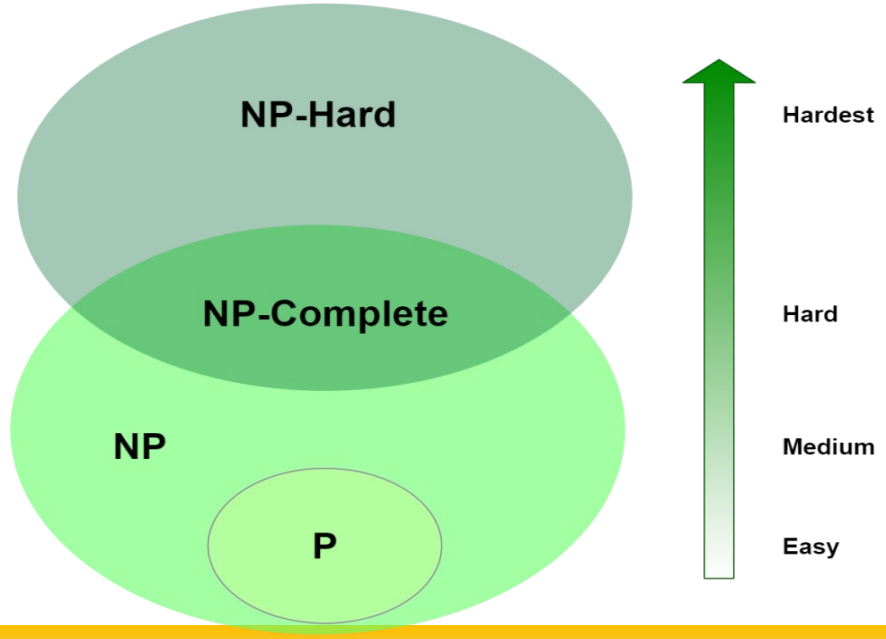


NP-Completeness

Dr. Bibhudatta Sahoo

Communication & Computing Group

CS235, Department of CSE, NIT Rourkela

Email: bdsahu@nitrkl.ac.in, 9937324437, 2462358

Introduction

- ***"Computers and Intractability: A Guide to the Theory of NP-Completeness " by*** Michael Garey and David S. Johnson; Freeman, New York, 1979.
- Scott Aaronson, a complexity researcher at MIT, explains his intuition about P vs. NP:
- ***"If P were equal to NP, then the world would be a profoundly different place than we usually assume it to be. There would be no special value in "creative leaps", no fundamental gap between solving a problem and recognizing the solution once it's found. Everyone who could appreciate a symphony would be Mozart; Everyone who could follow a set-by-step argument would be Gauss."***

Problem Taxonomy

Problem Taxonomy

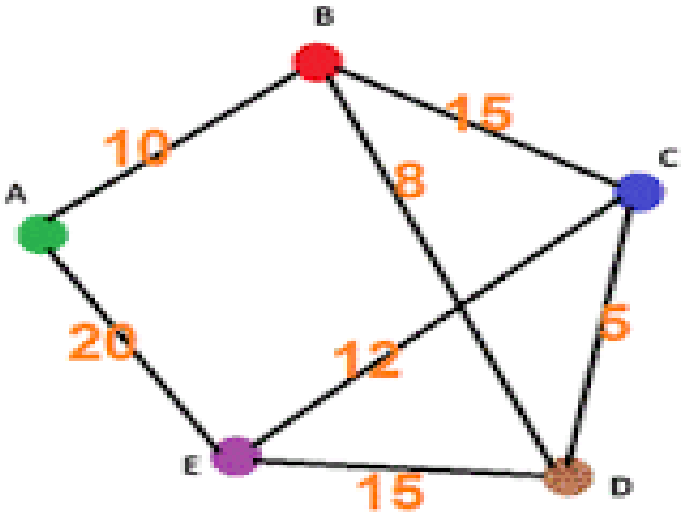
- **Tractable problem:** Have good algorithms with polynomial time complexity (irrespective of the degree)
- **Tractable Problem**: A problem that is solvable by a polynomial-time algorithm.
- **Intractable problem:** Have algorithms with super polynomial time complexity.
- **Intractable Problem**: A problem that cannot be solved by a polynomial-time algorithm.
- **Un-decidable Problem:** Do not have algorithms of any known complexity (polynomial or super-polynomial).
-

Intractable problems

- Intractable problems are problems for which there exist no efficient algorithms to solve them.
- Most intractable problems have an algorithm – the *same* algorithm – that provides a solution, and that algorithm is the brute-force search.
- This algorithm, however, does not provide an *efficient* solution and is, therefore, not feasible for computation with anything more than the smallest input.
- The reason there is no efficient algorithms for these problems is that these problems are all in a category which I like to refer to as "slightly less than random."
- They are so close to random, in fact, that they do not as yet allow for any meaningful algorithm other than that of brute-force.
- If any problem were *truly* random, there would not be even the possibility of any such algorithm.

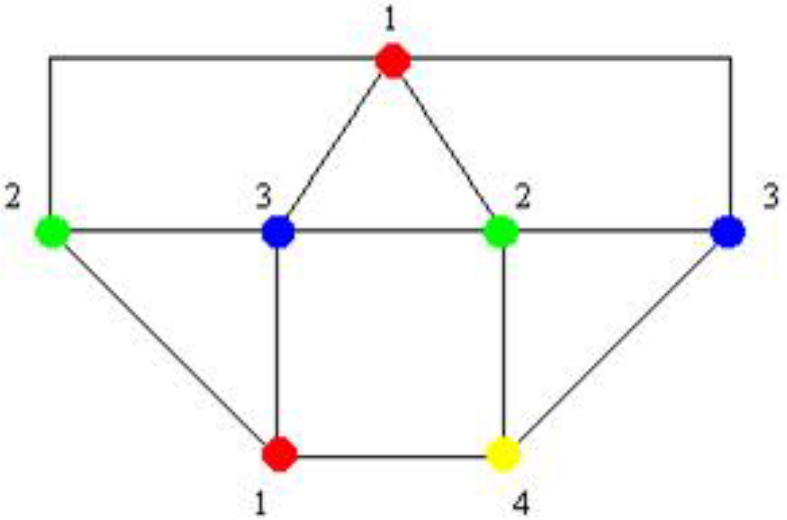
Example of intractable problem: Travelling Salesman Problem

- One example of an intractable problem, you **have to travel from the starting city to all cities on the map and back to the starting city**, for the lowest cost.
- To calculate the number of possible routes, one can find the factorial of n-1 to tell, how many routes there are, however you can divide this by 2 because each route has an identical route in reverse:
- So 5 cities will be: 4! = 24, 24 / 2 = 12 routes
- So 10 cities will be: 9! = 362880, 362880 / 2 = 181440 routes
- So 12 cities will be: 11! = 39916800, 39916800 / 2 = 19958400 routes



Example of intractable problem: **Graph coloring**

- Graph coloring : How many colors do you need to color a graph such that no two adjacent vertices are of the same color?
- Given any graph with a large number of vertices, we see that we are again faced with resorting to a systematic tracing of all paths, comparison of neighboring colors, backtracking, etc., resulting



Complexity of Algorithm: input size

- When we are determining whether an algorithm is **polynomial-time**, it is necessary to be careful about what we call the *input size*

Definition

- For a given algorithm, the ***input size*** is defined as the number of characters it takes to write the input.

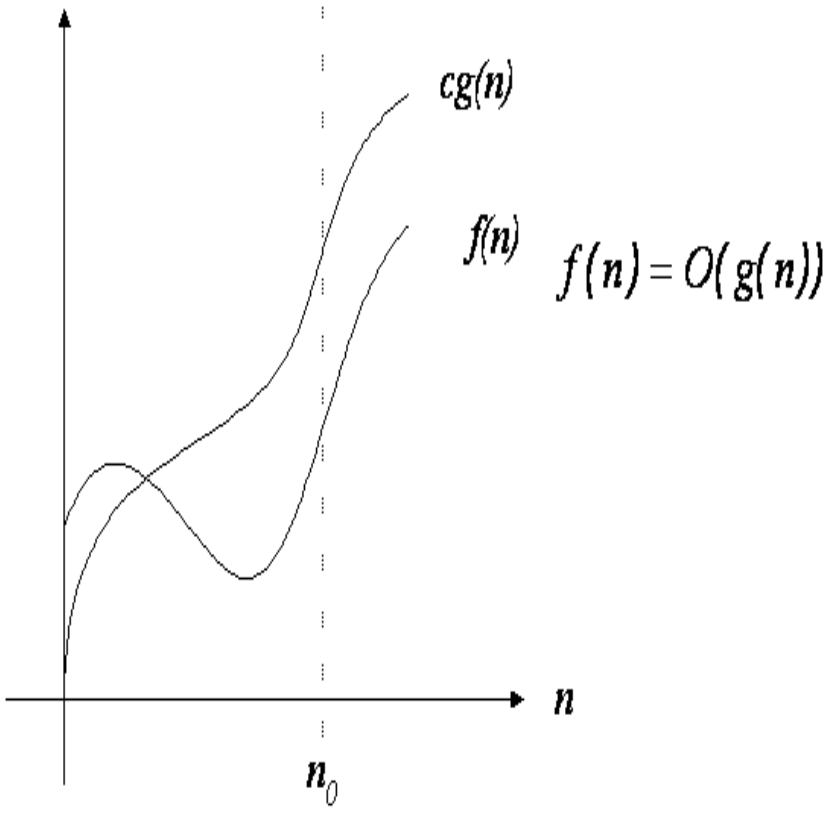
- int a[5][5], b[5][5], c[5][5];

- Total 3×5×5×2 =150bytes

P: Polynomial

- A simple program fragment

```
1. int
2. sum(int n)
3. {
4. int i, partialsum;
5. partialsum = 0;
6. for( i=1; i<= n; i++)
7. partialsum + = i*i*i;
8. return partialsum
9. }
```



6n+4 = O(n)

Recall the Hierarchy of Growth Rates

Easy

$$c < \log^k N < N < N \log N < N^2 < N^3 <$$

$$2^N < 3^N < N! < N^N$$

Hard

We can make a distinction between problems that have polynomial time algorithms and those that have algorithms that are worse than polynomial time.

Three general categories of intractable problems

1. Problems for which polynomial-time algorithms have been found
2. Problems that have been proven to be intractable
3. Problems that have not been proven to be intractable, but for which polynomial-time algorithms have never been found.

