HW 9 due tomorrow (Apr 29).
Quiz 13 due this Friday.

Class 28: Review. Gödel's Incompleteness Theorem.

University of Virginia
CS3120: DMT2
https://weikailin.github.io/cs3120-toc
Wei-Kai Lin

Plan

Review for Final Exam

Gödel's Incompleteness Theorem

Textbook [TCS] Section 11.2

https://introtcs.org/public/lec_09_godel.html#g%C3%B6dels-incompleteness-theorem-computational-variant

Gemini generated cover image.

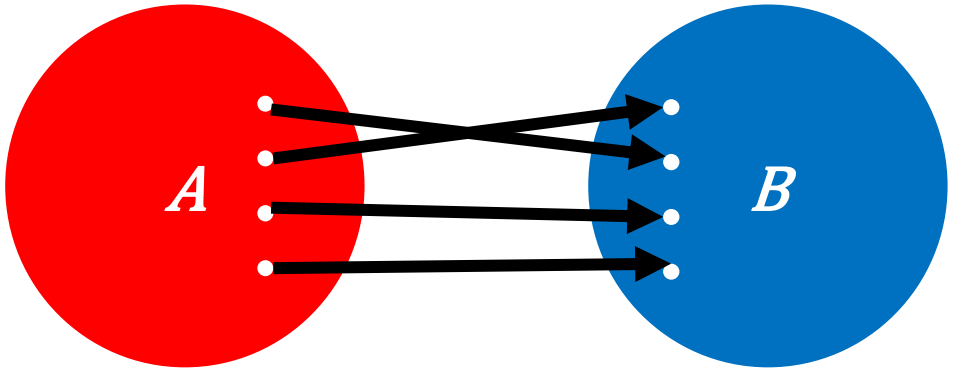
Module 0: Strings and Cardinality

Cardinality of (Infinite) Sets

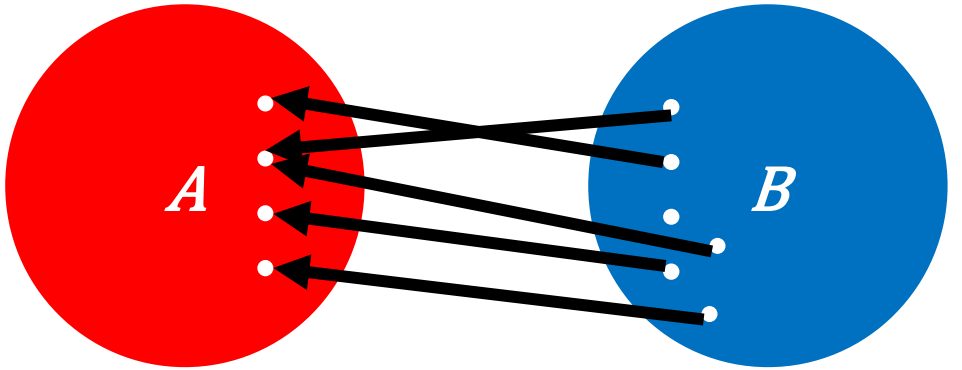
Definition. Two sets have the *same cardinality* if there is a bijection between the two sets.
We denote this as $|A| = |B|$.
The cardinality of the set

$$[k] = \{\, n \mid n \in \mathbb{N} \wedge n < k \,\}$$

is k**.**

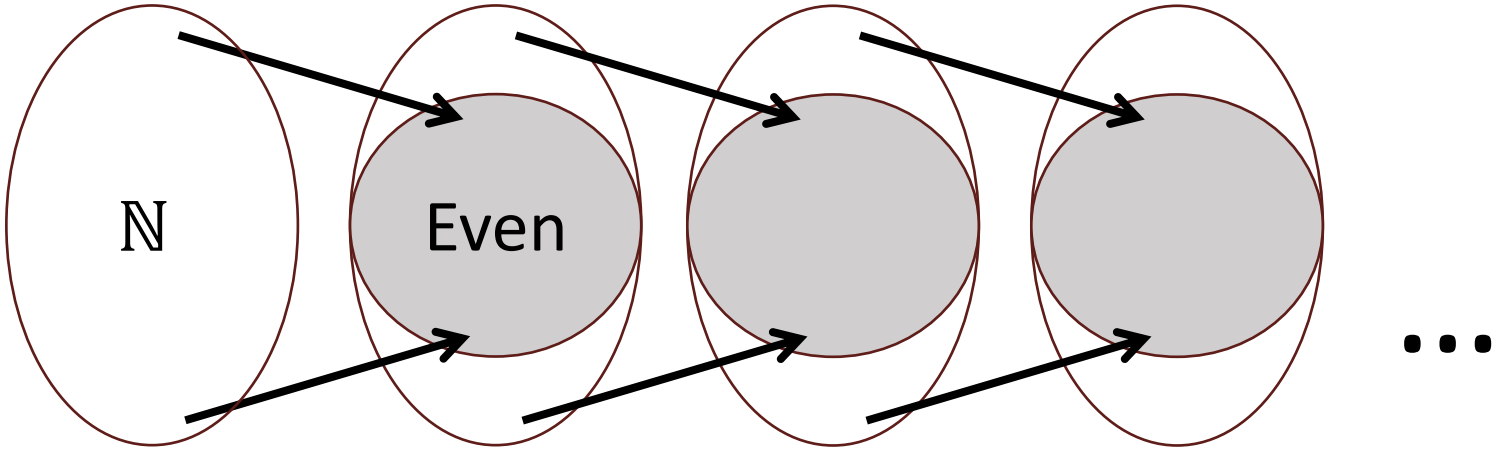


Definition. If there exists a **surjective function** from sets B to A, then we say the cardinality of B is ***greater than or equal to*** the cardinality of A.
We denote this as $|A| \leq |B|$.



