CS 341: Foundations of CS II

Marvin K. Nakayama
Computer Science Department
New Jersey Institute of Technology
Newark, NJ 07102

Chapter 7
Time Complexity

Contents

- Time and space as resources
- Big O/little o notation, asymptotics
- Time complexity
- Polynomial time (P)
- Nondeterministic polynomial time (NP)
- NP-completeness

Introduction

- Chapters 3–5 dealt with **computability theory**:
  - "What is and what is not possible to solve with a computer?"
- For the problems that are computable, this leads to the next question:
  - "If we can decide a language A, how easy or hard is it to do so?"
- **Complexity theory** tries to answer this.

Counting Resources

- Two ways of measuring "hardness" of problem:
  1. **Time Complexity**:
     How many time-steps are required in the computation of a problem?
  2. **Space Complexity**:
     How many bits of memory are required for the computation?
- We will only examine time complexity in this course.
- We will use the Turing machine model.
  - If we measure time complexity in a crude enough way, then results for TMs will also hold for all "reasonable" variants of TMs.

Example

- Consider language
$$A = \{ 0^k 1^k \mid k \geq 0 \}.$$
- Below is a single-tape Turing machine M_1 that decides A:

  $M_1 =$ "On input w, where $w \in \{0,1\}^*$ is a string:

  1. Scan across tape and *reject* if 0 is found to the right of a 1.
  2. Repeat the following if both 0s and 1s appear on tape:
     - Scan across tape, crossing off single 0 and single 1.
  3. If 0s still remain after all 1s crossed out, or vice-versa, *reject*. Otherwise, if all 0s and 1s crossed out, *accept*."

$\downarrow$

0	0	0	1	1	1	␣	␣	⋯

- **Question:** How much time does TM M_1 need to decide A?

How much time does M_1 need?

- Number of steps may depend on several parameters.
- **Example:** If input is a graph, this could depend on
  - number of nodes
  - number of edges
  - maximum degree
  - all, some, or none of the above
- **Definition:** Complexity is measured as function of length of input string.
  - Worst case: **longest** running time on input of given length.
  - Average case: **average** running time on input of given length.
- We will only consider worst-case complexity.

Running Time

- Let M be a deterministic TM that halts on all inputs.
- We will study the relationship between
  - the length of encoding of a problem instance and
  - the required time complexity of the solution for such an instance (worst case).
- **Definition:** The **running time** or **time complexity** of M is a function $f : \mathcal{N} \to \mathcal{N}$ defined by the maximization:
$$f(n) \;=\; \max_{|x|=n} (\text{ number of time steps of } M \text{ on input } x \text{ })$$
- Terminology
  - $f(n)$ is the running time of M.
  - M is an $f(n)$-time Turing machine.

Running Time

- The exact running time of most algorithms is quite complex.
- Instead use an approximation for large problems.
- Informally, we want to focus only on "important" parts of running time.
- **Examples:**
  - $6n^3 + 2n^2 + 20n + 45$ has four terms.
  - $6n^3$ most important when n is large.
  - Leading coefficient "6" does not depend on n, so only focus on n^3.

Asymptotic Notation

- Consider functions f and g, where
$$f, g : \mathcal{N} \to \mathcal{R}^+$$
- **Definition:** We say that
$$f(n) = O(g(n))$$
if there are two positive constants c and n_0 such that
$$f(n) \le c \cdot g(n) \text{ for all } n \ge n_0.$$
- We say that:
  - "$g(n)$ is an asymptotic upper bound on $f(n)$."
  - "$f(n)$ is big-O of $g(n)$."

Some big-O examples

- **Example 1:** Show $f(n) = O(g(n))$ for
$$f(n) = 15n^2 + 7n, \qquad g(n) = \frac{1}{2}n^3.$$
  - Let $n_0 = 16$ and $c = 2$, so we have $\forall\, n \ge n_0$:
$$f(n) = 15n^2 + 7n \le 16n^2 \le n^3 = 2 \cdot \frac{1}{2}n^3 = c \cdot g(n).$$
  - For first $\le$, if $7 \le n$, then $7n \le n^2$ by multiplying both sides by n.
  - For second $\le$, if $16 \le n$, then $16n^2 \le n^3$ (mult. by n^2).
- **Example 2:** $5n^4 + 27n = O(n^4)$.
  - Take $n_0 = 1$ and $c = 32$. (Also $n_0 = 3$ and $c = 6$ works.)
  - But $5n^4 + 27n$ is not $O(n^3)$: no values for c and n_0 work.
- **Basic idea:** ignore constant factor differences:
  - $2n^3 + 52n^2 + 829n + 2193 = O(n^3)$.
  - $2 = O(1)$ and $\sin(n) + 3 = O(1)$.

Polynomials vs Exponentials

- For a polynomial
$$p(n) = a_1 n^{k_1} + a_2 n^{k_2} + \cdots + a_d n^{k_d},$$
where $k_1 > k_2 > \cdots > k_d \ge 0$, then
  - $p(n) = O(n^{k_1})$.
  - Also, $p(n) = O(n^r)$ for all $r \ge k_1$, e.g., $7n^3 + 5n^2 = O(n^4)$.
- Exponential fcns like 2^n always eventually "overpower" polynomials.
  - For all constants a and k, polynomial $f(n) = a \cdot n^k + \cdots$ obeys:
$$f(n) = O(2^n).$$
  - For functions in n, we have
$$n^k = O(b^n)$$
for all positive constants k, and $b > 1$.

Big-O for Logarithms

- Let $\log_b$ denote logarithm with base b.
- Recall $c = \log_b n$ if $b^c = n$.
- $\log_b(x^y) = y \log_b x$ because $x = b^{\log_b x}$ and
$$b^{y \log_b x} = (b^{\log_b x})^y = x^y$$
- Note that $n = 2^{\log_2 n}$ and $\log_b(x^y) = y \log_b x$ imply
$$\log_b n = \log_b(2^{\log_2 n}) = (\log_2 n)(\log_b 2)$$
  - Changing base b changes value by only constant factor.
  - So when we say $f(n) = O(\log n)$, the base is unimportant.
- Note that $\log n = O(n)$.
- In fact, $\log n = O(n^d)$ for any $d > 0$.
  - Polynomials overpower logarithms,
    just like exponentials overpower polynomials.
- Thus, $n \log n = O(n^2)$.

Big-O Properties

- $O(n^2) + O(n) = O(n^2)$ and $O(n^2)O(n) = O(n^3)$
- Sometimes we have
$$f(n) = 2^{O(n)}.$$
What does this mean?
  - Answer: $f(n)$ has an asymptotic upper bound of 2^{cn} for some constant c.
- What does $f(n) = 2^{O(\log n)}$ mean?
  - Recall the identities:
$$n = 2^{\log_2 n},$$
$$n^c = 2^{c \log_2 n} = 2^{O(\log_2 n)}.$$
  - Thus, $2^{O(\log n)}$ means an upper bound of n^c for some constant c.

More Remarks

- **Definition:**
  - A bound of n^c, where $c > 0$ is a constant, is called **polynomial**.
  - A bound of $2^{(n^\delta)}$, where $\delta > 0$ is a constant, is called **exponential**.
- $f(n) = O(f(n))$ for all functions f.
- $[\log(n)]^k = O(n)$ for all constants k.
- $n^k = O(2^n)$ for all constants k.
- Because $n = 2^{\log_2 n}$, n is an exponential function of $\log n$.
- If $f(n)$ and $g(m)$ are polynomials, then $g(f(n))$ is polynomial in n.
  - **Example:** If $f(n) = n^2$ and $g(m) = m^3$, then
$$g(f(n)) = g(n^2) = (n^2)^3 = n^6.$$

Little-o Notation

Definition:

- Let f and g be two functions with $f, g : \mathcal{N} \to \mathcal{R}^+$.
- Then $f(n) = o(g(n))$ if
$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0.$$

Example: If

- $f(n) = 10n^2$
- $g(n) = 2n^3$

then $f(n) = o(g(n))$ because
$$\frac{f(n)}{g(n)} = \frac{10n^2}{2n^3} = \frac{5}{n} \to 0 \quad \text{as} \quad n \to \infty$$

Remarks

- Big-O notation is about "asymptotically less than or equal to".
- Little-o is about "asymptotically much smaller than".
- Make it clear whether you mean $O(g(n))$ or $o(g(n))$.
- Make it clear which variable the function is in:
  - $O(x^y)$ can be a polynomial in x or an exponential in y.
- Simplify!
  - Rather than $O(8n^3 + 2n)$, instead use $O(n^3)$.
- Try to keep your big-O as "tight" as possible.
  - Suppose $f(n) = 2n^3 + 8n^2$.
  - Although $f(n) = O(n^5)$, better to write $f(n) = O(n^3)$.

Back to Example of TM M_1 for $A = \{ 0^k 1^k \mid k \geq 0 \}$

$M_1 =$ "On input string $w \in \{0, 1\}^*$:

1. Scan across tape and *reject* if 0 is found to the right of a 1.
2. Repeat the following if both 0s and 1s appear on tape:
   - Scan across tape, crossing off single 0 and single 1.
3. If no 0s or 1s remain, *accept*;
   otherwise, *reject*."

Let's now analyze M_1's run-time complexity.

- We will examine each stage separately.
- Suppose input string w is of length n.

0	0	0	1	1	1	⊔	⊔	$\cdots$

Analysis of Stage 1

1. Scan across tape and *reject* if 0 is found to the right of a 1.

0	0	0	1	1	1	⊔	⊔	$\cdots$

Analysis:

- Input string w is of length n.
- Scanning requires n steps.
- Repositioning head back to beginning of tape requires n steps.
- Total is $2n = O(n)$ steps.

Analysis of Stage 2

2. Repeat the following if both 0s and 1s appear on tape:
   - Scan across tape, crossing off single 0 and single 1.

0	0	0	1	1	1	⊔	⊔	$\cdots$

Analysis:

- Each scan requires $O(n)$ steps.
- Because each scan crosses off two symbols,
  - at most $n/2$ scans can occur.
- Total is $O(\frac{n}{2})\, O(n) = O(n^2)$ steps.

Analysis of Stage 3 and Overall

3. If no 0s or 1s remain, *accept*;
   otherwise, *reject*.

$\cancel{0}$	$\cancel{0}$	$\cancel{0}$	$\cancel{1}$	$\cancel{1}$	$\cancel{1}$	⊔	⊔	$\cdots$

Analysis:

- Single scan requires $O(n)$ steps.

Total cost for each stage:

- Stage 1: $O(n)$
- Stage 2: $O(n^2)$
- Stage 3: $O(n)$

Overall complexity: $O(n) + O(n^2) + O(n) = O(n^2)$

Time Complexity Class

Definition: For a function $t : \mathcal{N} \to \mathcal{N}$,

$$\text{TIME}(t(n)) = \{\, L \mid \text{there is a 1-tape TM that decides language } L \text{ in time } O(t(n)) \,\}$$

Remarks:

- TM M_1 decides language $A = \{\, 0^k 1^k \mid k \geq 0 \,\}$
  - M_1 has run-time complexity $O(n^2)$.
- Thus, $A \in \text{TIME}(n^2)$.
- Can we do better?

Another TM for $A = \{\, 0^k 1^k \mid k \geq 0 \,\}$

M_2 = "On input string w:

1. Scan across tape and *reject* if 0 is found to the right of a 1.
2. Repeat the following if both 0s and 1s appear on tape:

   2.1 Scan across tape, checking whether total number of 0s and 1s is even or odd. If odd, *reject*.

   2.2 Scan across tape, crossing off every other 0 (starting with the leftmost), and every other 1 (starting with the leftmost).

3. If no 0s or 1s remain, *accept*;
   otherwise, *reject*."

Why M_2 Halts

- Stage 2.2: Scan across tape, crossing every other 0 and 1.
- On each scan in Stage 2.2,
  - Total number of 0s is decreased by (at least) half
  - Same for the 1s
- **Example:**
  - Start with 13 0s.

0	0	0	0	0	0	0	0	0	0	0	0	0

  - After first pass, 6 remaining.

$\cancel{0}$	0	$\cancel{0}$	0	$\cancel{0}$	0	$\cancel{0}$	0	$\cancel{0}$	0	$\cancel{0}$	0	$\cancel{0}$

  - After second pass, 3 remaining.

$\cancel{0}$	$\cancel{0}$	$\cancel{0}$	0	$\cancel{0}$	$\cancel{0}$	$\cancel{0}$	0	$\cancel{0}$	$\cancel{0}$	$\cancel{0}$	0	$\cancel{0}$

  - After third pass, 1 remaining.

$\cancel{0}$	$\cancel{0}$	$\cancel{0}$	$\cancel{0}$	$\cancel{0}$	$\cancel{0}$	$\cancel{0}$	0	$\cancel{0}$	$\cancel{0}$	$\cancel{0}$	$\cancel{0}$	$\cancel{0}$

  - After fourth pass, none remaining.

