FORMAL LANGUAGES, AUTOMATA AND COMPUTATION

COMPLEXITY

COMPLEXITY THEORY

QUESTION

Assume that a problem (language) is decidable. Does that mean we can realistically solve it?

ANSWER

NO, not always. It can require to much of time or memory resources.

Complexity Theory aims to make general conclusions of the resource requirements of decidable problems (languages).

- Henceforth, we only consider decidable languages and deciders.
- Our computational model is a Turing Machine.
  - Time: the number of computation steps a TM machine makes to decide on an input of size n.
  - Space: the maximum number of tape cells a TM machine takes to decide on a input of size n.

TIME COMPLEXITY – MOTIVATION

- How much time (or how many steps) does a single tape TM take to decide $A = \{0^k 1^k \mid k \geq 0\}$?

M = "On input *w*:

1. Scan the tape and *reject* if *w* is not of the form $0^* 1^*$.
2. Repeat if both 0s and 1s remain on the tape.
3. Scan across the tape crossing off one 0 and one 1.
4. If all 0's are crossed and some 1's left, or all 1's crossed and some 0's left, then *reject*; else *accept*.

QUESTION

How many steps does M take on an input w of length n?

ANSWER (WORST-CASE)

The number of steps M takes $\propto n^2$.

TIME COMPLEXITY – SOME NOTIONS

- The number of steps in measured as a function of n - **the size of the string representing the input.**
- In worst-case analysis, we consider the longest running time of all inputs of length n.
- In average-case analysis, we consider the average of the running times of all inputs of length n.

TIME COMPLEXITY

Let M be a deterministic TM that halts on all inputs. The time complexity of M if the function $f : \mathcal{N} \longrightarrow \mathcal{N}$, where $f(n)$ is the maximum number of steps that M uses on any input of length n.

If $f(n)$ is the running time of M we say

- M runs in time $f(n)$
- M is an $f(n)$-time TM.

ASYMPTOTIC ANALYSIS

- We seek to understand the running time when the input is “large”.
- Hence we use an asymptotic notation or big-O notation to characterize the behaviour of $f(n)$ when n is large.
- The exact value running time function is not terribly important.
- What is important is how $f(n)$ grows as a function of n, for large n.
- Differences of a constant factor are not important.

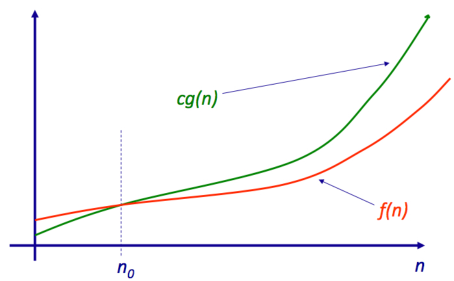
ASYMPTOTIC UPPER BOUND

DEFINITION – ASYMPTOTIC UPPER BOUND

Let $\mathcal{R}^+$ be the set of nonnegative real numbers. Let f and g be functions $f, g : \mathcal{N} \longrightarrow \mathcal{R}^+$. We say $f(n) = O(g(n))$, if there are positive integers c and n_0, such that for every $n \geq n_0$

$$f(n) \leq c\, g(n).$$

$g(n)$ is an asymptotic upper bound.



ASYMPTOTIC UPPER BOUND

- $5n^3 + 2n^2 + 5 = O(n^3)$ (what are c and n_0?)
- $5n^3 + 2n^2 + 5 = O(n^4)$ (what are c and n_0?)
- $\log_2(n^8) = O(\log n)$ (why?)
- $5n^3 + 2n^2 + 5$ is not $O(n^2)$ (why?)

- $2^{O(n)}$ means an upper bound $O(2^{cn})$ for some constant c.
- $n^{O(1)}$ is a polynomial upper bound $O(n^c)$ for some constant c.

REALITY CHECK

Assume that your computer/TM can perform 10^9 steps per second.