Definition. A set is *Dedekind-infinite* if and only if it has the same cardinality as some **strict subset** of itself.

$\mathbb{N}$ is Dedekind-infinite:



Definition. A set S is *countable* if and only if

$$|S| \leq |\mathbb{N}|.$$

We say S' is uncountable if S' is not countable.

Cantor's Theorem: (~1874)

For **all** sets S,

$$|pow(S)| > |S|.$$

Note: this isn't what the TCS book calls *Cantor's Theorem* but is what most people call "Cantor's Theorem". Cantor came up with the diagonalization argument, with help from Dedekind. The proof we'll see soon of Cantor's Theorem is believed to have been first done by Hessenberg (1906).

Assume $g: \mathbb{N} \to \{0,1\}^\infty$ a bijection.

$y \in \mathbb{N}$	g(y)[0]	g(y)[1]	g(y)[2]	g(y)[3]	g(y)[4]	g(y)[5]	
0	0	1	1	0	1	0	⋯
1	1	1	0	1	0	1	⋯
2	0	1	1	0	1	0	⋯
3	0	0	1	0	1	1	⋯
4	1	1	0	1	0	1	⋯
5	0	1	0	1	1	0	⋯
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋱

Let s := (¬g(0)[0], ¬g(1)[1], ¬g(2)[2], ¬g(3)[3], …).
$s \in \{0,1\}^\infty$ by definition of $\{0,1\}^\infty$.
Is there any $y \in \mathbb{N}$ s.t. $g(y) = s$?

Diagonal Argument

Some Sets

- $\mathbb{N}$, binary strings $\{0,1\}^*$, English strings, …
- Boolean functions $\{f : \{0,1\}^* \to \{0,1\}\}$
- Real numbers
- $0.b_0b_1b_2 \ldots \in [0,1]$ in base 2
- Infinite strings $(b_0, b_1, b_2, \ldots) \in \{0,1\}^\infty$

Also, power sets or product sets of them.
Exercise: Compare their cardinality.

Module 1: Regular Expressions and Deterministic Finite Automata

Problems and Instances

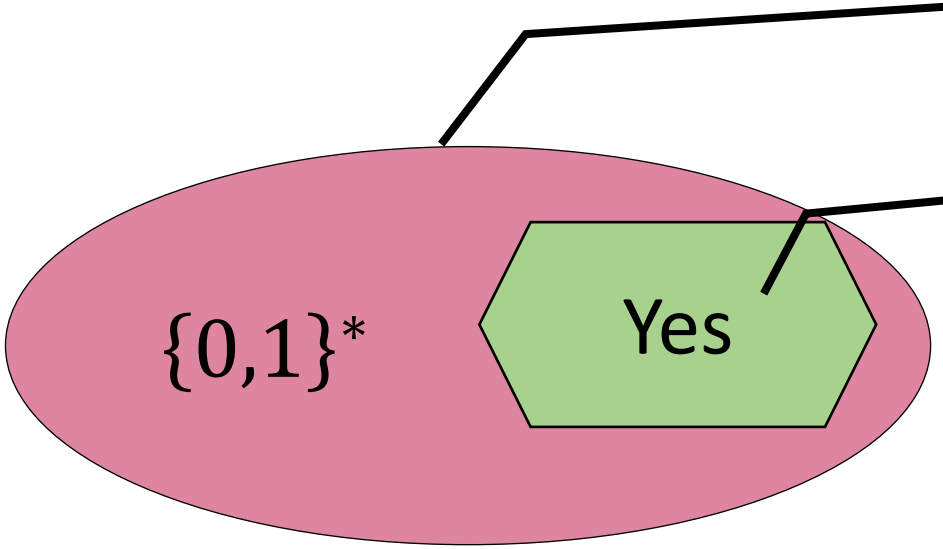
Definition. A *problem* is a subset $P \subseteq \{0,1\}^*$ of binary strings.

Definition. An *instance* (of a problem) is a binary string $x \subseteq \{0,1\}^*$.

Convention (ie, Definition)

Problem = Language = Binary Function

Any binary string $x \in \{0,1\}^*$ is an *instance*.



The subset of Yes instances:
The *Problem*, or the *Language,*
or the Boolean function
that outputs 1

Regular Expressions and DFA

DFA:
easier to think as graphs

RE: We use the notation frequently, eg,
$1^4 0 = 11110$

1^n = repeat 1 for n times

Definition 6.2 (Deterministic Finite Automaton)

A deterministic finite automaton (DFA) with C states over $\{0,1\}$ is a pair $(T, \mathcal{S})$ with $T : [C] \times \{0,1\} \to [C]$ and $\mathcal{S} \subseteq [C]$. The finite function T is known as the *transition function* of the DFA. The set $\mathcal{S}$ is known as the set of *accepting states*.

