

# Computational Models Lecture 11, Spring 2011

- Deterministic Time Classes
- NonDeterministic Time Classes
- Relationship between Deterministic and Nondeterministic Time
- The classes **P** and **NP**
- Examples of Problems in **P** and in **NP**
- Verifiability
- The class **coNP**
  
- Sipser, Chapter 7

# Time Classes Definition, Again

Let

$$t : \mathcal{N} \longrightarrow \mathcal{N}$$

be a function.

**Definition:**

$$\mathbf{DTIME}(t(n)) = \{L \mid L \text{ is a language, decided by an } O(t(n))\text{-time TM}\}$$

# Relations among Time Classes

Let  $t_1, t_2 : \mathcal{N} \rightarrow \mathcal{N}$  be two functions.

- Claim: If  $t_1(n) = O(t_2(n))$  then

$$\text{DTIME}(t_1(n)) \subseteq \text{DTIME}(t_2(n)) .$$

- Stated informally, more time does **not hurt**.
- But does it actually **help**?
- Claim: If  $t_1(n) = O(t_2(n)/\log(n))$  then

$$\text{DTIME}(t_1(n)) \subsetneq \text{DTIME}(t_2(n)) .$$

- Informally, **sufficiently more** time **does help**.
- Proofs – sophisticated diagonalizations (omitted).

# Non-Deterministic Time

Let  $N$  be a non-deterministic TM, and let

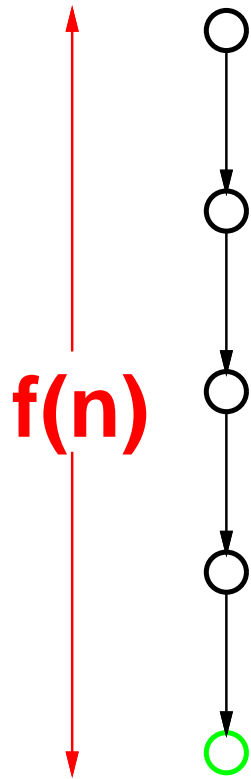
$$f : \mathcal{N} \longrightarrow \mathcal{N}$$

We say that  $N$  runs in time  $f(n)$  if

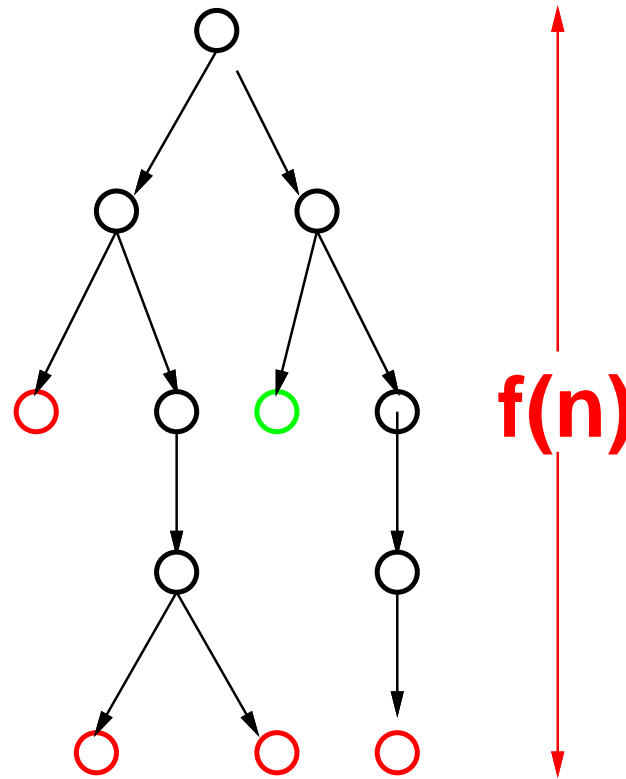
- For **every** input  $x$  of length  $n$ ,
- the **maximum** number of steps that  $N$  uses,
- on **any branch** of its computation tree on  $x$ ,
- is **at most**  $f(n)$ .

# Deterministic vs. Non-Deterministic

**deterministic**



**nondeterministic**



Notice that **non-accepting** branches must **reject** within  $f(n)$  many steps.

# Models and Complexity

**Claim:** Suppose  $N$  is a **nondeterministic** TM that runs in time  $t(n)$  and decides the language  $L$ .

Then there is an  $2^{O(t(n))}$ -time **deterministic** TM,  $D$ , that decided  $L$ .

Note contrast with multi-tape result.

