

Warm up:

What is the following recursively-defined set?

Basis Step:  $4 \in S, 5 \in S$

Recursive Step: If  $x \in S$  and  $y \in S$  then  $x - y \in S$



# Structural Induction and Regular Expressions

[xkcd.com/208](http://xkcd.com/208)

CSE 311 Spring 2022

Lecture 20

# Worried about grades?

Or just want a discussion of how to finish out the quarter strong

I'll have 15-minute blocks for one-on-one discussions end of this week/early next week:

<https://calendly.com/robbieweber/311checkin>

On zoom by default, but if needed we can talk in-person.

You should also take a look at the grade guarantees in the syllabus. If exams turn out harder than expected, we may lower those thresholds, but we'll be no less generous than those.

# Announcements

New on the resources tab: a note about backwards proofs with examples!

We're adding a fact you can use to HW6 Problem 2 [the proof by contradiction] (will be up by noon today).

You could probably have proven a "simple" version yourself, but it would have made the proof...uglier...and distracted from the point of practicing proof by contradiction.

# Strings

Why these recursive definitions?

They're the basis for regular expressions, which we'll introduce next week. Answer questions like "how do you search for anything that looks like an email address"

*Char*  
First, we need to talk about strings.

$\Sigma$  will be an **alphabet** the set of all the letters you can use in words.

$\Sigma^*$  is the set of all **words** all the strings you can build off of the letters.

*strings*

# Strings

$\varepsilon$  is "the empty string"

The string with 0 characters – "" in Java (not null!)

$\Sigma^*$ :

Basis:  $\varepsilon \in \Sigma^*$ .

Recursive: If  $w \in \Sigma^*$  and  $a \in \Sigma$  then  $wa \in \Sigma^*$

$wa$  means the string of  $w$  with the character  $a$  appended.

You'll also see  $w \cdot a$  ( $a \cdot$  to mean "concatenate" i.e. + in Java)

# Functions on Strings

Since strings are defined recursively, most functions on strings are as well.

Length:

$$\text{len}(\varepsilon) = 0;$$

$$\text{len}(wa) = \text{len}(w) + 1 \text{ for } w \in \Sigma^*, a \in \Sigma$$

Reversal:

$$\varepsilon^R = \varepsilon;$$

$$(wa)^R = aw^R \text{ for } w \in \Sigma^*, a \in \Sigma$$

Concatenation

$$x \cdot \varepsilon = x \text{ for all } x \in \Sigma^*;$$

$$x \cdot (wa) = (x \cdot w)a \text{ for } w \in \Sigma^*, a \in \Sigma$$

Number of  $c$ 's in a string

$$\#_c(\varepsilon) = 0$$

$$\#_c(wc) = \#_c(w) + 1 \text{ for } w \in \Sigma^*;$$

$$\#_c(wa) = \#_c(w) \text{ for } w \in \Sigma^*, a \in \Sigma \setminus \{c\}.$$

Claim  $\text{len}(x \cdot y) = \text{len}(x) + \text{len}(y)$  for all  $x, y \in \Sigma^*$ .

Let  $P(y)$  be " $\text{len}(x \cdot y) = \text{len}(x) + \text{len}(y)$  for all  $x \in \Sigma^*$ ."

Notice the strangeness of this  $P()$  there is a "for all  $x$ " inside the definition of  $P(y)$ .

That means we'll have to introduce an arbitrary  $x$  as part of the base case and the inductive step!

Claim  $\text{len}(x \cdot y) = \text{len}(x) + \text{len}(y)$  for all  $x, y \in \Sigma^*$ .

Define Let  $P(y)$  be " $\text{len}(x \cdot y) = \text{len}(x) + \text{len}(y)$  for all  $x \in \Sigma^*$ ."

We prove  $P(y)$  for all  $y \in \Sigma^*$  by structural induction.

Base Case:

Inductive Hypothesis:

Inductive Step:

$\Sigma^*$ : Basis:  $\varepsilon \in \Sigma^*$ .