Let $F : \{0,1\}^* \to \{0,1\}$ be a Boolean function with the infinite domain $\{0,1\}^*$. We say that $(T, \mathcal{S})$ *computes* a function $F : \{0,1\}^* \to \{0,1\}$ if for every $n \in \mathbb{N}$ and $x \in \{0,1\}^n$, if we define $s_0 = 0$ and $s_{i+1} = T(s_i, x_i)$ for every $i \in [n]$, then

$$s_n \in \mathcal{S} \Leftrightarrow F(x) = 1$$

Definition 6.6 (Regular expression)

A *regular expression* e over an alphabet Σ is a string over $\Sigma \cup \{(,), |, *, \emptyset, \text{""}\}$ that has one of the following forms:

1. $e = \sigma$ where $\sigma \in \Sigma$
2. $e = (e'|e'')$ where e', e'' are regular expressions.
3. $e = (e')(e'')$ where e', e'' are regular expressions. (We often drop the parentheses when there is no danger of confusion and so write this as $e'\ e''$.)
4. $e = (e')^*$ where e' is a regular expression.

Finally we also allow the following "edge cases": $e = \emptyset$ and $e = \text{""}$. These are the regular expressions corresponding to accepting no strings, and accepting only the empty string respectively.

Reg-Fun = DFA-Comp

Theorem 6.17 (DFA and regular expression equivalency)
Let $F : \{0,1\}^* \to \{0,1\}$. Then F is regular if and only if there exists a DFA $(T, \mathcal{S})$ that computes F.

Can build DFA if and only if

Can build regular expression (with constructions).

Implication: Closure:

Theorem 6.19 (Closure of regular expressions)
Let $f : \{0,1\}^k \to \{0,1\}$ be any finite Boolean function, and let $F_0, \ldots, F_{k-1} : \{0,1\}^* \to \{0,1\}$ be regular functions. Then the function $G(x) = f(F_0(x), F_1(x), \ldots, F_{k-1}(x))$ is regular.

- Functions differ from {DFA, Reg. Exp., Programs}.
- A function class is a set of functions (eg, Reg-Fun, DFA-Comp).
- Compare classes by constructions.

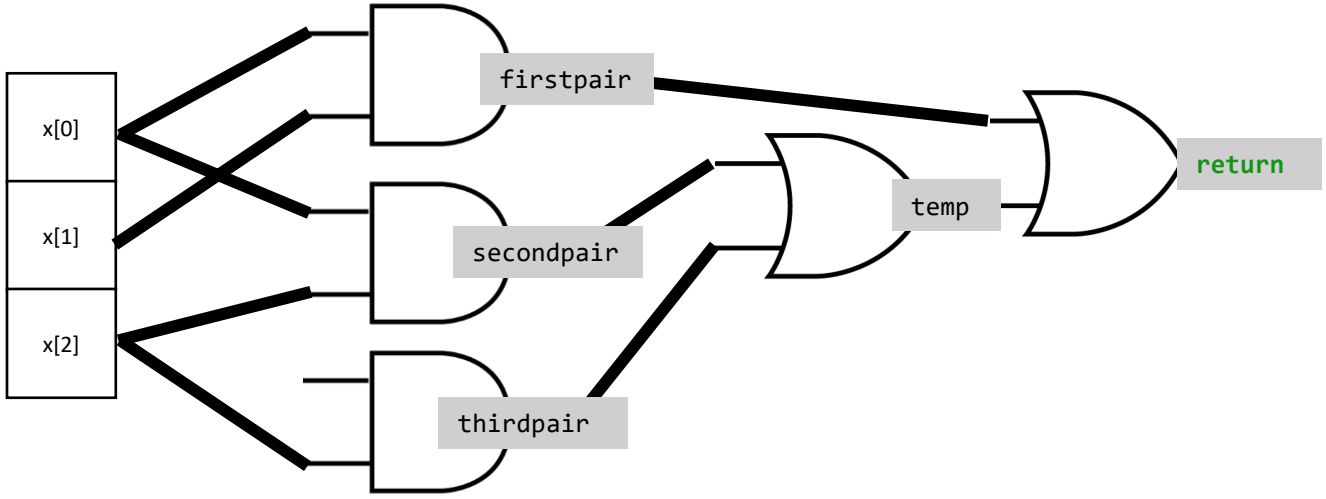
Module 2: Circuits

Circuit: Built from “Boolean Gates”

A set of gates: {AND, OR, NOT}

Another set: {NAND}

Written as a DAG, or a “straightline program”




```
def MED(X[0],X[1],X[2]):
    firstpair = AND(X[0],X[1])
    secondpair = AND(X[1],X[2])
    thirdpair = AND(X[0],X[2])
    temp = OR(secondpair,thirdpair)
    return OR(firstpair,temp)
```

Universal Set of Gates

{AND,OR,NOT} is universal.

{NAND} is universal.

{AND,OR} is Not.

Theorem 3.12 (NAND is a universal operation)

For every Boolean circuit C of s gates, there exists a NAND circuit C' of at most $3s$ gates that computes the same function as C.

Definition 3.20 (General straight-line programs)

Let $\mathcal{F} = \{f_0, \ldots, f_{t-1}\}$ be a finite collection of Boolean functions, such that $f_i : \{0,1\}^{k_i} \rightarrow \{0,1\}$ for some $k_i \in \mathbb{N}$. An *$\mathcal{F}$ program* is a sequence of lines, each of which assigns to some variable the result of applying some $f_i \in \mathcal{F}$ to k_i other variables. As above, we use `x[` i `]` and `y[` j `]` to denote the input and output variables.

We say that $\mathcal{F}$ is a *universal set of operations* (also known as a universal gate set) if there exists a $\mathcal{F}$ program to compute the function $NAND$.