It is a surprising phenomenon that most problems in computer science seem to fall into either the first or third

Mathematical notation	Name	Tractable / Intractable
$n!$	*Exponential time*	*Intractable*
2^n	*Exponential time*	*Intractable*
n^2	*Polynomial time*	*Tractable*
n	*Linear time*	*Tractable*
$\log n$	*Logarithmic time*	*Tractable*

Different type of decidable Problem

- **Decision Problems:** The class of problems, the output is either yes or no.
- **Example** : Whether a given number is prime?

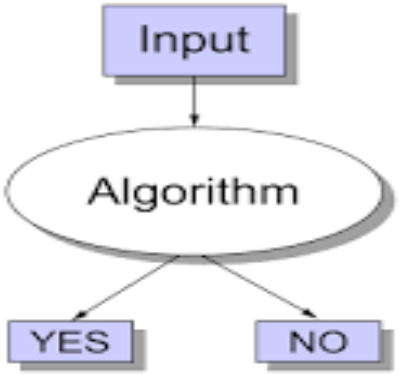
- **Counting Problem:** The class of problem, the output is a natural number
- **Example** :How many distinct factor are there for a given number

- **Optimization Problem:** The class of problem with some objective function based on the problem instance
- **Example** :Finding a minimal spanning tree for a weighted graph

Decidable problem

- A **decision problem** that can be solved by an algorithm that halts on all inputs in a finite number of steps.
- A problem is said to be decidable if we can always construct a corresponding algorithm that can answer the problem correctly.
- These problems are termed as **Turing Decidable** since a Turing machine always halts on every input, accepting or rejecting it.

Examples: A classic example of a decidable decision problem is **the set of prime numbers**. It is possible to effectively decide whether a given natural number is prime by testing every possible nontrivial factor.



Un-decidable Problem

- **Un-decidable Problem:** Do not have algorithms of any known complexity (polynomial or super-polynomial).
- A problem is undecidable if there is no Turing machine which will always halt in finite amount of time to give answer as 'yes' or 'no'. An undecidable problem has no algorithm to determine the answer for a given input.
- An NP-hard is a problem that is at least as **hard as any NP-complete problem**. Therefore an undecidable problem can be NP-hard.
- A problem is NP-hard if an oracle for it would make solving NP-complete problems easy (i.e. solvable in polynomial time).

Undecidable Problem

- Undecidable Problems intuitively by considering **Fermat's Theorem**, a popular Undecidable Problem which states that no three positive integers a, b and c for any n>2 can ever satisfy the equation: $a^n + b^n = c^n$.

- If we feed this problem to a Turing machine to find such a solution which gives a contradiction then a Turing Machine might run forever, to find the suitable values of n, a, b and c. But we are always unsure whether a contradiction exists or not and hence we term this problem as an **Undecidable Problem**.

Example : Undecidable Problems

Whether a CFG generates all the strings or not?

- As a CFG generates infinite strings, we can't ever reach up to the last string and hence it is Undecidable.

Whether two CFG L and M equal?

- Since we cannot determine all the strings of any CFG, we can predict that two CFG are equal or not.

Ambiguity of CFG?

- There exist no algorithm which can check whether for the ambiguity of a CFL. We can only check if any particular string of the CFL generates two different parse trees then the CFL is ambiguous.

Is it possible to convert a given ambiguous CFG into corresponding non-ambiguous CFL?

- It is also an Undecidable Problem as there doesn't exist any algorithm for the conversion of an ambiguous CFL to non-ambiguous CFL.

Is a language Learning which is a CFL, regular?

- This is an Undecidable Problem as we can not find from the production rules of the CFL whether it is regular or not.

Three categories of problems that can be solved on the computer

CATEGORY-I:

- **Those problems for which there is a polynomial time-algorithm that will produce a solution to all instances of the problem of input size n with a worst-case running time that is $O(n^k)$ for some constant k.**
- **All problems in category 1, are in a class of problems P or *The class of problems for which there is a polynomial-time solution.***

Three categories of problems that can be solved on the computer

CATEGORY-II

- **Those problems for which there is no known polynomial-time algorithm for finding a solution to all instances of the problem, but for which there are algorithms that will either find approximate solutions in polynomial-time, or will find a solution to an instance of the problem in polynomial-time with a probability p (find solutions some of the time).**

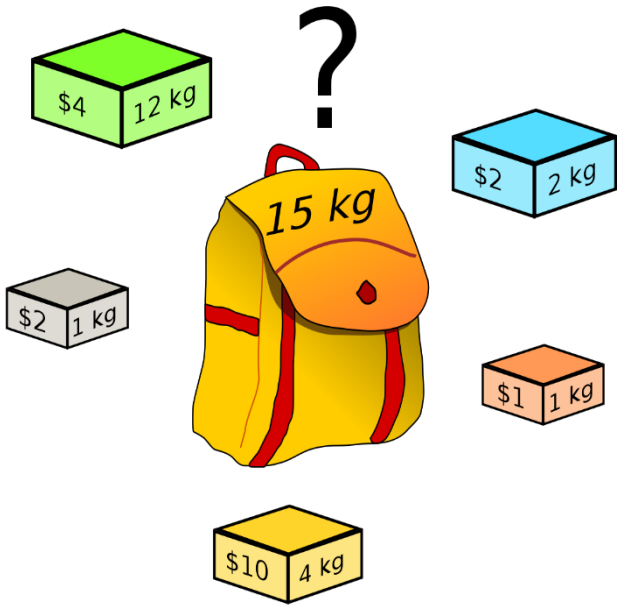
CATEGORY - III:

- **Those problems that are provably intractable (for which there can not be a polynomial-time algorithm for finding a solution.)**

0-1 Knapsack Problem is in which category

0-1 Knapsack Problem

Given a set of items, each with a weight and benefit, determine the items to include in a collection so that the total weight is less than or equal to a given weight limit and the total benefit is maximized



0-1 Knapsack Problem

Given a knapsack of capacity C and n objects of volumes $\{w_1, w_2 \ldots w_n\}$ and profits $\{p_1, p_2 \ldots p_n\}$, the objective is to choose a subset of n objects that fits into the knapsack and that maximizes the total profit.

In more formal terms, let x_i be 1 if object i is present in the subset and 0 otherwise.

The knapsack problem can be stated as

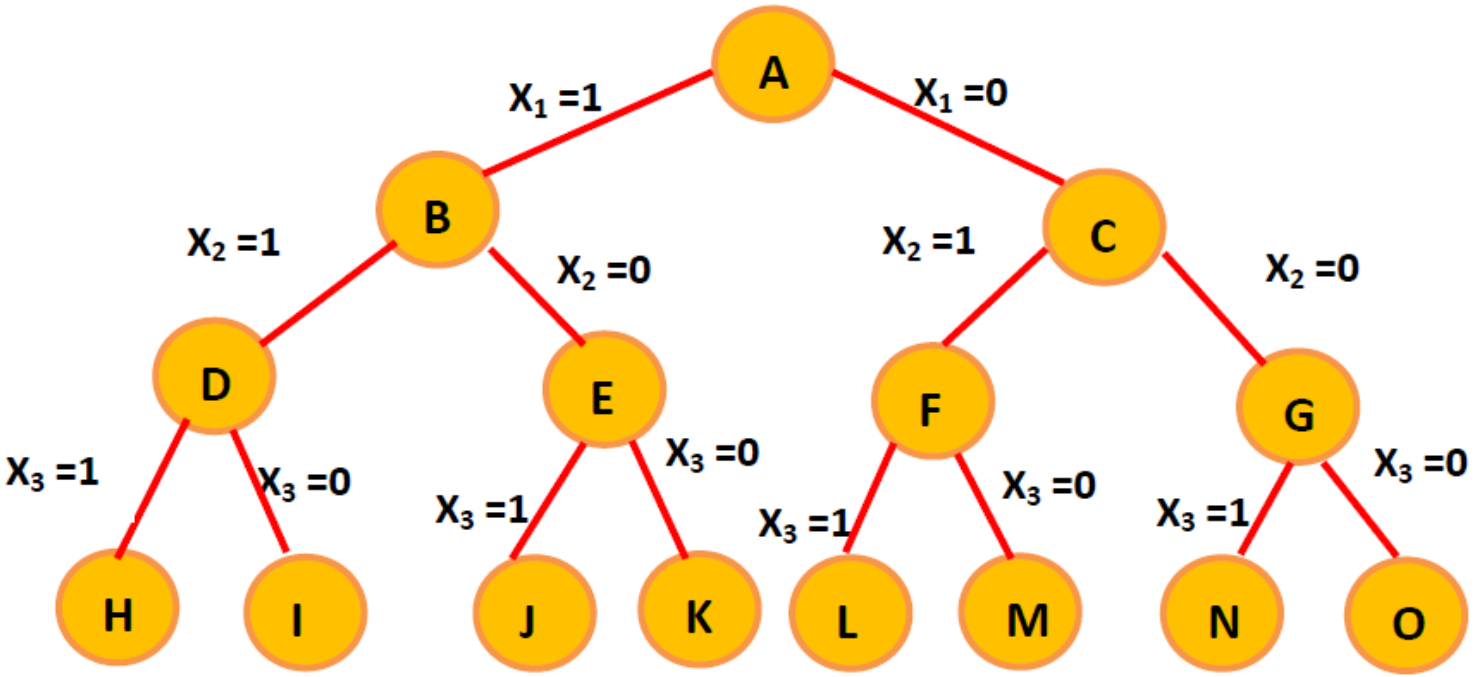
$$\text{Maximize} \sum_{i=0}^{n} x_i \cdot p_i \quad \text{subject to} \quad \sum_{i=0}^{n} x_i \cdot w_i \leq C$$

Note that the constraint $x_i \in \{0, 1\}$ is not linear. A simplistic approach will be to enumerate all subsets and select the one that satisfies the constraints and maximizes the profits. Any solution that satisfies the capacity constraint is called a *feasible* solution. The obvious problem with this strategy is the running time which is at least 2^n corresponding to the power-set of n objects.