Recursive: If  $w \in \Sigma^*$  and  $a \in \Sigma$  then  $wa \in \Sigma^*$



Claim  $\text{len}(x \cdot y) = \text{len}(x) + \text{len}(y)$  for all  $x, y \in \Sigma^*$ .

Define Let  $P(y)$  be " $\text{len}(x \cdot y) = \text{len}(x) + \text{len}(y)$  for all  $x \in \Sigma^*$ ."

We prove  $P(y)$  for all  $y \in \Sigma^*$  by structural induction.

Base Case: Let  $x$  be an arbitrary string,  $\text{len}(x \cdot \epsilon) = \text{len}(x) = \text{len}(x) + 0 = \text{len}(x) + \text{len}(\epsilon)$ . So we have  $P(\epsilon)$ .

Inductive Hypothesis: Suppose  $P(w)$  for an arbitrary  $w \in \Sigma^*$ .

Inductive Step:

$\Sigma^*$ :Basis:  $\epsilon \in \Sigma^*$ .

Recursive: If  $w \in \Sigma^*$  and  $a \in \Sigma$  then  $wa \in \Sigma^*$

$P(wa)$   
 $\text{len}(x \cdot wa) = \text{len}(x) + \text{len}(wa) = \text{len}(x) + \text{len}(w) + \text{len}(a)$

**Claim**  $\text{len}(x \cdot y) = \text{len}(x) + \text{len}(y)$  for all  $x, y \in \Sigma^*$ .

Define Let  $P(y)$  be " $\text{len}(x \cdot y) = \text{len}(x) + \text{len}(y)$  for all  $x \in \Sigma^*$ ."

We prove  $P(y)$  for all  $y \in \Sigma^*$  by structural induction.

Base Case: Let  $x$  be an arbitrary string,  $\text{len}(x \cdot \epsilon) = \text{len}(x) = \text{len}(x) + 0 = \text{len}(x) + \text{len}(\epsilon)$ . So we have  $P(\epsilon)$ .

Inductive Hypothesis: Suppose  $P(w)$  for an arbitrary  $w \in \Sigma^*$ .

Inductive Step: Let  $a$  be an arbitrary character and let  $x$  be an arbitrary string.

$$\begin{aligned}\text{len}(xwa) &= \text{len}(xw) + 1 \text{ (by definition of len)} \\ &= \text{len}(x) + \text{len}(w) + 1 \text{ (by IH)} \\ &= \text{len}(x) + \text{len}(wa) \text{ (by definition of len)}\end{aligned}$$

Therefore,  $\text{len}(xwa) = \text{len}(x) + \text{len}(wa)$ . I.e.,  $P(wa)$  holds.

By the principle of induction, we have  $P(y)$  for all  $y$ . Therefore for all  $x, y$  we have  $\text{len}(x \cdot y) = \text{len}(x) + \text{len}(y)$ .

$\Sigma^*$ : Basis:  $\epsilon \in \Sigma^*$ .

Recursive: If  $w \in \Sigma^*$  and  $a \in \Sigma$  then  $wa \in \Sigma^*$

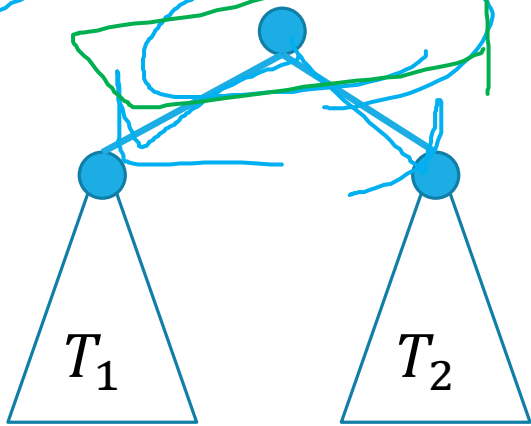
# More Structural Sets

Binary Trees are another common source of structural induction.

Basis: A single node is a rooted binary tree.



