MA/CSSE 474

Theory of Computation

Functions on Languages, Decision Problems
(if time) Logic: Some harder parts

Your Questions?

- Syllabus
- Yesterday's discussion
- Reading Assignments
- HW2
- Anything else

A representation of a number is not the same thing as the number itself

Some "Canonical" Languages from our textbook

- $A^nB^n = \{a^nb^n : n >= 0\}$
- Bal = { strings of balanced parentheses}
- $WW = \{ww : w \in \Sigma^*\}$
- PalEven $\{ww^R : w \in \Sigma^*\}$
- $A^nB^nC^n = \{a^nb^nc^n : n >= 0\}$
- HP_{ALL} = {<T> : T is a Turing machine that eventually halts, no matter what input it is given}
- PRIMES = {*w* : *w* is the binary encoding of a prime integer}

Equivalence Relations

A relation on a set A is any set of ordered pairs of elements of A.

A relation $R \subseteq A \times A$ is an ***equivalence relation*** iff it is:

- reflexive,
- symmetric, and
- transitive.

Examples of equivalence relations:

- Equality
- Lives-at-Same-Address-As
- Same-Length-As
- Contains the same number of a's as

Show that $\equiv_3$ is an equivalence relation

Functions on Languages

Functions whose domains and ranges are languages

maxstring(L) = $\{w \in L: \forall z \in \Sigma^* (z \neq \varepsilon \rightarrow wz \notin L)\}$.

Examples:

- *maxstring*(A^nB^n)
- *maxstring*($\{a\}^*$)

Exercise for later:
What language is *maxstring*($\{b^na: n \geq 0\}$) ?

Let INF be the set of all infinite languages.
Let FIN be the set of all finite languages.

Are the language classes FIN and INF closed under *maxstring*?

Functions on Languages

chop(L) =
$\{w : \exists x \in L\ (x = x_1cx_2,\ x_1 \in \Sigma_L^*,\ x_2 \in \Sigma_L^*,\ c \in \Sigma_L,\ |x_1| = |x_2|,\ \text{and } w = x_1x_2)\}$.

What is *chop*(A^nB^n)?

What is *chop*($A^nB^nC^n$)?

Are FIN and INF closed under *chop*?

Functions on Languages

firstchars(L) =
$\{w : \exists y \in L\ (y = cx \wedge c \in \Sigma_L \wedge x \in \Sigma_L^* \wedge w \in \{c\}^*)\}$.
.

What is *firstchars*(A^nB^n)?

What is *firstchars*($\{a, b\}^*$)?

Are FIN and INF closed under *firstchars*?

Decision Problems

Decision Problems

A ***decision problem*** is simply a problem for which the answer is yes or no (True or False).
A ***decision procedure*** answers a decision problem.

Examples:

- Given an integer *n*, is *n* the product of two consecutive integers?

- The language recognition problem: Given a language *L* and a string *w*, is *w* in *L*?

- We'll explore what we mean by "given a language"

The Power of Encoding

Anything can be encoded as a string.
For example, on a computer everything is encoded as strings of bits.
Assume that we have a scheme for encoding objects (integers, for example).

<*X*> is our notation for the string encoding of *X*.
<*X*, *Y*> is the string encoding of the pair *X*, *Y*.

Problems that don't look like decision problems about strings and languages can be recast into new problems that do look like that.

Example: Web Pattern Matching

Pattern matching on the web:

- Problem: Given a search string *w* and a web document *d*, do they “match”? In other words, should a search engine, on input *w*, consider returning *d*?

- **An instance of the problem has the form** (w, d)

- The language to be decided:
  {<*w*, *d*> : *d* is a candidate match for the string *w*}

The Halting Problem

Does a program always halt?

- Problem: Given a program *p*, written in some some standard programming language L, is *p* guaranteed to halt, no matter what input it is given?

- **An instance of the problem:** Does Python program "print(input())" always halt?