$\cancel{0}$	$\cancel{0}$	$\cancel{0}$	$\cancel{0}$	$\cancel{0}$	$\cancel{0}$	$\cancel{0}$	$\cancel{0}$	$\cancel{0}$	$\cancel{0}$	$\cancel{0}$	$\cancel{0}$	$\cancel{0}$

Why M_2 Works

- Consider parity of 0s and 1s in Stage 2.1.
- Example: Start with $0^{13}\,1^{13}$
  - Initially, odd-odd (13, 13)

0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1

  - Then, even-even (6, 6)

$\cancel{0}$	0	$\cancel{0}$	0	$\cancel{0}$	0	$\cancel{0}$	0	$\cancel{0}$	0	$\cancel{0}$	0	$\cancel{0}$	$\cancel{1}$	1	$\cancel{1}$	1	$\cancel{1}$	1	$\cancel{1}$	1	$\cancel{1}$	1	$\cancel{1}$	1	$\cancel{1}$

  - Then, odd-odd (3, 3)

$\cancel{0}$	$\cancel{0}$	$\cancel{0}$	0	$\cancel{0}$	$\cancel{0}$	$\cancel{0}$	0	$\cancel{0}$	$\cancel{0}$	$\cancel{0}$	0	$\cancel{0}$	$\cancel{1}$	$\cancel{1}$	$\cancel{1}$	1	$\cancel{1}$	$\cancel{1}$	$\cancel{1}$	1	$\cancel{1}$	$\cancel{1}$	$\cancel{1}$	1	$\cancel{1}$

  - Then, odd-odd (1, 1)

$\cancel{0}$	$\cancel{0}$	$\cancel{0}$	$\cancel{0}$	$\cancel{0}$	$\cancel{0}$	$\cancel{0}$	0	$\cancel{0}$	$\cancel{0}$	$\cancel{0}$	$\cancel{0}$	$\cancel{0}$	$\cancel{1}$	$\cancel{1}$	$\cancel{1}$	$\cancel{1}$	$\cancel{1}$	$\cancel{1}$	$\cancel{1}$	1	$\cancel{1}$	$\cancel{1}$	$\cancel{1}$	$\cancel{1}$	$\cancel{1}$

- Result is 1011, which is reverse of binary representation of 13.
- Each pass checks one binary digit.

$M_2 =$ "On input string w:

1. Scan across tape and *reject* if 0 is found to the right of a 1.
2. Repeat the following if both 0s and 1s appear on tape:
   2.1 Scan across tape, checking whether total number of 0s and 1s is even or odd. If odd, *reject*.
   2.2 Scan across tape, crossing off every other 0 (starting with the leftmost), and every other 1 (starting with the leftmost).
3. If no 0s or 1s remain, *accept*; otherwise, *reject*."

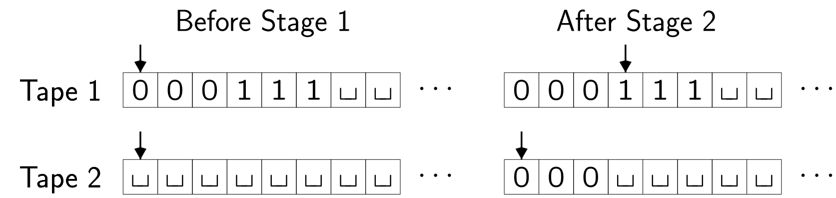
Analysis:

- Each stage requires $O(n)$ time.
- Stage 1 and 3 run once each.
- Stage 2.2 eliminates half of 0s and 1s: Stage 2 runs $O(\log_2 n)$ times.
- Total for stage 2 is $O(\log_2 n)O(n) = O(n \log n)$.
- Grand total: $O(n) + O(n \log n) = O(n \log n)$, so language $A \in \text{TIME}(n \log n)$.

2-Tape TM for $A = \{ 0^k 1^k \mid k \geq 0 \}$

$M_3 =$ "On input string w:

1. Scan across tape and *reject* if 0 is found to the right of a 1.
2. Scan across 0s to first 1, copying 0s to tape 2.
3. Scan across 1s on tape 1 until the end.
   For each 1 on tape 1, cross off a 0 on tape 2.
   If no 0s left, *reject*.
4. If any 0s left, *reject*; otherwise, *accept*."



Can show that running time of M_3 is $O(n)$.

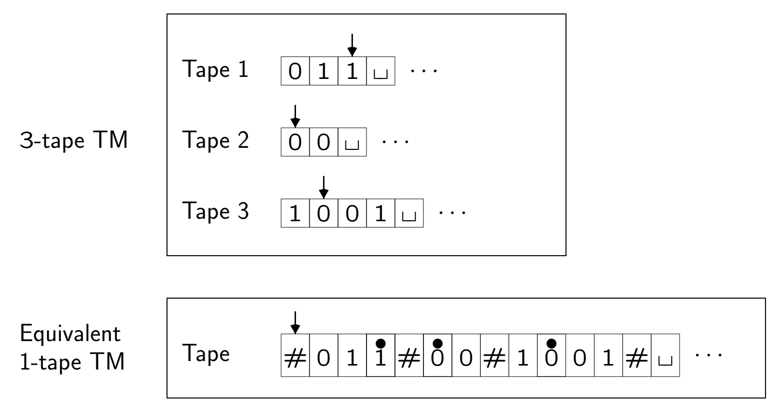
Runtimes of TMs for $A = \{ 0^k 1^k \mid k \geq 0 \}$

- Runtime depends on computational model:
  - 1-tape TM M_1: $O(n^2)$
  - 1-tape TM M_2: $O(n \log n)$
  - 2-tape TM M_3: $O(n)$.
- For **computability**, all reasonable computational models are equivalent (Church-Turing Thesis).
- For **complexity**, choice of computational model affects time complexity.

k-Tape TM can be Simulated on 1-Tape TM with Polynomial Overhead

Theorem 7.8

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



Review Thm 3.13: Simulating k-Tape TM M on 1-Tape TM S

On input $w = w_1 \cdots w_n$, the 1-tape TM S does the following:

- First S prepares initial string on single tape:

$\#$	$\dot{w}_1$	w_2	$\cdots$	w_n	$\#$	$\dot{\sqcup}$	$\#$	$\dot{\sqcup}$	$\#$	$\sqcup$	$\sqcup$	$\cdots$

- For each step of M, TM S scans tape twice
  1. Scans its tape from
     - first $\#$ (which marks left end of tape) to
     - $(k+1)$st $\#$ (which marks right end of tape)

     to read symbols under "virtual" heads
  2. Rescans to write new symbols and move heads
     - If S tries to move virtual head to the right onto $\#$, then
       - M is trying to move head onto unused blank cell.
       - So S has to write blank on tape and shift rest of tape right one cell.

Complexity of Simulation

- For each step of k-tape TM M, 1-tape TM S performs two scans
  - Length of active portion of S's tape determines how long S takes to perform each scan.
  - In r steps, TM M can read/write in $\leq k \times r$ different cells on its k tapes.
  - As M has $t(n)$ runtime, at any point during M's execution, total # active cells on all of M's tapes $\leq k \times t(n) = O(t(n))$.
  - Thus, each of S's scans requires $O(t(n))$ time.
- Overall runtime of S
  - Initial tape arrangement: $O(n)$ steps.
  - S simulates each of M's $t(n)$ steps using $O(t(n))$ steps.
    - Thus, total of $t(n) \times O(t(n)) = O(t^2(n))$ steps.
  - **Grand total**: $O(n) + O(t^2(n)) = O(t^2(n))$ steps.

Running Time of Nondeterministic TMs

- What about nondeterministic TMs (NTMs)?
- Informally, NTM makes "lucky guesses" during computation.
- In terms of **computability**, no difference between TMs and NTMs.
- For **time-complexity**, nondeterminism **seems** to make big difference.

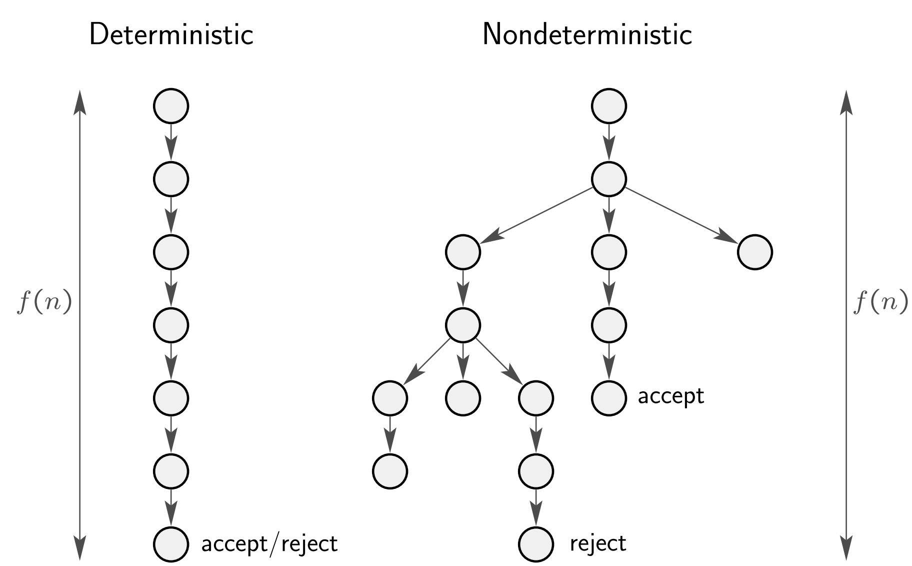
Definition:

- Let N be NTM that is a decider (no looping).
- **Running time** of NTM N is function $f : \mathcal{N} \to \mathcal{N}$, where

$$f(n) = \max_{|x|=n} (\text{ height of tree of configs for } N \text{ on input } x\ )$$

  - the maximum number of steps that NTM N uses
  - on any branch of the computation
  - on any input x of size n.

Deterministic vs. Nondeterministic TM Runtime



Simulating NTM N on 1-Tape DTM D Requires Exponential Overhead

Theorem 7.11

- Let $t(n)$ be a function with $t(n) \geq n$.
- Any $t(n)$-time nondeterministic TM has an equivalent $2^{O(t(n))}$-time deterministic 1-tape TM.

Proof Idea:

- Suppose N is NTM decider running in $t(n)$ time.
- On each input w, NTM N's computation is a tree of configurations.
- Simulate N on 3-tape DTM D using BFS of N's computation tree:
  - D tries all possible branches.
  - If D finds any accepting configuration, D accepts.
  - If all branches reject, D rejects.

Complexity of Simulating NTM N on 1-Tape DTM D

- Analyze NTM N's computation tree on input w with $|w| = n$
  - Root is starting configuration.
  - Each node has $\leq b$ children
    - $b =$ max number of legal choices given by N's transition fcn δ.
  - Each branch has length $\leq t(n)$.
  - Total number of leaves $\leq b^{t(n)}$.
  - Total number of nodes $\leq 2 \times$ (max number of leaves) $= O(b^{t(n)})$.
  - Time to travel from root to any node is $O(t(n))$.
- DTM's runtime $\leq$ time to visit all nodes:

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

- Simulating NTM by DTM requires 3 tapes by Theorem 3.16.
- By Theorem 3.13, simulating 3-tape DTM on 1-tape DTM requires $(2^{O(t(n))})^2 = 2^{2 \times O(t(n))} = 2^{O(t(n))}$ steps.

Summary of Simulation Results

- Simulating k-tape DTM on 1-tape DTM
  - increases runtime from $t(n)$ to $O(t^2(n))$
  - i.e., **polynomial** increase in runtime.
- Simulating NTM on 1-tape DTM
  - increases runtime from $t(n)$ to $2^{O(t(n))}$
  - i.e., **exponential** increase in runtime.

Polynomial Good, Exponential Bad

10^6 steps/second

	n					
$f(n)$	10	20	30	40	50	60
n	.00001 seconds	.00002 seconds	.00003 seconds	.00004 seconds	.00005 seconds	.00006 seconds
n^2	.0001 seconds	.0004 seconds	.0009 seconds	.0016 seconds	.0025 seconds	.0036 seconds
n^3	.001 seconds	.008 seconds	.027 seconds	.064 seconds	.125 seconds	.216 seconds
n^5	.1 seconds	3.2 seconds	24.3 seconds	1.7 minutes	5.2 minutes	13 minutes
2^n	.001 seconds	1.05 seconds	17.9 minutes	12.7 days	35.7 years	366 centuries
3^n	.059 seconds	58 minutes	6.5 years	3855 centuries	2×10^8 centuries	10^{13} centuries

Strong Church-Turing Thesis

- In general, every "reasonable" variant of DTM (k-tape, r-heads, etc.) can be simulated by a single-tape DTM with only **polynomial time/space overhead**.
  - Any one of these models can simulate another with only polynomial increase in running time or space required.
  - All "reasonable" models of computation are polynomially equivalent.
  - NTM is "unreasonable" variant: it can do $O(b^s)$ work on step s.
- If any version of a DTM can solve a problem in polynomial time, then any other type of DTM can also.
- If we ask if a particular problem is solvable in **linear time** (i.e., $O(n)$), answer **depends** on computational model used.
- If we ask if a particular problem A is solvable in **polynomial time**, answer is **independent** of deterministic computational model used.