# Simulation

Let  $N$  be a non-deterministic TM running in  $t(n)$  time. Want to describe the deterministic TM,  $D$ , simulating  $N$ .

Basic idea of simulation:

- $D$  tries all possible branches.
- If  $D$  finds any accepting state, it accepts.
- If all branches reject,  $D$  rejects.
- Notice  $N$  has **no looping branches**, so exactly one of two possibilities must occur.

# Simulation Details

$N$ 's computation is a tree:

- root is starting configuration,
- each node has bounded fanout  $\leq b$  (why?),
- each branch has length  $\leq t(n)$ ,
- total number of **leaves** at most  $b^{t(n)}$ ,
- total number of **nodes** in tree  $O(b^{t(n)})$ ,
- time to arrive from root to any node is  $O(t(n))$ .
- $\implies$  Time to visit **all nodes** is

$$O\left(t(n) \times b^{t(n)}\right) = O\left(2^{O(t(n))}\right).$$

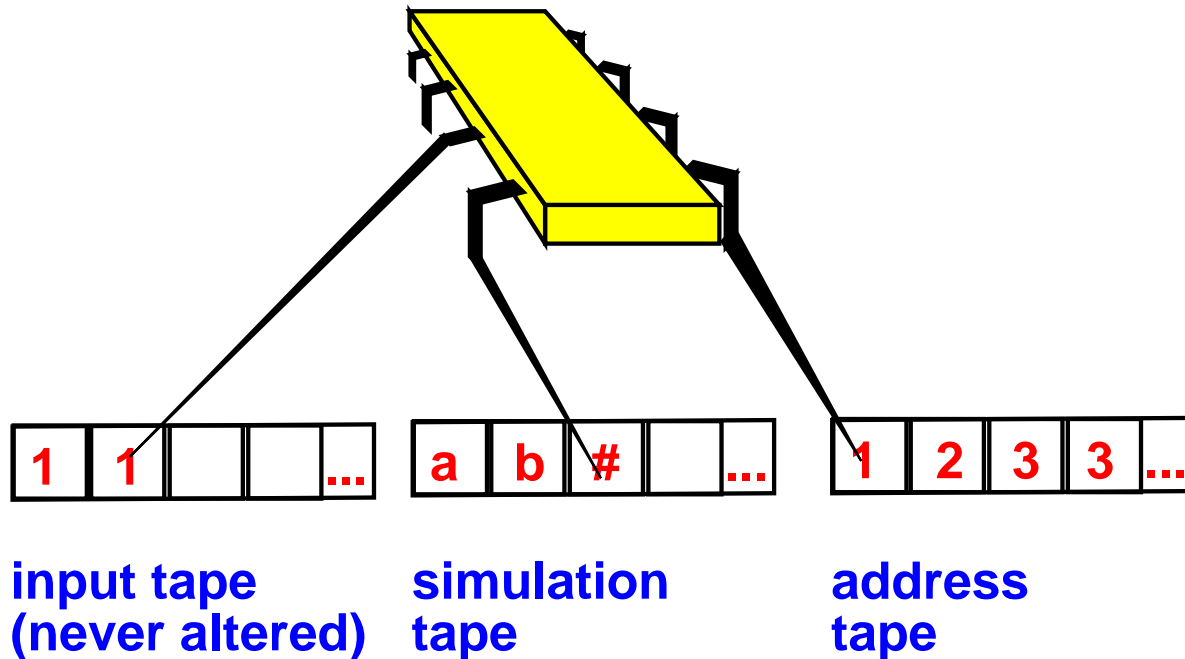


# Remark

## Breadth-first search used in simulation

- Inefficiently traverses from root to visit each node.
- Can be improved upon by using depth-first search (**why is it OK now?**) or other tree search strategies.
- Still, doing this may save constants, but nothing substantial (**why?**)

# Remark



- Simulation uses three-tape machine.
- Single-tape simulation:  $(2^{O(t(n))})^2 = 2^{O(2t(n))} = 2^{O(t(n))}$ .

# Important Distinction

- At most **polynomial** gap in time to perform tasks between different deterministic models (single- vs. multi-tape TMs, TM vs. **RAM**, etc.)
- compared to
- **Apparently exponential** gap in time to perform tasks between **deterministic** and **non-deterministic** models.