- The language to be decided:
  $HP_{ALL} = \{p \in L : p \text{ halts on all inputs}\}$

Primality Testing

- Problem: Given a nonnegative integer *n*, is it prime?
- **An instance of the problem:** Is 9 prime?
- To encode the problem we need a way to encode each instance: encode each nonnegative integer as a binary string.
- The language to be decided (2 ways to express it):

PRIMES = {w : *w* is the binary encoding of a prime integer}.

Equivalently:

PRIMES = {<n> : n is a prime integer}.

Graph Connectivity

- Problem: Given an undirected graph *G*, is it connected?
- Instance of the problem:

1 —— 2 —— 3

\ \

4 5

- Encoding of the problem: Let *V* be a set of binary representations of numbers, one for each vertex in *G*.

Then we construct $\langle G \rangle$ as follows:

- Write |*V*| as a binary number, then write "/".
- Write a list of edges, each pair of binary numbers represents one edge.
- Separate all such binary numbers by "/".

Full encoding of the above graph: 101/1/10/1/100/10/101/10/11 (the 101 is |*V*|)

- The language to be decided:

CONNECTED = $\{w \in \{0, 1, /\}^* : w = n_1/n_2/\ldots n_i$, where each n_i is a binary string and *w* encodes a connected graph, as described above}.

Protein Sequence Allignment

- Problem: Given a protein fragment *f* and a complete protein molecule *p*, could *f* be a fragment from *p*?

- Encoding of the problem: Represent each protein molecule or fragment as a sequence of amino acid residues. Assign a letter to each of the 20 possible amino acids. So a protein fragment might be represented as `AGHTYWDNR`.

- The language to be decided:

{<*f*, *p*> : *f* could be a fragment from *p*}.

Computation problems and their Language Formulations may be Equivalent

By equivalent we mean that either problem can be ***reduced to*** the other.

If we have a machine to solve either problem,

- we can use it to build a machine to solve the other,
- using only the starting machine
- and other functions that can be built using machines of equal or lesser power.

We will see that reduction does not always preserve efficiency!

Turning a Problem into a Language Recognition Problem

Cast multiplication as a language recognition problem:

- Problem: Given two nonnegative integers, compute their product.
- Encode the problem: Transform computing into verification.
- The language to be decided:

INTEGERPROD = {w of the form:
<int_1>x<int_2>=<int_3>,
where each <int_n> is an encoding (decimal in this case) of an integer,
and $int_3 = int_1 * int_2$}

`12x9=108` $\in$ *INTEGERPROD*
`12=12` $\notin$ *INTEGERPROD*
`12x8=108` $\notin$ *INTEGERPROD*

Show the Equivalence

INTEGERPROD = {w of the form: <int_1>x<int_2>=<int_3>, where each <int_n> is an encoding (decimal) of an integer, and $int_3 = int_1 * int_2$}

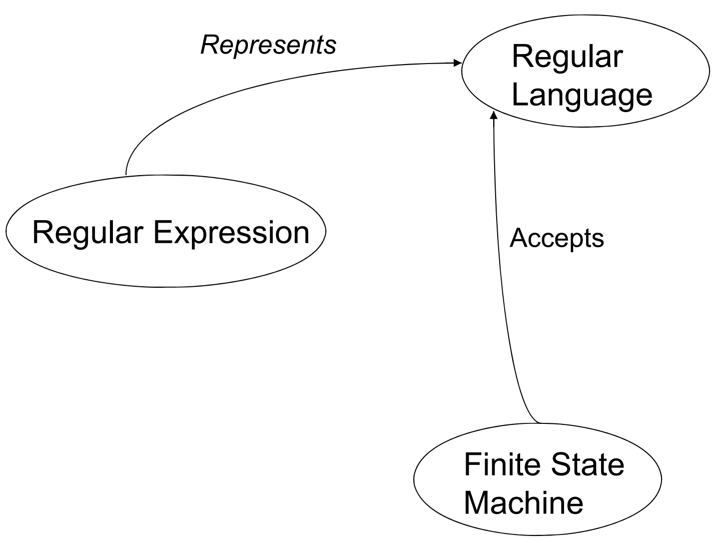
Reduce INTEGERPROD to Mult: Given a multiplication function Mult for integers, we can build a procedure that recognizes the *INTEGERPROD* language: ***We'll do this together***