$n/f(n)$	n	$n\log(n)$	n^2	n^3	2^n
10	0.01 μsec	0.03 μsec	0.1 μsec	1 μsec	1 μsec
20	0.02 μsec	0.09 μsec	0.4 μsec	8 μsec	1 *msec*
50	0.05 μsec	0.28 μsec	2.5 μsec	125 μsec	13 days
100	0.10 μsec	0.66 μsec	10 μsec	1 *msec*	$\approxeq 4 \times 10^{13}$ years
1000	1 μsec	3 μsec	1 *msec*	1 sec	$\approxeq 3.4x10^{281}$ centuries

Clearly, if the running time of your TM is an exponential function of n, it does not matter how fast the TM is!

SMALL-O NOTATION

DEFINITION – STRICT ASYMPTOTIC UPPER BOUND

Let f and g be functions $f, g : \mathcal{N} \longrightarrow \mathcal{R}^+$. We say $f(n) = o(g(n))$, if

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

- $n^2 = o(n^3)$
- $\sqrt{n} = o(n)$
- $n \log n = o(n^2)$
- $n^{100} = o(2^n)$
- $f(n)$ is never $o(f(n))$.

INTUITION

- $f(n) = O(g(n))$ means "asymptotically $f(n) \leq g(n)$"
- $f(n) = o(g(n))$ means "asymptotically $f(n) < g(n)$"

COMPLEXITY CLASSES

DEFINITION – TIME COMPLEXITY CLASS TIME($t(n)$)

Let $t : \mathcal{N} \longrightarrow \mathcal{R}^+$ be a function.
TIME($t(n)$) $= \{L(M) \mid M$ is a decider running in time $O(t(n))\}$

- TIME($t(n)$) is the class (collection) of languages that are decidable by TMs, running in time $O(t(n))$.
- $\text{TIME}(n) \subset \text{TIME}(n^2) \subset \text{TIME}(n^3) \subset \ldots \subset \text{TIME}(2^n) \subset \ldots$
- Examples:
    - $\{0^k1^k \mid k \geq 0\} \in \text{TIME}(n^2)$
    - $\{0^k1^k \mid k \geq 0\} \in \text{TIME}(n \log n)$ (next slide)
    - $\{w\#w \mid w \in \{0,1\}^*\} \in \text{TIME}(n^2)$

$\{0^k1^k \mid k \geq 0\} \in \mathrm{TIME}(n \log n)$

M = "On input *w*:

1. Scan the tape and *reject* if *w* is not of the form 0^*1^* .
2. Repeat as long as some 0s and some 1s remain on the tape.
    - Scan across the tape, checking whether the total number of 0s and 1s is even or odd. *Reject* if it is odd.
    - Scan across the tape, crossing off every other 0 starting with the first 0, and every other 1, starting with the first 1.
3. If no 0's and no 1's remain on the tape, *accept*. Otherwise, *reject*.

- Steps 2 take $O(n)$ time.
- Step 2 is repeated at most $1 + \log_2 n$ times. (why?)
- Total time is $O(n \log n)$.
- Hence, $\{0^k1^k \mid k \geq 0\} \in \mathrm{TIME}(n \log n)$.
- However, $\{0^k1^k \mid k \geq 0\}$ is decidable on a 2-tape TM in time $O(n)$ (How ?)

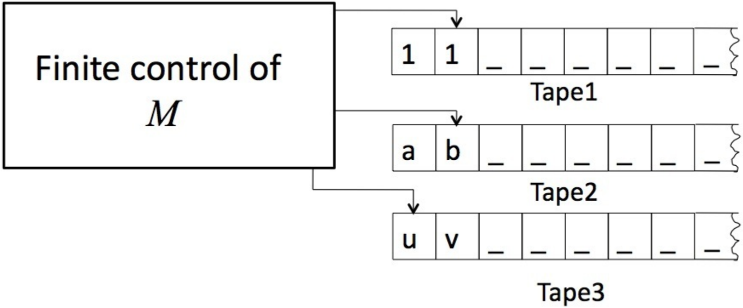
RELATIONSHIP BETWEEN k-TAPE AND SINGLE-TAPE TMS

THEOREM 7.8

Let $t(n)$ be a function and $t(n) \geq n$. Then every multitape TM has an equivalent $O(t^2(n))$ single tape TM.