The Class P

Because of polynomial equivalence of DTM models,

- group languages solvable in $O(n^2)$, $O(n \log n)$, $O(n)$, etc., together in the **polynomial-time class**.

Definition: The class of languages that can be decided by a single-tape DTM in polynomial time is denoted by P, where

$$\mathrm{P} = \bigcup_{k \geq 0} \mathrm{TIME}(n^k).$$

Remarks:

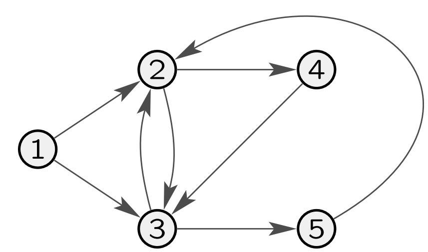
- If we ask if a particular problem A is solvable in polynomial time (i.e., is $A \in \mathrm{P}$?),
  - answer is independent of deterministic computational model used.
- Class P roughly corresponds to *tractable* (i.e., realistically solvable) problems.

Encoding of Problems

- Recall: TM running time defined as fcn of length of encoding $\langle x \rangle$ of input x.
- But for given problem, many ways to encode input x as $\langle x \rangle$.
- For integers
  - binary is good
  - unary is bad (exponentially worse)
  - **Example:** Suppose input to TM is the number 18 in decimal.
    - if encoding in binary, $\langle 18 \rangle = 10010$
    - if encoding in unary, $\langle 18 \rangle = 111111111111111111$
- For graphs
  - list of nodes and edges (good)
  - adjacency matrix (good)

Example of Problem in P: PATH

- **Decision problem**: Given directed graph G with nodes s and t, does G have a path from s to t?



- **Universe** $\Omega = \{ \langle G, s, t \rangle \mid G \text{ is directed graph with nodes } s, t \}$ of instances (for a particular encoding scheme).
- **Language** of decision problem comprises YES instances:

  PATH $= \{ \langle G, s, t \rangle \mid G \text{ is directed graph with path from } s \text{ to } t \} \subseteq \Omega$.
- For graph G above, $\langle G, 1, 5 \rangle \in$ *PATH*, but $\langle G, 2, 1 \rangle \notin$ *PATH*.

PATH $\in \mathrm{P}$

Theorem 7.14
PATH $\in \mathrm{P}$.

Brute-force algorithm:

- Input is instance $\langle G, s, t\rangle \in \Omega$
  - G is directed graph with nodes s and t.
- Let m be number of nodes in G.
  - $\le m^2$ edges.
  - m (or m^2) roughly measures **size** of instance $\langle G, s, t\rangle$.
- Any path from s to t need not repeat nodes.
- Examine each potential path in G of length $\le m$.
  - Check if the path goes from s to t.

What is complexity of this algorithm?

Complexity of Brute-Force Algorithm for PATH

Brute-force algorithm:

- Input is $\langle G, s, t\rangle \in \Omega$, where G is directed graph with nodes s and t.
- Any path from s to t need not repeat nodes.
- Examine each potential path in G of length $\le m$ ($=$ # nodes in G).
  - Check if the path goes from s to t.

Complexity analysis:

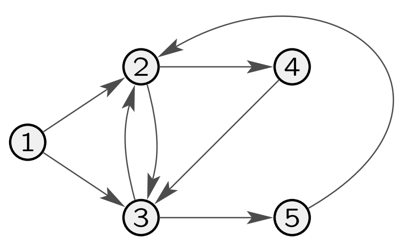
- There are roughly m^m potential paths of length $\le m$.
  - For each potential path length $k = 2, 3, \ldots, m$, check all $k!$ permutations of k distinct nodes from $\binom{m}{k}$ possibilities.
  - $k! = k \times (k-1) \times (k-2) \times \cdots \times 1, \quad \binom{m}{k} = \frac{m!}{k!(m-k)!}$
  - Stirling's approximation: $k! \sim \left(\frac{k}{e}\right)^k \sqrt{2\pi k}$.
- This is exponential in the number m of nodes.
- So brute-force algorithm's runtime is **exponential** in size of input.

A Better Algorithm Shows PATH $\in \mathrm{P}$

On input $\langle G, s, t\rangle \in \Omega$, where G is directed graph with nodes s and t:

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

Graph G



$\langle G, 1, 5\rangle \in$ *PATH*

$\langle G, 5, 3\rangle \in$ *PATH*

$\langle G, 2, 1\rangle \notin$ *PATH*

Complexity of Better Algorithm for PATH

On input $\langle G, s, t\rangle \in \Omega$, where G is a directed graph with nodes s and t:

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

Complexity of algorithm: (depends on how $\langle G, s, t\rangle$ is encoded)

- Suppose input graph G has m nodes, so $\le m^2$ edges.
- Stage 1 runs only once, running in $O(m)$ time
- Stage 2 runs at most m times
  - Each time (except last), it marks new nodes.
  - Each time requires scanning edges, which runs in $O(m^2)$ steps.
- Stage 3 runs only once, running in $O(m)$ time
- **Overall complexity:** $O(m) + O(m)O(m^2) + O(m) = O(m^3)$, so *PATH* $\in \mathrm{P}$.

Another Problem in P: RELPRIME

- **Definition:** Two integers x, y are **relatively prime** if 1 is largest integer that divides both; greatest common divisor $\text{GCD}(x, y) = 1$.
- **Examples:**
  - 10 and 21 are relatively prime.
  - 10 and 25 are not.
- **Decision problem**: Given integers x and y, are x, y relatively prime?
  - **Universe** $\Omega = \{ \langle x, y \rangle \mid x, y \text{ integers} \}$ of problem instances.
  - **Language** of decision problem:

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

  - So $\langle 10, 21 \rangle \in \textit{RELPRIME}$ and $\langle 10, 25 \rangle \notin \textit{RELPRIME}$.

Theorem 7.15
RELPRIME $\in \mathrm{P}$.

Bad Algorithm for RELPRIME

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

Bad Idea: Test all possible divisors (i.e., 2 to $\min(x, y)$).

Complexity of algorithm depends on how integers are encoded:

- If x, y encoded in **unary**, then
  - length of $\langle x \rangle$ is x; length of $\langle y \rangle$ is y.
  - testing $\min(x, y)$ values is **polynomial** in length of input $\langle x, y \rangle$.
- If x, y encoded in **binary**, then
  - length of $\langle x \rangle$ is $\log x$; length of $\langle y \rangle$ is $\log y$.
  - testing $\min(x, y)$ values is exponential in length of input $\langle x, y \rangle$ because n is an **exponential** function of $\log n$ (i.e., $n = 2^{\log_2 n}$).
- This algorithm is **pseudo-polynomial**.

A Better Algorithm for RELPRIME

Euclidean Algorithm E:

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

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

Algorithm R below solves *RELPRIME*, using E as a subroutine:

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

1. Run E on $\langle x, y \rangle$.
2. If output of E is 1, *accept*;
   otherwise, *reject*."

Complexity of Euclidean Algorithm

Euclidean Algorithm E:

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

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

Complexity of E:

- After first step of Stage 1, $x < y$ because of mod.
- Values then swapped, so $x > y$.
- Can show each subsequent execution of Stage 1 cuts x by at least half.
- # times Stage 1 executed $\leq \min(\log_2 x, \ \log_2 y)$.
- Thus, total running time of E (and R) is polynomial in $|\langle x, y \rangle|$, so *RELPRIME* $\in \mathrm{P}$.

CFLs are in P

Theorem 7.16
Every context-free language is in P.

Remarks:

- Will show that each CFL $\in$ TIME(n^3)
  - n is length of input string $w \in \Sigma^*$.
  - In contrast, each regular language $\in$ TIME(n). Why?
- Theorem 4.9 showed that every CFL is decidable, which we now review.
- Convert CFG into **Chomsky normal form**:
  - Each rule has one of the following forms:

$$A \to BC, \qquad A \to x, \qquad S \to \varepsilon$$

where A, B, C, S are variables; S is start variable; B, C are not start variable; x is a terminal.

Recall Previous Algorithm to Decide CFL

Lemma
If G is in Chomsky normal form and w is string with length $n > 0$, then w has a derivation with $2n - 1$ steps.

Theorem 4.9
Every CFL is a decidable language.

Proof.

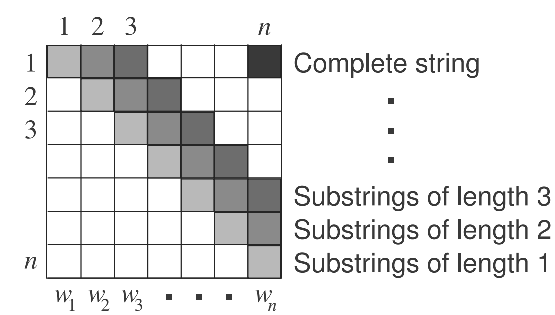
- Assume L is a CFL generated by CFG G in Chomsky normal form.
- Theorem 4.7: $\exists$ TM S that decides $A_{\text{CFG}} = \{ \langle G, w \rangle \mid G \text{ is a CFG that generates } w \}$.
- Following TM M_G decides CFL L:
  M_G = "On input w:
  1. Run TM S on input $\langle G, w \rangle$.
  2. If S accepts, *accept*; if S rejects, *reject*."

Previous Algorithm is Exponential

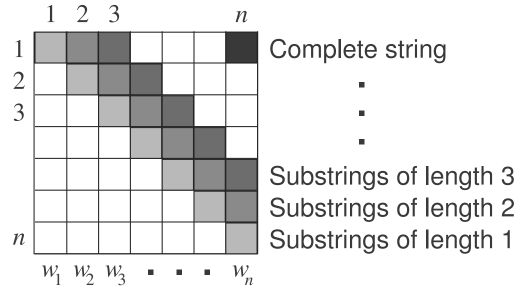
- Recall that to determine if $\langle G, w \rangle \in A_{\text{CFG}}$, TM S tries all derivations with $k = 2n - 1$ steps, where $n = |w|$.
  - But number of derivations taking k steps can be **exponential** in k.
  - So we need to use a different algorithm.
- Use **dynamic programming (DP)**
  - Powerful, general technique.
  - **Basic idea**: accumulate information about *smaller subproblems* to solve *larger subproblems*.
  - Store subproblem solutions in a *table* as they are generated.
  - Look up smaller subproblem solutions as needed when solving larger subproblems.
  - DP for CFGs: **Cocke-Younger-Kasami (CYK)** algorithm.

Dynamic Programming

- Fix CFG G in **Chomsky normal form**.
- Input to DP algorithm is string $w = w_1 w_2 \cdots w_n$ with $|w| = n$
- In our case of DP, **subproblems** are to determine which variables in G can generate each **substring** of w.
- Create an $n \times n$ table



Dynamic Programming Table



- For $i \leq j$, (i, j)th entry contains those variables that can generate substring $w_i\, w_{i+1} \cdots w_j$
- For $i > j$, (i, j)th entry is unused.
- DP starts by filling in all entries for substrings of length 1,
  then all entries for length 2,
  then all entries for length 3, etc.
- **Idea:** *Use shorter lengths to determine how to construct longer lengths.*

Filling in Dynamic Programming Table

- Suppose $s = uv$, $B \overset{*}{\Rightarrow} u$, $C \overset{*}{\Rightarrow} v$, and $\exists$ rule $A \to BC$.
  - Then $A \overset{*}{\Rightarrow} s$ because $A \Rightarrow BC \overset{*}{\Rightarrow} uv = s$.
- Suppose that algorithm has determined which variables generate each substring of length $\leq k$.
- To determine if variable A can generate substring of length $k+1$:
  - split substring into 2 non-empty pieces in all possible (k) ways.
  - For each split, algorithm examines rules $A \to BC$
    - Each piece is shorter than current substring,
      so table tells how to generate each piece.
    - Check if B generates first piece.
    - Check if C generates second piece.
    - If both possible, then add A to table.

Example: CYK Algorithm

Does the following CFG in Chomsky Normal Form generate $baaba$?

$$\begin{array}{lll} S \to XY \mid YZ & \quad & X \to YX \mid a \\ Y \to ZZ \mid b & \quad & Z \to XY \mid a \end{array}$$

	1	2	3	4	5
1					
2					
3					
4					
5					
string	b	a	a	b	a

- Build table t so that for $i \leq j$, entry $t(i, j)$ contains variables that can generate substring starting in position i and ending in position j
- Fill in one diagonal at a time.