# The Good, the Bad, and the Ugly

Complexity differences:  
Polynomial is small; Exponential is large

	10	20	30	40	50	60
$n$	.00001 second	.00002 second	.00003 second	.00004 second	.00005 second	.00006 second
$n^2$	.00001 second	.00004 second	.00009 second	.00016 second	.00025 second	.00036 second
$n^3$	.00001 second	.00008 second	.027 second	.064 second	.125 second	.216 second
$n^5$	.1 second	3.2 seconds	24.3 seconds	1.7 minute	5.2 minutes	13.0 minutes
$2^n$	.001 second	1.0 second	17.9 minutes	12.7 days	35.7 years	366 centuries
$3^n$	.059 second	58 minutes	6.5 years	3855 centuries	$2 \cdot 10^8$ centuries	$1.3 \cdot 10^{13}$ centuries

# Polynomial is Good, Exponential is Bad

**Claim:** All “reasonable” models of computation are polynomially equivalent.

Any one can simulate another with only **polynomial increase** in running time.

**Question:** Is a given problem solvable in

- **Linear** time? **model-specific**.
- **Polynomial** time? **model-independent**.
- We are interested in computation, not in models *per se*!

# The Class $P$

**Definition:**  $P$  is the set of languages decidable in polynomial time on deterministic TMs.

$$P = \bigcup_{c \geq 0} \text{DTIME}(n^c)$$

- The class  $P$  is important because:
- Invariant for all TMs with any number of tapes.
- Invariant for all models of computation polynomially equivalent to TMs.
- Roughly corresponds to **realistically solvable** (tractable) problems.

# The Class $P$

- Invariant for all models of computation polynomially equivalent to deterministic TMs
  - not affected by particulars of model ...
  - go ahead, have another tape, they're pretty small and inexpensive ...

# The Class $P$

- Roughly corresponds to realistically solvable (tractable) problems.
  - actually depends on context
  - going from exponential to polynomial algorithm usually requires major insight,
  - if you find an inefficient polynomial algorithm, you can often find a more efficient one.



# Examples: Problems in $P$

- **Integer arithmetic:** Addition, subtraction, multiplication, division with remainder.
- **Modular arithmetic:** Exponentiation (RSA), inverse.
- **Integer Algorithms:** Greatest common divisor (gcd).
- **Operations research:** Maximum network flow, linear programming,
- **Algebra:** Matrix multiplication, computing determinants, matrix inversion, solving systems of linear equations, factoring polynomials.
- **Graph algorithms:** BFS and DFS in graphs, minimum spanning trees, finding Eulerian path.

# Typical analysis

- break algorithm into stages
- check that each stage is polynomial
- check that number of stages is polynomial
- Ergo, the algorithm is polynomial.

# Encoding

## For numbers

- binary is good
- unary is **not** realistic (exponentially longer)

## For graphs

- list of nodes and edges (good)
- adjacency matrix (good)

# Path

Given

- directed graph  $G$
- nodes  $s$  and  $t$
- is there a path from  $s$  to  $t$ ?

$$\text{PATH} = \{ \langle G, s, t \rangle \mid G \text{ has directed path from } s \text{ to } t \}$$

# Complexity of PATH

Theorem:

$$\text{PATH} \in P$$

When in doubt, try brute force :-)

- let  $m$  be the number of nodes in  $G$
- any path from  $s$  to  $t$  need not repeat nodes
- examine each path in  $G$  of length  $\leq m$ ,
- check if it goes from  $s$  to  $t$ .

Question: What is the complexity of this algorithm?

# Complexity of PATH

- let  $m$  be the number of nodes in  $G$
  - any path from  $s$  to  $t$  need not repeat nodes
  - examine each path in  $G$  of length  $\leq m$ ,
  - check if it goes from  $s$  to  $t$ .
- 
- there are  $m^m$  possible paths
  - exponential in number of nodes
  - exponential in input size
  - Oh, oh. Does not sound like  $P$  to me ...

# Complexity of PATH

Theorem:

$$\text{PATH} \in P$$

1. Place mark on  $s$
2. Repeat until no additional nodes marked:
  - scan edges of  $G$ .
  - If edge  $(a, b)$  found from marked node  $a$  to unmarked node  $b$ ,
  - then mark node  $b$ .
3. If  $t$  marked, **accept**, otherwise **reject**.

**Question:** What is the complexity of this algorithm?

# Complexity of PATH

1. Place mark on  $s$
2. Repeat until no additional nodes marked:
  - scan edges of  $G$ .
  - If edge  $(a, b)$  found from marked  $a$  to unmarked  $b$ ,
  - then mark  $b$ .
3. If  $t$  marked, **accept**, otherwise **reject**.

How many stages?

