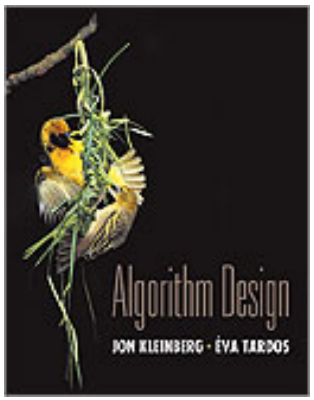
Complexity: Problems and Classes

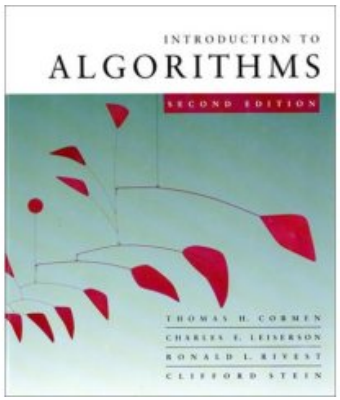
Maria Serna

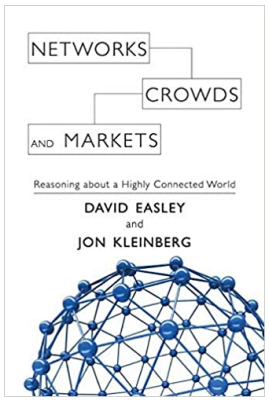
Spring 2026

Algorithmics: Basic references

- Kleinberg, Tardos. Algorithm Design, Pearson Education, 2006.
- Cormen, Leisserson, Rivest and Stein. Introduction to algorithms. Second edition, MIT Press and McGraw Hill 2001.
- Easley, Kleinberg. Networks, Crowds, and Markets: Reasoning About a Highly Connected World, Cambridge University Press, 2010

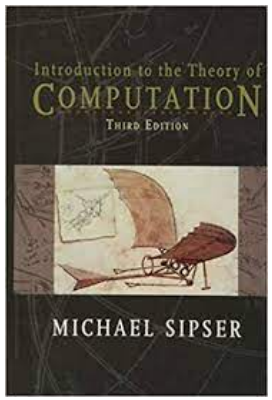


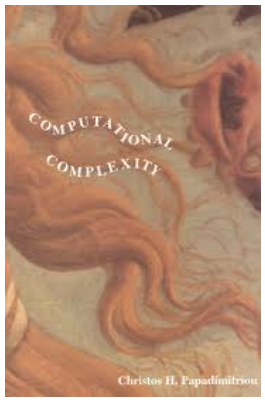


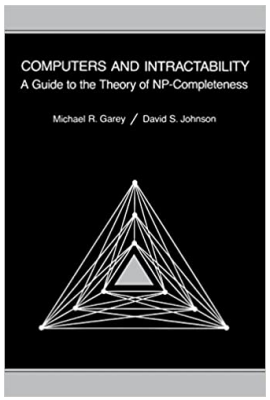


Computational Complexity: Basic references

- Sipser Introduction to the Theory of Computation 2013.
- Papadimitriou Computational Complexity 1994.
- Garey and Johnson Computers and Intractability: A Guide to the Theory of NP-Completeness 1979







Growth of functions: Asymptotic notations

We consider only functions defined on the natural numbers.

$$f, g : \mathbb{N} \to \mathbb{N}$$

O-notation

For a given function $g(n)$

$$O(g(n)) = \{f(n) \,|\text{there exists a positive constant } c \text{ and } n_0 \geq 0 \,\}$$
$$\text{such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0\}$$

Equivalently, the set of functions that verify

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

$$5n^3 + 2n^2 = O(2^n)$$
$$5n^3 + 2n^2 = O(n^4)$$
$$2^n = O(2^{2n})$$
$$2^n = O(2^{n \log n})$$

It is used for asymptotic upper bound.