0-1 Knapsack Problem

- C = 16, W={3,5,9,5}, P={45,30,45,10}
- S ={000, 001, 010, 011, 100, 101, 110, 111}
- Brut force search requires O(2^n) = 8

In this state space diagram 8 leaves, 8 paths and 8 solutions



Running times for different sizes of input.

n	$\log n$	n	$n \log n$	n^2	n^3	2^n
8	3 nsec	0.01 μ	0.02 μ	0.06 μ	0.51 μ	0.26 μ
16	4 nsec	0.02 μ	0.06 μ	0.26 μ	4.10 μ	65.5 μ
32	5 nsec	0.03 μ	0.16 μ	1.02 μ	32.7 μ	4.29 sec
64	6 nsec	0.06 μ	0.38 μ	4.10 μ	262 μ	5.85 cent
128	0.01 μ	0.13 μ	0.90 μ	16.38 μ	0.01 sec	10^{20} cent
256	0.01 μ	0.26 μ	2.05 μ	65.54 μ	0.02 sec	10^{58} cent
512	0.01 μ	0.51 μ	4.61 μ	262.14 μ	0.13 sec	10^{135} cent
2048	0.01 μ	2.05 μ	22.53 μ	0.01 sec	1.07 sec	10^{598} cent
4096	0.01 μ	4.10 μ	49.15 μ	0.02 sec	8.40 sec	10^{1214} cent
8192	0.01 μ	8.19 μ	106.50 μ	0.07 sec	1.15 min	10^{2447} cent
16384	0.01 μ	16.38 μ	229.38 μ	0.27 sec	1.22 hrs	10^{4913} cent
32768	0.02 μ	32.77 μ	491.52 μ	1.07 sec	9.77 hrs	10^{9845} cent
65536	0.02 μ	65.54 μ	1048.6 μ	0.07 min	3.3 days	10^{19709} cent
131072	0.02 μ	131.07 μ	2228.2 μ	0.29 min	26 days	10^{39438} cent
262144	0.02 μ	262.14 μ	4718.6 μ	1.15 min	7 mnths	10^{78894} cent
524288	0.02 μ	524.29 μ	9961.5 μ	4.58 min	4.6 years	10^{157808} cent
1048576	0.02 μ	1048.60 μ	20972 μ	18.3 min	37 years	10^{315634} cent

Note: “nsec" stands for nanoseconds, “ μ”is one microsecond and “cent" stands for centuries. The explosive running time (measured in centuries) when it is of the order 2^n.

0-1 Knapsack Problem is NP-Complete

- The decision version of the 0-1 knapsack problem is an **NP-Complete** problem. Given weights and values of items, and , respectively, can a subset of items be picked that satisfy the following constraints:

$$\text{Maximize} \sum_{i=0}^{n} x_i \cdot p_i \quad \text{subject to} \quad \sum_{i=0}^{n} x_i \cdot w_i \leq C$$

- A 'Yes' or 'No' solution to the above decision problem is NP-Complete.
- Solving the above inequalities is the same as solving the **Subset-Sum Problem**, which is proven to be NP-Complete. Therefore, the knapsack problem can be **reduced** to the Subset-Sum problem in polynomial time

0-1 Knapsack Problem is NP-Complete

- Further, the complexity of this problem depends on the size of the input values , weights and profit associated with the item.
- That is, if there is a way of rounding off the values making them more restricted, then we'd have a polynomial-time algorithm.
- This is to say that the non-deterministic part of the algorithm lies in the size of the input. **When the inputs are binary, it's complexity becomes exponential, hence making it an NP-Complete problem.**

Knapsack Problem is NP-complete

Theorem 1 *Knapsack is NP-complete.*

Proof: First of all, Knapsack is **NP**. The proof is the set S of items that are chosen and the verification process is to compute $\sum_{i \in S} s_i$ and $\sum_{i \in S} v_i$, which takes polynomial time in the size of input.

Second, we will show that there is a polynomial reduction from Partition problem to Knapsack. It suffices to show that there exists a polynomial time reduction $Q(\cdot)$ such that $Q(X)$ is a 'Yes' instance to Knapsack iff X is a 'Yes' instance to Partition. Suppose we are given $a_1, a_2, \ldots, a_n$ for the Partition problem, consider the following Knapsack problem: $s_i = a_i, v_i = a_i$ for $i = 1, \ldots, n$, $B = V = \frac{1}{2}\sum_{i=1}^{n} a_i$. $Q(\cdot)$ here is the process converting the Partition problem to Knapsack problem. It is clear that this process is polynomial in the input size.

If X is a 'Yes' instance for the Partition problem, there exists S and T such that $\sum_{i \in S} a_i = \sum_{i \in T} a_i = \frac{1}{2}\sum_{i=1}^{n} a_i$. Let our Knapsack contain the items in S, and it follows that $\sum_{i \in S} s_i = \sum_{i \in S} a_i = B$ and $\sum_{i \in S} v_i = \sum_{i \in S} a_i = V$. Therefore, $Q(X)$ is a 'Yes' instance for the Knapsack problem.

Conversely, if $Q(X)$ is a 'Yes' instance for the Knapsack problem, with the chosen set S, let $T = \{1, 2, ..n\} - S$. We have $\sum_{i \in S} s_i = \sum_{i \in S} a_i \leq B = \frac{1}{2}\sum_{i=1}^{n} a_i$, and $\sum_{i \in S} v_i = \sum_{i \in S} a_i \geq V = \frac{1}{2}\sum_{i=1}^{n} a_i$. This implies that $\sum_{i \in S} a_i = \frac{1}{2}\sum_{i=1}^{n} a_i$ and $\sum_{i \in T} a_i = \sum_{i=1}^{n} a_i - \frac{1}{2}\sum_{i=1}^{n} a_i = \frac{1}{2}\sum_{i=1}^{n} a_i$.

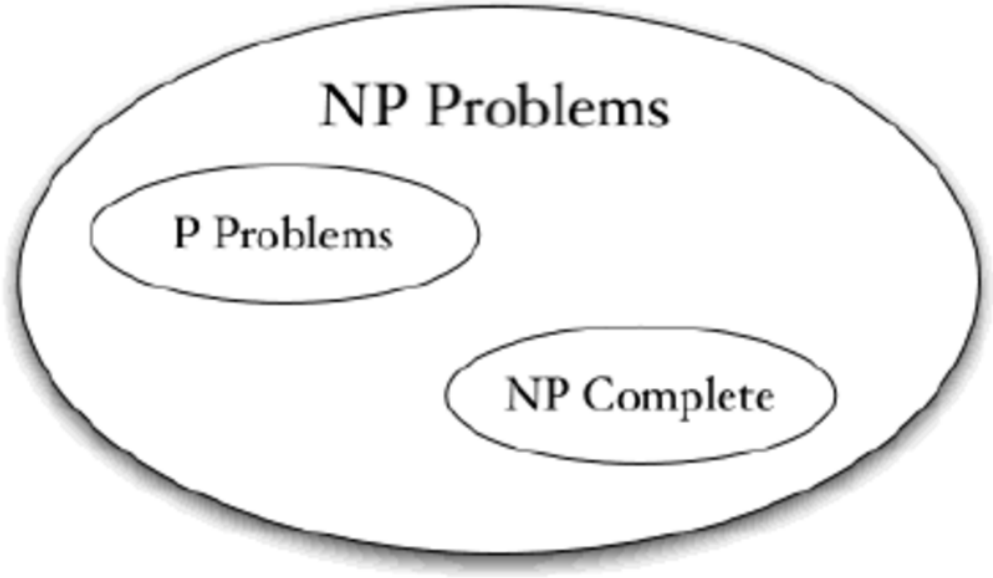
Therefore, $\{S, T\}$ is the desired partition, and X is a 'Yes' instance for the Partition problem. This establishes the **NP**-completeness of Knapsack problem. □

Show that 0-1 Knapsack Problem is NP-complete

- **Knapsack** is in **NP**: Clearly, Given an input set, it is very easy to check if the total weight is at most C and if the corresponding profit is at least P.
- **Knapsack** is **NP-hard**: We reduce the well-known **NP-hard problem** Subset-sum to **Knapsack problem**

The Class NP

"**Nondeterministic Polynomial Time**", meaning the class of problems that can be solved in polynomial time on a nondeterministic machine.

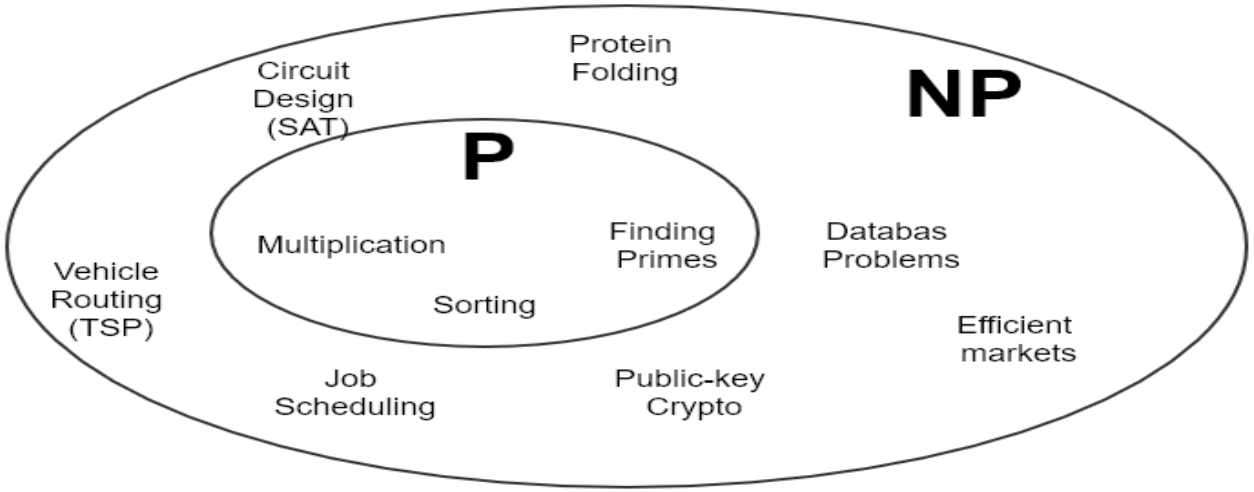


The Class NP

- P : Polynomial time
- NP : Nondeterministic polynomial time
- A deterministic machine, at each point in time, is executing an instruction.
  - Depending on instruction, it then goes to some next instruction, which is unique.
- A *non-deterministic machine* has a choice of next steps. It is free to choose any that it wishes, and if one of these steps leads to a solution, it will always choose the correct one.
- A *non-deterministic machine* thus has a power of extremely good(optimal) guessing

P Problem

- Problem for which there is a polynomial time algorithm are called **tractable**.
- A problem is assigned to the P represent the class of problems which can be solved by a deterministic Polynomial algorithm.