Reduce Mult to INTEGERPROD : Given a function $R(w)$ that recognizes *INTEGERPROD*, we can build a procedure *Mult*(m,n) that computes the product of two integers: ***You should figure this out before Monday's class***

Regular Languages (formally)

More on Finite State Machines

Regular Languages



A real-world FSM Example

A

B

C

D

Recap - Definition of a DFSM

$M = (K, \Sigma, \delta, s, A)$, where:

The D is for Deterministic

- K is a finite set of ***states***
- Σ is a (finite) ***alphabet***
- $s \in K$ is the ***initial state*** (a.k.a. start state)
- $A \subseteq K$ is the set of ***accepting states***
- $\delta: (K \times \Sigma) \rightarrow K$ is the ***transition function***

Sometimes we will put an M subscript on K, Σ, δ, s, or A (for example, s_M), to indicate that this component is part of machine M.

Acceptance by a DFSM

Informally, *M* ***accepts*** a string *w* iff *M* winds up in some element of *A* after it has finished reading *w*.

The ***language accepted by M,*** denoted *L*(*M*), is the set of all strings accepted by *M*.

But we need more formal notations if we want to prove things about machines and languages.

On day 1, we saw one notation, the extended delta function.

Today we examine the book's notation, ⊢. Unicode 22A2. That symbol is commonly called *turnstile* or *tee*. It is often read as "derives" or "yields"

Configurations of a DFSM

A ***configuration*** of a DFSM *M* is an element of:

$$K \times \Sigma^*$$

It captures the two things that affect *M*'s future behavior:

- its current state
- the remaining input to be read.

The ***initial configuration*** of a DFSM *M*, on input *w*, is:

$$(s_M, w)$$

Where s_M is the start state of M.

The "Yields" Relations

The ***yields-in-one-step*** relation: $\vdash_M$:

$(q, w) \vdash_M (q', w')$ iff

- $w = a\, w'$ for some symbol $a \in \Sigma$, and
- $\delta(q, a) = q'$

The ***yields-in-zero-or-more-steps*** relation: $\vdash_M{}^*$

$\vdash_M{}^*$ is the reflexive, transitive closure of $\vdash_M$.

Note that this accomplishes the same thing as the "extended delta function" that we considered on Day 1. Two notations for the same concept.

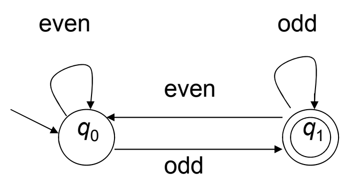
Computations Using FSMs

A ***computation*** by M is a finite sequence of configurations $C_0, C_1, \ldots, C_n$ for some $n \geq 0$ such that:

- C_0 is an initial configuration,
- C_n is of the form (q, ε), for some state $q \in K_M$,
- $\forall i \in \{0, 1, \ldots, n-1\}\ (C_i \vdash_M C_{i+1})$

An Example Computation

A FSM M that accepts decimal representations of odd integers:



On input 235, the configurations are:

$$
\begin{aligned}
(q_0, 235) &\vdash_M (q_0, 35)\\
&\vdash_M (q_1, 5)\\
&\vdash_M (q_1, \varepsilon)
\end{aligned}
$$

Thus $(q_0, 235) \vdash_M{}^* (q_1, \varepsilon)$

Accepting and Rejecting

A DFSM M ***accepts*** a string w iff:

$(s_M, w) \vdash_M{}^* (q, \varepsilon)$, for some $q \in A_M$

A DFSM M ***rejects*** a string w iff:

$(s_M, w) \vdash_M{}^* (q, \varepsilon)$, for some $q \notin A_M$

The ***language accepted by*** M, denoted $L(M)$, is the set of all strings accepted by M.