Although $O(g(n))$ is a set we write $f(n) = O(g(n))$ to indicate that $f(n)$ is a member of $O(g(n))$

Ω-notation

For a given function $g(n)$

$$\Omega(g(n)) = \{f(n) \,|\text{there exists positive constants } c \text{ and } n_0 \text{ such that}\}$$
$$0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0\}$$

Equivalently, the set of functions that verify

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

$$5n^3 + 2n^2 = \Theta(n^3)$$
$$5n^3 + 2n^2 = \Omega(n^3)$$
$$5n^3 + 2n^2 = \Omega(n^2)$$
$$2^n = \Omega(2^{n/2})$$

It is used for asymptotic lower bound.

Θ-notation

For a given function $g(n)$

$$\Theta(g(n)) = \{f(n) \mid \text{there are positive constants } c_1,\ c_2,\ \text{and } n_0 \geq 0 \ \} \\ \text{such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0\}$$

Equivalently, the set of functions that verify

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

$$5n^3 + 2n^2 = \Theta(n^3)$$

$$5n^3 + 2n^2 \notin \Theta(n^2)$$

It is used for asymptotic equivalence

o-notation

For a given function $g(n)$

$$o(g(n)) = \{f(n) \,|\text{for any positive constant } c \text{ there is } n_0 \geq 0 \text{ such that}\}$$
$$0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0\}$$

Note that $f(n) = O(n)$ implies $f(n) \leq cg(n)$ asymptotically for some c
but $f(n) = o(n)$ implies $f(n) \leq cg(n)$ asymptotically for any c
and when $f(n) = o(g(n))$ it holds that $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$

It is used for asymptotic upper bounds that are not asymtotically tight.

ω-notation

$f(n) \in \omega(g(n))$ iff $g(n) \in o(f(n))$

Algorithm's analysis

- Time
- Space

Algorithm $\mathcal{A}$ on input x takes time $t(x)$.
$|x|$ denotes the size of input x.

Definition

The cost function of algorithm $\mathcal{A}$ is a function from $\mathbb{N}$ to $\mathbb{N}$ defined as

$$\mathcal{C}_{\mathcal{A}}(n) = \max_{|x|=n} t(x)$$

Fundamental growth functions

- Polynomial time
- Exponential time

Fundamental growth functions

- Polynomial time
  $\mathcal{C}_{\mathcal{A}}(n) = O(n^c)$, for some constant c.
- Exponential time

Fundamental growth functions

- Polynomial time
  $\mathcal{C}_{\mathcal{A}}(n) = O(n^c)$, for some constant c.
- Exponential time
  $\mathcal{C}_{\mathcal{A}}(n) = 2^{O(n^c)}$, for some constant c.

Fundamental growth functions

- Polynomial time
  $\mathcal{C}_{\mathcal{A}}(n) = O(n^c)$, for some constant c.
- Exponential time
  $\mathcal{C}_{\mathcal{A}}(n) = 2^{O(n^c)}$, for some constant c.
- Quasi-polynomial time

- Pseudo polynomial time

Fundamental growth functions

- Polynomial time
  $\mathcal{C}_{\mathcal{A}}(n) = O(n^c)$, for some constant c.
- Exponential time
  $\mathcal{C}_{\mathcal{A}}(n) = 2^{O(n^c)}$, for some constant c.
- Quasi-polynomial time
  $\mathcal{C}_{\mathcal{A}}(n) = 2^{O(\log^c n)}$, for some constant c.
- Pseudo-polynomial time
  $\mathcal{C}_{\mathcal{A}}(n) = O((mW)^c)$, for some constant c, but input size is $O(m + \log W)$

Fundamental growth functions

- Polynomial time
  $\mathcal{C}_{\mathcal{A}}(n) = O(n^c)$, for some constant c.
- Exponential time
  $\mathcal{C}_{\mathcal{A}}(n) = 2^{O(n^c)}$, for some constant c.
- Quasi-polynomial time
  $\mathcal{C}_{\mathcal{A}}(n) = 2^{O(\log^c n)}$, for some constant c.
- Pseudo-polynomial time
  $\mathcal{C}_{\mathcal{A}}(n) = O((mW)^c)$, for some constant c, but input size is $O(m + \log W)$
- Similar definitions replacing time by space
  Most used PSPACE polynomial space

Problem types

- Decision
  Input x
  Property $P(x)$

  Example: Given a graph and two vertices, is there a path joining them?

