CSC236: Introduction to the Theory of Computation

Harry Sha

Summer 2023

CSC 236 Lecture 1: Welcome to CS Theory

Harry Sha

May 10, 2023

Contents

Logistics

An Invitation to Theory

A Motivating Problem

Functions

Properties of Functions
- Injective Functions
- Surjective Functions
- Bijective Functions

Cantor's Theorem

Programs and Problems

Logistics

An Invitation to Theory

A Motivating Problem

Functions

Properties of Functions
- Injective Functions
- Surjective Functions
- Bijective Functions

Cantor's Theorem

Programs and Problems

Communication

I will make all announcements through Ed (which will send you an email notification).

You can find all course material on the course website:

https://www.cs.toronto.edu/~shaharry/csc236/

Syllabus

All the course policies can be found on the syllabus, so make sure you read it carefully!

Prerequisites

CSC165 (or equivalent)

- Sets
- Graphs
- Proofs
- Mathematical Logic
- Asymptotics

Here are some course notes by David Liu and Toniann Pitassi if you'd like to review something!

Grading Breakdown

1.) TA check-ins: 10% ($5 \times 2\%$)
2.) Midterm: 40%
3.) Final: 50%
4.) Ed Contributor Prize 2%

HW + Check Ins (10%)

- There will be 5 HWs throughout the semester and one check in per HW.
- You will get full credit for check ins if you manage to convince your TAs that you made an honest attempt at trying to solve the HW problems.
- See the guide to hw, and guide to check ins for more information and tips for the HW and the check ins!

Exams (40% + 50%)

- At least 20% of the exam with be based on the HW problems.
- The midterm will June 28, 6-9PM at EX100.
- The final will be sometime August 17 - 25.

Support

Sign up for Ed! Please ask and answer questions!

Come to office hours!

The TAs and I are here for you, so please don't hesitate to reach out. The best way to get in touch is through Ed.

Course overview

Part 1: Mathematical tools.

Part 2: Algorithm correctness and runtime.

Part 3: Finite automata.

Logistics

An Invitation to Theory

A Motivating Problem

Functions

Properties of Functions
- Injective Functions
- Surjective Functions
- Bijective Functions

Cantor's Theorem

Programs and Problems

Welcome!

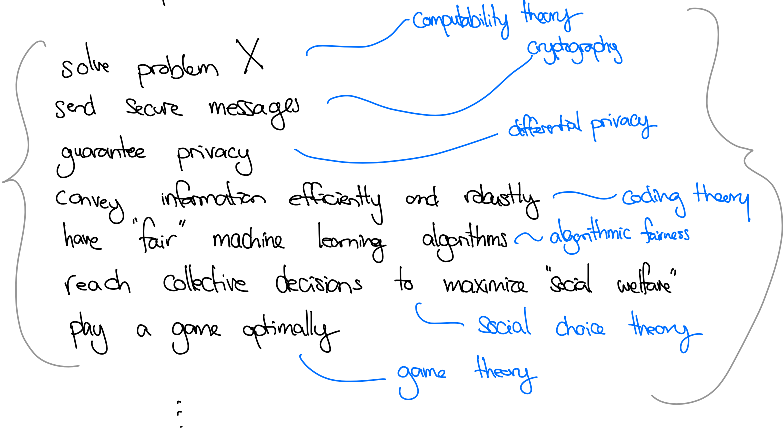
Questions we care about in Theory

Is it possible to ....

- solve problem X
- send secure messages
- guarantee privacy
- convey information efficiently and robustly
- have "fair" machine learning algorithms
- reach collective decisions to maximize "social welfare"
- play a game optimally

⋮

Questions we care about in Theory



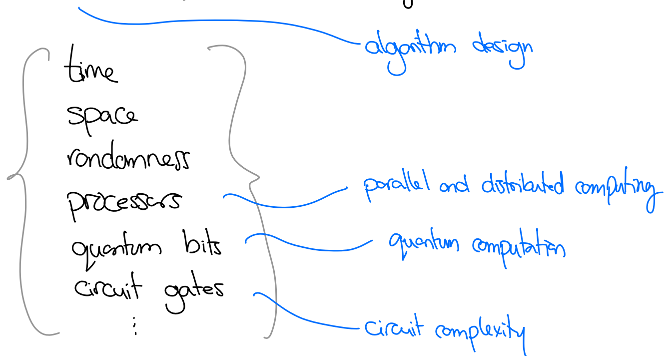
Questions we care about in Theory

If so, how? Also, how much/many

- time
- space
- randomness
- processors
- quantum bits
- circuit gates
- ⋮

do we need?

Questions we care about in Theory



The sell

For programmers:

- Rigorously analyze your programs.
- Model real world problems as well studied problems in theory and apply known algorithms.

The sell

For programmers:

- Rigorously analyze your programs.
- Model real world problems as well studied problems in theory and apply known algorithms.

In general:

- There are no compiler issues in Theory Land: problems are distilled to the core puzzle.
- CS Theory is a fascinating and new field! There are lots of unknowns, and breakthroughs happen very often.

Logistics

An Invitation to Theory

A Motivating Problem

Functions

Properties of Functions
- Injective Functions
- Surjective Functions
- Bijective Functions

Cantor's Theorem

Programs and Problems

Are there problems computers can't solve?

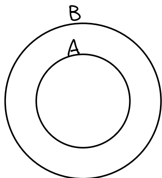
Set theory review

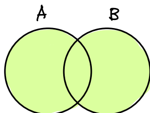
$A \subseteq B$, $A \subseteq B$ ← "subset" (\subseteq)

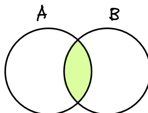
$A \cup B$ ← "union" (\cup)

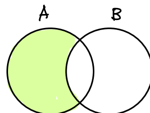
$A \cap B$ ← "intersection" (\cap)

$A \setminus B$ ← "difference" (\setminus)









$\forall x. (x \in A \Rightarrow x \in B)$

"emptyset" (\emptyset)

$\{\} = \emptyset$

$\wp(A) = \{B : B \subseteq A\}$

↑ "powerset of A" (\wp)

i.e. $\wp(A)$ is the set containing all the subsets of A

Set theory review

$$A \times B = \{(a,b) : a \in A,\ b \in B\}$$

↑ "A times B" (\times)

$$A^n = \underbrace{A \times A \times \dots \times A}_{n} = \{(a_1, a_2, \dots, a_n) : \forall i \in [n], a_i \in A\}.$$

"A to the n"

Logistics

An Invitation to Theory

A Motivating Problem

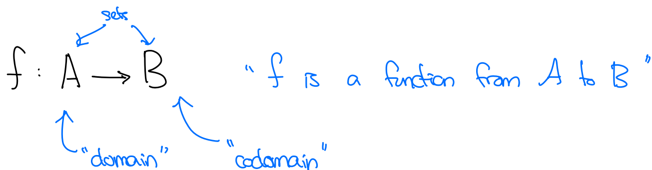
Functions

Properties of Functions
- Injective Functions
- Surjective Functions
- Bijective Functions

Cantor's Theorem

Programs and Problems

Functions

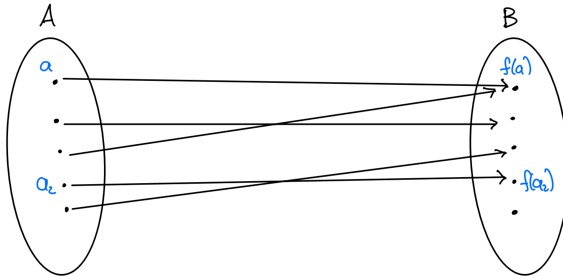


Think of the domain as a set of inputs and the codomain as a set of outputs.

$$\forall a \in A. \left(f(a) \in B \right)$$

Functions

$f : A \to B$

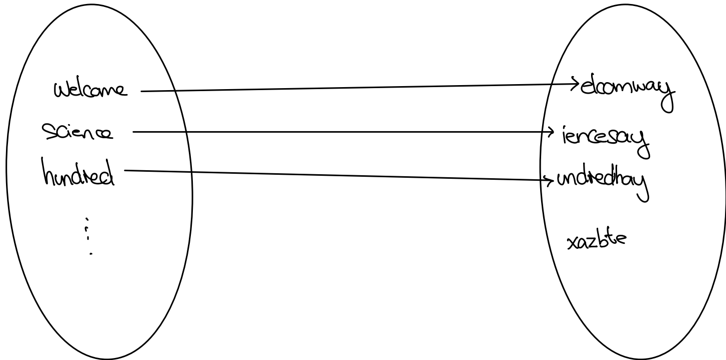


Everything in A is mapped to something in B.

Not everything in B needs to be mapped to.

Examples

PigLatin : English ⟶ Strings



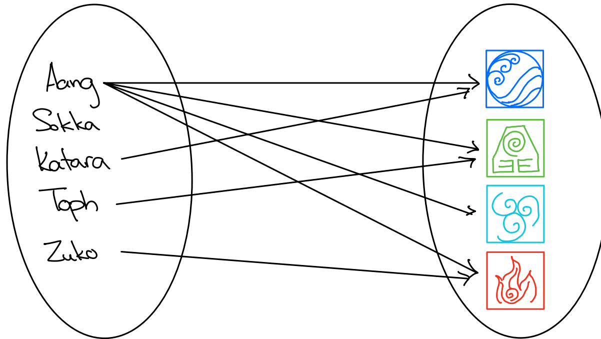
Examples

```
def product(xs: List[int]) -> int:
    accumulator = 1
    for x in xs:
        accumulator *= x
    return accumulator
```

What is the domain/codomain here?

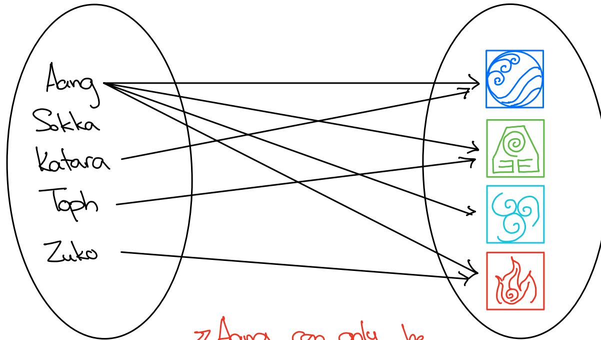
Is this a function?

Bendr : Gaang → Elements.



Is this a function?

Bender : Gaang → Elements.

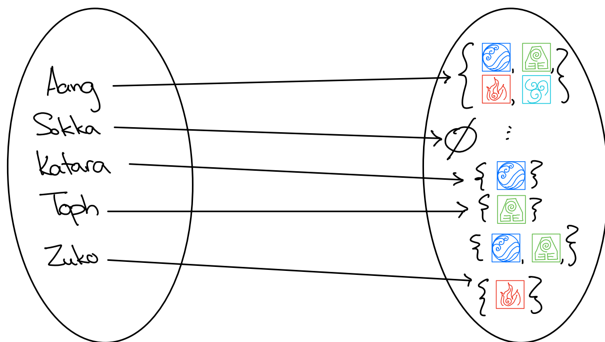


Not a function!

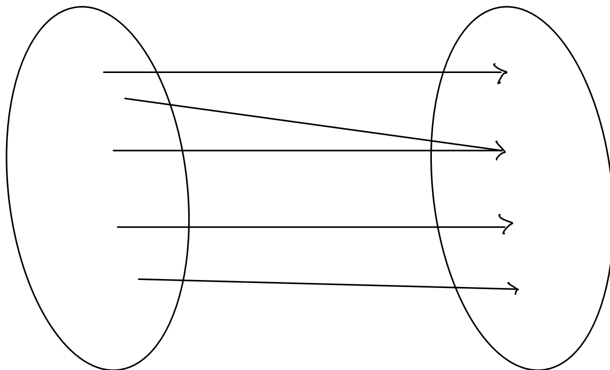
Aang can only be mapped to one thing.

Sokka must be mapped to something!

A fix - change the codomain!



A function that we're interested in



Logistics

An Invitation to Theory

A Motivating Problem

Functions

Properties of Functions
- Injective Functions
- Surjective Functions
- Bijective Functions

Cantor's Theorem

Programs and Problems

Injective

A function is injective if nothing in the codomain is hit more than once. Formally,

Definition (Injective)

A function $f : A \to B$ is injective if

$$\forall x, y \in A.(x \neq y \implies f(x) \neq f(y))$$

"If the inputs are different, the outputs are different"

Injective

A function is injective if nothing in the codomain is hit more than once. Formally,

Definition (Injective)

A function $f : A \to B$ is injective if

$$\forall x, y \in A.(x \neq y \implies f(x) \neq f(y))$$

"If the inputs are different, the outputs are different"

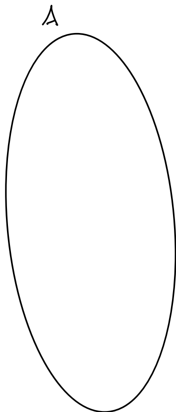
Sometimes, the equivalent contrapositive is easier to work with

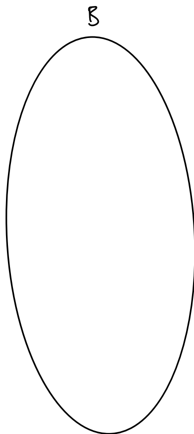
$$\forall x, y \in A.(f(x) = f(y) \implies x = y)$$

"If the outputs are the same, the inputs are the same"

$f : A \to B$ is injective if
$\forall x, y \in A.(x \neq y \implies f(x) \neq f(y))$

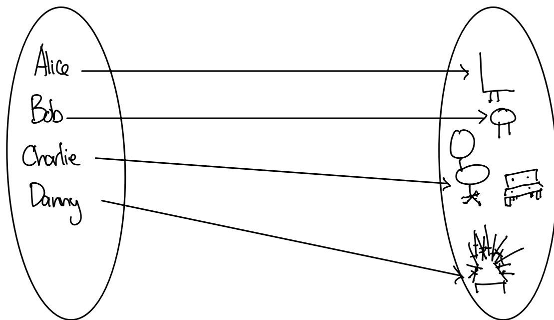
$f : A \to B$





Example - People and Chairs

$f : A \to B$ is injective if
$\forall x, y \in A.(x \neq y \implies f(x) \neq f(y))$

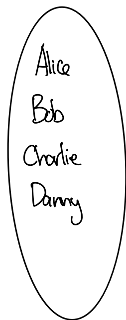


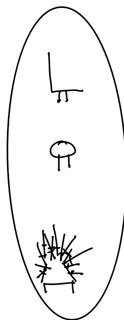
"Different people $\implies$ different chairs"

Example - Musical Chairs

$f : A \to B$ is injective if
$\forall x, y \in A.(x \neq y \implies f(x) \neq f(y))$

Is there an injective function between these two sets?





Example - Musical Chairs

$f : A \to B$ is injective if
$\forall x, y \in A.(x \neq y \implies f(x) \neq f(y))$

Is there an injective function between these two sets?

No! If there are fewer chairs than people, no matter how you assign people to chairs, at least one person will have to share. I.e., someone goes out after every round of musical chairs. This phenomenon has a special name...

The Pigeonhole Principle

$f : A \to B$ is injective if $\forall x, y \in A.(x \neq y \implies f(x) \neq f(y))$

Theorem (The Pigeonhole Principle)

Let A, B be finite sets where $|A| > |B|$. Then there is no injective function $f : A \to B$.

Think of A as a set of pigeons and B as a set of pigeonholes. The pigeonhole principle is a fancy way of saying that if you have more pigeons than you have pigeonholes, no matter how you assign pigeons to pigeonholes, some pigeonhole will have at least two pigeons.

We will see more of this in tutorial!

Example - Hilbert's Hotel

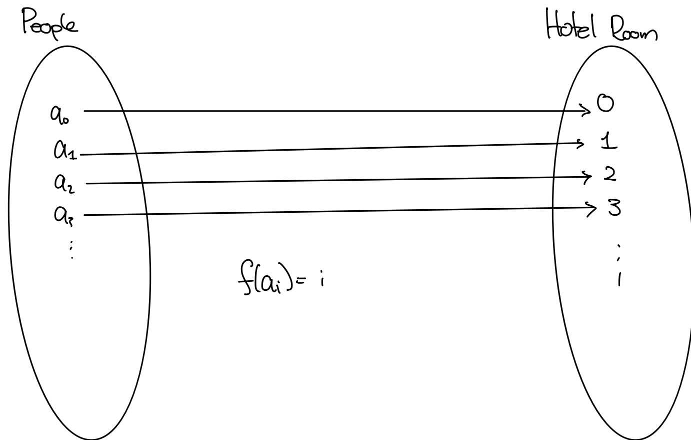
$f : A \to B$ is injective if $\forall x, y \in A.(x \neq y \implies f(x) \neq f(y))$

Imagine you're working at a unique hotel. The hotel has an infinite number of rooms. To be precise, it has one (single person) room for every natural number $0, 1, 2, \dots$

Your job is to assign customers to rooms. Just when you thought your job was easy, a bus containing an infinite number of people, let's call them $a_0, a_1, a_2, \dots$ shows up and requests rooms. Assume the hotel is empty to start. How do you assign the customers to rooms? Since only one person can stay in each room, we need the assignment to be injective.

Example - Hilbert's Hotel

$f : A \to B$ is injective if $\forall x, y \in A.(x \neq y \implies f(x) \neq f(y))$



Example - Hilbert's Hotel

$f : A \to B$ is injective if $\forall x, y \in A.(x \neq y \implies f(x) \neq f(y))$

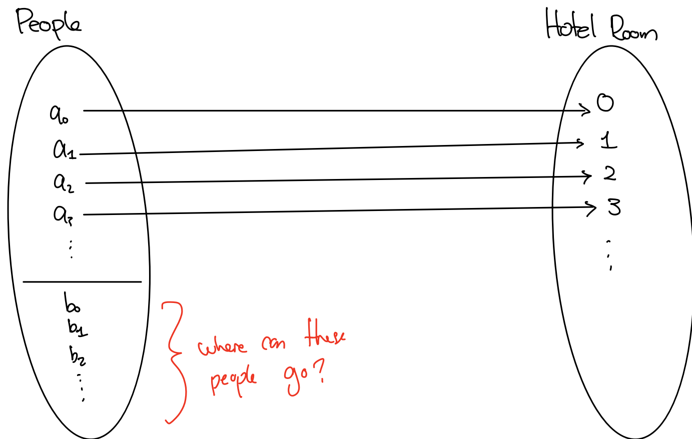
Nice! You managed to assign an infinite number of people to rooms! However, another bus arrives, and it again contains an infinite number of people. Let's call them $b_0, b_1, b_2 \ldots$.

You look at the rooms. Currently, each room is occupied! In particular, room number i is taken by customers a_i.

However, eager to impress your boss, you try to think of a way to do the impossible - fit an infinite number of people into an already filled hotel. So how do you do it?

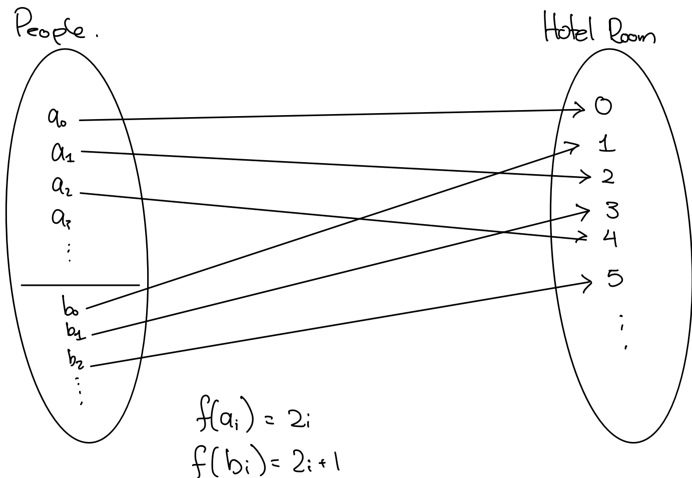
Example - Hilbert's Hotel

$f : A \to B$ is injective if $\forall x, y \in A.(x \neq y \implies f(x) \neq f(y))$



Example - Hilbert's Hotel

$f : A \to B$ is injective if $\forall x, y \in A.(x \neq y \implies f(x) \neq f(y))$



Proof: f is injective

$$f(x) = \begin{cases} 2i & x = a_i \\ 2i + 1 & x = b_i \end{cases}$$

$f : A \to B$ is injective if
$\forall x, y \in A.(x \neq y \implies f(x) \neq f(y))$

Let x, y be customers such that $x \neq y$. We'll show that $f(x) \neq f(y)$. There are several cases.

Proof: f is injective

$$f(x) = \begin{cases} 2i & x = a_i \\ 2i + 1 & x = b_i \end{cases}$$

$f : A \to B$ is injective if
$\forall x, y \in A.(x \neq y \implies f(x) \neq f(y))$

Let x, y be customers such that $x \neq y$. We'll show that $f(x) \neq f(y)$. There are several cases.

- Suppose x and y were on different busses. WLOG, assume x was on the first and y was on the second. Then $f(x) \neq f(y)$ since $f(x)$ is even and $f(y)$ is odd.

Proof: f is injective

$$f(x) = \begin{cases} 2i & x = a_i \\ 2i+1 & x = b_i \end{cases}$$

$f : A \to B$ is injective if
$\forall x, y \in A.(x \neq y \implies f(x) \neq f(y))$

Let x, y be customers such that $x \neq y$. We'll show that $f(x) \neq f(y)$. There are several cases.

- Suppose x and y were on different busses. WLOG, assume x was on the first and y was on the second. Then $f(x) \neq f(y)$ since $f(x)$ is even and $f(y)$ is odd.
- If x and y were both on the first bus, then since $x \neq y$, $x = a_i$ and $y = a_j$ for some $i \neq j$. Thus $2i \neq 2j$ and $f(x) \neq f(y)$ as required.

Proof: f is injective

$$f(x) = \begin{cases} 2i & x = a_i \\ 2i+1 & x = b_i \end{cases}$$

$f : A \to B$ is injective if
$\forall x, y \in A.(x \neq y \implies f(x) \neq f(y))$

Let x, y be customers such that $x \neq y$. We'll show that $f(x) \neq f(y)$. There are several cases.

- Suppose x and y were on different busses. WLOG, assume x was on the first and y was on the second. Then $f(x) \neq f(y)$ since $f(x)$ is even and $f(y)$ is odd.
- If x and y were both on the first bus, then since $x \neq y$, $x = a_i$ and $y = a_j$ for some $i \neq j$. Thus $2i \neq 2j$ and $f(x) \neq f(y)$ as required.
- The case where x and y were both the second bus is similar.

Surjective

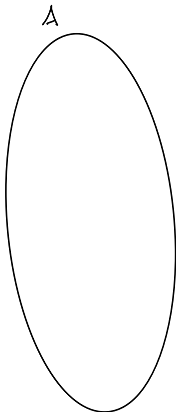
A function is surjective if everything in the codomain is hit.

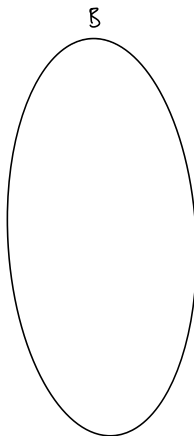
Formally, $f : A \to B$ is surjective if

$$\forall b \in B. \exists a \in A. (f(a) = b)$$

f is surjective if
$\forall b \in B. \exists a \in A. (f(a) = b)$.

$f : A \to B$





Example

f is surjective if
$\forall b \in B. \exists a \in A. (f(a) = b)$.

Is f defined by $f(n) = n$ surjective?

Example

f is surjective if
$\forall b \in B.\exists a \in A.(f(a) = b)$.

Is f defined by $f(n) = n$ surjective?

It depends on the domain/codomain! For example, $f : \mathbb{N} \to \mathbb{N}$ defined by $f(n) = n$ is surjective, but $f : \mathbb{N} \to \mathbb{R}$ defined by $f(n) = n$ is not!

It's essential to **always specify the domain and codomain when defining a function**.

Bijective

A function is bijective if everything in the codomain is hit exactly once. Formally,

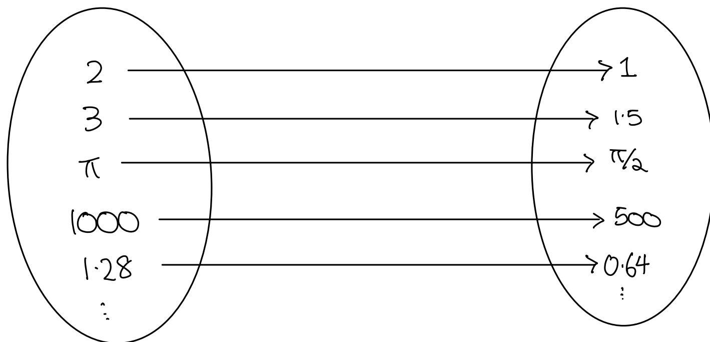
$f : A \to B$ is bijective if f is **both injective and surjective**.

Examples

$$f : \mathbb{R} \to \mathbb{R}$$

(\mathbb{R}) ← 'real numbers'

$$f(x) = x/2$$



Proof - Half is Bijective

injective: $\forall x, y \in A.(x \neq y \implies f(x) \neq f(y))$
surjective: $\forall b \in B. \exists a \in A.(f(a) = b)$.

We'll show $f : \mathbb{R} \to \mathbb{R}$ defined by $f(x) = x/2$ is a bijection. To do so, we'll show that f is both injective and surjective.

Proof - Half is Bijective

injective: $\forall x, y \in A.(x \neq y \implies f(x) \neq f(y))$
surjective: $\forall b \in B.\exists a \in A.(f(a) = b)$.

We'll show $f : \mathbb{R} \to \mathbb{R}$ defined by $f(x) = x/2$ is a bijection. To do so, we'll show that f is both injective and surjective.

Injective. Let $a, b \in \mathbb{R}$ and assume $f(a) = f(b)$. By the definition of f, this implies $a/2 = b/2$, which in turn implies $a = b$. Thus f is injective.

Proof - Half is Bijective

injective: $\forall x, y \in A.(x \neq y \implies f(x) \neq f(y))$
surjective: $\forall b \in B. \exists a \in A.(f(a) = b)$.

We'll show $f : \mathbb{R} \to \mathbb{R}$ defined by $f(x) = x/2$ is a bijection. To do so, we'll show that f is both injective and surjective.

Injective. Let $a, b \in \mathbb{R}$ and assume $f(a) = f(b)$. By the definition of f, this implies $a/2 = b/2$, which in turn implies $a = b$. Thus f is injective.

Surjective. Let $b \in \mathbb{R}$. To show f is surjective, we need to find some number $a \in \mathbb{R}$ such that $f(a) = b$. We claim that $a = 2b$ does the trick. We have

$$f(a) = f(2b) = 2b/2 = b,$$

and thus f is surjective.

Summary of definitions

Let $f : A \to B$ be a function.

f is...	if $\forall b \in B$, b is hit...	Formally...
Injective	1 or 0 times	$\forall x, y \in A.(x \neq y \implies f(x) \neq f(y))$
Surjective	at least 1 time	$\forall b \in B.\exists a \in A.(f(a) = b)$
Bijective	exactly 1 time	Injective and Surjective

$b \in B$ is hit k times by f if there are k distinct $a \in A$ are such that $f(a) = b$. I.e.

$$|\{a \in A : f(a) = b\}| = k.$$

Logistics

An Invitation to Theory

A Motivating Problem

Functions

Properties of Functions
- Injective Functions
- Surjective Functions
- Bijective Functions

Cantor's Theorem

Programs and Problems

Cantor's Theorem

- f is surjective if $\forall b \in B. \exists a \in A. (f(a) = b)$
- $\wp(A) = \{B : B \subseteq A\}$

Theorem (Cantor's Theorem)

For any set A, there is no surjection between A and $\wp(A)$

Proof of Cantor's Theorem

f is surjective if
$\forall b \in B. \exists a \in A. (f(a) = b)$.

Let $f: A \to \mathcal{P}(A)$ be any function.

	a_1	a_2	a_3	a_4	...
$f(a_1)$:	✓	×	✓	✓	...
$f(a_2)$:	×	×	✓	×	...
$f(a_3)$:	×	×	×	×	...
$f(a_4)$:	✓	✓	✓	×	...
⋮	⋮	⋮	⋮	⋮	

× here means $a_1 \notin f(a_2)$.

✓ here means $a_3 \in f(a_4)$

Proof of Cantor's Theorem

f is surjective if
$\forall b \in B. \exists a \in A. (f(a) = b)$.

	a_1	a_2	a_3	a_4	...
$f(a_1)$:	✗	✗	✓	✓	...
$f(a_2)$:	✗	✓	✓	✗	...
$f(a_3)$:	✗	✗	✓	✗	...
$f(a_4)$:	✓	✓	✓	✓	...
⋮	⋮	⋮	⋮	⋮	

D.

$D = \{ a \in A : a \notin f(a) \}$

Proof of Cantor's Theorem

surjective:
$\forall b \in B. \exists a \in A.(f(a) = b)$.
$D = \{a \in A : a \notin f(a)\}$

Let $f : A \to \wp(A)$ be any function. Let $D = \{a \in A : a \notin f(a)\}$ and note that $D \subseteq A$, and hence $D \in \wp(A)$. To show f is not surjective, we'll show that in particular, there is no $a \in A$ such that $f(a) = D$.

Proof of Cantor's Theorem

surjective:
$\forall b \in B. \exists a \in A. (f(a) = b)$.
$D = \{a \in A : a \notin f(a)\}$

Let $f : A \to \wp(A)$ be any function. Let $D = \{a \in A : a \notin f(a)\}$ and note that $D \subseteq A$, and hence $D \in \wp(A)$. To show f is not surjective, we'll show that in particular, there is no $a \in A$ such that $f(a) = D$.

By contradiction, assume $D = f(a)$ for some $a \in A$. There are two cases, either $a \in D$ or $a \notin D$.

- If $a \in D$, then $a \notin f(a)$ by the definition of D. But $f(a) = D$, so $a \notin D$, which is a contradiction.
- Otherwise $a \notin D$, then $a \in f(a)$ but $f(a) = D$ by assumption, so $a \in D$, which is again a contradiction.

Since we reached a contradiction in both cases, our assumption must have been false. Thus, f is not surjective.

Logistics

An Invitation to Theory

A Motivating Problem

Functions

Properties of Functions
- Injective Functions
- Surjective Functions
- Bijective Functions

Cantor's Theorem

Programs and Problems

Are the problems computers can't solve?

Let's apply our knowledge of functions to answer the question!

Formalizing the question - Problems

Let $\mathrm{Strings}$ be the set of all possible strings. For each subset $A \subseteq \mathrm{Strings}$ (each $A \in \wp(\mathrm{Strings})$), there is the problem of determining whether or not a given input is in A or not in A.

Formalizing the question - Problems

Let $\mathrm{Strings}$ be the set of all possible strings. For each subset $A \subseteq \mathrm{Strings}$ (each $A \in \wp(\mathrm{Strings})$), there is the problem of determining whether or not a given input is in A or not in A.

For example

$$A = \{w \in \mathrm{Strings} : w \text{ is a palindrome}\},$$

Formalizing the question - Problems

Let $\mathrm{Strings}$ be the set of all possible strings. For each subset $A \subseteq \mathrm{Strings}$ (each $A \in \wp(\mathrm{Strings})$), there is the problem of determining whether or not a given input is in A or not in A.

For example

$$A = \{w \in \mathrm{Strings} : w \text{ is a palindrome}\},$$

or

$$A = \{w : w \text{ is a C program with no syntax errors}\}.$$

Formalizing the question - Problems

Let $\mathrm{Strings}$ be the set of all possible strings. For each subset $A \subseteq \mathrm{Strings}$ (each $A \in \wp(\mathrm{Strings})$), there is the problem of determining whether or not a given input is in A or not in A.

For example

$$A = \{w \in \mathrm{Strings} : w \text{ is a palindrome}\},$$

or

$$A = \{w : w \text{ is a C program with no syntax errors}\}.$$

Let's just consider problems of this type. So set

$$\mathrm{Problems} = \wp(\mathrm{Strings}).$$

Formalizing the question - Programs

For concreteness, let's say, a program P solves a problem $A \subseteq \text{Strings}$, if $\forall w \in \text{Strings}$,

$$w \in A \iff P \text{ run on input } w \text{ prints 1 and nothing else}$$

Identify every program with its source code (a string), so $\text{Programs} \subseteq \text{Strings}$.

Formalizing the question - Solves

Let $\mathrm{Solves} : \mathrm{Programs} \to \wp(\mathrm{Strings})$ be the function that maps each program to the problem it solves. Since each program solves at most one problem, this function is well defined.

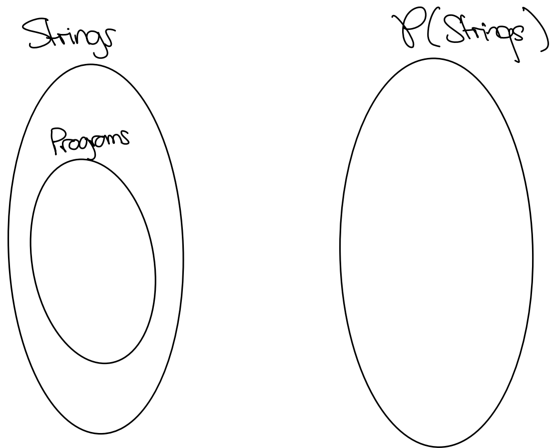
Our question about whether or not computers can solve all problems is the question of whether or not Solves is ...

Formalizing the question - Solves

Let $\mathrm{Solves} : \mathrm{Programs} \to \wp(\mathrm{Strings})$ be the function that maps each program to the problem it solves. Since each program solves at most one problem, this function is well defined.

Our question about whether or not computers can solve all problems is the question of whether or not Solves is surjective!

Proof (Picture)



Proof - There is a problem that computers can't solve

The answer to our question is no - computers can not solve all problems! We will prove this by showing that Solves is not surjective.

Proof - There is a problem that computers can't solve

The answer to our question is no - computers can not solve all problems! We will prove this by showing that Solves is not surjective.

By contradiction, assume $\mathrm{Solves} : \mathrm{Programs} \to \wp(\mathrm{Strings})$ is surjective. The strategy will be to construct a surjective function from $\mathrm{Strings} \to \wp(\mathrm{Strings})$ which contradicts Cantor's Theorem.

Proof - There is a problem that computers can't solve

The answer to our question is no - computers can not solve all problems! We will prove this by showing that Solves is not surjective.

By contradiction, assume $\mathrm{Solves} : \mathrm{Programs} \to \wp(\mathrm{Strings})$ is surjective. The strategy will be to construct a surjective function from $\mathrm{Strings} \to \wp(\mathrm{Strings})$ which contradicts Cantor's Theorem.

Let $f : \mathrm{Strings} \to \wp(\mathrm{Strings})$ be defined as follows

$$f(w) = \begin{cases} \mathrm{Solves}(w) & w \in \mathrm{Programs} \\ \emptyset & \text{else} \end{cases}$$

Proof - Cont.

$$f(w) = \begin{cases} \mathrm{Solves}(w) & w \in \mathrm{Programs} \\ \emptyset & \text{else} \end{cases}$$

We claim that f is surjective. Let $b \in \wp(Strings)$, since Solves is surjective, there is some $a \in \mathrm{Programs}$ such that $\mathrm{Solves}(a) = b$. Since $\mathrm{Programs} \subseteq \mathrm{Strings}$, $a \in \mathrm{Strings}$. By the definition of f,

$$f(a) = \mathrm{Solves}(a) = b.$$

Thus, f is surjective, contradicting Cantor's Theorem.

Since we have reached a contradiction, the assumption must have been false. Therefore, Solves is not surjective.

There is a problem that computers can't solve

What are your questions?

FAQ

- “How many problems can/can’t computers solve?”
- “What is a particular problem that we can’t solve with computers”
- “How can we tell if a problem can or can’t be solved by computers?”
- “Some problems can be solved and other can not - so some problems are computationally “harder” than others. Are there more ways to compare how computationally hard problems are?”

FAQ

- "How many problems can/can't computers solve?"
- "What is a particular problem that we can't solve with computers"
- "How can we tell if a problem can or can't be solved by computers?"
- "Some problems can be solved and other can not - so some problems are computationally "harder" than others. Are there more ways to compare how computationally hard problems are?"

Answer: Take CSC448 and CSC463 :)

Additional Notes

- 'hits' in the definitions of injective/surjective/bijective is not standard terminology. The standard way to express the same meaning is to use the word 'preimage.' In your proofs you should use the formal FOL definitions.
- The visual part of the proof of Cantor's Theorem is a little misleading. It makes an additional assumption that you can list the elements of A using the natural numbers ($\mathbb{N}$) (this property is called 'countable'). But this is not true for all sets! For example, apply Cantor's Theorem to $\mathbb{N}$ to show that $\wp(\mathbb{N})$ can not be listed using the natural numbers!
- Cantor's Theorem is fundamental and deep. In particular, it implies that there are bigger and smaller infinities (!). I.e., the powerset of an infinite set is strictly bigger. The powerset of that set is strictly bigger again, and so on. If this interests you, take a class on Set Theory!

CSC 236 Lecture 2: Graphs

Harry Sha

May 17, 2023

Contents

Definitions

Modelling with Graphs
- Problem 1. Matching
- Problem 2. Shortest Path
- Problem 3. The Traveling Salesman

Trees - A special type of graph
- Problem 4. Minimum Spanning Tree

The goal

The goal of today's lecture is to have you see graphs everywhere in the world.