Ex. (cont.): CYK for Substrings of Length 1

Chomsky CFG:

$$\begin{array}{lll} S \to XY \mid YZ & \quad & X \to YX \mid a \\ Y \to ZZ \mid b & \quad & Z \to XY \mid a \end{array}$$

	1	2	3	4	5
1	Y				
2					
3					
4					
5					
string	b	a	a	b	a

- $t(1, 1)$: substring b starts in position 1 and ends in position 1.
  - CFG has rule $Y \to b$, so put Y in $t(1, 1)$.

Ex. (cont.): CYK for Substrings of Length 1

Chomsky CFG: $S \to XY \mid YZ$ $\quad X \to YX \mid a$
$Y \to ZZ \mid b$ $\quad Z \to XY \mid a$

	1	2	3	4	5
1	Y				
2		X, Z			
3			X, Z		
4				Y	
5					X, Z
string	b	a	a	b	a

- $t(2,2)$: substring a starts in position 2 and ends in position 2.
  - CFG has rules $X \to a$ and $Z \to a$, so put X, Z in $t(2,2)$.
- Similarly fill in other $t(i,i)$.

Ex. (cont.): CYK for Substrings of Length 2

Chomsky CFG: $S \to XY \mid YZ$ $\quad X \to YX \mid a$
$Y \to ZZ \mid b$ $\quad Z \to XY \mid a$

	1	2	3	4	5
1	Y	S, X			
2		X, Z			
3			X, Z		
4				Y	
5					X, Z
string	b	a	a	b	a

- $t(1,2)$: substring ba starts in position 1 and ends in position 2.
  - split $ba = b\,a$:

    $$Y \overset{*}{\Rightarrow} b \text{ by } t(1,1); \qquad X, Z \overset{*}{\Rightarrow} a \text{ by } t(2,2).$$
  - If a rule has YX or YZ on RHS, then LHS $\overset{*}{\Rightarrow} ba$:

    $$X \Rightarrow YX \overset{*}{\Rightarrow} ba, \qquad S \Rightarrow YZ \overset{*}{\Rightarrow} ba$$

Ex. (cont.): CYK for Substrings of Length 2

Chomsky CFG: $S \to XY \mid YZ$ $\quad X \to YX \mid a$
$Y \to ZZ \mid b$ $\quad Z \to XY \mid a$

	1	2	3	4	5
1	Y	S, X			
2		X, Z	Y		
3			X, Z		
4				Y	
5					X, Z
string	b	a	a	b	a

- $t(2,3)$: substring aa starts in position 2 and ends in position 3.
  - split $aa = a\,a$:

    $$X, Z \overset{*}{\Rightarrow} a \text{ by } t(2,2); \qquad X, Z \overset{*}{\Rightarrow} a \text{ by } t(3,3).$$
  - If rule has XX, XZ, ZX, or ZZ on RHS, then LHS $\overset{*}{\Rightarrow} aa$:

    $$Y \Rightarrow ZZ \overset{*}{\Rightarrow} aa$$

Ex. (cont.): CYK for Substrings of Length 2

Chomsky CFG: $S \to XY \mid YZ$ $\quad X \to YX \mid a$
$Y \to ZZ \mid b$ $\quad Z \to XY \mid a$

	1	2	3	4	5
1	Y	S, X			
2		X, Z	Y		
3			X, Z	S, Z	
4				Y	S, X
5					X, Z
string	b	a	a	b	a

- $t(3,4)$: substring ab starts in position 3 and ends in position 4.
  - split $ab = a\,b$: $\quad X, Z \overset{*}{\Rightarrow} a$ by $t(3,3)$; $\quad Y \overset{*}{\Rightarrow} b$ by $t(4,4)$.
  - If rule has XY or ZY on RHS, then LHS $\overset{*}{\Rightarrow} ab$:

    $$S \Rightarrow XY \overset{*}{\Rightarrow} ab, \qquad Z \Rightarrow XY \overset{*}{\Rightarrow} ab$$
- $t(4,5)$: similarly handle substring ba ...

Ex. (cont.): CYK for Substrings of Length 3

Chomsky CFG: $S \rightarrow XY \mid YZ$ $X \rightarrow YX \mid a$
$Y \rightarrow ZZ \mid b$ $Z \rightarrow XY \mid a$

	1	2	3	4	5
1	Y	S, X	—		
2		X, Z	Y		
3			X, Z	S, Z	
4				Y	S, X
5					X, Z
string	b	a	a	b	a

- $t(1,3)$: substring baa starts in position 1 and ends in position 3.
  - split $baa = b\,aa$: $Y \stackrel{*}{\Rightarrow} b$ by $t(1,1)$; $Y \stackrel{*}{\Rightarrow} aa$ by $t(2,3)$; so if rule has RHS YY, then LHS $\stackrel{*}{\Rightarrow} baa$.
  - split $baa = ba\,a$: $S, X \stackrel{*}{\Rightarrow} ba$ by $t(1,2)$; $X, Z \stackrel{*}{\Rightarrow} a$ by $t(3,3)$; so if rule has RHS SX, SZ, XX, or XZ, then LHS $\stackrel{*}{\Rightarrow} baa$.

Ex. (cont.): CYK for Substrings of Length 3

Chomsky CFG: $S \rightarrow XY \mid YZ$ $X \rightarrow YX \mid a$
$Y \rightarrow ZZ \mid b$ $Z \rightarrow XY \mid a$

	1	2	3	4	5
1	Y	S, X	—		
2		X, Z	Y	Y	
3			X, Z	S, Z	Y
4				Y	S, X
5					X, Z
string	b	a	a	b	a

- $t(2,4)$: substring aab starts in position 2 and ends in position 4.
  - split $aab = a\,ab$: $X, Z \stackrel{*}{\Rightarrow} a$ by $t(2,2)$; $S, Z \stackrel{*}{\Rightarrow} ab$ by $t(3,4)$; so if rule has RHS XS, XZ, ZS, or ZZ, then LHS $\stackrel{*}{\Rightarrow} aab$:
  $$Y \Rightarrow ZZ \stackrel{*}{\Rightarrow} a\,ab$$
  - split $aab = aa\,b$: $Y \stackrel{*}{\Rightarrow} aa$ by $t(2,3)$; $Y \stackrel{*}{\Rightarrow} b$ by $t(4,4)$; so if rule has RHS YY, then LHS $\stackrel{*}{\Rightarrow} aab$.

Ex. (cont.): CYK for Substrings of Length 4

Chomsky CFG: $S \rightarrow XY \mid YZ$ $X \rightarrow YX \mid a$
$Y \rightarrow ZZ \mid b$ $Z \rightarrow XY \mid a$

	1	2	3	4	5
1	Y	S, X	—	—	
2		X, Z	Y	Y	
3			X, Z	S, Z	Y
4				Y	S, X
5					X, Z
string	b	a	a	b	a

- $t(1,4)$: substring $baab$ starts in position 1 and ends in position 4.
  - split $b\,aab$: $Y \stackrel{*}{\Rightarrow} b$ by $t(1,1)$; $Y \stackrel{*}{\Rightarrow} aab$ by $t(2,4)$; so if rule has RHS YY, then LHS $\stackrel{*}{\Rightarrow} baab$.
  - split $ba\,ab$: $S, X \stackrel{*}{\Rightarrow} ba$ by $t(1,2)$; $S, Z \stackrel{*}{\Rightarrow} ab$ by $t(3,4)$; so if rule has RHS SS, SZ, XS, or XZ, then LHS $\stackrel{*}{\Rightarrow} baab$.
  - split $baa\,b$: Nothing $\stackrel{*}{\Rightarrow} baa$ as $t(1,3) = \emptyset$; $Y \stackrel{*}{\Rightarrow} b$ by $t(4,4)$.

Ex. (cont.): CYK for Substrings of Length 4

Chomsky CFG: $S \rightarrow XY \mid YZ$ $X \rightarrow YX \mid a$
$Y \rightarrow ZZ \mid b$ $Z \rightarrow XY \mid a$

	1	2	3	4	5
1	Y	S, X	—	—	
2		X, Z	Y	Y	S, X, Z
3			X, Z	S, Z	Y
4				Y	S, X
5					X, Z
string	b	a	a	b	a

- $t(2,5)$: substring $aaba$ starts in position 2 and ends in position 5.
  - split $a\,aba$: $X, Z \stackrel{*}{\Rightarrow} a$ by $t(2,2)$; $Y \stackrel{*}{\Rightarrow} aba$ by $t(3,5)$; so if rule has RHS XY or ZY, then LHS $\stackrel{*}{\Rightarrow} aaba$:
  $$S \Rightarrow XY \stackrel{*}{\Rightarrow} a\,aba, \qquad Z \Rightarrow XY \stackrel{*}{\Rightarrow} a\,aba$$
  - split $aa\,ba$: $Y \stackrel{*}{\Rightarrow} aa$ by $t(2,3)$; $S, X \stackrel{*}{\Rightarrow} ba$ by $t(4,5)$; so if rule has RHS YS or YX, then LHS $\stackrel{*}{\Rightarrow} aaba$:
  $$X \Rightarrow YX \stackrel{*}{\Rightarrow} aa\,ba$$
  - split $aab\,a$: $Y \stackrel{*}{\Rightarrow} aab$ by $t(2,4)$; $X, Z \stackrel{*}{\Rightarrow} a$ by $t(5,5)$; so if rule has RHS YX or YZ, then LHS $\stackrel{*}{\Rightarrow} aaba$:
  $$X \Rightarrow YX \stackrel{*}{\Rightarrow} aab\,a$$

Ex. (cont.): CYK for Substrings of Length 5

Does the following CFG in Chomsky Normal Form generate $baaba$?

$$S \rightarrow XY \mid YZ \qquad X \rightarrow YX \mid a$$
$$Y \rightarrow ZZ \mid b \qquad Z \rightarrow XY \mid a$$

	1	2	3	4	5
1	Y	S, X	—	—	S, X, Z
2		X, Z	Y	Y	S, X, Z
3			X, Z	S, Z	Y
4				Y	S, X
5					X, Z
string	b	a	a	b	a

- $t(1, 5)$: substring $baaba$ starts in position 1 and ends in position 5.
- Answer is **YES** iff start variable $S \in t(1, 5)$.

Overall CYK Algorithm to show every CFL $\in \mathrm{P}$

$D =$ "On input string $w = w_1 w_2 \cdots w_n$:

1. For $w = \varepsilon$, if $S \rightarrow \varepsilon$ 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 $A \rightarrow b$ is a rule, where $b = w_i$.
5. If so, put A in *table*(i, i).
6. For $\ell = 2$ to n, [ℓ is length of substring]
7. For $i = 1$ to $n - \ell + 1$, [i is start position of substring]
8. Let $j = i + \ell - 1$, [j is end position of substring]
9. For $k = i$ to $j - 1$, [k is split position]
10. For each rule $A \rightarrow BC$,
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*."

Complexity of CYK Algorithm

- Each stage runs in polynomial time.
- Examine stages 2–5:

2. For $i = 1$ to n, [examine each substring of length 1]
3. For each variable A,
4. Test whether $A \rightarrow b$ is a rule, where $b = w_i$.
5. If so, put A in *table*(i, i).

- **Analysis:**
  - Stage 2 runs n times
  - Each time stage 2 runs, stage 3 runs v times, where
    - v is number of variables in G
    - v is independent of n.
  - Thus, stages 4 and 5 run at most nv times, which is $O(n)$ because v is independent of n.

Complexity (cont)

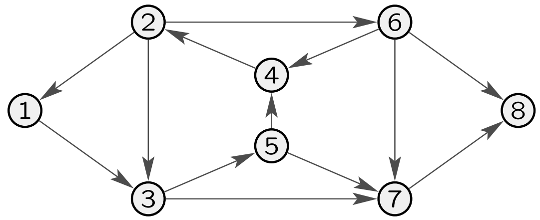
6. For $\ell = 2$ to n, [ℓ is length of substring]
7. For $i = 1$ to $n - \ell + 1$, [i is start position of substring]
8. Let $j = i + \ell - 1$, [j is end position of substring]
9. For $k = i$ to $j - 1$, [k is split position]
10. For each rule $A \rightarrow BC$,
11. If *table*(i, k) contains B and *table*$(k + 1, j)$ contains C, put A in *table*(i, j).
12. If S is in *table*$(1, n)$, *accept*. Otherwise, *reject*.

Analysis:

- Stage 6 runs at most n times
- Each time stage 6 runs, stage 7 runs at most n times
- Each time stage 7 runs, stage 9 runs at most n times
- Each time stage 9 runs, stage 10 runs r times ($r = \#$ rules $=$ constant)
- Thus, stage 8 runs $O(n^2)$ times, and stage 11 runs $O(n^3)$ times

Grand total: $O(n^3)$

Hamiltonian Path



- **Definition:** A **Hamiltonian path** in a directed graph G visits each node exactly once, e.g., $1 \to 3 \to 5 \to 4 \to 2 \to 6 \to 7 \to 8$.
- **Decision problem:** Given a directed graph G with nodes s and t, does G have a Hamiltonian path from s to t?
- **Universe** $\Omega = \{ \langle G, s, t \rangle \mid$ directed graph G with nodes $s, t \}$, and **language** is

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

- If G is above graph, $\langle G, 1, 8 \rangle \in \textit{HAMPATH}$, $\langle G, 2, 8 \rangle \notin \textit{HAMPATH}$.

