Summary

- Union: use the regex $+$ operator.
- Complement: swap accepts and rejects on a DFA
- Intersection: Use product construction.

15 Pumping Lemma

Reversing or swapping orders preserve regularity.

15.1 Proving Regularity

Suppose $L \subseteq \{0,\ 1,\ 2\}^*$ is regular.

Is $L' = \left\{ s \in \{0,\ 1,\ 2\}^* \ : \ \begin{array}{c} s = 1x \text{ for some } x \in L, \text{ or} \\ s = 0x \text{ for some } x \in \overline{L} \end{array} \right\}$ regular? How do we argue it?

Yes, L' is still regular.

Two arguments:

- Work with DFAs
  a. Now we have all three models, we can rely on all properties.
  b. L is regular $\Rightarrow \exists$ DFA A_1 such that $\mathscr{L}(A_1) = L$
  c. $\Rightarrow \overline{L}$ is regular $\Rightarrow \exists$ DFA A_2 such that $\mathscr{L}(A_2) = \overline{L}$
  d. Then, $\boxed{A' = \rightarrow \circ \overset{1}{\underset{0}{\rightrightarrows}} \begin{array}{c} A_1 \\ A_2 \end{array}}$, where $\mathscr{L}(A') = L' \Rightarrow L'$ is regular.
  e. To be formal, we must do is symbolically. This may not be enough, but the argument remains effective.
- Working with regular expressions
  a. L is regular $\Rightarrow \exists$ RE R_1 such that $\mathscr{L}(R_1) = L$
  b. $\Rightarrow \overline{L}$ regular $\Rightarrow \exists$ RE R_2 such that $\mathscr{L}(R_2) = \overline{L}$
  c. Then, $\mathscr{L}(1 \cdot R_1 + 0 \cdot R_2) = \mathscr{L}(1) \cdot \mathscr{L}(R_1) \cup \mathscr{L}(0) \cdot \mathscr{L}(R_2) = \{1\} \cdot L \cup \{0\} \cdot L$
  d. This matches the exact description of L', and is the clearest. This is formal enough.

15.2 Non-Regular Languages

Consider $L = \{0^n 1^n \ : n \in \mathbb{N}\}$ (some numbers of zeroes followed by the same number of 1s) $= \{\varepsilon,\ 01,\ 0011,\ 000111,\ \ldots\}$

Claim 1. L is NOT regular.

Question. Isn't $L \stackrel{?}{=} \mathscr{L}(0^{\circledast}1^{\circledast})$? Well, $L \subseteq \mathscr{L}(0^{\circledast}1^{\circledast})$, but $\mathscr{L}(0^{\circledast}1^{\circledast}) \nsubseteq L$.

What does it mean for a language **to not** be regular. We know what regular means. When a language is regular, there's an RE, DFA, and NFA for it.

Proof. WTP: **NO** DFA accepts L. This means

$$\forall \text{ DFAs } A,\ \mathscr{L}(A) \neq L$$

- Let A be an arbitrary DFA.
- Either A accepts all strings in L, or it doesn't.
- **Case 1.** If A rejects some $s \in L$, then $\mathscr{L}(A) \neq L$.
- **Case 2.** If A accepts all $s \in L$, then we show A also accepts some string $s' \notin L$.
  - **Insight.** This only works because L is an infinite set, yet every single string in L is finite. This means that the strings in L keep getting longer and longer. But any DFA has a fixed number of states. I get any DFA. It has a fixed number of states. If I have it process a string of 0s and 1s that is long enough (and there will be, long enough refers to more 0s in the initial portion than I have states in my DFA)
  - Let $k =$ no. states of A.
  - Consider computation of A on input $0^{k+1}1^{k+1}$.
  - $\left(0^{k+1}1^{k+1} \in L \Rightarrow A \text{ accepts it}\right)$
  - Tracking the states: $q_0 \xrightarrow{0} \boxdot \xrightarrow{0} \cdots \xrightarrow{0} q_m \xrightarrow{1} \boxdot \xrightarrow{1} \cdots \xrightarrow{1} \circledcirc$
  - According to the pigeonhole principle, the sequence of states encountered contains at least one repeated state, because the number of 0s is greater than the number of states.
    - $\Rightarrow$ DFA contains at least one loop. It could be a loop over multiple states (one state or multiple states).
    - That loop processes some number $i \geq 1$ of 0s.

* If I'm back in the same state as earlier, what I know about the string is the same. I don't know anything else I knew before.
* If I feed to my DFA, a string that contains enough extra 0s to go through the loop one more time
* $q_0 \to \cdots \to \underbrace{\boxed{q_j \xrightarrow{0} \square \xrightarrow{0} \cdots \xrightarrow{0} q_j}}_{i} \to \cdots \to \circledcirc$
* Then, $0^{k+1+i}1^{k+1}$ results in an accepting state.
* Then, $0^{k+1+2i}1^{k+1}$ is also accepted, and
* $0^{k+1-i}1^{k+1}$ is accepted.

■

15.3 Pumping Lemma

In practice, the pumping lemma is used to prove that languages are not regular.