Definitions

Modelling with Graphs
Problem 1. Matching
Problem 2. Shortest Path
Problem 3. The Traveling Salesman

Trees - A special type of graph
Problem 4. Minimum Spanning Tree

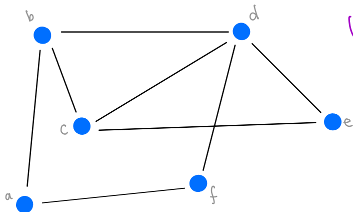
Definitions

A graph $G = (V, E)$ is a pair of sets (V, E), where V is a set of vertices and E is a set of pairs of vertices.

If E is a set of unordered pairs, the graph is called undirected and if the E is a set of ordered pairs, the graph is called directed.

Examples

$V = \{a, b, c, d, e, f\}$

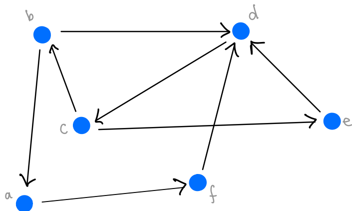


Undirected

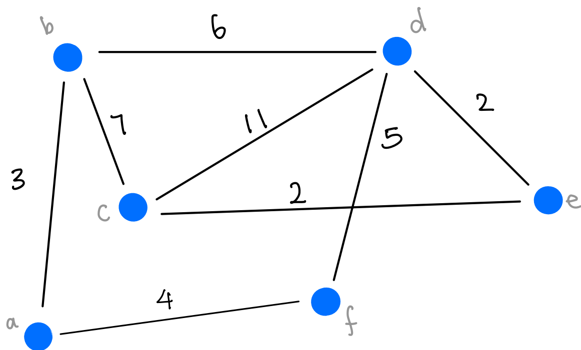
$E = \{ \{b,d\}, \{a,b\}, \{d,c\}, \{d,e\}, \{e,c\}, \{a,f\}, \{b,c\}, \{f,d\} \}$

Directed

$E = \{ (b,a), (a,f), (c,b), (d,c), (b,d), (e,d), (c,e), (f,d) \}$



Weights



Sometimes edges may have an associated weights. Formally, we can define a function $w : E \to \mathbb{R}$ where $w(\{u, v\})$ is the weight of the edge $\{u, v\}$.

Seeing graphs everywhere

Seeing graphs everywhere

- Functions
- Binary relations
- Maps
- Web links
- Tournament brackets
- Game trees
- ...

Definitions

Modelling with Graphs
- Problem 1. Matching
- Problem 2. Shortest Path
- Problem 3. The Traveling Salesman

Trees - A special type of graph
- Problem 4. Minimum Spanning Tree

An Important Skill

The ability to model problems in real life as graph problems is super useful.

You will study algorithms to solve graph problems in CSC373.

Sometimes modelling the problem is enough since there are libraries that implement the algorithms for you!

One such library is the networkx (Python). We will see some examples...

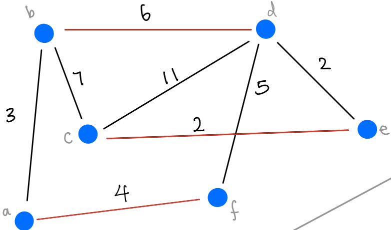
Matching

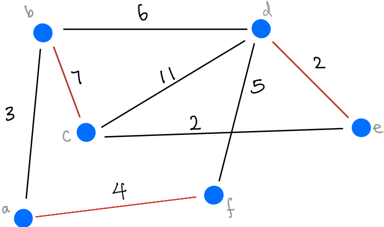
Let $G = (V, E)$ be a graph. A matching $M \subseteq E$ is a subset of edges that do not share any endpoints. I.e. every vertex appears in at most one edge in M.

A matching is perfect if every vertex appears exactly once in the matching.

If each edges has a weight, then the weight of a matching is the sum of the weights of edges in M.

Examples





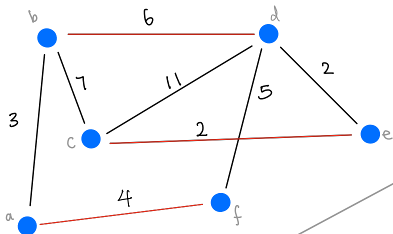
Matching problem

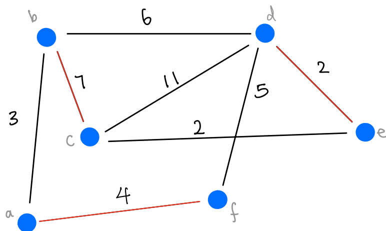
Input: A graph $G = (V, E)$.

Output: A matching $M \subseteq E$.

Usually we want $|M|$ to be as large as possible. Sometimes, we want to maximize/minimize the weight of the matching as well.

How would you find the maximum matching?





Example: Matchings in the real world

Here's the set up: There are n students in a class that need to be matched with partners. Each student fills out a form to indicate a list of times they are available to work and if they prefer to work in-person or virtually.

Let a, b be any two distinct students. a and b are incompatible if they don't share any available times. Otherwise, the compatibility score for a and b is the number of timeslots in which they overlap in their availability plus one if they additionally have the same preference to work in person or virtually.

Your task is to find a pairing with the following properties

- As many students should be matched as possible.
- Aim to have high compatibility score.

Modelling as a matching problem.

- What are the vertices of the graph?
- What are the edges of the graph?
- What are the edge weights?
- What properties of the matching do we want?

Modelling as a matching problem.

- Students.
- There's an edge if they have non-zero compatibility score.
- Compatibility score.
- We want the largest matching (match as many students as possible) with the highest total compatibility!

Coding (?!)

Screenshots of Coding[1]

Username	Virtual	Weekday afternoon	Weekend evening	Weekday evening	Weekend morning	Weekend afternoon	Weekday morning
Mildred Havercroft	0	1	1	0	0	0	0
Melody Mastroianni	1	0	1	1	1	1	0
Diana Williams	0	1	1	0	1	1	0
Kim Massaro	0	1	1	0	0	1	0
Larry Vass	1	1	1	1	0	1	0
Ethel Roberts	1	1	1	0	1	1	0
Laura Morello	0	1	1	1	0	1	0
Paula Mercado	1	0	1	1	1	1	0
Miriam Hurst	1	1	1	1	1	1	1
Mary Hutto	1	0	1	1	1	1	0

Note: Real data from this class with fake names.

[1] Screenshots are from when I taugh last summer, names are replaced with fake names

Screenshots of Coding[1]

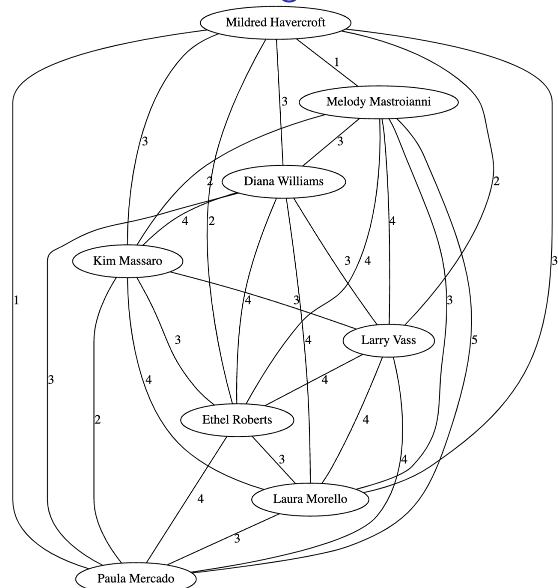
```
import networkx as nx
import itertools

def table_to_graph(data):
  data = data.head(8) # remove line for full example
  graph = nx.Graph()
  edge_list = []

  # for every pair of distinct students...
  for (user1, p1), (user2, p2) in itertools.combinations(data.iterrows(), 2):
    weight = sum(p1[TIMES].values & p2[TIMES].values)
    if weight > 0:
      weight += int(p1.Virtual == p2.Virtual)
      edge_list.append((user1, user2, weight))
  graph.add_weighted_edges_from(edge_list)
  return graph
```

[1] Screenshots are from when I taugh last summer, names are replaced with fake names

Screenshots of Coding[1]



[1] Screenshots are from when I taugh last summer, names are replaced with fake names

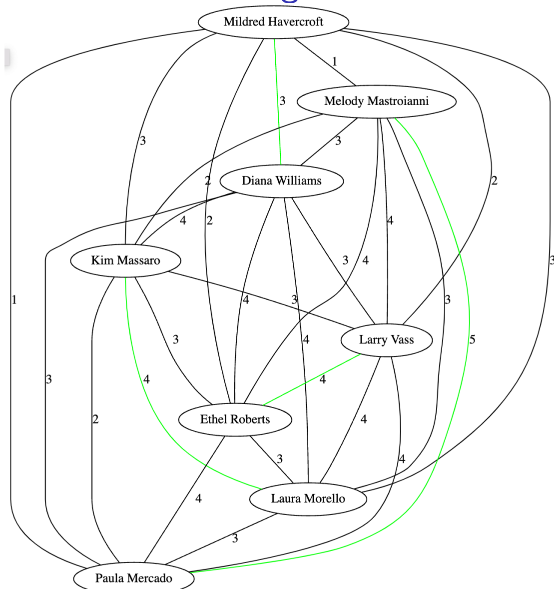
Screenshots of Coding[1]

```
matching = nx.max_weight_matching(g, maxcardinality=True)

print(matching)
plot_graph_with_matching(g, matching)
```

[1] Screenshots are from when I taugh last summer, names are replaced with fake names

Screenshots of Coding[1]



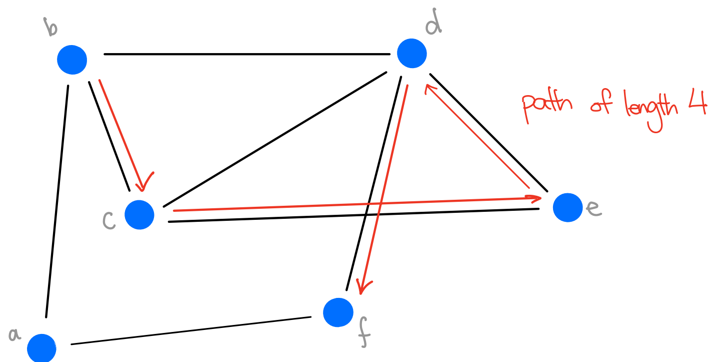
[1] Screenshots are from when I taugh last summer, names are replaced with fake names

Paths

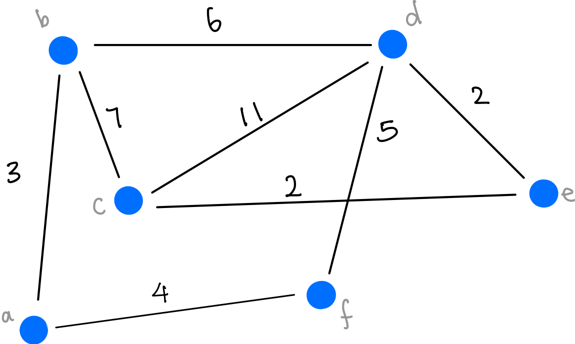
Let $G = (V, E)$ be a graph.

- Two vertices $u, v \in V$ are adjacent if $\{u, v\} \in E$
- A sequence of distinct vertices $(v_1, ..., v_n)$ is a path from v_1 to v_n if for every $i \in \{1, ..., n-1\}$, v_i and v_{i+1} are adjacent. The length of the path is the number of edges in the path.

Example



Example



What's the shortest path from a to e?

Path Finding Problem

Input: A graph $G = (V, E)$ and two vertices $u, v \in V$.

Output: A path from u to v in G. I.e. a sequence of vertices $(v_1, v_2, ..., v_n)$ where $v_1 = u$ and $v_n = v$.

Typically, we want to find the path with the smallest length. If each edge has a weight we also may want to find the path with the smallest total weight.

Wikipedia Game

WIKIPEDIA
The Free Encyclopedia

Main page
Contents
Current events
Random article
About Wikipedia
Contact us
Donate

Here are the rules:

- Start with a random article
- Your goal is to find your way to the University of Toronto wiki page
- The only way you can move is by clicking on links

Let's play!

Modelling as a shortest path problem

Modelling as a shortest path problem

- $V = \{\text{wiki pages}\}$
- $E = \{(u, v) \in V \times V : u \text{ links to } v\}$

Let $G = (V, E)$. Then, given a random Wikipedia page u, the shortest path from u to $\mathrm{University of Toronto}$ is the optimal solution for the Wikipedia Game.

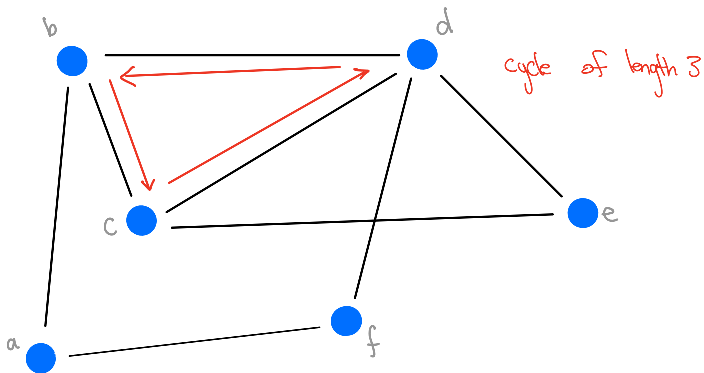
Cycles

Let $G = (V, E)$ be a graph

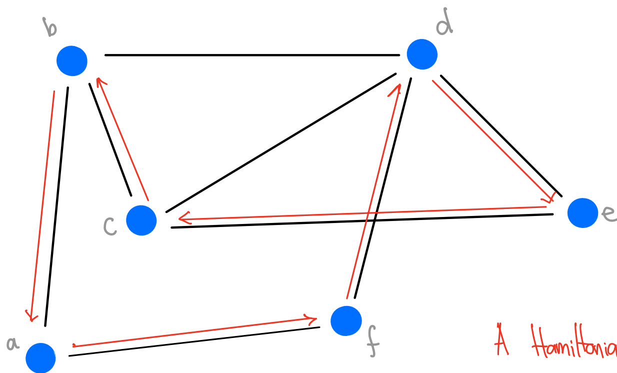
- A sequence of vertices $(v_1, ..., v_n)$, is a cycle if $(v_1, ..., v_{n-1})$ is a path, $v_1 = v_n$, and $\{v_{n-1}, v_n\} \in E$.
- A cycle is called Hamiltonian if every vertex appears in the cycle exactly once (except for the start/end vertex which appears twice).

The minimum length of a cycle is 3 (i.e. $n \geq 4$).

Example - Cycle



Example - Hamiltonian Cycle



A Hamiltonian cycle

Traveling Salesman Problem

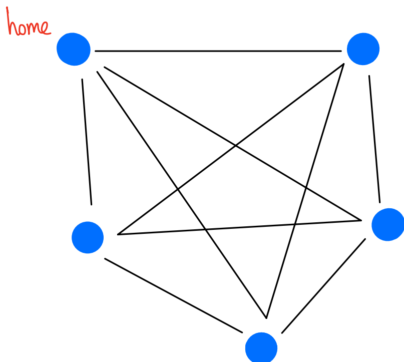
Input: A graph $G = (V, E)$ and a starting vertex h

Output: A Hamiltonian cycle in G starting from h that minimizes the total edge weights.

Selling Strawberries

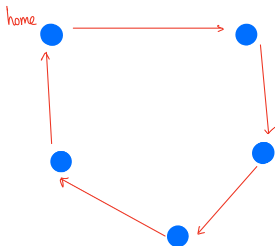
Imagine you're a door to door strawberries salesperson. Let H be the set of homes in your neighborhood. You live at $h \in H$.

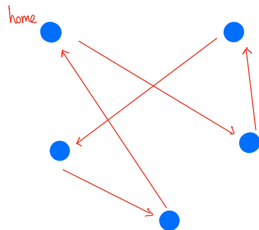
Selling Strawberries



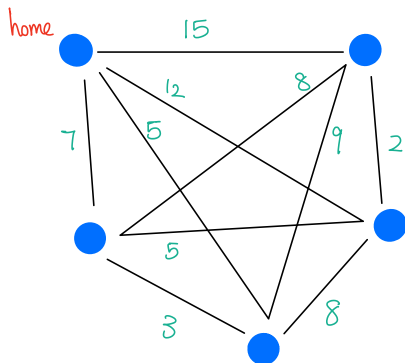
Your goal is to start at home, and visit every house in your neighborhood while walking the shortest distance possible.

Selling Strawberries





Selling Strawberries



This corresponds exactly to the solution to the Traveling Salesman Problem. I.e. we're looking for the Hamiltonian cycle that minimizes the total weight!

Definitions

Modelling with Graphs
- Problem 1. Matching
- Problem 2. Shortest Path
- Problem 3. The Traveling Salesman

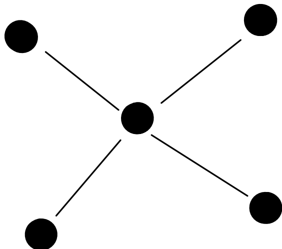
Trees - A special type of graph
- Problem 4. Minimum Spanning Tree

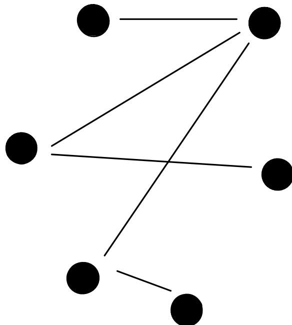
Trees

Let $G = (V, E)$ be any graph.

- G is connected if for every pair of distinct vertices u, v, there is some path from u to v.
- G is acyclic if there are no cycles in G.
- G is called a tree if G is both connected and acyclic.

Examples





Trees - on a knife's edge

If G has many edges, it's more likely to be connected, but also more likely to have a cycle.

If G has fewer edges, it's more likely to be acyclic but less likely to be connected.

Since trees are both connected and acyclic, trees represent a perfect compromise. However, as we will see on the next slide, any addition or subtraction of an edge will destroy the balance.

Trees - on a knife's edge

Trees are minimally connected graph, meaning that it is connected but removing any edge causes the tree to be disconnected.

Trees - on a knife's edge

Trees are minimally connected graph, meaning that it is connected but removing any edge causes the tree to be disconnected.

Trees are maximally acyclic graph, meaning that there is no cycle but adding any edge creates a cycle.

Proof: Minimally Connected

Tree: connected and acyclic.
WTS: removing any edge causes tree to be disconnected.

Proof: Minimally Connected

Tree: connected and acyclic.
WTS: removing any edge causes tree to be disconnected.

Let $G = (V, E)$ be a tree. Suppose $\{u, v\}$ is an edge in G. Let $G' = (V, E \setminus \{\{u, v\}\})$ be the graph with $\{u, v\}$ removed. We'll show that G' is disconnected - in particular, there is no path from u to v.

By contradiction, if there was a path $P = (u = v_1, v_2, ..., v_n = v)$ from u to v in G', then the $C = (u = v_1, v_2, ..., v_n = v, u)$ is a cycle in G. However, G is acyclic so this is a contradiction.

Proof: Maximally Acyclic

Tree: connected and acyclic.
WTS: adding any edge creates a cycle

Proof: Maximally Acyclic

Tree: connected and acyclic.
WTS: adding any edge creates a cycle

Let $G = (V, E)$ be a tree. Suppose $\{u, v\}$ is an edge not in G. Let $G' = (V, E \cup \{\{u, v\}\})$ be the graph with the edge $\{u, v\}$ added. We'll show that G' has a cycle.

Since G is connected, there is some path $P = (u = v_1, ..., v_n = v)$ from u to v in G. Then $C = (u = v_1, ..., v_n = v, u)$ is a cycle in G'.

Converses

Converses

Let G be a graph.

- If G is minimally connected. Then G is a tree.
- If G is maximally acyclic. Then G is a tree.

Are these also true?

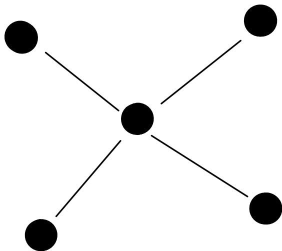
Converses

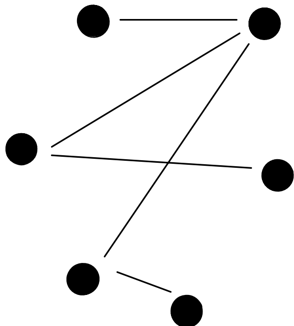
Let G be a graph.

- If G is minimally connected. Then G is a tree.
- If G is maximally acyclic. Then G is a tree.

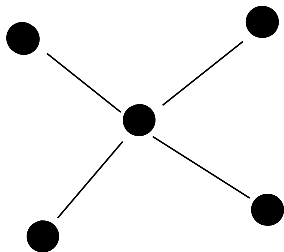
Are these also true? Yes!

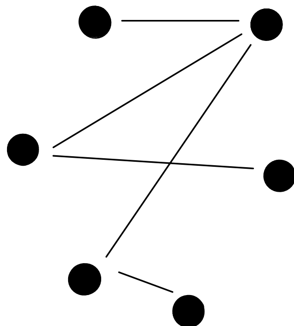
How many edges does a tree have?





How many edges does a tree have?





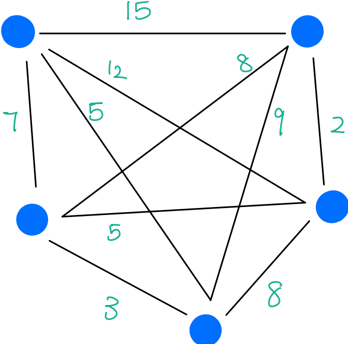
$|V| - 1$. We will prove this next time.

Electrical Grid

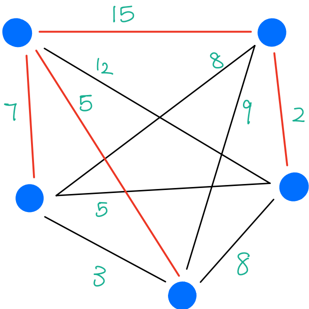
Given a set of houses, what is the most efficient way to connect an electrical grid?

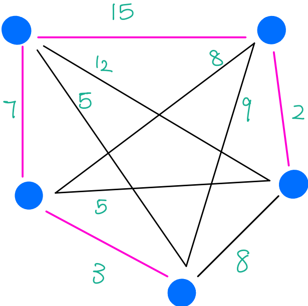
The goal is to connect homes so that all of the homes are connected to each other by some path using the least amount of cable.

Electrical Grid



Electrical Grid





two spanning trees.

Minimum Spanning Tree

Input: A connected, weighted graph $G = (V, E)$.

Output: A graph $T = (V, E')$, where $E' \subseteq E$, such that T is a tree that minimizes the total edge weights.

Which of these problems seem more difficult?

- Minimum Spanning Tree
- Shortest Path
- Matching
- Traveling Salesman

Which of these problems seem more difficult?

- Minimum Spanning Tree
- Shortest Path
- Matching
- Traveling Salesman

We know fast algorithms for the first three, but NOT for the last one. In fact, the Traveling Salesman problem is conjectured to have no efficient algorithm.

Additional Notes

- The skill I want you to develop here is to identify how real world problems can be translated to known problems on graphs.
- We did not study in detail HOW to solve such problems (that's the main topic of CSC373).
- Even though we don't know of a good algorithm for TSP, we do have fast approximation algorithms for it.

Additional Notes

Here are some references for some algorithms in case you are curious.

- Matching: [1 (Blossom Algorithm)], [2]
- Shortest Path: [1 (Dijkstra's Algorithm)]
- TSP: [1 (Christofides Algorithm)]
- MST: [1 (Prim's Algorithm)], [2 (Kruskal's Algorithm)]

CSC 236 Lecture 3: Induction

Harry Sha

May 24, 2023

Today

Induction

Examples of Proofs by Induction

Complete Induction

Induction

Examples of Proofs by Induction

Complete Induction

What is induction used for

Induction is used to prove statements of the following form:

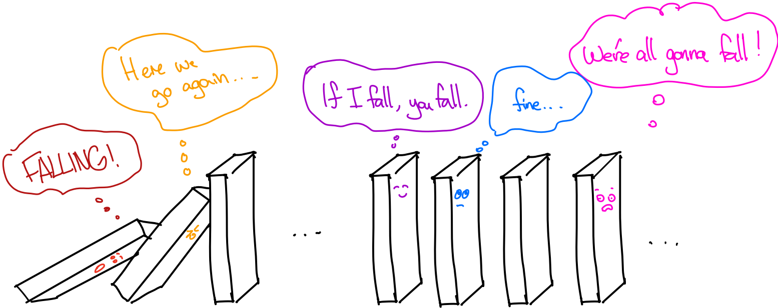
$$\forall n \in \mathbb{N}.(P(n)).$$

Note that P is a predicate on the natural numbers. I.e. for any natural number n, $P(n)$ is either true or false. For example, $P(n)$ might be

- The sum of the first n odd numbers is n^2.
- $(12^n - 1)$ is divisible by 11.
- Trees with n vertices have $n - 1$ edges.

Induction is super useful when analyzing the correctness and runtime of algorithms.

Induction



Principle of Mathematical Induction.

Induction

$$(P(0) \land \forall k \in \mathbb{N}.(P(k) \implies P(k+1))) \implies \forall n \in \mathbb{N}.(P(n))$$

Induction

$$(P(0) \land \forall k \in \mathbb{N}.(P(k) \implies P(k+1))) \implies \forall n \in \mathbb{N}.(P(n))$$

"If I can show that the first domino falls, and I can show that for any domino, if that domino falls, the next one falls, every domino falls".

Here, the kth domino falling is analogous to $P(k)$ being true.

Proofs by induction

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

If we want to prove a statement of the form $\forall n \in \mathbb{N}.(P(n))$, it suffices to prove

1. $P(0)$ (base case)
2. $\forall k \in \mathbb{N}.(P(k) \implies P(k+1))$ (inductive step)

Induction template

Say we wanted to prove $\forall n \in \mathbb{N}.(P(n))$. Here is the template:

By induction.

Base case. [Prove $P(0)$ is true]

Inductive step. Let $k \in \mathbb{N}$ be an arbitrary natural number, and assume $P(k)$. We'll show $P(k+1)$. [Prove $P(k+1)$ assuming $P(k)$]. This completes the induction.

The assumption, $P(k)$, in the inductive step is called the inductive hypothesis (IH).

Flexibility

There is some flexibility in the proof by induction template. For example, sometimes, we want to prove a statement is true for all $n \geq 1$, so the base case starts at 1 instead of 0. Sometimes, for the inductive step, we need to show that if the previous **two** dominoes fall, then the next one falls. In this case, we would need to prove two base cases. These will come up in the examples, and you will see why we can do this next week.

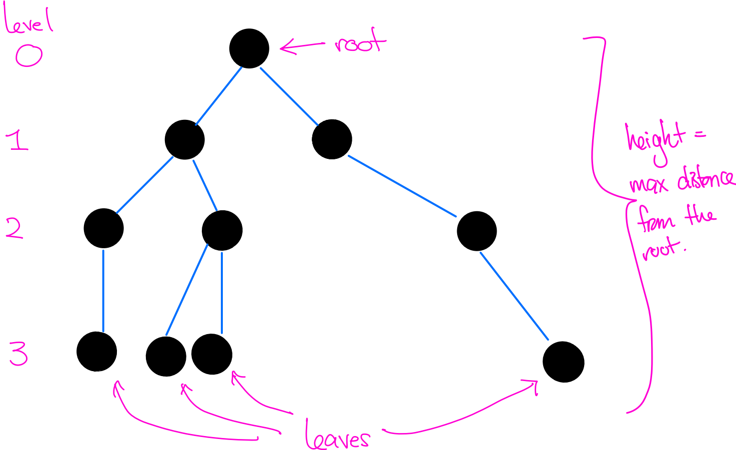
You should be okay as long as you showed enough so that "all the dominoes fall".

Induction

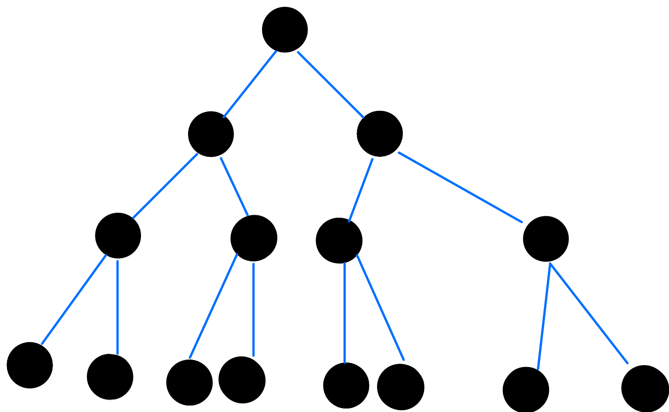
Examples of Proofs by Induction

Complete Induction

Binary Trees

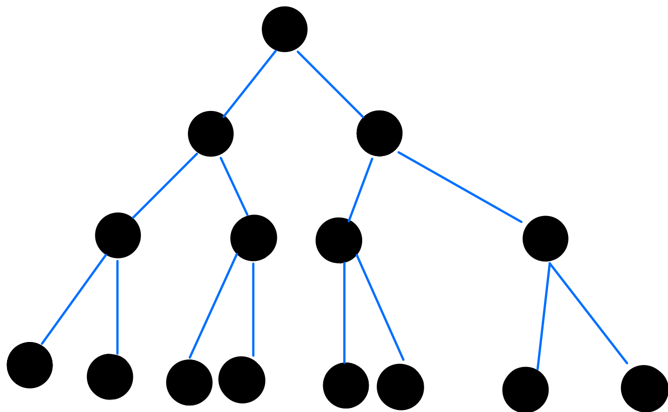


Perfect binary trees



In a perfect binary tree, all leaves are at the same level, and every other vertex has two children and one parent (except for the root, which does not have a parent).

Number of vertices of perfect binary trees



How many vertices does perfect binary tree of height n have?

Number of vertices of perfect binary trees

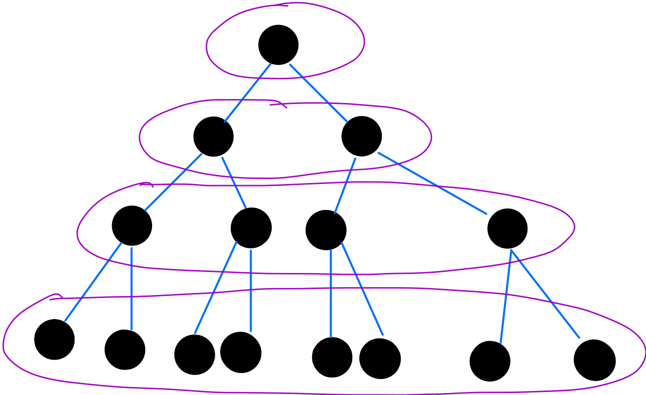
$1 = 2^0$

$2 = 2^1$

$4 = 2^2$

$8 = 2^3$

$\vdots$



$$\forall n \in \mathbb{N}.(2^0 + 2^1 + ... + 2^n = 2^{n+1} - 1)$$

$\forall n \in \mathbb{N}.(2^0 + 2^1 + ... + 2^n = 2^{n+1} - 1)$

By induction.

Base case. For the base case, we need to check $2^0 = 2^{0+1} - 1$. This holds, since they are both equal to 1.

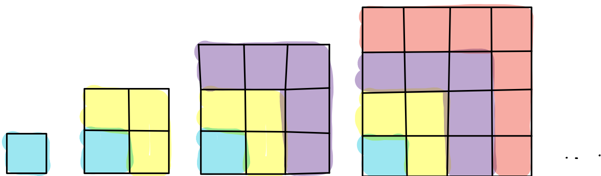
Inductive step. Let $k \in \mathbb{N}$ be an arbitrary natural number, and assume $\sum_{i=0}^{k} 2^i = 2^{k+1} - 1$. We'll show $\sum_{i=0}^{k+1} 2^i = 2^{k+2} - 1$. We have

$$\begin{aligned}\sum_{i=0}^{k+1} 2^i &= 2^{k+1} + \sum_{i=0}^{k} 2^i \\ &= 2^{k+1} + 2^{k+1} - 1 \\ &= 2^{k+2} - 1\end{aligned}$$

as required.

Sum of first n odd numbers is the nth square

Sum of first n odd numbers is the nth square



Sum of first n odd numbers is the nth square

We want to prove $\forall n \in \mathbb{N}, n \geq 1.(\sum_{i=1}^{n} 2i - 1 = n^2)$

[2] Here we prove $P(1)$ instead of $P(0)$ since the statement we are trying to prove starts at 1.

Sum of first n odd numbers is the nth square

We want to prove $\forall n \in \mathbb{N}, n \geq 1.(\sum_{i=1}^{n} 2i - 1 = n^2)$
Base case.[2]

$$2 \cdot 1 - 1 = 1 = 1^2,$$

so the base case holds.
Inductive step. Let $k \in \mathbb{N}, k \geq 1$ be an arbitrary natural number, and assume $\sum_{i=1}^{k} 2i - 1 = k^2$. We'll show $\sum_{i=1}^{k+1} 2i - 1 = (k+1)^2$.

$$\begin{aligned} \sum_{i=1}^{k+1} 2i - 1 &= 2(k+1) - 1 + \sum_{i=1}^{k} 2i - 1 \\ &= 2k + 1 + k^2 \\ &= (k+1)^2 \end{aligned}$$

as required.

[2]Here we prove $P(1)$ instead of $P(0)$ since the statement we are trying to prove starts at 1.

$\forall n \in \mathbb{N}.(n^3 - n + 3)$ is divisible by 3

$\forall n \in \mathbb{N}.(n^3 - n + 3)$ is divisible by 3

By induction.

Base case. For the base case, we have $0^3 - 0 + 3 = 3$ which is divisible by 3.

Inductive step. Let $k \in \mathbb{N}$ by any natural number and assume $k^3 - k + 3$ is divisible by 3, i.e. $k^3 - k + 3 = 3p$ for some $p \in \mathbb{N}$. We will show $(k+1)^3 - (k+1) + 3$ is also divisible by 3. We have

$$\begin{aligned}(k+1)^3 - (k+1) + 3 &= k^3 + 3k^2 + 3k + 1 - k - 1 + 3 \\ &= (k^3 - k + 3) + 3k^2 + 3k + 1 - 1 \\ &= 3p + 3k^2 + 3k \\ &= 3(p + k^2 + k)\end{aligned}$$

as required.

$\forall n \in \mathbb{N}$. the units digit of 7^n is 1, 3, 7, or 9

$\forall n \in \mathbb{N}$. the units digit of 7^n is 1, 3, 7, or 9

By induction.

Base case. $7^0 = 1$, so the base case holds.

Inductive step. Let $k \in \mathbb{N}$ be an arbitrary natural number, and assume the units digit of 7^k is either $1, 3, 7$ or 9. There are several cases.

- If it was 1, 7^{k+1} has units digit 7.
- If it was 3, 7^{k+1} has units digit 1.
- If it was 7, 7^{k+1} has units digit 9.
- If it was 9, 7^{k+1} has units digit 3.

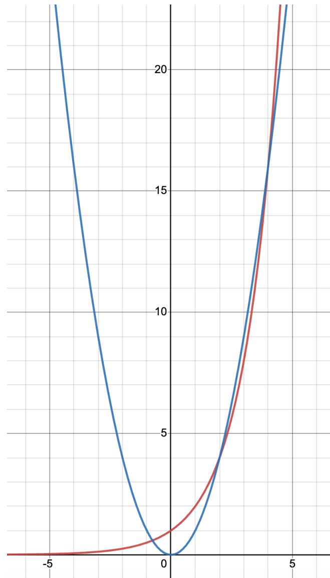
Thus, the inductive step holds and we are done.

$\forall n \in \mathbb{N}.(n^2 \leq 2^n)$

$\forall n \in \mathbb{N}.(n^2 \leq 2^n)$

False! E.g. for $n = 3$, the LHS is 9 and the RHS is 8.

$\forall n \in \mathbb{N}, n \geq 4.(n^2 \leq 2^n)$



$$\forall n \in \mathbb{N}, n \geq 4.(n^2 \leq 2^n)$$

$\forall n \in \mathbb{N}, n \geq 4.(n^2 \leq 2^n)$

By induction.

Base case. $4^2 = 16$ and $2^4 = 16$, so the base case holds.
Inductive step. Let $k \in \mathbb{N}$ be an arbitrary natural number with $k \geq 4$, and assume $k^2 \leq 2^k$. We'll show $(k+1)^2 \leq 2^{k+1}$. We have

$$\begin{aligned}
(k+1)^2 &= k^2 + 2k + 1 \\
&\leq k^2 + (k-2)k + 1 && (k - 2 \geq 2) \\
&= k^2 + k^2 - 2k + 1 \\
&\leq 2k^2 && (-2k + 1 \leq 0) \\
&\leq 2 \cdot 2^k && \text{(IH)} \\
&= 2^{k+1},
\end{aligned}$$

completing the induction.

All birds have the same color

Claim. $\forall n \in \mathbb{N}$, a set of n birds will all have the same color.

Is this claim true?

[3] This example is usually "all **horses** have the same color," but I do not know how to draw horses - hence "all birds have the same color"

All birds have the same color

Claim. $\forall n \in \mathbb{N}$, a set of n birds will all have the same color.

Is this claim true?