Hamiltonian Path

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

- **Question:** How hard is it to decide *HAMPATH*?
- Suppose graph G has m nodes.
- Easy to come up with (exponential) **brute-force algorithm**
  - Generate each of the $(m-2)!$ potential paths.
  - Check if any of these is Hamiltonian.
- Currently **unknown** if *HAMPATH* is **solvable** in polynomial time.

Hamiltonian Path

- But *HAMPATH* has feature known as **polynomial verifiability**.
- A claimed Hamiltonian path can be **verified** in **polynomial time**.
  - Consider $\langle G, s, t \rangle \in \textit{HAMPATH}$, where graph G has m nodes.
  - Then (# edges in G) $\leq m(m-1) = O(m^2)$.
  - Suppose G encoded as list of nodes followed by list of edges.
  - Suppose given list $p_1, p_2, \ldots, p_m$ of nodes that is claimed to be Hamiltonian path in G from s to t.
  - Can verify claim by checking
    1. if each node in G appears exactly once in claimed path, which takes $O(m^2)$ time,
    2. if each pair (p_i, p_{i+1}) is edge in G, which takes $O(m^3)$ time.
  - So **verification** takes time $O(m^3)$, which is polynomial in m.
- Thus, **verifying** a given path is Hamiltonian **may be** easier than **determining** its existence.

Composite Numbers

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

- a composite number is not prime.
- **Decision problem:** Given natural number x, is x composite?
- **Universe** $\Omega = \{ \langle x \rangle \mid$ natural number $x \}$, and **language** is

$$\textit{COMPOSITES} = \{ \langle x \rangle \mid x = pq, \text{ for integers } p, q > 1 \} \subseteq \Omega.$$

Remarks:

- Can easily verify that a number is composite.
  - If someone claims a number x is composite and provides a divisor p, just need to verify that x is divisible by p.
- In 2002, Agrawal, Kayal and Sexena proved that $\textit{PRIMES} \in \mathrm{P}$.
  - But $\textit{COMPOSITES} = \overline{\textit{PRIMES}}$, so $\textit{COMPOSITES} \in \mathrm{P}$.

Verifiability

- Some problems **may not** be polynomially verifiable.
  - Consider $\overline{HAMPATH}$, which is complement of *HAMPATH*.
  - No known way to verify $\langle G, s, t \rangle \in \overline{HAMPATH}$ in polynomial time.
- **Definition: Verifier** for language A is (deterministic) algorithm V, where
$$A = \{ w \mid V \text{ accepts } \langle w, c \rangle \text{ for some string } c \}$$
- String c used to verify string $w \in A$
  - c is called a **certificate**, or **proof**, of membership in A.
  - Certificate is only for YES instance, not for NO instance.
- We measure verifier runtime only in terms of length of w.
- A **polynomial-time verifier** runs in (deterministic) time that is polynomial in $|w|$.
- Language is **polynomially verifiable** if it has polynomial-time verifier.

Examples of Verifiers and Certificates

- For *HAMPATH*, a certificate for
$$\langle G, s, t \rangle \in \textit{HAMPATH}$$
is simply the Hamiltonian path from s to t.
  - Can verify in time polynomial in $|\langle G, s, t \rangle|$ if path is Hamiltonian.
- For *COMPOSITES*, a certificate for
$$\langle x \rangle \in \textit{COMPOSITES}$$
is simply one of its divisors.
  - Can verify in time polynomial in $|\langle x \rangle|$ that the given divisor actually divides x
- **Remark:** Certificate c is only for YES instance, not for NO instance.

Class NP

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

Remarks:

- Class NP important because it contains many problems of practical interest
  - *HAMPATH*
  - Travelling salesman
  - All of P
- The term NP comes from **nondeterministic polynomial time**.
  - Can define NP in terms of nondeterministic polynomial-time TMs.
- Recall: a nondeterministic TM (NTM) makes "lucky guesses" in computation.

NTM N_1 for HAMPATH

N_1 = "On input $\langle G, s, t \rangle \subseteq \Omega$, for directed graph G with nodes s, t:

1. Write list of m numbers $p_1, p_2, \ldots, p_m$, where m is # of nodes in G. Each number in list selected **nondeterministically** between 1 and m.
2. Check for repetitions in list. If any found, *reject*.
3. Check whether $p_1 = s$ and $p_m = t$. If either fails, *reject*.
4. For $i = 1$ to $m - 1$, check whether (p_i, p_{i+1}) is an edge of G. If any is not, *reject*. Otherwise, *accept*."

Complexity of N_1 (when G encoded as list of nodes and list of edges):

- Stage 1 takes **nondeterministic** polynomial time: $O(m)$.
- Stages 2 and 3 are simple deterministic poly-time checks: $O(m^2)$.
- Stage 4 runs in deterministic polynomial time: $O(m^3)$.

Equivalent Definition of NP

Theorem 7.20
A language is in NP if and only if it is decided by some polynomial-time nondeterministic TM.

Proof Idea:

- Recall language in NP has (deterministic) poly-time verifier.
- Given a poly-time verifier, build NTM that guesses the certificate c and then runs verifier using c.
  - NTM runs in nondeterministic polynomial time.
- Given a poly-time NTM, build verifier with certificate c that tells NTM which is accepting branch.
  - Verifier runs in deterministic polynomial time.

Proof: "$A \in \mathrm{NP}$" $\Rightarrow$ "A Decided by Poly-time NTM"

- Let V be polynomial-time verifier for A.
  - Assume V is DTM with n^k runtime, where n is length of input w.
- Using V as subroutine, construct NTM N as follows:

  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*."
- NTM N runs in nondeterministic polynomial time.
  - Verifier V runs in time n^k, so certificate c must have length $\leq n^k$; otherwise, V can't even read entire certificate.
  - Stage 1 of NTM N takes $O(n^k)$ nondeterministic time.

Proof: "A Decided by Poly-time NTM" $\Rightarrow$ "$A \in \mathrm{NP}$"

- Assume A decided by polynomial-time NTM N.
- Use N to construct 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 each step's nondeterministic choice.
  2. If this branch of N's computation accepts, *accept*;
     otherwise, *reject*."
- V runs in deterministic polynomial time.
  - NTM N originally runs in nondeterministic polynomial time.
  - Certificate c tells NTM N how to compute, eliminating nondeterminism in N's computation.

NTIME$(t(n))$ and NP

Definition:

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

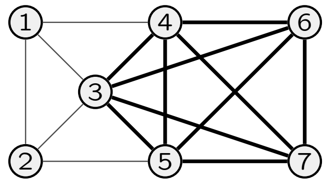
Corollary 7.22

$$\mathrm{NP} = \bigcup_{k \geq 0} \text{NTIME}(n^k).$$

Remark:

- NP is insensitive to choice of "reasonable" nondeterministic computational model.
  - This is because all such models are polynomially equivalent.

Example: CLIQUE



- **Definition:** A **clique** in a graph is a subgraph in which every two nodes are connected by an edge, i.e., clique is **complete subgraph**.
- **Definition:** A k-**clique** is a clique of size k.
- **Decision problem**: Given graph G and integer k, does G have k-clique?
  - **Universe** $\Omega = \{ \langle G, k \rangle \mid G$ is undirected graph, k integer $\}$
  - **Language** of decision problem

    $\mathit{CLIQUE} = \{ \langle G, k \rangle \mid G$ is undirected graph with k-clique $\} \subseteq \Omega$.
  - For graph G above, $\langle G, 5 \rangle \in \mathit{CLIQUE}$, but $\langle G, 6 \rangle \notin \mathit{CLIQUE}$.

CLIQUE $\in$ NP

Theorem 7.24
CLIQUE $\in \mathrm{NP}$.

Proof.

- The clique is the certificate c.
- Here is a verifier for *CLIQUE*:

  $V =$ "On input $\langle \langle G, k \rangle, c \rangle$:
  1. Test whether c is a set of k different nodes in G.
  2. Test whether G contains all edges connecting nodes in c.
  3. If both tests pass, *accept*; otherwise, *reject*."
- If graph G (encoded as list of nodes and edges) has m nodes, then
  - Stage 1 takes $O(k)O(m) = O(km)$ time.
  - Stage 2 takes $O(k^2)O(m^2) = O(k^2 m^2)$ time.

Example: SUBSET-SUM

- **Decision problem:** Given
  - collection S of numbers $x_1, \ldots, x_k$
  - target number t

  does some subcollection of S add up to t?
- **Universe** $\Omega = \{\langle S, t \rangle \mid$ collection $S = \{x_1, \ldots, x_k\}$, target $t\}$.
- **Language**

$$\mathit{SUBSET\text{-}SUM} = \{ \langle S, t \rangle \mid S = \{x_1, \ldots, x_k\} \text{ and } \exists \{y_1, \ldots, y_\ell\} \subseteq \{x_1, \ldots, x_k\} \text{ with } \textstyle\sum_{i=1}^{\ell} y_i = t \} \subseteq \Omega$$

Example:

- $\langle \{4, 11, 16, 21, 27\},\ 32 \rangle \in$ *SUBSET-SUM* as $11 + 21 = 32$.
- $\langle \{4, 11, 16, 21, 27\},\ 17 \rangle \notin$ *SUBSET-SUM*.

Remark: Collections are **multisets**: repetitions allowed.
If number x appears r times in S, then sum can include $\leq r$ copies of x.

SUBSET-SUM $\in$ NP

Theorem 7.25
SUBSET-SUM $\in \mathrm{NP}$.

Proof.

- The subset is the certificate c.
- Here 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 every number in c belongs to S.
  3. If both tests pass, *accept*; otherwise, *reject*."
- When $|S| = k$,
  - $|c| \leq k$, so V takes $O(k^2)$ time.

Class coNP

- The complements $\overline{CLIQUE}$ and $\overline{SUBSET\text{-}SUM}$ are not obviously members of NP.
  - $\overline{CLIQUE} = \{ \langle G, k \rangle \mid$ undirected graph G does **not** have k-clique $\}$
  - Not clear how to define certificates so that we can verify in polynomial time.
- It **seems** harder to verify that something does **not** exist.

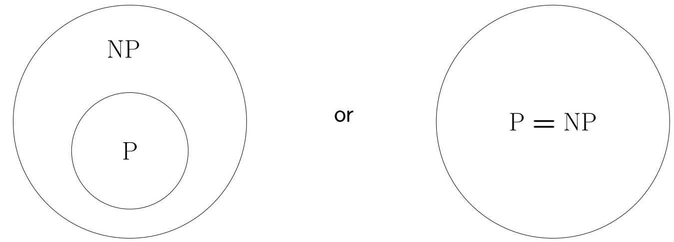
Definition: The class coNP consists of languages whose complements belong to NP.

- Language $A \in \mathrm{coNP}$ iff $\overline{A} \in \mathrm{NP}$.

Remark: Currently not known if coNP is different from NP.

P vs. NP Question

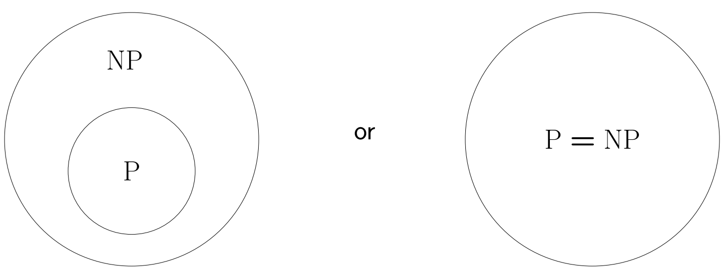
- Language in P has polynomial-time **decider**.
- Language in NP has polynomial-time **verifier** (or poly-time NTM).
- $\mathrm{P} \subseteq \mathrm{NP}$ because each poly-time DTM is also poly-time NTM.



- Answering question whether $\mathrm{P} = \mathrm{NP}$ or not is one of the great unsolved mysteries in computer science and mathematics.
  - Most computer scientists believe $\mathrm{P} \neq \mathrm{NP}$; e.g., jigsaw puzzle.
  - Clay Math Institute (www.claymath.org) has \$1,000,000 prize to anyone who can prove either $\mathrm{P} = \mathrm{NP}$ or $\mathrm{P} \neq \mathrm{NP}$.

Remarks on P vs. NP Question

- If $\mathrm{P} \neq \mathrm{NP}$, then
  - languages in P are **tractable** (i.e., solvable in polynomial time)
  - languages in $\mathrm{NP} - \mathrm{P}$ are **intractable** (i.e., polynomial-time solution doesn't exist).



- If any NP language $A \notin \mathrm{P}$, then $\mathrm{P} \neq \mathrm{NP}$.
  - Nobody has been able to (dis)prove $\exists$ language $\in \mathrm{NP} - \mathrm{P}$.

