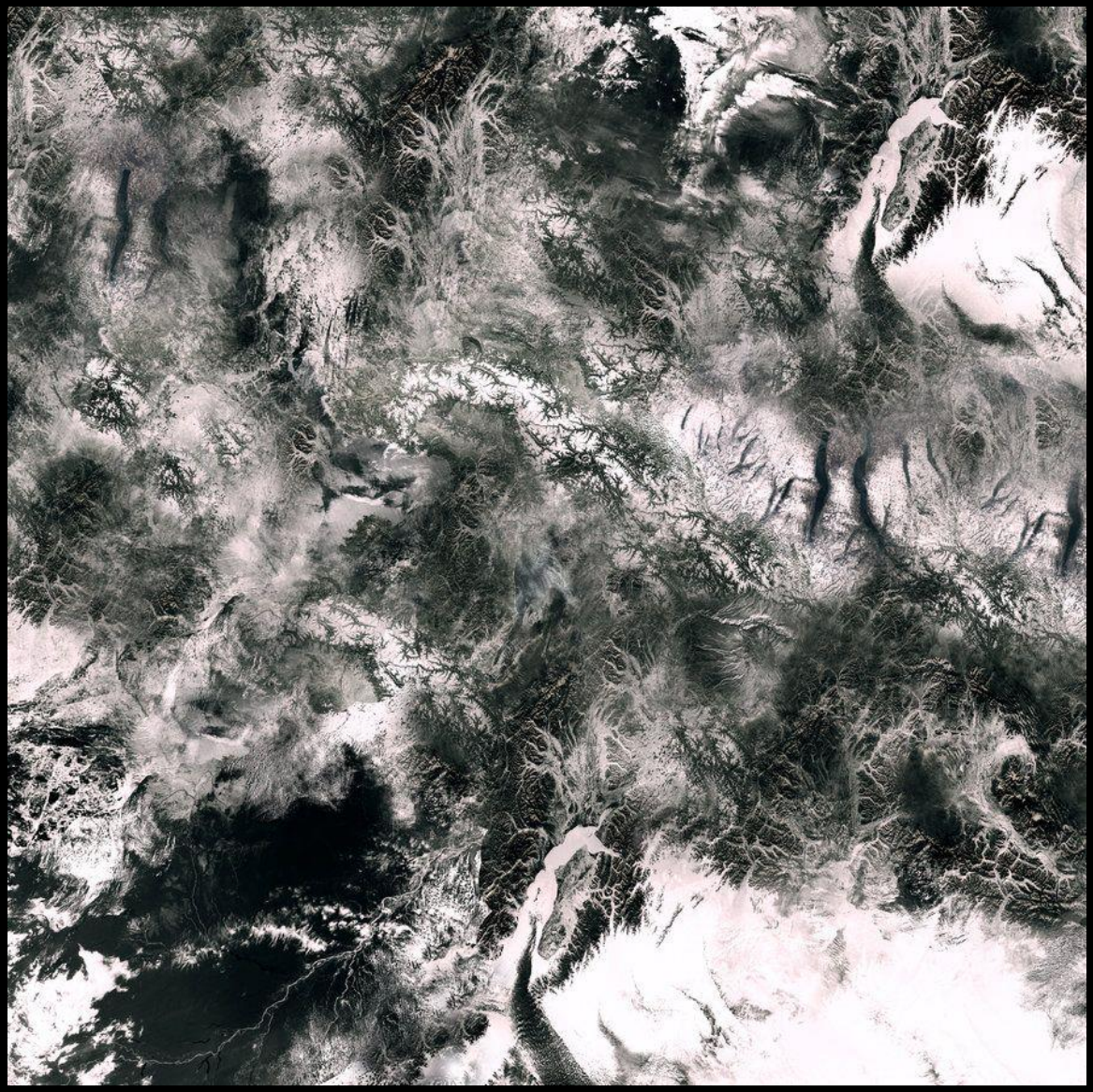


Computational Complexity

Contents

- Measuring Complexity
- The Class P
- The Class NP
- NP-Completeness

Measuring Complexity

• To date, we have only considered the question as to whether a problem is decidable, i.e. computationally solvable in principle.

• We now investigate the **computationally complexity** of a problem – that is, the time, memory, or other resources required to solve it.

Measuring Complexity

• We will analyze the time complexity of an algorithm by considering the number of steps the algorithm uses as a function of its input.

• In **worst-case analysis**, we consider the longest running time of all inputs of a given length; in **average-case analysis**, we consider the average of all the running times of inputs of a particular length.

Time Complexity

• We will analyze the time complexity of an algorithm by considering the number of steps the algorithm uses as a function of its input.

• In **worst-case analysis**, we consider the longest running time of all inputs of a given length; in **average-case analysis**, we consider the average of all the running times of inputs of a particular length.

Definition. Let M be a deterministic Turing machine that halts on all inputs. The **running time or time complexity** of M is the function $f: \mathbb{N} \longrightarrow \mathbb{N}$ 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 that M runs in time $f(n)$ and that M is an $f(n)$ time TM. Customarily we use n to represent the length of the input.

Big-O Notation

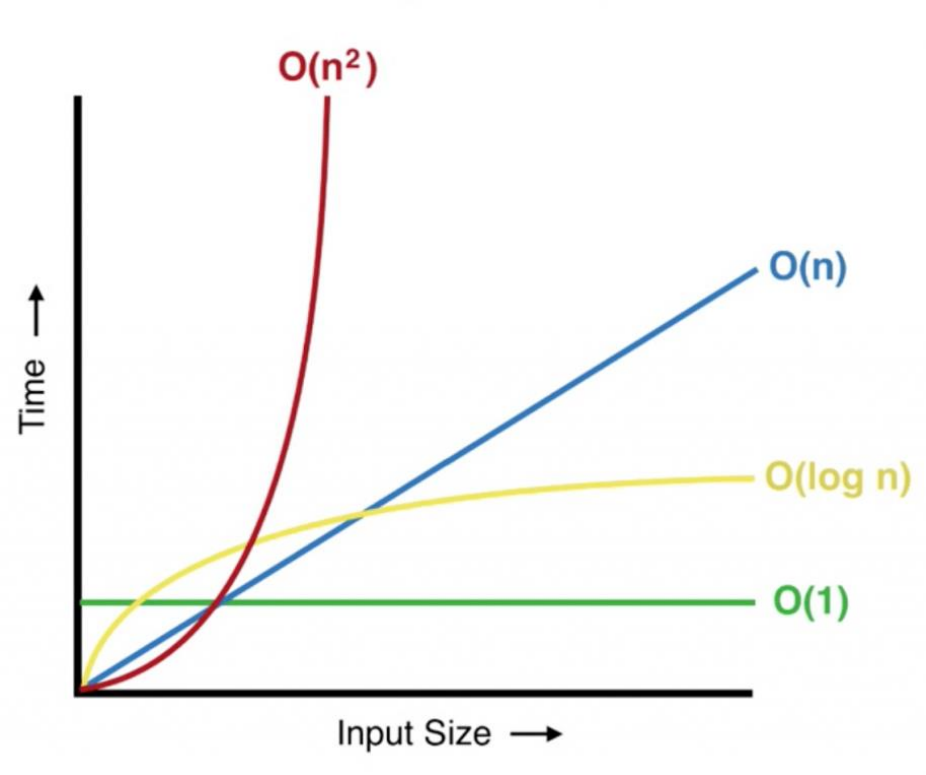
Definition. Let f and g be functions $f, g: \mathbb{N} \longrightarrow \mathbb{R}^{+}$. Say that $\boldsymbol{f(n) = O(g(n))}$ if positive integers c and n_0 exist such that for every $n > n_0$:

$$f(n) \leq cg(n)$$

Big-O Notation

Definition. Let f and g be functions $f, g: \mathbb{N} \longrightarrow \mathbb{R}^{+}$. Say that $\boldsymbol{f(n) = O(g(n))}$ if positive integers c and n_0 exist such that for every $n > n_0$:

$$f(n) \leq cg(n)$$



Big-O Notation

Definition. Let f and g be functions $f, g: \mathbb{N} \longrightarrow \mathbb{R}^{+}$. Say that $\boldsymbol{f(n) = O(g(n))}$ if positive integers c and n_0 exist such that for every $n > n_0$:

$$f(n) \leq cg(n)$$

• When $f(n) = O(g(n))$, we say that $g(n)$ is an **upper bound** (more precisely: an **asymptotic upper bound**) for $f(n)$.