- Stages 1 and 3 run once.
- Stage 2 runs at most  $m$  times, because each time (except last) it marks at least one new node.

Total number of stages is polynomial.



# Path

1. Place mark on  $s$
2. Repeat until no additional nodes marked:
  - scan edges of  $G$ .
  - If edge  $(a, b)$  found from marked  $a$  to unmarked  $b$ ,
  - then mark  $b$ .
3. If  $t$  marked, **accept**, otherwise **reject**.

How much is each stage?

- Stages 1 and 3 polynomial.
- Stage 2 scans and marks nodes in graph, also polynomial.

Total time complexity is polynomial.



# Relative Primality

Two numbers are **relatively prime** if 1 is the largest integer that evenly divides them both.

- 10 and 21 are relatively prime
- 10 and 22 are not.

**Definition:**

$$\text{RELPRIME} = \{ \langle x, y \rangle \mid x \text{ and } y \text{ are relatively prime} \}$$

# Relative Primality

$\text{RELPRIME} = \{ \langle x, y \rangle \mid x \text{ and } y \text{ are relatively prime} \}$

**First Idea:** Search through all possible divisors of  $x, y$  and test divisibility.

If  $x, y$  in unary:

- size of  $\langle x \rangle$  is  $x$
- testing all potential divisors of  $x, y$  is polynomial

If  $x, y$  in binary:

- size of  $\langle x \rangle$  is  $\log x$
- testing all potential divisors of  $x, y$  is exponential

**Important Notation:** Such algorithm is called **pseudo-polynomial**.

# GCD

$\text{RELPRIME} = \{ \langle x, y \rangle \mid x \text{ and } y \text{ are relatively prime} \}$

Euclid's greatest common divisor algorithm,  $E$ :

On input  $\langle x, y \rangle$

1. Repeat until  $y = 0$ 
  - $x \leftarrow x \bmod y$
  - exchange  $x$  and  $y$
2. output  $x$

$R$  for  $\text{RELPRIME}$  : on input  $\langle x, y \rangle$

- Run  $E$  on  $\langle x, y \rangle$
- if the result is 1, **accept**, otherwise **reject**.

# Euclid's Algorithm

Enough to check that Euclid's Algorithm is polynomial.

- each execution of Stage 1 cuts  $x$  by **at least half**  
(case analysis with for  $y < x/2$  and  $x/2 \leq y < x$ )
- after each loop  $x < y$
- values swapped
- number of stages is  $\min(\log_2 x, \log_2 y)$

total running time is polynomial.

Correctness holds (but should be verified).

Consequently, **RELPRIME**  $\in P$ .



# NTime Classes Definition

Let

$$f : \mathcal{N} \longrightarrow \mathcal{N}$$

be a function.

**Definition:**

$$\mathbf{NTIME}(f(n)) = \{L \mid L \text{ is a language, decided by an } O(f(n))\text{-time } \mathbf{NTM}\}$$