No, of course not![3]

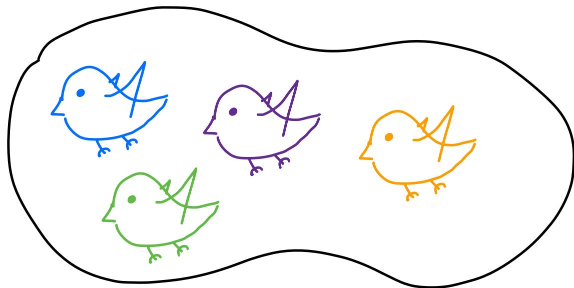
[3] This example is usually "all **horses** have the same color," but I do not know how to draw horses - hence "all birds have the same color"

All birds have the same color - “proof”

Claim. $\forall n \in \mathbb{N}$, a set of n birds will all have the same color.

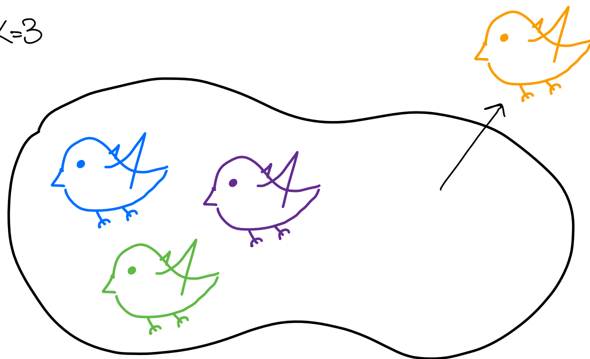
Base Case. For $n = 0$, the claim is vacuously true.

All birds have the same color - "proof"



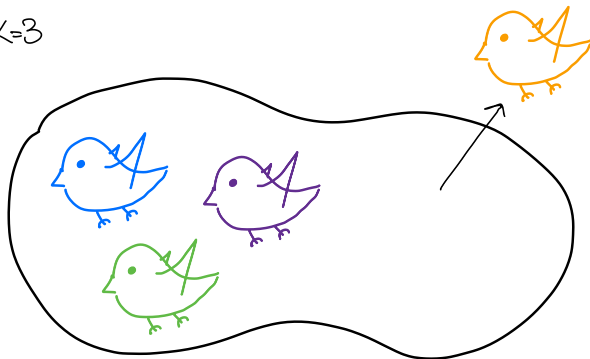
Inductive step. Let $k \in \mathbb{N}$ be any number and assume a set of k birds will all have the same color. Let S be a set of $k+1$ birds, we'll show that all the birds in S have the same color.

All birds have the same color - “proof”



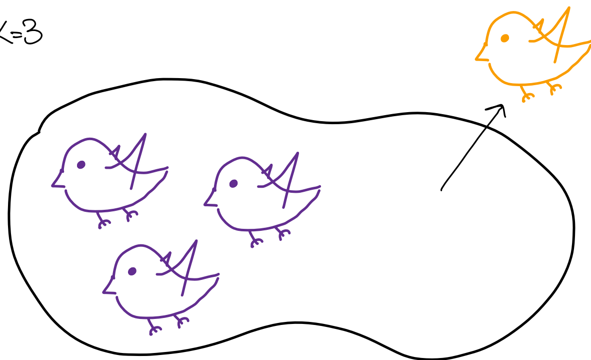
Remove an arbitrary bird b_1.

All birds have the same color - “proof”



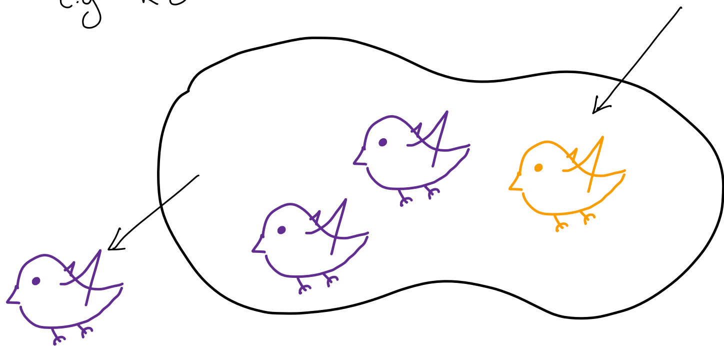
The remaining k birds must have the same color by the inductive hypothesis.

All birds have the same color - “proof”



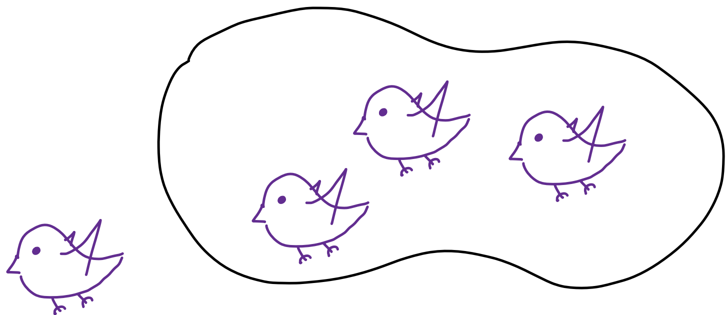
The remaining k birds must have the same color by the inductive hypothesis.

All birds have the same color - “proof”



Add back the removed bird and now remove a different bird, b_2.

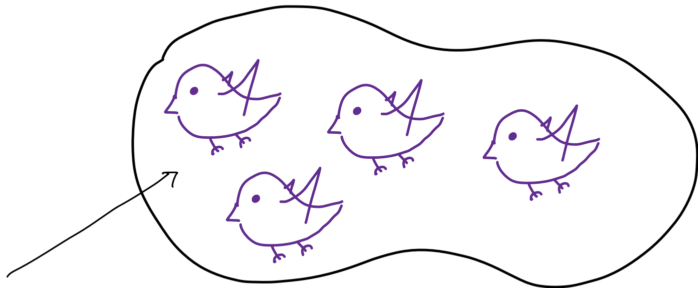
All birds have the same color - “proof”



By the same reasoning, the remaining k birds must have the same color. Therefore b_1 has the same color as birds which have not been removed which in turn have the same color as b_2.

All birds have the same color - “proof”

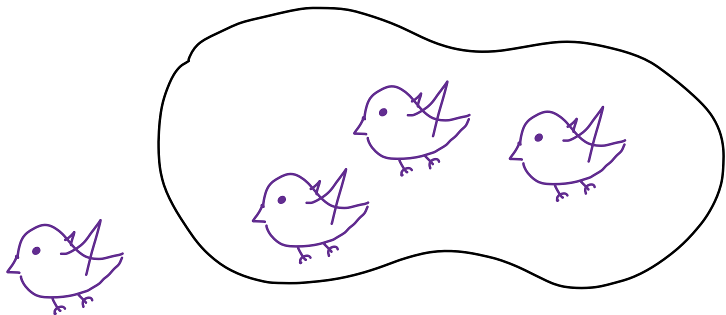
e.g $k=3$



All birds have the same color. □

What went wrong?

What went wrong?



By the same reasoning, the remaining k birds must have the same color. Therefore b_1 has the same color as the birds which have not been removed which in turn have the same color as b_2.[a]

[a] There is an implicit assumption here that there is another bird other than b_1 and b_2! I.e. for this to hold, we need $|S| \geq 3$! In particular, it does not hold for $k = 1$, ($|S| = 2$).

Takeaway

Induction can be tricky! Make sure your inductive step does not assume anything more than what you claim! For example, in the $n^2 \leq 2^n$ example, we could assume $k \geq 4$ since we were restricting to the case where $k \geq 4$, but we couldn't do the same for the "all birds have the same color" example.

Induction

Examples of Proofs by Induction

Complete Induction

Complete Induction

Complete induction is another way to prove statements of the form $\forall n \in \mathbb{N}.(P(n))$.

Another way to get all the dominoes to fall

If I want to show $\forall n \in \mathbb{N}.(P(n))$, it suffices to prove

- $P(0)$
- ~~$\forall k \in \mathbb{N}.(P(k) \implies P(k+1))$~~
- $\forall k \in \mathbb{N}.((P(0) \wedge P(1) \wedge \ldots \wedge P(k)) \implies P(k+1))$

"If I can show that the first domino falls, and I can show that for any domino, if that domino falls **and all previous dominoes fall**, that the next one also falls, every domino falls".

Complete Induction template

Say we wanted to prove $\forall n \in \mathbb{N}.(P(n))$. Here is the template:

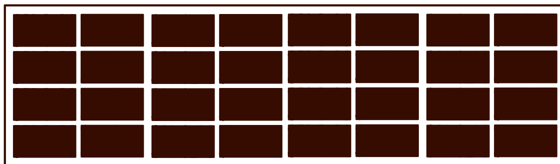
By complete induction.

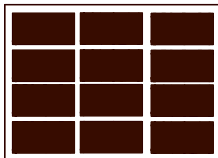
Base case. [Prove $P(0)$ is true]

Inductive step. Let $k \in \mathbb{N}$ be an arbitrary natural number, and assume for every $i \in \mathbb{N}$ with $i \leq k$, $P(i)$ holds. We'll show $P(k+1)$. [Prove $P(k+1)$ using this assumption]

Note. $\forall i \in \mathbb{N}, i \leq k.(P(k))$ is just another way of writing $P(0) \wedge P(1) \wedge ... \wedge P(k)$. This is again called the inductive hypothesis.

Chocolate





How many breaks do you need to split the chocolate bar into individual pieces?

Chocolate - Attempted proof by regular induction

Claim: Let $n \in \mathbb{N}, n \geq 1$ be any natural number. A chocolate bar with n individual pieces requires $n-1$ breaks to split the bar into individual pieces.

Let $P(n)$ be the predicate: A bar of chocolate composed of n individual pieces requires $n-1$ breaks.

Base case. For $n = 1$, the chocolate bar is already a single piece of chocolate and so requires $1 - 1 = 0$ breaks.

Inductive step. Let $k \in \mathbb{N}, k \geq 1$ be an arbitrary natural number at least 1, and assume $P(k)$ is true. We need to show $P(k+1)$ is true. Let B be a bar of chocolate with $k+1$ pieces and pick a way break a chocolate. We are left with two blocks of chocolate of size a and b respectively where $a + b = k + 1$.

Chocolate - Attempted proof by regular induction

Claim: Let $n \in \mathbb{N}, n \geq 1$ be any natural number. A chocolate bar with n individual pieces requires $n-1$ breaks to split the bar into individual pieces.

Let $P(n)$ be the predicate: A bar of chocolate composed of n individual pieces requires $n-1$ breaks.

Base case. For $n = 1$, the chocolate bar is already a single piece of chocolate and so requires $1 - 1 = 0$ breaks.

Inductive step. Let $k \in \mathbb{N}, k \geq 1$ be an arbitrary natural number at least 1, and assume $P(k)$ is true. We need to show $P(k+1)$ is true. Let B be a bar of chocolate with $k+1$ pieces and pick a way break a chocolate. We are left with two blocks of chocolate of size a and b respectively where $a + b = k + 1$. What now? We can't apply the IH since a, b might not be k!

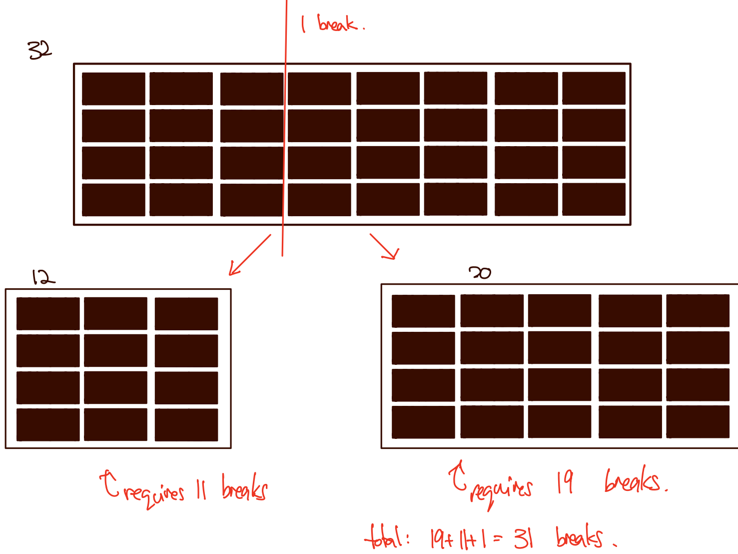
Chocolate - Proof by Complete Induction

Let $P(n)$ be the predicate: A bar of chocolate composed of n individual pieces requires $n-1$ breaks.

Base case. For $n=1$, the chocolate bar is already a single piece of chocolate and so requires $1-1=0$ breaks.

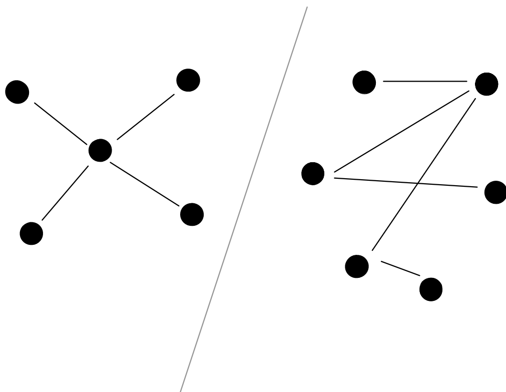
Inductive step. Let $k \in \mathbb{N}, k \geq 1$ be an arbitrary natural number at least 1, and assume $P(i)$ is true for all $i \leq k$. We need to show $P(k+1)$ is true. Let B be a bar of chocolate with $k+1$ pieces and select any way break the bar. We are left with two blocks of chocolate of size a and b respectively where $a+b=k+1$, $a \leq k$, and $b \leq k$. Applying the inductive hypothesis to each of these blocks, we have that the two blocks require $a-1$ and $b-1$ breaks respectively, so the total number of breaks (including the initial) is $a+b-2+1=k$, as required.

Chocolate - Proof by Complete Induction



Number of Edges in a Tree

As a reminder, a tree is a graph $G = (V, E)$ that is both acyclic (has no cycles) and connected (every pair of vertices is connected by some path). What was our conjecture from last time?



A tree has $|V|$ - 1 edges

Why would I ever use regular induction if I can just use complete induction?

Why would I ever use regular induction if I can just use complete induction?

You can always use complete induction if you wish! The inductive hypothesis is stronger (I.e., you get to assume $P(0) \wedge ... \wedge P(k)$ instead of just $P(k)$), but still lets you prove the same statement: $\forall n \in \mathbb{N}.(P(n))$.

That being said, some mistakes are easier to make when using complete induction, and sometimes regular induction is easier to work with.

Induction and Algorithms

Proof by induction is an incredibly powerful technique that will allow us to prove strong guarantees about the runtime and correctness of algorithms.

I.e. Induction let's us prove $\forall n \in \mathbb{N}.(P(n))$ consider what this means when

- $P(n)$ is: Algorithm X is correct on inputs of size n.
- $P(n)$ is: Algorithm X is correct if the `for` loop runs for at most n iterations.
- $P(n)$ is: The runtime of Algorithm X is $\Theta(n^2)$.
- ..etc.

A note about style

Sometimes you might find it easier to define a predicate in the following way: “Let $P(n)$ be the predicate...” for example, in the chocolate example. This approach allows you to refer to the predicate easily. For example, defining P allows you to say “Assume $P(k)$ is true...”

Other times, you might find it easier to directly work with the predicate without giving it a name, for example, in the divisibility example. This approach makes stating the inductive hypothesis a little more troublesome but reminds the reader of exactly what you’re trying to prove.

Both are fine!

Additional Notes

- Induction and recursive algorithms are closely linked. Think of how! We will explore this in future classes.
- Induction is a hard concept to grasp. In particular, it takes a little bit of faith to believe that simply showing a base case and an inductive step allows us to prove a statement is true for all natural numbers. It's good to keep the domino analogy in mind, i.e., when writing your proofs, ask - *'did I show all the dominoes fall?'*
- Although intuition is important, at the end of the day, remember that the base case and inductive step are both mathematical statements that you need to prove. I.e., you should approach proving $\forall k \in \mathbb{N}.(P(k) \implies P(k+1))$ like you would approach proving any other FOL statement.

CSC 236 Lecture 4: Induction 2

Harry Sha

May 31, 2023

Today

Induction

Structural Induction

Well Ordering Principle and Proof by Infinite Descent

Induction

Structural Induction

Well Ordering Principle and Proof by Infinite Descent

Induction and Complete Induction

To prove

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

it is enough to prove $P(0)$ and one of the following

- $\forall k \in \mathbb{N}.[P(k) \implies P(k+1)]$
- $\forall k \in \mathbb{N}.[(P(0) \land P(1) \land ... \land P(k)) \implies P(k+1)]$

Intuition: why does (regular) induction work again?

Say I managed to show $P(0)$, and $\forall k \in \mathbb{N}.(P(k) \implies P(k+1))$.
Then let $n \in \mathbb{N}$ be any number, here's why $P(n)$ is true:

- $P(0) \implies P(1)$, and $P(0)$, so $P(1)$
- $P(1) \implies P(2)$, and $P(1)$, so $P(2)$
- ...
- $P(n-1) \implies P(n)$, and $P(n)$, so $P(n)$.

Intuition: why does (complete) induction work again?

Say I managed to show $P(0)$, and $\forall k \in \mathbb{N}.((P(0) \wedge P(1) \wedge ... P(k)) \implies P(k+1))$. Then let $n \in \mathbb{N}$ be any number, here's why $P(n)$ is true:

- $P(0) \implies P(1)$, and $P(0)$, so $P(1)$
- $P(0) \wedge P(1) \implies P(2)$, and $P(0) \wedge P(1)$, so $P(2)$
- ...
- $(P(0) \wedge .... \wedge P(n-1)) \implies P(n)$, and $P(0) \wedge ... \wedge P(n-1)$, so $P(n)$.

Postage Stamps

Say that you have an unlimited number of 3 cent and 5 cent postage stamps. Can you make any postage exactly?

Postage Stamps

Say that you have an unlimited number of 3 cent and 5 cent postage stamps. Can you make any postage exactly?

No, i.e. 1, 2, 4, 7 can't be made.

Can you make any postage ≥ 8 cents exactly?

Rephrasing the problem mathematically

Claim: For any $n \geq 8$, there exists $a, b \in \mathbb{N}$ such that $n = 3a + 5b$

Proof, attempt 1 (wrong!)

Claim: For any $n \geq 8$, there exists $a, b \in \mathbb{N}$ such that $n = 3a + 5b$

By complete induction.

Base case. We can make an 8 cent postage using one 3 cent stamp and one 5 cent stamp.

Inductive step. Let $k \geq 8$ and assume for any $8 \leq i \leq k$, we can make a postage of i cents using only 3 and 5 cent stamps. We'll show that you can also make a $k+1$ postage. Use one 3-cent stamp. We now need to make a $k-2$ postage. By the induction hypothesis, we can make $k-2$ using only 3 cent and 5 cent stamps, so together, we have made a $k+1$ postage.

What's the problem here?

Proof, attempt 1 (wrong!)

Claim: For any $n \geq 8$, there exists $a, b \in \mathbb{N}$ such that $n = 3a + 5b$

By complete induction.

Base case. We can make an 8 cent postage using one 3 cent stamp and one 5 cent stamp.

Inductive step. Let $k \geq 8$ and assume for any $8 \leq i \leq k$, we can make a postage of i cents using only 3 and 5 cent stamps. We'll show that you can also make a $k + 1$ postage. Use one 3-cent stamp. We now need to make a $k - 2$ postage. By the induction hypothesis, we can make $k - 2$ using only 3 cent and 5 cent stamps[4], so together, we have made a $k + 1$ postage.

[4] $k - 2$ might be 6 which is not covered by the induction hypothesis

Problem

Out induction hypothesis was $P(8), P(9), ..., P(k)$, and we wanted to show $P(k+1)$. However, when $k = 8$ or 9, $k - 2$ is 6 or 7 which is not covered by the induction hypothesis! So our argument in the inductive step doesn't work for $k = 8$ or $k = 9$.

To fix this, we can just prove $P(k+1)$ directly for these cases for $k = 8$ and $k = 9$.

Proof 1. multiple base cases

Claim: For any $n \geq 8$, there exists $a, b \in \mathbb{N}$ such that $n = 3a + 5b$

Proof 1. multiple base cases

Claim: For any $n \geq 8$, there exists $a, b \in \mathbb{N}$ such that $n = 3a + 5b$

By complete induction.

Base cases. Since $8 = 3 + 5, 9 = 3 + 3 + 3, 10 = 5 + 5$, we can make postages of $8, 9, 10$.

Inductive step. Let $k \geq 10$ and assume for any $8 \leq i \leq k$, we can make a postage of i cents using only 3 and 5 cent stamps. We'll show that you can also make a $k + 1$ postage. Use one 3 cent stamp, we now need to make a $k - 2$ postage. Since $8 \leq k - 2$, the induction hypothesis applies, and we can make $k - 2$ using only 3 cent and 5 cent stamps, so together, we have made a $k + 1$ postage.

Proof 2. Regular induction

Claim: For any $n \geq 8$, there exists $a, b \in \mathbb{N}$ such that $n = 3a + 5b$

Proof 2. Regular induction

> **Claim:** For any $n \geq 8$, there exists $a, b \in \mathbb{N}$ such that $n = 3a + 5b$

By regular induction.

Base case. Same as before
Inductive step. Let $k \geq 8$, and assume there are $a, b \in \mathbb{N}$ such that $k = 3a = 5b$. There are two cases

- If $b \geq 1$, we can create $k + 1$ by removing a 5 cent stamp and adding two 3 cent stamps.
- If $b = 0$, then since $k \geq 8$, $a \geq 3$, and we can create $k + 1$ by removing three 3 cent stamps and adding two 5 cent stamps.

$\forall n \in \mathbb{N}.(2n = 0)$

By complete induction.

Base case. $2 \cdot 0 = 0$ so the base case holds.

Inductive step. Let $k \in \mathbb{N}$ be an arbitrary natural number and assume $2 \cdot k = 0$, we'll show $2 \cdot (k+1) = 0$. Write $k + 1 = i + j$ for some smaller natural numbers i, j. Then we have

$$2(k+1) = 2(i+j) = 2i + 2j = 0 + 0,$$

where we used the inductive hypothesis on i and j in the last equality.

$\forall n \in \mathbb{N}.(2n = 0)$

By complete induction.

Base case. $2 \cdot 0 = 0$ so the base case holds.

Inductive step. Let $k \in \mathbb{N}$ be an arbitrary natural number and assume $2 \cdot k = 0$, we'll show $2 \cdot (k+1) = 0$. Write $k+1 = i+j$ for some smaller natural numbers i, j[5]. Then we have

$$2(k+1) = 2(i+j) = 2i + 2j = 0 + 0,$$

where we used the inductive hypothesis on i and j in the last equality.

[5] you can't do this for $k = 0$

Induction

Structural Induction

Well Ordering Principle and Proof by Infinite Descent

Induction

So far, we've been able to use the powerful tools of induction and complete induction to prove statements of the form.

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

However, in life, we are also interested in objects other than the natural numbers. For example, lists, trees, and logical formulas. I.e., we may want to prove statements like

$$\forall \mathrm{Trees}\, T.(P(T)),$$

and

$$\forall \mathrm{Formulas} f.(P(f)).$$

We "need"[6] a more general tool.

[6]the quotes here will be explained later

Another view of $\mathbb{N}$

Here's another one way to define $\mathbb{N} = \{0, 1, 2, ...\}$.
Let AddOne be the function that maps $x \to x + 1$.

Then, $\mathbb{N}$ is the set of objects can be reached by applying AddOne to $\{0\}$ a finite number of times.

Defining Sets Inductively

- Let $B \subseteq U$ (think B for **b**ase cases)
- Let F be a set of **f**unctions, where each function $f \in F$ has domain U^m and codomain U. I.e. f maps a tuple of elements of U to a single element of U (think of F as a set of construction operations)

The set **generated** from B by the functions in F is the set of elements that can be obtained by applying functions in F to elements of B a finite number of times.

Alternatively

An equivalent way to express
"A is the set of elements that can be obtained by applying functions in F to elements of B a finite number of times."
is to define

A is the smallest set satisfying the following conditions.

- $B \subseteq A$
- $\forall a \in A, f \in F,\ f(a) \in A$.

Example: $\mathbb{N}$

- $B = \{0\}$
- $F = \{\text{AddOne}\}$

1. $\mathbb{N}$ is the set generated from $\{0\}$ by $\{\text{AddOne}\}$

Example: $\mathbb{N}$

- $B = \{0\}$
- $F = \{\text{AddOne}\}$

1. $\mathbb{N}$ is the set generated from $\{0\}$ by $\{\text{AddOne}\}$
2. Alternatively, $\mathbb{N}$ is the smallest set that contains 0, and for each $n \in \mathbb{N}$, $\mathbb{N}$ also contains $\text{AddOne}(n)$.

Example: $\mathbb{Z}$

1. $\mathbb{Z}$ is the set generated from $\{0\}$ by $\{\mathrm{AddOne}, \mathrm{MinusOne}\}$

Example: $\mathbb{Z}$

1. $\mathbb{Z}$ is the set generated from $\{0\}$ by $\{\mathrm{AddOne}, \mathrm{MinusOne}\}$
2. Alternatively, $\mathbb{Z}$ is the smallest set that contains 0, and for each $z \in \mathbb{Z}$, $\mathbb{Z}$ also contains $\mathrm{AddOne}(z)$, and $\mathrm{MinusOne}(z)$.

Example: $\mathrm{List}[X]$

Let X be some set, and let $\mathrm{List}[X]$ be the set of lists of elements in X.

Example: $\mathrm{List}[X]$

Let X be some set, and let $\mathrm{List}[X]$ be the set of lists of elements in X.

For each $x \in X$ define the function Append_x be the function that takes in a l and appends x to l.

- $B = \{[]\}$
- $F = \{\mathrm{Append}_x : x \in X\}$

$\mathrm{List}[X]$ is the set generated from B by functions in F.

Propositional logic

Propositional logic is logic without predicates or quantifiers. For example $((A \wedge B) \vee (\neg C))$ is a propositional formula. Let Prop be the set of propositional formulas. Define Prop inductively.

Propositional logic

Propositional logic is logic without predicates or quantifiers. For example $((A \wedge B) \vee (\neg C))$ is a propositional formula. Let Prop be the set of propositional formulas. Define Prop inductively.

- $B = \{A, B, C, ...\}$ be a set of variables
- $F = \{\mathbf{E}_{\neg}, \mathbf{E}_{\wedge}, \mathbf{E}_{\vee}\}$

Where $\mathbf{E}_{\neg}(A) = (\neg A)$, $\mathbf{E}_{\wedge}(A, B) = (A \wedge B)$, and $\mathbf{E}_{\vee}(A, B) = (A \vee B)$.

Structural Induction

Let C be a set generated from B by the functions in F.

If

- for every $b \in B$, $P(b)$,
- and for every $f \in F$ on m inputs, for every $a_1, ..., a_m \in C$, $(P(a_1) \land P(a_2) \land ... \land P(a_m)) \implies P(f(a_1, ..., a_m))$

Then $\forall x \in C.(P(x))$

Structural Induction in English

Let P be any predicate.

- If I can show P is true of all the base cases,
- and I can show that for every construction function, if P holds for the the inputs to the construction function then P must hold for the output of the construction function,

Then P holds for every element constructed from the bases cases and the construction functions.

Recovering regular induction

$\mathbb{N}$ is generated by $\{0\}$ and AddOne. So substituting $\mathbb{N}$ for C, $\{0\}$ for B and $\{\mathrm{AddOne}\}$ for F in structural induction, we get

- for every $b \in \{0\}$, $P(b)$, this is just $P(0)$
- and for every $f \in \{\mathrm{AddOne}\}$ on m inputs, for every $a_1, \ldots, a_m \in \mathbb{N}$,

  $$(P(a_1) \wedge P(a_2) \wedge \ldots \wedge P(a_m)) \implies P(f(a_1, \ldots, a_m))$$

  this is just $\forall k \in \mathbb{N}.(P(k) \implies P(k+1))$

Then $\forall x \in \mathbb{N}.(P(x))$.

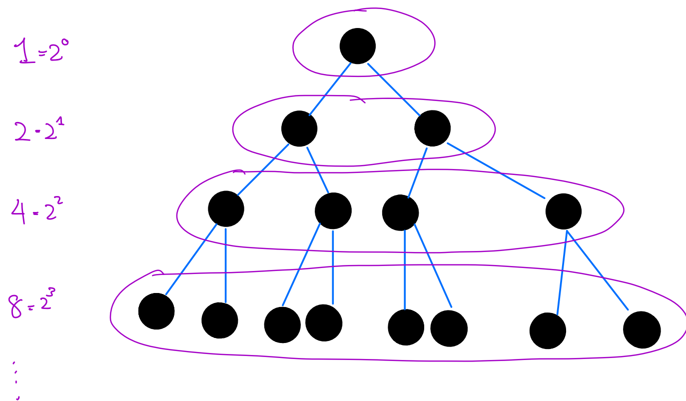
Flexibility

This more general version formalizes the intuition for why were were able to change the base cases when trying to prove, for example, $\forall n \in \mathbb{N}, n \geq 4.(P(n))$.

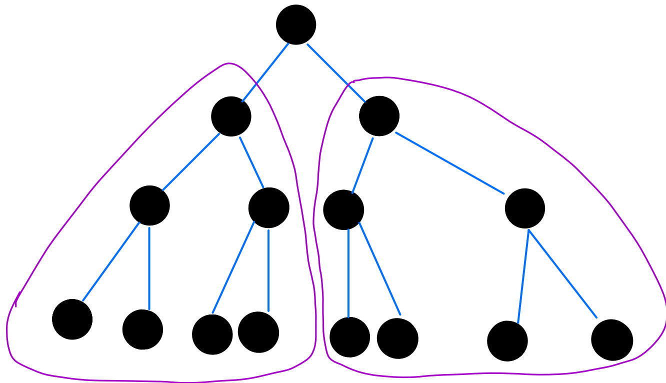
We really just showed P holds for every number in the set $\mathbb{N}_{\geq 4} = \{4, 5, 6, ...\}$ which is generated from the singleton set $\{4\}$, and the function AddOne.

Perfect Binary Trees (Again)

Last time we showed a perfect binary tree of height h has $2^{h+1} - 1$ vertices. By showing $2^0 + 2^1 + ... + 2^h = 2^{h+1} - 1$ for all $h \in \mathbb{N}$.



An alternate way of looking at things



Perfect Binary Trees

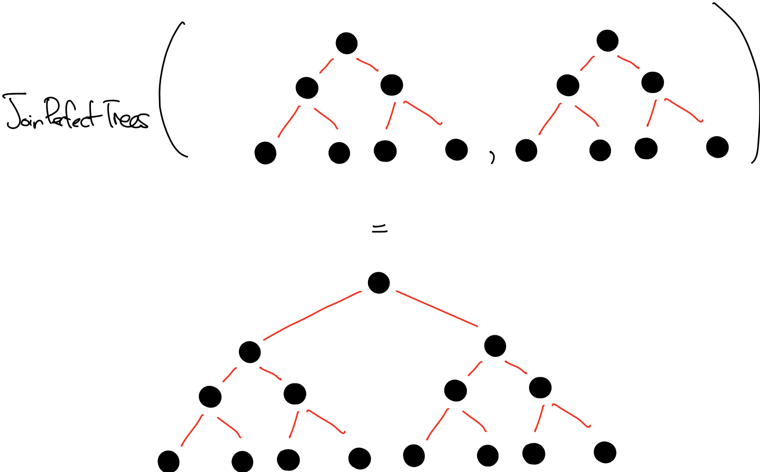
Let $\mathrm{PerfectBinaryTrees}$ be the set of perfect binary trees, and let's write it as being generated from a set by some function.

Perfect Binary Trees

Let $\text{PerfectBinaryTrees}$ be the set of perfect binary trees, and let's write it as being generated from a set by some function.

- U (for example might be the set of all graphs).
- $B = \{\text{single node}\}$
- $\text{JoinPerfectTrees} : U \times U \to U$ maps (G_1, G_2) to the tree with G_1 as left subtree and G_2 as right subtree if and only if G_1 and G_2 are perfect binary trees of the same height. Otherwise, map to the graph with a single node.

JoinPerfectTrees



A perfect binary tree of height h has $2^{h+1} - 1$ vertices

A perfect binary tree of height h has $2^{h+1} - 1$ vertices

Let $P(G)$ be the predicate that if G is a perfect binary tree of height h, then G has $2^{h+1} - 1$ vertices.

By structural induction.

Base case. The base case is the graph consisting of a single node. It has height 0 and $2^{0+1} - 1 = 1$ vertices so the base case is true.

A perfect binary tree of height h has $2^{h+1} - 1$ vertices

Inductive step. Let $G_1 = (V_1, E_1), G_2 = (V_2, E_2)$ be perfect binary trees and assume P holds for G_1 and G_2. We'll show that P also holds for $\mathrm{JoinPerfectTrees}(G_1, G_2)$.

Note that if G_1 and G_2 not are of the same height h, $\mathrm{JoinPerfectTrees}(G_1, G_2)$ is the single node graph, which is just the base case. Otherwise, G_1 and G_2 are both perfect binary trees of the same height h. By the induction hypothesis, $|V_1| = |V_2| = 2^{h+1} - 1$. $\mathrm{JoinPerfectTrees}(G_1, G_2)$ is then a perfect binary tree of height $h + 1$ with

$$1 + 2(2^{h+1} - 1) = 2^{h+2} - 1$$

vertices as required.

Postage Stamps (Again)

By structural induction.

$\mathbb{N}_{\geq 8}$ is generated by $\{8, 9, 10\}$ and $\{\text{Add3}\}$. (You need to justify this part...)

Base Case. You can make $8 = 3 + 5$, $9 = 3 \cdot 3$, $10 = 5 \cdot 2$.

Inductive Step. Let $k \in \mathbb{N}_{\geq 8}$, and assume $k = 3a + 5b$ for some $a, b \in \mathbb{N}$. Then $\text{Add3}(k) = k + 3 + 3a + 5b + 3 = 3(a + 1) + 5b$.

Structural vs. Complete Induction

If you prefer complete induction to structural induction, you can always opt to use complete induction instead. The following slides will detail why.

Construction Sequences

Let C be the set generated from B by the functions in F. Define a construction sequence of length n, to be a sequence of elements $(x_0, .., x_n)$ where for each x_i in the sequence, either

- $x_i \in B$,
- or $x_i = f(x_{j_1}, ..., x_{j_m})$ for some $f \in F$, and $j_1, ..., j_m < i$.

I.e., **every element in the sequence is either in the base set B or is constructed by applying a construction function to earlier elements in the sequence.**

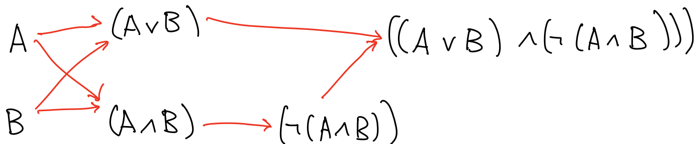
Example - construction sequence

Construction Fns : $\mathcal{E}_{\neg}$, $\mathcal{E}_{\wedge}$, $\mathcal{E}_{\vee}$

$\mathcal{E}_{\neg}(x) = (\neg x) \quad \mathcal{E}_{\wedge}(x,y) = (x \wedge y) \quad \mathcal{E}_{\vee}(x,y) = (x \vee y)$

Construct: $((A \vee B) \wedge (\neg (A \wedge B)))$

$A,\ B,\ (A \vee B),\ (A \wedge B),\ (\neg (A \wedge B)),\ ((A \vee B) \wedge (\neg (A \wedge B))),$



Structural vs. Complete Induction

Define C_i be the set where $x \in C_i$ if there exists some construction sequence of length at most i ending in x. Then $C = C_0 \cup C_1 \cup \ldots$.

Instead of doing structural induction, we can do induction on the length of the construction sequence. I.e., show that if P holds for every element with construction sequences of at most k, then P also holds for elements with construction sequences of length at most $k + 1$.

Usually, *length of construction sequence* is represented by some measure of complexity of the object, for example, 'height of a tree' or 'number of parenthesis,' or 'length of the list.'

Perfect Binary Trees (again again)

Claim. A perfect binary tree of height h has $2^{h+1} - 1$ vertices.

Base case. A perfect binary tree of height 0 has $2^{0+1} - 1 = 1$ vertices so the base case holds.

Inductive step. let $k \in \mathbb{N}$ be any natural number and assume perfect binary trees of height k have $2^{k+1} - 1$ vertices. Let T be a perfect binary tree of height $k + 1$. Note that T is constructed of an additional node joining two perfect binary trees of height k. Thus T has $2(2^{k+1} - 1) + 1 = 2^{k+2} - 1$ vertices as required.

Note: Here, we used the tree's height as a proxy for the length of the construction sequence.

Level of Formality

So far, we have seen many examples of proof by induction. You can use any approach you wish.

You don't need to talk about construction sequences in your proofs and can instead say, for example, 'by induction on the height of the tree.'

Structural induction is usually trickier to get right, so I'd recommend sticking to complete/regular induction whenever possible. We present it here since

1. It allows us to introduce iterative/recursive definitions.
2. Its generality allows us to explain some variants of regular induction (e.g., why we can start at $n = 4$ if we want to.)

Induction

Structural Induction

Well Ordering Principle and Proof by Infinite Descent

The Well Ordering Principle

Let $S \subseteq \mathbb{N}$ be a non-empty subset, a is a minimal element of S if $\forall b \in S.(a \leq b)$

The Well Ordering Principle states that for any non-empty subset $S \subseteq \mathbb{N}$, S has a minimal element.