NP-Problem

- A problem is assigned to the NP (nondeterministic polynomial time) class if it is solvable in polynomial time by a **nondeterministic Turing machine**.
- A problem is assigned to the NP represents the class of **decision problem** which can be solved by a non-deterministic polynomial algorithm**.**
- A **P-problem** (whose solution time is bounded by a polynomial) is always also NP.
- If a problem is known to be NP, and a solution to the problem is somehow known, then demonstrating the correctness of the solution can always be reduced to a single P (polynomial time) verification.
- If P and NP are *not* equivalent, then the solution of NP-problems requires (in the worst case) an **exhaustive search**.

NP: Nondeterministic Polynomial Time

- **Does**

  3797659517717669537970249147937411727262759330195046268899636749366507845369942177663592040922984159043233985090696289604041707209619788051365080241649482160288592712696862946431304735342639520488192047545612916330509384696811968391223240543368805156786230378533714918428119696774380580083081544267990372093 3

  have a prime factor ending in 7?

Problem class NP

- A second major class of problems, the class to which most of the problems in our second category belong, is the class NP.
- These are the class of problems for which there is a polynomial-time algorithm for verifying that a "certificate", containing an instance of the problem and its solution, is correct.
- Definition: *Class NP* is the class of decision problems that that can be solved by nondeterministic polynomial algorithms. This class of problems is called **nondeterministic polynomial**
- The class P is a sub-class of NP (since any problem that can be solved in polynomial time can also be verified in polynomial-time, if only by solving the problem and comparing solutions).
- But NP also contains many problems that can be verified in polynomial-time but for which there is no known polynomial-time solution.

P vs. NP

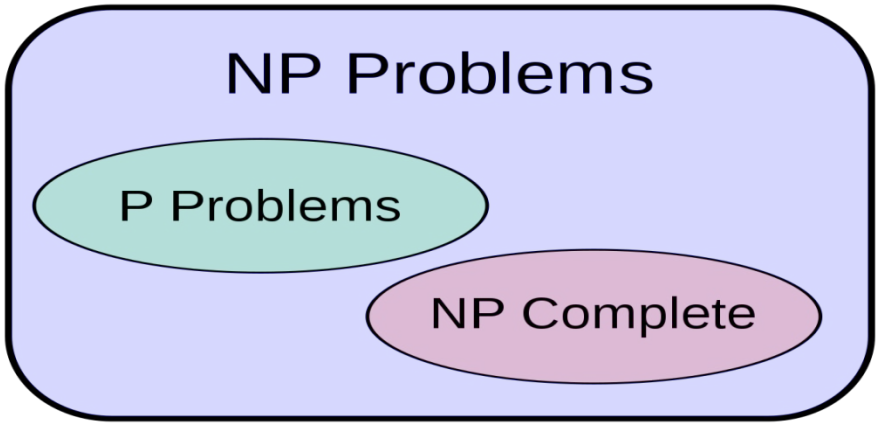
- Scott Aaronson, a complexity researcher at MIT, explains his intuition about **P vs. NP**:
- ***"If P were equal to NP, then the world would be a profoundly different place than we usually assume it to be. There would be no special value in "creative leaps", no fundamental gap between solving a problem and recognizing the solution once it's found. Everyone who could appreciate a symphony would be Mozart; Everyone who could follow a set-by-step argument would be Gauss."***

P and NP

- P corresponds to a class of problems that can be solved in polynomial time
- NP corresponds to a class of problems whose **solution can be checked in polynomial time**.

❑ **P=NP?**

- We know that P is a subset of NP
- Whether NP is also a subset of P, is not known.
- Probably false, but nobody has been able to prove.

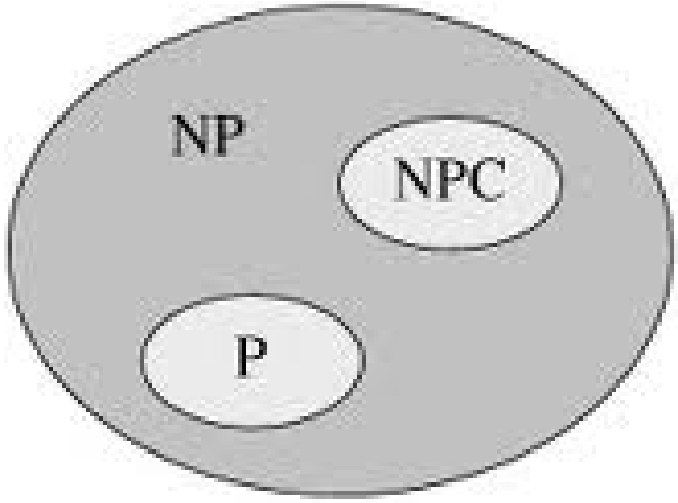


Class NP Problems

- Problems are "NP" if it is easy to check the correctness of a claimed solution. In other words the solution can be checked in polynomial time.
  - This doesn't say it is easy to find a solution.
  - In fact, it is often **hard** to find a solution!
  - Example problems include the *Traveling Salesman Problem* and the *Hamiltonian Circuit Problem*. Note it is easy to verify a solution.

NP complete Problems

- Among all the problems known to be NP, there is a subset known as NP-complete problems, which contains the hardest.
- An NP-complete problem has a property that any problem in NP can be ***polynomially reduced*** to it.



NP-complete problems

- A problem **P1** is NP complete, if every problem in NP can be **reduced** to P1 in polynomial time.
- That means if you solved **P1**, you solved every problem in NP.
- First problem that was proven to be NP-complete was **CNF satisfiability**. Given a logical expression in conjunctive normal form, is there an assignment for the logical variables for which the expression will evaluate to true.
- Cook proved that CNF-satisfiability is NP-complete
- Since then 1000's of problems have been shown to be NP-complete.
- The task of proving that a problem **P2** is NP-complete is much simpler than faced by Cook

NP-complete problems

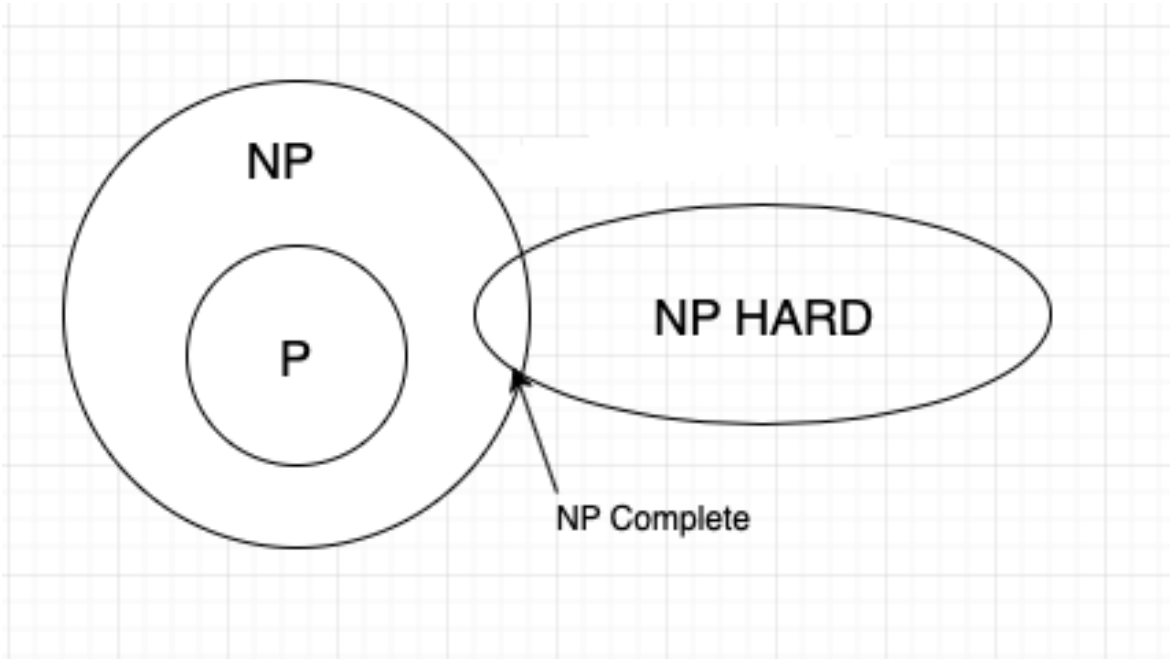
- You can pick another problem **P1** that is known to be **NP-complete** and show that P1 can be **reduced** to P2, then P2 is NP-complete.
- If you can show that **one NP-complete problem P** can be solved in polynomial time, you have proven that **P=NP**. Because all the other problems can be reduced (using a polynomial reduction) to P and solved in polynomial time.
- If you prove that one problem in NP has a lower bound which is exponential (non-polynomial) then you have proved that **P != NP**.
- Guaranteed question.

Polynomially reduced

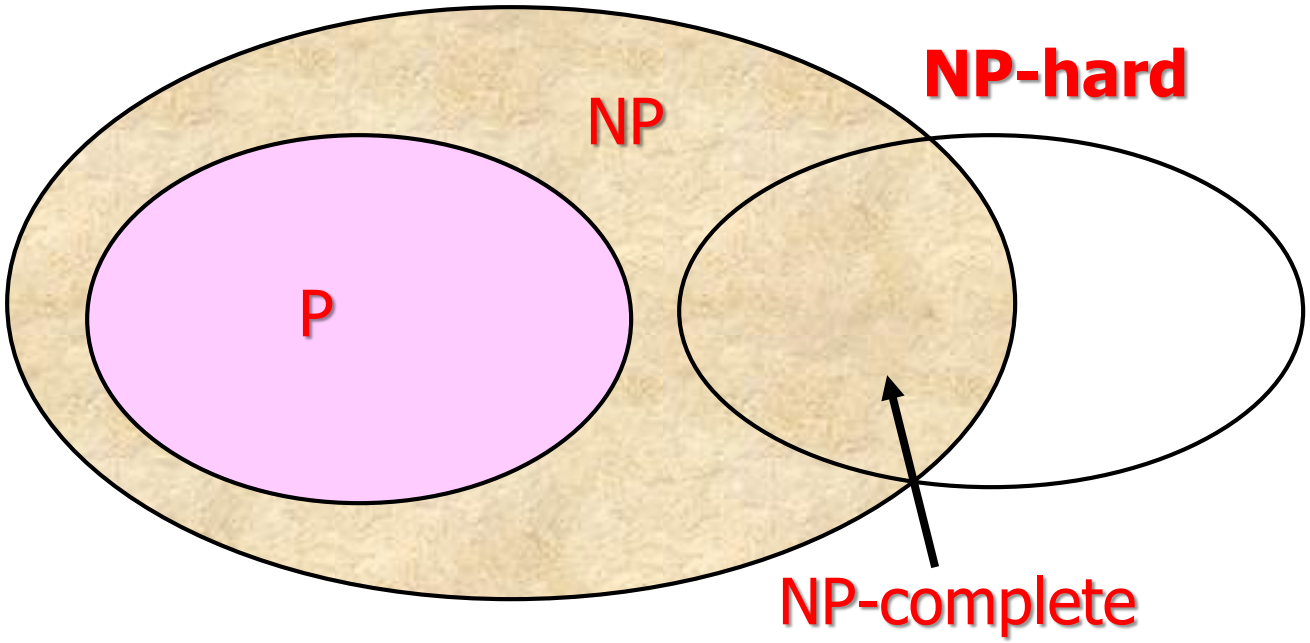
- A problem P1 can be **reduced** to P2 as follows:
- Provide a mapping so that any instance of P1 can be transformed to an instance of P2
- Solve P2, and then map the answer back to the original.
- ***Example:*** In a pocket calculator, the decimal numbers are converted to binary and all calculations are performed in binary, the final answer is converted back to decimal display
- P1 to be **polynomially reducible** to P2 means that , all the work associated with the transformation must be performed in polynomial time.