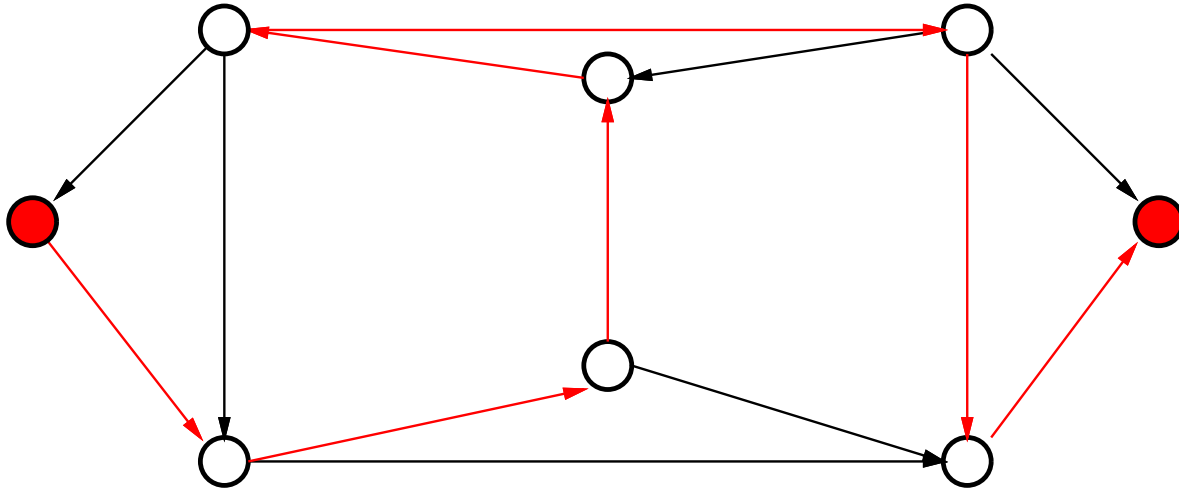
# The Class $NP$

**Definition:**  $NP$  is the set of languages decidable in polynomial time on **non**-deterministic TMs.

$$NP = \bigcup_{c \geq 0} \text{NTIME}(n^c)$$

- The class  $NP$  is important because:
- Invariant for all TMs with any number of tapes.
- $NP$  is insensitive to choice of reasonable non-deterministic computational model.
- Roughly corresponds to problems whose **positive solutions** cannot be efficiently **generated** ( $\Rightarrow$  intractable), but can be efficiently **verified**.

# Hamiltonian Path



A **Hamiltonian path** in a directed  $\mathcal{G}$  visits each **node** exactly once.



# Hamiltonian Path

**HAMPATH** =  $\{\langle G, s, t \rangle \mid G \text{ has Hamiltonian path from } s \text{ to } t\}$

**Question:** How hard is it to decide this language?

# Hamiltonian Path

**HAMPATH** =  $\{\langle G, s, t \rangle \mid G \text{ has Hamiltonian path from } s \text{ to } t\}$

Easy to obtain **exponential time** algorithm:

- **generate** each potential path
- **check** whether it is Hamiltonian

# The Class $\mathcal{NP}$

Here is an **NTM** that decides **HAMPATH** in poly time.

On input  $\langle G, s, t \rangle$ ,

1. **Guess** and write down a list of numbers  $p_1, \dots, p_m$ , where  $m$  is number of nodes in  $G$ , and  $1 \leq p_i \leq m$ .
2. Check for repetitions in list. If any found, **reject**.
3. Check whether  $p_1 = s$  and  $p_m = t$ . If either does not hold, **reject**.
4. For  $i, 1 \leq i \leq m - 1$ , check whether  $(p_i, p_{i+1})$  is an **edge** in  $G$ . If any is not, **reject**. Otherwise **accept**.

# NP

On input  $\langle G, s, t \rangle$ ,

1. **Non-deterministically guess** and write down a list of numbers  $p_1, \dots, p_m$ .
2. Check that there are no repetitions.
3. Check whether  $p_1 = s$  and  $p_m = t$ .
4. Check whether  $(p_i, p_{i+1})$  is an **edge** in  $G$ .
5. If 1 through 4 satisfied, **accept**, else **reject**.

- Stage 1 polynomial time
- Stages 2 and 3 simple checks.
- Stage 4 simple poly-time too. ✓

# Hamiltonian Path

This problem has one very interesting feature: **polynomial verifiability**.

- we don't know a fast way to **find** a Hamiltonian path
- but we can **check** whether a **given path** is Hamiltonian in polynomial time.

In other words,

- **verifying** correctness of a path is much **easier**
- than **determining** whether one exists

# Composite Numbers

A natural number is **prime** if it is divisible by only 1 and itself.

$$\text{COMPOSITES} = \{x \mid x = pq \text{ for integers } p, q > 1\}$$

- We can easily **verify** that a number is composite, given a factor (**how?**)
- What about testing for primes?!

# Prime Numbers

A natural number is **composite** if it is the product of two integers greater than one.

$$PRIMES = \{x \mid x \text{ is a prime} \}$$

- Primality Certificate, Pratt (1975)
- Randomized polynomial time, Rabin-Miller (1976)
- Deterministic Polynomial time, Manindra Agrawal, Neeraj Kayal, Nitin Saxena (2002)

# Verifiability

**Not all** problems are polynomially verifiable.

There is no known way to verify  $\overline{\text{HAMPATH}}$  in polynomial time.

In fact, we will see many examples where  $L$  is polynomially verifiable, but its complement,  $\overline{L}$ , is not known to be polynomially verifiable.



# Verifiability

A **verifier** for a language  $\mathcal{A}$  is an algorithm  $\mathcal{V}$  where

$$\mathcal{A} = \{w \mid \mathcal{V} \text{ accepts } \langle w, c \rangle \text{ for some string } c\}$$

- The verifier uses the additional information  $c$  to verify  $w \in \mathcal{A}$ .
- We measure verifier run time by length of  $w$ .
- The string  $c$  is called a **certificate** (or **proof**) for  $w$  if  $\mathcal{V}$  accepts  $\langle w, c \rangle$ .
- A **polynomial verifier** runs in polynomial time in  $|w|$  (so  $|c| \leq |w|^{O(1)}$ ).
- A language  $\mathcal{A}$  is **polynomially verifiable** if it has a polynomial verifier.