In particular, this is true even for infinite subsets.

The Well Ordering Principle

Let $S \subseteq \mathbb{N}$ be a non-empty subset, a is a minimal element of S if $\forall b \in S.(a \leq b)$

The Well Ordering Principle states that for any non-empty subset $S \subseteq \mathbb{N}$, S has a minimal element.

In particular, this is true even for infinite subsets.

Thoughts? Is this obvious?

Well Ordering Principle

Well Ordering Principle: For any non-empty subset $S \subseteq \mathbb{N}$, S has a minimal element.

Well Ordering Principle

Well Ordering Principle: For any non-empty subset $S \subseteq \mathbb{N}$, S has a minimal element.

What if we replace $\mathbb{N}$ with $\mathbb{Q}, \mathbb{Z}, \mathbb{R}$? Is it still true?

Well Ordering Principle

Well Ordering Principle: For any non-empty subset $S \subseteq \mathbb{N}$, S has a minimal element.

What if we replace $\mathbb{N}$ with $\mathbb{Q}, \mathbb{Z}, \mathbb{R}$? Is it still true?
No, for example $\mathbb{Z} \subseteq \mathbb{Z} \subseteq \mathbb{Q} \subseteq \mathbb{R}$ has no minimal element!

Proofs using the Well Ordering Principle

The Well Ordering Principle also lets us prove statements of the form $\forall n \in \mathbb{N}.(P(n))$. Here's how:

- Check $P(0)$ is true.
- By contradiction, assume $\exists n \in \mathbb{N}.(\neg P(n))$. So the set $S = \{n \in \mathbb{N} : \neg P(n)\}$ is non-empty.
- By the Well Ordering Principle, S has a minimal element, m (i.e. m is the smallest natural number for which P doesn't hold.) Since we know $P(0)$, $m \geq 1$.
- Derive a contradiction by showing $P(m)$, or by finding a $m' < m$, for which $\neg P(m')$.

For all $n \in \mathbb{N}, n \geq 2$. n has a prime divisor

For all $n \in \mathbb{N}, n \geq 2$. n has a prime divisor

A prime divisor p of a number n is a prime number such that there exists $k \in \mathbb{N}$ such that $pk = n$.

By contradiction, assume $\exists k \in \mathbb{N}, k \geq 2$ that does have a prime divisor. Let

$$S = \{k \in \mathbb{N} : k \geq 2, \text{and } k \text{ doesn't have a prime divisor}\}$$

By WOP, S has a minimal element m. If m is prime, then m is a prime divisor of itself. Thus, m is not prime. Hence $m = ab$ for $1 < a, b < m$. Since $a < m$, and m was minimal in S, $a \notin S$, and hence a has a prime divisor c. But since $m = ab$, c is also a prime divisor of m, which contradicts the fact that $m \in S$.

$\sqrt{2}$ is irrational (a classic)

$\sqrt{2}$ is irrational (a classic)

Let $P(n)$ be the predicate, there does not exists $m \in \mathbb{N}, m \geq 1$ such that $n/m = \sqrt{2}$. If $\forall n.(P(n))$, then $\sqrt{2}$ can not be written as a fraction as is therefore irrational.

By contradiction, assume the set $S = \{n \in \mathbb{N} : \neg P(n)\}$ is non-empty. Then by the WOP, S has a minimal element x. Since $x \in S$, there exists $y \in \mathbb{N}$ such that $x/y = \sqrt{2}$. Squaring both sides, we have $x^2/y^2 = 2$. Thus, $x^2 = 2y^2$ and x must be even. Therefore $x = 2z$ for some $z \in \mathbb{N}$. But then $(2z)^2 = 2y^2$, so $2z^2 = y^2$, so y must also be even! Therefore, $(x/2)$ is an positive integer and $(y/2)$ is positive integer such that and with

$$\frac{(x/2)}{(y/2)} = \sqrt{2}.$$

Thus, $x/2 \in S$, which contradicts the minimality of x in S.

Induction in disguise

Let's take another look at the complete induction. We want to show that $P(0)$ and

$$(\forall k \in \mathbb{N}.(P(0), ..., P(k))) \implies P(k+1)$$

Usually, we prove the inductive step directly by picking an arbitrary $k \in \mathbb{N}$ and assuming $P(0) \land ... \land P(k)$, and then showing $P(k+1)$.

If we instead chose to do it by contradiction, it might look like this. Let $k \in \mathbb{N}$ be any natural number, and assume $P(0) \land ... \land P(k)$, by contradiction, assume $\neg P(k+1)$. At this point, our assumptions are

$$P(0) \land ... \land P(k) \land \neg P(k+1).$$

Induction in disguise

But

$$P(0) \wedge ... \wedge P(k) \wedge \neg P(k+1)$$

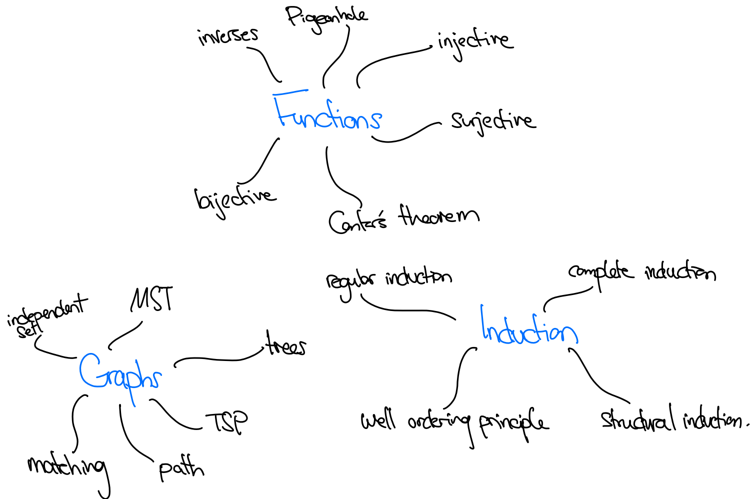
is exactly what it means for $k+1$ to be the minimal element of the set $S = \{n \in \mathbb{N} : \neg P(n)\}$.

Thus, proving the inductive step for complete induction by contradiction amounts to finding a contradiction by assuming there was a minimal element of the set $S = \{n \in \mathbb{N} : \neg P(n)\}$, which is exactly the same as what we'd do in a proof using the WOP.

Additional notes

- This presentation of structural induction loosely follows the one in *A Mathematical Introduction to Logic* by Herbert Enderton. So check that out as supplementary reading.
- Our approach for proving a mathematical statement using the Well Ordering Principle is sometimes called 'proof by infinite descent'. Read all about that here.

A trusty toolkit



CSC 236 Lecture 5: Recurrences 1

Harry Sha

June 7, 2023

Today

Recurrences

Asymptotics Review

Dominoes

Binary Search

Recurrences

Asymptotics Review

Dominoes

Binary Search

Recurrences

A recursive function is one that depends on itself. Here are some examples.

- $F(n) = F(n-1) + 1$
- $F(n) = 2F(n-1) + 1$
- $F(n) = F(n-1) + F(n-2)$
- $F(n) = 2F(n/2) + n$
- $F(n) = F(n/2) + 1$
- $F(n) = 2F(n-2) + F(n-1)$
- ...

Recursive ambiguity

There can be many functions that satisfy a single recurrence relation, for example
$F(n) = n + 5$ and $F(n) = n + 8$ both satisfy

$$F(n) = F(n-1) + 1$$

Thus, to specify a recursive function completely, we need to give it a (some) base case(s). I.e.

$$F(n) = \begin{cases} 5 & n = 0 \\ F(n-1) + 1 & n > 0 \end{cases}$$

Specifies the function $F(n) = n + 5$.

Why do we care about recurrences?

- The runtime of recursive programs can be expressed as recursive functions.
- Expressing something recursively is often an easier than expressing something explicitly.

The problem with recurrences

Here's the bad news. It's hard to answer questions like the following.

If

$$F(n) = \begin{cases} 2 & n = 0 \\ 7 & n = 1 \\ 2F(n-2) + F(n-1) + 12 & n > 1, \end{cases}$$

what is $F(100)$?

The problem with recurrences

Here's the bad news. It's hard to answer questions like the following.

If

$$F(n) = \begin{cases} 2 & n = 0 \\ 7 & n = 1 \\ 2F(n-2) + F(n-1) + 12 & n > 1, \end{cases}$$

what is $F(100)$?

We could do it - it would just take a while. Also, it is not immediately obvious what the asymptotics are. As computer scientists, we care about asymptotics. A lot.

Recursive to Explicit

Thus, once we have modelled something as a recurrence, it's still useful to convert that to an explicit definition of the same function.

Actually, since we're computer scientists, what we really care about is the asymptotics - we usually don't need a fully explicit expression.

Recursive to Explicit Examples

What is the explicit formula for

$$F(n) = \begin{cases} 1 & n = 0 \\ 2F(n-1) & n \geq 1 \end{cases}$$

Recursive to Explicit Examples

What is the explicit formula for

$$F(n) = \begin{cases} 1 & n = 0 \\ 2F(n-1) & n \geq 1 \end{cases}$$

$F(n) = 2^n$. How can we prove it?

Recursive to Explicit Examples

What is the explicit formula for

$$F(n) = \begin{cases} 1 & n = 0 \\ 2F(n-1) & n \geq 1 \end{cases}$$

$F(n) = 2^n$. How can we prove it? By induction!

Recursive to Explicit Examples

$$F(n) = \begin{cases} 1 & n = 0 \\ 2F(n-1) & n \geq 1 \end{cases}$$

Claim. $F(n) = 2^n$.

Base case. The base case holds because $F(0) = 2^0 = 1$.

Inductive step. Let $k \in \mathbb{N}$ be any natural number and assume $F(k) = 2^k$. Then we have

$$F(k+1) = 2 \cdot F(k) = 2 \cdot 2^k = 2^{k+1},$$

where the first inequality holds by the recursive definition of F and then second holds by the inductive hypothesis.

Recursive to Explicit Examples

What is the explicit formula for

$$F(n) = \begin{cases} 4 & n = 0 \\ 3 + F(n-1) & n \geq 1 \end{cases}$$

Recursive to Explicit Examples

What is the explicit formula for

$$F(n) = \begin{cases} 4 & n = 0 \\ 3 + F(n-1) & n \geq 1 \end{cases}$$

$F(n) = 4 + 3n$

Recursive to Explicit Examples

What is the explicit formula for

$$F(n) = \begin{cases} 1 & n = 0 \\ -F(n-1) & n \geq 1 \end{cases}$$

Recursive to Explicit Examples

What is the explicit formula for

$$F(n) = \begin{cases} 1 & n = 0 \\ -F(n-1) & n \geq 1 \end{cases}$$

$F(n) = (-1)^n$

Recursive to Explicit Examples

What is the explicit formula for

$$F(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ 2F(n-1) - F(n-2) + 2 & n \geq 2 \end{cases}$$

Recursive to Explicit Examples

What is the explicit formula for

$$F(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ 2F(n-1) - F(n-2) + 2 & n \geq 2 \end{cases}$$

Claim. $F(n) = n^2$.

Proof

$$F(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ 2F(n-1) - F(n-2) + 2 & n \geq 2 \end{cases}$$

Base case. The base cases hold because $F(0) = 0 = 0^2$, and $F(1) = 1 = 1^2$.

Inductive step. Let $k \in \mathbb{N}$ be any natural number with $k \geq 1$. Assume $F(i) = i^2$ for all $i \in \mathbb{N}, i \leq k$. We'll show $F(k+1) = (k+1)^2$. By the definition of F, we have

$$\begin{aligned} F(k+1) &= 2F(k) - F(k-1) + 2 \\ &= 2k^2 - (k-1)^2 + 2 \\ &= 2k^2 - k^2 + 2k - 1 + 2 \\ &= (k+1)^2 \end{aligned}$$

as required.

The functions we care about

Let's always imagine the function in question is the runtime of some algorithm. I.e., it maps the size of the input to the running time of the algorithm. Thus, we assume it has the following properties.

- domain $\mathbb{N}$.
- codomain $\mathbb{R}_{>0}$. An algorithm can't take negative time
- non-decreasing. An algorithm shouldn't get faster for larger inputs.

Recurrences

Asymptotics Review

Dominoes

Binary Search

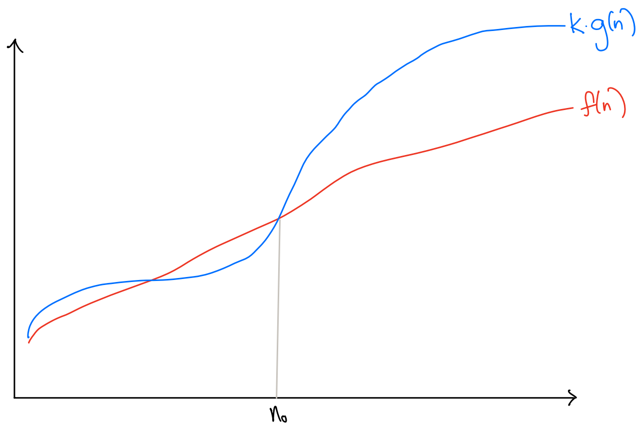
Big-O

$f = O(g)$ means

$$
\begin{aligned}
&\exists k \in \mathbb{R}_{>0}(\\
&\quad \exists n_0 \in \mathbb{N}.(\\
&\quad\quad \forall n \in \mathbb{N}.(n > n_0 \implies \\
&\quad\quad\quad f(n) \leq k \cdot g(n) \\
&\quad\quad) \\
&\quad) \\
&).
\end{aligned}
$$

Less formally, f is **at most** $kg(n)$ for large enough inputs, where k is some constant.

Big-O



In the following slides, the differences in the formal definitions from Big-O are highlighted in red.

Big-Omega

$f = \Omega(g)$ means

$$
\begin{aligned}
&\exists k \in \mathbb{R}_{>0}(\\
&\quad \exists n_0 \in \mathbb{N}.(\\
&\quad\quad \forall n \in \mathbb{N}.(n > n_0 \implies \\
&\quad\quad\quad f(n) \geq k \cdot g(n) \\
&\quad\quad) \\
&\quad) \\
&).
\end{aligned}
$$

Less formally, f is **at least** $kg(n)$ for large enough inputs, where k is some constant.

Big-Theta

$f = \Theta(g)$ means

$$
\begin{aligned}
&\exists k_1, k_2 \in \mathbb{R}_{>0}(\\
&\quad \exists n_0 \in \mathbb{N}.(\\
&\quad\quad \forall n \in \mathbb{N}.(n > n_0 \implies \\
&\quad\quad\quad k_1 \cdot g(n) \leq f(n) \leq k_2 \cdot g(n) \\
&\quad\quad) \\
&\quad) \\
&).
\end{aligned}
$$

Less formally, f is **between** $k_1 \cdot g(n)$ and $k_2 \cdot g(n)$ for large enough inputs, where k_1 and k_2 are some constants.

Equivalently, $f = O(g)$ and $f = \Omega(g)$.

Little-o

$f = o(g)$ if

$$
\begin{aligned}
&\forall k \in \mathbb{R}_{>0}(\\
&\quad \exists n_0 \in \mathbb{N}.(\\
&\quad\quad \forall n \in \mathbb{N}.(n > n_0 \implies \\
&\quad\quad\quad f(n) < k \cdot g(n) \\
&\quad\quad) \\
&\quad) \\
&).
\end{aligned}
$$

Less formally, no matter how small a constant k I multiply g by, for all large enough inputs, $f(n)$ is less than $g(n)$.

Little-omega

$f = \omega(g)$ if

$$
\begin{aligned}
&\forall k \in \mathbb{R}_{>0}(\\
&\quad \exists n_0 \in \mathbb{N}.(\\
&\quad\quad \forall n \in \mathbb{N}.(n > n_0 \implies \\
&\quad\quad\quad f(n) > k \cdot g(n) \\
&\quad\quad) \\
&\quad) \\
&).
\end{aligned}
$$

Less formally, no matter how large a constant k I multiply g by, for all large enough inputs, $f(n)$ is greater than $g(n)$.

A note about the definitions

There is some flexibility in these definitions. I.e. You can replace $<$ with $\leq$ and $>$ with $\geq$ (and vice versa) wherever your want.

You can also change the side the constant k is multiplied on if you want. I.e. multiply k to f instead of g.

It's a good exercise to prove this.

Asymptotics and orders

We can think of these asymptotics relations as

- $f = o(g)$ is like $f < g$
- $f = O(g)$ is like $f \leq g$
- $f = \Theta(g)$ is like $f \approx g$
- $f = \Omega(g)$ is like $f \geq g$
- $f = \omega(g)$ is like $f > g$

We'll sometimes use $\prec, \preceq, \approx, \succeq, \succ$ for $o, O, \Theta, \Omega, \omega$ respectively.

Logs in this class

log in this class is always $\log_2$ unless otherwise specified. It is the true inverse of the the function that maps $x \mapsto 2^x$. I.e., for any $x \in \mathbb{R}$.

$$\log(2^x) = x,$$

and for any $y \in \mathbb{R}_{>0}$

$$2^{\log(y)} = y.$$

Fast Rules

$$1 \prec \log(n) \prec n^{0.001} \prec n \prec n\log(n) \prec n^{1.001} \prec n^{1000} \prec 1.001^n \prec 2^n$$

Helpful alternative definition for little-o if you know limits:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \iff f \prec g$$

Recurrences

Asymptotics Review

Dominoes

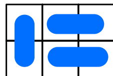
Binary Search

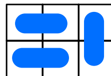
Dominoes

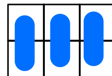
How many ways are there to tile a $2 \times n$ grid using 2×1 dominoes?

Examples

2×3.







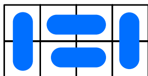
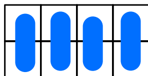
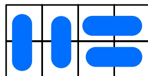
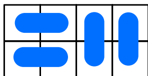
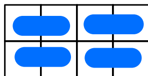
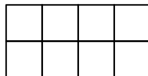
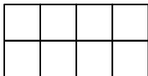
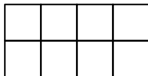
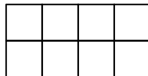
$T(3) = 3$

Number of tilings

Let $T(n)$ be the number of tilings of a $2 \times n$ grid using 2×1 dominoes.

$T(4)$

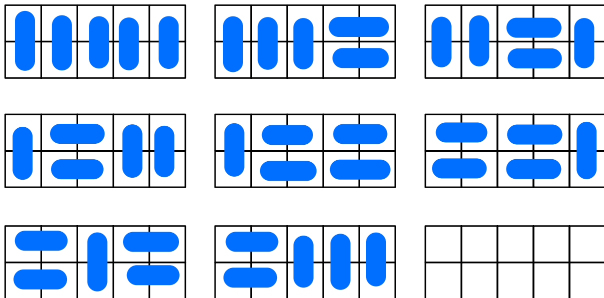
$T(4)$

$T(4) = 5$

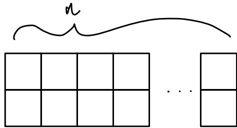
$T(5)$

$T(5)$

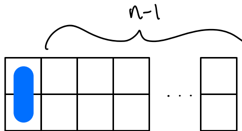


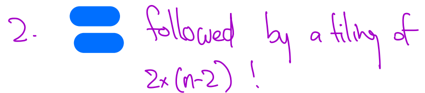
$T(5) = 8$

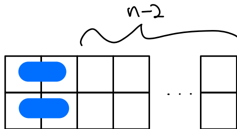
Number of tilings - Recursively



Every tiling of 2xn is either







$$\Rightarrow T(n) = T(n-1) + T(n-2).$$

Fibonacci Numbers

$$\mathrm{Fib}(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ \mathrm{Fib}(n-1) + \mathrm{Fib}(n-2) & n \geq 2 \end{cases}$$

We'll now study the asymptotics of $\mathrm{Fib}(n)$.

Note that $T(n)$ and $\mathrm{Fib}(n)$ have the same recursive relation. $T(n)$ are just the Fibonacci numbers shifted by one. I.e. $T(n) = \mathrm{Fib}(n+1)$.

An upper bound

Claim. $\forall n \in \mathbb{N}.(F(n) \leq 2^n)$

Base case. $\mathrm{Fib}(0) = 0 \leq 2^0 = 1$, and $\mathrm{Fib}(1) = 1 \leq 2^1 = 2$. Thus the base case holds.

Inductive step. Let $k \in \mathbb{N}$ with $k \geq 1$, and assume $\mathrm{Fib}(i) \leq 2^i$ for all $i \leq k$, we'll show $\mathrm{Fib}(k+1) \leq 2^{k+1}$. We have

$$\begin{aligned}
\mathrm{Fib}(k+1) &= \mathrm{Fib}(k) + \mathrm{Fib}(k-1) \\
&\leq 2^k + 2^{k-1} && (IH) \\
&\leq 2^k + 2^k \\
&= 2^{k+1}
\end{aligned}$$

so we're done.

Tightening the analysis

Claim. $\forall n \in \mathbb{N}.(F(n) \leq 2^n)$

Base case. $\mathrm{Fib}(0) = 0 \leq 2^0 = 1$, and $\mathrm{Fib}(1) = 1 \leq 2^1 = 2$. Thus the base case holds.

Inductive step. Let $k \in \mathbb{N}$ with $k \geq 1$, and assume $\mathrm{Fib}(i) \leq 2^i$ for all $i \leq k$, we'll show $\mathrm{Fib}(k+1) \leq 2^{k+1}$. We have

$$\begin{aligned}
\mathrm{Fib}(k+1) &= \mathrm{Fib}(k) + \mathrm{Fib}(k-1) \\
&\leq 2^k + 2^{k-1} && (IH) \\
&\leq^* 2^k + 2^k \\
&= 2^{k+1}
\end{aligned}$$

so we're done.

*This inequality is pretty loose!

Tightening the analysis

Let's try the same proof with $1.8 = 9/5$ instead of 2, does it still work? (Forget the base case for now).

Inductive step. Let $k \in \mathbb{N}$ with $k \geq 1$, and assume $\mathrm{Fib}(i) \leq 1.8^i$ for all $i \leq k$, we'll show $\mathrm{Fib}(k+1) \leq 1.7^{k+1}$. We have

$$\begin{aligned} \mathrm{Fib}(k+1) &= \mathrm{Fib}(k) + \mathrm{Fib}(k-1) \\ &\leq 1.8^k + 1.8^{k-1} && (IH) \\ &= 1.8^k(1 + 5/9) \\ &\leq 1.8^k(1.56) \\ &\leq 1.8^{k+1} \end{aligned}$$

so we're done.

Tightening the analysis

Let's try the same proof with $1.5 = 3/2$ instead, does that still work? (Forget the base case for now).

Inductive step. Let $k \in \mathbb{N}$ with $k \geq 1$, and assume $\mathrm{Fib}(i) \leq 1.5^i$ for all $i \leq k$, we'll show $\mathrm{Fib}(k+1) \leq 1.5^{k+1}$. We have

$$\begin{aligned}
\mathrm{Fib}(k+1) &= \mathrm{Fib}(k) + \mathrm{Fib}(k-1) \\
&\leq 1.5^k + 1.5^{k-1} && (IH) \\
&= 1.5^k(1 + 2/3) \\
&\leq 1.5^k(1.67)
\end{aligned}$$

but $1.67 \not\leq 1.5$, so we can't say $1.5^k(1.67) \leq 1.5^{k+1}$. The real answer must be somewhere in between 1.5^n and 1.8^n!

Fibonacci - Code!

Fibonacci - Code!

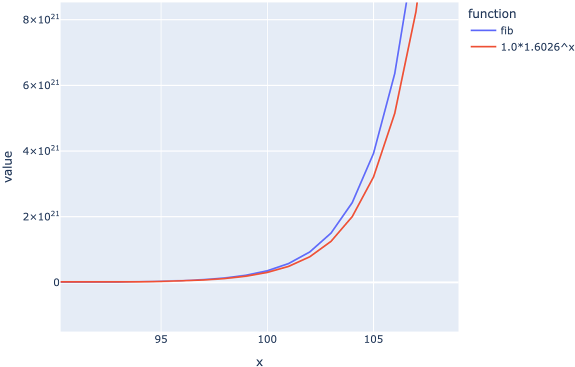
Fib vs. Exponential

Show code

base 1.60

constant 1.00

N 110

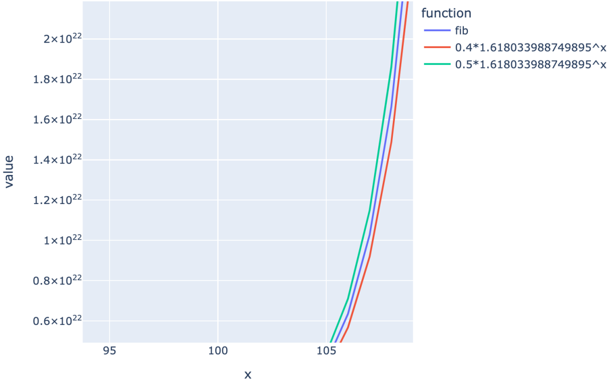


Fibonacci - Code!

constant1 0.40

constant2 0.50

N 110



Optimizing the base

Let's run the proof this time with the base of the exponent as a variable x.

Inductive step. Let $k \in \mathbb{N}$ with $k \geq 1$, and assume $\mathrm{Fib}(i) \leq x^i$ for all $i \leq k$, we'll show $\mathrm{Fib}(k+1) \leq x^{k+1}$. We have

$$\begin{aligned} \mathrm{Fib}(k+1) &= \mathrm{Fib}(k) + \mathrm{Fib}(k-1) \\ &\leq x^k + x^{k-1} && (IH) \\ &= x^k(1 + 1/x) \end{aligned}$$

To prove the inductive step, we need $x^k(1 + 1/x) \leq x^{k+1}$. I.e. $x^k + x^{k-1} \leq x^{k+1}$ dividing through by x^{k-1} and rearranging, we need $x^2 - x - 1 \geq 0$. Finding the minimum value of x for which this happens will give us a tight bound on the base.

$x^2 - x - 1 \geq 0$

Solving this quadratic, we find that $x \geq \frac{1+\sqrt{5}}{2}$ or $x \leq \frac{1-\sqrt{5}}{2}$.

Note that this means $\varphi^2 - \varphi - 1 = 0$, or $\varphi^2 = \varphi + 1$.

So the smallest positive value for x that makes this happen is $\frac{1+\sqrt{5}}{2} \approx 1.618$. This value has a name and is called φ 'phi' or the 'golden ratio'.

Upper bound

Claim. $\forall n \in \mathbb{N}.(\mathrm{Fib}(n) \leq \varphi^n)$

Base case. $\mathrm{Fib}(0) = 0 \leq \varphi^0 = 1$, and $\mathrm{Fib}(1) = 1 \leq \varphi^1 \approx 1.618$. Thus the base case holds.

Inductive step. Let $k \in \mathbb{N}$ with $k \geq 1$, and assume $\mathrm{Fib}(i) \leq \varphi^i$ for all $i \leq k$, we'll show $\mathrm{Fib}(k+1) \leq \varphi^{k+1}$. We have

$$\begin{aligned}
\mathrm{Fib}(k+1) &= \mathrm{Fib}(k) + \mathrm{Fib}(k-1) \\
&\leq \varphi^k + \varphi^{k-1} && (IH) \\
&\leq \varphi^{k-1}(\varphi + 1) \\
&= \varphi^{k-1}\varphi^2 && (\varphi^2 = \varphi + 1) \\
&= \varphi^{k+1}
\end{aligned}$$

so we're done.

A lower bound

The previous slide shows that $\mathrm{Fib}(n) = O(\varphi^n)$, we'll show here that $\mathrm{Fib}(n) = \Omega(\varphi^n)$. In particular, for all $n \in \mathbb{N}, n \geq 1.(\mathrm{Fib}(n) \geq 0.3 \cdot \varphi^n)$

Base case. For the base cases we have $\mathrm{Fib}(1) = 1 \geq 0.3 \cdot \varphi \approx 0.48$, $\mathrm{Fib}(2) = 1 \geq 0.3 \cdot \varphi^2 \approx 0.78$

Inductive step. Let $k \in \mathbb{N}$ with $k \geq 2$, and assume $\mathrm{Fib}(i) \geq 0.3\varphi^i$ for all $i \leq k$, we'll show $\mathrm{Fib}(k+1) \geq 0.3\varphi^{k+1}$. We have

$$\begin{aligned}
\mathrm{Fib}(k+1) &= \mathrm{Fib}(k) + \mathrm{Fib}(k-1) \\
&\geq 0.3(\varphi^k + \varphi^{k-1}) && (IH) \\
&= 0.3\varphi^{k-1}(\varphi + 1) \\
&= 0.3\varphi^{k-1}\varphi^2 \\
&= 0.3\varphi^{k+1}.
\end{aligned}$$

Fibonacci

$$\mathrm{Fib}(n) = \Theta(\varphi^n).$$

The complete answer - Binet's formula

$$\mathrm{Fib}(n) = \frac{\varphi^n - (1-\varphi)^n}{\sqrt{5}}$$

The complete answer - Binet's formula

$$\mathrm{Fib}(n) = \frac{\varphi^n - (1-\varphi)^n}{\sqrt{5}}$$

Note that $1 - \varphi \approx -0.618$, so the $(1-\varphi)^n$ term goes to zero really quickly and becomes irrelevant.

In fact, since $|(1-\varphi)^n/\sqrt{5}|$ is always less than $1/2$, $\mathrm{Fib}(n)$ is just $\frac{\varphi^n}{\sqrt{5}}$ rounded to the nearest whole number!

See the suggestions on slide 262 for further reading on solving recurrences exactly.

Takeaway

You can show the asymptotics of recurrences by induction!

Recurrences

Asymptotics Review

Dominoes

Binary Search

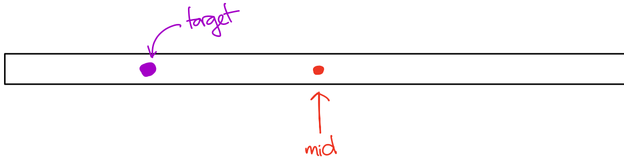
Searching in a sorted array

Inputs:

- A sorted list `l`
- A target value `target`

Output: The index of `target` in `l`. `None` if `target` is not in `l`.

Binary search intuition



if mid = target: were done.
if mid $<$ target, target is in the right half.
if mid $>$ target, target is in the left half.

Binary Search - Code

Binary Search - Code

```python
def bin_search(l, target):
  return _bin_search(l, target, 0, len(l))

def _bin_search(l, target, low, high):
  if high == low:
    return None
  else:
    mid = (low + high)//2
    if l[mid] == target:
      return mid
    elif l[mid] < target:
      return _bin_search(l, target, mid+1, high)
    elif l[mid] > target:
      return _bin_search(l, target, low, mid)
```
[7]

[7] searches `l` between indices `low` inclusive and `high` **exclusive**

Binary Search - Code

```
bin_search_verbose(l, 26)
```

```
Sublist:  [6, 7, 17, 26, 28, 30, 33, 35, 40, 51, 58, 60, 68, 78, 78, 86, 88, 92, 95, 97]
Middle value:  58
Middle value is greater than the target - looking left.

Sublist:  [6, 7, 17, 26, 28, 30, 33, 35, 40, 51]
Middle value:  30
Middle value is greater than the target - looking left.

Sublist:  [6, 7, 17, 26, 28]
Middle value:  17
Middle value is less than the target - looking right.

Sublist:  [26, 28]
Middle value:  28
Middle value is greater than the target - looking left.

Sublist:  [26]
Middle value:  26
Middle value is equal to the target - done.
3
```

Binary Search

Let $T_{\text{BinSearch}}$ be the the function that maps the length of the input array to the worst case running time of the binary search algorithm. What is the recurrence for $T_{\text{BinSearch}}$?

Binary Search

Let $T_{\text{BinSearch}}$ be the the function that maps the length of the input array to the worst case running time of the binary search algorithm. What is the recurrence for $T_{\text{BinSearch}}$?

Let's say doing a comparison and returning a value takes 1 unit of work (we could replace this with a constant amount of work c). The point is, the amount of work required to do these operations does not grow with the array length.

If $n = 0$ we return `None`, so $T_{\text{BinSearch}}(0) = 1$. In the recursive case, the value in the middle is not equal to the target. One side of the middle has $\lceil (n-1)/2 \rceil$ values and the other has $\lfloor (n-1)/2 \rfloor$. In the worst case, we call the algorithm recursively on a list of size $\lceil (n-1)/2 \rceil = \lfloor n/2 \rfloor$. Thus, for $n \geq 1$,

$$T_{\text{BinSearch}}(n) = T_{\text{BinSearch}}(\lfloor n/2 \rfloor) + 1$$

Some values

n	$T_{\mathrm{BinSearch}}(n)$

Some values

n	$T_{\mathrm{BinSearch}}(n)$
0	1
1	2
2	3
3	3
4	4
5	4
6	4
7	4
8	5
15	5
16	6

An upper bound

$$T_{\text{BinSearch}}(n) = T_{\text{BinSearch}}(\lfloor n/2 \rfloor) + 1$$

Claim. $T_{\text{BinSearch}}(n) = O(n)$. In particular, we claim for all $n \in \mathbb{N}. n \geq 1$, $T_{\text{BinSearch}}(n) \leq n + 1$.

Base case. $T_{\text{BinSearch}}(1) = 2 \leq 1 + 1$.

An upper bound

$$T_{\text{BinSearch}}(n) = T_{\text{BinSearch}}(\lfloor n/2 \rfloor) + 1$$

Claim. $T_{\text{BinSearch}}(n) = O(n)$. In particular, we claim for all $n \in \mathbb{N}. n \geq 1$, $T_{\text{BinSearch}}(n) \leq n + 1$.

Inductive step. Let $k \in \mathbb{N}, k \geq 1$, and assume $T_{\text{BinSearch}}(i) \leq i$ for all $1 \leq i \leq k$. Then we have

$$\begin{aligned} T_{\text{BinSearch}}(k+1) &= T_{\text{BinSearch}}(\lfloor (k+1)/2 \rfloor) + 1 \\ &\leq \lfloor (k+1)/2 \rfloor + 2 && (IH) \\ &\leq (k+1)/2 + 2 && (\lfloor x \rfloor \leq x) \\ &\leq 2k/2 + 2 && (k \geq 1) \\ &= (k+1) + 1 \end{aligned}$$

as required.

What should the actual runtime be?

Everytime the array gets halved! $\Rightarrow$ if we start w/ 2^n elements we get to a single element in n steps.
if we started with n elements we need $\approx \log(n)$ steps.

$T_{\text{BinSearch}}(n) = O(log(n))$

$$T_{\text{BinSearch}}(n) = T_{\text{BinSearch}}(\lfloor n/2 \rfloor) + 1$$

Claim: $T_{\text{BinSearch}}(n) = O(\log(n))$. In particular, for all $n \geq 1$, $T_{\text{BinSearch}}(n) \leq c\log(n) + d$ where c and d are constants that we will pick later.

A better upper bound

$$T_{\mathrm{BinSearch}}(n) = T_{\mathrm{BinSearch}}(\lfloor n/2 \rfloor) + 1$$

Base case. For the base case we have

$$T_{\mathrm{BinSearch}}(1) = 2 \leq c \log(1) + d$$

Note that $\log(1) = 0$ so we will need $d \geq 2$.

A better upper bound

$$T_{\text{BinSearch}}(n) = T_{\text{BinSearch}}(\lfloor n/2 \rfloor) + 1$$

Inductive step. Let $k \in \mathbb{N}$ with $k \geq 1$. Assume for all $1 \leq i \leq k$, $T_{\text{BinSearch}}(i) \leq c \log(i) + d$. Then we have

$$\begin{aligned}
T_{\text{BinSearch}}(k+1) &= T_{\text{BinSearch}}(\lfloor (k+1)/2 \rfloor) + 1 \\
&\leq c \log(\lfloor (k+1)/2 \rfloor) + d + 1 \\
&\leq c \log((k+1)/2) + d + 1 \\
&= c \log(k+1) - c \log(2) + d + 1 \\
&= c \log(k+1) - c + d + 1
\end{aligned}$$

Where we get the second line by the inductive hypothesis, the third by the fact that log is increasing and $\lfloor x \rfloor \leq x$ for all $x \in \mathbb{R}$, the fourth by log rules and the fifth by the fact that $\log(2) = 1$.

A better upper bound

$$T_{\text{BinSearch}}(n) = T_{\text{BinSearch}}(\lfloor n/2 \rfloor) + 1$$

We want

$$c \log(k+1) - c + d + 1 \leq c \log(k+1) + d.$$

Which is true, for example, when $c = 1$. Thus, $T_{\text{BinSearch}}(n) = O(\log(n))$

Tips

Here are some tips for showing $T(n) = O(f(n))$

- Try proving $T(n) \leq cf(n) + d$ for some numbers c and d. After running the proof go back and figure out what c and d need to be for your proof to work.
- Sometimes in the inductive step, you might find it helpful to assume k is larger that some constant for example, $k \geq 3$. If this is the case, show $\forall n \in \mathbb{N}, n \geq 3. T(n) \leq f(n)$, and change the base case! (This is like the $n^2 \leq 2^n$ example from two lectures ago where we used the assumption that $k \geq 4$ in the inductive step.)

The exact answer

n	$T_{\text{BinSearch}}(n)$
1	2
2	3
3	3
4	4
5	4
6	4
7	4
8	5
15	5
16	6

Write any $n \in \mathbb{N}$ with $n \geq 1$ as $2^i + x$ for some $i \in \mathbb{N}, x \in \mathbb{N}$ with $x \leq 2^i - 1$.