Theorem: Every DFSM M, in configuration (q, w), halts after $|w|$ steps.

Thus every string is either accepted or rejected by a DFSM.

Proof of Theorem

Theorem: Every DFSM *M*, in configuration (q, w), halts after |*w*| steps.

Proof: by induction on |w|

Base case: If w is ε, it halts in 0 steps.

Induction step: Assume true for strings of length n and show for strings of length n+1.

Let $w \in \Sigma^*$, $w \neq \varepsilon$. Then $|w| = n+1$ for some $n \in \mathbb{N}$.

So w must be au for some $a \in \Sigma$, $u \in \Sigma^*$, $|u| = n$.

Let q' be $\delta(q, a)$. By definition of $\vdash$, $(q,w) \vdash_M (q', x)$

By the induction hypothesis, starting from configuration (q', u), M halts after n steps.

Thus, starting from the original configuration, M halts after n+1 steps.

Turning Problems Into Language Recognition Problems

Cast sorting as language recognition decision problem:

- Problem: Given a list of integers, sort it.
- Encoding of the problem: Transform the sorting problem into one of examining a pair of lists.
- The language to be decided:

$L = \{w_1 \# w_2 : \exists n \geq 1$
(w_1 is of the form $<int_1, int_2, \ldots int_n>$,
w_2 is of the form $<int_1, int_2, \ldots int_n>$, and
w_2 contains the same objects as w_1 and
w_2 is sorted)}

Examples:

`<1,5,3,9,6>#<1,3,5,6,9>` $\in L$

`<1,5,3,9,6>#<1,2,3,4,5,6,7>` $\notin L$

Turning Problems Into Decision Problems

Cast database querying as decision:

- Problem: Given a database and a query, execute the query.
- Encoding of the problem: Transform the query execution problem into evaluating a reply for correctness.
- The language to be decided:

L = {*d* # *q* # *a*:
d is an encoding of a database,
q is a string representing a query, and
a is the correct result of applying *q* to *d*}

Example:

```
(name, age, phone), (John, 23, 567-1234)
(Mary, 24, 234-9876)#(select name age=23)#
(John)
```
$\in L$

The Traditional Problems and their Language Formulations are Equivalent

By *equivalent* we mean that either problem can be ***reduced to*** the other.

If we have a machine to solve one, we can use it to build a machine to do the other, using only the starting machine and other functions that can be built using machines of equal or lesser power.

Reduction does not always preserve efficiency!

Turning Problems into Language Recognition Problems

Cast multiplication as language recognition:

- Problem: Given two nonnegative integers, compute their product.
- Encode the problem: Transform computing into verification.
- The language to be decided:

INTEGERPROD = {w of the form:
$<int_1>$x$<int_2>$=$<int_3>$, where each $<int_n>$ is an encoding (decimal in this case) of an integer, and
$int_3 = int_1 * int_2$}

`12x9=108` $\in$ *INTEGERPROD*
`12=12` $\notin$ *INTEGERPROD*
`12x8=108` $\notin$ *INTEGERPROD*

Show the Equivalence

Consider the multiplication language example:
INTEGERPROD = {w of the form:
$<int_1>$x$<int_2>$=$<int_3>$, where each $<int_n>$ is an encoding (decimal in this case) of an integer, and
$int_3 = int_1 * int_2$}

Given a multiplication function for integers, we can build a procedure that recognizes the *INTEGERPROD* language: **(We will do this today)**

Given a function *R*(*w*) that recognizes *INTEGERPROD*, we can build a procedure *Mult*(*m*,*n*) that computes the product of two integers: **(figure this out during the weekend)**