NP complete

- The group of problems what are both NP and NP hard is called as NP Complete Problem



P, NP, NP-Complete and NP-Hard Problems



NP-Completeness

- Poly time algorithm: input size n (in some encoding), worst case running time – $O(n^c)$ for some constant c.

- Three classes of problems
  - P: problems solvable in poly time.
  - NP: problems verifiable in poly time.
  - NPC: problems in NP and as hard as any problem in NP.

Relation among P, NP, NPC

- $P \subseteq NP$ (Sure)
- $NPC \subseteq NP$ (sure)
- $P = NP$ (or $P \subset NP$, or $P \neq NP$) ???
- $NPC = NP$ (or $NPC \subset NP$, or $NPC \neq NP$) ???
- $P \neq NP$: one of the deepest, most perplexing open research problems in (theoretical) computer science since 1971.

Arguments about P, NP, NPC

- No poly algorithm found for any NPC problem (even so many NPC problems)

- No proof that a poly algorithm cannot exist for any of NPC problems, (even having tried so long so hard).

- Most theoretical computer scientists believe that NPC is intractable (i.e., hard, and $P \neq NP$).

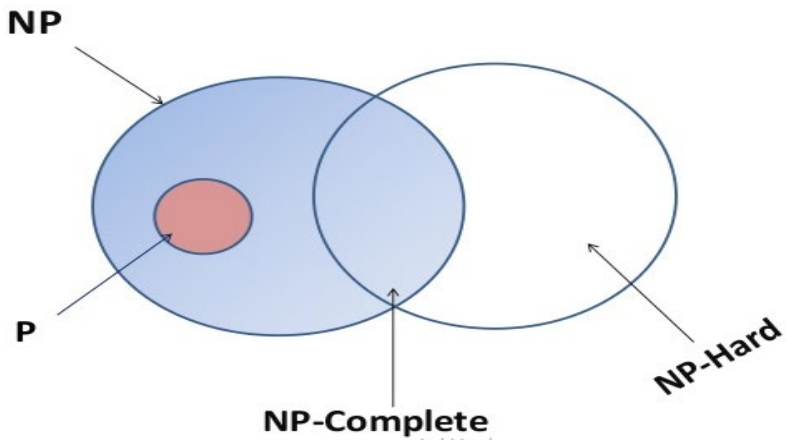
Commonly Believed Relationship between P, NP , NP-hard, NP-Complete

- Take two problems A and B both are **NP problems**.
- **Reducibility**- If we can convert **one instance** of a problem A into problem B (NP problem) then it means that A is reducible to B.

- **NP-hard**-- Now suppose we found that A is reducible to B, then it means that B is at least as hard as A.

- **NP-Complete** -- The group of problems which are both in NP and NP-hard are known as NP-Complete problem.

- Now suppose we have a NP-Complete problem R and it is reducible to Q then Q is at least as hard as R and since R is an NP-hard problem. therefore Q will also be at least NP-hard , it may be NP-complete also.



Even Harder Problems

- Are there problems harder than NP? Yes.
- One example is the “Halting Problem”:
  - Can you write a **program A** that can determine whether program B will terminate? Think of this as a souped-up compiler.
  - No, this turns out to be **impossible** to do.

Halting Problem

- It is natural to wonder whether all problems can be solved in polynomial time.
- The answer is no. For example, the ***Halting Problem***

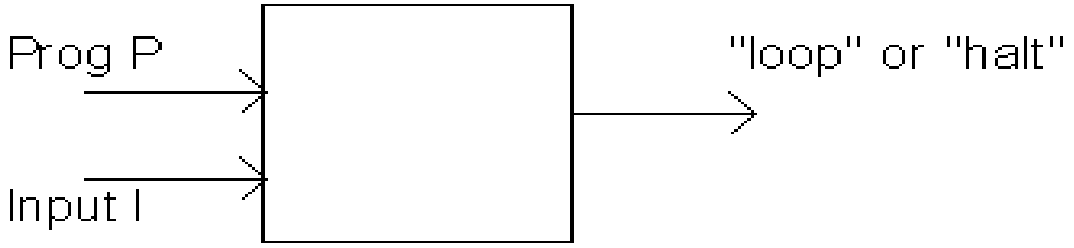
  The halting problem:

  "*Given a description of an arbitrary computer program, decide whether the program finishes running or continues to run forever*".

  This is equivalent to the problem of deciding, given a program and an input, whether the program will eventually halt when run with that input, or will run forever.

A proof that the Halting Problem is unsolvable

- The **Halting problem:** *Given a program and an input to the program, determine if the program will eventually stop when it is given that input.*
- This proof was devised by Alan Turing, 1936
- Suppose you have a solution to the halting problem called H. H takes two inputs: a program P and an input I for the program P.
- H generates an output "***halt***" if H determines that P stops on input I or it outputs "***loop***" otherwise