Claim. $\forall i \in \mathbb{N}, \forall x \in \mathbb{N}, x \leq 2^i - 1.(T_{\text{BinSearch}}(2^i + x) = i + 2)$

$T_{\mathrm{BinSearch}}(2^i + x) = i + 2$

By induction on i.

Base case. For $i = 0$, we have
$T_{\mathrm{BinSearch}}(2^0) = T_{\mathrm{BinSearch}}(1) = 2 = 0 + 2.$

$T_{\text{BinSearch}}(2^i + x) = i + 2$

Inductive step. Let $i \in \mathbb{N}$ be any integer and assume for all $x \leq 2^i - 1$, $T_{\text{BinSearch}}(2^i + x) = i + 2$. Now we consider the $i + 1$ case. Let $x \leq 2^{i+1} - 1$, then we have

$$\begin{aligned}
T_{\text{BinSearch}}(2^{i+1} + x) &= T_{\text{BinSearch}}\left(\left\lfloor \frac{2^{i+1} + x}{2} \right\rfloor\right) + 1 \\
&= T_{\text{BinSearch}}\left(2^i + \left\lfloor \frac{x}{2} \right\rfloor\right) + 1 \\
&= i + 2 + 1 \\
&= (i + 1) + 2,
\end{aligned}$$

where the third inequality holds by the inductive hypothesis since $\left\lfloor \frac{x}{2} \right\rfloor \leq 2^i - 1$, as $x \leq 2^{i+1} - 1$. This completes the induction. Note this means $T_{\text{BinSearch}}(n) = \lfloor \log(n) \rfloor + 2$ for $n \in \mathbb{N}, n \geq 1$.

Getting a good guess

Making a good guess is important in solving recurrences by induction. We'll see a method to do this next week.

Additional Notes

If you want a general method for fully solving recurrences, you'll need to study generating functions and partial fraction decomposition. See chapter 7 of *Concrete Mathematics* by Don Knuth for an excellent introduction. Or take a class on combinatorics.

CSC 236 Lecture 6: Recurrences 2

Harry Sha

June 14, 2023

Today

Recurrences

Merge Sort

The Master Theorem

Recurrences

Merge Sort

The Master Theorem

Recurrences

Last time we used induction to prove asymptotic bounds on recursive functions. For example, we showed that

$$\mathrm{Fib}(n) = \Theta(\varphi^n),$$

and

$$T_{\mathrm{BinSearch}}(n) = \Theta(\log(n)).$$

Last time's approach

Last time, the process looked like

- Guess an upper bound.
- Try to prove the upper bound.
- Try to prove a tighter upper bound or a matching lower bound.

Last time's approach

Last time, the process looked like

- Guess an upper bound.
- Try to prove the upper bound.
- Try to prove a tighter upper bound or a matching lower bound.

Here are two weaknesses to this approach.

- What if you get unlucky with your guess?
- The proofs were slow, technical and not incredibly intuitive.

Today's approach

Today we will see how to

1. Remove technical details.
2. Make better guesses.
3. Streamline the process for solving certain types of recurrences.

Technicalities

Technicalities - Base Cases

The base case typically involves calculating some values of the recursive function, and picking constants large enough so that things work out.

The base usually works out and is a little tedious to check, so for the rest of this class, you may skip this step - as long as you swear the following oath

> *I swear that I understand that a full proof by induction requires a base case*

Technicalities - Floors and Ceilings

In *divide and conquer* algorithms, we typically split up the problem in to subproblems of roughly even size. For example, we might split a problem of size n into 2 sub problems of size $n/2$. When n is not divisible by 2, this is really one subproblem of $\lfloor n/2 \rfloor$ and another of $\lceil n/2 \rceil$.

However, replacing $\lceil n/2 \rceil$ and $\lfloor n/2 \rfloor$ with $n/2$ has a negligible impact on the asymptotics so we can just ignore floors and ceilings. See *Introduction To Algorithms (CLRS)*, section 4.62 for a discussion on this.

The substitution method

The substitution method for solving recurrences is proof by (complete) induction with the simplifications applied.

The substitution method

1. Remove all the floors and ceilings from the recurrence T.
2. Make a guess for f such that $T(n) = O(f(n))$.
3. Write out the recurrence: $T(n) = \ldots$.
4. Whenever $T(k)$ appears on the RHS of the recurrence, *substitute* it with $cf(k)$.
5. Try to prove $T(n) \leq cf(n)$.
6. Pick c to make your analysis work!

The substitution method

- If you want to show $T = \Theta(f)$, you also need to show $T(n) = \Omega(f(n))$. This is the same as steps 3-6 where the $\leq$ in step 5 is replaced by a $\geq$.
- You can also add as many lower order terms as you want. I.e. you can show $T(n) = cf(n) + d$.
- The constant c that you pick when trying to show $T = \Omega(f)$ can be different to the constant that you picked when trying to show $T = O(f)$.

Today's approach

$$T_{\mathrm{BinSearch}}(n) = T_{\mathrm{BinSearch}}(n/2) + 1$$

Claim. $T_{\mathrm{BinSearch}}(n) = O(\log(n))$.

Use the substitution method.

$$\begin{aligned} T_{\mathrm{BinSearch}}(n) &= T_{\mathrm{BinSearch}}(n/2) + 1 \\ &\leq c\log(n/2) + 1 \\ &= c(\log(n) - \log(2)) + 1 \\ &= c\log(n) - c + 1, \end{aligned}$$

which is at most $c\log(n)$ when $c \geq 1$.

Recurrences

Merge Sort

The Master Theorem

The Sorting Problem

Input. A list l.

Output. l, but sorted.

The Sorting Problem

Input. A list l.

Output. l, but sorted.

Let's think of $l \in \mathrm{List}[\mathbb{N}]$, i.e. l is a list of natural numbers is sorted iff $i \leq j \implies l[i] \leq l[j]$.

In general, sorting makes sense for $l \in \mathrm{List}(A)$, as long as the elements of A can be ordered.

Merge Sort - Code

Screenshots of Code

```
def merge_sort(l):
    n = len(l)
    if n <= 1:
        return l
    else:
        left = merge_sort(l[:n//2]) # Sort the left subarray
        right = merge_sort(l[n//2:]) # Sort the right subarray
        return merge(left, right) # Merge the sorted arrays
```

Screenshots of Code

```python
def merge(l1, l2):
    """
    Input: sorted lists: l1, l2
    Output: l, a sorted list of elements from both l1 and l2
    """
    l = []
    while True:
        # If either list is empty, concatenate the other list to the end and return
        if len(l1) == 0:
            return l + l2
        if len(l2) == 0:
            return l + l1

        # Otherwise, both lists are non-empty, so append the smallest element in either list
        if l1[0] <= l2[0]:
            l.append(l1.pop(0)) # pop(0) retrives first and removes it from the list
        else:
            l.append(l2.pop(0))
```

Screenshots of Code

```
RIGHT: [7, 0, 6, 3, 2]
        START Sorting: [7, 0, 6, 3, 2]
        LEFT: [7, 0]
                START Sorting: [7, 0]
                LEFT: [7]
                        START Sorting: [7]
                        END: [7]
                RIGHT: [0]
                        START Sorting: [0]
                        END: [0]
                MERGE [7] and [0]
                END: [0, 7]
        RIGHT: [6, 3, 2]
                START Sorting: [6, 3, 2]
                LEFT: [6]
                        START Sorting: [6]
                        END: [6]
                RIGHT: [3, 2]
                        START Sorting: [3, 2]
                        LEFT: [3]
                                START Sorting: [3]
                                END: [3]
                        RIGHT: [2]
                                START Sorting: [2]
                                END: [2]
                        MERGE [3] and [2]
                        END: [2, 3]
                MERGE [6] and [2, 3]
                END: [2, 3, 6]
        MERGE [0, 7] and [2, 3, 6]
        END: [0, 2, 3, 6, 7]
MERGE [1, 4, 5, 8, 9] and [0, 2, 3, 6, 7]
END: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Merge Sort Complexity

```python
def merge_sort(l):
    n = len(l)
    if n <= 1:
        return l
    else:
        left = merge_sort(l[:n//2]) # Sort the left subarray
        right = merge_sort(l[n//2:]) # Sort the right subarray
        return merge(left, right) # Merge the sorted arrays
```

Merge Sort Complexity

```python
def merge_sort(l):
    n = len(l)
    if n <= 1:
        return l
    else:
        left = merge_sort(l[:n//2]) # Sort the left subarray
        right = merge_sort(l[n//2:]) # Sort the right subarray
        return merge(left, right) # Merge the sorted arrays
```

$$T_{MS}(n) = 2T_{MS}(n/2) + T_{\mathrm{Merge}}(n)$$

Merge Complexity

```python
def merge(l1, l2):
    """
    Input: sorted lists: l1, l2
    Output: l, a sorted list of elements from both l1 and l2
    """
    l = []
    while True:
        # If either list is empty, concatenate the other list to the end and return
        if len(l1) == 0:
            return l + l2
        if len(l2) == 0:
            return l + l1

        # Otherwise, both lists are non-empty, so append the smallest element in either list
        if l1[0] <= l2[0]:
            l.append(l1.pop(0)) # pop(0) retrives first and removes it from the list
        else:
            l.append(l2.pop(0))
```

Let n be the total number of elements in `l1` and `l2`, what is the complexity of `merge` in terms of n?

Merge Complexity

```python
def merge(l1, l2):
    """
    Input: sorted lists: l1, l2
    Output: l, a sorted list of elements from both l1 and l2
    """
    l = []
    while True:
        # If either list is empty, concatenate the other list to the end and return
        if len(l1) == 0:
            return l + l2
        if len(l2) == 0:
            return l + l1

        # Otherwise, both lists are non-empty, so append the smallest element in either list
        if l1[0] <= l2[0]:
            l.append(l1.pop(0)) # pop(0) retrives first and removes it from the list
        else:
            l.append(l2.pop(0))
```

Let n be the total number of elements in `l1` and `l2`, what is the complexity of `merge` in terms of n?

$\Theta(n)$. Explanation: each iteration of the while loop adds at least one element to the merged list.

Merge Sort Complexity

$$T_{MS}(n) = 2T_{MS}(n/2) + n$$

Merge Sort Complexity

$$T_{MS}(n) = 2T_{MS}(n/2) + n$$

Note that the $+n$ is really a $+\Theta(n)$, but since we care about asymptotics, this is another simplification that is ok!

Recurrences as Sums

$$T_{MS}(n) = 2T_{MS}(n/2) + n$$

$$\begin{aligned}
T_{MS}(n) &= 2T_{MS}(n/2) + n \\
&= 2(2T_{MS}(n/4) + n/2) + n \\
&= 4T_{MS}(n/4) + 2n \\
&= 4(2T_{MS}(n/8) + n/4) + 2n \\
&= 8T_{MS}(n/8) + 3n \\
&= 8(2T_{MS}(n/16) + n/8) + 3n \\
&= 16T_{MS}(n/16) + 4n \\
\ldots
\end{aligned}$$

Let's say $n = 2^k$ for some k. Then eventually, we get to...

$$T_{MS}(n) = 2^k T_{MS}(n/2^k) + kn = nT_{MS}(1) + kn = \Theta(n\log(n))$$

Recursion Trees

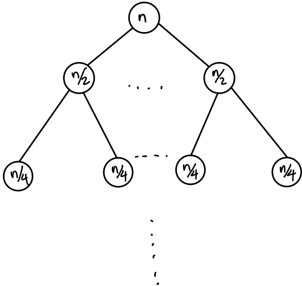
Recursion Trees are a great way to visualize the sum

Recursion Trees

$$T_{MS}(n) = 2T_{MS}(n/2) + n$$

# nodes	work/node	total
1	n	n
2	n/2	n
4	n/4	n
n	1	n

n · log(n).

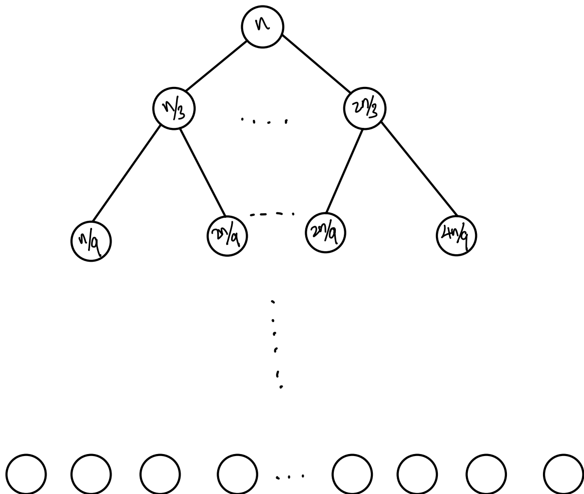


Using recursion trees

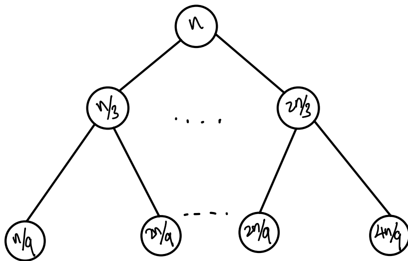
Like in the previous example, we can sometimes use the recursion tree to compute the runtime directly.

Other times, we won't be able to compute the runtime directly, but we can still use recursion trees to make a good guess. We can then prove our guess was correct using the substitution method.

$$T(n) = T(n/3) + T(2n/3) + n$$

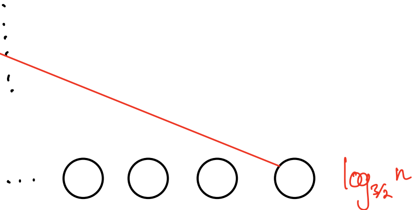


$$T(n) = T(n/3) + T(2n/3) + n$$

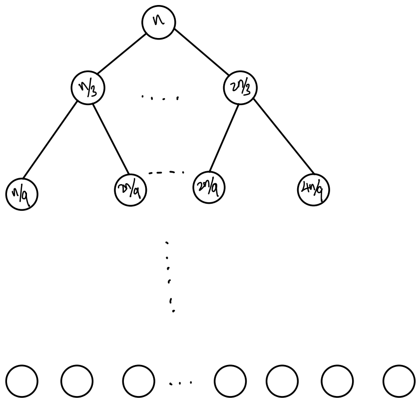


Each level still does Θ(n) work, however, the tree is not perfect!

The left side reaches the base case first!

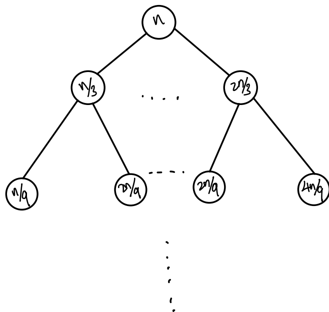


$T(n) = T(n/3) + T(2n/3) + n$



Lower bound: Remove all nodes with height $> \log_3(n)$. The remaining tree is perfect, has height $\log_3(n)$, and n work at each level. So guess $n \log(n)$.

$T(n) = T(n/3) + T(2n/3) + n$



Upper bound: Imagine levels below $\log_3(n)$ also do n work. In this case, we do n work for $\log_{3/2}(n)$ levels, so again guess $n \log(n)$.

$$T(n) = T(n/3) + T(2n/3) + n$$

Prove the guess using the substitution method (exercise).

Recurrences

Merge Sort

The Master Theorem

Standard Form Recurrences

A recurrence is in standard form if it is written as

$$T(n) = aT(n/b) + f(n)$$

For some constants $a \geq 1$, $b > 1$, and some function $f : \mathbb{N} \to \mathbb{R}$.

Most divide and conquer algorithms will have recurrences that look like this.

Thinking about the parameters

$$T(n) = aT(n/b) + f(n)$$

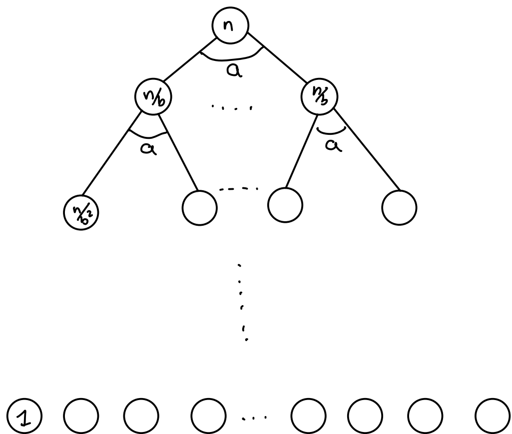
- a is the branching factor of the tree - how many children does each node have?
- b is the reduction factor - how much smaller is the subproblem in the next level of the tree compared to this level?
- $f(n)$ is the non-recursive work - how much work is done outside of the recursive call on inputs of size n? Again we make the assumption that f is positive and non-decreasing.

Recursion tree for standard form recurrences

Draw a recursion tree for the standard form recurrence. In terms of $a, b, f \ldots$

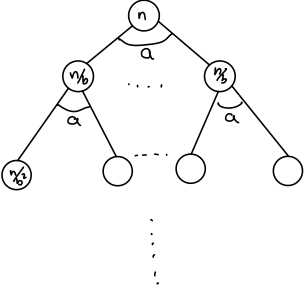
- What is the height of the tree?
- What is the number of vertices at height h?
- What is the subproblem size at height h?
- What is the total non-recursive work at level h?

Recursion tree for standard form recurrences



Recursion tree for standard form recurrences

height	non-rec. work
0	$f(n)$
1	$a^1 f(n/b^1)$
2	$a^2 f(n/b^2)$
$\log_b(n)$	$a^{\log_b(n)} f(1)$
	$= \Theta(n^{\log_b a})$



Summary

- The height of the tree is $\log_b(n)$
- The number of vertices at level h is a^h
- The total non-recursive work done at level h is $a^h f(n/b^h)$. Of note are
  - Root work. $f(n)$
  - Leaf work. $a^{\log_b(n)} \cdot f(1) = \Theta(n^{\log_b(a)})$[7].
- Summing up the levels, the total amount of work done is

$$\sum_{h=0}^{\log_b(n)} a^h f(n/b^h).$$

[7] for calculation see slide 308

The Master Theorem

The Master Theorem is a way to solve most standard form recurrences quickly.

We get the Master Theorem by analyzing the recursion tree for a generic standard form recurrence.

The Master Theorem

Theorem (The Master Theorem)

Let $T(n) = aT(n/b) + f(n)$. Define the following cases based on how the root work compares with the leaf work.

1. *Leaf heavy.* $f(n) = O(n^{\log_b(a)-\epsilon})$ *for some constant* $\epsilon > 0$.
2. *Balanced.* $f(n) = \Theta(n^{\log_b(a)})$
3. *Root heavy.* $f(n) = \Omega(n^{\log_b(a)+\epsilon})$ *for some constant* $\epsilon > 0$, *and* $af(n/b) \leq cf(n)$ *for some constant* $c < 1$ *for all sufficiently large* n.

Then,

$$T(n) = \begin{cases} \Theta(n^{\log_b(a)}) & \textit{Leaf heavy case} \\ \Theta(f(n)\log(n)) & \textit{Balanced case} \\ \Theta(f(n)) & \textit{Root heavy case} \end{cases}$$

ϵ

$f(n) = O(n^{\log_b(a)-\epsilon})$ for some $\epsilon > 0$ means that $f(n)$ is smaller than $n^{\log_b(a)}$ by a factor of at least n^ϵ. You might find it easier to think of ϵ as 0.0001, and $n^{\log_b(a)-\epsilon}$ as $\frac{n^{\log_b(a)}}{n^\epsilon}$.

For example,

$$n^{1.9} = O(n^{2-\epsilon})$$

for some $\epsilon > 0$ (e.g $\epsilon = 0.01$), but

$$n^2/\log(n) \neq O(n^{2-\epsilon})$$

for any $\epsilon > 0$ since $n^{2-\epsilon} = n^2/n^\epsilon$, and $\log(n) = O(n^\epsilon)$ for any choice of $\epsilon > 0$.

Root heavy case additional regularity condition.

The condition in the root heavy case that $af(n/b) \leq cf(n)$ for some constant $c < 1$ for all sufficiently large n is called the regularity condition.

In the root heavy case, most of the work is done at the root. $af(n/b)$ is the total work done at level 1 of the tree. The regularity condition says if most of the work is done at the root, we better do more at the root than at level 1 of the tree!

What you need to know

I will now present the proof of the Master Theorem.

In this class, you only need to know how to apply the master theorem.

However, understanding the proof is incredibly helpful for getting an intuition for the case splits, remembering the conditions, and applying the theorem.

Here we go.

Proof Outline for Master Theorem

Analyze the recursion tree for the generic standard form recurrence. Apply the case splits to f.

Geometric Series

Before we prove the Master Theorem, let's remind ourselves about geometric series. A geometric series is a sum that looks like

$$S = a + ar + ar^2 + ...ar^{n-1} = \sum_{i=0}^{n-1} ar^i$$

I.e. each term in the sum is obtained by multiplying the previous term by r.

The closed-form solution for S is

$$S = a\left(\frac{r^n - 1}{r - 1}\right)$$

Proof

$$S = a + ar + ar^2 + ...ar^{n-1}$$

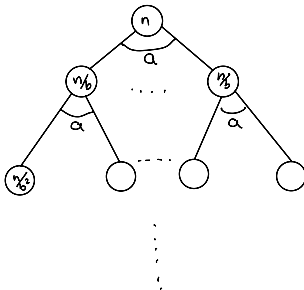
A short proof. $rS = ar + ar^2 + ...ar^n = S + ar^n - a$. Rearrange.

Balanced case

$$f(n) = \Theta(n^{\log_b(a)})$$

height	non-rec. work
0	$f(n)$
1	$a^1 f(n/b^1)$
2	$a^2 f(n/b^2)$
$\log_b(n)$	$a^{\log_b(n)} f(1)$

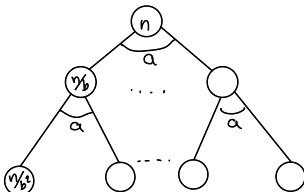
$$= \Theta(n^{\log_b a})$$



Balanced case

$$f(n) = \Theta(n^{\log_b(a)})$$

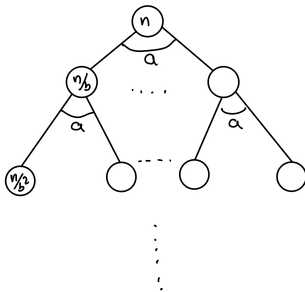
height	non-rec. work
0	$n^{\log_b a}$
1	$a\left(\frac{n}{b}\right)^{\log_b a} = \frac{a}{a} \cdot n^{\log_b a}$
2	$a^2\left(\frac{n}{b^2}\right)^{\log_b a} = \frac{a^2}{a^2} n^{\log_b a}$
⋮	⋮
$\log_b(n)$	$\Theta\left(n^{\log_b a}\right)$



Balanced case

$f(n) = \Theta(n^{\log_b(a)})$

height	non-rec. work
0	$n^{\log_b a}$
1	$n^{\log_b a}$
2	$n^{\log_b a}$
⋮	⋮
$\log_b(n)$	$\Theta(n^{\log_b a})$

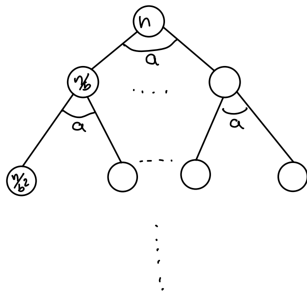


Total: $\log_b(n) \cdot n^{\log_b a} = \Theta(n^{\log_b a} \cdot \log(n))$

Leaf heavy case

$$f(n) = O(n^{\log_b(a)-\epsilon})$$

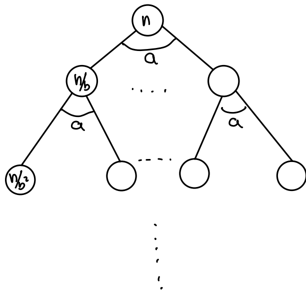
height	non-rec. work
0	$n^{\log_b a} / n^{\varepsilon}$
1	$a (n/b)^{\log_b a} / (n/b)^{\varepsilon}$
2	$a^2 (n/b^2)^{\log_b a} / (n/b^2)^{\varepsilon}$
⋮	⋮
$\log_b(n)$	$\Theta(n^{\log_b a})$



Leaf heavy case

$$f(n) = O(n^{\log_b(a)-\epsilon})$$

height	non-rec. work
0	$n^{\log_b a}/n^{\varepsilon}$
1	$n^{\log_b a}/\frac{n^{\varepsilon}}{b^{\varepsilon}}$
2	$n^{\log_b a}/\frac{n^{\varepsilon}}{b^{2\varepsilon}}$
⋮	⋮
$\log_b(n)$	$\Theta(n^{\log_b a})$



Leaf heavy case

$$f(n) = O(n^{\log_b(a) - \epsilon})$$

Total:

all levels except leaf (brace over the first term); leaf (brace over the last term)

$$\left(n^{\log_b a} / n^{\varepsilon}\right)\left(1 + b^{\varepsilon} + b^{2\varepsilon} + \dots + b^{(\log_b(n) - 1)\varepsilon}\right) + \Theta(n^{\log_b a})$$

geometric series w/ ratio b^{ε} and $\log_b(n)$ terms

$$= \left(n^{\log_b a} / n^{\varepsilon}\right) \cdot \left(\frac{(b^{\varepsilon})^{\log_b(n)} - 1}{b^{\varepsilon} - 1}\right) + \Theta(n^{\log_b a})$$

$$= \left(n^{\log_b a} / n^{\varepsilon}\right) \cdot \Theta(n^{\varepsilon}) + \Theta(n^{\log_b a})$$

$$= \Theta(n^{\log_b a})$$

Root heavy case

The third case is similar to the previous cases. Check *CLRS* section 4.6.1 for the details.

Applying the Master Theorem

1. Write the recurrence in standard form to find the parameters a, b, f
2. Compare $n^{\log_b(a)}$ to f to determine the case split.
3. Read off the asymptotics from the relevant case.

Master Theorem applied to Merge Sort

$$T_{MS}(n) = 2T_{MS}(n/2) + n$$

Master Theorem applied to Merge Sort

$$T_{MS}(n) = 2T_{MS}(n/2) + n$$

T_{MS} is a standard form recurrence with $a = 2, b = 2, f(n) = n$. We have $n^{\log_2(2)} = n^1$. Thus, $f = \Theta(n^{\log_b(a)})$ and we are in the balanced case of the Master Theorem. Hence $T_{MS}(n) = \Theta(n \log(n))$.

Master Theorem applied to Binary Search

$$T_{\mathrm{BinSearch}}(n) = T_{\mathrm{BinSearch}}(n/2) + 1$$

Master Theorem applied to Binary Search

$$T_{\mathrm{BinSearch}}(n) = T_{\mathrm{BinSearch}}(n/2) + 1$$

$T_{\mathrm{BinSearch}}$ is a standard form recurrence with $a = 1, b = 2, f(n) = 1$. We have $n^{\log_2(1)} = n^0 = 1$. Thus, we are in case 2 of the Master Theorem. Hence $T_{\mathrm{BinSearch}}(n) = \Theta(\log(n))$.

Summary of Methods

Method	Pros	Cons
Induction	Always works, can get more precision	Requires a guess, can get technical, and proofs can get quite complex.
Substitution	Always works	Requires a guess and is slower than the below.
Recursion Tree	More Intuitive/Visual	Doesn't always work but is a good starting point and good for generating guesses.
Master Theorem	Proofs are super short	Restricted scope (recurrence must be in standard form and must fall into one of the cases).

Log calculation

$$
\begin{aligned}
a^{\log_b(n)} &= a^{\frac{\log_a(n)}{\log_a(b)}} && \text{(Change of base)} \\
&= \left(a^{\log_a(n)}\right)^{1/\log_a(b)} \\
&= n^{1/\log_a(b)} \\
&= n^{\log_b(a)} && (1/\log_a(b) = \log_b(a))
\end{aligned}
$$

CSC 236 Lecture 7: Correctness

Harry Sha

July 5, 2023

Today

Correctness - Merge Sort

Multiplication

Correctness - Binary Search

Algorithm Correctness

Today, we will see how to prove algorithms are "correct".

What does it mean for an algorithm to be correct?

Correctness (formally)

For any algorithm/function/program, define a precondition and a postcondition.

- The precondition is an assertion about the inputs to a program.
- The postcondition is an assertion about the end of a program.

An algorithm is correct if the **precondition implies the postcondition**.

I.e. “If I gave you valid inputs, your algorithm should give me the expected outputs.”

This is essentially a design specification.

Documentation Analogy

```
def transpose(a, axes=None):
    """
    Reverse or permute the axes of an array; returns the modified array.

    For an array a with two axes, transpose(a) gives the matrix transpose.

    Refer to `numpy.ndarray.transpose` for full documentation.

    Parameters
    ----------
    a : array_like
        Input array.
    axes : tuple or list of ints, optional
        If specified, it must be a tuple or list which contains a permutation of
        [0,1,..,N-1] where N is the number of axes of a.  The i'th axis of the
        returned array will correspond to the axis numbered ``axes[i]`` of the
        input.  If not specified, defaults to ``range(a.ndim)[::-1]``, which
        reverses the order of the axes.

    Returns
    -------
    p : ndarray
        `a` with its axes permuted.  A view is returned whenever
        possible.
```

What is the pre/post conditions for `mergesort`(l)?

What is the pre/post conditions for `mergesort`(l)?

- Precondition: l should be a list of natural numbers.
- Postcondition: The return value of `mergesort` should contain the elements of l in sorted order.

What is the pre/post condition for `binsearch`(l, t, a, b)?

What is the pre/post condition for `binsearch`(l, t, a, b)?

- Precondition: $l \in \texttt{List}[\mathbb{N}]$, l is sorted, $t, a, b \in \mathbb{N}$, $a, b \leq \texttt{len}(l)$.
- Postcondition: If t is in $l[a:b]$ return the index of t in l, otherwise, return `None`.

How do you prove correctness for recursive functions?

How do you prove correctness for recursive functions?

By induction on the size of the inputs!

Notation

Let's use CS/Python notation. I.e., the elements of a list of length n in order are $l[0], l[1], ..., l[n-1]$.

Slicing:

$$l[i:j] = [l[i], l[i+1], ..., l[j-1]]$$

By convention, if $j \leq i$, then $l[i:j] = []$.

Conventions

Today, let's think of all lists as being lists of natural numbers.

Correctness - Merge Sort

Multiplication

Correctness - Binary Search

Merge Sort

```python
def merge_sort(l):
    n = len(l)
    if n <= 1:
        return l
    else:
        left = merge_sort(l[:n//2])
        right = merge_sort(l[n//2:])
        return merge(left, right)
```

Merge Sort - Correctness

As usual, we break this down and show for all $n \in \mathbb{N}$, if $l \in \texttt{List}[\mathbb{N}]$ is a list of length n, then `mergesort` works on l.

Correctness

$P(n)$: Let $l \in \texttt{List}[\mathbb{N}]$ be a list of natural numbers of length n, then $\texttt{mergesort}(l)$ returns the sorted list.

Claim: $\forall n \in \mathbb{N}.(P(n))$.

Correctness of Merge

For now, let's assume `merge` is correct. I.e. that on sorted lists `left` and `right`, `merge(left, right)` returns a sorted list containing all the elements in either list.

We'll come back and prove that later!

Base case

Let l be a list of length 0 or 1. Note that l is already sorted. In this case, `mergesort(`l`)` returns l as expected.

Inductive step

Let $k \in \mathbb{N}$ with $k \geq 1$, and assume for all $i \in \mathbb{N}$ with $0 \leq i \leq k$, `mergesort` works on lists of length i. We'll show that `mergesort` also works on lists of length $k+1$. Let l be a list of length $k+1$.

Inductive step

Let $k \in \mathbb{N}$ with $k \geq 1$, and assume for all $i \in \mathbb{N}$ with $0 \leq i \leq k$, `mergesort` works on lists of length i. We'll show that `mergesort` also works on lists of length $k+1$. Let l be a list of length $k+1$.

Since $k+1 \geq 2$, we fall into the else case. The left sublist is a list of length $\lfloor (k+1)/2 \rfloor$. We have

$$\begin{aligned} \lfloor (k+1)/2 \rfloor &\leq (k+1)/2 \\ &\leq (k+k)/2 && (k \geq 1) \\ &\leq k \end{aligned}$$

Thus, by the inductive hypothesis, `mergesort` correctly sorts the left sublist, and `left` contains the sorted left sublist.

Inductive Step

The right sublist is a list of length $\lceil (k+1)/2 \rceil$. Since $k \geq 1$, $\lceil (k+1)/2 \rceil \leq \lceil (k+k)/2 \rceil = k$.

Thus, by the inductive hypothesis, `mergesort` correctly sorts the right sublist, and `right` contains the sorted right sublist.

Inductive Step

The right sublist is a list of length $\lceil (k+1)/2 \rceil$. Since $k \geq 1$, $\lceil (k+1)/2 \rceil \leq \lceil (k+k)/2 \rceil = k$.

Thus, by the inductive hypothesis, `mergesort` correctly sorts the right sublist, and `right` contains the sorted right sublist.

Since we're assuming that `merge` works, and `left`, and `right` are sorted lists, and `l` is composed of the elements in `left` and `right`, we return `merge(left, right)` which is the sorted version of l.

Notes

The fact that the algorithm terminates (i.e. doesn't get stuck in an infinite loop) is implied by the statement of the claim in the word **returns**.

Correctness - Merge Sort

Multiplication

Correctness - Binary Search

Recursive Algorithms

This next example will put together what we have studied so far on recursive runtime and correctness.

Multiplication

Let's study multiplication!

If I gave you two 10 digit numbers, how would you multiply them?

Grade School Multiplcation

Lower bound for the Grade School Multiplcation Algorithm

Suppose I gave you two n-digit numbers. What is a lower bound for the runtime of the Grade School Multiplication Algorithm?

Lower bound for the Grade School Multiplcation Algorithm

Suppose I gave you two n-digit numbers. What is a lower bound for the runtime of the Grade School Multiplication Algorithm?
n^2. For each digit of the second number, I need to multiply it with every digit of the first number.

Can we do better?

Karatsuba's Algorithm

```
def karatsuba(x, y):
    if x < 10 and y < 10:
        return x * y

    n = max(len(str(x)), len(str(y)))
    m = n // 2
    x_h, x_l =  x // 10**m, x % 10**m
    y_h, y_l =  y // 10**m, y % 10**m

    z0 = karatsuba(x_l, y_l)
    z1 = karatsuba((x_l + x_h), (y_l + y_h))
    z2 = karatsuba(x_h, y_h)

    return (z2 * 10**(2*m)) + ((z1 - z2 - z0) * 10**m) + z0
```

What are `x // 10**m`, and `x % 10**m`? the first $\lceil n/2 \rceil$, and last $\lfloor n/2 \rfloor$ digits of x.

Karatsuba's Algorithm

```
def karatsuba(x, y):
    if x < 10 and y < 10:
        return x * y

    n = max(len(str(x)), len(str(y)))
    m = n // 2
    x_h, x_l =  x // 10**m, x % 10**m
    y_h, y_l =  y // 10**m, y % 10**m

    z0 = karatsuba(x_l, y_l)
    z1 = karatsuba((x_l + x_h), (y_l + y_h))
    z2 = karatsuba(x_h, y_h)

    return (z2 * 10**(2*m)) + ((z1 - z2 - z0) * 10**m) + z0
```

Trace the algorithm by hand on inputs $a = 31$ and $b = 79$, report the values of each of the variables $m, a_l, a_u, b_l, b_u, z_0, z_1, z_2$ as well as the result.

Karatsuba's Algorithm