Logic: Propositional and first-order

Review of material form *Grimaldi* Chapter 2
Based on *Rich* Chapter 8

Logic: Propositional and first-order

From Rich, Appendix A

Most of this material also appears in Grimaldi's Discrete Math book, Chapter 2

I used these slides and exercises in the past. Since 2012, I have not been going through them in class because most are background material from the perquisite course. I am keeping all of the slides, for context and in case you find them helpful. If you want to look at these, but only at the most important slides, focus on the ones whose titles are in color,

Boolean (Propositional) Logic Wffs

A ***wff*** (well-formed formula) is any string that is formed according to the following rules:

1. A propositional symbol (variable or constant) is a wff.
2. If P is a wff, then $\neg P$ is a wff.
3. If P and Q are wffs, then so are:
   $P \vee Q$, $P \wedge Q$, $P \rightarrow Q$, $P \leftrightarrow Q$, and (P).

P	Q	$\neg P$	$P \vee Q$	$P \wedge Q$	$P \rightarrow Q$	$P \leftrightarrow Q$
True	*True*	*False*	*True*	*True*	*True*	*True*
True	*False*	*False*	*True*	*False*	*False*	*False*
False	*True*	*True*	*True*	*False*	*True*	*False*
False	*False*	*True*	*False*	*False*	*True*	*True*

Note that $P \rightarrow Q$ is an abbreviation for $\neg P \vee Q$. What does $P \leftrightarrow Q$ abbreviate?

When are Wffs True?

- A wff is ***valid*** or is a ***tautology*** iff it is true for all assignments of truth values to the variables it contains.
- A wff is ***satisfiable*** iff it is true for at least one assignment of truth values to the variables it contains.
- A wff is ***unsatisfiable*** iff it is false for all assignments of truth values to the variables it contains.
- Two wffs P and Q are ***equivalent***, written $P \equiv Q$, iff they have the same truth values for every assignment of truth values to the variables they contain.

$P \vee \neg P$ is a tautology:

P	$\neg P$	$P \vee \neg P$
True	*False*	*True*
False	*True*	*True*

Entailment

A set *S* of wffs ***logically implies*** or ***entails*** a conclusion *Q* iff, whenever all of the wffs in *S* are true, *Q* is also true.

Example:

$\{A \wedge B \wedge C, D\}$ (trivially) entails $A \rightarrow D$

Inference Rules

- An inference rule is ***sound*** iff, whenever it is applied to a set *A* of axioms, any conclusion that it produces is entailed by *A*.
- An entire proof is sound iff it consists of a sequence of inference steps each of which was constructed using a sound inference rule.
- A set of inference rules *R* is ***complete*** iff, given any set *A* of axioms, all statements that are entailed by *A* can be proved by applying the rules in *R*.

Some Sound Inference Rules

You do not have to memorize the rules or their names, but given the list of rules, you should be able to use them in simple ways

- ***Modus ponens***: From ($P \rightarrow Q$) and P, conclude Q.
- ***Modus tollens***: From ($P \rightarrow Q$) and $\neg Q$, conclude $\neg P$.
- ***Or introduction***: From P, conclude ($P \vee Q$).
- ***And introduction***: From P and Q, conclude ($P \wedge Q$).
- ***And elimination***: From ($P \wedge Q$), conclude P or conclude Q.
- ***Syllogism***: From ($P \rightarrow Q$) and ($Q \rightarrow R$), conclude ($P \rightarrow R$).

Additional Sound Inference Rules

- ***Quantifier exchange***:
  - From $\neg\exists x$ (P), conclude $\forall x$ ($\neg P$).
  - From $\forall x$ ($\neg P$), conclude $\neg\exists x$ (P).
  - From $\neg\forall x$ (P), conclude $\exists x$ ($\neg P$).
  - From $\exists x$ ($\neg P$), conclude $\neg\forall x$ (P).
- ***Universal instantiation***: For any constant C, from $\forall x$ ($P(x)$), conclude $P(C)$.
- ***Existential generalization***: For any constant C, from $P(C)$ conclude $\exists x$ ($P(x)$).