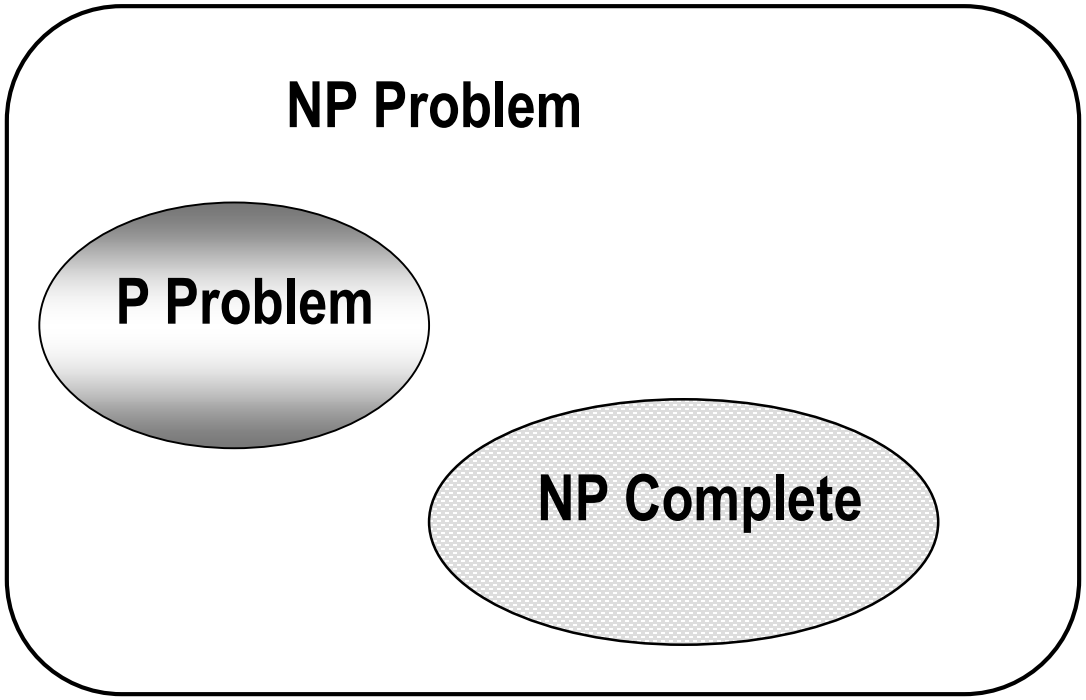


A proof that the Halting Problem is unsolvable

- http://jcsites.juniata.edu/faculty/rhodes/intro/theory2.htm

View of Theoretical Computer Scientists on P, NP, NPC

- $\mathbf{P \subset NP, NPC \subset NP, P \cap NPC = \varnothing}$

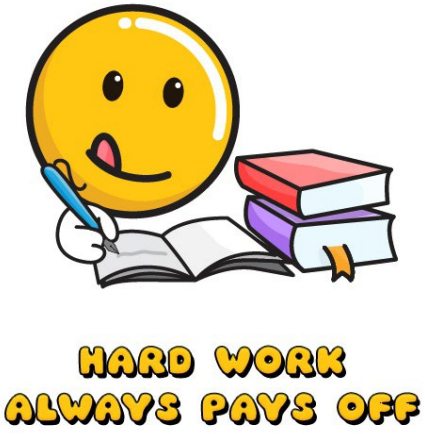


Why discussion on NPC

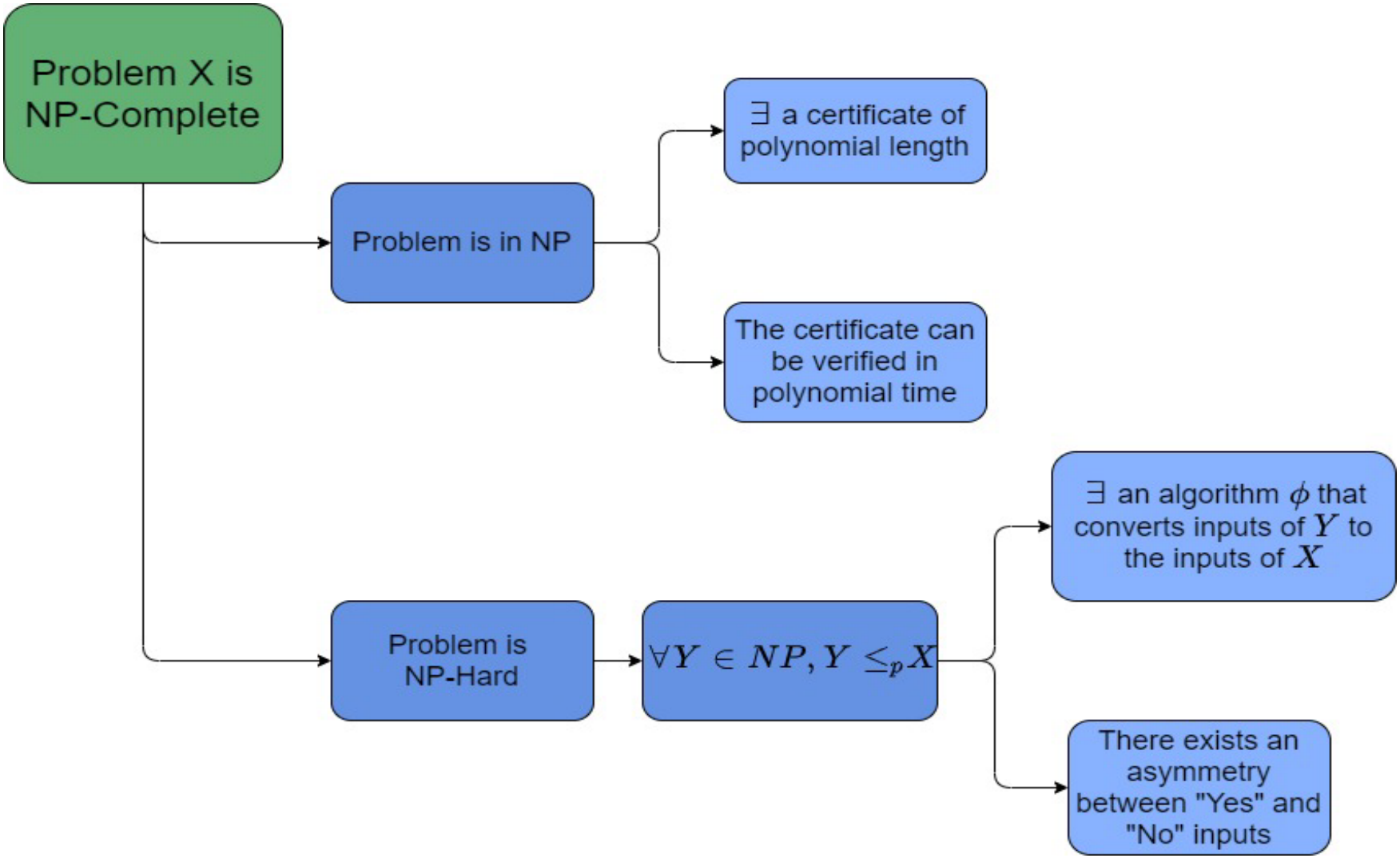
- If a problem is proved to be NPC, a good evidence for its intractability (hardness).
- Not waste time on trying to find efficient algorithm for it
- Instead, focus on design **approximate algorithm** or a solution for a **special case** of the problem
- Some problems looks very easy on the surface, but in fact, is hard (NPC).

How do we prove a problem is NP complete

- Given a problem U, the steps involved in proving that it is **NP-complete** are the following:
- Step 1: Show that U is in NP.
- Step 2: Select a known NP-complete problem V.
- Step 3: Construct a **reduction from** V to U.
- Step 4: Show that the reduction requires **polynomial time.**



How to Prove That a Problem Is NP-Complete?



Relevant facts about NP-complete

1. If any NP-complete has a polynomial time algorithm then they all do. Another way to think of this is that all pairs of NP-complete problems are reducible to each other via a reduction that runs in **polynomial time**.
2. There are thousands of known natural NP-complete problems from every possible area of application that people would like to find polynomial time algorithms for.
3. No one knows of a polynomial time algorithm for any NP-complete problem.
4. No one knows of a proof that no NP-complete problem can have a polynomial time algorithm.
5. Almost all computer scientists believe that NP-complete problems do not have **polynomial time algorithms**.

Relevant facts about NP-complete

6. If we can find a polynomial time algorithm for an NP-Complete problem, then all problems in NP have polynomial time algorithms.
7. If we can prove that one problem in NP can't be solved in polynomial time, then all NP-Complete problems are not solvable in polynomial time.
8. All NP-Complete problems appear to be difficult.
9. We are not aware of any polynomial time algorithms to solve them.
10. However, there is no proof that polynomial time algorithms do not exist!

Methods adopted to cope with NP-complete Problem

- Use dynamic programming, backtracking or branch-and-bound technique to reduce the computational cost. This might work if the problem size is not too big.
- Find the sub-problem of the original problem that have polynomial time solution.
- Use **approximation algorithms** to find approximate solution in polynomial time.
- Use **randomized algorithm** to find solutions is affordable time with a high probability of correctness of the solution.
- Use **heuristic** like greedy method, simulated annealing or genetic algorithms etc. However, the solutions produced cannot be guaranteed to be within a certain distance from the optimal solution.

NP-problem world

Source: Clay Mathematics Institute : *www.claymath.org*

Problem

- **Suppose that you are organizing housing accommodations for a group of four hundred university students. Space is limited and only one hundred of the students will receive places in the dormitory.**
- **To complicate matters, the Dean has provided you with a list of pairs of incompatible students, and requested that no pair from this list appear in your final choice.**
- **This is an example of what computer scientists call an NP-problem.**

Source: Clay Mathematics Institute : *www.claymath.org*

- **This is an example of what computer scientists call an NP-problem, since it is easy to check if a given choice of one hundred students proposed by a coworker is satisfactory (i.e., no pair taken from your coworker's list also appears on the list from the Dean's office), however the task of generating such a list from scratch seems to be so hard as to be completely impractical.**
- **Indeed, the total number of ways of choosing one hundred students from the four hundred applicants is greater than the number of atoms in the known universe! Thus no future civilization could ever hope to build a supercomputer capable of solving the problem by brute force; that is, by checking every possible combination of 100 students. However, this apparent difficulty may only reflect the lack of ingenuity of your programmer.**

Source: Clay Mathematics Institute : *www.claymath.org*

- **In fact, one of the outstanding problems in computer science is determining whether questions exist whose answer can be quickly checked, but which require an impossibly long time to solve by any direct procedure.**
- **Problems like the one listed above certainly seem to be of this kind, but so far no one has managed to prove that any of them really are so hard as they appear, i.e., that there really is no feasible way to generate an answer with the help of a computer.**
- **Stephen Cook and Leonid Levin formulated the P (i.e., easy to find) versus NP (i.e., easy to check) problem independently in 1971**

Source: Clay Mathematics Institute : *www.claymath.org*

- The **seven Millennium Prize** Problems were chosen by the founding Scientific Advisory Board of CMI, which conferred with leading experts worldwide. The focus of the board was on important classic questions that have resisted solution for many years.
- Follwing the decision of the Scientific Advisory Board, the Board of Directors of CMI designated a $7 million prize fund for the solution to these problems, with $1 million allocated to the solution of each problem.
- http://www.claymath.org/millennium/
    - **P vs NP Problem**

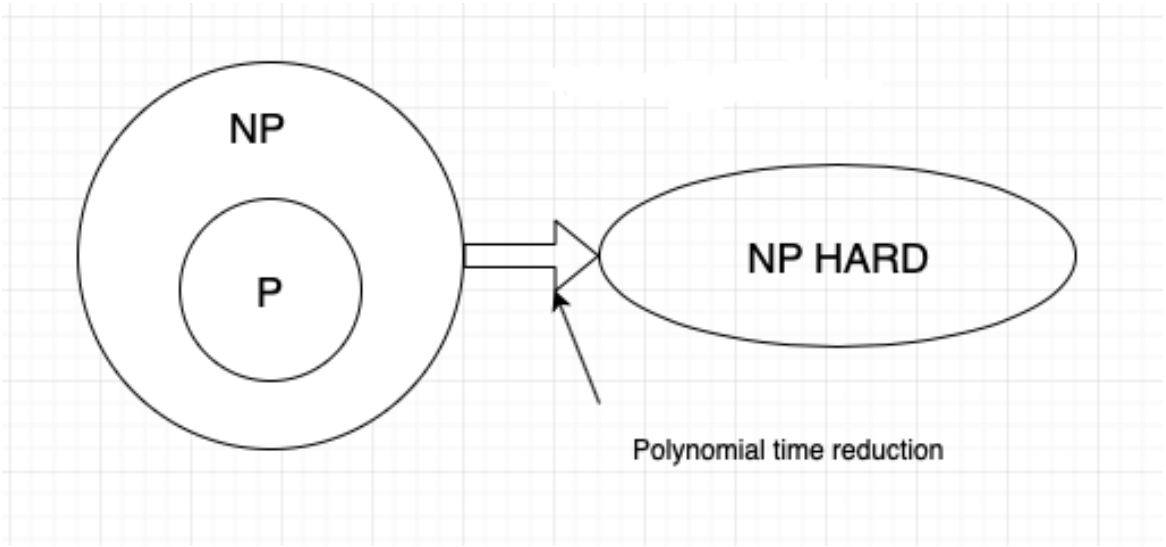
NP-Hard

NP hard problems

- **These are problems whose solution cannot be checked in polynomial time**.
- Typically, these are optimization problems
  - Can a given graph be colored with k-colors? (NP complete)
  - What are the minimum number of colors required to color a graph? (NP hard)
- If we can solve an **NP hard problem** in polynomial time, we can solve all the NP-complete problems in polynomial time (P=NP).
- *Note that just because you solved (**in your dreams**) an NP hard problem in polynomial time doesn't mean that you can solve every other problem in polynomial time.*

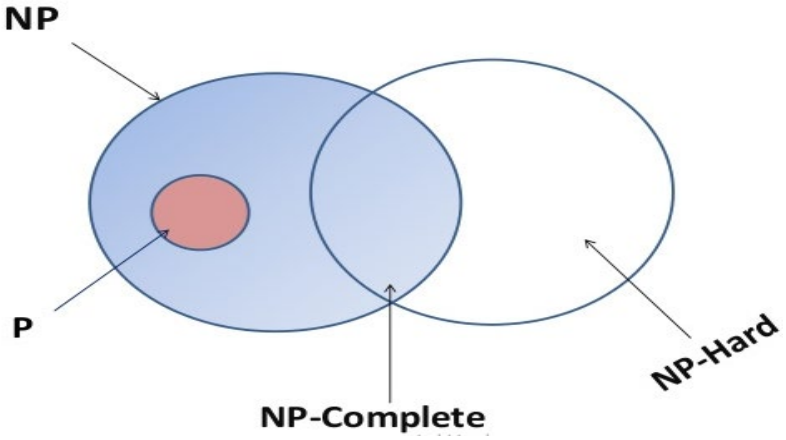
NP hard Problem:

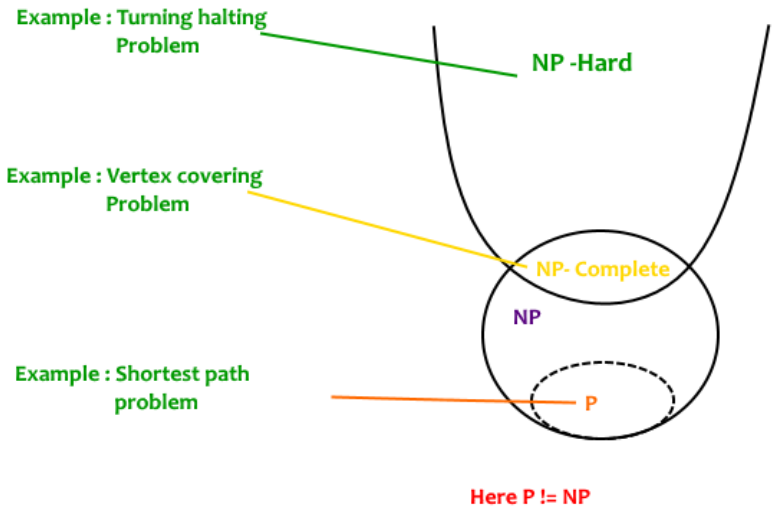
- A problem is called as NP hard, if all the problems in NP can be reduced to polynomial time.
- It means if I have a set of problems in P and NP, and they can be **reduced** in polynomial time, then those set of problems are called as NP based problems.



Example:

- If you can find the minimum number of colors required to color a graph (NP-hard) in polynomial time,
- you can definitely tell if the graph can be colored with k-colors (NP-complete) in polynomial times.
- Since you solved one NP-complete problem in polynomial time, you can solve all NP-complete problems in polynomial time.





NP hard problems

- A problem L is NP-hard if a polynomial time algorithm for L yields a polynomial time algorithm for any/all NP-complete problem(s), or equivalently, if there is an NP-complete problem C that polynomial time reduces to L.
- A problem L is NP-easy if a polynomial time algorithm for any NP-complete problem yields a polynomial time algorithm for L, or equivalently, if there is an NP-complete problem C such that L is polynomial time reducible to C.
- A problem is NP-equivalent, or for us NP-complete if it is both NP-hard and NP-easy.

NP hard problems

- NP-complete and NP-Hard problems are just the tip of the iceberg. There are many well defined problems for which we cannot even form an algorithm to solve.
- **Decidable** problems are those for which you can write an algorithm
- These problems are called **undecidable** (Turing)
    1. Halting problem
    2. Blank tape problem

NP hard problems

- The **Halting problem** Given a program and an input to the program, determine if the program will eventually stop when it is given that input.

- Bank tape halting problem is the problem of finding an algorithm that, for any Turing machine, decides whether the machine eventually stops if it started on an empty tape; it has been proved that no such algorithm exists

NP-Hard problems

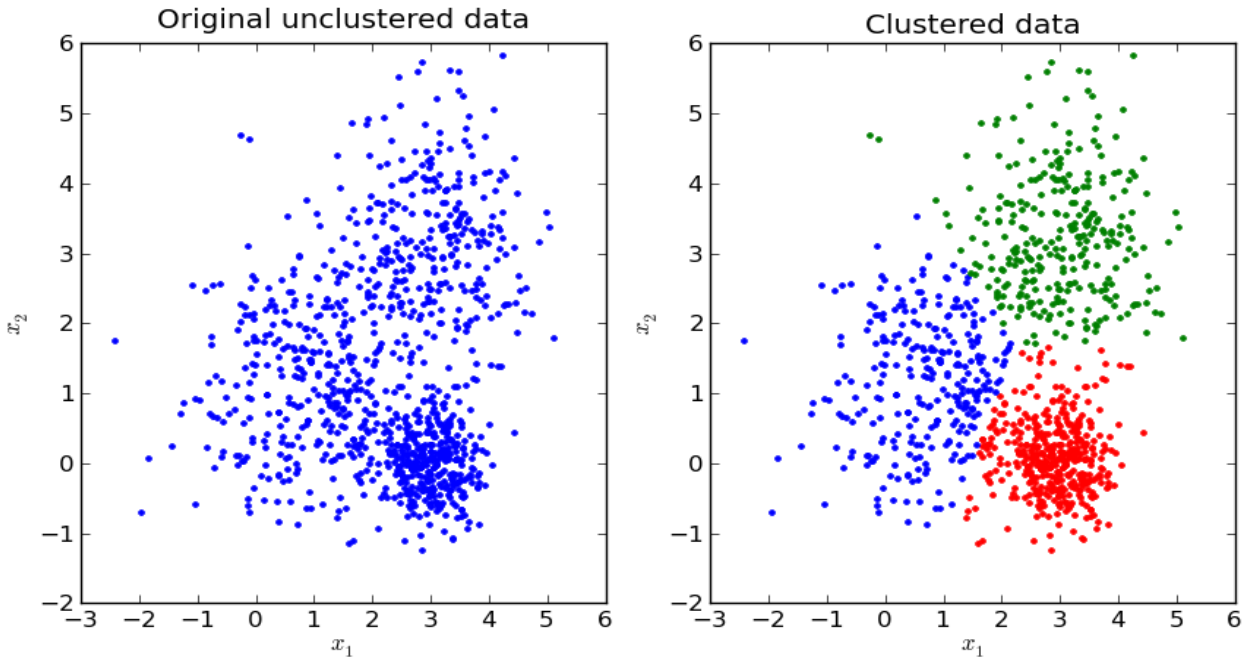
- NP-hard problems are Most complex problems in computer science. They are not only hard to solve but are hard to **verify** as well. In fact, some of these problems aren't even decidable.

Example of the hardest computer science problems are:

- **K-means Clustering**
- **Traveling Salesman Problem,**
- **Graph Coloring**

The hardness of k-means clustering

- K-Means is a famous **hard** clustering algorithm whereby the data items are clustered into K clusters such that each item only blogs to one cluster.
- K-Means is NP-Hard even on a plane because, if you solve it in P, you can also solve planar 3-SAT in P. Planar 3-SAT has already been proved to be NP-Hard, and thus K-Means is NP-Hard.



NP-Hard problems

- **NP-Hard** problemshave a property similar to ones in **NP-Complete** – they can all be reduced to any problem in NP. Because of that, these are in **NP-Hard** and are at least as hard as any other problem in **NP**.
- A problem can be both in **NP** and **NP-Hard**, which is another aspect of being **NP-Complete** .
- This characteristic has led to a debate about whether or not Traveling Salesman is indeed **NP-Complete**. Since **NP** and **NP-Complete** problems can be verified in polynomial time, proving that an algorithm cannot be verified in polynomial time is also sufficient for placing the algorithm in **NP-Hard**.

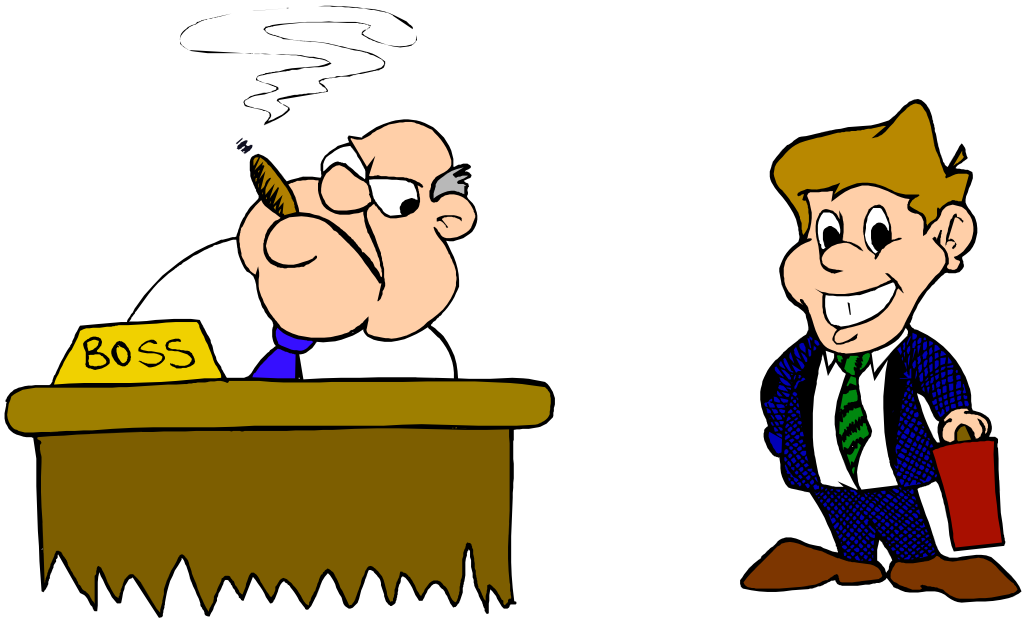
Dealing with Hard Problems

- What to do when we find a problem that looks hard…

I couldn't find a polynomial-time algorithm;
I guess I'm too dumb.