For all regular languages $L \subseteq \Sigma^*$:

- There exists a constant $p \in \mathbb{Z}^+$ such that
- For all strings $w \in L$ with $|w| \geq p$ (if I can't find a w, then I have a vacuous truth)
- There exist strings $x,\ y,\ z \in \Sigma^*$ such that:
  - $w = x \cdot y \cdot z$
  - $|x \cdot y| \leq p$
  - $|y| \geq 1$
  - $\forall i \in \mathbb{N}$: $x \cdot y^i \cdot z \in L$

Lemma 1 (Pumping Lemma)**.** *For all regular languages $L \subseteq \Sigma^*$:*

- *There exists a constant $p \in \mathbb{Z}^+$ such that*
- *For all strings $w \in L$ with $|w| \geq p$*
- *There exist strings $x,\ y,\ z \in \Sigma^*$ such that:*
  - $w = x \cdot y \cdot z$
  - $|x \cdot y| \leq p$
  - $|y| \geq 1$

- $\forall i \in \mathbb{N}: x \cdot y^i \cdot z \in L$

I get any regular language. There is a constant, so that all the strings that contain that much more than that many characters such that the middle part is non-empty, and it can be used to generate more strings in the language.

If the language is regular, there is a DFA in the language. Look at the state count. Any string that is longer than the number of states means that there's a loop in the DFA

If the DFA accepts any of those strings, there's a non-empty part inside the string that can be repeated any number $\in \mathbb{N}$ of times.

This is an abstraction of the argument through DFAs, of what we know about regular languages. How does this help show that a language is not regular?

$$w = \underbrace{x \underbrace{\ y\ }_{\geq 1} z}_{\leq p}$$

15.4 Using The Pumping Lemma in the Other Direction

To show that L is not regular, negate the pumping lemma.

Let $p \in \mathbb{Z}^+$ be arbitrary.

- **Choose** a string $w \in L$ with $|w| \geq p$.
  - Let $x,\ y,\ z \in \Sigma^*$ be arbitrary such that
    - $w = x \cdot y \cdot z$
    - $|x \cdot y| \leq p$
    - $|y| \geq 1$
    - $x \cdot y \cdot z \in L$ is implied already
- Choose $i \in \mathbb{N}$ such that $x \cdot y^i \cdot z \notin L$.

Use this way to quickly memorize what is arbitrary and what is to be chosen

> Arbitrary $\forall p \rightarrow$ chosen $\exists w \rightarrow$ arbitrary $\forall x,\ y,\ z \rightarrow$ chosen $\exists i$ such that $x \cdot y^i \cdot z \notin L$

For example, if we want to show that $\{0^n 1^n \ : n \in \mathbb{N}\}$ is not regular:

Proof. Let $p \in \mathbb{Z}^+$ be arbitrary.

- Choose $w = 0^p 1^p = \underbrace{000\cdots000}_{p \text{ times}}\underbrace{111\cdots111}_{p \text{ times}}$
- Let $x,\ y,\ z \in \Sigma^*$ be arbitrary such that
  - $w = x \cdot y \cdot z$
  - $|x \cdot y| \leq p$
  - $|y| \geq 1$
- This means that xy only consists of 0s
  - Hence, y is 0^k for some $k \geq 1$ (because y is only zeroes and we know the length of it is ≥ 1)
  - But we can also construct $w_2 = xy^2z = 0^{p+k}1^p \notin L$
  - This is enough to conclude that L is not regular.

■

15.5 Some Harder Proofs Of Non-Regularity

15.5.1 Prime

The proof that $L = \{w \in \Sigma^* \ : \exists p \in \mathbb{N},\ |w| = p \wedge p \text{ is prime}\}$ isn't regular.

Proof. Let p be arbitrary. Let m be any prime greater than p.

- Choose $w = h^m$.
  - Let $x,\ y,\ z$ be arbitrary strings such that
    - $w = xyz$
    - $|xy| \leq p$ and $|y| \geq 1$
  - Then, we have
  - But we also get $w_2 = xy^{1+m}z$
    - We can say $w_2 = xyy^m z$
    - Hence, $|w_2| = m + |y| \cdot m$
    - $= m(1 + |y|)$
    - Very sure it's a composite number, so it is not in L

■

15.5.2 Palindrome

The proof that $L = \{w \in \Sigma^* \ : \ w \text{ is a palindrome}\}$ isn't regular.

Proof. Let $p \in \mathbb{N}^{\geq 1}$ be arbitrary.

- Choose $w = 0^p 10^p$.
  - Let $x,\ y,\ z$ be arbitrary such that:
    - $w = xyz$
    - $|xy| \leq p$
    - $|y| \geq 1$
  - Then, xy matches $0^{\circledast}$.
  - $w_2 = xy^2z$ is not a palindrome and is $\notin L$
    - As this pushes the only 1 in w_2 to the right by an offset of $|y|$, so 1 is not centered; yet there is no 1 to the left of the center of w_2.

■

16 Review

This course has three parts:

1. **Induction.** Simple induction, complete induction, structural induction, and PWO. We used these so we can apply their results. All of these have very specific proof structures, and they're important particularly in time-pressured environments.
2. **Algorithm analysis.** Iterative and recursive algorithms. We saw these two in the context of correctness.
   a. For recursive algorithms, we partition input sizes in lengths that are smaller.
   b. For iterative algorithms, you need
      i. Loop invariants
      ii. The loop ends
      iii. Partial correctness (preconditions and the loop runs and stop implies postconditions)
   c. Runtime analysis: look at the algorithm, look at how much work it does, and get the expression about the runtime of the algorithm. We get the recurrence relation of the runtime, and we work to get a closed form.

    i. To prove things more in rigor, use matching bounds or the master theorem.

3. **Formal Language Theory.** Way lower level.

There were a lot of things that we looked at from a high-level perspective (things that can be proved by induction) – we can use a lot of tools we've developed to prove things on formal language theory. That's one of the very important things that allows us to write more rigorous proofs.