NP-Complete

Informally, the class $\mathrm{NP\text{-}Complete}$ comprise languages that are

- "hardest" languages in NP
- "least likely" to be in P
- If any $\mathrm{NP\text{-}Complete}$ language $A \in \mathrm{P}$, then $\mathrm{P} = \mathrm{NP}$.
  - If $\mathrm{P} \neq \mathrm{NP}$, then every $\mathrm{NP\text{-}Complete}$ language $A \notin \mathrm{P}$.
- Because $\mathrm{NP\text{-}Complete} \subseteq \mathrm{NP}$,
  - if any $\mathrm{NP\text{-}Complete}$ language $A \notin \mathrm{P}$, then $\mathrm{P} \neq \mathrm{NP}$.

We will give a formal definition of $\mathrm{NP\text{-}Complete}$ later.

Satisfiability Problem

- A **Boolean variable** is a variable that can take on only the values TRUE (1) and FALSE (0).
- Boolean operations
  - AND: $\wedge$
  - OR: $\vee$
  - NOT: $\neg$ or overbar ($\overline{x} = \neg x$)
- Examples

$$\begin{aligned} 0 \wedge 1 &= 0 \\ 0 \vee 1 &= 1 \\ \overline{0} &= 1 \end{aligned}$$

Satisfiability Problem

- A **Boolean formula** (or function) is an expression involving Boolean variables and operations, e.g.,

$$\phi_1 = (\overline{x} \wedge y) \vee (x \wedge \overline{z})$$

- **Definition:** A formula is **satisfiable** if some assignment of 0s and 1s to the variables makes the formula evaluate to 1.
  - **Example:** ϕ_1 above is satisfiable by $(x, y, z) = (0, 1, 0)$. This assignment **satisfies** ϕ_1.
  - **Example:** The following formula is not satisfiable:

$$\phi_2 = (\overline{x} \vee y) \wedge (z \wedge \overline{z}) \wedge (y \vee x)$$

- **Decision problem** *SAT*: Given Boolean fcn ϕ, is ϕ satisfiable?
  - **Universe** $\Omega = \{ \langle \phi \rangle \mid \phi \text{ is a Boolean fcn} \}$
  - **Language** of **satisfiability problem**:

$$\mathit{SAT} = \{ \langle \phi \rangle \mid \phi \text{ is a } \textbf{satisfiable} \text{ Boolean function} \} \subseteq \Omega$$

  so $\langle \phi_1 \rangle \in \mathit{SAT}$ and $\langle \phi_2 \rangle \notin \mathit{SAT}$.

More Definitions Related to Satisfiability

- A **literal** is a variable or negated variable: x or $\overline{x}$
- A **clause** is several literals joined by ORs ($\vee$): $(x_1 \vee \overline{x_3} \vee \overline{x_7})$
  - Clause is TRUE iff at least one of its literals is TRUE.
- A Boolean function is in **conjunctive normal form**, called a **cnf-formula**, if it comprises several clauses connected with ANDs ($\wedge$):

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

- **3cnf-formula** has all clauses with 3 literals:

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

- **Decision problem** *3SAT*: Given a 3cnf-formula ϕ, is ϕ satisfiable?
  - **Universe** $\Omega = \{ \langle \phi \rangle \mid \phi \text{ is 3cnf-formula} \}$
  - **Language** of decision problem:

$$\mathit{3SAT} = \{ \langle \phi \rangle \mid \phi \text{ is a } \textbf{satisfiable} \text{ 3cnf-function} \} \subseteq \Omega.$$

  - $\langle \phi \rangle \in \mathit{3SAT}$ iff each clause in ϕ has at least one literal assigned 1.

Polynomial-Time Computable Functions

Definition: A **polynomial-time computable function** is

$$f : \Sigma_1^* \to \Sigma_2^*$$

if $\exists$ Turing machine that

- starts with input $w \in \Sigma_1^*$,
- halts with only $f(w) \in \Sigma_2^*$ on the tape, and
- has runtime that is polynomial in $|w|$ for $w \in \Sigma_1^*$.

Polynomial-Time Mapping Reducible

Consider

- language A defined over alphabet Σ_1; i.e., universe $\Omega_1 = \Sigma_1^*$.
- language B defined over alphabet Σ_2; i.e., universe $\Omega_2 = \Sigma_2^*$.

Definition: A is **polynomial-time mapping reducible** to B, written

$$A \leq_{\mathrm{P}} B$$

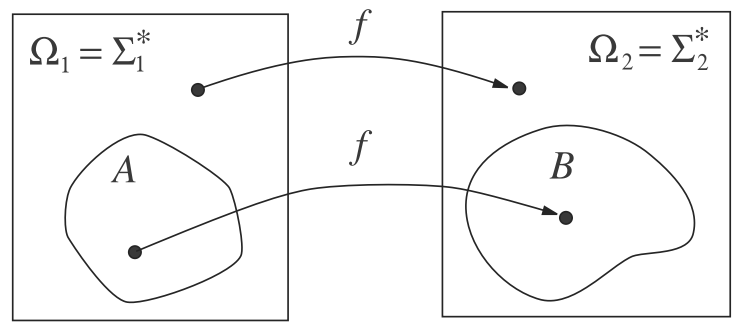
if there is a polynomial-time computable function

$$f : \Sigma_1^* \to \Sigma_2^*$$

such that, for every string $w \in \Sigma_1^*$,

$$w \in A \iff f(w) \in B.$$

Polynomial-Time Mapping Reducible

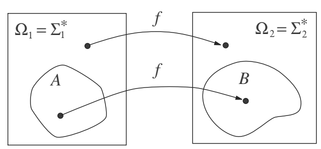


$$w \in A \iff f(w) \in B$$

YES instance for problem A $\iff$ YES instance for problem B

- converts questions about membership in A to membership in B
- conversion is done **efficiently** (i.e., in polynomial time).

Polynomial-Time Mapping Reducible



Theorem 7.31
If $A \leq_{\mathrm{P}} B$ and $B \in \mathrm{P}$, then $A \in \mathrm{P}$.

Proof.

- $B \in \mathrm{P} \Rightarrow \exists$ TM M that is polynomial-time decider for B.
- $A \leq_{\mathrm{P}} B \Rightarrow \exists$ function f that reduces A to B in polynomial time.
- Define TM N that decides A as follows:

  $N =$ "On input w,
  1. Compute $f(w)$.
  2. Run M on input $f(w)$ and output whatever M outputs."
- **Analysis of Time Complexity of TM N:**
  - Each stage runs once.
  - Stage 1 is polynomial because f is polynomial-time function.
  - Stage 2 is polynomial because M is polynomial-time decider for B.

3SAT $\leq_{\mathrm{P}}$ CLIQUE

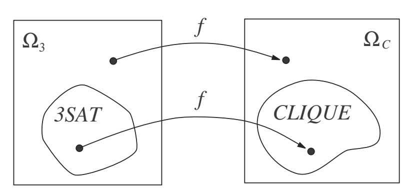
Theorem 7.32
3SAT is polynomial-time mapping reducible to *CLIQUE*.

Proof Idea: Convert instance ϕ of *3SAT* problem with k clauses into instance $\langle G, k\rangle$ of clique problem: $\langle \phi \rangle \in$ *3SAT* iff $\langle G, k\rangle \in$ *CLIQUE*.

- Recall

$$\begin{aligned}
\textit{3SAT} &= \{ \langle \phi \rangle \mid \text{3cnf-fcn } \phi \text{ is satisfiable} \} \\
&\subseteq \{ \langle \phi \rangle \mid \text{3cnf-fcn } \phi \} \equiv \Omega_3, \\
\textit{CLIQUE} &= \{ \langle G, k \rangle \mid \text{undirected graph } G \text{ has } k\text{-clique} \} \\
&\subseteq \{ \langle G, k \rangle \mid \text{undirected graph } G, \text{ integer } k \} \equiv \Omega_C.
\end{aligned}$$

- Need poly-time reducing function $f : \Omega_3 \to \Omega_C$



3SAT is Mapping Reducible to CLIQUE

Proof Idea: Convert instance ϕ of *3SAT* problem with k clauses into instance $\langle G, k\rangle$ of clique problem: $\langle \phi \rangle \in$ *3SAT* iff $\langle G, k\rangle \in$ *CLIQUE*.

- Suppose ϕ is a 3cnf-function with k clauses, e.g.,

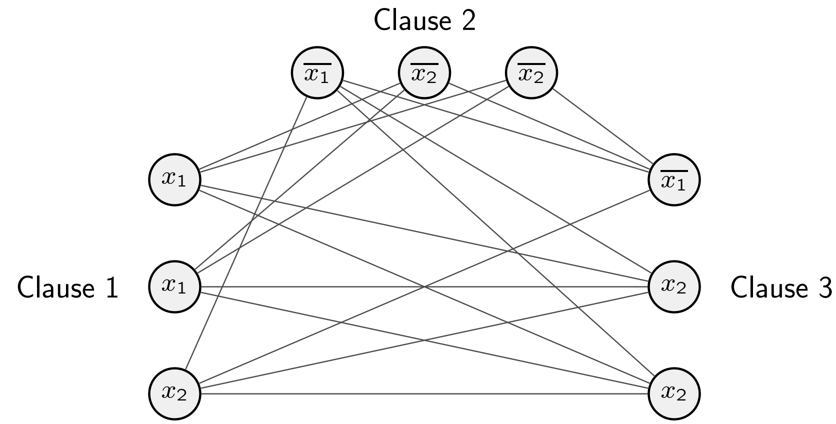
  $$\phi = (x_1 \vee \overline{x_2} \vee \overline{x_3}) \wedge (x_3 \vee \overline{x_5} \vee x_6) \wedge (x_3 \vee \overline{x_6} \vee x_4) \wedge (x_2 \vee x_1 \vee x_5)$$
- Convert ϕ into a graph G as follows:
  - Each literal in ϕ corresponds to a node in G.
  - Nodes in G are organized into k triples $t_1, t_2, \ldots, t_k$.
  - Triple t_i corresponds to the ith clause in ϕ.
  - Add edges between each pair of nodes, except
    - within same triple
    - between contradictory literals, e.g., x_1 and $\overline{x_1}$

3SAT is Mapping Reducible to CLIQUE

Example: 3cnf-function with $k = 3$ clauses and $m = 2$ variables:

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

Corresponding Graph:

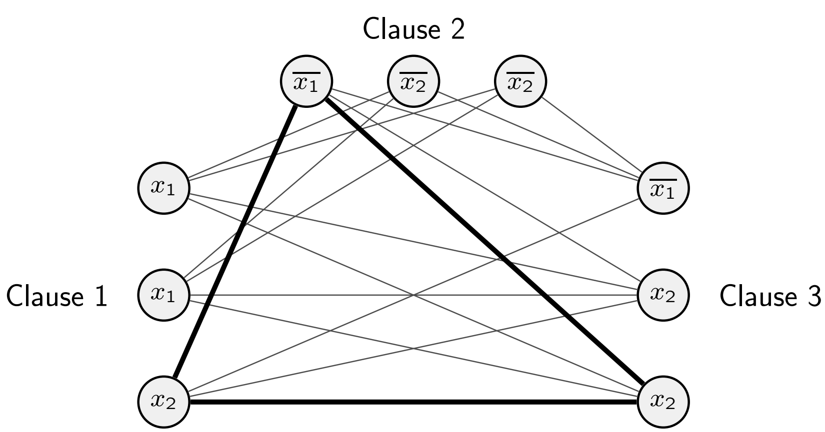


3SAT is Mapping Reducible to CLIQUE

- 3cnf-formula with $k = 3$ clauses and $m = 2$ variables

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

  is satisfiable by assignment $x_1 = 0$, $x_2 = 1$.
- Corresponding graph has k-clique:



3SAT is Mapping Reducible to CLIQUE

Need to show 3cnf-fcn ϕ with k clauses is satisfiable iff G has a k-clique.

- **Key Idea:** $\langle \phi \rangle \in$ *3SAT* iff each clause in ϕ has ≥ 1 true literal.
- Recall: G has node triples corresponding to clauses in ϕ.
- Add edges between each pair of nodes, except
  - within same triple
  - between contradictory literals, e.g., x_1 and $\overline{x_1}$
- If $\langle \phi \rangle \in$ *3SAT*, then choose node corresponding to satisfied literal in each clause to get k-clique in G.
- If $\langle G, k\rangle \in$ *CLIQUE*, then literals corresponding to k-clique satisfy ϕ.

Conclusion: $\langle \phi \rangle \in$ *3SAT* iff $\langle G, k\rangle \in$ *CLIQUE*, so

$$\textit{3SAT} \leq_{\mathrm{m}} \textit{CLIQUE}.$$

Reducing 3SAT to CLIQUE Takes Polynomial Time

Claim: The mapping $\phi \to \langle G, k \rangle$ is polynomial-time computable.