```
def karatsuba(x, y):
    if x < 10 and y < 10:
        return x * y

    n = max(len(str(x)), len(str(y)))
    m = n // 2
    x_h, x_l =  x // 10**m, x % 10**m
    y_h, y_l =  y // 10**m, y % 10**m

    z0 = karatsuba(x_l, y_l)
    z1 = karatsuba((x_l + x_h), (y_l + y_h))
    z2 = karatsuba(x_h, y_h)

    return (z2 * 10**(2*m)) + ((z1 - z2 - z0) * 10**m) + z0
```

Write a recurrence for the runtime of Karatsuba's algorithm. Solve the recurrence. $T(n) = 3T(n/2) + n$, which is $\Theta(n^{\log_3(2)})$ by the Master Theorem - this is asymptotically better than the Grade School Algorithm!

Correctness

Precondition. $x, y \in \mathbb{N}$, **Postcondition.** Return xy

- $m = \lfloor n/2 \rfloor$, $x_h = \lfloor x/10^m \rfloor$, $x_l = x\%10^m$
- $z_0 = x_l y_l$, $z_1 = (x_l + x_h)(y_l + y_h)$, $z_2 = x_h y_h$
- `return` $(z_2 \cdot 10^{2m}) + ((z_1 - z_2 - z_0) \cdot 10^m) + z_0$

$P(n)$: If $\max(x, y) = n$, then $\texttt{karat}(x, y)$ returns xy. We'll show $\forall n \in \mathbb{N}, n \geq 1. P(n)$

Base case(s). We will show $P(0), ..., P(9)$ In these cases, both x and y are a single digit and we enter the base case and return xy as required.

Correctness

Precondition. $x, y \in \mathbb{N}$, **Postcondition.** Return xy

- $m = \lfloor n/2 \rfloor$, $x_h = \lfloor x/10^m \rfloor$, $x_l = x\%10^m$
- $z_0 = x_l y_l$, $z_1 = (x_l + x_h)(y_l + y_h)$, $z_2 = x_h y_h$
- `return` $(z_2 \cdot 10^{2m}) + ((z_1 - z_2 - z_0) \cdot 10^m) + z_0$

Inductive step. Let k be an arbitrary natural number with $k \geq 10$, and suppose $P(0), ..., P(k-1)$. We'll show $P(k)$. To apply the inductive hypotheses to the recursive calls, we need $x_h, x_l, y_h, y_l, x_h + x_l, y_h + y_l$ to all be $< k$. Since $k \geq 10$, we have that $m \geq 1$.

Since $x \leq k$, and $x = x_h 10^m + x_l$, we have $x_h + x_l < k$. Additionally, since x_h, x_l are non-negative, they must also be individually less than k. Same for y_h and y_l. Thus, the inductive hypothesis applies to the recursive calls, and z_0, z_1, z_2 contain the correct products.

Correctness

Precondition. $x, y \in \mathbb{N}$, **Postcondition.** Return xy

- $m = \lfloor n/2 \rfloor$, $x_h = \lfloor x/10^m \rfloor$, $x_l = x \% 10^m$
- $z_0 = x_l y_l$, $z_1 = (x_l + x_h)(y_l + y_h)$, $z_2 = x_h y_h$
- `return` $(z_2 \cdot 10^{2m}) + ((z_1 - z_2 - z_0) \cdot 10^m) + z_0$

Inductive step cont... Then, the return value is

$$
\begin{aligned}
&(z_2 \cdot 10^{2m}) + ((z_1 - z_2 - z_0) \cdot 10^m) + z_0 \\
&= x_h y_h 10^{2m} + ((x_h + x_l)(y_l + y_h) - x_h y_h - x_l y_l) \cdot 10^m + x_l y_l \\
&= x_h y_h 10^{2m} + (x_h y_l + y_h x_l) \cdot 10^m + x_l y_l \\
&= (x_h 10^m + x_l)(y_h 10^m + y_l) \\
&= xy
\end{aligned}
$$

An alternate $P(n)$

Instead of letting n be the maximum value of x and y, could we have set n to be the maximum length of x and y? You will run into trouble in the inductive step. For example if $x = 99$, then $x_l = x_h = 9$, which have fewer digits, but $x_l + x_h = 18$ which is still 2 digits long so you can't use the inductive hypothesis! You can fix this by extending the base case to 3, but this is more work!

Summary: Correctness for Recursive Algorithms

Prove the correctness of recursive algorithms by *induction*. The link between recursive algorithms and inductive proofs is strong.

- The base case of the recursive algorithm corresponds to the inductive proof's base case(s).
- The recursive case of the recursive algorithm corresponds to the inductive step.
- The 'leap of faith' in believing that the recursive calls works correspond to the inductive hypothesis.

Correctness - Merge Sort

Multiplication

Correctness - Binary Search

Binary Search

```
def bin_search(l, t, a, b):
    if b == a:
        return None
    else:
        m = (a + b)//2
        if l[m] == t:
            return m
        elif l[m] < t:
            return bin_search(l, t, m+1, b)
        elif l[m] > t:
            return bin_search(l, t, a, m)
```

Correctness Claim: First attempt

$P(n)$: If $l \in$ `List`$[\mathbb{N}]$ is a list of length n, `binsearch`$(l, t, a = 0, b = n)$ returns the index of t if t is in l and `None` otherwise.

Claim: for all $n \in \mathbb{N}.(P(n))$.

Base case

Consider the $n = 0$ case. let l be any list of length 0, t be any object, and consider `binsearch`$(l, t, 0, 0)$.

The check `a==b` is true since both variables are 0 so we return `None`. This is the expected result since an empty list surely does not contain t.

Inductive Step

Let $k \in \mathbb{N}$ and assume for all $i \in \mathbb{N}, i \leq k$, $\texttt{binsearch}(l, t, 0, i)$ returns the desired result for all lists of length i.

Let $l \in \texttt{List}[\mathbb{N}]$ be a list of length $k + 1$, and let $t \in \mathbb{N}$. Consider the execution of $\texttt{binsearch}(l, t, 0, k + 1)$. Since $k + 1 \geq 1$, the if condition fails. Let $m = (k + 1)//2 = \lfloor (k + 1)/2 \rfloor$ There are then 3 cases.

- Case 1. $\texttt{l[m]} == \texttt{t}$.
- Case 2. $\texttt{l[m]} < \texttt{t}$.
- Case 3. $\texttt{l[m]} > \texttt{t}$.

Case 1. `l[m] == t`

In this case `binsearch` returns m, which is indeed the index of t in l.

Case 2. `l[m]` $<$ `t`

In this case, we return `binsearch`$(l, t, m+1, k+1)$.

Case 2. `l[m]` $<$ `t`

In this case, we return `binsearch`$(l, t, m+1, k+1)$.

...

Case 2. `l[m]` $<$ `t`

In this case, we return $\texttt{binsearch}(l, t, m+1, k+1)$.

...

Here was our inductive hypothesis.

> Let $k \in \mathbb{N}$ and assume for all $i \in \mathbb{N}, i \leq k$, $\texttt{binsearch}(l, t, 0, i)$ returns the desired result for all lists of length i.

The inductive hypothesis doesn't apply here! Since $a \neq 0$!

Case 2. `l[m]` $<$ `t`

In this case, we return `binsearch`$(l, t, m+1, k+1)$.

...

Here was our inductive hypothesis.

> Let $k \in \mathbb{N}$ and assume for all $i \in \mathbb{N}, i \leq k$, `binsearch`$(l, t, 0, i)$ returns the desired result for all lists of length i.

The inductive hypothesis doesn't apply here! Since $a \neq 0$!

How can we fix it?

A fix that doesn't quite work

Instead of calling `binsearch`$(l, t, m+1, k+1)$ make the recursive call

$$\texttt{binsearch}(l[m+1:k+1], t, 0, k+1)$$

Why doesn't this work?

A fix that doesn't quite work

Instead of calling `binsearch`$(l, t, m+1, k+1)$ make the recursive call

$$\texttt{binsearch}(l[m+1:k+1], t, 0, k+1)$$

Why doesn't this work?

The index of t in $l[m+1:k+1]$ is different from the index of t in l!

Correctness Claim, Corrected

Instead of doing induction on the length of the list, do induction on the length of the search window!
$P(n)$: For all lists $l \in \texttt{List}[\mathbb{N}]$ and $t \in \mathbb{N}$, if $b - a = n$, then $\texttt{binsearch}(l, t, a, b)$ returns `None` if t is not in $l[a : b]$ and the index of t in l otherwise.

Claim: $\forall n \in \mathbb{N}. P(n)$.

Base Case

Let l be any list and t suppose $b - a = 0$. Then $l[a : b] = []$, so t can is not in l and we expect the algorithm to return `None`.

Indeed, since `b == a`, the first if check passes and `binsearch`(l, t, a, b) returns `None`.

Inductive Step

$P(n)$: For all sorted lists $l \in \texttt{List}[\mathbb{N}]$ and $t \in \mathbb{N}$, if $b - a = n$, then $\texttt{binsearch}(l, t, a, b)$ returns None if t is not in $l[a:b]$ and the index of t in l otherwise.

Let $k \in \mathbb{N}$ with $k \geq 1$, and assume for all $i \in \mathbb{N}$, $i < k$, $P(i)$. We'll show $P(k)$. Let $l \in \texttt{List}[\mathbb{N}]$ be a sorted list, and $t, a, b, \in \mathbb{N}$ such that $b - a = k$.

We'll show that $\texttt{binsearch}(l, t, a, b)$ returns None if t is not in $l[a:b]$ and the index of t in l otherwise

Since $b - a = k \geq 1$, the first if check fails. Let $m = \texttt{(a + b)//2} = \lfloor (a + b)/2 \rfloor$. There are then 3 cases.

```
l[m] == t
```

$P(n)$: For all sorted lists $l \in \texttt{List}[\mathbb{N}]$ and $t \in \mathbb{N}$, if $b - a = n$, then $\texttt{binsearch}(l, t, a, b)$ returns None if t is not in $l[a : b]$ and the index of t in l otherwise.

In this case, the algorithm returns m, which is the index of t in l.

```
l[m] == t
```

$P(n)$: For all sorted lists $l \in \texttt{List}[\mathbb{N}]$ and $t \in \mathbb{N}$, if $b - a = n$, then $\texttt{binsearch}(l, t, a, b)$ returns None if t is not in $l[a:b]$ and the index of t in l otherwise.

In this case, the algorithm returns m, which is the index of t in l.

To prove the exact form of the statement, we need to check that $l[m] = t$ is actually in $l[a:b]$. I.e. that $a \leq m \leq b - 1$.

`l[m] == t`

$P(n)$: For all sorted lists $l \in \texttt{List}[\mathbb{N}]$ and $t \in \mathbb{N}$, if $b - a = n$, then $\texttt{binsearch}(l, t, a, b)$ returns `None` if t is not in $l[a:b]$ and the index of t in l otherwise.

We have

$$
\begin{aligned}
m &= \lfloor (a+b)/2 \rfloor \\
&\geq \lfloor (2a+1)/2 \rfloor && (b - a \geq 1) \\
&\geq \lfloor a + (1/2) \rfloor \\
&\geq a
\end{aligned}
$$

`l[m] == t`

$P(n)$: For all sorted lists $l \in \texttt{List}[\mathbb{N}]$ and $t \in \mathbb{N}$, if $b - a = n$, then $\texttt{binsearch}(l, t, a, b)$ returns `None` if t is not in $l[a:b]$ and the index of t in l otherwise.

On the other side, we have

$$\begin{aligned} m &= \lfloor (a+b)/2 \rfloor \\ &\le \lfloor (2b-1)/2 \rfloor && (b - a \ge 1) \\ &\le \lfloor b - (1/2) \rfloor \\ &= b - 1 \end{aligned}$$

`l[m] < t`

$P(n)$: For all sorted lists $l \in \texttt{List}[\mathbb{N}]$ and $t \in \mathbb{N}$, if $b - a = n$, then $\texttt{binsearch}(l, t, a, b)$ returns None if t is not in $l[a : b]$ and the index of t in l otherwise.

Since l is sorted and t is greater than the $l[m]$, if t is to be in $l[a : b]$, it must have an index greater than m. So $\texttt{binsearch}(l, t, a, b) = \texttt{binsearch}(l, t, m + 1, b)$.

We claim that our inductive hypothesis applies to $\texttt{binsearch}(l, t, m + 1, b)$. We just need to show $b - (m + 1) < k$.

`l[m] < t`

$P(n)$: For all sorted lists $l \in \texttt{List}[\mathbb{N}]$ and $t \in \mathbb{N}$, if $b - a = n$, then $\texttt{binsearch}(l, t, a, b)$ returns `None` if t is not in $l[a : b]$ and the index of t in l otherwise.

WTS: $b - (m + 1) \leq k$. From the previous part, we know that $m \geq a$.

`l[m] < t`

> $P(n)$: For all sorted lists $l \in \texttt{List}[\mathbb{N}]$ and $t \in \mathbb{N}$, if $b - a = n$, then $\texttt{binsearch}(l, t, a, b)$ returns None if t is not in $l[a:b]$ and the index of t in l otherwise.

WTS: $b - (m+1) \leq k$. From the previous part, we know that $m \geq a$.

Then $b - (m+1) \leq b - (a+1) \leq b - a - 1 = k - 1$. Thus, the inductive hypothesis applies to $\texttt{binsearch}(l, t, m+1, b)$, and we are done!

`l[m] > t`

$P(n)$: For all sorted lists $l \in \texttt{List}[\mathbb{N}]$ and $t \in \mathbb{N}$, if $b - a = n$, then $\texttt{binsearch}(l, t, a, b)$ returns `None` if t is not in $l[a : b]$ and the index of t in l otherwise.

Since l is sorted and t is less than the $l[m]$, if t is to be in $l[a : b]$, it must have an index less than m. So $\texttt{binsearch}(l, t, a, b) = \texttt{binsearch}(l, t, a, m)$.

We claim that our inductive hypothesis applies to $\texttt{binsearch}(l, t, a, m)$. We need to show $m - a \leq k$.

`l[m] > t`

$P(n)$: For all sorted lists $l \in \texttt{List}[\mathbb{N}]$ and $t \in \mathbb{N}$, if $b - a = n$, then $\texttt{binsearch}(l, t, a, b)$ returns `None` if t is not in $l[a : b]$ and the index of t in l otherwise.

WTS: $m - a \leq k$. From the previous part, we have $m \leq b - 1$.

`l[m] > t`

$P(n)$: For all sorted lists $l \in \texttt{List}[\mathbb{N}]$ and $t \in \mathbb{N}$, if $b - a = n$, then $\texttt{binsearch}(l, t, a, b)$ returns $\texttt{None}$ if t is not in $l[a : b]$ and the index of t in l otherwise.

WTS: $m - a \leq k$. From the previous part, we have $m \leq b - 1$.

Then $m - a \leq b - 1 - a \leq k - 1$, so the inductive hypothesis holds for $\texttt{binsearch}(l, t, a, m)$, and we are done!

CSC 236 Lecture 8: Correctness 2

Harry Sha

July 12, 2023

Today

Correctness for Iterative Algorithms

Correctness of `merge`

Correctness for Iterative Algorithms

Correctness of `merge`

Iterative Algorithms

Iterative Algorithms are algorithms with a `for` loop or a `while` loop in them.

Conventions

```
i = 0
while i < N:
    ...
    i += 1
```

- "After the kth iteration" refers to the point in the execution of the program just before the loop condition is evaluated for the $k+1$st time.
- "Before the $k+1$th iteration" is the exact same thing as "after the kth iteration"
- For example, after the kth iteration, the value of i is k

A multiplication algorithm for natural numbers

```
def mult(x, y):
    i = 0
    total = 0
    while i < x:
        total = total + y
        i += 1
    return total
```

What is the pre/post condition for `mult`(x, y)?

What is the pre/post condition for `mult`(x, y)?

- Precondition: $x, y \in \mathbb{N}$
- Postcondition: return xy.

Proof

Assume the precondition that $x, y \in \mathbb{N}$, we'll show that on input x, y, `mult`(x, y) return xy.

Let $\texttt{total}_n$ and i_n be the value of the variables `total` and i after the nth iteration. Let $P(n)$ be the following predicate: after the nth iteration,

a. $i = n$, and
b. $\texttt{total} = ny$

We'll start by showing $\forall n \in \mathbb{N}. P(n)$. If there is no nth iteration, $P(n)$ is vacuously true.

By induction on n.

Proof

$P(n)$:

a. $i = n$, and
b. `total` $= ny$

Base Case. We'll start with $P(0)$. After the 0th iteration (before the first iteration), we have $i = 0$, an `total` $= 0$, so the base case holds.

Proof

$P(n)$:

a. $i = n$, and
b. $\texttt{total} = ny$

Inductive Step. Let $k \in \mathbb{N}$ be a natural number and suppose $P(k)$ is true, we'll show $P(k+1)$. If there was no $k+1$th iteration, then $P(k+1)$ is vacuously true, so suppose the $k+1$ iteration ran. Then we have

$$\texttt{total}_{k+1} = \texttt{total}_k + y = ky + y = (k+1)y,$$

and

$$i_{k+1} = i_k + 1 = k + 1.$$

This completes the induction, and hence $\forall n. P(n)$.

Proof

$P(n)$:

a. $i = n$, and
b. `total` $= ny$

Then, since $x \in \mathbb{N}$, we have $P(0), P(1), ..., P(x)$, by a.) we have that the loop condition passes after the ith iteration for all $i < x$, and fails after the xth iteration at which point we return the value of `total` after the xth iteration which by b.) is equal to xy.

Convention

Use subscripts to denote the value of a variable after iteration i.
E.g., $\texttt{total}_i$ is the value of the variable `total` after iteration i.

General Strategy

Define a loop invariant - some property that is true at the end of every iteration. Note that it can depend on the iteration number. Call it, for example, $P(n)$. Another common one is $P(i)$ if you use i as the iteration counter.

Tip: Since the value of variables in code can change at each iteration, it is useful to use the convention in the previous slide to refer to the value of a variable after a certain iteration.

General Strategy

Prove the following:

Initialization. Show that the loop invariant is true at the start of the loop if the precondition holds.

Maintenance. Show that if the loop invariant is true at the start of any iteration, it is also true at the start of the next iteration.

Termination. Show that the loop terminates and that when the loop terminates, the loop invariant applied to the last iteration implies the postcondition.

Runtime?

```
def mult(x, y):
    i = 0
    total = 0
    while i < x:
        total = total + y
        i += 1
    return total
```

Let's say x and y are both n-**digit** numbers and it takes time $O(n)$ time to add two n digit numbers.

What is the worst-case time complexity of `mult` in terms of n, the number of digits?

Runtime?

Since y is a n digit number, it can be as large as 999...99 (n-times), which is equal to $10^n - 1 = \Theta(10^n)$. Thus, the loop runs for $O(10^n)$ iterations!

The eventual result has as many $2n$ digits. Thus, each addition takes time $O(2n) = O(n)$. In total, the running time is $\Theta(n10^n)$.

This is terrible. For reference, Grade School Multiplication gets $O(n^2)$, and Karatsuba's Algorithm from last week gets $O(n^{1.59})$. The best-known algorithm for multiplying has runtime $O(n \log(n))$. By the way, this fast algorithm was just discovered in 2019 and published in 2021!

for Loops

`for` loops are another type of loop. You can think of loops as while loops with an appropriate loop condition. For example

```
for i in range(0, 10):
    ...

# is the same as

i = 0
while i < 10:
    ...
    i+1
```

Termination

Termination can usually be proved as a consequence of the loop invariant.

Usually, the argument will go something like this.

- By contradiction, suppose the loop didn't terminate. Then it reaches iteration N (where N is some value you chose, big enough to derive a contradiction).
- Then the loop invariant $P(N)$ implies that the value of some variables is something. This implies the loop condition will be false in the next iteration, which is a contradiction.

Termination

If you're more precise, you can often find the exact number of iterations using the Loop Invariant. That looks something like

- Claim: The loop exits after the Nth iteration
- Let $i < N$, $P(i)$ implies that the loop condition is true.
- Furthermore $P(N)$ implies that the loop condition is false. Therefore, the loop exists after the Nth iteration.

Mystery algorithm

What does the following algorithm do?
Precondition: $x, y \in \mathbb{N}$, $y > 0$.

```python
def mystery(x, y):
    val = 0
    c = 0
    while val < x:
        val += y
        c += 1
    return c
```

Mystery algorithm

What does the following algorithm do?
Precondition: $x, y \in \mathbb{N}$, $y > 0$.

```
def mystery(x, y):
    val = 0
    c = 0
    while val < x:
        val += y
        c += 1
    return c
```

Postcondition: Returns $\lceil x/y \rceil$.

Proof of Correctness

Loop Invariant. $P(n)$ is the following predicate. After the nth iteration

a.) $c = n$

b.) $\texttt{val} = ny$

We'll show $\forall n \in \mathbb{N}.P(n)$

Proof of Correctness

$P(n)$: After the nth iteration

a.) $c = n$

b.) $\texttt{val} = ny$

Initialization. For $n = 0$, c and `val` are both initialized to be 0, so $c = 0$, and $\texttt{val} = 0 \cdot y = 0$. Thus, the base case holds.

Proof of Correctness

$P(n)$: After the nth iteration

a.) $c = n$

b.) $\texttt{val} = ny$

Maintenance. Suppose $P(k)$, we'll show that $P(k+1)$ also holds. If the $k+1$th iteration did not run, $P(k+1)$ is vacuously true. Suppose the $k+1$th iteration runs. Then, the variables are updated as follows

- $c_{k+1} = c_k + 1 = k + 1$, and
- $\texttt{val}_{k+1} = \texttt{val}_k + y = ky + y = (k+1)y$

this completes the induction.

Proof of Correctness

$P(n)$: After the nth iteration

a.) $c = n$

b.) $\texttt{val} = ny$

Termination. We claim that the loop terminates after at most $n = \lceil x/y \rceil$ iterations. Indeed, $P(n)$.b implies that after the nth iteration, `val` is equal to $\lceil x/y \rceil \cdot y \geq x$. Thus, the loop terminates. Let n be the last iteration that runs so that we return $c_n = n$. Since the n was the last iteration that runs by the loop condition, we have

$$\texttt{val}_{n-1} < x \leq \texttt{val}_n \implies (n-1)y < x \leq ny \implies n - 1 < x/y \leq n.$$

In other words, n is the first integer greater than or equal to x/y, i.e., $\lceil x/y \rceil$.

Convention

If the predicate $P(n)$ has multiple parts like

a.) ...

b.) ...

Use $P(n)$.a, $P(n)$.b,... to refer to specific parts of the predicate.

Variations

- One after the other. Prove the correctness of each loop in sequence.
- Nested loops. "inside out". Decompose (or imagine) the inner loop as a separate function. Prove the correctness of that function as a lemma, and then prove the correctness of the outer loop. We will see an example in the tutorial.

Another way to prove termination: descending sequence

Another way to prove termination is to define a descending sequence of natural numbers, $a_1, a_2, \ldots$ indexed by the iteration number.

Another way to prove termination: descending sequence

Another way to prove termination is to define a descending sequence of natural numbers, $a_1, a_2, ...$ indexed by the iteration number.

By the WOP, this sequence must be finite; otherwise, the set $\{a_1, a_2, ...\}$ has no minimal element!

Example

How can we define a descending sequence of natural numbers for this algorithm?

```python
def mystery(x, y):
    val = 0
    c = 0
    while val < x:
        val += y
        c += 1
    return c
```

Idea: $a_n = x + y - \texttt{val}_n$. Where $\texttt{val}_n$ is the value of `val` at the start of iteration n.

Descending Sequence

Claim. $a_n \in \mathbb{N}$ and is decreasing.

By induction. The base case holds from the precondition.

Assuming the claim is true at the start of iteration k, we'll show that it is also true at the start of iteration $k+1$. We have

$$\begin{aligned} a_{k+1} &= x + y - \mathtt{val}_{k+1} \\ &= x + y - \mathtt{val}_k - y \\ &= a_k - y \\ &< a_k && (y > 0) \end{aligned}$$

Furthermore, canceling the ys in the second line, we get $a_{k+1} = x - \mathtt{val}_k$, which is greater than 0 since the `while` check passes. This combined with the fact that $y \in \mathbb{N}$ and $a_k \in \mathbb{N}$ implies that $a_{k+1} \in \mathbb{N}$.

Thus, a_n is indeed a decreasing sequence of natural numbers, and the algorithm terminates. (Then argue again that the LI implies the postcondition after the loop ends.)

Proofs of termination: as a part of the LI vs. descending sequence

Most of the time, the LI will imply termination, saving you from having to do another induction proof. I prefer this method.

Proofs of termination: as a part of the LI vs. descending sequence

Most of the time, the LI will imply termination, saving you from having to do another induction proof. I prefer this method.

However, it is easier to define a descending sequence of natural numbers in some cases - we'll see some examples in the tutorial.

Correctness for Iterative Algorithms

Correctness of `merge`

Merge