- Let's remind ourselves on how the simulation operates.

MULTITAPE TURING MACHINES



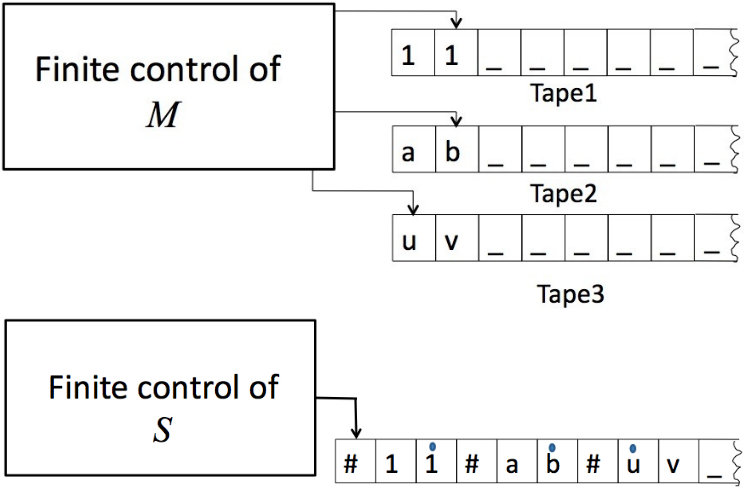
MULTITAPE TURING MACHINES

- A multitape Turing Machine is like an ordinary TM
  - There are k tapes
  - Each tape has its own independent read/write head.
- The only fundamental difference from the ordinary TM is δ – the state transition function.

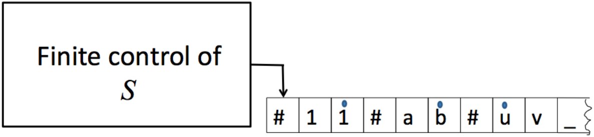
$$\delta : Q \times \Gamma^k \rightarrow Q \times \Gamma^k \times \{L, R\}^k$$

- The δ entry $\delta(q_i, a_1, \ldots, a_k) = (q_j, b_1, \ldots, b_k, L, R, L, ...L)$ reads as :
  - If the TM is in state q_i and
  - the heads are reading symbols a_1 through a_k,
  - Then the machine goes to state q_j, and
  - the heads write symbols b_1 through b_k, and
  - Move in the specified directions.

SIMULATING A MULTITAPE TM WITH AN ORDINARY TM



SIMULATING A MULTITAPE TM WITH AN ORDINARY TM



- We use # as a delimiter to separate out the different tape contents.
- To keep track of the location of heads, we use additional symbols
  - Each symbol in Γ (except ⊔) has a “dotted” version.
  - A dotted symbol indicates that the head is on that symbol.
  - Between any two #’s there is only one symbol that is dotted.
- Thus we have 1 real tape with *k* “virtual’ tapes, and
- 1 real read/write head with *k* “virtual” heads.

SIMULATING A MULTITAPE TM WITH AN ORDINARY TM

- Given input $w = w_1 \cdots w_n$, S puts its tape into the format that represents all k tapes of M

$$\# \ \overset{\bullet}{w_1} \ w_2 \cdots w_n \# \overset{\bullet}{\sqcup} \# \overset{\bullet}{\sqcup} \# \cdots \#$$

- To simulate a single move of M, S starts at the leftmost $\#$ and scans the tape to the rightmost $\#$.
  - It determines the symbols under the “virtual” heads.
  - This is remembered in the finite state control of S. (How many states are needed?)
- S makes a second pass to update the tapes according to M.
- If one of the virtual heads, moves right to a $\#$, the rest of tape to the right is shifted to “open up” space for that “virtual tape”. If it moves left to a $\#$, it just moves right again.