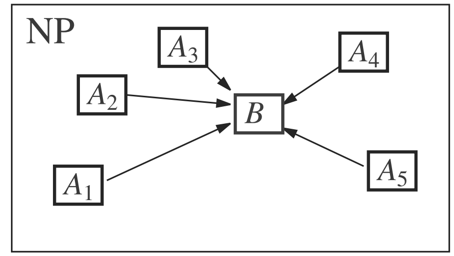
Proof.

- Size of given 3cnf-function ϕ
  - k clauses
  - m variables.
- Constructed graph G of $\langle G, k \rangle$ has
  - $3k$ nodes
  - (# of edges in G) $< \binom{3k}{2} = \frac{3k(3k-1)}{2} = O(k^2)$
  - Size of graph G is polynomial in size of 3cnf-function ϕ.

NP-Complete

Definition: Language B is **NP-Complete** if

1. $B \in \mathrm{NP}$, and
2. For every language $A \in \mathrm{NP}$, we have $A \leq_{\mathrm{P}} B$.



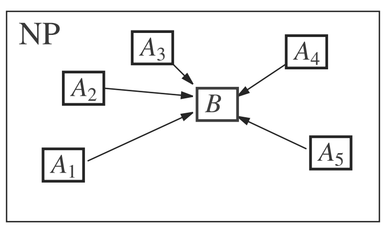
Remarks:

- NP-Complete problems are the most difficult problems in NP.
- **Definition:** Language B is **NP-Hard** if B satisfies part 2 of NP-Complete.

NP-Complete and P vs. NP Question

Theorem 7.35
If there is an NP-Complete language B and $B \in \mathrm{P}$, then $\mathrm{P} = \mathrm{NP}$.

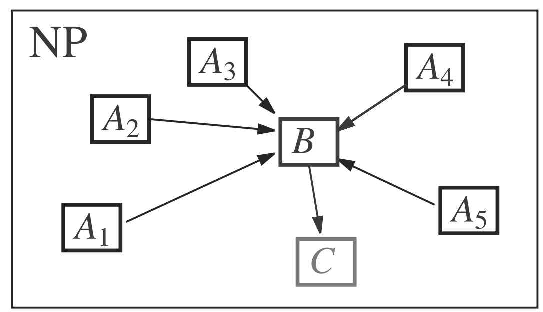
Proof.



- Consider any language $A \in \mathrm{NP}$.
- As $A \in \mathrm{NP}$, defn of NP-completeness implies $A \leq_{\mathrm{P}} B$.
- Recall Theorem 7.31: If $A \leq_{\mathrm{P}} B$ and $B \in \mathrm{P}$, then $A \in \mathrm{P}$.
- Because $B \in \mathrm{P}$, it follows that also $A \in \mathrm{P}$ by Theorem 7.31.

Identifying New NP-Complete Problems from Known Ones

Theorem 7.36
If B is NP-Complete and $B \leq_{\mathrm{P}} C$ for $C \in \mathrm{NP}$, then C is NP-Complete.



Identifying New NP-Complete Problems from Known Ones

Recall Theorem 7.36:
If B is NP-Complete and $B \leq_{\mathrm{P}} C$ for $C \in \mathrm{NP}$, then C is NP-Complete.

Proof.

- Assume that $C \in \mathrm{NP}$.
- Must show that every $A \in \mathrm{NP}$ satisfies $A \leq_{\mathrm{P}} C$.
- Because B is NP-Complete,
  - every language in NP is polynomial-time reducible to B.
  - Thus, $A \leq_{\mathrm{P}} B$ when $A \in \mathrm{NP}$.
- By assumption, B is polynomial-time reducible to C.
  - Hence, $B \leq_{\mathrm{P}} C$.
- But polynomial-time reductions compose.
  - So $A \leq_{\mathrm{P}} B$ and $B \leq_{\mathrm{P}} C$ imply $A \leq_{\mathrm{P}} C$.

Cook-Levin Theorem

- Once we have one NP-Complete problem,
  can identify others by using polynomial-time reduction (Theorem 7.36).
- But identifying the first NP-Complete problem requires some effort.
- Recall **satisfiability problem:**

$$SAT = \{ \langle \phi \rangle \mid \phi \text{ is a satisfiable Boolean function} \}$$

Theorem 7.37
SAT is NP-Complete.

Proof Idea:

- *SAT* $\in \mathrm{NP}$ because a polynomial-time NTM can guess assignment to formula ϕ and *accept* if assignment satisfies ϕ.
- Show that *SAT* is NP-Hard: $A \leq_{\mathrm{P}} SAT$ for every language $A \in \mathrm{NP}$.

Proof Outline of Cook-Levin Theorem

- Let $A \subseteq \Sigma_1^*$ be a language in NP.
- Need to show that $A \leq_{\mathrm{P}} SAT$.
- For every $w \in \Sigma_1^*$, we want a (CNF) formula ϕ such that
  - $w \in A$ iff $\langle \phi \rangle \in SAT$
  - polynomial-time reduction that constructs ϕ from w.
- Let N be poly-time NTM that decides A in time at most n^k for input w with $|w| = n$.
- **Basic approach:**

$$\begin{aligned} w \in A &\iff \text{NTM } N \text{ accepts input } w \\ &\iff \exists \text{ accepting computation history of } N \text{ on } w \\ &\iff \exists \text{ Boolean function } \phi \text{ and variables } x_1, \ldots, x_m \\ &\qquad \text{with } \phi(x_1, \ldots, x_m) = \text{TRUE} \end{aligned}$$

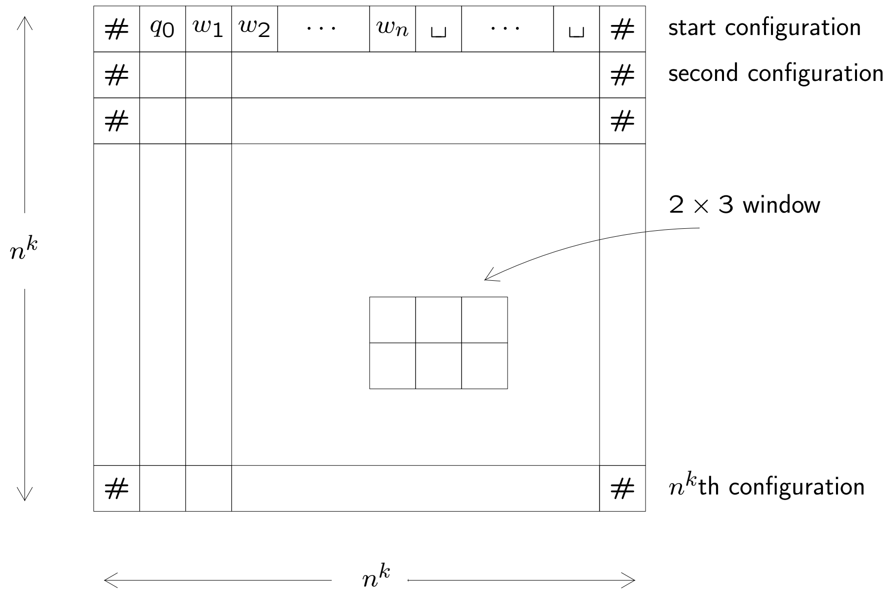
Proof Outline of Cook-Levin Theorem

Idea: "Satisfying assignments of ϕ"
$\leftrightarrow$ "accepting computation history of NTM N on w"

Step 1: Describe computations of NTM N on w by Boolean variables.

- Any computation history of $N = (Q, \Sigma, \Gamma, \delta, q_0, q_A, q_R)$ on w with $|w| = n$ has $\leq n^k$ configurations since assumed N runs in time n^k.
- Each configuration is an element of $C^{(n^k)}$, where $C = Q \cup \Gamma \cup \{\#\}$ (mark left and right ends with $\#$, where $\# \notin \Gamma$).
- Computation described by $n^k \times n^k$ "tableau"
  - Each row of tableau represents one configuration.
  - Each cell in tableau contains one element of C.
- Represent contents of cell (i, j) by $|C|$ Boolean variables $\{ x_{i,j,s} \mid s \in C \}$
  - $x_{i,j,s} = 1$ means "cell (i, j) contains s" (variable is "on")

Tableau is an $n^k \times n^k$ table of configurations



Proof Outline of Cook-Levin Theorem

Step 2: Express conditions for an accepting sequence of configurations of NTM N on w by Boolean formulas:

$$\begin{aligned}
\phi_{\text{cell}} &= \text{"for each cell } (i,j)\text{, exactly one } s \in C \text{ with } x_{i,j,s} = 1\text{"},\\
\phi_{\text{start}} &= \text{"first row of tableau is the starting configuration of } N \text{ on } w\text{"},\\
\phi_{\text{accept}} &= \text{"last row of tableau is an accepting configuration of } N \text{ on } w\text{"},\\
\phi_{\text{move}} &= \text{"every } 2 \times 3 \text{ window is consistent with } N\text{'s transition fcn"}.
\end{aligned}$$

For example,

$$\phi_{\text{cell}} = \underbrace{\bigwedge_{1 \le i,j \le n^k}}_{\text{for each cell } (i,j),} \left[\underbrace{\Big(\bigvee_{s \in C} x_{i,j,s} \Big)}_{\ge 1 \text{ symbol used}} \wedge \underbrace{\Big(\bigwedge_{\substack{s,t \in C \\ s \ne t}} (\overline{x_{i,j,s}} \vee \overline{x_{i,j,t}}) \Big)}_{\text{not } \ge 2 \text{ symbols used}} \right].$$

Step 3: Show that each of the above formulas can be

- expressed by a formula of size $O((n^k)^2) = O(n^{2k})$
- constructed from w in time polynomial in $n = |w|$.

Proof Outline of Cook-Levin Theorem

Step 4: Show that N has an accepting computation history on w iff

$$\phi = \phi_{\text{cell}} \wedge \phi_{\text{start}} \wedge \phi_{\text{accept}} \wedge \phi_{\text{move}}$$

has a satisfying assignment of the $x_{i,j,s}$ variables.

Thus, we constructed ϕ using a polynomial-time reduction from A to *SAT*:

$$A \le_{\text{P}} \textit{SAT}$$

Because construction holds for every $A \in \text{NP}$,
SAT is then NP-Complete.

3SAT is NP-Complete

Recall

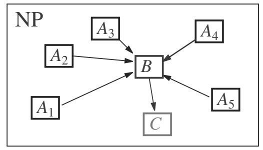
$$\textit{3SAT} = \{ \langle \phi \rangle \mid \phi \text{ is a satisfiable 3cnf-function} \}$$

Corollary 7.42
3SAT is NP-Complete.

Proof Idea:
Can modify proof that *SAT* is NP-Complete (Theorem 7.37) so that resulting Boolean function is a 3cnf-function.

Proving NP-Completeness

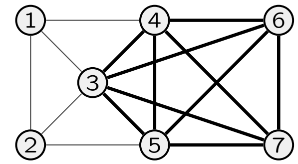
- Tedious to prove a language C is $\mathrm{NP\text{-}Complete}$ using definition:
  1. $C \in \mathrm{NP}$, and
  2. C is $\mathrm{NP\text{-}Hard}$: For every language $A \in \mathrm{NP}$, we have $A \leq_{\mathrm{P}} C$.
- Recall Theorem 7.36:
  If B is $\mathrm{NP\text{-}Complete}$ and $B \leq_{\mathrm{P}} C$ for $C \in \mathrm{NP}$, then C is $\mathrm{NP\text{-}Complete}$.



- Typically prove a language C is $\mathrm{NP\text{-}Complete}$ by applying Thm 7.36
  1. Prove that language $C \in \mathrm{NP}$.
  2. Reduce a known $\mathrm{NP\text{-}Complete}$ problem B to C.
     - At this point, have shown that *SAT* and *3SAT* are $\mathrm{NP\text{-}Complete}$.
  3. Show that reduction takes polynomial time.

CLIQUE is NP-Complete

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



Corollary 7.43
CLIQUE is $\mathrm{NP\text{-}Complete}$.

Proof.

- Theorem 7.24: *CLIQUE* $\in \mathrm{NP}$.
- Corollary 7.42: *3SAT* is $\mathrm{NP\text{-}Complete}$.
- Theorem 7.32: *3SAT* $\leq_{\mathrm{P}}$ *CLIQUE*.
- Thus, Theorem 7.36 implies *CLIQUE* is $\mathrm{NP\text{-}Complete}$.