• Intuitively, $f(n) = O(g(n))$ means that f is less than or equal to g if we disregard differences up to constant factors.

Big-O Notation

Definition. Let f and g be functions $f, g: \mathbb{N} \longrightarrow \mathbb{R}^{+}$. Say that $\boldsymbol{f(n) = O(g(n))}$ if positive integers c and n_0 exist such that for every $n > n_0$:

$$f(n) \leq cg(n)$$

• Example: $5n^3 + 2n^2 + 22n = O(n^3)$, since $5n^3 + 2n^2 + 22n \leq 6n^3 \; \forall \; n \geq 10$.

Notice that upper bounds are non-unique, for instance: $5n^3 + 2n^2 + 22n = O(n^4)$, $5n^3 + 2n^2 + 22n = O(n^5)$, etc.

Big-O Notation

- Example: **Log bases are irrelevant with Big-O notation**.

Recall the *log-base conversion formula* from elementary mathematics:

$$log_b n = \frac{\ln n}{\ln b}$$

Since changing the base only changes the value of the expression factor, it follows, for instance, that: $O(log_{10} n) = O(log_2 n)$, etc.

Big-O Notation

• Example: **Log bases are irrelevant with Big-O notation**.

Recall the *log-base conversion formula* from elementary mathematics:

$$log_b n = \frac{\ln n}{\ln b}$$

Since changing the base only changes the value of the expression factor, it follows, for instance, that: $O(log_{10} n) = O(log_2 n)$, etc.

• Example: $3nlog_2 n + 5nlog_2 log_2 n + 2 = O(n \log n)$.

• Example: $f(n) = 2^{O(\log n)}$ represents an upper bound for n^c for some c, since $n = 2^{log_2 n}$ and so $n^c = 2^{clog_2 n}$.

Little-O Notation

• Bounds of the form n^c are called **polynomial bounds**; bounds of the form 2^n are called **exponential bounds**.

• The companion to Big-O notation is known as **little-O notation**.

• Whereas Big-O notation captures the notion of an upper bound, $f(n) = O(g(n))$ meaning that $f(n)$ is asymptotically no more than $g(n)$, $f(n) = o(g(n))$ connotes the fact that $f(n)$ is asymptotically strictly less than $g(n)$.

Little-O Notation

Definition. Let f and g be functions $f, g: \mathbb{N} \longrightarrow \mathbb{R}^+$. Say that $\boldsymbol{f(n) = o(g(n))}$ if:

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

Little-O Notation

Definition. Let f and g be functions $f, g: \mathbb{N} \longrightarrow \mathbb{R}^+$. Say that $\boldsymbol{f(n) = o(g(n))}$ if:

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

In other words, $f(n) = o(g(n))$ means that for any real number $c > 0$, a number n_0 exists, where $f(n) < cg(n) \ \forall \ n \geq n_0$.

Little-O Notation

Definition. Let f and g be functions $f, g: \mathbb{N} \longrightarrow \mathbb{R}^+$. Say that $\boldsymbol{f(n) = o(g(n))}$ if:

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

In other words, $f(n) = o(g(n))$ means that for any real number $c > 0$, a number n_0 exists, where $f(n) < cg(n) \ \forall\, n \geq n_0$.

Example: $\sqrt{n} = o(n)$, since $lim_{n\to\infty} \frac{\sqrt{n}}{n} = lim_{n\to\infty} \frac{1}{\sqrt{n}} = 0$.

Little-O Notation

Definition. Let f and g be functions $f, g: \mathbb{N} \longrightarrow \mathbb{R}^+$. Say that $\boldsymbol{f(n) = o(g(n))}$ if:

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

In other words, $f(n) = o(g(n))$ means that for any real number $c > 0$, a number n_0 exists, where $f(n) < cg(n) \ \forall\, n \geq n_0$.

Example: $n\ log\ log\ n = o(n \log n)$,

since $lim_{n\to\infty} \frac{n\ log\ log\ n}{n \log n} = lim_{n\to\infty} \frac{log\ log\ n}{log\ n} =^*$

$lim_{n\to\infty} \frac{\frac{1}{nlog\ n}}{\frac{1}{n}} = 0.$ (*indicates use of *L'Hôpital's rule*)

Analyzing Algorithms

We now analyze a TM algorithm for recognizing the language $A = \{0^k 1^k | k \geq 0\}$.

M_1 = "On input string w:

1. Scan across the tape and *reject* if a 0 is found to the right of a 1.
2. Repeat if both 0s and 1s remain on the tape:
3. Scan across the tape, crossing off a single 0 and a single 1.
4. If 0s still remain after all the 1s have been crossed off, or if 1s still remain after all the 0s have been crossed off, *reject*. Otherwise, if neither 0s nor 1s remain on the tape, *accept*."

Analyzing Algorithms

We now analyze a TM algorithm for recognizing the language $A = \{0^k 1^k | k \geq 0\}$.

M_1 = "On input string w:

1. Scan across the tape and *reject* if a 0 is found to the right of a 1.
2. Repeat if both 0s and 1s remain on the tape:
3. Scan across the tape, crossing off a single 0 and a single 1.
4. If 0s still remain after all the 1s have been crossed off, or if 1s still remain after all the 0s have been crossed off, *reject*. Otherwise, if neither 0s nor 1s remain on the tape, *accept*."

(1) In stage (1), we scan the input tape in its entirety and then return the tape head to the left-most position, this takes $2n = O(n)$ steps.

(2) & (3) The machine scans the input; for each scan crosses off two symbols, so at most n/2 scans can occur. So the total time for steps (2) and (3) is: $O(n)\ O(n/2) = O(n^2)$.

(4) One final scan: $O(n)$.

Total complexity: $O(n) + O(n^2) + O(n) = O(n^2)$.

Analyzing Algorithms

Here is an asymptotically faster TM algorithm for recognizing the language $A = \{0^k 1^k | k \geq 0\}$.

M_2 = "On input string w:

1. Scan across the tape and *reject* if a 0 is found to the right of a 1.
2. Repeat as long as some 0s and some 1s remain on the tape:
3. Scan across the tape, checking whether the total number of 0s and 1s remaining is even or odd. If it is odd, *reject*.
4. Scan again across the tape, crossing off every other 0 starting with the first 0, and then crossing off every other 1 starting with the first 1.
5. If no 0s and no 1s remain on the tape, *accept*. Otherwise, *reject*."

Analyzing Algorithms

Here is an asymptotically faster TM algorithm for recognizing the language $A = \{0^k 1^k | k \geq 0\}$.

M_2 = "On input string w:

1. Scan across the tape and *reject* if a 0 is found to the right of a 1.
2. Repeat as long as some 0s and some 1s remain on the tape:
3. Scan across the tape, checking whether the total number of 0s and 1s remaining is even or odd. If it is odd, *reject*.
4. Scan again across the tape, crossing off every other 0 starting with the first 0, and then crossing off every other 1 starting with the first 1.
5. If no 0s and no 1s remain on the tape, *accept*. Otherwise, *reject*."