Circuit Size, and Class SIZE(s)

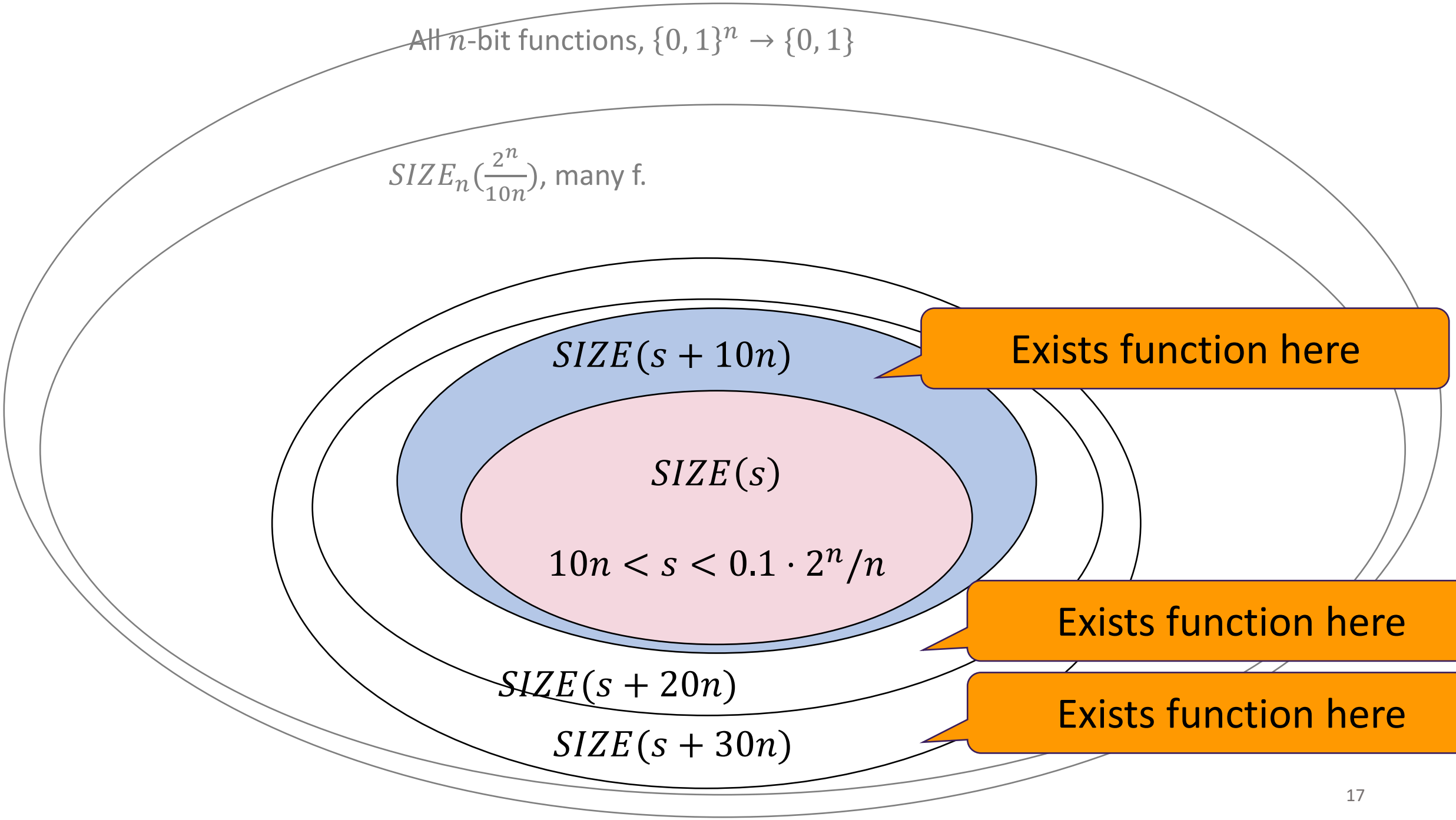
Circuit size: number of gates in a circuit

SIZE(s): the set { function f | there exists s-gate circuit such that computes f}

Is circuit $C \in SIZE(s)$?

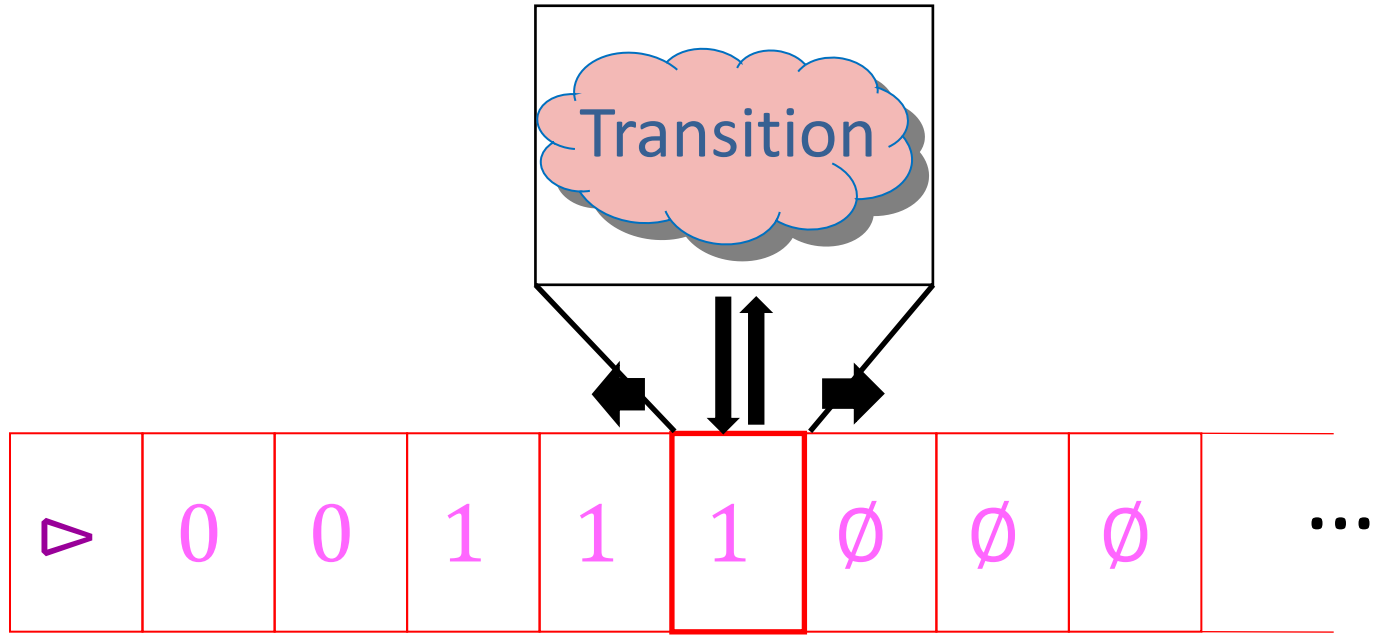
$\in EVEN$?