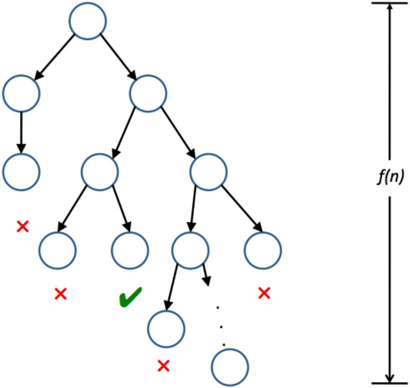
ANALYSIS OF THE MULTI-TAPE TM SIMULATION

- Preparing the single simulation tape takes $O(n)$ time.
- Each step of the simulation makes two passes over the tape:
  - One pass to see where the heads are.
  - One pass to update the heads (possibly with some shifting)
- Each pass takes at most $k \times t(n) = O(t(n))$ steps (why?)
- So each simulation step takes 2 scans + at most k rightward shifts. So the total time per step is $O(t(n))$.
- Simulation takes $O(n) + t(n) \times O(t(n))$ steps $= O(t^2(n))$.
- So, a single-tape TM is only polynomially slower than the multi-tape TM.
- If the multi-tape TM runs in polynomial time, the single-tape TM will also run in polynomial time (where polynomial time is defined as $O(n^m)$ for some m.)

NONDETERMINISTIC TMS

DEFINITION – NONDETERMINISTIC RUNNING TIME

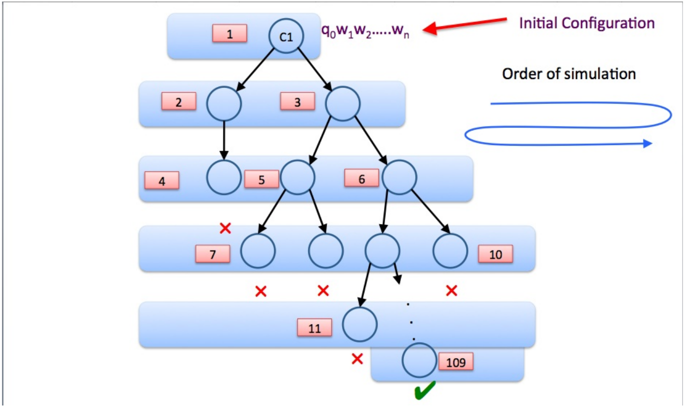
Let N be a nondeterministic TM that is a decider. The running time of N is the function $f : \mathcal{N} \longrightarrow \mathcal{N}$, where $f(n)$ is the maximum number of steps that N uses, on any branch of its computation on any input of length n.



NONDETERMINISTIC TMS

THEOREM 7.11

Let $t(n)$ be a function and $t(n) \geq n$. Then every $t(n)$ time nondeterministic TM has an equivalent $2^{O(t(n))}$ time deterministic single tape TM.



NONDETERMINISTIC TMS

THEOREM 7.11

Let $t(n)$ be a function and $t(n) \geq n$. Then every $t(n)$ time nondeterministic TM has an equivalent $2^{O(t(n))}$ time deterministic single tape TM.

PROOF

- On an input of n, every branch of N's nondeterministic computation has length at most $t(n)$ (why?)
- Every node in the tree can have at most b children where b is the maximum number of nondeterministic choices a state can have.
- So, the computation tree has at most $1 + b^2 + \cdots + b^{t(n)} = O(b^{t(n)})$ nodes.
- The deterministic machine D takes at most $O(b^{t(n)}) = 2^{O(t(n))}$ steps.
- D has 3 tapes. Converting it to a single tape TM at most squares its running time (previous Theorem):$(2^{O(t(n))})^2 = 2^{2O(t(n))} = 2^{O(t(n))}$

THE CLASS P

DEFINITION

P is the class of languages that are decidable in polynomial time on a deterministic single-tape TM.

$$\mathsf{P} = \bigcup_k \mathsf{TIME}(n^k).$$

- The class P is important for two main reasons:
  1. P is robust: The class remains invariant for all models of computation that are polynomially equivalent to deterministic single-tape TMs.
  2. P (roughly) corresponds to the class of problems that are realistically solvable on a computer.
- Even though the exponents can be large (though most useful algorithms have “low” exponents), the class P provides a reasonable definition of practical solvability.