• Let's first confirm that the algorithm above decides language A.

• After the scan performed in (4), the total number of 0s remaining is cut in half and the remainder is discarded. So if we started with 13 0s, we would get 6 0s after step 4, then 3 0s, then 1 0 and then none. This stage has the same effect on the number of 1s.

• The sequence of parities found (e.g. odd, even, odd, odd) gives the (reverse) binary representation of the number of zeros (and ones). If all parities agree, the numbers are equal.

Analyzing Algorithms

Here is an asymptotically faster TM algorithm for recognizing the language $A = \{0^k 1^k | k \geq 0\}$.

M_2 = "On input string w:

1. Scan across the tape and *reject* if a 0 is found to the right of a 1.
2. Repeat as long as some 0s and some 1s remain on the tape:
3. Scan across the tape, checking whether the total number of 0s and 1s remaining is even or odd. If it is odd, *reject*.
4. Scan again across the tape, crossing off every other 0 starting with the first 0, and then crossing off every other 1 starting with the first 1.
5. If no 0s and no 1s remain on the tape, *accept*. Otherwise, *reject*."

• Now we analyze the run-time of M_2.

• Every stage requires $O(n)$ time. Next we consider how many times each stage is executed:

(1) and (5): once each.

(4) At most $1 + log_2(n)$ (crosses off at least half the 0s and 1s)

Total: $\left(1 + log_2(n)\right) O(n) = O(n \log n)$

Analyzing Algorithms

Definition. Let $t: \mathbb{N} \longrightarrow \mathbb{R}^+$ be a function. Define the time complexity class $TIME(t(n))$, to be the collection of all languages that are decidable by an $O(t(n))$ time Turing machine.

Analyzing Algorithms

Definition. Let $t: \mathbb{N} \longrightarrow \mathbb{R}^+$ be a function. Define the time complexity class $TIME(t(n))$, to be the collection of all languages that are decidable by an $O(t(n))$ time Turing machine.

$$A = \{0^k 1^k | k \geq 0\}$$

- By the second-to-last example, M, we showed $A \in TIME(n^2)$.

- Using the previous example, M_2, we rendered a tighter bound, showing $A \in TIME(n \log n)$.

- In fact, one can show* that $A \in TIME(n)$. This is the best possible bound for A.

*See Sipser text for proof.

Analyzing Algorithms

• Next we explore how the choice of computational model can affect the time complexity of languages.

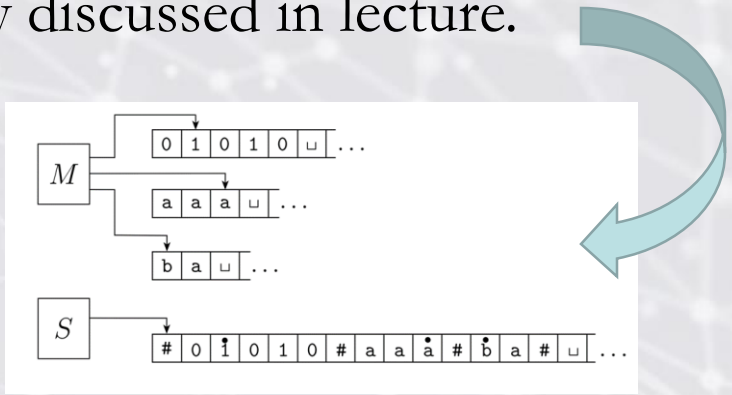
• We consider (3) models: single-tape TMs, multi-tape TMs, and non-deterministic TMs.

Theorem. Let $t(n)$ be a function, where $t(n) \geq n$. Then every $t(n)$ time multiple-tape TM has an equivalent $O(t^2(n))$ time single-tape TM.

Analyzing Algorithms

Theorem. Let $t(n)$ be a function, where $t(n) \geq n$. Then every $t(n)$ time multiple-tape TM has an equivalent $O(t^2(n))$ time single-tape TM.

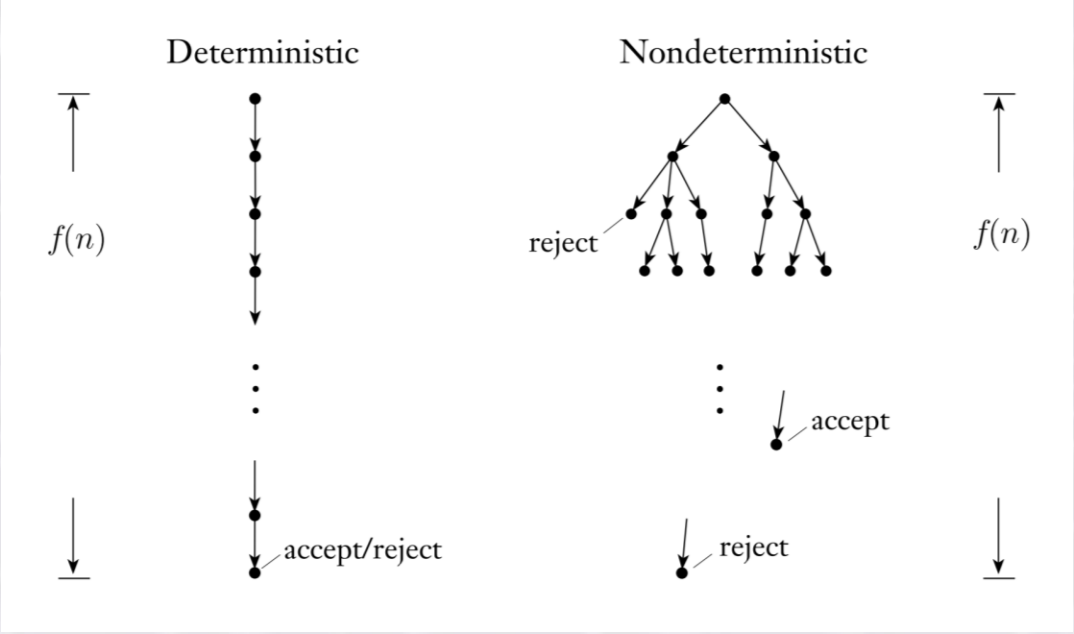
Proof Idea: Use the multi-tape to single-tape TM conversion algorithm we previously discussed in lecture.



The total time used by the simulation is given as follows: The initial stage, where the TM puts its tape in the proper format requires $O(n)$ steps. Afterward, the simulation of the initial $\mathrm{t}(n)$ steps requires $O(\mathrm{t}(n))$, so this gives $\mathrm{t}(n) \cdot O(\mathrm{t}(n)) = O(t^2(n))$ steps.

Analyzing Algorithms

Definition. Let N be a non-deterministic TM that is a decider. The **running time** of N is the function f: $\mathbb{N} \longrightarrow \mathbb{N}$, where $f(n)$ is the maximum number of steps that N uses on any branch of its computation on any input of length n, as shown in the figure.



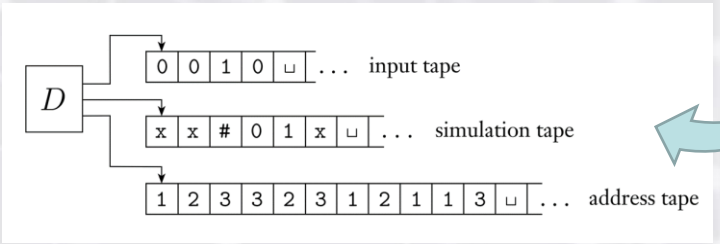
Analyzing Algorithms

Theorem. Let $t(n)$ be a function, where $t(n) \geq n$. Then every $t(n)$ non-deterministic TM has an equivalent $2^{O(t(n))}$ time deterministic TM.