# Examples

For **HAMPATH**, a certificate for

$$\langle G, s, t \rangle \in \text{HAMPATH}$$

is simply the Hamiltonian path from  $s$  to  $t$ .

Can **verify** in time polynomial in  $|\langle G \rangle|$  whether given path is Hamiltonian.

# Examples

For **COMPOSITES**, a certificate for

$$x \in \text{COMPOSITES}$$

is simply one of its divisors.

Can **verify** in time polynomial in  $|x|$  if given divisor indeed divides  $x$ .

# Examples

For **PRIMES**, a certificate for

$$p \in PRIMES$$

- A number  $p$  is prime iff: there exists a  $g \leq p - 1$ :
  - $GCD(p, g) = 1$ .
  - $g^{p-1} \bmod p = 1$ .
  - for any prime factor  $q$  of  $p - 1$  we have  $g^{(p-1)/q} \bmod p \neq 1$ .
- The certificate needs to include certificates to the prime factors  $q$  of  $p$ .
- Size of certificate is  $O(\log p)$ .

# NP and Verifiability

**Theorem:** A language is in NP if and only if it has a polynomial time verifier.

**Proof – Intuition:**

- NTM simulates verifier by guessing the certificate.
- Verifier simulates NTM by using accepting branch as certificate.

# NP

**Claim:** If  $\mathcal{A}$  has a poly-time verifier, then it is decided by some polynomial-time NTM.

Let  $\mathcal{V}$  be poly-time verifier for  $\mathcal{A}$ .

- single-tape TM
- runs in time  $n^k$

$\mathcal{N}$ : on input  $w$  of length  $n$


- Nondeterministically select *string*  $c$  of length  $n^k$ .
- Run  $\mathcal{V}$  on  $\langle w, c \rangle$
- If  $\mathcal{V}$  accepts, *accept*; otherwise *reject*.

# NP

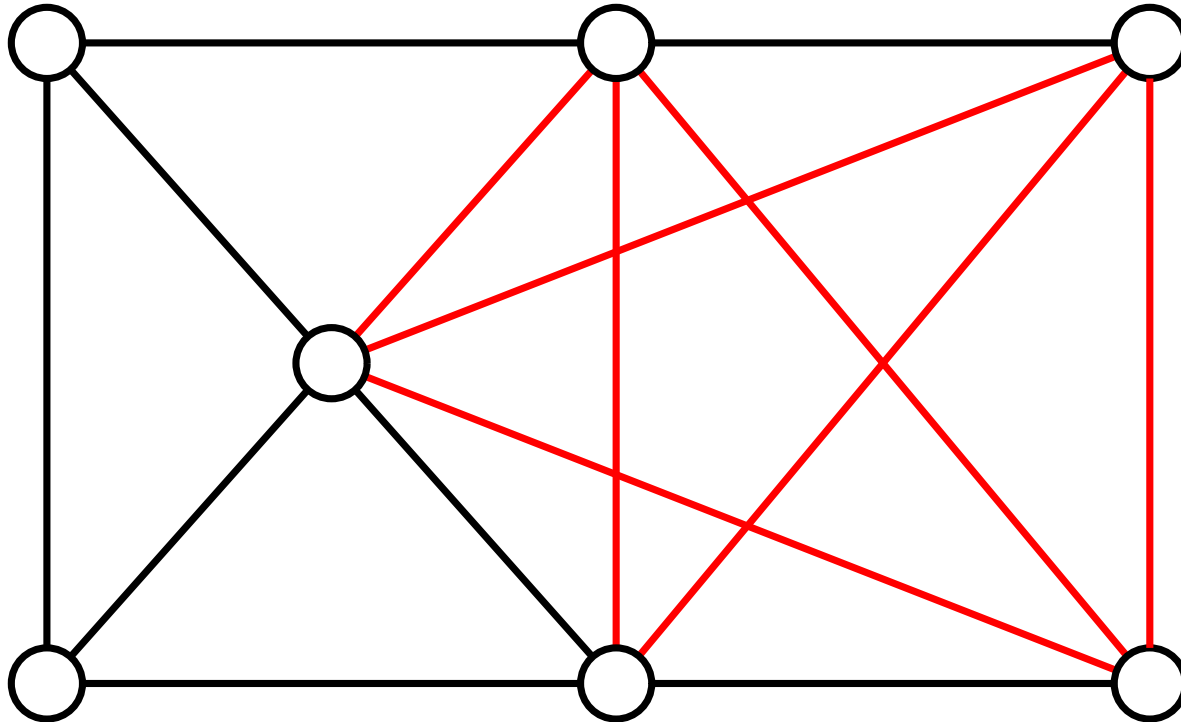
**Claim:** If  $\mathcal{A}$  is decided by a polynomial-time NTM  $\mathcal{N}$ , running in time  $n^k$ , then  $\mathcal{A}$  has a poly-time verifier.