EXAMPLES OF PROBLEMS IN P

- We will give high-level algorithms with numbered stages just as we gave for decidability arguments.
- We analyze such algorithms to show that they run in polynomial time.
    1. We give a polynomial upper bound on the number of stages the algorithm uses when it runs on an input of length n.
    2. We examine each stage, to make sure that each can be implemented in polynomial time on a reasonable deterministic time.
- We assume a “reasonable” encoding of the input.
    - For example, when we represent a graph G, we assume that $\langle G \rangle$ has a size that is poynomial the number of nodes.

EXAMPLES OF PROBLEMS IN P

THEOREM

PATH $= \{\langle G, s, t\rangle \mid G$ is a directed graph with n nodes that has a path from s to $t\} \in$ P.

PROOF

$M =$ "On input $\langle G, s, t\rangle$

1. Place a mark on s.
2. Repeat 3 until no new nodes are marked
3. Scan edges of G. If (a, b) is an edge and a is marked and b is unmarked, mark b.
4. If t is marked, *accept* else *reject*."

- Steps 1 and 4 are executed once
  - Each takes at most $O(n)$ time on a TM.
- Step 3 is executed at most n times
  - Each execution takes at most $O(n^2)$ steps ($\propto$ number of edges)
- Total execution time is thus a polynomial in n.

EXAMPLES OF PROBLEMS IN P

THEOREM

$A_{CFG} \in \mathrm{P}$

PROOF.

The CYK algorithm decides A_{CFG} in polynomial time. □

EXAMPLES OF PROBLEMS IN P

DEFINITION

Natural numbers x and y are relatively prime iff $gcd(x, y) = 1$.

- $gcd(x, y)$ is the greatest natural number that evenly divides both x and y.
- $RELPRIME = \{\langle x, y\rangle \mid x \text{ and } y \text{ are relatively prime numbers}\}$
- Remember that the length of $\langle x, y\rangle$ is $\log_2 x + \log_2 y = n$, that is the size of the input is logarithmic in the values of the numbers.
  - So if the number of steps is proportional to the values of x and y, it is exponential in n.

BRUTE FORCE ALGORITHM IS EXPONENTIAL

Given an input $\langle x, y\rangle$ of length $n = \log_2 x + \log_2 y$, going through all numbers between 2 and $\min\{x, y\}$, and checking if they divide both x and y takes time exponential in n.

EXAMPLES OF PROBLEMS IN P

THEOREM 7.15

RELPRIME $\in$ P

PROOF

E implements the Euclidian algorithm.

$E =$' "On input $\langle x, y \rangle$

```
1. Repeat until $y = 0$
2.     Assign $x \leftarrow x \mod y$.
3.     Exchange $x$ and $y$.
4. Output $x$."
```

PROOF

R solves *RELPRIME*, using *E* as a subroutine.

$R =$ "On input $\langle x, y \rangle$

```
1. Run $E$ on $\langle x, y \rangle$.
2. If the result is 1, accept.
   Otherwise, reject."
```

- If $E \in$ P then $R \in$ P.
- Each of x and y is reduced by a factor of 2 every other time through the loop.
- Loop is executed at most $\min\{2\log_2 x, 2\log_2 y\}$ times which is $O(n)$.