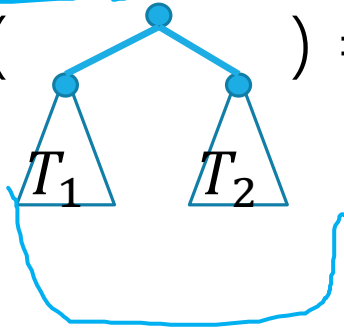
Recursive Step: If  $T_1$  and  $T_2$  are rooted binary trees with roots  $r_1$  and  $r_2$ , then a tree rooted at a new node, with children  $r_1, r_2$  is a binary tree.



# Functions on Binary Trees

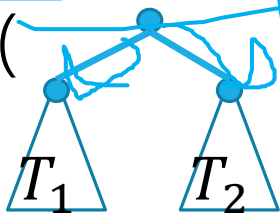
$$\text{size}(\bullet) = 1$$

$$\text{size}(\begin{array}{c} \bullet \\ / \quad \backslash \\ T_1 \quad T_2 \end{array}) = \text{size}(T_1) + \text{size}(T_2) + 1$$



$$\text{height}(\bullet) = 0$$

$$\text{height}(\begin{array}{c} \bullet \\ / \quad \backslash \\ T_1 \quad T_2 \end{array}) = 1 + \max(\text{height}(T_1), \text{height}(T_2))$$



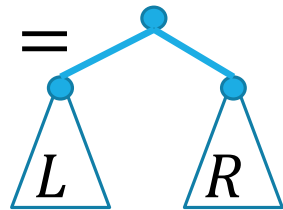
# Structural Induction on Binary Trees

Let  $P(T)$  be "size(T)  $\leq 2^{\text{height}(T)+1} - 1$ ". We show  $P(T)$  for all binary trees  $T$  by structural induction.

Base Case: Let  $T = \bullet$ . size(T)=1 and height(T) = 0, so size(T)=1  $\leq 2 - 1 = 2^{0+1} - 1 = 2^{\text{height}(T)+1} - 1$ .

Inductive Hypothesis: Suppose  $P(L)$  and  $P(R)$  for arbitrary binary trees  $L, R$ .

Inductive Step: Let  $T =$

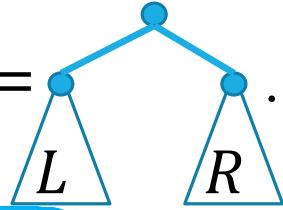


$P(T)$ !

# Structural Induction on Binary Trees (cont.)

Let  $P(T)$  be " $\text{size}(T) \leq 2^{\text{height}(T)+1} - 1$ ". We show  $P(T)$  for all binary trees  $T$  by structural induction.

Inductive Step: Let  $T =$



$$\text{height}(T) = 1 + \max\{\text{height}(L), \text{height}(R)\}$$

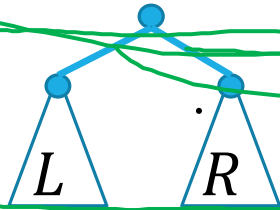
$$\text{size}(T) = 1 + \text{size}(L) + \text{size}(R)$$

So  $P(T)$  holds, and we have  $P(T)$  for all binary trees  $T$  by the principle of induction.

# Structural Induction on Binary Trees (cont.)

Let  $P(T)$  be " $\text{size}(T) \leq 2^{\text{height}(T)+1} - 1$ ". We show  $P(T)$  for all binary trees  $T$  by structural induction.

Inductive Step: Let  $T =$



$$\text{height}(T) = 1 + \max\{\text{height}(L), \text{height}(R)\}$$

$$\text{size}(T) = 1 + \text{size}(L) + \text{size}(R)$$

$$\text{size}(T) = 1 + \text{size}(L) + \text{size}(R) \leq 1 + 2^{\text{height}(L)+1} - 1 + 2^{\text{height}(R)+1} - 1 \quad (\text{by IH})$$

$$\leq 2^{\text{height}(L)+1} + 2^{\text{height}(R)+1} - 1 \quad (\text{cancel 1's})$$

$$\leq 2^{\text{height}(T)} + 2^{\text{height}(T)} - 1 = 2^{\text{height}(T)+1} - 1 \quad (T \text{ taller than subtrees})$$

So  $P(T)$  holds, and we have  $P(T)$  for all binary trees  $T$  by the principle of induction.