Analyzing Algorithms

Theorem. Let $t(n)$ be a function, where $t(n) \geq n$. Then every $t(n)$ non-deterministic TM has an equivalent $2^{O(t(n))}$ time deterministic TM.

Proof Idea: Use the non-deterministic to deterministic three-tape TM conversion algorithm we previously discussed in lecture.



Using the non-determinism computation tree of original TM, perform BFS. The total number of nodes searched is bounded by $2^{O(t(n))}$.

Finally, converting the three-tape TM to a single-tape, deterministic TM at most squares the running time (by the previous theorem).

Thus, the running time of the final TM is: $\left(2^{O(t(n))}\right)^2 = 2^{O(t(n))}$ steps.

The Class P

Notice that the previous results confirm (2) important facts:

(1) There is at **most a polynomial difference** between the time complexity of problems measured on deterministic single-tape and multi-tape TMs.

(2) There is at **most an exponential difference** between the time complexity of problems on deterministic and non-deterministic TMs.

The Class P

• Generally speaking, **we consider polynomial-time algorithms to be efficient** (although certainly there exist practical exceptions to this standard).

• Conversely, **exponential-time algorithms are considered inefficient** – and are largely associated with brute-force (i.e. exhaustive) search algorithms.

All "reasonable"* computation models are polynomially equivalent – meaning that any one of them can simulate another with only a polynomial increase in run time.

* By reasonable, we loosely mean any physically-realizable computation device.

The Class P

Definition. **P** is the class of languages that are decidable in polynomial time on a deterministic, single-tape TM. In other words,

$$P = \bigcup_k TIME(n^k)$$

The Class P

Definition. **P** is the class of languages that are decidable in polynomial time on a deterministic, single-tape TM. In other words,

$$P = \bigcup_k TIME(n^k)$$

- The class P plays a crucial role in theory of computation:

(1) P is invariant for all models of computation that are polynomially equivalent to the deterministic, single-tape TM.

(2) P roughly corresponds to the class of problems that are realistically solvable on a real-world computer.

The Class P

• To **show that an algorithm is in the class P**, we generally demonstrate (2) things:

(1) We **give a polynomial upper bound** (typically in big-O notation) on the number of steps required for execution as a function of the input length n.

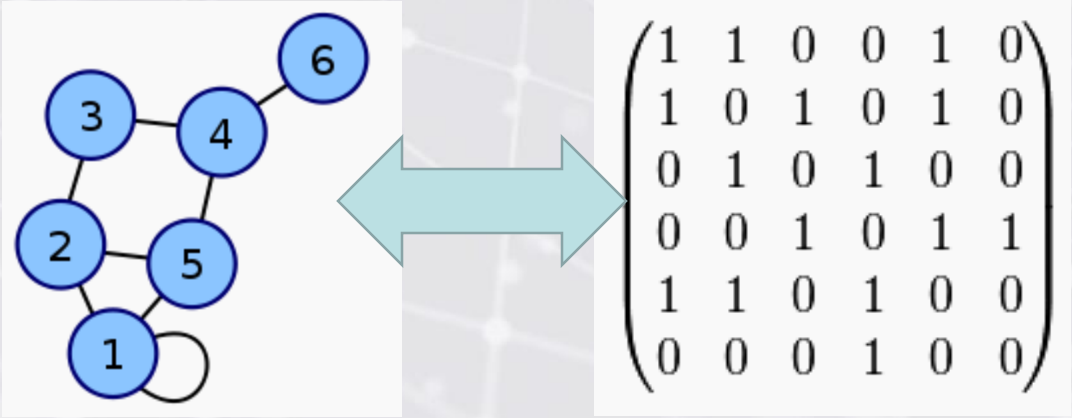
(2) We show **each step in the algorithm can be implemented in polynomial time** on a reasonable deterministic model.

The Class P

• Sometimes it is necessary to specifically describe the encoding method used in a particular problem in order to guarantee polynomial time execution of an algorithm.

• Notice, for instance, that **unary encoding grows exponentially** larger than other, "reasonable" encodings (e.g. 12 -> 111111111111).

• Conventionally, for a graph G, a reasonable encoding is given by a binary matrix called the **adjacency matrix** of G.



The Class P

- Consider the problem:

$$PATH = \{\langle G, s, t\rangle | G \text{ is a directed graph that has a directed path from } s \text{ to } t.\}$$

Next, we show that $PATH \in P$ by produced a polynomial time algorithm solving $PATH$.

The Class P

$PATH = \{\langle G, s, t\rangle | G \text{ is a directed graph that has a directed path from } s \text{ to } t.\}$

Theorem. $PATH \in P$

Proof. Note that a **brute-force algorithm**, where one proceeds by examining all potential paths in G and determining whether any is a directed path from s to t requires checking roughly m^m such paths (where m is the number of nodes in the graph). Naturally, this **won't work**.

The Class P

$PATH = \{\langle G, s, t\rangle | G \text{ is a directed graph that has a directed path from } s \text{ to } t.\}$

Theorem. $PATH \in P$

Proof. To get a polynomial time algorithm, we need to avoid brute-force. One alternative is to employ BFS (breadth first search). Here, we successively mark all nodes in G that are reachable from s by directed paths of length 1, length 2, length 3, etc.

M = "On input $\langle G, s, t\rangle$, where G is a directed graph with nodes s and t:

1. Place a mark on node s.
2. Repeat the following until no additional nodes are marked:
3. Scan all the edges of G. If an edge (a, b) is found going from a marked node a to an unmarked node b, mark node b.
4. If t is marked, *accept*. Otherwise, *reject*."

The Class P

$PATH = \{\langle G, s, t\rangle | G \text{ is a directed graph that has a directed path from } s \text{ to } t.\}$

Theorem. $PATH \in P$

Proof. To get a polynomial time algorithm, we need to avoid brute-force. One alternatively is to employ BFS (breadth first search). Here, we successively mark all nodes in G that are reachable from s by directed paths of length 1, length 2, length 3, etc.

M = "On input $\langle G, s, t\rangle$, where G is a directed graph with nodes s and t:

1. Place a mark on node s.
2. Repeat the following until no additional nodes are marked:
3. Scan all the edges of G. If an edge (a, b) is found going from a marked node a to an unmarked node b, mark node b.
4. If t is marked, *accept*. Otherwise, *reject*."

Lastly, we show that the algorithm runs in polynomial time. Stages 1 and 4 are run only once; Stage 3 runs at most m times. So the total number of stages is: $1 + 1 + m$.

Stages 1 and 4 are implemented in polynomial time; Stage 3 requires a scan of the input and test whether nodes are marked, which is easily executed in polynomial time.

Consequently, $PATH \in P$.

The Class P

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

Theorem. $RELPRIME \in P$

The Class P

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

Theorem. $RELPRIME \in P$

Proof. The problem is easily solved with the **Euclidean algorithm** (one of the first known algorithms in history). Recall that the Euclidean algorithm generates the GCD of two numbers by repeatedly computing $x \bmod y$ and then exchanging x and y. We denote the Euclidean algorithm as E.