Category Error!


All n-bit functions, $\{0,1\}^n \to \{0,1\}$
$SIZE_n(\frac{2^n}{10n})$, many f.
$SIZE(s+10n)$
$SIZE(s)$
$10n < s < 0.1 \cdot 2^n/n$
$SIZE(s+20n)$
$SIZE(s+30n)$
Exists function here
Exists function here
Exists function here

Module 3: Turing Machines and Computability

Turing Machine



Tape initially contains the input followed by blanks

If transition “halts”, → tape contains the output

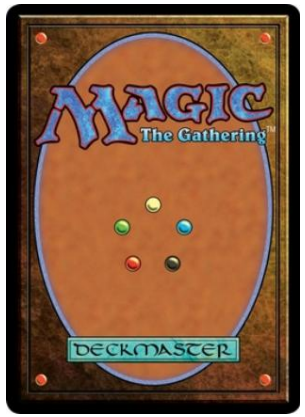
Semi-infinite tape (memory)

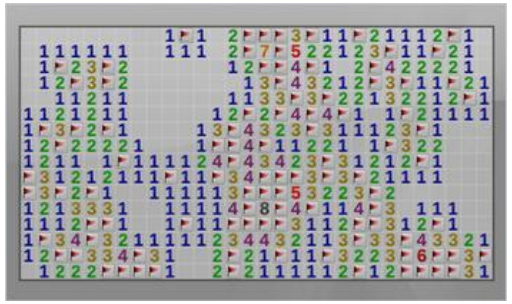
RAM is equivalent to One-Tape TM

RAM ≡ Two-Tape-TM ≡ One-Tape-TM

≡ (Ideal) Python ≡ { other program langs}

≡ Conway's Game of Life, Minecraft, (infinite) Minesweeper, (infinite) Magic: The Gathering





All are Turing Complete computation models!

Computability

Definition (Definition 7.2):

For any Boolean function $F: \{0,1\}^* \rightarrow \{0,1\}^*$ and any Turing Machine M, we say M ***computes*** F iff for all $x \in \{0,1\}^*$, $M(x) = F(x)$.

We say F is ***computable*** if and only if there exists a Turing machine M such that computes F.

Rice's Theorem (9.15)

Definition 9.14:

Two machines M_1, M_2 are **functionally equivalent** (denoted $M_1 \equiv M_2$**)**
if $\forall x \in \{0,1\}^*, M_1(x) = M_2(x)$

A function $F: \{0,1\}^* \to \{0,1\}$, defined on Turin machines, is **semantic** if for every pair of functionally equivalent TMs (M_1, M_2), $F(M_1) = F(M_2)$.

If F is a sematic property, then either F is **uncomputable**, or F is trivial.