- Function
  Input x
  Compute y *such that* $Q(x, y)$

  Example: Given a graph and two vertices, compute the minimum distance between them.

Problem types

- Decision
  Input x
  Property $P(x)$

Problem types

- Decision
  Input x
  Property $P(x)$

  Coding inputs on alphabet Σ a problem is a set

Problem types

- Decision
  Input x
  Property $P(x)$

  Coding inputs on alphabet Σ a problem is a set $\{x \mid P(x)\} \in \mathcal{P}(\Sigma^*)$

Problem types

- Decision
  Input x
  Property $P(x)$

  Coding inputs on alphabet Σ a problem is a set $\{x \mid P(x)\} \in \mathcal{P}(\Sigma^*)$

- Function
  Input x
  Compute y *such that* $Q(x, y)$

Problem types

- Decision
  Input x
  Property $P(x)$

  Coding inputs on alphabet Σ a problem is a set $\{x \mid P(x)\} \in \mathcal{P}(\Sigma^*)$

- Function
  Input x
  Compute y *such that* $Q(x, y)$

  Coding inputs/outputs on alphabet Σ a deterministic algorithm solving a problem determines a function

Problem types

- Decision

  Input x

  Property $P(x)$

  Coding inputs on alphabet Σ a problem is a set $\{x \mid P(x)\} \in \mathcal{P}(\Sigma^*)$

- Function

  Input x

  Compute y *such that* $Q(x, y)$

  Coding inputs/outputs on alphabet Σ a deterministic algorithm solving a problem determines a function $f : \Sigma^* \to \Sigma^*$ s.t., for any x, $Q(x, f(x))$ is true.

Decision problem classes

- Undecidable
  No algorithm can solve the problem.
- Decidable
  There is an algorithm solving them.
  - P:
    There is an algorithm solving it with polynomial cost.
  - EXP
    There is an algorithm solving it with exponential cost.
  - PSPACE
    There is an algorithm solving it within polynomial space.

NP: non-deterministic polynomial time

It is possible to define a certificate y and a property $P(x, y)$ such that

- If x is an input with answer yes, there is y such that $P(x, y)$ is true,
- $P(x, y)$ can be decided in polynomial time, given x and y.
- y has polynomial size with respect to $|x|$.

Problems with a polynomial time verifier

NP: non-deterministic polynomial time

It is possible to define a certificate y and a property $P(x, y)$ such that

- If x is an input with answer yes, there is y such that $P(x, y)$ is true,
- $P(x, y)$ can be decided in polynomial time, given x and y.
- y has polynomial size with respect to $|x|$.

Problems with a polynomial time verifier
$\{x \mid \exists y P(x, y)\}$

Some decision problems

Bipartiteness (BIP)
Given a graph determine whether it is bipartite.

Perfect matching (PMATCH)
Given a graph determine whether it has a perfect matching.

Hamiltonian Cycle (HC)
Given a graph determine whether it has a Hamiltonian circuit.

In which classes?

NP-hardness

- It is an open question whether P = NP or NP = EXP. Most believed is that P $\neq$ NP
- Π is NP-hard means that a polynomial time algorithm for Π can be reused to solve in polynomial time any problem in NP.
- Decision problem A is NP-complete iff $A \in$ NP and A is NP-hard. Look at Garey and Johnson's book for a big list of NP-hard/complete problems.