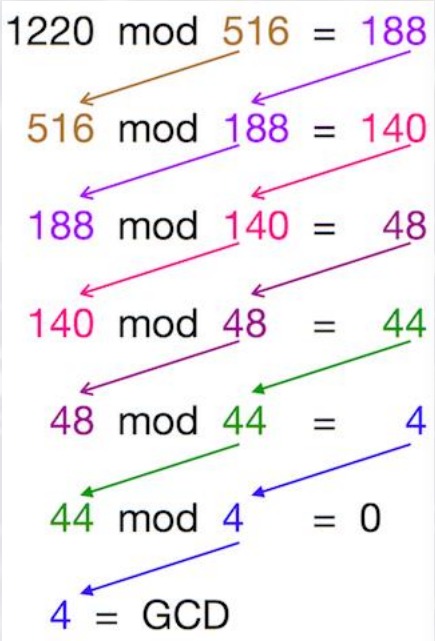
E = "On input $\langle x, y\rangle$, where x and y are natural numbers in binary:

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

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

R = "On input $\langle x, y\rangle$, where x and y are natural numbers in binary:

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



The Class P

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

Theorem. $RELPRIME \in P$

Proof.

E = "On input $\langle x, y\rangle$, where x and y are natural numbers in binary:

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

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

R = "On input $\langle x, y\rangle$, where x and y are natural numbers in binary:

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

To complete our proof, **we must confirm that the algorithm runs in polynomial time**.

Each execution of step (2) reduces the value of x by at least one half. At step (3) x and y are exchanged, so each of the original values of x and y are reduced by at least half every other time through the loop.

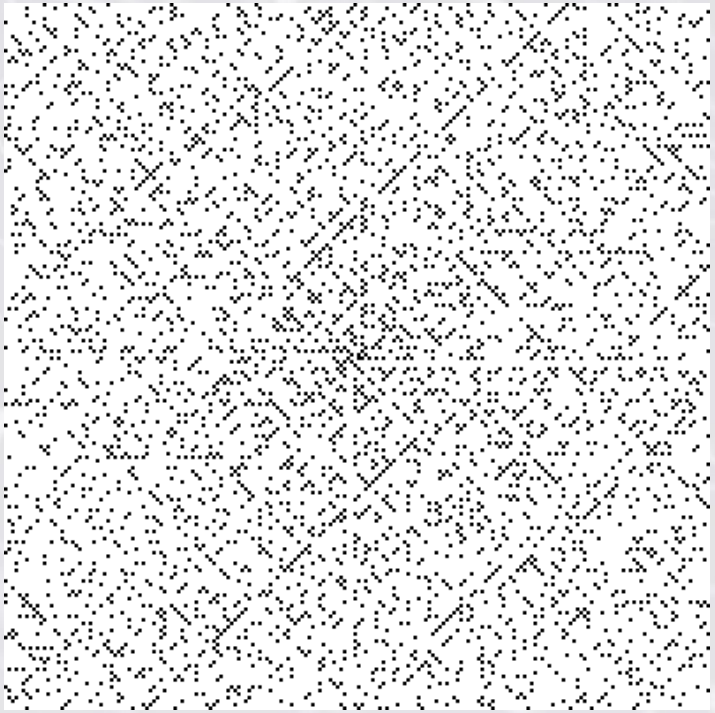
The maximum number of times the algorithm runs steps(2) and (3) is $max(2log_2 x, 2log_2 y)$. Thus, $RELPRIME \in P$.

The Class P

• Aside: In 2002, in a landmark result, it was shown that ***PRIME ∈ P***, using the **AKS primality test**.

• The authors **Agrawal–Kayal–Saxena**, were awarded the Gödel prize in 2006.

• Other notable P problems include LP (linear programming) and maximum matching.



Input	Integer $n > 1$
Step 1	If $(n = a^b$ for $a \in N$ and $b > 1)$, output COMPOSITE.
Step 2	Find the smallest r such that $\phi_r(n) > 4(log_2 n)^2$
Step 3	If $1 < (a, n) < n$ for some $a = r$, output COMPOSITE.
Step 4	If $n = r$, output PRIME.
Step 5	For $a = 1$ to $\lfloor 2log_2 n\sqrt{\varphi(r)} \rfloor$ do If $((x+a)^n \neq x^n + a \pmod{x^r - 1, n})$, output COMPOSITE;
Step 6	Output PRIME.

The Class P

Theorem. Every CFL is in P.

Proof Idea. Recall that we previously proved every CFL is decidable. In doing so, we relied on the fact that every CFG can be re-expressed in CNF (Chomsky normal form); from CNF we know that every derivation of a string w (of length n) requires $2n - 1$ steps.

The decider for the CFL worked by trying all possible derivations of length $2n - 1$. However, this algorithm doesn't run in polynomial time; the number of derivations with k steps may be exponential in k.

To derive a polynomial time algorithm, we use **dynamic programming**.

Recall that dynamic programming is an algorithmic paradigm that (through recursion) utilizes the accumulation of information about subproblems to solve a larger problem.

The Class P

Theorem. Every CFL is in P.

Proof. Consider the subproblem of determining **whether each variable in G generates each substring of w**. We store this information in an $n \times n$ table.

For $i \leq j$, the $(i,j)th$ entry of the table contains the collection of variables that generate the substring: $w_i\ w_{i+1} \ldots w_j$ (for $i > j$ the table entries are unused). The algorithm fills in the table entries of w. First it fills in the entries for the substrings of length 1, then length 2, and so on.

The Class P

Theorem. Every CFL is in P.

Proof.

For example, suppose the algorithm has already determined which variables generate all substrings up to length k. Now, to determine whether a variable A generates a particular substring of length $k+1$, the algorithm splits that substring into two non-empty pieces in the k possible ways.

For each split, the algorithm examines each rule $A \to BC$ to determine whether B generates the first piece and C generates the second piece, using the previously computed table entries. If both B and C generate the respective pieces, A generates the substring and so it is added to the associated table entry.

The Class P

Theorem. Every CFL is in P.

PROOF The following algorithm D implements the proof idea. Let G be a CFG in Chomsky normal form generating the CFL L. Assume that S is the start variable. (Recall that the empty string is handled specially in a Chomsky normal form grammar. The algorithm handles the special case in which $w = \varepsilon$ in stage 1.) Comments appear inside double brackets.

D = "On input $w = w_1 \cdots w_n$:

1. For $w = \varepsilon$, if $S \to \varepsilon$ is a rule, *accept*; else, *reject*. [[$w = \varepsilon$ case]]
2. For $i = 1$ to n: [[examine each substring of length 1]]
3. For each variable A:
4. Test whether $A \to \mathtt{b}$ is a rule, where $\mathtt{b} = w_i$.
5. If so, place A in $table(i, i)$.
6. For $l = 2$ to n: [[l is the length of the substring]]
7. For $i = 1$ to $n - l + 1$: [[i is the start position of the substring]]
8. Let $j = i + l - 1$. [[j is the end position of the substring]]
9. For $k = i$ to $j - 1$: [[k is the split position]]
10. For each rule $A \to BC$:
11. If $table(i, k)$ contains B and $table(k + 1, j)$ contains C, put A in $table(i, j)$.
12. If S is in $table(1, n)$, *accept*; else, *reject*."

* It can be shown that D runs in $O(n^3)$ steps.

The Class NP

• As we have seen, avoiding brute-force search often leads to a polynomial time solution to a problem.

• However, the question remains: **When does such a polynomial time solution exist for a problem**?

• Unfortunately, in general we do not know the answer to this question. In fact, this question is one of most important unsolved problem in the sciences.

The Class NP

• Perhaps many of these "difficult" problems (that is, ones that do not immediately admit of a polynomial time solution) have a polynomial time solution, but we have simply yet to discover them.

• Or, conversely, perhaps these problems are simply *intrinsically difficult*, in which case they cannot be solved with a polynomial time algorithm.