By definition, F is trivial iff for all Turing machine M,

$$F(M) = 0$$
$$F(M) = 1$$

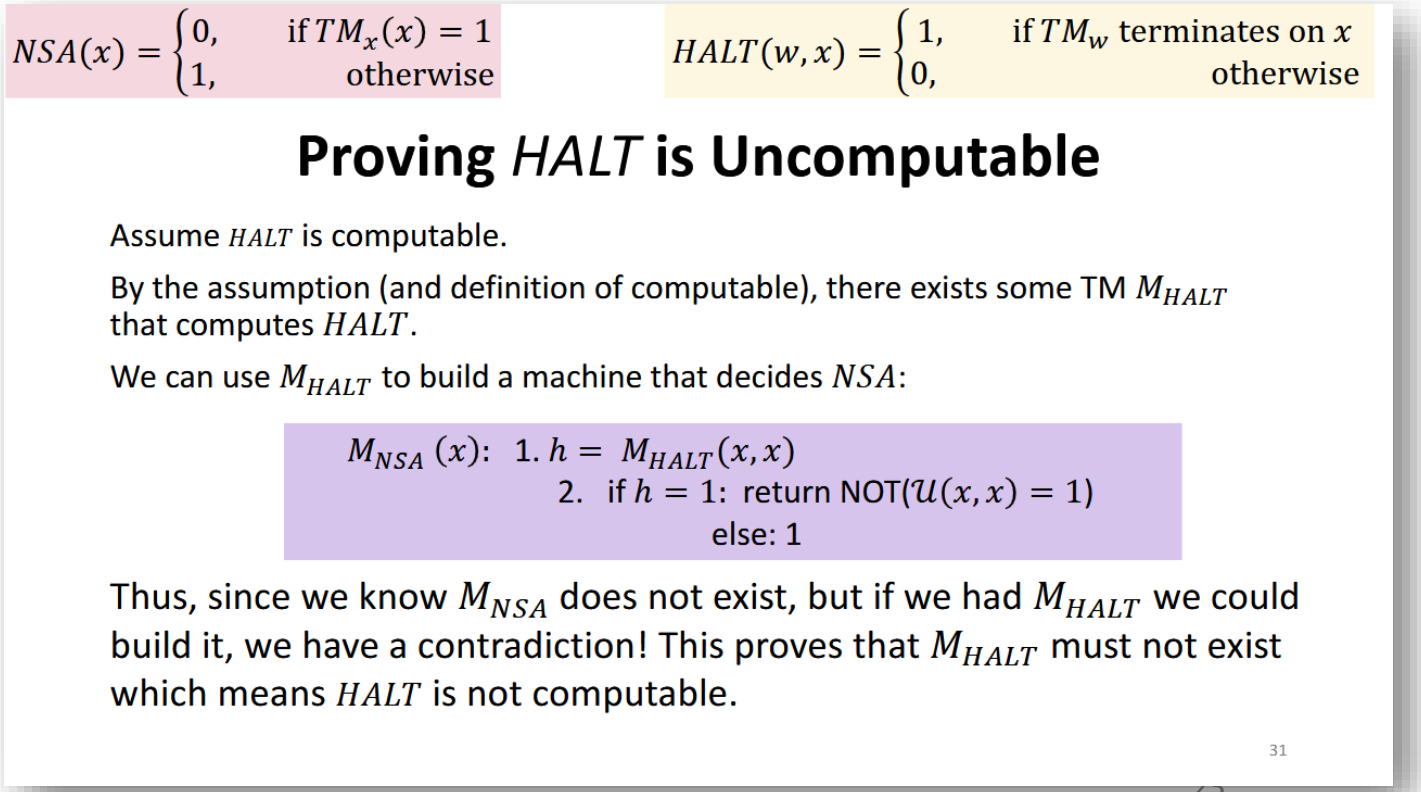
Proof by Reduction

$$NSA(x) = \begin{cases} 0, & \text{if } TM_x(x) = 1 \\ 1, & \text{otherwise} \end{cases}$$

$$HALT(w,x) = \begin{cases} 1, & \text{if } TM_w \text{ terminates on } x \\ 0, & \text{otherwise} \end{cases}$$

Proof: If HALT is computable, then NSA is also computable…

1. **Assume** M_{HALT} computes HALT
2. **Construct** $M_{NSA}(x)$
3. **Prove** that M_{NSA} computes NSA (assuming M_{HALT} computes HALT)



Module 4: Complexity and Classes P, EXP, NP

Time Complexity Classes

Definition (13.1):
$TIME_{\text{TM}}(T(n))$ is the set of functions for which a Turing Machine M exists such that M halts after at most $T(n)$ steps for all n-bit input and M computes the function.

P = $\bigcup_{c \in \mathbb{N}} TIME_{TM}(n^c)$

EXP = $\bigcup_{c \in \mathbb{N}} TIME_{TM}(2^{n^c})$

P and **EXP** are robust to machine definitions (eg, Python or pseudocode algorithm)

Compare Problems: Polynomial-Time Reduction

Definition 14.1 (Polynomial-time reductions)

Let $F, G : \{0,1\}^* \to \{0,1\}$. We say that F *reduces to* G, denoted by $F \leq_p G$ if there is a polynomial-time computable $R : \{0,1\}^* \to \{0,1\}^*$ such that for every $x \in \{0,1\}^*$,

$$F(x) = G(R(x)) \ . \quad (14.1)$$

We say that F and G have *equivalent complexity* if $F \leq_p G$ and $G \leq_p F$.

Maybe the most important technique / concept in this course!

The Class P

Functions that can be computed in polynomial time by a standard Turing Machine.

