NP

No Problem!

Definition 1

- $NP = \{L\colon L \text{ decidable by a polynomial time nondeterministic TM}\}$

- $NP = \bigcup_{k\in\mathbb{N}} NTIME(n^k)$

- $NTIME\big(f(n)\big) =$
  $\{L\colon L \text{ is a language decidable by an } O\big(f(n)\big) \text{ nondeterministic TM}\}$

- $P \subseteq NP$

Running Time for nondeterministic TMs

- is the maximum number of steps the TM uses on any branch of its computation

NP Definition 2

- Note that the definition is based on nondeterministic TMs
- $NP = \{L: L \text{ has a polynomial time verifier}\}$
- What is a verifier?

Verifiers

- Given a language L, a TM V is called a verifier for L if
$$L = \{s\text{: for some string } c, V \text{ accepts } (s, c)\}$$

- When we say polynomial time verifier, we mean in the length of s alone (this implicitly requires that the size of c is poly in s)

- c is called a certificate or witness (extra information)

- Suppose a language *L* is verifiable by the machine *V*, then

- If $x \in L$, then $\exists y\ V(x, y)$ accepts (there is a proof y that x is in L)
- If $x \notin L$, then $\forall y\ V(x, y)$ does not accept

Intuition (Subset Sum problem)

- $L = \{S \subseteq \mathbb{Z} :$
  $S \text{ has a nonempty subset whose } elements\ add\ up\ to\ 0\}$

- Input: A finite set of integers
- Output: Yes/No

- Yes, if A has a nonempty subset of numbers that add up to 0
- No, otherwise

Deciding L

- Given a finite set of integers $T = \{-7, -3, -2, 5, 8\}$, say.

- A computer goes through all nonempty subsets of T, and adds up their elements

- Going through the subsets takes exponential time in the size of the set

- Adding the elements in one of them takes polynomial time

Relation to nondeterministic TM

- Finding a subset could happen in polynomial time by luck (nondeteministic choice)

- On the previous page we described a deterministic way

Verifying for L

- Membership in L can be verified within polynomial time (a number of steps that follows a polynomial function in the size of the input set)

- For $T = \{-7, -3, -2, 5, 8\}$ given before, there is $c = \{-3, -2, 5\}$

c is a subset of T, and the elements of c add up to 0

- c witnesses/proves/verifies that T is in L

What is the verifier V in the previous example?

- V takes as input two finite sets of integers (or two inputs, each is a finite set of integers)

1. V checks if the second input set is a subset of the first
2. V adds the elements in the second input and checks if the sum is 0

- 1 happens in polytime in the size of the first input
- 2 happens in polytime in the size of the first input?

A Famous Problem (factorization)

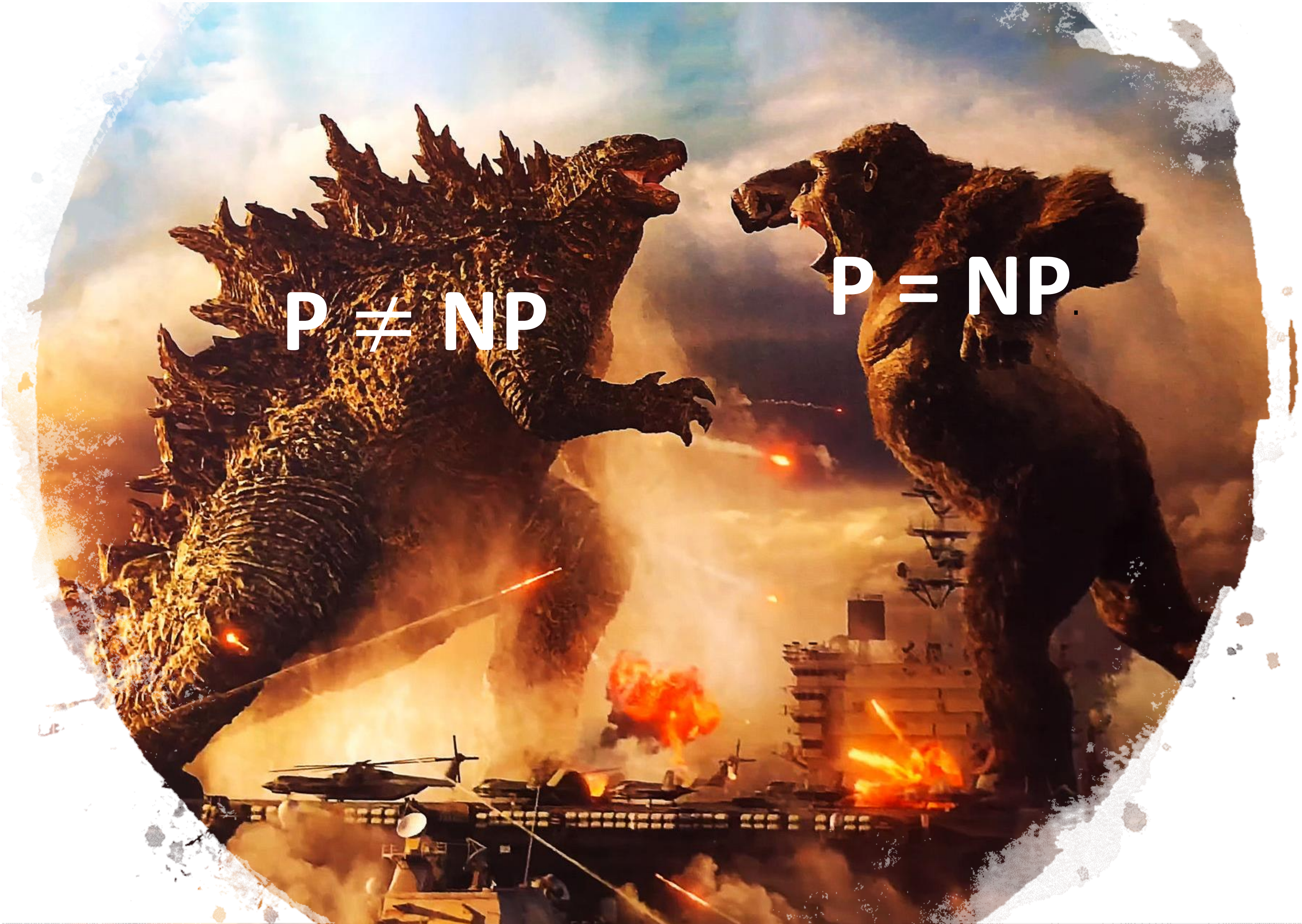
- Find the prime factors of a natural number (large one)
- This requires trying many pairs of numbers
- However, given a factorization, it can be verified just by multiplication
- Note that this is different problem from deciding if a number is prime or not