• Remarkably, the complexity of many (seemingly) difficult problems are linked – and a polynomial time algorithm for one such problem can be used to solve an entire class of problems (there problems are called **NP-complete** problems).

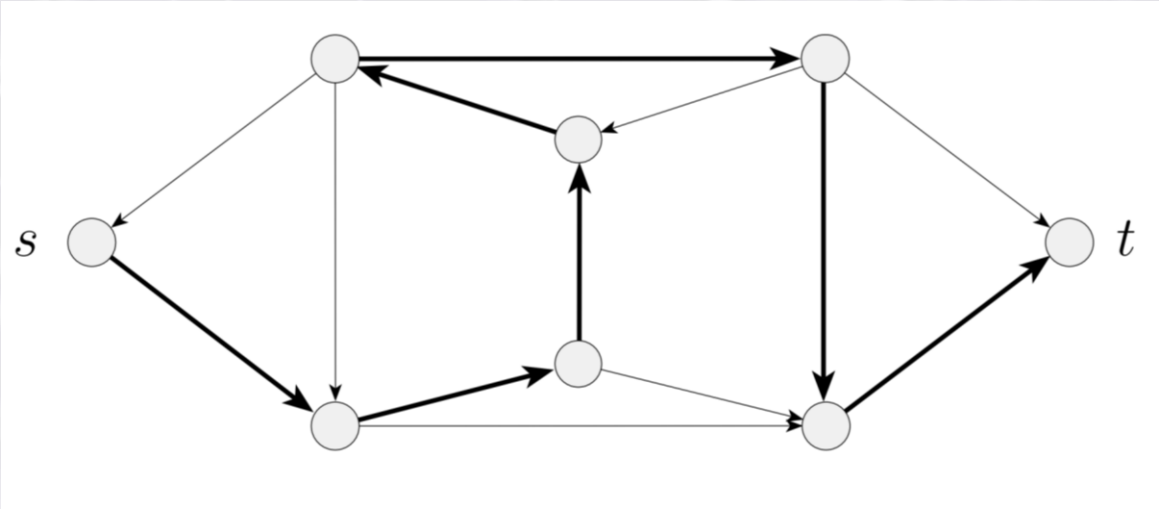
The Class NP

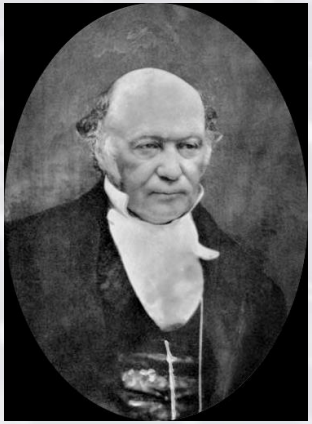
• Let's now consider a classic "hard" problem, that of finding a **Hamiltonian path** in a directed graph.

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

Define:

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





The Class NP

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

• Notice that we can easily obtain an exponential time algorithm for the HAMPATH problem by modifying the brute-force algorithm for $PATH$ given previously. We need only add a check to verify that the potential path is Hamiltonian.

• To date, no one knows whether $HAMPATH \in P$.

• The $HAMPATH$ problem has a feature called **polynomial verifiability**. Even though we don't know of a fast (i.e. polynomial time) algorithm to determine when a graph contains a Hamiltonian path, if such a candidate path were discovered, **we could nonetheless easily verify whether this path is Hamiltonian**.

The Class NP

• In summary, **verifying the existence of a Hamiltonian path is much easier than determining its existence**.

• Consider the problem:

$$COMPOSITES = \{x | x = pq, for\ integers\ p, q > 1.\}$$

• Like $HAMPATH$, $COMPOSITES$ is easy to verify in polynomial time. In fact, as we have seen, PRIME $\in P$, so $COMPOSITES \in P$ follows.

The Class NP

• We have seen that many – even ostensibly hard – problems admit of polynomial time verification. Even so, there exist problems that do not even admit of (known) polynomial time verification.

• For example, $\overline{HAMPATH}$ would require a polynomial time verification to confirm that a graph does *not* have a Hamiltonian path; we don't know of a way for someone to verify the nonexistence of such a path without using the previously mentioned exponential time algorithm.

The Class NP

Definition. A **verifier** for a language A is an algorithm V, where:

$$A = \{w | \, V \; accepts \; \langle w, c \rangle \; for \; some \; string \; c\}.$$

We measure the time of a verifier only in terms of the length of w, so a **polynomial time verifier** runs in polynomial time in the length of w. A language A is polynomially verifiable if it has a polynomial time verifier.

*Here the symbol c is known as a **certificate**, or **proof**, of membership in A. The certificate provides extra information to verify that a string w is a member of A.

For $HAMPATH$, the certificate could simply be the proposed path from s to t; for $COMPOSITES$ the certificate could be one of the divisors of x.

The Class NP

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

The Class NP

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

• The class NP is of immense importance in computational complexity theory, as it contains a large number of problems of practical importance, including: $HAMPATH$, $COMPOSITES$, $PRIME$, etc.

• The term NP relates to the notion of **nondeterministic polynomial time**; as we show subsequently, the class NP can be equivalently defined as languages decided by some nondeterministic polynomial time TM.

The Class NP

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

• The term NP relates to the notion of **nondeterministic polynomial time**; as we show subsequently, the class NP can be equivalently defined as languages decided by some nondeterministic polynomial time TM.

Here is a nondeterministic polynomial time algorithm for $HAMPATH$:

N_1 = "On input $\langle G, s, t \rangle$, where G is a directed graph with nodes s and t:

1. Write a list of m numbers, $p_1, \ldots, p_m$, where m is the number of nodes in G. Each number in the list is nondeterministically selected to be between 1 and m.
2. Check for repetitions in the list. If any are found, *reject*.
3. Check whether $s = p_1$ and $t = p_m$. If either fail, *reject*.
4. For each i between 1 and $m - 1$, check whether (p_i, p_{i+1}) is an edge of G. If any are not, *reject*. Otherwise, all tests have been passed, so *accept*."

The Class NP

Theorem. A language is in NP *iff* it is decided by some nondeterministic polynomial time TM.

The Class NP

Theorem. A language is in NP *iff* it is decided by some nondeterministic polynomial time TM.

Proof. $(\rightarrow)$ For the forward direction, let $A \in NP$ and show that A is decided by a polynomial time NTM N. Let V be the polynomial time verifier for A that exists by definition of NP.

Assume that V is a TM that runs in time n^k and construct N as follows:

$N =$ "On input w of length n:

(1) Nondeterminstically 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. A language is in NP *iff* it is decided by some nondeterministic polynomial time TM.

Proof. $(\leftarrow)$ To prove the other direction, assume A is decided by a polynomial time NTM N and construct a polynomial time verifier V as follows:

$V =$ "On input $\langle w, c \rangle$ where w and c are strings:

(1) Simulate N on input w, treating each symbol of c as a description of the nondeterministic choice to make at each step .

(2) If this branch of N's computation accepts, **accept**; otherwise **reject**."

The Class NP

Def.

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

The Class NP

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

• We define the nondeterminstic time complexity class $NTIME(t(n))$ as analogous to the deterministic time complexity class $TIME(t(n))$.