First-Order Logic

A **term** is a variable, constant, or function application.
A ***well-formed formula*** ***(wff)*** in first-order logic is an expression that can be formed by:

- If P is an n-ary ***predicate*** and each of the expressions $x_1, x_2, \ldots, x_n$ is a term, then an expression of the form $P(x_1, x_2, \ldots, x_n)$ is a wff. If any variable occurs in such a wff, then that variable occurs ***free*** *in* $P(x_1, x_2, \ldots, x_n)$.
- If P is a wff, then $\neg P$ is a wff.
- If P and Q are wffs, then so are $P \vee Q$, $P \wedge Q$, $P \rightarrow Q$, and $P \leftrightarrow Q$.
- If P is a wff, then (P) is a wff.
- If P is a wff, then $\forall x\ (P)$ and $\exists x\ (P)$ are wffs. Any free instance of x in P is ***bound*** by the quantifier and is then no longer free.

Note that the definition is recursive, so proofs about wffs are likely to be by induction.

Example of a ternary predicate:
Pythagorean(a, b, c) is true iff $a^2 + b^2 = c^2$.
Pythagorean(5, 12, 13) has no free variables,
Pythagorean(x, y, 13) has free variables

For last bullet, consider: $\exists x\ (\exists y\ (x \in \mathbb{N} \wedge y \in \mathbb{N} \wedge$ Pythagorean(x, y, 13))) . x and y are bound by the $\exists$ quantifier here.
We can abbreviate this $\exists x, y \in \mathbb{N}$ (Pythagorean(x, y, 13))

Sentences

A wff with no free variables is called a ***sentence*** or a ***statement***.

1. *Bear*(*Smokey*).
2. $\forall x\ (Bear(x) \rightarrow Animal(x))$.
3. $\forall x\ (Animal(x) \rightarrow Bear(x))$.
4. $\forall x\ (Animal(x) \rightarrow \exists y\ (Mother\text{-}of(y, x)))$.
5. $\forall x\ ((Animal(x) \wedge \neg Dead(x)) \rightarrow Alive(x))$.

Which of these sentences are true in the everyday world?

A ***ground instance*** is a sentence that contains no variables, such as #1.

"Smokey" is a constant, as is the Bear predicate.

The first is a sentence, if we assume that Smokey is a constant

True
True
False
True (if we assume that "exists" is not temporal)
True

Interpretations and Models

- An ***interpretation*** for a sentence *w* is a pair (*D*, *I*), where *D* is a universe of objects.
  I assigns meaning to the symbols of *w*:
  - it assigns values, drawn from *D*, to the constants in *w*
  - it assigns functions and predicates (whose domains and ranges are subsets of *D*) to the function and predicate symbols of *w*.
- A ***model*** of a sentence *w* is an interpretation that makes *w* true. For example, let *w* be the sentence:
  $\forall x\,(\exists y\,(y < x))$. Find a model for this sentence.
- A sentence *w* is ***valid*** iff it is true in all interpretations.
- A sentence *w* is ***satisfiable*** iff there exists *some* interpretation in which *w* is true.
- A sentence *w* is ***unsatisfiable*** iff $\neg w$ is valid.

An interpretation of the sentence on this page is the integers, with < assigned to the normal < predicate.
Note that we use infix x < y instead of the formal <(x, y).

What about the sentence $\exists x\,(\forall y\,(x*y = 0))$? A model for this sentence is the integers with the normal meanings of =, 0, and *.
Note that this involves assigning a value to the constant 0 in the expression.

Examples (Valid, satisfiable, unsatisfiable?)

- $\forall x\,((P(x) \wedge Q(Smokey)) \rightarrow P(x))$.
- $\neg(\forall x\,(P(x) \vee \neg(P(x)))$.
- $\forall x\,(P(x, x))$.

First one is valid, independent of the values of P, Q, and Smokey

Second is invalid

Third depends on(Domain, Interpretation) Example: satisfied by (integers, <=), but not (integers, <)

A Simple Proof

Assume the following three axioms:

[1] $\forall x\ (P(x) \wedge Q(x) \rightarrow R(x))$.
[2] $P(X_1)$.
[3] $Q(X_1)$.