# What does the inductive step look like?

Here's a recursively-defined set:

**Basis:**  $0 \in T$  and  $5 \in T$

**Recursive:** If  $x, y \in T$  then  $x + y \in T$  and  $x - y \in T$ .

Let  $P(x)$  be " $5|x$ "

What does the inductive step look like?

Well there's two recursive rules, so we have two things to show



# Just the IS (you still need the other steps)

Inductive hypothesis: Suppose  $P(x)$  and  $P(y)$  hold for some arbitrary  $x, y \in T$ .

Consider  $x + y$

By IH  $5|x$  and  $5|y$  so  $5a = x$  and  $5b = y$  for integers  $a, b$ .

Adding, we get  $x + y = 5a + 5b = 5(a + b)$ . Since  $a, b$  are integers, so is  $a + b$ , and  $P(x + y)$  holds.

Consider  $x - y$

By IH  $5|x$  and  $5|y$  so  $5a = x$  and  $5b = y$  for integers  $a, b$ .

Subtracting, we get  $x - y = 5a - 5b = 5(a - b)$ . Since  $a, b$  are integers, so is  $a - b$ , and  $P(x - y)$  holds.



# CAUTION



Structural induction *looks like* we're violating the rule of "introduce an arbitrary variable to prove a for-all statement"

We're not!

What structural induction really says is "consider an arbitrary element of the recursively-defined set. By the exclusion rule, it's either a basis element, or made from other elements by a rule" and then break into all possible cases.

*Only* when we have an explicit recursive definition of a set can we do this. You should not be "building up" elements in inductive steps.

# If you don't have a recursively-defined set

You won't do structural induction.

You can do weak or strong induction though.

For example, Let  $P(n)$  be "for all elements of  $S$  of "size"  $n$  <something> is true"

To prove "for all  $x \in S$  of size  $n$ ..." you need to start with "let  $x$  be an arbitrary element of size  $k + 1$  in your IS.

You CAN'T start with size  $k$  and "build up" to an arbitrary element of size  $k + 1$  it isn't arbitrary (we only *seem* to do that with structural induction, but we're using the exclusion rule there).

# Induction: Hats!

You have  $n$  people in a line ( $n \geq 2$ ). Each of them wears either a **purple hat** or a **gold hat**. The person at the front of the line wears a purple hat. The person at the back of the line wears a gold hat.

Show that for every arrangement of the line satisfying the rule above, there is a person with a purple hat next to someone with a gold hat.

Yes this is kinda obvious. I promise this is good induction practice.

Yes you could argue this by contradiction. I promise this is good induction practice.

# Induction: Hats!

Define  $P(n)$  to be "in every line of  $n$  people with gold and purple hats, with a purple hat at one end and a gold hat at the other, there is a person with a purple hat next to someone with a gold hat"

We show  $P(n)$  for all integers  $n \geq 2$  by induction on  $n$ .

Base Case:  $n = 2$

Inductive Hypothesis:

Inductive Step:

By the principle of induction, we have  $P(n)$  for all  $n \geq 2$

# Induction: Hats!

Define  $P(n)$  to be "in every line of  $n$  people with gold and purple hats, with a purple hat at one end and a gold hat at the other, there is a person with a purple hat next to someone with a gold hat"

We show  $P(n)$  for all integers  $n \geq 2$  by induction on  $n$ .

Base Case:  $n = 2$  The line must be just a person with a purple hat and a person with a gold hat, who are next to each other.

Inductive Hypothesis: Suppose  $P(k)$  holds for an arbitrary  $k \geq 2$ .

Inductive Step: Consider an arbitrary line with  $k + 1$  people in purple and gold hats, with a gold hat at one end and a purple hat at the other.

Target: there is someone in a purple hat next to someone in a gold hat.