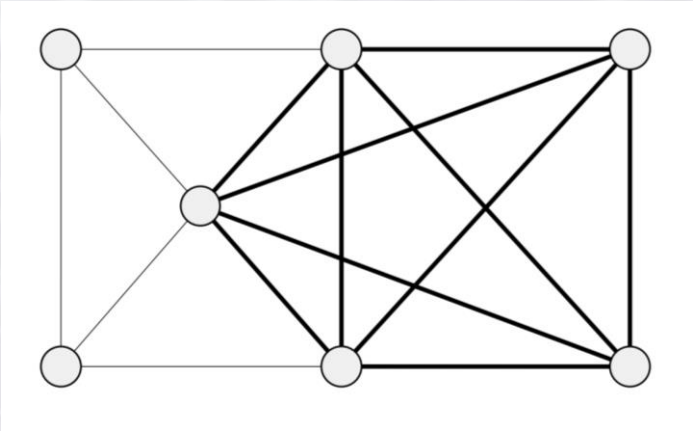
• By definition, the following corollary follows immediately:

Corollary. $NP = \bigcup_k NTIME(n^k)$

• Observe that the class NP is insensitive to the choice of reasonable nondeterministic computational model because all such models are polynomially equivalent.

The Class NP

• A **clique** in an undirected graph is a set of nodes that are all mutually adjacent. By extension, a **k-clique** is a clique that contains k nodes.



• Define the **clique problem**:

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

The Class NP

Theorem. $CLIQUE \in NP$

The Class NP

Theorem. $CLIQUE \in NP$

- The proof simply uses the clique as certificate.

PROOF The following is a verifier V for *CLIQUE*.

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

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

ALTERNATIVE PROOF If you prefer to think of NP in terms of nondeterministic polynomial time Turing machines, you may prove this theorem by giving one that decides *CLIQUE*. Observe the similarity between the two proofs.

N = "On input $\langle G, k\rangle$, where G is a graph:

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

The Class NP

• Define the **SUBSET-SUM** problem: givens a collection of numbers $x_1, \dots x_k$ and a target t, we want to determine whether the collection contains a subcollection that adds up to the target (repetition is allowed).

• This problem is closely related to the classic combinatorial optimization problem known colloquially as the **knapsack problem** (1897).

$$SUBSET - SUM = \left\{ \begin{matrix} \langle S, t \rangle | S = \{x_1, \dots x_k\}, and\ for\ some \\ \{y_1, \dots y_l\} \subseteq \{x_1, \dots x_k\}, we\ have\ \Sigma y_i = t \end{matrix} \right\}$$

The Class NP

Theorem. $SUBSET - SUM \in NP$

- The proof simply uses the subset as certificate.

PROOF The following is a verifier V for *SUBSET-SUM*.

V = "On input $\langle \langle S, t \rangle, c \rangle$:

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

ALTERNATIVE PROOF We can also prove this theorem by giving a nondeterministic polynomial time Turing machine for *SUBSET-SUM* as follows.

N = "On input $\langle S, t \rangle$:

1. Nondeterministically select a subset c of the numbers in S.
2. Test whether c is a collection of numbers that sum to t.
3. If the test passes, *accept*; otherwise, *reject*."

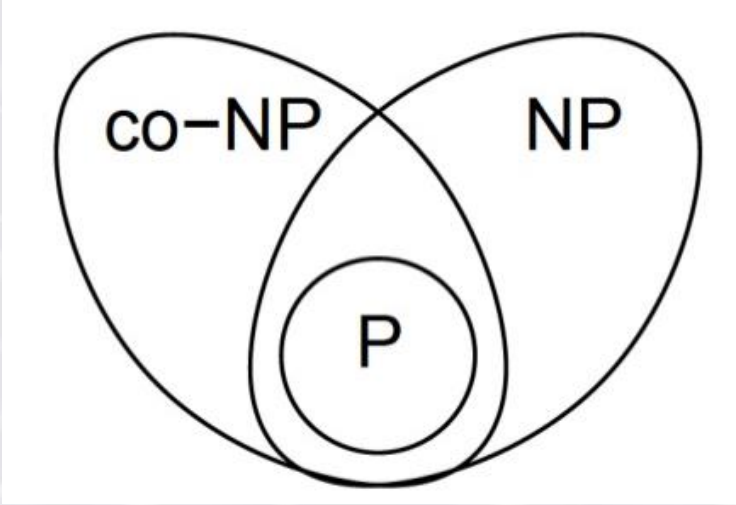
The Class NP

• We note that the complements of the previous sets, namely: $\overline{SUBSET - SUM}$ and $\overline{CLIQUE}$ are not obviously members of NP.

• Verifying that something is not present seems generally more difficult than verifying it is present. The complexity class **coNP** denotes the languages that are complements of languages in NP.

• We currently **don't know if coNP is different from NP**.

• The question of whether $\boldsymbol{P = NP}$ is one of the greatest unsolved problems in theoretical computer science and contemporary mathematics. Most researchers believe that the two classes are not equal.



NP-Completeness

• Previously we defined the notion of **reducing one problem to another** using mapping reducibility. The motivation for introducing this concept was to show that **if A reduced to B, a solution to B can be used to solve A**.

• We now consider the specific case of an efficient reduction from A to B.

NP-Completeness

• Previously we defined the notion of reducing one problem to another using mapping reducibility. The motivation for introducing this concept was to show that if A reduced to B, a solution to B can be used to solve A.

• We now consider the specific case of an efficient reduction from A to B.

Def. A function $f: \Sigma^* \rightarrow \Sigma^*$ is a **polynomial time computable function** if some polynomial time TM M exists that halts with just $f(w)$ on its tape, when started on any input w.

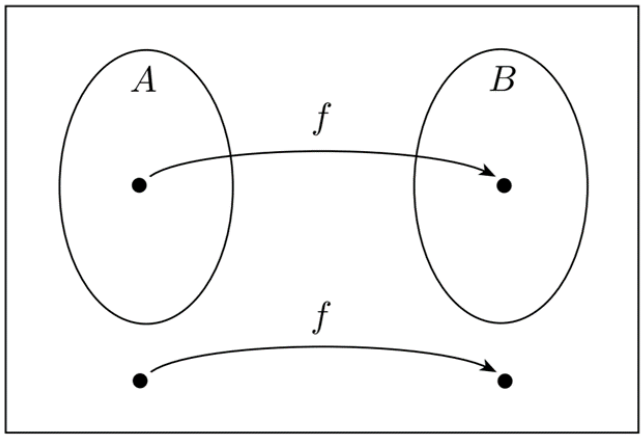
NP-Completeness

Def. A function $f: \Sigma^* \rightarrow \Sigma^*$ is a **polynomial time computable function** if some polynomial time TM M exists that halts with just $f(w)$ on its tape, when started on any input w.

Def. Language A is **polynomial time mapping reducible**, or simply polynomial time reducible, to language B , written $A \leq_p B$, if a polynomial time computable function $f: \Sigma^* \rightarrow \Sigma^*$ exists, where for every w:

$$w \in A \leftrightarrow f(w) \in B$$

The function f is called the **polynomial time reduction** from A to B.



NP-Completeness

Theorem. If $A \leq_p B$ and $B \in$ P, then $A \in$ P.

NP-Completeness

• We now define the well-known **3*SAT* problem** – first some terminology.

• A **literal** is a Boolean variable or its negation; a **clause** is several literals connected with the OR operation.

• Finally, a Boolean formula is said to be in **conjunctive normal form** (CNF) if it comprises several clauses connected with ANDs, for example:

$$\varphi = (x_1 \vee \overline{x_2} \vee \overline{x_3}) \wedge (x_3 \vee \overline{x_5} \vee x_6) \wedge (x_2 \vee \overline{x_4} \vee \overline{x_6})$$

• We say that clause φ is **satisfiable** if there exist Boolean values which can be assigned to each variable in φ yielding a TRUE assignment.

$$3SAT = \{\langle\varphi\rangle | \varphi \text{ is a satisfiable } 3-CNF \text{ formula}\}$$

NP-Completeness

Theorem. $3SAT$ is polynomial time reducible to $CLIQUE$.