NP-hardness

- It is an open question whether P = NP or NP = EXP. Most believed is that P $\neq$ NP
- Π is NP-hard means that a polynomial time algorithm for Π can be reused to solve in polynomial time any problem in NP.
- Decision problem A is NP-complete iff $A \in$ NP and A is NP-hard. Look at Garey and Johnson's book for a big list of NP-hard/complete problems.
- The NP-hardness of a problem is assessed through reductions.

Reductions

- Let A and B be decision problems
- A function $f : \mathbb{N} \to \mathbb{N}$ is a reduction from A to B if $x \in A$ iff $f(x) \in B$

Reductions

- Let A and B be decision problems
- A function $f : \mathbb{N} \to \mathbb{N}$ is a reduction from A to B if $x \in A$ iff $f(x) \in B$
- f is a polynomial time reduction if in addition f can be computed in polynomial time.

Reductions

- Let A and B be decision problems
- A function $f : \mathbb{N} \to \mathbb{N}$ is a reduction from A to B if $x \in A$ iff $f(x) \in B$
- f is a polynomial time reduction if in addition f can be computed in polynomial time.
- A is reducible to B ($A \leq B$) if there is a polynomial time reduction from A to B.

Reductions

- Let A and B be decision problems
- A function $f : \mathbb{N} \to \mathbb{N}$ is a reduction from A to B if $x \in A$ iff $f(x) \in B$
- f is a polynomial time reduction if in addition f can be computed in polynomial time.
- A is reducible to B ($A \leq B$) if there is a polynomial time reduction from A to B.

Theorem

If $A \leq B$ and $B \in P$ then $A \in P$

Reductions

- Let A and B be decision problems
- A function $f : \mathbb{N} \to \mathbb{N}$ is a reduction from A to B if $x \in A$ iff $f(x) \in B$
- f is a polynomial time reduction if in addition f can be computed in polynomial time.
- A is reducible to B ($A \leq B$) if there is a polynomial time reduction from A to B.

Theorem

If $A \leq B$ and $B \in P$ then $A \in P$

This type of reduction is calle many-one polynomial reduction, sometiomes we use $\leq_m^p$ to distinguish from other reducibilities.

Completeness

- Let $\leq$ be a reducibility among problems and $\mathcal{C}$ a class of problems. $\mathcal{C}$ is closed under $\leq$ if $A \leq B$ and $B \in \mathcal{C}$ implies $A \in \mathcal{C}$.

Completeness

- Let $\leq$ be a reducibility among problems and $\mathcal{C}$ a class of problems. $\mathcal{C}$ is closed under $\leq$ if $A \leq B$ and $B \in \mathcal{C}$ implies $A \in \mathcal{C}$.
- Let $\leq$ be a reducibility among problems and $\mathcal{C}$ a class of problems closed under $\leq$

Completeness

- Let $\leq$ be a reducibility among problems and $\mathcal{C}$ a class of problems. $\mathcal{C}$ is closed under $\leq$ if $A \leq B$ and $B \in \mathcal{C}$ implies $A \in \mathcal{C}$.
- Let $\leq$ be a reducibility among problems and $\mathcal{C}$ a class of problems closed under $\leq$
  - Problem B is hard for class $\mathcal{C}$ under $\leq$ if, for each $A \in \mathcal{C}$, $A \leq B$.

Completeness

- Let $\leq$ be a reducibility among problems and $\mathcal{C}$ a class of problems. $\mathcal{C}$ is closed under $\leq$ if $A \leq B$ and $B \in \mathcal{C}$ implies $A \in \mathcal{C}$.
- Let $\leq$ be a reducibility among problems and $\mathcal{C}$ a class of problems closed under $\leq$
  - Problem B is hard for class $\mathcal{C}$ under $\leq$ if, for each $A \in \mathcal{C}$, $A \leq B$.
  - Problem B is complete for class $\mathcal{C}$ under $\leq$ if, B is hard for $\mathcal{C}$ and $B \in \mathcal{C}$.