By the principle of induction, we have  $P(n)$  for all  $n \geq 2$

# Induction: Hats!

Define  $P(n)$  to be "in every line of  $n$  people with gold and purple hats, with a purple hat at one end and a gold hat at the other, there is a person with a purple hat next to someone with a gold hat"

We show  $P(n)$  for all integers  $n \geq 2$  by induction on  $n$ .

Base Case:  $n = 2$  The line must be just a person with a purple hat and a person with a gold hat, who are next to each other.

Inductive Hypothesis: Suppose  $P(k)$  holds for an arbitrary  $k \geq 2$ .

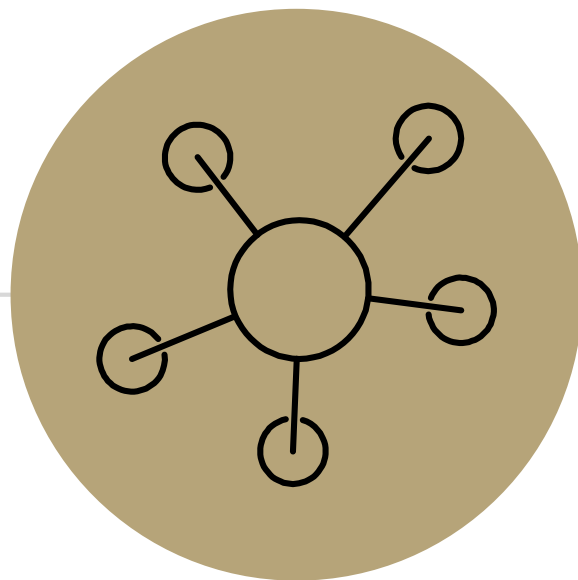
Inductive Step: Consider an arbitrary line with  $k + 1$  people in purple and gold hats, with a gold hat at one end and a purple hat at the other.

Case 1: There is someone with a purple hat next to the person in the gold hat at one end. Then those people are the required adjacent opposite hats.

Case 2: There is a person with a gold hat next to the person in the gold hat at the end. Then the line from the second person to the end is length  $k$ , has a gold hat at one end and a purple hat at the other. Applying the inductive hypothesis, there is an adjacent, opposite-hat wearing pair.

In either case we have  $P(k + 1)$ .

By the principle of induction, we have  $P(n)$  for all  $n \geq 2$



Part 3 of the course!



# Course Outline

Symbolic Logic (training wheels)

Just make arguments in mechanical ways.

Set Theory/Number Theory (bike in your backyard)

Models of computation (biking in your neighborhood)

Still make and communicate rigorous arguments

But now with objects you haven't used before.

- A first taste of how we can argue rigorously about computers.

This week: regular expressions and context free grammars – understand these “simpler computers”

Next week: what these simple computers can do

Then: what simple computers can't do.

Last week: A problem our computers cannot solve.



# Regular Expressions

---

# Regular Expressions

I have a giant text document. And I want to find all the email addresses inside. What does an email address look like?

[some letters and numbers] @ [more letters] . [com, net, or edu]

We want to ctrl-f for a **pattern of strings** rather than a single string

# Languages

A set of strings is called a **language**.

$\Sigma^*$  is a language

“the set of all binary strings of even length” is a language.

“the set of all palindromes” is a language.

“the set of all English words” is a language.

“the set of all strings matching a given **pattern**” is a language.

# Regular Expressions

Every pattern automatically gives you a language.  
The set of all strings that match that pattern.

We'll formalize "patterns" via "regular expressions"

$\epsilon$  is a regular expression. The empty string itself matches the pattern (and nothing else does).

$\emptyset$  is a regular expression. No strings match this pattern.

$a$  is a regular expression, for any  $a \in \Sigma$  (i.e. any character). The character itself matching this pattern.

# Regular Expressions

## Basis:

$\varepsilon$  is a regular expression. The empty string itself matches the pattern (and nothing else does).

$\emptyset$  is a regular expression. No strings match this pattern.

$a$  is a regular expression, for any  $a \in \Sigma$  (i.e. any character). The character itself matching this pattern.