NP-Completeness

Theorem. $3SAT$ is polynomial time reducible to $CLIQUE$.

Proof Idea: Given a formula φ with k clauses such as:

$$(a_1 \vee b_1 \vee c_1) \wedge (a_2 \vee b_2 \vee c_2) \wedge \ldots \wedge (a_k \vee b_k \vee c_k)$$

The reduction f generates the string $\langle G, k \rangle$ where G is an undirected graph as follows.

The nodes in G are organized into k groups of three nodes each called **triples**, $t_1, \ldots, t_k$. Each triple corresponds to one of the clauses in φ, and each node in a triple correspond to a literal in the associated clause.

NP-Completeness

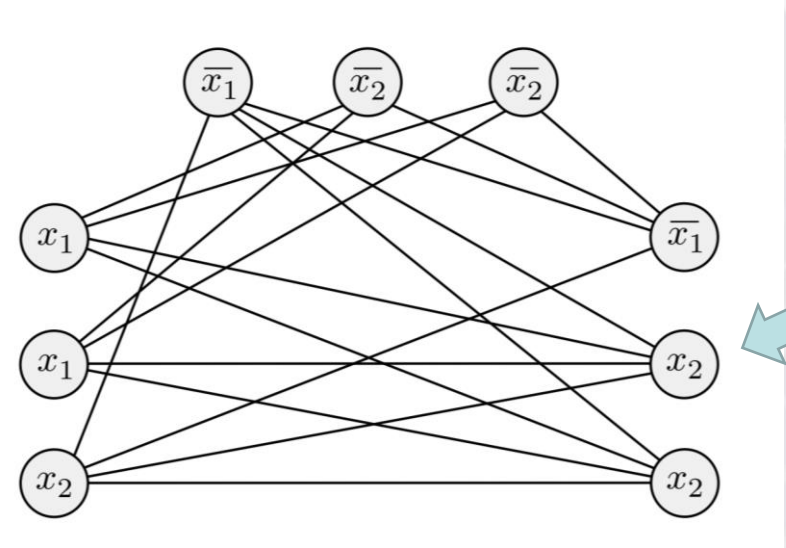
Theorem. $3SAT$ is polynomial time reducible to $CLIQUE$.

Proof Idea: Given a formula φ with k clauses such as:

$$(a_1 \vee b_1 \vee c_1) \wedge (a_2 \vee b_2 \vee c_2) \wedge \ldots \wedge (a_k \vee b_k \vee c_k)$$

The edges of G connect all but two types of pairs of nodes in G: (1) no edge is present between nodes in the same triple; (2) no edge is present between two nodes with contradictory labels (e.g.)

From this construction, it is not difficult to show that φ satisfiable *iff* G has a k -clique.



$$\phi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$$

NP-Completeness

Def. A language B is **NP-complete** if it satisfies:
(1) B is in NP
(2) Every $A \in NP$ is polynomial time reducible to B.

NP-Completeness

Def. A language B is **NP-complete** if it satisfies:
(1) B is in NP
(2) Every $A \in NP$ is polynomial time reducible to B.

Theorem. If B is NP-complete and $B \in P$, then $P = NP$.

Proof. The proof is automatic from the definition of polynomial time reducibility.

NP-Completeness

Theorem. If B is NP-complete and $B \leq_p C$ for $C \in NP$, then C is NP-complete.

NP-Completeness

Theorem. If B is NP-complete and $B \leq_p C$ for $C \in NP$, then C is NP-complete.

Proof. We already know that C is in NP, so we must show that every A in NP is polynomial time reducible to C.

Because B is NP-complete, every language in NP is polynomial time reducible to B, and B in turn is polynomial time reducible to C.

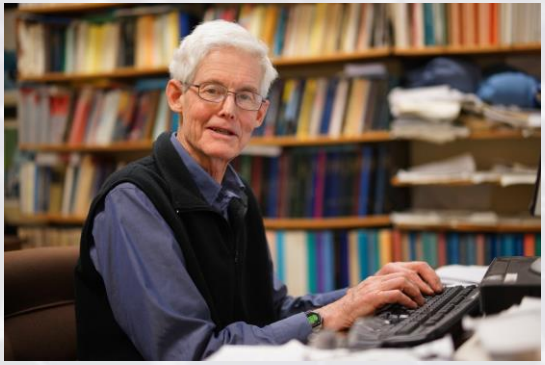
Polynomial time reductions compose; that is, if A is polynomial time reducible to B and B is polynomial time reducible to C, then A is polynomial time reducible to C.

Hence every language in NP is polynomial time reducible to C.

NP-Completeness

Cook-Levin Theorem. $SAT \in NP$-complete

• The Cook-Levin Theorem was a landmark result in computational complexity theorem; as it stands, it represents the culmination of several seminal papers published at the end of the 1960s and early 1970s. Cook later won the Turing award (1982) for his contributions to computational complexity theory, Levin later won the Knuth prize (2012) for this work.



• The formal proof is quite long; for brevity if I have omitted it from these lecture slides (please see text for full treatment).

The **key challenge of the proof** is to show that any language in NP is polynomial time reducible to SAT. The reduction for A takes a string w and produces a Boolean formula φ that simulates the NP machine for A on input w.

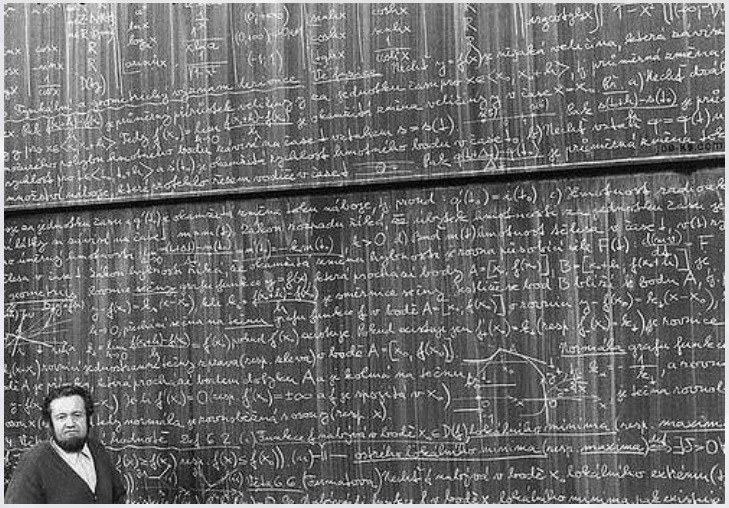
NP-Completeness

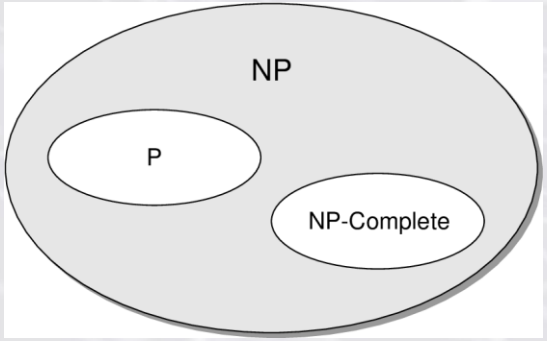
Cook-Levin Theorem. $SAT \in NP$-complete

• Following the Cook-Levin Theorem, showing the NP-completeness of other languages generally doesn't require such a lengthy proof. Instead, NP-completeness can be proved with a polynomial time reduction from a language that is already known to be NP-complete.

• We can use SAT for this purpose; but using $3SAT$ is usually easier.

• To this end, one can likewise show: **Theorem**. $3SAT \in NP$-complete.





Fin