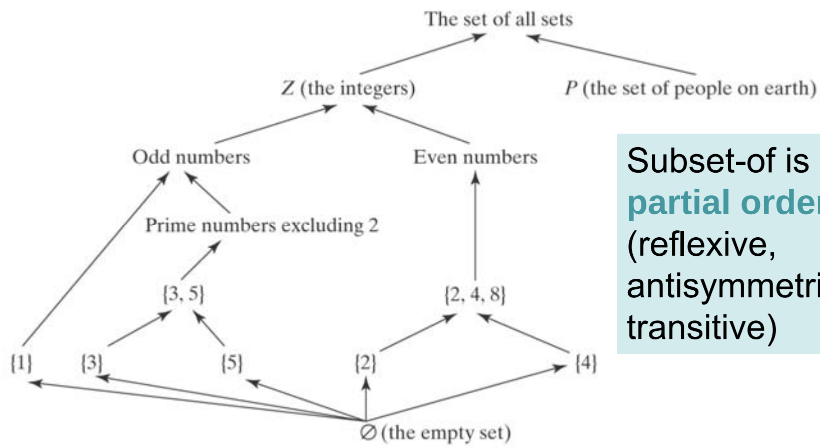
We prove $R(X_1)$ as follows:

[4]	$P(X_1) \wedge Q(X_1) \rightarrow R(X_1)$.	(Universal instantiation, [1].)
[5]	$P(X_1) \wedge Q(X_1)$.	(And introduction, [2], [3].)
[6]	$R(X_1)$.	(Modus ponens, [5], [4].)

Definition of a Theory

- A first-order **theory** is a set of axioms and the set of all theorems that can be proved, using a set of sound and complete inference rules, from those axioms.
- A theory is logically **complete** iff, for every sentence P in the language of the theory, either P or ¬P is a theorem.
- A theory is **consistent** iff there is no sentence P such that both P and ¬ P are theorems.
  - If there is such a sentence, then the theory contains a **contradiction** and is **inconsistent**.
- Let w be an interpretation of a theory. The theory is **sound** with respect to w if every theorem in the theory corresponds to a statement that is true in w.

Subset-of as a Partial Order



Subset-of is a **partial order** (reflexive, antisymmetric, transitive)

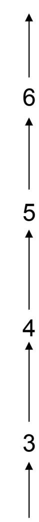
Total Order

A ***total order*** $R \subseteq A \times A$ is a partial order that has the additional property that:

$$\forall x, y \in A\ ((x, y) \in R \lor (y, x) \in R).$$

Example: $\leq$ on the rational numbers

If R is a total order defined on a set A, then the pair (A, R) is a ***totally ordered set***.



Infinite Descending Chain

- A partially ordered set (S, <) has an infinite descending chain if there is an infinite set of elements $x_0, x_1, x_2, \ldots \in S$ such that
 $\forall i \in \mathbb{N}(x_{i+1} < x_i)$
- Example:
In the rational numbers with <,
 1/2 > 1/3 > 1/4 > 1/5 > 1/6 > …
is an infinite descending chain

Well-Founded and Well-Ordered Sets

Given a partially ordered set (*A*, *R*), an ***infinite descending chain*** is subset *B* of *A* that is a totally ordered with respect to *R*, that has no minimal element.

If (*A*, *R*) contains no infinite descending chains then it is called a ***well-founded set***.

- Used for halting proofs.

If (*A*, *R*) is a well-founded set and *R* is a total order, then (*A*, *R*) is called a ***well-ordered set***.

- Used in induction proofs
- The positive integers are well-ordered
- The positive rational numbers are not well-ordered (with respect to normal <)

Exercise: With one or two other students, come up with a relation R on S={rεrationals: 0 < r < 1} suc that (S,R) is well-ordered. R does not need to be consistent with the usual < ordering. Hint: Think diagonal.