Completeness

- Let $\leq$ be a reducibility among problems and $\mathcal{C}$ a class of problems. $\mathcal{C}$ is closed under $\leq$ if $A \leq B$ and $B \in \mathcal{C}$ implies $A \in \mathcal{C}$.
- Let $\leq$ be a reducibility among problems and $\mathcal{C}$ a class of problems closed under $\leq$
  - Problem B is hard for class $\mathcal{C}$ under $\leq$ if, for each $A \in \mathcal{C}$, $A \leq B$.
  - Problem B is complete for class $\mathcal{C}$ under $\leq$ if, B is hard for $\mathcal{C}$ and $B \in \mathcal{C}$.
- P, NP, PSPACE, EXP are closed under $\leq$.

Completeness

- Let $\leq$ be a reducibility among problems and $\mathcal{C}$ a class of problems. $\mathcal{C}$ is closed under $\leq$ if $A \leq B$ and $B \in \mathcal{C}$ implies $A \in \mathcal{C}$.
- Let $\leq$ be a reducibility among problems and $\mathcal{C}$ a class of problems closed under $\leq$
  - Problem B is hard for class $\mathcal{C}$ under $\leq$ if, for each $A \in \mathcal{C}$, $A \leq B$.
  - Problem B is complete for class $\mathcal{C}$ under $\leq$ if, B is hard for $\mathcal{C}$ and $B \in \mathcal{C}$.
- P, NP, PSPACE, EXP are closed under $\leq$.
- A NP-complete problem is a problem complete for NP under $\leq$.

Completeness

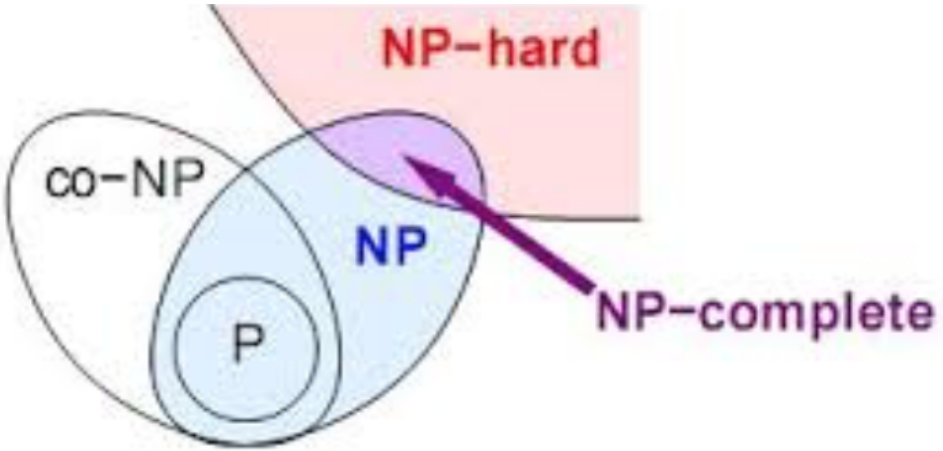
- Let $\leq$ be a reducibility among problems and $\mathcal{C}$ a class of problems. $\mathcal{C}$ is closed under $\leq$ if $A \leq B$ and $B \in \mathcal{C}$ implies $A \in \mathcal{C}$.
- Let $\leq$ be a reducibility among problems and $\mathcal{C}$ a class of problems closed under $\leq$
  - Problem B is hard for class $\mathcal{C}$ under $\leq$ if, for each $A \in \mathcal{C}$, $A \leq B$.
  - Problem B is complete for class $\mathcal{C}$ under $\leq$ if, B is hard for $\mathcal{C}$ and $B \in \mathcal{C}$.
- P, NP, PSPACE, EXP are closed under $\leq$.
- A NP-complete problem is a problem complete for NP under $\leq$.
- $\leq$ is a transitive relation.