Construct polynomial-time verifier  $\mathcal{V}$  as follows.

$\mathcal{V}$ : on input  $w$  of length  $n$ , and on a string  $c$  of length  $n^k$

- Simulate  $\mathcal{N}$  on input  $w$ , treating each symbol of  $c$  as a description of the non-deterministic choice in each step of  $\mathcal{N}$ .
- If this branch accepts, *accept*, otherwise *reject*. 

## Example: Clique



A **clique** in a graph is a subgraph where every two nodes are connected by an edge.

A  **$k$ -clique** is a clique of size  $k$ .

What is the **largest  $k$ -clique** in the figure? (ignore colors)



# Example: Clique

Define the language

$$\text{CLIQUE} = \{ \langle G, k \rangle \mid G \text{ is an undirected graph} \\ \text{with a } k\text{-clique} \}$$

# Examples: Clique

Theorem:

$\text{CLIQUE} \in NP$

The clique is the certificate.

Here is a verifier  $\mathcal{V}$ : on input  $(\langle G, k \rangle, c)$

- if  $c$  is not a  $k$ -clique, *reject*
- if  $G$  does not contain all vertices of  $c$ , *reject*
- *accept*

# Examples: SUBSET-SUM

An instance of the problem

- A collection of non-negative integers  $x_1, \dots, x_k$
- A target number  $t$
- Question: does some subcollection add up to  $t$ ?

$$\text{SUBSET-SUM} = \{ \langle S, t \rangle \mid S = \{x_1, \dots, x_k\} \}$$

$$\exists \{y_1, \dots, y_\ell\} \subseteq \{x_1, \dots, x_k\}, \sum_{j=1}^{\ell} y_j = t$$

Collections are sets: repetitions not allowed.

# Examples: SUBSET-SUM

We have

$$(\{4, 11, 16, 21, 27\}, 25) \in \text{SUBSET-SUM}$$

because  $4 + 21 = 25$ .

$$(\{4, 11, 16, 21, 27\}, 26) \notin \text{SUBSET-SUM}$$

(why?)

# Examples: SUBSET-SUM

Theorem:

$\text{SUBSET-SUM} \in NP$

The **subset** is the **certificate**.

Here is a verifier:

$\mathcal{V}$ : on input  $(\langle S, t \rangle, c)$

- test whether  $c$  is a collection of numbers summing to  $t$ .
- test whether  $c$  is a subset of  $S$
- if either fail, **reject**, otherwise **accept**.

# Examples: SUBSET-SUM

**Theorem:** There is a Pseudo-polynomial algorithm for SUBSET-SUM

Proof:

- Running time  $O(|S|t)$
- not  $poly(\log_2 t)$  but  $poly(t)$  (i.e., Pseudo-polynomial)
- Show a reduction from SUBSET-SUM to PATH.
- Given an input to SUBSET-SUM of the form  $\langle S, t \rangle$ ,
- create an input to PATH of the form  $\langle G, s, t \rangle$ .
- On the Board.