More intuition

- Given an equation
- Find a solution (NP)
- Or given a solution and check it works (P)

More and more

- Given a theorem
- Prove it (NP)
- OR given a proof and check it works (P)


P ≠ NP
P = NP

P = NP ?

- If the solution to a problem can be verified in polynomial time, can it be found in polynomial time?

- At least it gives hope, the hope that there is an efficient solution

NP-completeness

- A language *L* is NP-complete if (two things):

1. *L* is in the class NP
2. Every language *L'* in NP is p-reducible to *L*

- NP-complete sets are the hardest

If to prove P=NP

- This requires a proof that some NP-complete language is in P
- In other words, take a problem which is known to be NP-complete, then show that there is a polynomial time solution for it
- The majority of NP problems which seem to require exponential time are NP-complete

Wisdom from Sipser

- Sipser points out that some algorithms for NP-complete problems exhibit exponential complexity only in the worst-case scenario and that, in the average case, they can be more efficient than polynomial-time algorithms (even more than polytime)

- Instead of spending all of your time looking for a fast algorithm, you can spend half your time looking for a fast algorithm and the other half of your time looking for a proof of NP-completeness.

- On the practical side, the phenomenon of NP-completeness may prevent wasting time searching for a nonexistent polynomial time algorithm

SAT (The Booelan satisfiability problem)

- Given a *Boolean formula*, find an *assignment* that satisfies it

- Example of a Boolean formula: $(\neg P \& Q)\ \mathrm{OR}\ (P \& \neg Z)$

- Example of a satisfying assignment (solution):
$$P = FALSE, Q = TRUE, Z = FALSE$$

- Sipser uses small letter for the variables, and 1,0 for True, False

SAT

- $SAT = \{\varphi : \varphi \text{ is a satisfiable Boolean formula}\}$
- This is the first known NP-complete problem (language)
- Proved by Stephen Cook here at U of T in 1971
- Independently proved by Leonid Levin
- Cook-Levin theorem: SAT is NP-complete

Algorithms for SAT

- Only algorithms with exponential worst-case scenario

Remarks

- We use only the connectives $\neg, \&, \mathrm{OR}$, they are more than enough to express all logical formulas without quantifiers
- In fact, $\neg, \&$ (or $\neg, \mathrm{OR}$) are enough, but having $\&, \mathrm{OR}$ together makes life easier and easily mimicked by electrical circuits
- Boolean formulas can take many shapes, but any Boolean formula is equivalent to a CNF (conjunctive normal form)

CNF

- $(x_1 \vee \neg x_2 \vee \neg x_3 \vee x_4) \wedge (x_3 \vee \neg x_5 \vee x_6) \wedge (\neg x_6)$

3CNF

- Every clause has three literals
- Example: $(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_3 \vee \neg x_5 \vee x_6)$
- Every Boolean formulas is *equisatisfiable* to a 3CNF one

i.e., given a CNF formula, we can transform it to a 3CNF formulas such that the first formula is satisfiable iff the second is satisfiable.

3SAT

- $3SAT \;= \{\varphi\colon \varphi \text{ is a satisfiable 3CNF formula}\}$

- 3SAT is also NP-complete

- The proof is a modification of the proof for SAT

Subset Sum (general)

- Inputs: an integer value (target) t, and a set of integers $a_1, \dots, a_n$
- Output: **YES** if there is a subset that adds up to *t*, **NO** otherwise

- *SUBSET-SUM* $= \{\langle S, t\rangle : S = \{x_1, \dots, x_n\}, \text{and for some } \{y_1, \dots, y_l\} \subseteq \{x_1, \dots, x_n\}, \sum_{i=1}^{l} y_i = t\}$

- *SUBSET-SUM* is NP-complete

SUBSET-SUM is in NP

- Proof: See Sipser (easy)

SUBSET-SUM is NP-complete

- We need to prove that all languages in NP are polynomial time reducible to *SUBSET-SUM*

- How? We bring a language which we know is NP-complete, and show that it is p-reducible to *SUBSET-SUM*

- Indeed, a possible proof shows that $3SAT \leq_p$*SUBSET-SUM*