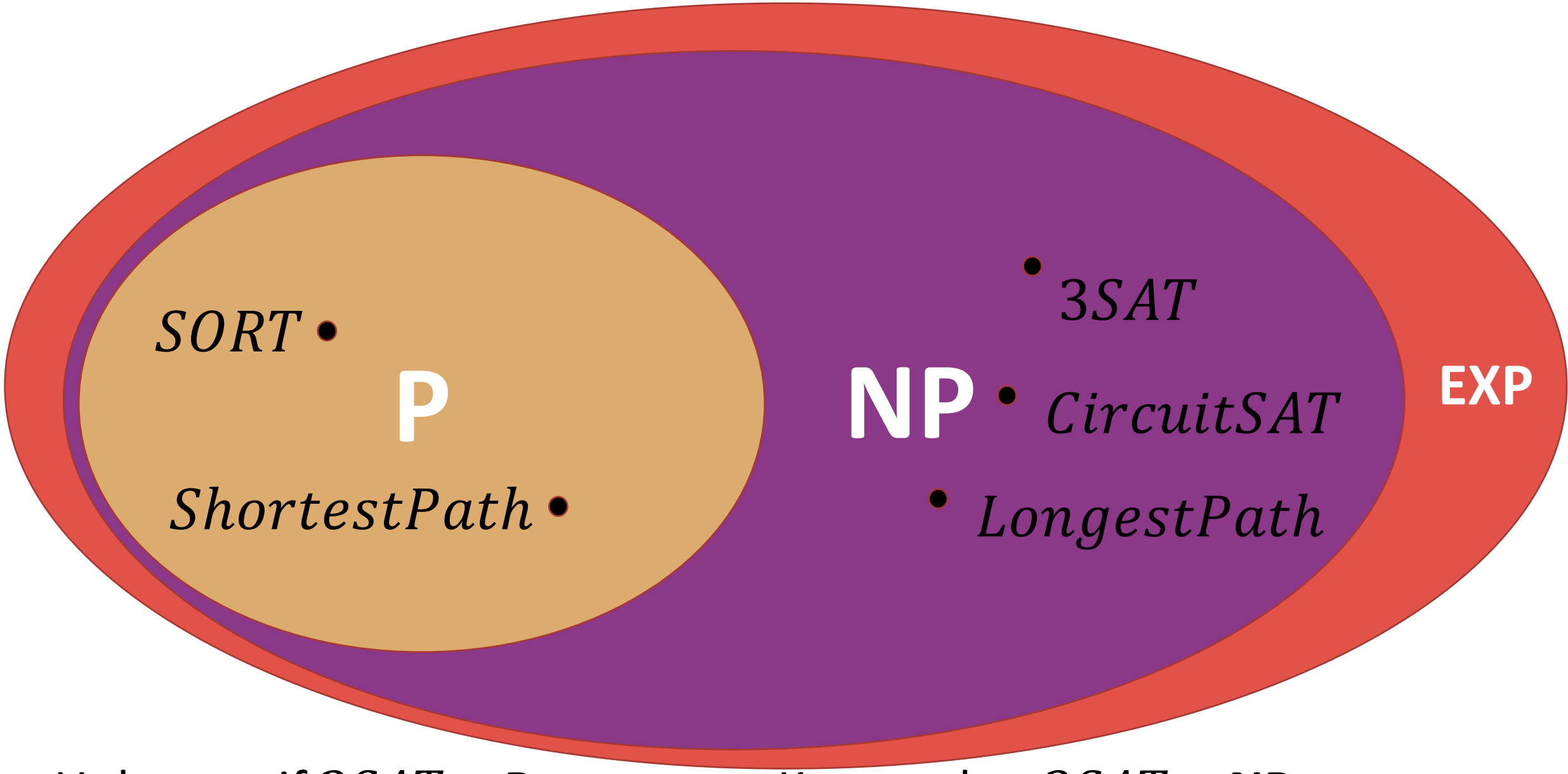
$$\bigcup_{c \in \mathbb{N}} TIME_{TM}(n^c)$$

The Class NP

Functions that can be **verified** in polynomial time by a standard Turing Machine.

Correctness of a **1** output can be *verified* in polynomial time given a witness.

A function $F: \{0,1\}^* \to \{0,1\}$ is in **NP** if there exists some $a \in \mathbb{N}^+$ and $V: \{0,1\}^* \to \{0,1\}$ such that $V \in \mathbf{P}$ and $\forall x \in \{0,1\}^n, F(x) = 1 \Leftrightarrow \exists w \in \{0,1\}^{n^a}$ such that $V(x,w) = 1$.