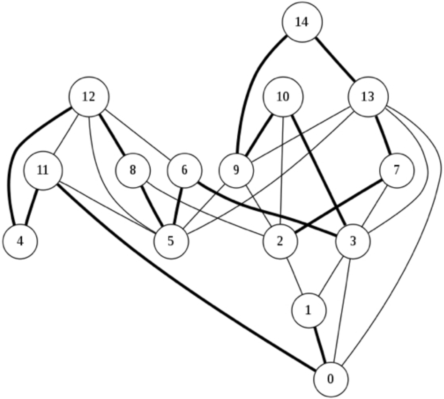
THE CLASS NP

- For some problems, even though there is a exponentially large search space of solutions (e.g., for the path problem), we can avoid a brute force solution and get a polynomial-time algorithm.
- For some problems, it is not possible to avoid a brute force solution and such problems have so far resisted a polynomial time solution.
- We may not yet know the principles that would lead to a polynomial time algorithm, or they may be “intrinsically difficult.”
- How can we characterize such problems?

THE HAMILTONIAN PATH PROBLEM

DEFINITION – HAMILTONIAN PATH

A Hamiltonian path in a directed graph *G* is a directed path that goes through each node exactly once.



THE HAMILTONIAN PATH PROBLEM

HAMILTONIAN PATH PROBLEM

$HAMPATH = \{\langle G, s, t\rangle \mid G$ is a directed graph with a Hamiltonian path from s to $t\}$.

- We can easily obtain an exponential time algorithm with a brute force approach.
  - Generate all possible paths between s and t and check if all nodes appear on a path!
- The *HAMPATH* problem has a property called polynomial verifiability.
  - If we can (magically) get a Hamiltonian path, we can verify that it is a Hamiltonian path, in polynomial time.
- *Verifying* the existence of a Hamiltonian path is “easier” than *determining* its existence.

POLYNOMIAL VERIFIABILITY

COMPOSITES PROBLEM

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

- We can easily verify if a number is composite, given a divisor of that number.
- A recent (but very complicated) algorithm for testing whether a number is prime or composite has been discovered.

$\overline{HAMPATH}$ PROBLEM

The $\overline{HAMPATH}$ problem has a solution if there is NO Hamiltonian path between s and t.

- Even if we knew, the graph did not have a Hamiltonian path, there is no easy way to verify this fact. We may need to take exponential time to verify it.

VERIFIERS

VERIFIER

A verifier for a language A is an algorithm V where

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

- We measure the time of a verifier only in terms of the length of w.
- A language A is polynomially verifiable if it has a polynomial time verifier.
- c is called certificate or proof of membership in A.
  - For the *HAMPATH* problem, the certificate is simply the Hamiltonian path from s to t.
  - For the *COMPOSITES* problem, the certificate is one of the divisors.

THE CLASS NP

THE CLASS NP

NP is the class of languages that have polynomial time verifiers.

- NP stands for nondeterministic polynomial time.
- Problems in NP are called NP-Problems.
- P $\subset$ ($\subseteq$?) NP.

A NONDETERMINISTIC DECIDER FOR *HAMPATH*

N_1 = "On input $\langle G, s, t \rangle$

1. Nondeterministically select list of m numbers $p_1, p_2, \ldots p_m$ with $1 \leq p_i \leq m$.
2. Check for repetitions in the list; if found, *reject*.
3. Check whether $p_1 = s$ and $p_m = t$, otherwise *reject*.
4. For $1 \leq i < m$, check if (p_i, p_{i+1}) is an edge of G. If any are not, *reject*. Otherwise *accept*."

- Stage 1 runs in polynomial time.
- Stages 2 and 3 take polynomial time.
- Stage 4 takes poynomial time.
- Thus the algorithm runs in nondeterministic polynomial time.

THE CLASS NP

THEOREM 7.20

A language is in NP, iff it is decided by some nondeterministic polynomial time Turing machine.

PROOF IDEA

- We show polynomial time verifier $\Leftrightarrow$ polynomial time decider TM.
  - NTM simulates the verifier by guessing a certificate.
  - The verifier simulates the NTM

PROOF: NTM GIVEN THE VERIFIER.

Let $A \in$ NP. Let V be a verifier that runs in time $O(n^k)$. N decides A in nondeterministic polynomial time.