Integer Linear Programming

Definition: An **integer linear program (ILP)** is

- set of variables $y_1, y_2, \ldots, y_n$, which **must take integer values.**
- set of m linear inequalities:

$$\begin{array}{ccccccccc} a_{11}\,y_1 & + & a_{12}\,y_2 & + & \cdots & + & a_{1n}\,y_n & \leq & b_1 \\ a_{21}\,y_1 & + & a_{22}\,y_2 & + & \cdots & + & a_{2n}\,y_n & \leq & b_2 \\ \vdots & & \vdots & & \ddots & & \vdots & & \vdots \\ a_{m1}\,y_1 & + & a_{m2}\,y_2 & + & \cdots & + & a_{mn}\,y_n & \leq & b_m \end{array}$$

  where the a_{ij} and b_i are given constants.
- In matrix notation, $Ay \leq b$, with matrix A and vectors y, b:

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}, \quad y = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}, \quad b = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix}.$$

Integer Linear Programming

Example: Can transform $\geq$ and $=$ relations into $\leq$ relations:

$$\begin{array}{rcl} 5y_1 - 2y_2 + y_3 \leq 7 & & \\ y_1 \geq 2 & \longleftrightarrow & -y_1 \leq -2 \\ y_2 + 2y_3 = 8 & \longleftrightarrow & y_2 + 2y_3 \leq 8 \quad \& \quad y_2 + 2y_3 \geq 8 \end{array}$$

becomes ILP

$$\begin{array}{rcrcrcr} 5\,y_1 & - & 2\,y_2 & + & 1y_3 & \leq & 7 \\ -1\,y_1 & + & 0\,y_2 & + & 0\,y_3 & \leq & -2 \\ 0\,y_1 & + & 1\,y_2 & + & 2\,y_3 & \leq & 8 \\ 0\,y_1 & - & 1\,y_2 & - & 2\,y_3 & \leq & -8 \end{array}$$

so

$$A = \begin{pmatrix} 5 & -2 & 1 \\ -1 & 0 & 0 \\ 0 & 1 & 2 \\ 0 & -1 & -2 \end{pmatrix}, \quad y = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix}, \quad b = \begin{pmatrix} 7 \\ -2 \\ 8 \\ -8 \end{pmatrix}.$$

ILP is NP-Complete

- **Decision problem:** Given matrix A and vector b, is there an **integer** vector y such that $Ay \leq b$?

$$\begin{aligned} ILP &= \{ \langle A, b \rangle \mid \text{matrix } A \text{ and vector } b \text{ satisfy } Ay \leq b \\ &\qquad \text{with } y \text{ an integer vector} \} \\ &\subseteq \{ \langle A, b \rangle \mid \text{matrix } A, \text{ vector } b \} \equiv \Omega_I \end{aligned}$$

- **Example:** The instance $\langle A, b \rangle \in \Omega_I$, where

$$A = \begin{pmatrix} 1 & 2 \\ 2 & 4 \end{pmatrix}, \quad b = \begin{pmatrix} 3 \\ 7 \end{pmatrix},$$

satisfies $Ay \leq b$ for $y = (1, 1)^\top$, so $\langle A, b \rangle \in$ *ILP*.

- **Example:** The instance $\langle C, d \rangle \in \Omega_I$, where

$$C = \begin{pmatrix} 2 & 0 \\ -2 & 0 \end{pmatrix}, \quad d = \begin{pmatrix} 3 \\ -3 \end{pmatrix},$$

requires $2y_1 \leq 3$ & $-2y_1 \leq -3$, which means $2y_1 = 3$, so only non-integer solutions $y = (3/2,\ y_2)^\top$ for any y_2; thus, $\langle C, d \rangle \notin$ *ILP*.

- **Theorem:** *ILP* is NP-Complete.

ILP $\in \mathrm{NP}$

Proof.

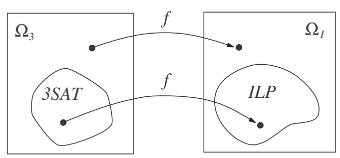
- The certificate c is an integer vector satisfying $Ac \leq b$.
- Here is a verifier for *ILP*:

  $V =$ "On input $\langle \langle A, b \rangle, c \rangle$:
  1. Test whether c is a vector of all integers.
  2. Test whether $Ac \leq b$.
  3. If both tests pass, *accept*; otherwise, *reject*."

- If $Ay \leq b$ has m inequalities and n variables, then
  - Stage 1 takes $O(n)$ time
  - Stage 2 takes $O(mn)$ time
  - So verifier V runs in $O(mn)$, which is polynomial in size of problem.

Now prove *ILP* is NP-Hard by showing *3SAT* $\leq_{\mathrm{P}}$ *ILP*.

3SAT $\leq_{\mathrm{m}}$ ILP



- Reducing fcn $f : \Omega_3 \to \Omega_I$
  - $\langle \phi \rangle \in$ *3SAT* iff $f(\langle \phi \rangle) = \langle A, b \rangle \in$ *ILP*
- Consider 3cnf-formula with $m = 4$ variables and $k = 3$ clauses:

$$\phi = (x_1 \vee x_2 \vee \overline{x_3}) \wedge (\overline{x_1} \vee \overline{x_2} \vee x_4) \wedge (\overline{x_2} \vee \overline{x_4} \vee \overline{x_3})$$

- Define integer linear program with
  - $2m = 8$ variables $y_1, y_1', y_2, y_2', y_3, y_3', y_4, y_4'$
    - y_i corresponds to x_i
    - y_i' corresponds to $\overline{x_i}$
  - 3 sets of inequalities for each of pair y_i, y_i', which must be integers:

$$\begin{array}{lll} 0 \leq y_1 \leq 1, & 0 \leq y_1' \leq 1, & y_1 + y_1' = 1 \\ 0 \leq y_2 \leq 1, & 0 \leq y_2' \leq 1, & y_2 + y_2' = 1 \\ 0 \leq y_3 \leq 1, & 0 \leq y_3' \leq 1, & y_3 + y_3' = 1 \\ 0 \leq y_4 \leq 1, & 0 \leq y_4' \leq 1, & y_4 + y_4' = 1 \end{array}$$

  which guarantee that exactly one of y_i and y_i' is 1, and other is 0.

3SAT $\leq_{\mathrm{m}}$ ILP

- Recall 3cnf-formula with $m = 4$ variables and $k = 3$ clauses:

$$\phi = (x_1 \vee x_2 \vee \overline{x_3}) \wedge (\overline{x_1} \vee \overline{x_2} \vee x_4) \wedge (\overline{x_2} \vee \overline{x_4} \vee \overline{x_3})$$

  - ϕ satisfiable iff each clause evaluates to 1.
  - A clause evaluates to 1 iff at least one literal in the clause equals 1.
  - For each clause $(x_i \vee \overline{x_j} \vee x_\ell)$, create inequality $y_i + y_j' + y_\ell \geq 1$.
  - For our example, ILP has $k = 3$ inequalities of this type:

$$\begin{array}{c} y_1 + y_2 + y_3' \geq 1 \\ y_1' + y_2' + y_4 \geq 1 \\ y_2' + y_4' + y_3' \geq 1 \end{array}$$

  which guarantee that each clause evaluates to 1.

3SAT $\leq_m$ ILP

- Given 3cnf-formula:

$$\phi = (x_1 \vee x_2 \vee \overline{x_3}) \wedge (\overline{x_1} \vee \overline{x_2} \vee x_4) \wedge (\overline{x_2} \vee \overline{x_4} \vee \overline{x_3})$$

- Constructed ILP:

$$\begin{array}{lll} 0 \leq y_1 \leq 1, & 0 \leq y_1' \leq 1, & y_1 + y_1' = 1 \\ 0 \leq y_2 \leq 1, & 0 \leq y_2' \leq 1, & y_2 + y_2' = 1 \\ 0 \leq y_3 \leq 1, & 0 \leq y_3' \leq 1, & y_3 + y_3' = 1 \\ 0 \leq y_4 \leq 1, & 0 \leq y_4' \leq 1, & y_4 + y_4' = 1 \end{array}$$

$$\begin{array}{l} y_1 + y_2 + y_3' \geq 1 \\ y_1' + y_2' + y_4 \geq 1 \\ y_2' + y_4' + y_3' \geq 1 \end{array}$$

- Note that:

ϕ satisfiable $\iff$ constructed ILP has solution (with values of variables $\in \{0, 1\}$)

Reducing 3SAT to ILP Takes Polynomial Time

- Given 3cnf-formula ϕ with
  - m variables: $x_1, x_2, \ldots, x_m$
  - k clauses
- Constructed ILP has
  - $2m$ (integer) variables: $y_1, y_1', y_2, y_2', \ldots, y_m, y_m'$
  - $6m + k$ inequalities:
    - 3 sets of inequalities for each pair y_i, y_i':

      $$0 \leq y_i \leq 1, \qquad 0 \leq y_i' \leq 1, \qquad y_i + y_i' = 1,$$

      so total of $6m$ inequalities of this type (convert $=$ into $\leq$ & $\geq$)
    - For each clause in ϕ, ILP has corresponding inequality, e.g.,

      $$(x_1 \vee x_2 \vee \overline{x_3}) \quad \longleftrightarrow \quad y_1 + y_2 + y_3' \geq 1,$$

      so total of k inequalities of this type.
  - Thus, size of ILP is polynomial in m and k.

Many Other NP-Complete Problems

- *HAMPATH*, *SUBSET-SUM*, ...
- *Travelling Salesman Problem* (TSP): Given a graph G with weighted edges and a threshold value d, is there a tour that visits each node once and has total length at most d?
- *Long-Path Problem*: Given a graph G with weighted edges, two nodes s and t in G, and a threshold value d, is there a path from s to t with length *at least* d?
- *Scheduling Final Exams*: Is there a way to schedule final exams in a d-day period so no student is scheduled to take 2 exams at same time?
- *Minesweeper*, *Sudoku*, *Tetris*
- See Garey and Johnson (1979), *Computers and Intractability: A Guide to the Theory of NP-Completeness*, for many reductions.

NP-Hard Optimization Problems

- Decision problems have YES/NO answers.
- Many decision problems have corresponding **optimization** version.
- Optimization version of NP-Complete problems are NP-Hard.

Problem	Decision Version	Optimization Version
CLIQUE	Does a graph G have a clique of size k ?	Find largest clique
ILP	Does $\exists$ integer vector y such that $Ay \leq b$?	Find integer vector y to max $d^\top y$ s.t. $Ay \leq b$
TSP	Does a graph G have tour of length $\leq d$?	Find min length tour
Scheduling	Given set of tasks and constraints, can we finish all tasks in time d ?	Find min time schedule

Why are NP-Complete and NP-Hard Important?

- Suppose you are faced with a problem and you can't come up with an efficient algorithm for it.
- If you can prove the problem is NP-Complete or NP-Hard, then there is no known efficient algorithm to solve it.
  - No known polynomial-time algorithms for NP-Complete and NP-Hard problems.
- How to deal with an NP-Complete or NP-Hard problem?
  - Approximation algorithm
  - Probabilistic algorithm
  - Special cases
  - Heuristic

Summary of Chapter 7

- Time complexity: In terms of size n of input w, how many time steps are required by TM to solve problem?
- Big-O notation: $f(n) = O(g(n))$
  - $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.
  - $g(n)$ is an asymptotic upper bound on $f(n)$.
  - Polynomials $a_k n^k + a_{k-1} n^{k-1} + \cdots = O(n^k)$.
  - Polynomial $= O(n^c)$ for constant $c \geq 0$
  - Exponential $= O(2^{n^\delta})$ for constant $\delta > 0$
  - Exponentials are asymptotically much bigger than any polynomial
- $t(n)$-time k-tape TM has equivalent $O(t^2(n))$-time 1-tape TM.
- $t(n)$-time NTM has equivalent $2^{O(t(n))}$-time 1-tape DTM.
- Strong Church-Turing Thesis: all reasonable variants of DTM are polynomial-time equivalent.

- Class P comprises problems that can be **solved** in polynomial time
  - P includes *PATH*, *RELPRIME*, CFLs (using dynamic programming)
- Class NP: problems that can be **verified** in deterministic polynomial time (equivalently, **solved** in **nondeterministic polynomial** time).
  - NP includes all of P and *HAMPATH*, *CLIQUE*, *SUBSET-SUM*, *3SAT*, *ILP*
- P vs. NP problem:
  - Know P $\subseteq$ NP: poly-time DTM is also poly-time NTM.
  - Unknown if P $=$ NP or P $\neq$ NP.

- Polynomial-time mapping reducible: $A \leq_{\mathsf{P}} B$ if $\exists$ polynomial-time computable function f such that

$$w \in A \iff f(w) \in B.$$

- Defn: language B is NP-Complete if $B \in \mathrm{NP}$ and $A \leq_{\mathsf{P}} B$ for all $A \in \mathrm{NP}$.
  - If any NP-Complete language B is in P, then P $=$ NP.
  - If any NP language B is **not** in P, then P $\neq$ NP.
  - If B is NP-Complete and $B \leq_{\mathsf{P}} C$ for $C \in \mathrm{NP}$, then C is NP-Complete.
  - Cook-Levin Theorem: *SAT* is NP-Complete.
  - *3SAT*, *CLIQUE*, *ILP*, *SUBSET-SUM*, *HAMPATH*, etc. are all NP-Complete