Mathematical Induction

Because the integers ≥ *b are well-ordered:*

The ***principle of mathematical induction***:

If: $P(b)$ is true for some integer base case b, and
For all integers $n \geq b$, $P(n) \rightarrow P(n+1)$

Then: For all integers $n \geq b$, $P(n)$

An induction proof has three parts:

1. A clear statement of the assertion P.
2. A proof that that P holds for some base case b, the smallest value with which we are concerned.
3. A proof that, for all integers $n \geq b$, if $P(n)$ then it is also true that $P(n+1)$. We'll call the claim $P(n)$ the ***induction hypothesis***.

Sum of First *n* Positive Odd Integers

The sum of the first n odd positive integers is n^2. We first check for plausibility:

(n = 1) $1 = 1 = 1^2$.
(n = 2) $1 + 3 = 4 = 2^2$.
(n = 3) $1 + 3 + 5 = 9 = 3^2$.
(n = 4) $1 + 3 + 5 + 7 = 16 = 4^2$, and so forth.

The claim appears to be true, so we should prove it.

Sum of First *n* Positive Odd Integers

Let $Odd_i = 2(i - 1) + 1$ denote the i^{th} odd positive integer. Then we can rewrite the claim as:

$$\forall n \geq 1 \quad (\sum_{i=1}^{n} Odd_i = n^2)$$

For reference; we will not do this in class

The proof of the claim is by induction on n:
Base case: take 1 as the base case. $1 = 1^2$.

Prove: $$\forall n \geq 1((\sum_{i=1}^{n} Odd_i = n^2) \rightarrow (\sum_{i=1}^{n+1} Odd_i = (n+1)^2))$$

$$\sum_{i=1}^{n+1} Odd_i = \sum_{i=1}^{n} Odd_i + Odd_{n+1}$$

$= n^2 + Odd_{n+1}$. (Induction hypothesis.)
$= n^2 + 2n + 1$. ($Odd_{n+1} = 2(n+1-1) + 1 = 2n + 1$.)
$= (n + 1)^2$.

Note that we start with one side of the equation we are trying to prove, and transform to get the other side. We do **not** treat it like solving an equation, where we transform both sides in the same way.

Strong induction

- To prove that predicate P(n) is true for all n≥b:
  - Show that P(b) is true [and perhaps P(b+1) *]
  - Show that for all j>b, if P(k) is true for all k with b≤ k<j, then P(j) is true. In symbols:

$$\forall j > b\ ((\forall k\ (b \leq k < j \rightarrow P(k)) \rightarrow P(j))$$

* We may have to show it directly for more than one or two values, but there should always be a finite number of base cases.

Fibonacci Running Time

- From Weiss, Data Structures and Problem Solving with Java, Section 7.3.4
- Consider this function to recursively calculate Fibonacci numbers:
  $F_0=0$ $F_1=1$ $F_n = F_{n-1}+F_{n-2}$ if $n\geq 2$.
  -
```
def fib(n):
    if n <= 1:
        return n
    return fib(n-1) + fib(n-2)
```
- Let C_N be the total number of calls to *fib* during the computation of *fib(N)*.
- It's easy to see that $C_0=C_1=1$, and if $N \geq 2$, $C_N = C_{N-1} + C_{N-2} + 1$.
- **Prove that** for $N \geq 3$, $C_N = F_{N+2} + F_{N-1} -1$.

Base cases, N=3, N=4
Assume by induction that if $N>=3$, then C_N and C_{N+1} are the right things.
Show that C_{N+2} is the right thing.

$$\begin{aligned} C_{N+2} &= 1 + C_N + C_{N+1} \\ &= (F_{N+2} + F_{N-1} - 1) + (F_{N+3} + F_N - 1) + 1 \\ &= F_{N+4} + F_{N+1} - 1 \\ &= F_{N+2+2} + F_{N+2-1} - 1 \end{aligned}$$