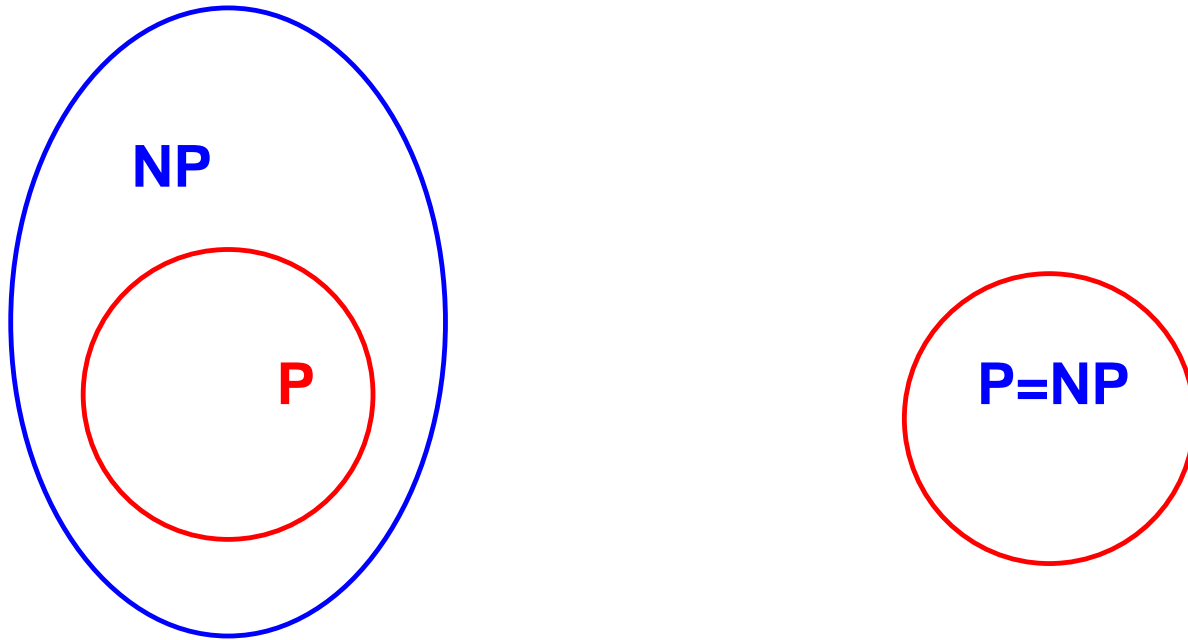
# Complementary Problems

CLIQUE and SUBSET-SUM seem **not** to be members of NP. It is harder to efficiently verify that something does **not** exist than to efficiently verify that something **does** exist..

**Definition:** The class  $coNP$ :  
 $L \in coNP$  if  $\bar{L} \in NP$ .

So far, no one knows if  $coNP$  is distinct from **NP** (recall first slide on complexity).

# P vs. NP



The question  $P = NP?$  is one of the great unsolved mysteries in contemporary mathematics.

- most computer scientists believe the two classes are not equal
- most bogus proofs show them equal (why?)



# Observations

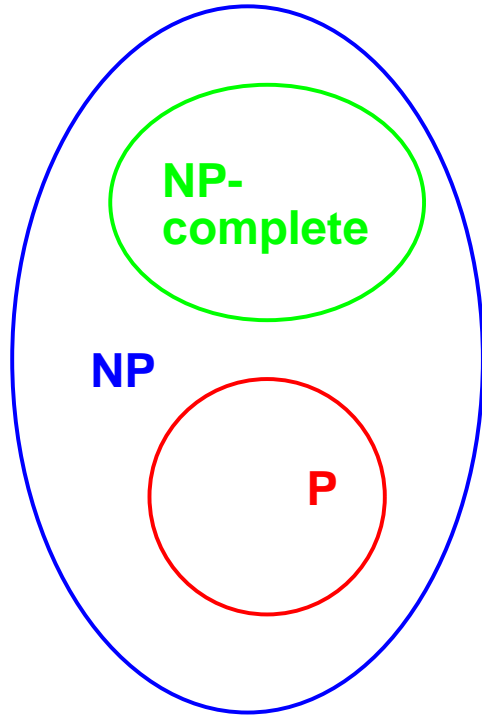
If  $\mathcal{P}$  differs from  $NP$ , then the distinction between  $\mathcal{P}$  and  $NP - \mathcal{P}$  is meaningful and important.

- languages in  $\mathcal{P}$  tractable
- languages in  $NP - \mathcal{P}$  intractable

Until we can prove that  $\mathcal{P} \neq NP$ , there is no hope of proving that a **specific language** lies in  $NP - \mathcal{P}$ .

Nevertheless, we **can** prove statements of the form “If  $\mathcal{P} \neq NP$  then  $A \in NP - \mathcal{P}$ .”

# NP Completeness



The class of **NP-complete** languages are

- “hardest” languages in  $\mathcal{NP}$
- “least likely” to be in  $\mathcal{P}$
- If any NP-complete  $A \in \mathcal{P}$ , then  $\mathcal{NP} = \mathcal{P}$ .

# Cook–Levin (1971-1973)

**Theorem:** There is a language  $S \in NP$  such that  $S \in P$  if and only if  $P = NP$ .

This theorem establishes the class of **NP-complete languages**. Such languages, “carry on their backs” the burden of all of  $NP$ .