NP-completeness

A problem A is NP-complete if:

1. $A \in$ NP, and
2. for every $B \in$ NP, $B \leq A$.

If for every $B \in$ NP, $B \leq A$ but $A \notin$ NP then A is said to be NP-hard.



Lemma

If A is NP-complete, then $A \in P$ iff $P = NP$.

Lemma

If A is NP-complete, then $A \in P$ iff $P = NP$.

- Once we prove that a problem is NP-complete, either A has no efficient algorithm or all NP problems are in P.

Lemma

If A is NP-complete, then $A \in P$ iff $P = NP$.

- Once we prove that a problem is NP-complete, either A has no efficient algorithm or all NP problems are in P.
- *Majority conjecture:* $P \neq NP$

Hard problems

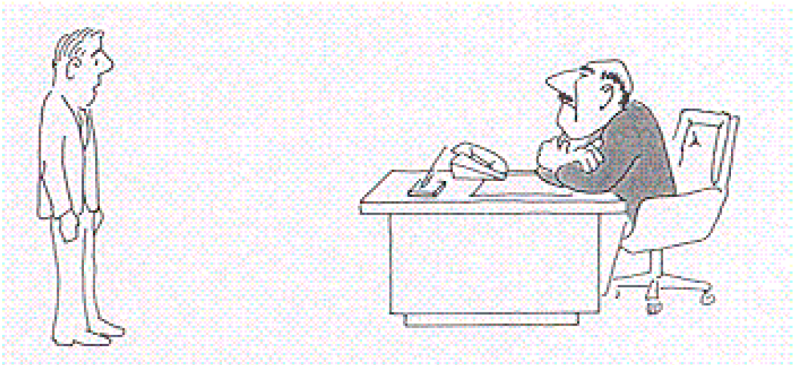
Hard problems

- To prove a problem is NP-hard, we just have to find a reduction from a problem known to be NP-complete/hard.

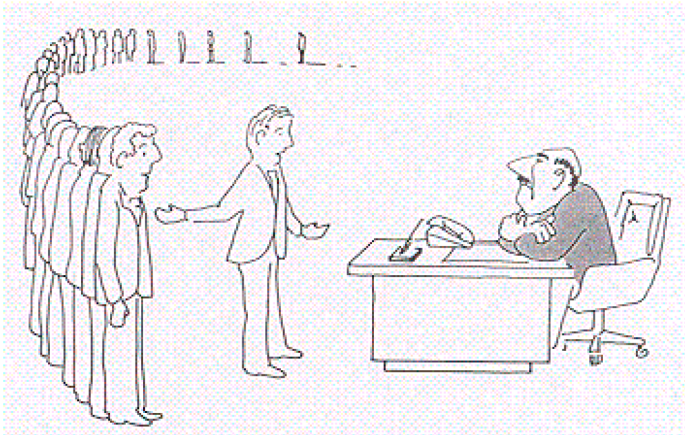
Hard problems

- To prove a problem is NP-hard, we just have to find a reduction from a problem known to be NP-complete/hard.
- We need as a seed a first NP-complete problem.

Cartoons from Garey-Johnson's book



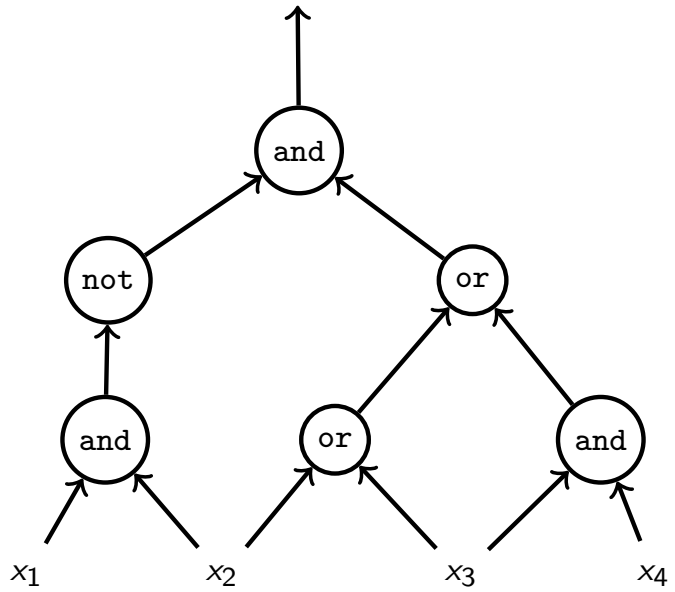
"I can't find an efficient algorithm,
I guess I'm just too dumb."