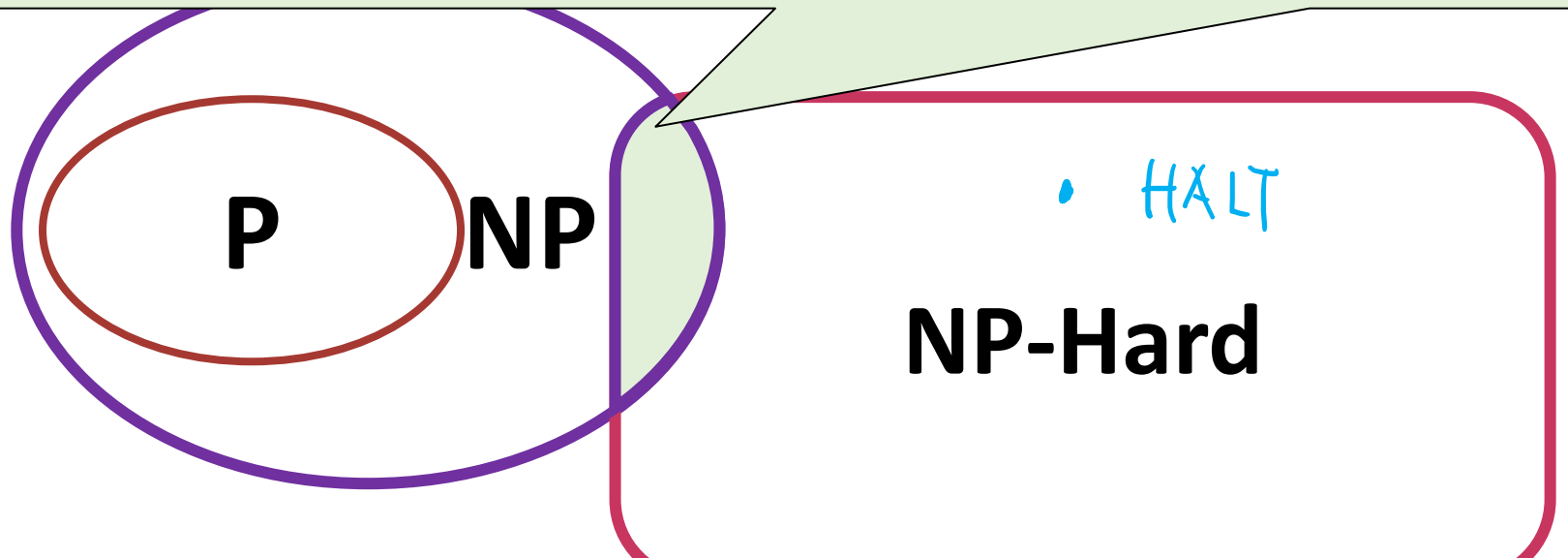
Unknown if $3SAT \in$ **P**

Known that $3SAT \in$ **NP**

NP-Hard and NP-Complete

Definition: A Boolean function G is **NP-Hard** if every $F \in \mathbf{NP}$ can be reduced to G: $F \leq_p G$.

Definition: A function G is **NP-Complete** if $G \in \mathbf{NP}$ and G is **NP-Hard**.



Cook-Levin Theorem

(Theorem 15.6 in the TCS Book):

For every $F \in \mathbf{NP}$, $F \leq_p$ **3SAT.**

Equivalently: **3SAT** is (in) **NP-Hard**

By reductions from 3SAT, other problems are also proven NP-Complete, eg, LongestPath, 3-Coloring, Independent Set, and many optimization, including games.

Gödel’s Incompleteness Theorem

What is a proof?

"Take any definite unsolved problem, such as …. However unapproachable these problems may seem to us and however helpless we stand before them, we have, nevertheless, the firm conviction that their solution must follow by a finite number of purely logical processes…", David Hilbert, 1900.

Mathematical statements

A mathematical statement is simply a piece of text, binary string $x \in \{0,1\}^*$.

1+1 = 2

1+2 = 4

1+2+ … + n = n(n+1)/2

+=1

"The number $2,696,635,869,504,783,333,238,805,675,613,\ 588,278,597,832,162,617,892,474,670,798,113$ *is prime"*.

The following Python function halts on every positive integer `n`

```
def f(n):
    if n==1: return 1
    return f(3*n+1) if n % 2 else f(n//2)
```

Big idea: A *proof* is just a string of text whose meaning is given by a *Verification Algorithm*.

Definition 11.2 (Proof systems)

Let $\mathcal{T} \subseteq \{0,1\}^*$ be some set (which we consider the "true" statements). A *proof system* for $\mathcal{T}$ is an algorithm V that satisfies:

1. *(Effectiveness)* For every $x, w \in \{0,1\}^*$, $V(x,w)$ halts with an output of either 0 or 1.
2. *(Soundness)* For every $x \notin \mathcal{T}$ and $w \in \{0,1\}^*$, $V(x,w) = 0$.

Trivial: V(x,w) = 0 for all x,w (not useful).

E.g., $T = \{\text{"1+1=2"}\}$, or $T = \{\}$

False statement: No sound proof w
True statement: want a proof w

Definition 11.2 (Proof systems)

Let $\mathcal{T} \subseteq \{0,1\}^*$ be some set (which we consider the "true" statements). A *proof system* for $\mathcal{T}$ is an algorithm V that satisfies:

1. *(Effectiveness)* For every $x, w \in \{0,1\}^*$, $V(x,w)$ halts with an output of either 0 or 1.
2. *(Soundness)* For every $x \notin \mathcal{T}$ and $w \in \{0,1\}^*$, $V(x,w) = 0$.

A true statement $x \in \mathcal{T}$ is *unprovable* (with respect to V) if for every $w \in \{0,1\}^*$, $V(x,w) = 0$. We say that V is *complete* if there does not exist a true statement x that is unprovable with respect to V.

$x \in T$ is *provable* if exists w such that $V(x,w) = 1$.
V is *complete* if for all $x \in T$, x is provable.

Gödel's Incompleteness Theorem

Theorem 11.3 (Gödel's Incompleteness Theorem: computational variant)

There does not exist a complete proof system for $\mathcal{H}$.

There is a set of true statements H, but:
H is incomplete for all prove system V

$H = \{\text{"M not halt on 0"} \mid \text{Turing Machine } M \text{ not halt on 0}\}$

Some V can prove that $x \in H$ for some x.
But exists $x' \in H$ such that the same V can not prove.

Reduction from Halting to Incompleteness

Algorithm 11.4 Halting from proofs

Input: Turing machine M

Output: 1 M if halts on input 0; 0 otherwise.

for$\{n = 1, 2, 3, \ldots\}$

  for$\{w \in \{0,1\}^n\}$

    if$\{V(\text{"}M \text{ does not halt on 0"}, w) = 1\}$

      return 0

    endif

    Simulate M for n steps on 0.

    if$\{M$ halts$\}$

      return 1

    endif

  endfor

endfor

Case 1, M halts in T steps:
$V(M, w)$ always 0, Algo never return 0.
Algo return 1 when $n = T$.

Case 2, M never halt:
Algo never return 1.
By V is complete, exists w of N bits such that $V(M, w)$ = 1.
When $n = N$, Algo return 0.

Logistics

- **Final Exam**: **Monday, 4 May, 2:00pm - 4:00pm**. (We use only 2 hours.)
- Module 4 (Complexity) roughly 1/3 weight, other Modules roughly same (1/6 each).
- Wei-Kai's office hour is available next Monday, or by appointment.
- Quiz? Previous Exams

https://in.virginia.edu/CourseXperience