$N =$ "On input w of length n

1. Nondeterministically select string c of length at most n^k.
2. Run V on input $\langle w, c\rangle$.
3. If V accepts, *accept*; otherwise *reject*."

THE CLASS NP

THEOREM 7.20

A language is in NP, iff it is decided by some nondeterministic polynomial time Turing machine.

PROOF IDEA

- We show polynomial time verifier $\Leftrightarrow$ polynomial time decider TM.
  - NTM simulates the verifier by guessing a certificate.
  - The verifier simulates the NTM

PROOF: VERIFIER GIVEN THE NTM.

Assume A is decided by a polynomial time NTM N. We construct the following verifier V
$V =$ "On input $\langle w, c \rangle$

1. Simulate N on input w, treating each symbol of c as a description of the nondeterministic choice at each step.
2. If this branch of N's computation accepts, *accept*; otherwise, *reject*."

THE CLASS NP

DEFINITION

$\text{NTIME}(t(n)) = \{L \mid L$ is a language decided by a $O(t(n))$ time nondeterministic TM.$\}$

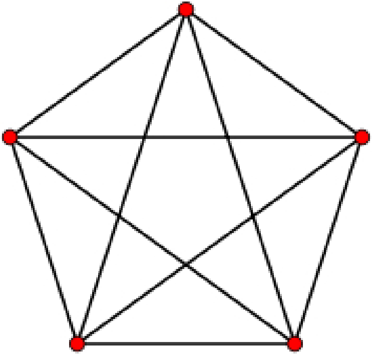
COROLLARY

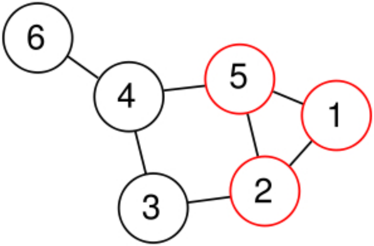
$\text{NP} = \bigcup_k \text{NTIME}(n^k)$.

THE CLIQUE PROBLEM

DEFINITION - CLIQUE

A clique in an undirected graph is a subgraph, wherein every two nodes are connected by an edge.
A *k*-clique is a clique that contains *k* nodes.





THE CLIQUE PROBLEM

THEOREM 7.24

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

PROOF

The clique is the certificate.

$V =$ "On input $\langle\langle G, k\rangle, c\rangle$:

1. Test whether c is a set of k nodes in G.
2. Test whether G has all edges connecting nodes in c.
3. If both pass, *accept*; otherwise *reject*."

ALTERNATIVE PROOF

Use a NTM as a decider.

$N =$ "On input $\langle G, k\rangle$:

1. Nondeterministically select a subset c of k nodes of G.
2. Test whether G has all edges connecting nodes in c.
3. If yes *accept*; otherwise *reject*."

- All steps take polynomial time.

THE SUBSET-SUM PROBLEM

THEOREM 7.25

$SUBSET\text{-}SUM = \{\langle S, t\rangle \mid S = \{x_1, \ldots, x_k\}$ and for some
$\{y_1, \ldots, y_l\} \subseteq S, \sum y_i = t\} \in \mathrm{NP}$.

PROOF

The clique is the certificate.
$V =$ "On input $\langle\langle S, t\rangle, c\rangle$:

1. Test whether c is a set of numbers summing to t.
2. Test whether S contains all numbers in c.
3. If both pass, *accept*; otherwise *reject*."

ALTERNATIVE PROOF

Use a NTM as a decider.
$N =$ "On input $\langle S, k\rangle$:

1. Nondeterministically select a subset c of numbers in S.
2. Test whether S contains all numbers in c.
3. If yes *accept*; otherwise *reject*."

- All steps take polynomial time.

THE CLASS coNP

- It turns out $\overline{\textit{CLIQUE}}$ or $\overline{\textit{SUBSET-SUM}}$ are NOT in NP.
- Verifying something is NOT present seems to be more difficult than verifying it IS present.
- The class coNP contains all problems that are complements of languages in NP.
- We do not know if coNP $\neq$ NP.