"I can't find an efficient algorithm,
but neither can all these famous people."

CIRCUIT SAT.

CIRCUIT SAT: Given a Boolean circuit with gates *AND*, *OR*, *NOT*, the input gates, and one output gate. Is there an assignment of 0/1 to the input gates, such that the circuit evaluates to 1?



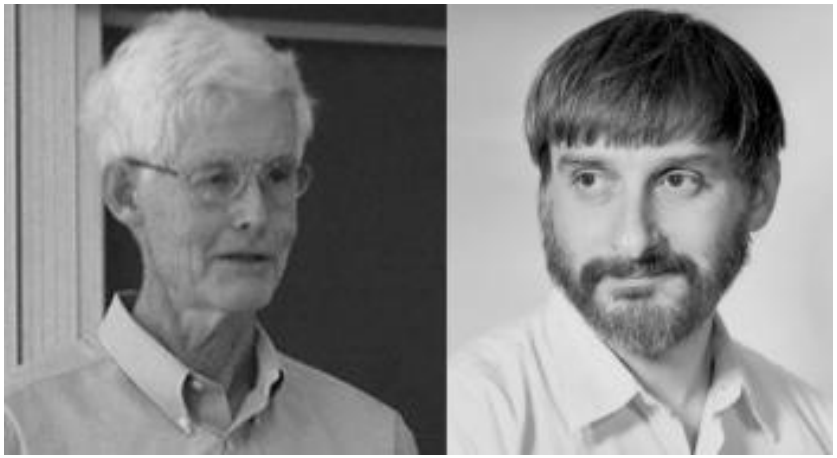
For example, if the input is $(1, 1, 0, 1)$ the output is 0.
if the input is $(1, 0, 1, 1)$ the output is 1.

CIRCUIT SAT in NP

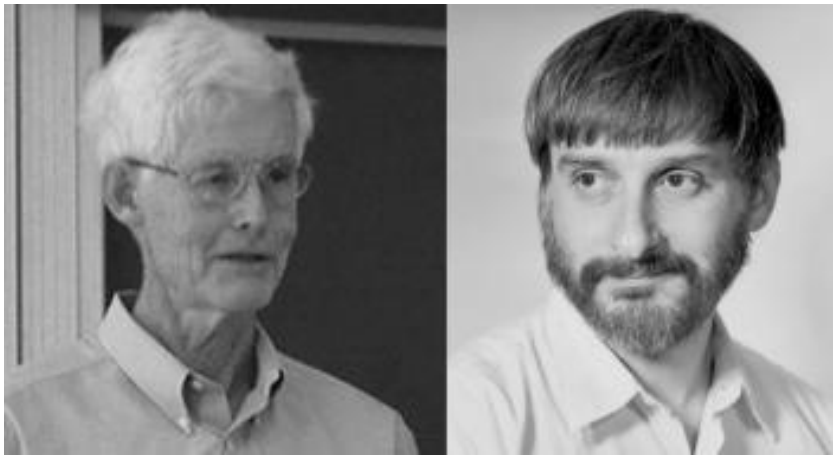
- CIRCUIT SAT belongs to NP, as there is a polynomial time algorithm that, given the circuit and an assignment of values to the inputs, computes the corresponding output.
  The certificate is the assignment which has polynomial size.