```python
def merge(x, y):
    l = []
    while len(x) > 0 or len(y) > 0:
        if len(x) > 0 and len(y) > 0:
            if y[0] <= x[0]:
                l.append(y.pop(0)) # 1.
            else:
                l.append(x.pop(0)) # 2.
        elif len(x) == 0:
            l.append(y.pop(0)) # 3.
        else: # len(y) == 0
            l.append(x.pop(0)) # 4.
    return l
```

Correctness of Merge

- Precondition?
- Postcondition?

Correctness of Merge

- Precondition: x and y are sorted lists.
- Postcondition: A sorted list containing the elements from x and y.

Counters

For a list l of natural numbers, let `Counter`(l) be a mapping of the elements of l to the number of times they appear. Ways to think about this

- collections.Counter
- `Counter`(l) can be thought of as a multiset (an unordered collection of objects where the same object can appear multiple times)
- `Counter`$(l) : \mathbb{N} \to \mathbb{N}$ where `Counter`$(l)(x)$ is the number of times x appears in l.

Counters

We can use Counters to express the pre and postconditions more formally.

Precondition. x and y are sorted lists of natural numbers.

Postcondition. Returns a sorted list l such that $\texttt{Counter}(l) = \texttt{Counter}(x + y)$, note that the $+$ here is concatenation of lists. This means the returned list is sorted and contains all the elements in x and y with the correct frequencies.

Correctness of `merge`

Loop Invariant.

$P(n)$: After the nth iteration,

a.) $(a \in x_n + y_n \land b \in l_n) \implies a \geq b$.

b.) $\texttt{Counter}(x_n + y_n + l_n) = \texttt{Counter}(x_0 + y_0)$.

c.) $\texttt{len}(l_n) = n$.

d.) x_n, y_n, l_n are all sorted.

Correctness of `merge`

$P(n)$: After the nth iteration,

a.) $(a \in x_n + y_n \wedge b \in l_n) \implies a \geq b$.

b.) $\texttt{Counter}(x_n + y_n + l_n) = \texttt{Counter}(x_0 + y_0)$.

c.) $\texttt{len}(l_n) = n$.

d.) x_n, y_n, l_n are all sorted.

Initialization. We'll show $P(0)$:

1. is vacuously true since l_0 is empty
2. $\texttt{Counter}(x_0 + y_0 + l_0) = \texttt{Counter}(x_0 + y_0 + []) = \texttt{Counter}(x_0 + y_0)$
3. $\texttt{len}(l_n) = \texttt{len}([]) = 0$.
4. x_0, y_0 are sorted by the precondition, and l_0 is vacuously sorted.

Correctness of `merge`

$P(n)$: After the nth iteration,

a.) $(a \in x_n + y_n \wedge b \in l_n) \implies a \geq b$.

b.) $\texttt{Counter}(x_n + y_n + l_n) = \texttt{Counter}(x_0 + y_0)$.

c.) $\texttt{len}(l_n) = n$.

d.) x_n, y_n, l_n are all sorted.

Maintenance. Let $k \in \mathbb{N}$ be any natural number and suppose $P(k)$. We'll show that $P(k+1)$. There are several cases marked by the numbers in the comments. Let's start with case 1.

The variables are updated as follows.

- $l_{k+1} = l_k + [y_k[0]]$.
- $y_{k+1} = y_k[1:]$.
- $x_{k+1} = x_k$.

In this case, x and y are both non-empty and $y[0] \leq x_0$. We'll show $P(k+1)$

Correctness of `merge`

$P(n)$: After the nth iteration,

a.) $(a \in x_n + y_n \wedge b \in l_n) \implies a \geq b$.

b.) $\texttt{Counter}(x_n + y_n + l_n) = \texttt{Counter}(x_0 + y_0)$.

c.) $\texttt{len}(l_n) = n$.

d.) x_n, y_n, l_n are all sorted.

Maintenance. (a.) We need to show that $a \in x_{k+1} + y_{k+1} \wedge b \in l_{k+1} \implies a \geq b$. Since $y_k[0]$ is the only element that moved, by $P(k)$.a, it suffices to consider when $b = y_k[0]$. I.e., we need to show that $y_k[0]$ is minimal in $x_k + y_k$.

Since y_k and x_k are sorted by $P(k)$.d, $y_k[0]$ is minimal in y_k, since $y_k[0] \leq x_k[0]$, and x_k is sorted, $y_k[0]$ is also minimal in x_k.

Correctness of `merge`

$P(n)$: After the nth iteration,

a.) $(a \in x_n + y_n \wedge b \in l_n) \implies a \geq b$.

b.) $\texttt{Counter}(x_n + y_n + l_n) = \texttt{Counter}(x_0 + y_0)$.

c.) $\texttt{len}(l_n) = n$.

d.) x_n, y_n, l_n are all sorted.

Maintenance. (b.) We have

$$\begin{aligned}\texttt{Counter}(x_{k+1} + y_{k+1} + l_{k+1}) &= \texttt{Counter}(x_k + y_k[1:] + l_k + [y_k[0]]) \\ &= \texttt{Counter}(x_k + y_k + l_k) \\ &= \texttt{Counter}(x_0 + y_0).\end{aligned}$$

The second line holds because `Counter`is unordered, and the third line holds because of $P(k)$.b

Correctness of `merge`

$P(n)$: After the nth iteration,

a.) $(a \in x_n + y_n \wedge b \in l_n) \implies a \geq b$.

b.) $\texttt{Counter}(x_n + y_n + l_n) = \texttt{Counter}(x_0 + y_0)$.

c.) $\texttt{len}(l_n) = n$.

d.) x_n, y_n, l_n are all sorted.

Maintenance. **(c.)** $\texttt{len}(l_{k+1}) = \texttt{len}(l_k) + 1 = k + 1$

Correctness of `merge`

$P(n)$: After the nth iteration,

a.) $(a \in x_n + y_n \wedge b \in l_n) \implies a \geq b$.

b.) $\texttt{Counter}(x_n + y_n + l_n) = \texttt{Counter}(x_0 + y_0)$.

c.) $\texttt{len}(l_n) = n$.

d.) x_n, y_n, l_n are all sorted.

Maintenance. (d.) By $P(k)$.d we have x_k, y_k, and l_k are all sorted.

- $x_{k+1} = x_k$, and so is still sorted.
- y_{k+1} is a sublist of y_k and so is also sorted.
- $l_{k+1} = l_k + [y_k[0]]$ is sorted because $P(k)$.a implies that $\forall b \in l_k,\ y_k \geq b$.

Correctness of `merge`

$P(n)$: After the nth iteration,

a.) $(a \in x_n + y_n \wedge b \in l_n) \implies a \geq b$.

b.) $\texttt{Counter}(x_n + y_n + l_n) = \texttt{Counter}(x_0 + y_0)$.

c.) $\texttt{len}(l_n) = n$.

d.) x_n, y_n, l_n are all sorted.

The other cases are similar; I'll leave this to you.

Correctness of `merge`

$P(n)$: After the nth iteration,

a.) $(a \in x_n + y_n \wedge b \in l_n) \implies a \geq b$.

b.) $\texttt{Counter}(x_n + y_n + l_n) = \texttt{Counter}(x_0 + y_0)$.

c.) $\texttt{len}(l_n) = n$.

d.) x_n, y_n, l_n are all sorted.

Termination. Let $n = \texttt{len}(x_0 + y_0)$. I claim that the algorithm terminates after n iterations. By $P(n)$.c, we have $\texttt{len}(l_n) = \texttt{len}(x) + \texttt{len}(y)$, by $P(n)$.b, we have $\texttt{Counter}(x_n + y_n + l_n) = \texttt{Counter}(x_0 + y_0)$, since $\texttt{len}(l_n) = \texttt{len}(x_0 + y_0)$, x_n and y_n must be empty and thus the loop condition fails. Thus, the algorithm terminates.

Correctness of `merge`

$P(n)$: After the nth iteration,

a.) $(a \in x_n + y_n \wedge b \in l_n) \implies a \geq b$.

b.) $\texttt{Counter}(x_n + y_n + l_n) = \texttt{Counter}(x_0 + y_0)$.

c.) $\texttt{len}(l_n) = n$.

d.) x_n, y_n, l_n are all sorted.

Now let n be the last iteration that runs. Since the next iteration did not run, we have x_n and y_n are both empty. By $P(n).b$, we have $\texttt{Counter}(x_n + y_n + l_n) = \texttt{Counter}(l_n) = \texttt{Counter}(x_0 + y_0)$. Furthermore, by $P(n).d$, l_n is sorted as required.

Loop Invariants

- It's normal for the loop invariant to have many parts!
- If you're trying to prove a loop invariant and you get stuck and wish some other property holds, try adding what you need as part of the loop invariant.
- For example, it's common for part 4. of a loop invariant to imply part 1. of the loop invariant.

Summary - Correctness of Algorithms

- If the algorithm is recursive, prove correctness directly by induction.
- For algorithms with loops, prove the correctness of the loop by defining a Loop Invariant, proving the Loop Invariant, and showing that the Loop Invariant holding at the end of the algorithm implies the postcondition.

What are your questions?

CSC 236 Lecture 9: Formal Language Theory 1

Harry Sha

July 19, 2023

Today

- Formal languages
- Deterministic Finite Automata
- Designing DFAs
- Non-Deterministic Finite Automata (NFAs)
- Regular Languages

Formal languages

Deterministic Finite Automata

Designing DFAs

Non-Deterministic Finite Automata (NFAs)

Regular Languages

Reference

This lecture loosely follows chapter 1 of *Introduction to the Theory of Computation* by Michael Sipser.

The presentation there is a little more formal, but we'll use the same notation so it might be useful to check that out for additional examples.

Problems

In lecture 1 we used Cantor's Theorem to show that there are problems that can't be solved. In that lecture, we defined a problem for every set.

If A is a set of strings, there is the problem problem of deciding whether a given input is in A or not.

Examples:

$$A = \{w \in \mathrm{Strings} : w \text{ is a palindrome}\},$$

$$A = \{w : w \text{ is a C program with no syntax errors}\}.$$

Formal Languages

The problems that we consider are still going to be of this type!

This time, we'll make things a little more formal...

Definitions

- An alphabet, Σ is a non-empty, finite, set of symbols.

Definitions

- An alphabet, Σ is a non-empty, finite, set of symbols.
- A string w over an alphabet Σ is a finite (0 or more) sequence of symbols from Σ.

Definitions

- An alphabet, Σ is a non-empty, finite, set of symbols.
- A string w over an alphabet Σ is a finite (0 or more) sequence of symbols from Σ.
- The set of all strings is denoted Σ^*. (Before now, we've been calling it $\mathrm{Strings}$).

Definitions

- An alphabet, Σ is a non-empty, finite, set of symbols.
- A string w over an alphabet Σ is a finite (0 or more) sequence of symbols from Σ.
- The set of all strings is denoted Σ^*. (Before now, we've been calling it $\mathrm{Strings}$).
- A language is any subset of Σ^*.

Definitions

- An alphabet, Σ is a non-empty, finite, set of symbols.
- A string w over an alphabet Σ is a finite (0 or more) sequence of symbols from Σ.
- The set of all strings is denoted Σ^*. (Before now, we've been calling it $\mathrm{Strings}$).
- A language is any subset of Σ^*.
- Denote the empty string by ϵ and note that it is the unique string with length 0.

Definitions

- An alphabet, Σ is a non-empty, finite, set of symbols.
- A string w over an alphabet Σ is a finite (0 or more) sequence of symbols from Σ.
- The set of all strings is denoted Σ^*. (Before now, we've been calling it $\mathrm{Strings}$).
- A language is any subset of Σ^*.
- Denote the empty string by ϵ and note that it is the unique string with length 0.

More definitions

Let $x, y \in \Sigma^*$ be strings, $A, B \subseteq \Sigma^*$ be languages, and $n \in \mathbb{N}$ be a natural number

- Write xy to mean the concatenation of x and y.

More definitions

Let $x, y \in \Sigma^*$ be strings, $A, B \subseteq \Sigma^*$ be languages, and $n \in \mathbb{N}$ be a natural number

- Write xy to mean the concatenation of x and y.
- Write x^n to mean $\underbrace{xx\ldots x}_{n \text{ times}}$.

More definitions

Let $x, y \in \Sigma^*$ be strings, $A, B \subseteq \Sigma^*$ be languages, and $n \in \mathbb{N}$ be a natural number

- Write xy to mean the concatenation of x and y.
- Write x^n to mean $\underbrace{xx\ldots x}_{n \text{ times}}$.
- What is x^0?

More definitions

Let $x, y \in \Sigma^*$ be strings, $A, B \subseteq \Sigma^*$ be languages, and $n \in \mathbb{N}$ be a natural number

- Write xy to mean the concatenation of x and y.
- Write x^n to mean $\underbrace{xx\ldots x}_{n \text{ times}}$.
- What is x^0? ϵ

More definitions

Let $x, y \in \Sigma^*$ be strings, $A, B \subseteq \Sigma^*$ be languages, and $n \in \mathbb{N}$ be a natural number

- Write xy to mean the concatenation of x and y.
- Write x^n to mean $\underbrace{xx\ldots x}_{n \text{ times}}$.
- What is x^0? ϵ
- Let $AB = \{ab : a \in A, b \in B\}$.

More definitions

Let $x, y \in \Sigma^*$ be strings, $A, B \subseteq \Sigma^*$ be languages, and $n \in \mathbb{N}$ be a natural number

- Write xy to mean the concatenation of x and y.
- Write x^n to mean $\underbrace{xx...x}_{n \text{ times}}$.
- What is x^0? ϵ
- Let $AB = \{ab : a \in A, b \in B\}$.
- Let $A^n = \underbrace{AA...A}_{n \text{ times}}$.

More definitions

Let $x, y \in \Sigma^*$ be strings, $A, B \subseteq \Sigma^*$ be languages, and $n \in \mathbb{N}$ be a natural number

- Write xy to mean the concatenation of x and y.
- Write x^n to mean $\underbrace{xx...x}_{n \text{ times}}$.
- What is x^0? ϵ
- Let $AB = \{ab : a \in A, b \in B\}$.
- Let $A^n = \underbrace{AA...A}_{n \text{ times}}$.
- What is A^0?

More definitions

Let $x, y \in \Sigma^*$ be strings, $A, B \subseteq \Sigma^*$ be languages, and $n \in \mathbb{N}$ be a natural number

- Write xy to mean the concatenation of x and y.
- Write x^n to mean $\underbrace{xx...x}_{n \text{ times}}$.
- What is x^0? ϵ
- Let $AB = \{ab : a \in A, b \in B\}$.
- Let $A^n = \underbrace{AA...A}_{n \text{ times}}$.
- What is A^0? $\{\epsilon\}$.

More definitions

Let $x, y \in \Sigma^*$ be strings, $A, B \subseteq \Sigma^*$ be languages, and $n \in \mathbb{N}$ be a natural number

- Write xy to mean the concatenation of x and y.
- Write x^n to mean $\underbrace{xx...x}_{n \text{ times}}$.
- What is x^0? ϵ
- Let $AB = \{ab : a \in A, b \in B\}$.
- Let $A^n = \underbrace{AA...A}_{n \text{ times}}$.
- What is A^0? $\{\epsilon\}$.
- Let $A^* = A^0 \cup A^1 \cup A^2 \cup A^3 \cup ... = \bigcup_{i \in \mathbb{N}} A^i$

Generated by...

A while ago we studied what it meant for a set to be generated from B by the functions in F.

Question. How would you generate Σ^*?

Generated by...

A while ago we studied what it meant for a set to be generated from B by the functions in F.

Question. How would you generate Σ^*?

- $B = \Sigma \cup \{\epsilon\}$
- $F = \{\texttt{concat}\}$

Where $\texttt{concat}(x, y) = xy$

Example - English

- $\Sigma = \{a, b, c, ..., z\}$
- $\Sigma^* = \{\epsilon, a, aa, ab, ac, ..., ba, ..., aaa, ...\}$
- $\text{English} \subseteq \Sigma^*$

Example - Even

- $\Sigma = \{0, 1\}$
- $\Sigma^* = \{\epsilon, 0, 1, 00, 01, ...\}$
- $\mathrm{Even} = \{w \in \Sigma^* : w \text{ has an even number of 1s}\}$

Alphabet

Any finite set works for the alphabet! However, to make things simple, we'll usually take the alphabet to be $\Sigma = \{0, 1\}$, or $\{a, b\}$.

The Problem (again)

Given a language A over an alphabet Σ, come up with a program that decides whether a given string $x \in \Sigma^*$ is in A or not.

The Problem (again)

Given a language A over an alphabet Σ, come up with a program that decides whether a given string $x \in \Sigma^*$ is in A or not.

Formal languages

Deterministic Finite Automata

Designing DFAs

Non-Deterministic Finite Automata (NFAs)

Regular Languages

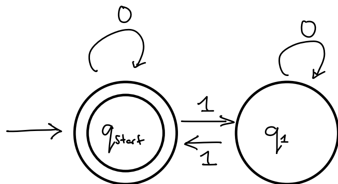
What is a program?

To talk formally about computation it's important to specify a model of computation.

- What does a program look like?
- What can the program do?

DFAs

DFAs are one model of computation. They look like this.



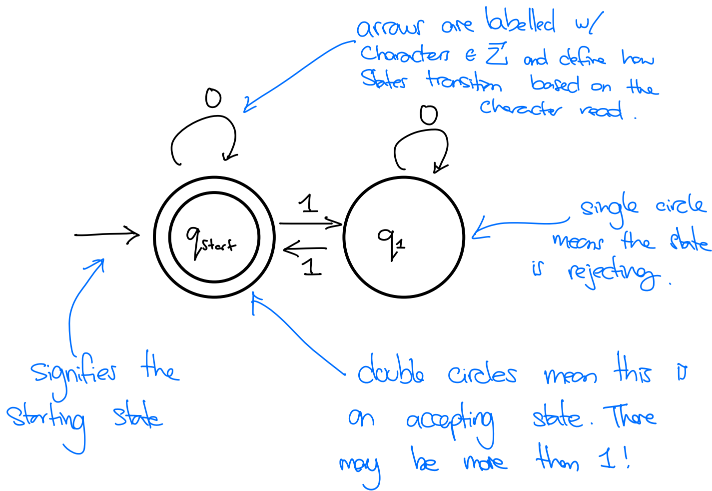
DFAs

A single DFA corresponds to what we think of as a program.

Computation of a DFA

Input: a string $w \in \Sigma^*$.
Output: accept/reject.

- The DFA starts at a predefined start state.
- The DFA reads in the input string one character at a time. Depending on the character read and the current state, the DFA deterministically moves to a new state.
- When it has read the entire string, the DFA will be in some state. If that state is one of the accept states, the DFA accepts. Otherwise, the DFA rejects.

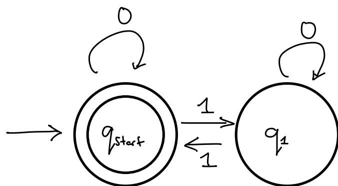


arrows are labelled w/ characters ∈ Σ and define how states transition based on the character read.
0
0
1
1
q_{start}
q_1
single circle means the state is rejecting.
signifies the starting state
double circles mean this is an accepting state. There may be more than 1!

Language of a DFA

Let M be a DFA, the language of a DFA, denoted $L(M)$, is the set of strings $w \in \Sigma^*$ such that M accepts w.

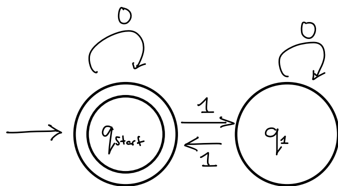
Example



Does this DFA accept

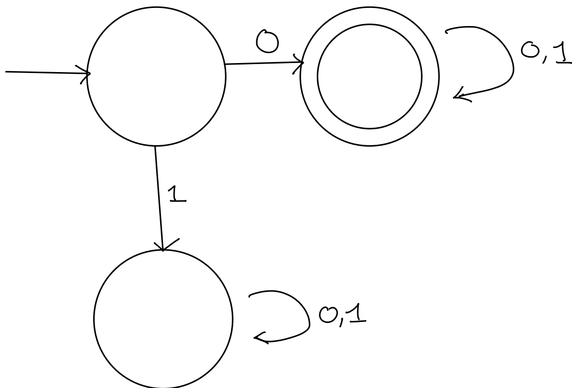
- 001101?
- 100110?
- 111100?
- 011000?
- ϵ?

Example



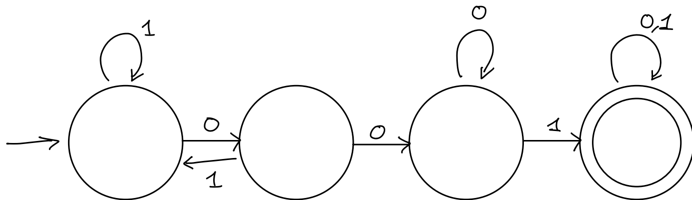
What is the language of the DFA?

Example



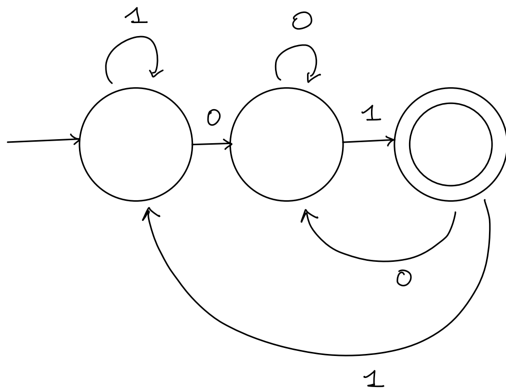
What is the language of the DFA?

Example



How about this one?

Example



How about this one?

DFAs

- Deterministic. Given the current state and character read, the next state is always the same.
- Finite. There are a finite number of states.

States are analogous to "memory", since we can store information about the input by transitioning to different states.

Formal languages

Deterministic Finite Automata

Designing DFAs

Non-Deterministic Finite Automata (NFAs)

Regular Languages

Defining a DFA

When defining a DFA, you need to do the following

- Tell me the alphabet, Σ of the DFA.
- Define a **finite** set of states, Q.
- Tell me a start state $q_{\text{start}} \in Q$.
- Tell me which states are accepting. Formally, find a subset $F \subseteq Q$ of accept states.
- For each state q and each character $x \in \Sigma$, tell me which state to go to next if I read x from state q.

In this class, **you may capture all of this information in a drawing.**

Tips

- States $\iff$ Memory. Use states to capture information about the input read so far! What do you need to remember about the input in order to answer the question?
- It takes some time getting used to the fact that we read the input left to right, and only get to read each character once! Keep this in mind when designing DFAs!
- Common pattern: The garbage state.

Example

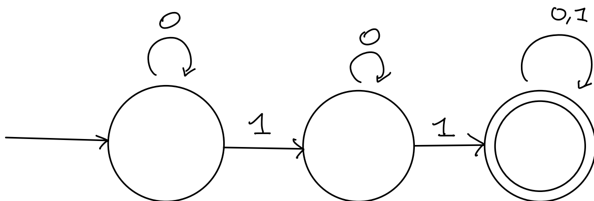
Design a DFA M such that

$$L(M) = \{w \in \{0,1\}^* : w \text{ has at least two 1s}\}$$

Example

Design a DFA M such that

$$L(M) = \{w \in \{0,1\}^* : w \text{ has at least two 1s}\}$$



Example

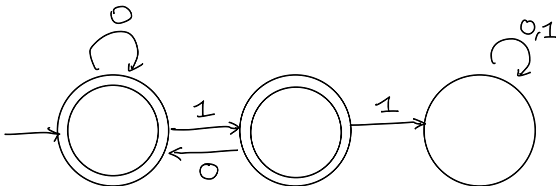
Design a DFA M such that

$$L(M) = \{w \in \{0,1\}^* : w \text{ does not contain the substring } 11\}$$

Example

Design a DFA M such that

$$L(M) = \{w \in \{0,1\}^* : w \text{ does not contain the substring } 11\}$$



Example

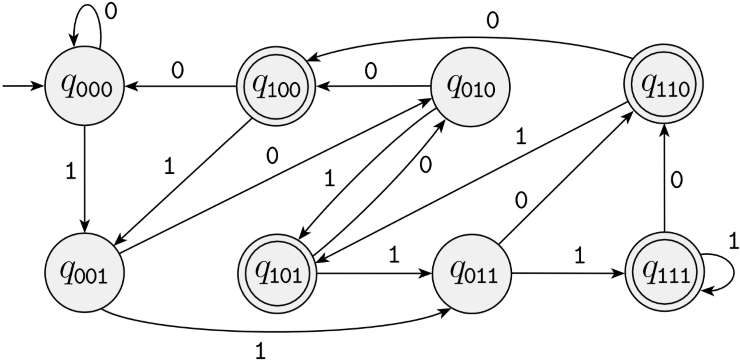
Design a DFA M such that

$$L(M) = \{w \in \{0,1\}^* : \text{ The third last character of } w \text{ is } 1\}$$

Example

Design a DFA M such that

$$L(M) = \{w \in \{0,1\}^* : \text{ The third last character of } w \text{ is } 1\}$$



Reference: Sipser

Formal languages

Deterministic Finite Automata

Designing DFAs

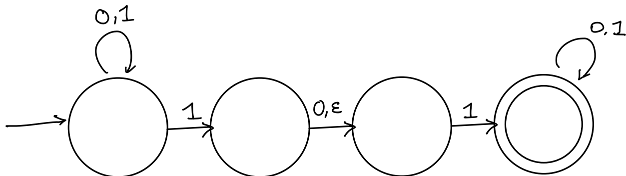
Non-Deterministic Finite Automata (NFAs)

Regular Languages

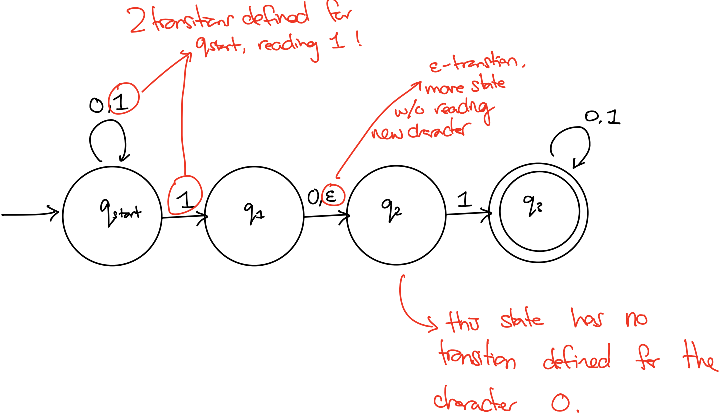
Another model of computation - Nondeterministic Finite Automata (NFAs)

Here's an example of an NFA

$\Sigma : \{0,1\}$



Key differences



Key differences

- For each state, there can be multiple arrows labelled with the SAME character!
- States do not need to have one arrow for each character in Σ.
- Arrows can be labelled with ϵ.

Computation of a NFA

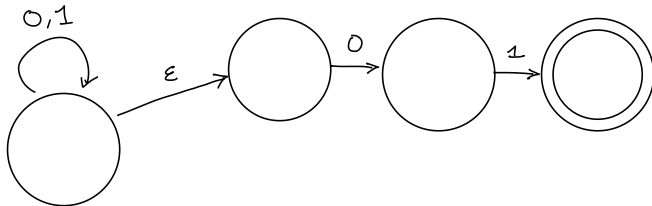
Input: a string $w \in \Sigma^*$.
Output: accept/reject.

- The NFA starts at a predefined start state.
- The NFA reads in the input string one character at a time. Let c be the character that was read. There are several cases depending on the current state.
  1. If the state has no arrows coming out of it labelled with c, halt execution and immediately reject.
  2. Otherwise, choose one of the arrows labelled with c and follow it and read the next character.
  3. If there is an arrow labelled with ϵ, you may choose to follow it. In this case, you don't read the input character (i.e. the next character to be read is still the same)
- When it has read the entire string, the NFA will be in some state. Depending on the choices made, the final state will either be accepting or rejecting. **The NFA accepts the string if ANY sequence of choices leads to an accept state.**

Language of a NFA

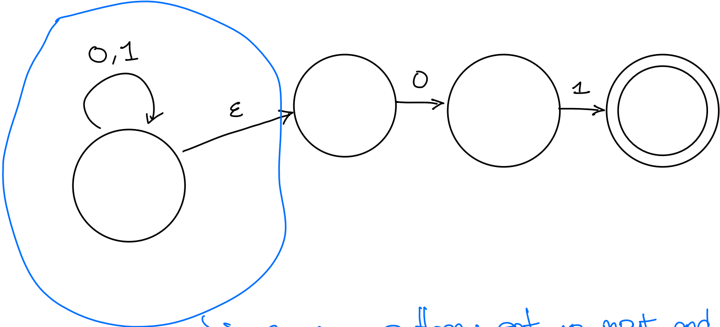
Let N be a NFA, the language of a NFA, denoted $L(N)$, is the set of strings $w \in \Sigma^*$ such that N accepts w.

What is the language of...



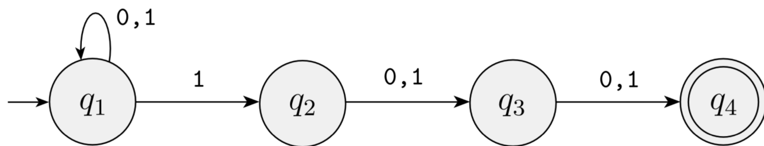
$\{w \in \Sigma^* : w \text{ ends in } 01\}$

Common Pattern



Common pattern: eat up input and "guess" where a certain substring starts!

NFA for third last character is a 1



Formal languages

Deterministic Finite Automata

Designing DFAs

Non-Deterministic Finite Automata (NFAs)

Regular Languages

Regular Languages

A language A is regular if and only if there is a DFA M such that $L(M) = A$.

Closure

Suppose A and B are regular languages, then

- $\overline{A}$,
- AB,
- $A \cup B$,
- $A \cap B$,
- A^n, and
- A^*

are also regular.

Let's prove this!

$\overline{A}$

Suppose A is regular. Let M be a DFA for A. Let M' be the DFA with all the states of M flipped. Then $L(M') = \overline{A}$.

$A \cup B$

Let M and N be DFAs for A and B respectively. We need to find a DFA D for $A \cup B$.

$A \cup B$

Let M and N be DFAs for A and B respectively. We need to find a DFA D for $A \cup B$.

Idea 1: Here's what we'd like to do. Let w be the input. D runs M on w to check if $w \in A$ and then D runs N on w to check if $w \in B$. Accept if either was accept and reject if both were rejected.

$A \cup B$

Let M and N be DFAs for A and B respectively. We need to find a DFA D for $A \cup B$.

Idea 1: Here's what we'd like to do. Let w be the input. D runs M on w to check if $w \in A$ and then D runs N on w to check if $w \in B$. Accept if either was accept and reject if both were rejected. What's the problem with this approach?

$A \cup B$

Let M and N be DFAs for A and B respectively. We need to find a DFA D for $A \cup B$.

Idea 1: Here's what we'd like to do. Let w be the input. D runs M on w to check if $w \in A$ and then D runs N on w to check if $w \in B$. Accept if either was accept and reject if both were rejected. What's the problem with this approach? we only get to read the input once!

$A \cup B$

Idea 2: Instead, we need to run the two machines in **parallel**. Thus, each state in D corresponds to a 2-tuple where the first element is the 'M' state and the second element is the 'N' state.

$A \cup B$

Idea 2: Instead, we need to run the two machines in **parallel**. Thus, each state in D corresponds to a 2-tuple where the first element is the 'M' state and the second element is the 'N' state.

If we're at state (x, y) and we read a character σ. Then we transition to (x', y') where x' is the state that x goes to (in M) when reading σ and y' is the state that y goes to (in N) when reading σ.

$A \cup B$

Idea 2: Instead, we need to run the two machines in **parallel**. Thus, each state in D corresponds to a 2-tuple where the first element is the 'M' state and the second element is the 'N' state.

If we're at state (x, y) and we read a character σ. Then we transition to (x', y') where x' is the state that x goes to (in M) when reading σ and y' is the state that y goes to (in N) when reading σ.

What should the accepting states be?

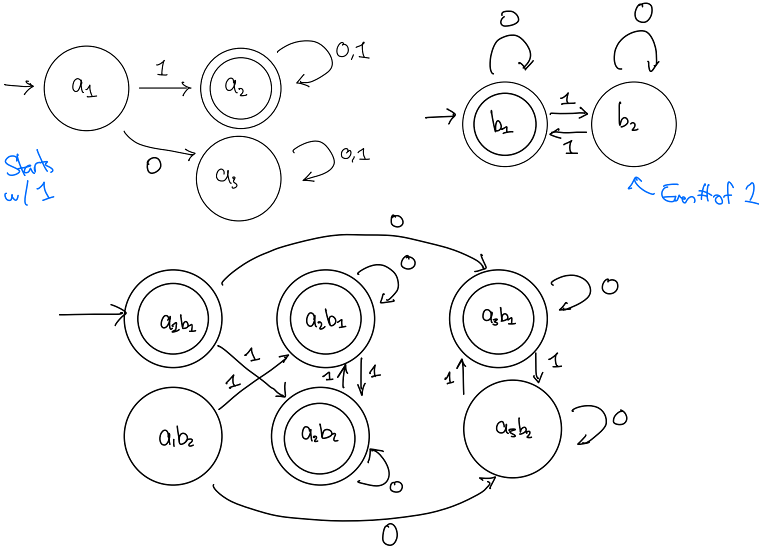
$A \cup B$

Idea 2: Instead, we need to run the two machines in **parallel**. Thus, each state in D corresponds to a 2-tuple where the first element is the 'M' state and the second element is the 'N' state.

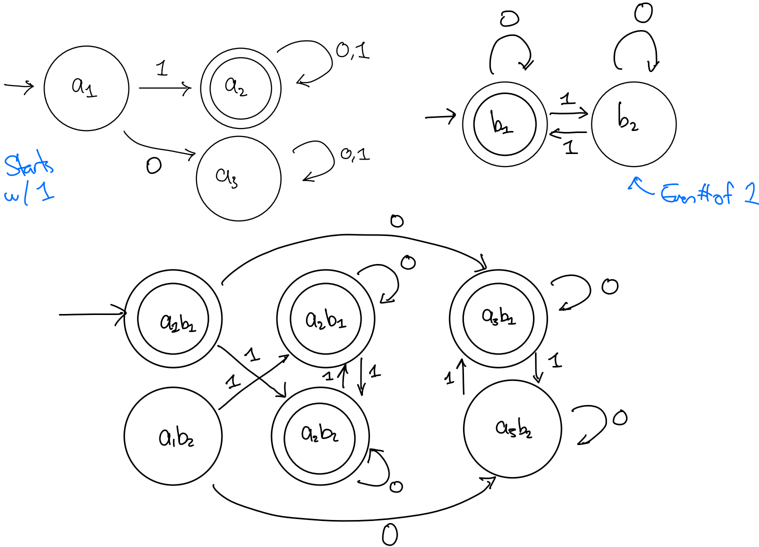
If we're at state (x, y) and we read a character σ. Then we transition to (x', y') where x' is the state that x goes to (in M) when reading σ and y' is the state that y goes to (in N) when reading σ.

What should the accepting states be? (x, y) should be accepting if either x was an accept state in M or y was an accept state in N (or both).

Example: Starts with 1 or has an even number of 1s



Example: Starts with 1 or has an even number of 1s



Note some states are never reached here (and can be removed)

$A \cap B$

Homework.

CSC 236 Lecture 10: Formal Language Theory 2

Harry Sha

July 26, 2023

Today

Review

Equivalence

Regex

Equivalence of NFAs and Regular Expressions (!)

Review

Equivalence

Regex

Equivalence of NFAs and Regular Expressions (!)

Review

- DFAs
- A language A is regular iff there is a DFA M such that $L(M) = A$

Closure

Suppose A and B are regular languages, then

- $\overline{A}$,
- $A \cup B$,
- $A \cap B$,
- AB ,
- A^n, and
- A^*,

are also regular.

Closure

Suppose A and B are regular languages, then

- $\overline{A}$ (by flipping states),
- $A \cup B$ (by running two DFAs in parallel),
- $A \cap B$ (hw),
- AB (today!),
- A^n(today!), and
- A^*(today!),

are also regular.

AB

The same trick of running the DFAs in parallel doesn't work.

AB

The same trick of running the DFAs in parallel doesn't work.
How would you write code to solve this?

AB

The same trick of running the DFAs in parallel doesn't work.
How would you write code to solve this?

On input w:

- For $i = 0, 1, ..., n$:
  - inA = $M(w[: i])$
  - inB = $M(w[i :])$
  - if inA and inB: accept
- reject

The idea is to try all the different ways to split up the string, if there is a way to split the string up such that the first half is in A and the second half is in B, then accept. If no way to split up the string works, reject.

Review: Nondeterminism

"If any sequence of **choices** leads to an accept state, accept"

"Reject only if every sequence of choices leads to reject"

Choices

- If there are multiple arrows marked with the read character, choose which arrow to take.
- If there is an ϵ-transition, choose whether or not to take it!

Review

Equivalence

Regex

Equivalence of NFAs and Regular Expressions (!)

Equivalence of DFAs and NFAs

Theorem

Let A be any language. There is a DFA M such that $A = L(M)$ if and only if there exists a NFA N such that $A = L(N)$.

Equivalence of DFAs and NFAs

Theorem

Let A be any language. There is a DFA M such that $A = L(M)$ if and only if there exists a NFA N such that $A = L(N)$.

The forward direction is true since every DFA is already a NFA (it just doesn't use the extra features). The backwards direction might be surprising to you!

Equivalence of DFAs and NFAs

Theorem

Let A be any language. There is a DFA M such that $A = L(M)$ if and only if there exists a NFA N such that $A = L(N)$.

The forward direction is true since every DFA is already a NFA (it just doesn't use the extra features). The backwards direction might be surprising to you!

Essentially it says all the extra features we give NFAs don't in fact give them more "power". *If I can solve it with an NFA, I can also solve it with a DFA*.

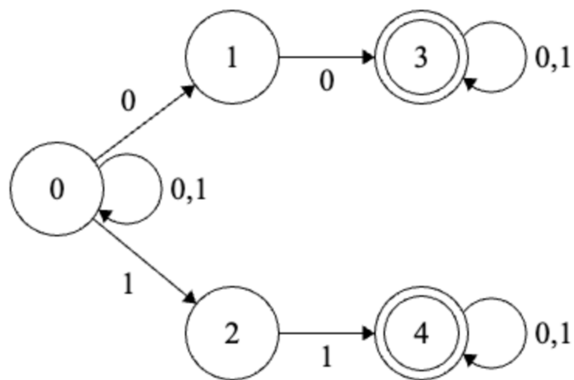
NFA $\implies$ DFA

We need to simulate a NFA N with a DFA M
High level idea:

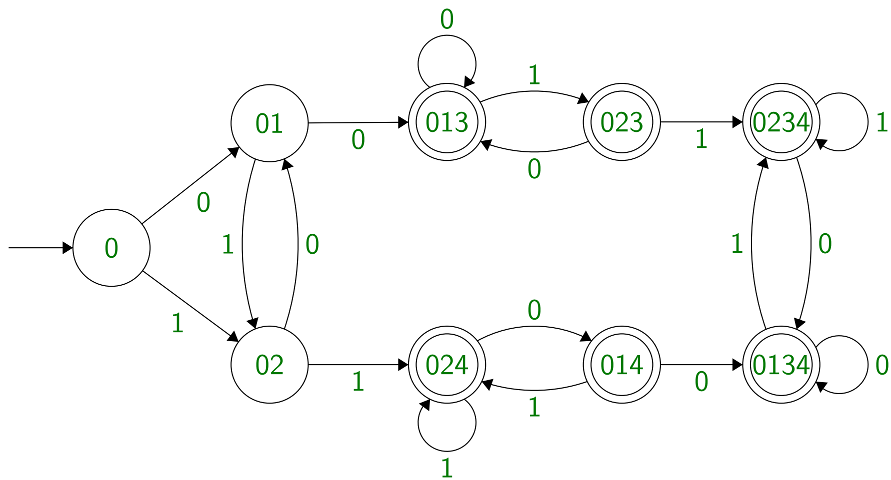
- Have a state in M for every subset of states in N.
- Let S, T be two states in M. Then S transitions to T reading σ if some choice allows me to go from a state in S to a state in T. The choice includes an unlimited number of ϵ transitions.
- The accepts states are the subsets that contain accept states.
- The start state is the subset containing the start state in N and all states reachable from the start state using an ϵ transition

Intuitively, if I'm at state S after having read w. S should contain all the possible states I could have been in N.

Example



Example



Takeaway

For every NFA N, there exists a (potentially huge) DFA M such that $L(N) = L(M)$.

Regular Languages (again)

The following are equivalent

- A is regular
- There is a DFA M such that $L(M) = A$
- There is a NFA N such that $L(N) = A$

Equivalent models of computation

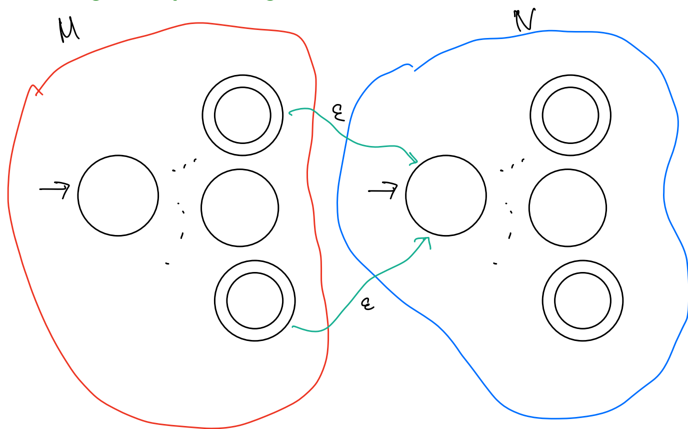
The more complex a model of computation, the easier it is to use. I.e. NFAs are more complex so finding NFAs for a language is easier than finding DFAs for language.

The more limited a model of computation, the easier it is to prove things about. Since DFAs are so restrictive, we can prove a lot of nice properties about them!

Since these two models of computation are equivalent, we can pick the right model for the right situation!

AB is regular

Let M and N be DFAs for A and B respectively. We'll show that AB is regular by finding an **NFA** H for AB.



AB

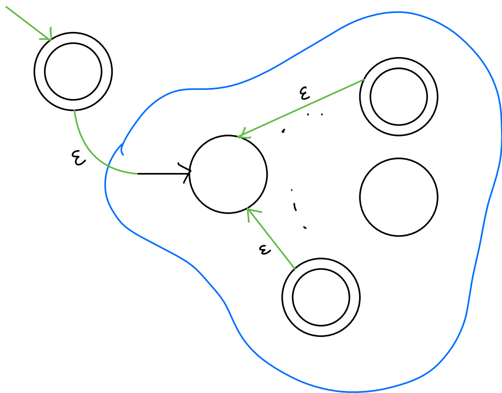
This NFA **guesses** when the string in B should start. It starts in M and can only move to N from accepts states in M since we need the first part of the string to be part of A.

A^n

Suppose A was regular, how would you show that $\forall n \in \mathbb{N}.A^n$ is regular?

By induction :)

A^*



Review

Equivalence

Regex

Equivalence of NFAs and Regular Expressions (!)

Regular Expressions

Regular expressions are all about matching patterns in strings.

Examples:

- Working with the terminal (live)
- valid email checking

Regular Expressions - Formally

Let Σ be an alphabet. Define the set of regular expressions $\mathcal{R}_\Sigma$ recursively as follows.

$\mathcal{R}_\Sigma$ is the smallest set such that

- $\emptyset \in \mathcal{R}_\Sigma$
- $\epsilon \in \mathcal{R}_\Sigma$
- $a \in \mathcal{R}_\Sigma$ for each $a \in \Sigma$
- $R \in \mathcal{R}_\Sigma \implies (R)^* \in \mathcal{R}_\Sigma$
- $R_1, R_2 \in \mathcal{R}_\Sigma \implies (R_1 R_2) \in \mathcal{R}_\Sigma$
- $R_1, R_2 \in \mathcal{R}_\Sigma \implies (R_1 | R_2) \in \mathcal{R}_\Sigma$

Note that $\mathcal{R}_\Sigma$ is defined inductively.

Operator Precedence

* comes before concatenation which comes before |.

Operator Precedence

* comes before concatenation which comes before $|$.

For example, $a^*b|bc$ means $((a^*)b)|(bc)$.

Language of a Regular Expression

The language if a regular expression R, denoted $L(R)$ is the set of strings that R matches. Formally,

- $L(\emptyset) = \emptyset$
- $L(\epsilon) = \{\epsilon\}$
- $L(a) = \{a\}$, for $a \in \Sigma$
- $L(R^*) = L(R)^*$ for $R \in \mathcal{R}_\Sigma^*$
- $L(R_1 R_2) = L(R_1)L(R_2)$ for $R_1, R_2 \in \mathcal{R}_\Sigma$
- $L(R_1|R_2) = L(R_1) \cup L(R_2)$ for $R_1, R_2 \in \mathcal{R}_\Sigma$

Examples

What are the languages of the following regular expressions?
(Assume the alphabet is $\{0, 1\}$)

- $((0|1)(0|1)(0|1))^*$

Examples

What are the languages of the following regular expressions?
(Assume the alphabet is $\{0, 1\}$)

- $((0|1)(0|1)(0|1))^*$
- $(1^*01^*0)^*$

Examples

What are the languages of the following regular expressions?
(Assume the alphabet is $\{0, 1\}$)

- $((0|1)(0|1)(0|1))^*$
- $(1^*01^*0)^*$
- $(0|1)^*1(0|1)(0|1)$

Examples

What are the languages of the following regular expressions?
(Assume the alphabet is $\{0, 1\}$)

- $((0|1)(0|1)(0|1))^*$
- $(1^*01^*0)^*$
- $(0|1)^*1(0|1)(0|1)$
- $(0|1)^*010(0|1)^*$

Examples

What are the languages of the following regular expressions?
(Assume the alphabet is $\{0, 1\}$)

- $((0|1)(0|1)(0|1))^*$
- $(1^*01^*0)^*$
- $(0|1)^*1(0|1)(0|1)$
- $(0|1)^*010(0|1)^*$
- $(0|\epsilon)1^*$

Examples

What are the languages of the following regular expressions?
(Assume the alphabet is $\{0, 1\}$)

- $((0|1)(0|1)(0|1))^*$
- $(1^*01^*0)^*$
- $(0|1)^*1(0|1)(0|1)$
- $(0|1)^*010(0|1)^*$
- $(0|\epsilon)1^*$

Examples

What are the languages of the following regular expressions? (Assume the alphabet is $\{0, 1\}$)

- $L(((0|1)(0|1)(0|1))^*) = \{w : |w| = 0 \mod 3\}$.
- $L((1^*01^*0)^*) = \{w :$ w ends with 0 contains an even number of 0s$\}$.
- $L((0|1)^*1(0|1)(0|1)) = \{w :$ third last character of w is 1$\}$.
- $L((0|1)^*010(0|1)^*) = \{w : w$ contains 010$\}$.
- $L((0|\epsilon)1^*) = \{w :$ w is zero or more 1s or 0 followed by zero or more 1s$\}$.

Shorthand

If R is a regex, and $m \in \mathbb{N}$, then

- R^m means m copies of R.
- $R^+ = RR^*$, i.e. at least one copy of R.
- $R^{m+} = R^m R^*$ means at least m copies of R.

If $S = \{s_1, s_2, ..., s_n\}$ is a finite set of strings, then S is shorthand for $s_1|s_2|...|s_n$. A common one is to use the alphabet, Σ.

Examples

Write regular expressions for the following languages

- Starts with 010.

Examples

Write regular expressions for the following languages

- Starts with 010.
- The second and the second last letter of w are the same.

Examples

Write regular expressions for the following languages

- Starts with 010.
- The second and the second last letter of w are the same.
- Contains 110.

Examples

Write regular expressions for the following languages

- Starts with 010.
- The second and the second last letter of w are the same.
- Contains 110.
- 1s always occur in pairs.

Examples

Write regular expressions for the following languages

- Starts with 010.
- The second and the second last letter of w are the same.
- Contains 110.
- 1s always occur in pairs.
- Doesn't contain more than four 1s in a row.

Examples

Write regular expressions for the following languages

- Starts with 010. $010\Sigma^*$
- The second and the second last letter are the same. $\Sigma 0 \Sigma^* 0 \Sigma | \Sigma 1 \Sigma^* 1 \Sigma$
- Contains 110. $\Sigma^* 110 \Sigma^*$
- 1s always occur in pairs. $\{0, 11\}^*$
- Doesn't contain more than four 1s in a row. $\{0, 01, 011, 0111\}^*$

Review

Equivalence

Regex

Equivalence of NFAs and Regular Expressions (!)

Equivalence of NFAs and Regex

For every regular expression R, there exists a NFA N such that $L(R) = L(N)$.

Regex $\to$ NFA

We want to show $\forall R \in \mathcal{R}_\Sigma$, there is an equivalent NFA. How do we do this?

Regex $\to$ NFA

We want to show $\forall R \in \mathcal{R}_\Sigma$, there is an equivalent NFA. How do we do this? By structural induction!

Regex $\rightarrow$ NFA

Base cases.

- $\emptyset$ has an equivalent NFA - one without an accept state!
- ϵ has an equivalent NFA - one with just an accept state!
- For each $a \in \Sigma$, a has an equivalent NFA - the following:

Regex $\to$ NFA

Inductive step. Suppose $R, S \in \mathcal{R}_\Sigma$, and have equivalent NFAs M, and N. We need to show that $R|S, RS, R^*$ all have equivalent NFAs.

- $R|S$. $L(R|S) = L(R) \cup L(S)$. By the inductive hypothesis, this is then equal to $L(M) \cup L(N)$. Since $L(M)$ and $L(N)$ are the languages of NFAs, they are regular. Since regular languages are closed under $\cup$ (from last lecture), $L(M) \cup L(N)$ is regular and hence has some NFA N'.
- RS. This follows from an identical argument as above, using the observation that regular languages are closed under concatenation.
- R^*. This follows from an identical argument as above, using the observation that regular languages are closed under *.

NFA $\to$ Regex

We will show this direction next time!

Regular Languages

The following are equivalent

- A is regular
- There is a DFA M such that $L(M) = A$
- There is a NFA N such that $L(N) = A$
- There is a regular expression R such that $L(R) = A$

Consequences

If I ask you to show me a language A is regular, you can choose to give me either a DFA, NFA or a regular expression!

How to choose

- I typically use regular expressions for languages that seem to require some form of 'matching'. For example *contains 121* as a substring, or *ends with 11*. Regular expressions are typically faster to find and write out in an exam setting.
- I'll use NFAs when I can't easily figure out a regular expression for something. These are usually languages for which memory seems to be useful like the $\mathrm{Dogwalk}$ example from hw.
- Stuff involving negations also seems easier to do with NFAs than with regular expressions. For example, *contains the substring* 011 is easy with regular expression, but *doesn't contain the substring* 011 is a bit more complicated.

CSC 236 Lecture 11: Formal Language Theory 3

Harry Sha

August 2, 2023

Today

Recap

NFA to Regex

Non-regular Languages

Myhill-Nerode Theorem
- Statement and Proof
- Applying the Theorem

Pumping Lemma
- Pumping Lemma vs. Myhill Nerode

Recap

NFA to Regex

Non-regular Languages

Myhill-Nerode Theorem
- Statement and Proof
- Applying the Theorem

Pumping Lemma
- Pumping Lemma vs. Myhill Nerode

Regular Languages

The following are equivalent

- A is regular
- There is a DFA M such that $L(M) = A$
- There is a NFA N such that $L(N) = A$
- There is a regular expression R such that $L(R) = A$

Closure

If A, B are regular, so are

- $\overline{A}$
- $A \cup B$
- $A \cap B$
- AB
- A^n
- A^*

Recap

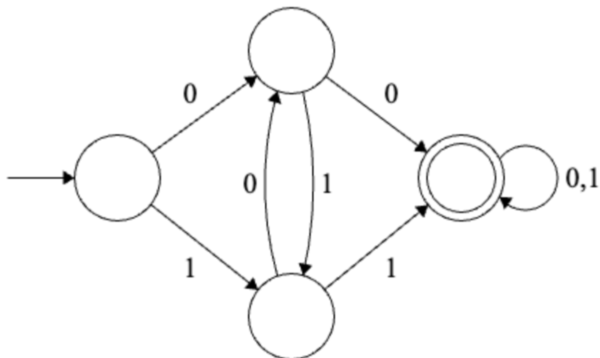
NFA to Regex

Non-regular Languages

Myhill-Nerode Theorem
- Statement and Proof
- Applying the Theorem

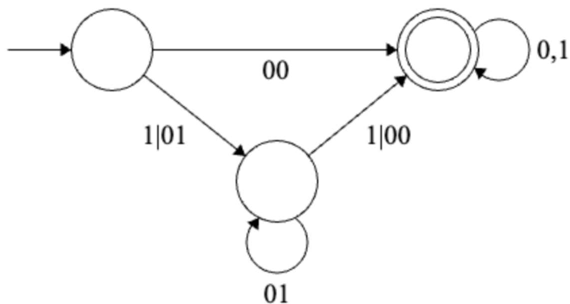
Pumping Lemma
- Pumping Lemma vs. Myhill Nerode

Example [8]



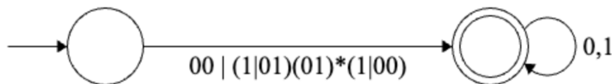
[8] Reference: CSC236 2022 Fall

Example [8]



[8] Reference: CSC236 2022 Fall

Example [8]

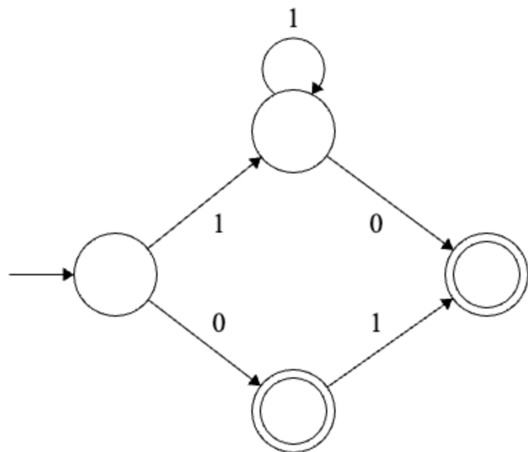


Final regular expression:

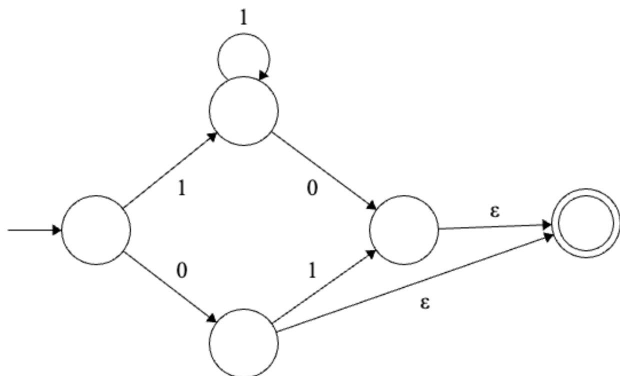
$$(00|(1|01)(01)^*(1|00))(0|1)^*$$

[8] Reference: CSC236 2022 Fall

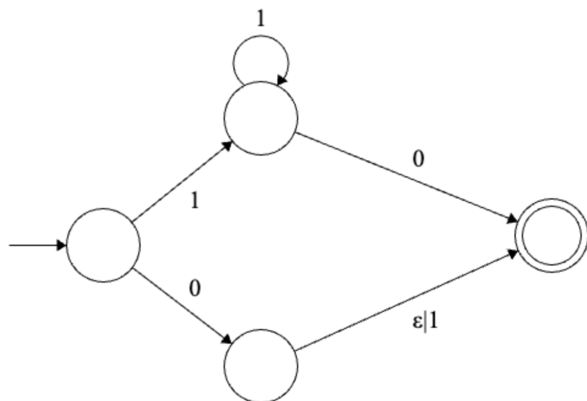
Example 2



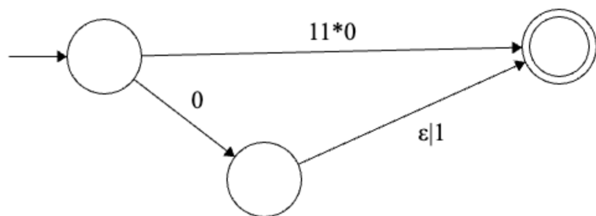
Example 2



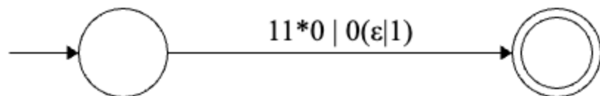
Example 2



Example 2



Example 2



Final regular expression: $1^+0|0(\epsilon|1)$

Sketch

- Alter the NFA so there's just one accepting state (using ϵ transitions).
- Iteratively rip out states, replacing transitions with regular expressions until you have something that looks like

R is the equivalent regular expression.

"Ripping" out states

For two states q_1, q_2 with a transition between them, let $f(q_1, q_2)$ be the regular expression labelling the transition.

Here are the steps to rip out a state q.

1. **Remove the loop**: If there is a self loop on state q, for each state s with a transition into q, update the transition $f(s, q) = f(s, q)f(q, q)^*$. For each state s' with a transition out of q, update the transition $f(q, s') = f(q, q)^* f(q, s')$
2. **Bypass** q: for each path (s, q, t) of length 2 through q, update $f(s, t) = f(s, t)|f(s, q)f(q, t)$. Note that it is possible that $s = t$, in which case this step adds a loop.
3. Remove q.

Recap

NFA to Regex

Non-regular Languages

Myhill-Nerode Theorem
- Statement and Proof
- Applying the Theorem

Pumping Lemma
- Pumping Lemma vs. Myhill Nerode

We showed a bunch of languages were regular...

However, from lecture 1, we know that there are some problems that computers can't solve...

... so what do non-regular languages look like?

What are some limitations for DFAs and NFAs?

Regular languages KEY intuition

D**F**As has a finite number of states.

Regular languages KEY intuition

D**F**As has a finite number of states.

States correspond to memory.

Thus, DFAs can compute languages that only need a finite amount of memory (and read the input once left to right).

In particular, a DFA has a fixed amount of memory, no matter how large the input is.

Example

Even is regular because no matter how large the input is, I only need to store one bit corresponding to whether or not the input has an even number of 1s so far.

Infinite Memory Required

What are some things you can't do with a fixed amount of memory?

Even simple things like storing the length of the input or the number of 1s - we don't know in advance how long our input string can be!

Example

Here's an example of a language that can't be computed using finite memory.

$$\{a^n b^n : n \in \mathbb{N}\}$$

Why?

Example

Here's an example of a language that can't be computed using finite memory.

$$\{a^n b^n : n \in \mathbb{N}\}$$

Why?

I don't know ahead of time how many *a*s there are, and I need to keep track of them to see how many *b*s I should expect.

Proving not regular

Intuitively,

$$X = \{a^n b^n : n \in \mathbb{N}\}$$

requires infinite memory so is not regular. However, this doesn't prove that it is not regular.

Proving not regular

Intuitively,

$$X = \{a^n b^n : n \in \mathbb{N}\}$$

requires infinite memory so is not regular. However, this doesn't prove that it is not regular.

To show X is not regular, we need to show **that there does not exist** a DFA M such that $L(M) = X$.

$X = \{a^n b^n : n \in \mathbb{N}\}$ is not regular

By contradiction, suppose there was a DFA M such that $L(M) = X$.

Claim: Suppose $m, n \in \mathbb{N}$ such that $m \neq n$, then M run on a^m and a^n end up in different states.

Proof of claim. Let q_m, q_n be the states reached after reading a^m and a^n respectively. By contradiction, suppose $q_m = q_n$. Suppose from this state, we then read b^m, let q' be the final state. Since $a^m b^m \in X$, q' should be accepting. However, since $a^n b^m \notin X$, q' should be rejecting, we have reached a contradiction since q' cannot be both.

$X = \{a^n b^n : n \in \mathbb{N}\}$ is not regular

By the claim, the DFA must reach a unique state for each $a, aa, aaa, \ldots$. Thus, M must have infinitely many states, which is a contradiction since M is supposed to be a DFA.

Key Insights

- Same state $\implies$ same fate. If two strings x, y led the DFA to the same state. **No matter what string** w **was read after**, either xw and yw both get accepted or yw both get rejected.

Key Insights

- Same state $\implies$ same fate. If two strings x, y led the DFA to the same state. **No matter what string** w **was read after**, either xw and yw both get accepted or yw both get rejected.
- The language $\{a^n b^n : n \in \mathbb{N}\}$ had infinitely many strings that do NOT share the same fate (and hence must have distinct states).

“Same state same fate” but more formal

Let A be any language and $x, y \in \Sigma^*$. Call x and y distinguishable relative to A if there exists w such that one of xw and yw are in A and the other is not. If x and y are not distinguishable, call them indistinguishable relative to A[9].

[9] If the language A is evident from the context, you can omit the “relative to A” part.

“Same state same fate” but more formal

Let A be any language and $x, y \in \Sigma^*$. Call x and y distinguishable relative to A if there exists w such that one of xw and yw are in A and the other is not. If x and y are not distinguishable, call them indistinguishable relative to A[9].

Lemma (Same state same fate)

Suppose M is a DFA such that $L(M) = A$, and let q_x and q_y be the states reached after reading x and y, respectively. If $q_x = q_y$, then x and y are indistinguishable relative to A.

[9] If the language A is evident from the context, you can omit the “relative to A” part.

Proof (informal)

Essentially, the DFA is deterministic, depending only on the current state and the character read.

Recap

NFA to Regex

Non-regular Languages

Myhill-Nerode Theorem
Statement and Proof
Applying the Theorem

Pumping Lemma
Pumping Lemma vs. Myhill Nerode

Myhill-Nerode Theorem (corollary)

Theorem

Let A be a language over Σ. Suppose there exists a set $S \subseteq \Sigma^*$ with the following properties

- *(Infinite). S is infinite*
- *(Pairwise distinguishable). $\forall x, y \in S$, with $x \neq y$. x, and y are distinguishable relative to A.*

Then A is not regular.

Proof

Let A be language, and suppose $S \subseteq \Sigma^*$ is infinite and pairwise distinguishable relative to A. WTS A is not regular.

By contradiction, suppose A was regular, then there exists some DFA M such that $L(M) = A$. Since M is a DFA, it has some finite set of states Q.

Let $g : S \to Q$, be a function mapping strings $x \in S$ to the state the DFA reaches after reading x from the start state.

Since S is infinite, and Q is finite, g is not injective. Therefore, there exist two strings $x, y \in S$ such that $g(x) = g(y)$. That is x and y reach the same state. Since x and y are in S, they are distinguishable; however, by the lemma, they are indistinguishable, which is a contradiction.

Using The Myhill Nerode Theorem

By The Myhill-Nerode Theorem, it suffices to find a set of strings, S, such that S is infinite and pairwise distinguishable relative to A.

Proof: $X = \{a^n b^n : n \in \mathbb{N}\}$ is not regular

Consider the set $S = \{a^n : n \in \mathbb{N}\}$ and note that S is infinite since it has one element for every natural number.

Proof: $X = \{a^n b^n : n \in \mathbb{N}\}$ is not regular

Consider the set $S = \{a^n : n \in \mathbb{N}\}$ and note that S is infinite since it has one element for every natural number.

Let $x, y \in S$ with $x \neq y$. Then $x = a^i$, $y = a^j$ for some $i \neq j$. We'll show that $x \not\sim y$, in particular, $w = b^i$ is such that $xw = a^i b^i \in X$, but $yw = a^j b^i \notin X$. Thus, S is an infinite set of pairwise distinguishable strings relative to X, so X is not regular.

Recap

NFA to Regex

Non-regular Languages

Myhill-Nerode Theorem
- Statement and Proof
- Applying the Theorem

Pumping Lemma
- Pumping Lemma vs. Myhill Nerode

An alternate proof

$X = \{a^n b^n : n \in \mathbb{N}\}$

By contradiction, suppose there was a DFA M such that $L(M) = X$.

Let k be the number of states of M. Let $(q_0, q_1, q_2, ..., q_k)$ be the sequence of states reached when running a^k on M. Since the sequence has length $k+1$, and there are only k states, by The Pigeonhole Principle, some state appears twice in the sequence. I.e. there exists $i, j \in \mathbb{N}$ with $i \neq j$, and $0 \leq i, j \leq k$ such that $q_i = q_j$. WLOG suppose $i < j$. Now here's what happens when we read $a^k b^k$.

1. After reading a^i we get to q_i.
2. Continuing, reading a^{j-i}, we get to $q_i = q_j$ again.
3. Continuing, reading a^{k-j} takes us to q_k.
4. We then read b^k and end up in some accept state since $a^k b^k \in X$.

An alternate proof

$X = \{a^n b^n : n \in \mathbb{N}\}$

Note that step 2 takes us in a loop! In particular, we started at q_i, read a^{j-i}, and then ended up back where we started!
What happens if we read $a^{j-i}a^{j-i}$ at q_i? We STILL end up at q_i.
Continuing to step three, we reach q_k again, this time after reading $a^i a^{j-i} a^{j-i} a^{k-j} = a^{k+j-i}$. Reading b^k, we still reach the same accepting state as before. However, this is a contradiction since $a^{k+j-i}b^k \notin X$ since $k + j - i \neq k$!

The Pumping Lemma

Suppose A is a regular language. Then there exists $k \in \mathbb{N}$ such that for all $w \in A$ with $|w| \geq k$, we can write w as xyz such that

1. $|xy| \leq k$
2. $|y| > 0$
3. For $i \in \mathbb{N}$, $xy^iz \in A$.

The Pumping Lemma explained

Suppose A is a regular language. Then there exists $k \in \mathbb{N}$ such that for all $w \in A$ with $|w| \geq k$, we can write w as xyz such that

1. $|xy| \leq k$
2. $|y| > 0$
3. For $i \in \mathbb{N}$, $xy^i z \in A$.

- Think of k as the number of states in the DFA.
- $w = xyz$. x is the part of the string that takes us to the start of the loop. y is the string that takes us in a loop. z is the remainder of the string.
- The three conditions mean the following.
  1. The loop occurs within the first k steps.
  2. The length of the looping string is non-zero. I.e., it's actually a loop.
  3. You can take the loop as many times as you like.

Using The Pumping Lemma to prove a language is not regular

Template:

By contradiction, suppose A is regular. Then, by the pumping lemma, there exists a pumping length $k \in \mathbb{N}$.

[find a string $w \in A$ with $|w| \geq k$.]
Thus, we can write $w = xyz$ satisfying the conditions of the pumping lemma.

[use conditions 1, 2 to argue something about what y looks like]

[use condition 3 to find another string in A of the form xy^iz for some $i \in \mathbb{N}$ which should actually NOT be in A.]

Example

$X = \{a^n b^n : n \in \mathbb{N}\}$

By contradiction, suppose X is regular. Then there is some pumping length k.

Note that $w = a^k b^k \in X$, and $|w| = 2k \geq k$. Thus, we can write $w = xyz$ satisfying the conditions of The Pumping Lemma.

Since $|xy| \leq k$, and the first k letters in w are all k, we know that $y = a^i$ for some $i \in \mathbb{N}$. By condition 2, $i > 0$.

Then by the third condition of The Pumping Lemma, we have $xy^0z = xz \in X$, which is a contradiction since $xz = a^{k-i}b^k$, which should not be in X.

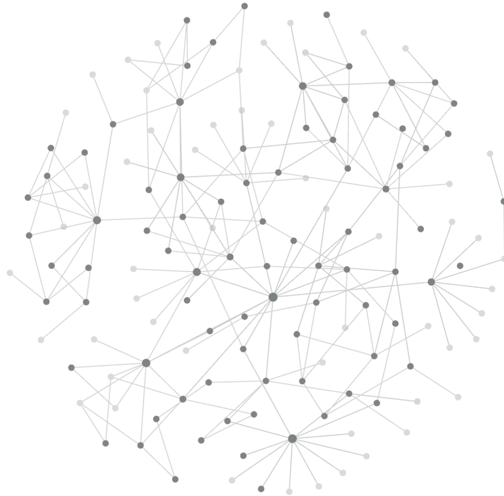
Pumping Lemma vs. Myhill Nerode for showing a language is not regular.

I prefer The Myhill Nerode Theorem - I find the arguments easier and harder to mess up.

The Pumping Lemma is an excellent backup to know.

You should try all the problems that require you to show a language is not regular using both methods and see if you develop a preference!

What do you think this is a graph of?



CSC 236 Lecture 12

Harry Sha

August 9, 2023

Today

Exam Tips

Recap - a look back

More in theory - a look ahead

- Error Correcting Codes
- Cryptography
- Complexity Theory

Exam Tips

Recap - a look back

More in theory - a look ahead

- Error Correcting Codes
- Cryptography
- Complexity Theory

Question Types

Similar to stuff you have seen in homework, lecture and tutorial. Here are the main question types that we have seen in the course and my recommendations for how to approach them in an exam and how to study for them.

Proofs about Functions

Write out the formal first order logic definitions for what you're trying to prove. Let that guide your solution.

Study tips

- Remember the intuition for injective/surjective/bijective. This might help you come up with a solution.
- Remember the formal definitions for injective/surjective/bijective. This will help you write the proof.

Modeling with Graphs

Writing it up

- Explicitly define your graph $G = (V, E)$.
- Tell me exactly what the vertex set V is.
- Tell me exactly what the edge set E is.
- State the corresponding graph problem.
- Explain why a solution to the graph problem is a solution to the problem in question and vice versa.

Study tips

- Review the graph problem we have studied, and all the practice problems.

Proofs by induction

Recognizing when to do something by induction.

- When the problem is something like $\forall n \in \mathbb{N}.P(n)$.
- When I give you an inductively/recursively defined set - that's almost always structural induction.

Writing it up

- When proving a statement, translate it into $\forall n \in \mathbb{N}.P(n)$ (or for structural induction $\forall x \in X.P(x)$, were X is an inductively defined set. This step will clarify your proof process.
- Clearly label the base case and inductive step.
- For the inductive step:
  - Clearly state your inductive hypothesis.
  - State what you're trying to prove, i.e. $P(k+1)$.
  - Clearly state where you are applying the inductive hypothesis.

Study tips

- Lots of practice.

Solving Recurrences

Deciding on a method

- Can it be done using the Master Method? If so, use it!
- Otherwise, draw the recursion tree to form a guess, and then use the substitution method.

Master Theorem

- State clearly that the master theorem applies in this case. "This is a standard form recurrence with parameters $a = \ldots$, $b = \ldots$, $f(n) = \ldots$"
- Calculate the leaf work, $n^{\log_b(a)}$, and determine the case split.
- Find an explicit ϵ like 0.0000001 in the leaf/root heavy case.
- In the root heavy case don't forget the regularity condition!

Study tips

- Come up with some of your own recurrences and solve them (or test each other).

Substitution Method

- Draw recursion tree for guess.
- Write out your guess $f(n)$.
- Write out the recurrence $T(n) = T(...) + ...$
- Substitute all the $T(...)$s on the RHS with $c \cdot f(...)$s, and replace the $=$ with $\leq$ for Big O or $\geq$ for Big Ω
- You want the RHS to then be $\leq cf(n)$.
- Rearrange to find an appropriate value of c.

Recursive Algorithm Correctness

Correctness is precondition implies postcondition.

- Do complete induction on the size of the input.
- The IH is that precondition implies postcondition for smaller instances.
- To apply the IH, you need to prove the precondition holds for the recursive step, AND that the recursive step is indeed a **smaller** instance (and thus captured in the IH)

Iterative Algorithm Correctness

- Do NOT try to do induction on the size of the input.
- Define a loop invariant.
  - Trace the algorithm on some example inputs to get an idea of what each variable corresponds to.
  - Use that intuition to come up with a loop invariant.
  - It's almost always useful to include: after the kth iteration $i_k = ...$ where i is the iteration variable.
- Sketch initialization, maintenance, and termination BEFORE writing it up. If everything seems to work out ok, you can start writing it up. Otherwise, you found something that didn't work, so you need to update your LI.
- Explicitly state your loop invariant: "$P(n)$: After the nth iteration ..."
- Prove initialization/maintenance/termination
- Termination: Either use the loop invariant or the descending sequence strategy, and prove that the LI after the last iteration implies the postcondition.

Determining if a Language is Regular

- Intuition: regular $\iff$ can be computed using fixed, finite memory.

Showing a langauge is regular

	Pros	Cons
Regex	Fast, great for matching substrings/beginnings/endings	Easy to match too many things
NFA	Powerful, great for counting # of occurrences mod N	Slower than regex and can harder to check than DFAs
DFA	Safe. Once you write a DFA it is easier to check (don't need to follow different sequence of choices)	Takes longer to write.

Showing a langauge is regular

Study tips

- For all the language I have given you in lectures, tutorials, and homeworks, come up with DFAs/NFAs and regular expressions for them.
- Get a lot of practice. This is particularly important for these kinds of problems since it increases the likelihood of you being able to relate a language on the exam to something you've seen before.

Closure

Questions of the form, suppose A, B are regular, then so is (some transformation of A, B)

- Assume you have **DFA**s for A, and B.
- Define a **NFA** based on the DFAs for A and B.[10]

[10] Why start with DFAs and then define an NFA?

- DFAs are simpler so easier to control (don't have to worry about the non-determinism with the machines for A and B)
- NFAs give you more power to show the new language is regular.

Showing a langauge is not regular

To show A is not regular,

- Find $S \subseteq \Sigma^*$ such that S is infinite and pairwise distinguishable relative to A. Remember to cite that this is enough b/c of the Myhill-Nerode Theorem.[11]
- OR use the Pumping Lemma

As I have said previously, I think the Myhill-Nerode Theorem is easier to use.

[11] Note S can be any set of strings, a common misconception is that S must be a subset of A.

Exam Tips

Recap - a look back

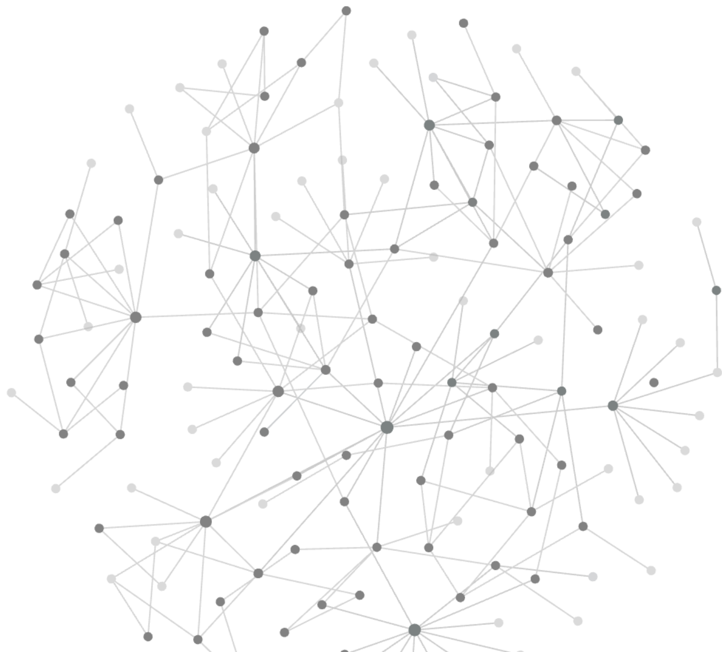
More in theory - a look ahead

- Error Correcting Codes
- Cryptography
- Complexity Theory

Takeaways

- A set of mathematical tools to analyze and model the world.
- Correctness and runtime of algorithms.
- A formal model of computation.

What was that graph from the first slide?



What was that graph from the first slide?

- $G_{236} = (V, E)$
- V is the set of things we studied.
- $\{u, v\} \in E$ if and only if u and v are "directly related".

That was a LOT of stuff - congratulations

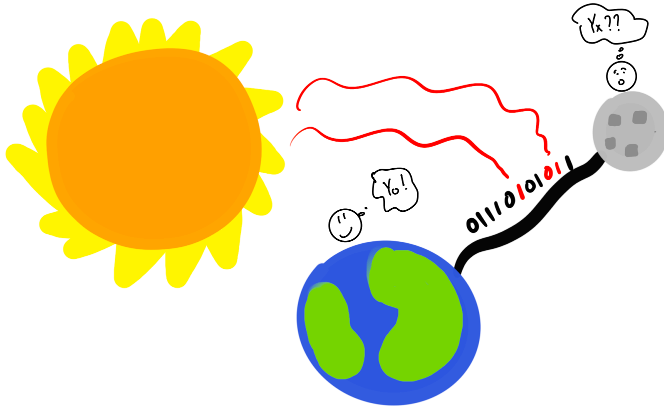
Exam Tips

Recap - a look back

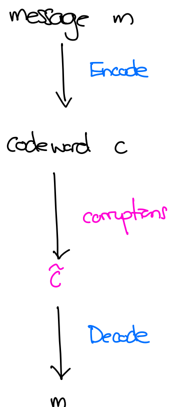
More in theory - a look ahead

- Error Correcting Codes
- Cryptography
- Complexity Theory

A noisy channel



Error Correcting Codes



What do we want from error correcting codes?

- We want to be able to decode from many errors.
- There should be minimal overhead. I.e. c shouldn't be much longer than m itself.
- $\mathrm{Enc}, \mathrm{Dec}$ should be fast.

Other applications

- QR Codes
- Covid Testing
- Hard drives

An Example

$$m = \texttt{hello}$$

$$c = \mathrm{Enc}(m) = \texttt{hellohellohello}$$

$$\tilde{c} = \texttt{jellohillahelgo}$$

An Example

$$m = \texttt{hello}$$

$$c = \mathrm{Enc}(m) = \texttt{hellohellohello}$$

$$\tilde{c} = \texttt{jellohillahelgo}$$

How do you correct the errors?

An Example

$$m = \texttt{hello}$$

$$c = \mathrm{Enc}(m) = \texttt{hellohellohello}$$

$$\tilde{c} = \texttt{jellohillahelgo}$$

When will this strategy work/fail? It works for any single corruption. If there are two corruptions that occur at the same index in multiple copies (e.g. two of the three hs are corrupted to a), the decoded message will be wrong!

An Example

$$m = \texttt{hello}$$

$$c = \mathrm{Enc}(m) = \texttt{hellohellohello}$$

$$\tilde{c} = \texttt{jellohillahelgo}$$

How many errors can I guarantee to work on if I sent 5 copies instead of just 3? How about $2a + 1$ in general for $a \in \mathbb{N}$?

An Example

$$m = \texttt{hello}$$

$$c = \mathrm{Enc}(m) = \texttt{hellohellohello}$$

$$\tilde{c} = \texttt{jellohillahelgo}$$

Performance summary

- Size blow up: 3 $(2a + 1)$
- Errors decodable from: 1 (a)

A size blow up of 3 to correct just a single error doesn't seem that worth it! Can we do better?

A Slightly Better Example

$$m \in \{0,1\}^4$$

$$c = m_1 m_2 m_3 m_4 (m_2 \oplus m_3 \oplus m_4)(m_1 \oplus m_3 \oplus m_4)(m_1 \oplus m_2 \oplus m_4)$$

A Slightly Better Example

$$m \in \{0,1\}^4$$

$$c = m_1 m_2 m_3 m_4 (m_2 \oplus m_3 \oplus m_4)(m_1 \oplus m_3 \oplus m_4)(m_1 \oplus m_2 \oplus m_4)$$

E.g.
$m = 1001$
$c = 1001100$

Claim: We can correct from one error!

A Slightly Better Example

$$m \in \{0,1\}^4$$

$$c = m_1 m_2 m_3 m_4 (m_2 \oplus m_3 \oplus m_4)(m_1 \oplus m_3 \oplus m_4)(m_1 \oplus m_2 \oplus m_4)$$

What is the message if this was the received codeword (with one error)

$$\tilde{c} = 1110011$$

A Slightly Better Example

$$m \in \{0,1\}^4$$

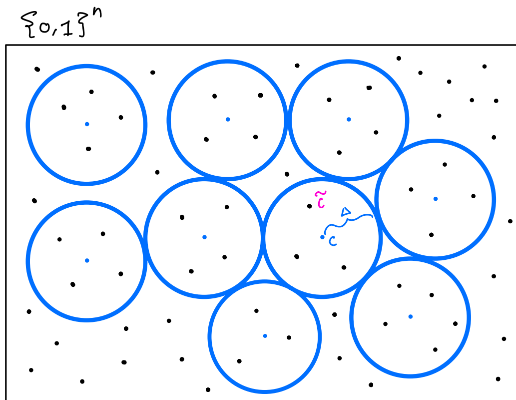
$$c = m_1 m_2 m_3 m_4 (m_2 \oplus m_3 \oplus m_4)(m_1 \oplus m_3 \oplus m_4)(m_1 \oplus m_2 \oplus m_4)$$

Performance summary

- Size blow up: $7/4$
- Errors decodable from: 1

We get the same error correction performance, but the blow up in the size of the message is only $7/4$ instead of 3!

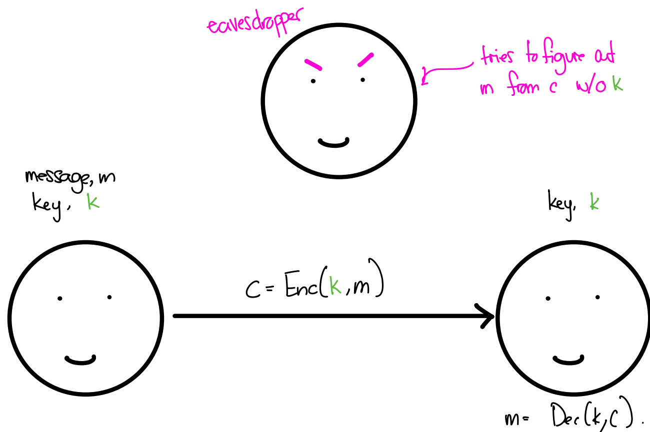
A Mathematical Model



Interesting Facts and Questions

- What is the optimal tradeoff between the size blow up and the number of errors you can correct from?
- How can we construct codes that get a good tradeoff that also have good encoding and decoding algorithms?
- A "random" code is pretty good. You can get a blow up factor $1/\epsilon^2$ and correct $1/4 - \epsilon/2$ errors. For example, $\epsilon = 0.1$, then we can blow up the message by a factor of 100 to correct a message that is corrupted in $1/5$ of all of its bits!

Secure Messaging



Shift Cipher