## Recursive

If  $A, B$  are regular expressions then  $(A \cup B)$  is a regular expression  
matched by any string that matches  $A$  or that matches  $B$  [or both]).

If  $A, B$  are regular expressions then  $AB$  is a regular expression.  
matched by any string  $x$  such that  $x = yz$ ,  $y$  matches  $A$  and  $z$  matches  $B$ .

If  $A$  is a regular expression, then  $A^*$  is a regular expression.  
matched by any string that can be divided into 0 or more strings that match  $A$ .

# Regular Expressions

$(a \cup bc)$

$0(0 \cup 1)1$

$0^*$

$(0 \cup 1)^*$

# Regular Expressions

$(a \cup bc)$

Corresponds to  $\{a, bc\}$

$0(0 \cup 1)1$

Corresponds to  $\{001, 011\}$  all length three strings that start with a 0 and end in a 1.

$0^*$

Corresponds to  $\{\varepsilon, 0, 00, 000, 0000, \dots\}$

$(0 \cup 1)^*$

Corresponds to the set of all binary strings.



# More Examples

$(0^*1^*)^*$

$0^*1^*$

$(0 \cup 1)^*(00 \cup 11)^*(0 \cup 1)^*$

$(00 \cup 11)^*$

# More Examples

$(0^*1^*)^*$

All binary strings

$0^*1^*$

All binary strings with any 0's coming before any 1's

$(0 \cup 1)^*(00 \cup 11)^*(0 \cup 1)^*$

This is all binary strings again. Not a "good" representation, but valid.

$(00 \cup 11)^*$

All binary strings where 0s and 1s come in pairs

# Practical Advice

Check  $\varepsilon$  and 1 character strings to make sure they're excluded or included (easy to miss those edge cases).

If you can break into pieces, that usually helps.

"nots" are hard (there's no "not" in standard regular expressions)

But you can negate things, usually by negating at a low-level. E.g. to have binary strings without 00, your building blocks are 1's and 0's followed by a 1

$(01 \cup 1)^*(0 \cup \varepsilon)$  then make adjustments for edge cases (like ending in 0)

Remember  $*$  allows for 0 copies! To say "at least one copy" use  $AA^*$ .

# Regular Expressions In Practice

EXTREMELY useful. Used to define valid "tokens" (like legal variable names or all known keywords when writing compilers/languages)

Used in `grep` to actually search through documents.

```
Pattern p = Pattern.compile("a*b");
Matcher m = p.matcher("aaaaab");
boolean b = m.matches();
```

`^` start of string

`$` end of string

`[01]` a 0 or a 1

`[0-9]` any single digit

`\.` period    `\,` comma    `\-` minus

`.` any single character

`ab` a followed by b                    **(AB)**

`(a|b)` a or b    **(A ∪ B)**

`a?` zero or one of a                    **(A ∪ ε)**

`a*` zero or more of a                    **A\***

`a+` one or more of a                    **AA\***

e.g. `^[\\-+]?[0-9]*(\\.|\\,)?[0-9]+$`

General form of decimal number e.g. 9.12 or -9,8 (Europe)

# Regular Expressions In Practice

When you only have ASCII characters (say in a programming language)

| usually takes the place of  $\cup$

? (and perhaps creative rewriting) take the place of  $\varepsilon$ .

E.g.  $(0 \cup \varepsilon)(1 \cup 10)^*$  is  $0?(1|10)^*$

# A Final Vocabulary Note

Not everything can be represented as a regular expression.

E.g. “the set of all palindromes” is not the language of any regular expression.

Some programming languages define features in their “regexes” that can’t be represented by our definition of regular expressions.

Things like “match this pattern, then have exactly that **substring** appear later.

So before you say “ah, you can’t do that with regular expressions, I learned it in 311!” you should make sure you know whether your language is calling a more powerful object “regular expressions”.

But the more “fancy features” beyond regular expressions you use, the slower the checking algorithms run, (and the harder it is to force the expressions to fit into the framework) so this is still very useful theory.