- To prove hardness We have to show that if $A \in$ NP, then $A \leq$ CIRCUIT SAT.

The seminal theorem: Cook-Levin's Theorem



The seminal theorem: Cook-Levin's Theorem



Theorem (Cook-Levin's theorem 1971)

CIRCUIT-SAT is NP-complete.

Any NP problem can be "expressed" as CIRCUIT SAT.

Any NP problem can be "expressed" as CIRCUIT SAT.

- A is a NP problem $\Rightarrow$ there is a polynomial-time TM M which given an instance x and a witness solution c of A, checks in polynomial time (in the length of $|x|$) if c is a valid certificate for x.

Any NP problem can be "expressed" as CIRCUIT SAT.

- A is a NP problem $\Rightarrow$ there is a polynomial-time TM M which given an instance x and a witness solution c of A, checks in polynomial time (in the length of $|x|$) if c is a valid certificate for x.
- A computation of a polynomial time decider D on input y of size n, can be expressed as a polynomial-size circuit C_D whose inputs are y. Furthermore, C_D on input y outputs 1 iff D accepts y.

Any NP problem can be "expressed" as CIRCUIT SAT.

- A is a NP problem $\Rightarrow$ there is a polynomial-time TM M which given an instance x and a witness solution c of A, checks in polynomial time (in the length of $|x|$) if c is a valid certificate for x.
- A computation of a polynomial time decider D on input y of size n, can be expressed as a polynomial-size circuit C_D whose inputs are y. Furthermore, C_D on input y outputs 1 iff D accepts y.
  To get the final circuit $C_{(M,x)}$, from M and x, we construct C_M and add a circuit that generates the bits of x in order. This leaves a s input the part corresponding to the certificate.
- Then, $x \in A$ iff $C_{(M,x)}$ is satisfiable.

Turing reductions

- *Problem A is* Turing reducible *to problem B ($A \leq_T B$)*
  if there is an algorithm solving A, using calls to an algorithm solving B.

Turing reductions

- *Problem A is* Turing reducible *to problem B ($A \leq_T B$)*
  if there is an algorithm solving A, using calls to an algorithm solving B.
- *Problem A is* poly time Turing reducible *to problem B ($A \leq_T^p B$)*
  if there is an algorithm solving A, using calls to an algorithm solving B, that runs in polynomial time assuming that a call to problem B takes time $O(1)$.

Turing reductions

- *Problem A is* Turing reducible *to problem B ($A \leq_T B$)*
  if there is an algorithm solving A, using calls to an algorithm solving B.
- *Problem A is* poly time Turing reducible *to problem B ($A \leq_T^p B$)*
  if there is an algorithm solving A, using calls to an algorithm solving B, that runs in polynomial time assuming that a call to problem B takes time $O(1)$.

If $A \leq_T B$

- B is decidable $\Rightarrow$ A is decidable.
- A is undecidable $\Rightarrow$ B is undecidable.

Turing reductions

- *Problem A is* Turing reducible *to problem B ($A \leq_T B$)*
  if there is an algorithm solving A, using calls to an algorithm solving B.
- *Problem A is* poly time Turing reducible *to problem B ($A \leq_T^p B$)*
  if there is an algorithm solving A, using calls to an algorithm solving B, that runs in polynomial time assuming that a call to problem B takes time $O(1)$.

Turing reductions

- *Problem A is* Turing reducible *to problem B ($A \leq_T B$)*
  if there is an algorithm solving A, using calls to an algorithm solving B.
- *Problem A is* poly time Turing reducible *to problem B ($A \leq_T^p B$)*
  if there is an algorithm solving A, using calls to an algorithm solving B, that runs in polynomial time assuming that a call to problem B takes time $O(1)$.

If $A \leq_T^p B$

- B is polynomial time solvable $\Rightarrow$ A is polynomial time solvable.
- A is NP-hard $\Rightarrow$ B is NP-hard.