```python
CHAR2INT = {"a":0,"b":1,...,"z":25}
INT2CHAR = {0:"a",1:"b",...,25:"z"}

def shift_encode(message, key):
    new_message = ""
    for char in message:
        new_char = INT2CHAR[CHAR2INT[char] + key]
        new_message += new_char
    return new_message
```

$$m = \texttt{hello}, k = 3$$

$$c = \texttt{jhoor}$$

Shift Cipher is not great

Suppose some eavesdropper who doesn't know the key saw the ciphertext `jhoor`. What can they learn about the message?

Shift Cipher is not great

Suppose some eavesdropper who doesn't know the key saw the ciphertext `jhoor`. What can they learn about the message?

They actually get to learn a lot about the message! For example, they know that the third and fourth character must have been the same in the message and can rule out messages like `peach`. In fact, of all the 26^5 possible sequences of letters that could have been the message, they can narrow it down to just 26 possible messages.

Shift Cipher is not great

Suppose some eavesdropper who doesn't know the key saw the ciphertext `jhoor`. What can they learn about the message?

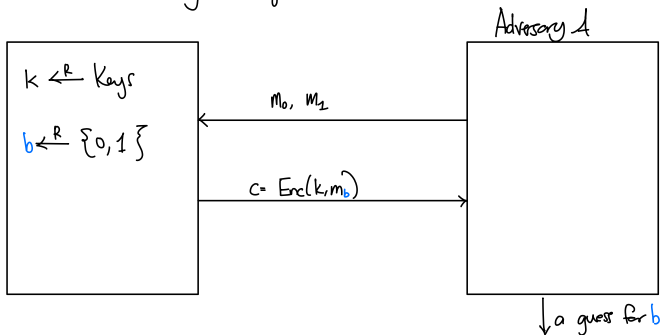
They actually get to learn a lot about the message! For example, they know that the third and fourth character must have been the same in the message and can rule out messages like `peach`. In fact, of all the 26^5 possible sequences of letters that could have been the message, they can narrow it down to just 26 possible messages.

For security, we want the eavesdropper to learn **nothing** about the real message!

Security

Here's how we can mathematically define "learns nothing".

m_0, m_1 are any messages.



Security: No eff. adversary win the game w/ any significant advantage.

The assumptions - some problems are hard

Since we don't have a way of checking all algorithms, we need to make some assumptions of the form: "There is no efficient algorithm to solve X" i.e. X is a hard problem.

We then use problem X to design an encryption scheme. To prove the security of that scheme, we prove that if there is an adversary that can distinguish b/w the encryption of two messages, then they can solve problem X efficiently, which is impossible under our assumption.

The assumptions - some problems are hard

What are some hard problems you know that can be used?

The assumptions - some problems are hard

What are some hard problems you know that can be used?

Factoring large number is a common one. Finding the discrete logarithm is another.

Interestingly, these problems are not hard if there are good enough quantum computers :)

Cool things you can theoretically do

- Zero Knowledge Proofs
- Secret Sharing
- Private Information Retrieval
- Homomorphic Encryption

Complexity Theory

Resources

We looked at time as a valuable resource for computation. What are some other resources you can think of?

Resources

We looked at time as a valuable resource for computation. What are some other resources you can think of?

- Space
- Randomness/Nondeterminism
- Number of processors

How much __________ do we need to solve problem X?

Exchange rates?

How much extra time do I need to simulate random or nondeterministic computation?

How much time can I buy with extra space?

The Power of Nondeterminism

For DFAs, we saw that adding nondeterminism gave us no additional power.

We can simulate a NFA using a DFA!

An analogous fact is not known for more general computation.

P and NP

P is the set of formal languages that can be solved deterministically in polynomial time.

NP is the set of formal languages that can be solved nondeterministically in polynomial time.

NP

NP is equivalent to the set of formal langauge for which we can verify a solution to a problem deterministically in polynomial time.

For example,

$$\mathrm{Sudoku} = \{S : S \text{ is a sudoku puzzle with a solution}\}$$

Given a sudoku puzzle S, and proposed solution S', I can quickly check whether or not S' is indeed a solution for S and hence $S \in \mathrm{Sudoku}$

NP

NP is equivalent to the set of formal langauge for which we can verify a solution to a problem deterministically in polynomial time.

For example,

$$\mathrm{Sudoku} = \{S : S \text{ is a sudoku puzzle with a solution}\}$$

Given a sudoku puzzle S, and proposed solution S', I can quickly check whether or not S' is indeed a solution for S and hence $S \in \mathrm{Sudoku}$

NP represents the class of problems we can reasonably hope to solve. If we can't check whether a proposed solution is correct, how can I be confident about any answer?

Sudoku

Does this sudoku have a solution?

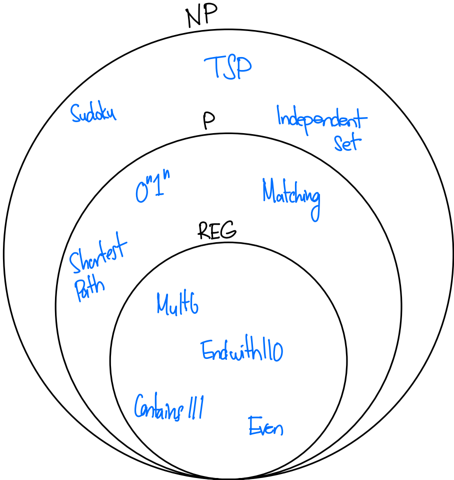
5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Sudoku

Is this a valid solution to the sudoku puzzle?

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Examples



Huge open problem

Is $P = NP$?

I.e. If I can check a solution to a problem deterministically in polynomial time, can I solve the problem deterministically in polynomial time?

Huge open problem

Is $P = NP$?

I.e. If I can check a solution to a problem deterministically in polynomial time, can I solve the problem deterministically in polynomial time?

Intuition: No! It's much easier to check a solution than to solve the problem!

Huge open problem

Is $P = NP$?

I.e. If I can check a solution to a problem deterministically in polynomial time, can I solve the problem deterministically in polynomial time?

Intuition: No! It's much easier to check a solution than to solve the problem!

This is one of 7 Millennium Prize Problems. There is a 1000000 USD prize for each problem. So far, just one out of 7 of the problems have been solved.

Next steps

Depending what you liked here are some topics to explore next! If you liked...

- Mathematical foundations...
  - MAT377 - Mathematical Probability
  - MAT344 - Introduction to Combinatorics
  - MAT332 - Introduction to Graph Theory
  - MAT221/223/224 - Linear Algebra
- Algorithms...
  - CSC263 - Data Structures and Analysis
  - CSC373 - Algorithm Design, Analysis & Complexity
  - CSC473 - Advanced Algorithm Design
  - CSC324 - Principles of Programming Languages
- Formal Languages...
  - CSC448 - Formal Languages and Automata
  - CSC463 - Computational Complexity and Computability