Dealing with Hard Problems

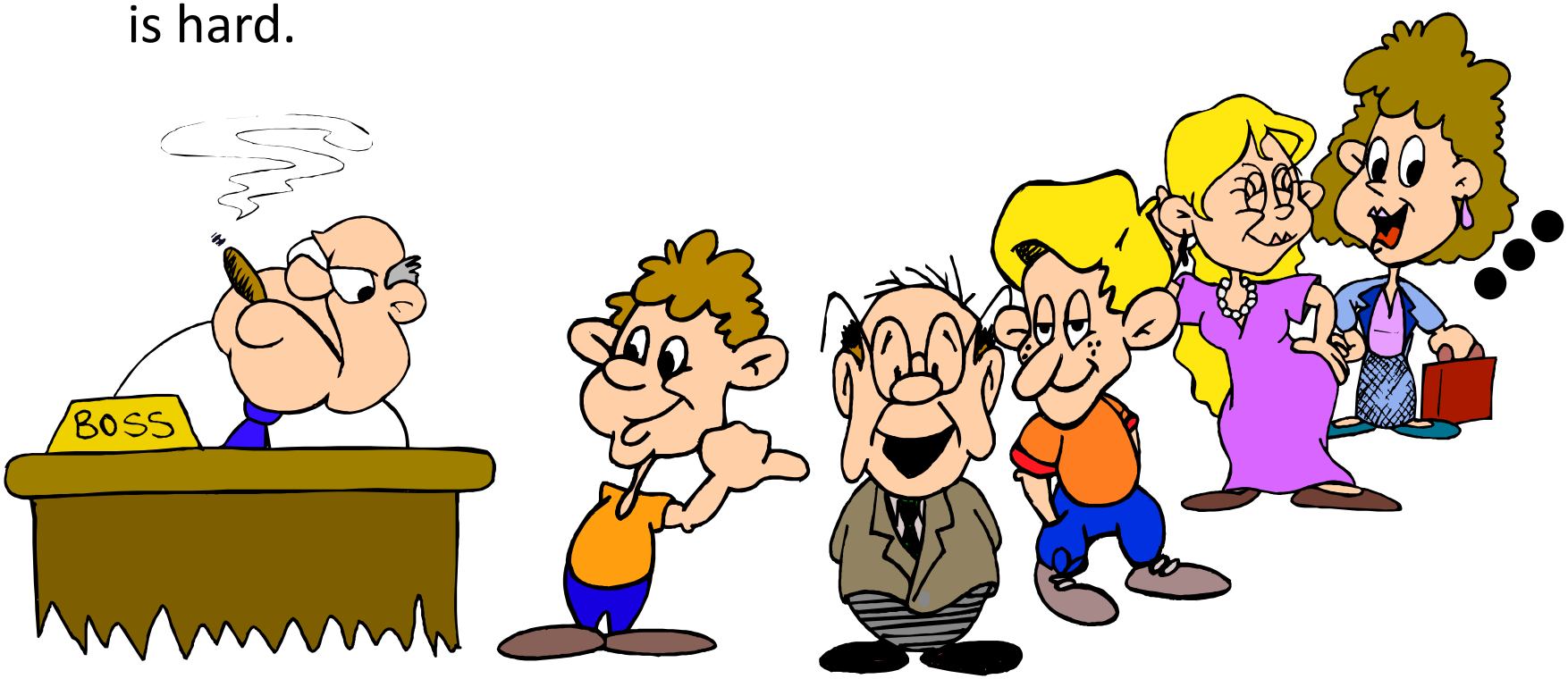
- Sometimes we can prove a strong lower bound… (but not usually)



I couldn't find a polynomial-time algorithm, because no such algorithm exists!

Dealing with Hard Problems

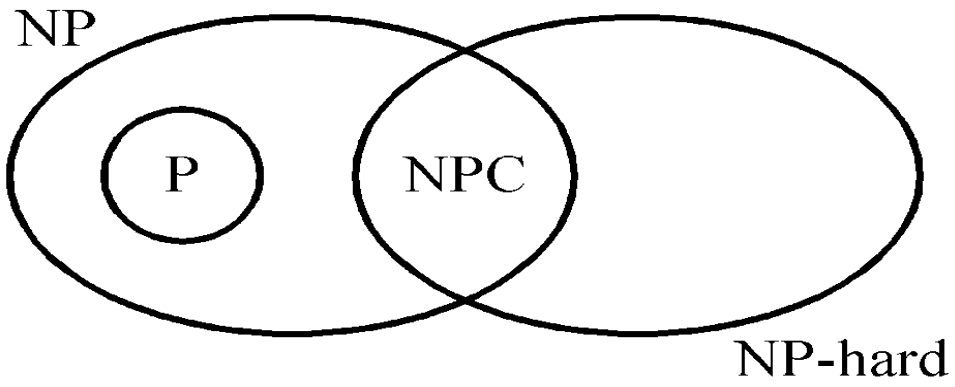
- NP-completeness let's us show collectively that a problem is hard.



I couldn't find a polynomial-time algorithm,
but neither could all these other smart people.

NP-hard and NP-complete problems

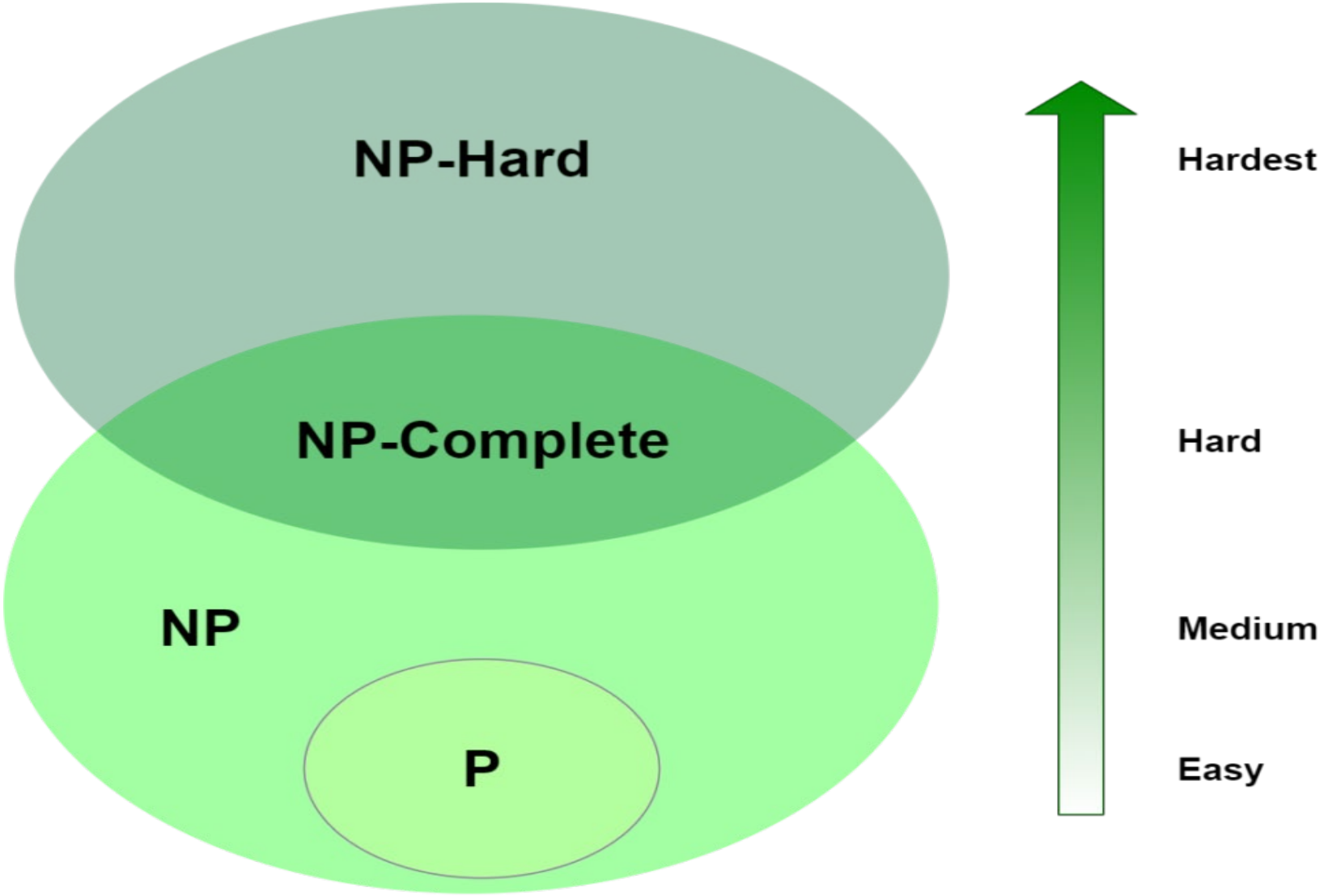
- A Problem that is NP-complete has the property that it can be solved in polynomial time if and only if all **other NP-complete** problem can also be solved in polynomial time.
- If an **NP-hard** problem can be solved in polynomial time , then all NP-complete problems can be known to be solved in polynomial time.
- All NP-complete problems are NP-hard, but some NP-hard problem are not known to be NP-complete?



What is the definition of P, NP, NP-complete and NP-hard?

	P	NP	NP-complete	NP-hard
Solvable in polynomial time	✓			
Solution verifiable in polynomial time	✓	✓	✓	
Reduces any NP problem in polynomial time			✓	✓

P, NP, NP-Complete and NP-Hard Problems in Computer Science



Some interesting observations:

1. NP-Hardness theory focuses on the 'hardness'- how difficult the problem is in relation to other NP problems. It does not comment about the solvability of the problem, whereas NP is defined for solvable problems. Establishing NP-Hardness result does not imply the status of NP.
2. Unsolvable problems such as 'Halting problem' and 'post-correspondence problem' are NP-Hard as there is a reduction from 3-SAT. However, they do not belong to NP.
3. All problems in NP have a trivial deterministic algorithm, typically runs in time exponential in the input size. Therefore, every problem in NP is solvable.

Some interesting observations:

4. A problem is NP-complete if it is NP-Hard and belongs to the set NP (the problem is complete for the complexity class NP). This shows that **NP-complete** problems are the most difficult problems in NP and solving one efficiently will yield efficient solutions to every other problem in NP.
5. Since the set of decision problems are more than the set of optimization problems, the study of NP theory focuses on the set of decision problems. Moreover, the black box technique helps to obtain the solution to optimization problem by incurring a polynomial-time effort.
6. Many interesting problems such as sorting, matrix multiplication, finding the square root of a number are polynomial-time solvable, however, **they are not in Class P** as these problems do not have the corresponding **decision versions.**

Problem & language

Problems and Language

- **An Algorithm A accepts an input string *x* if A outputs *"YES"* on input *x*.**

- **A DECISION PROBLEM can be viewed as a set L of strings (i.e. the strings that should accepted by an algorithm that correctly solves the problem)**

- **The letter L is used to denote a DECISION PROBLEM, because a set of strings is often referred as a LANGUAGE.**

- **An algorithm A accepts language L if A outputs *"YES"* for each x in Land outputs *"NO"* otherwise.**

The Complexity Class (P Polynomial)

- **The complexity class P is the set of all decision problems (or Language) L that can be accepted in worst-case polynomial time.**
- **There is an algorithm A that, if $x \in L$, then on input x, algorithm A outputs *"YES"* in p(n) time, where n is the size of x and p(n) is a polynomial.**
- **When an algorithm A outputs "No", such cases are referred as the complement of a language L.**
- ***Complement of Language*: Consists of all binary strings that are not in L.**

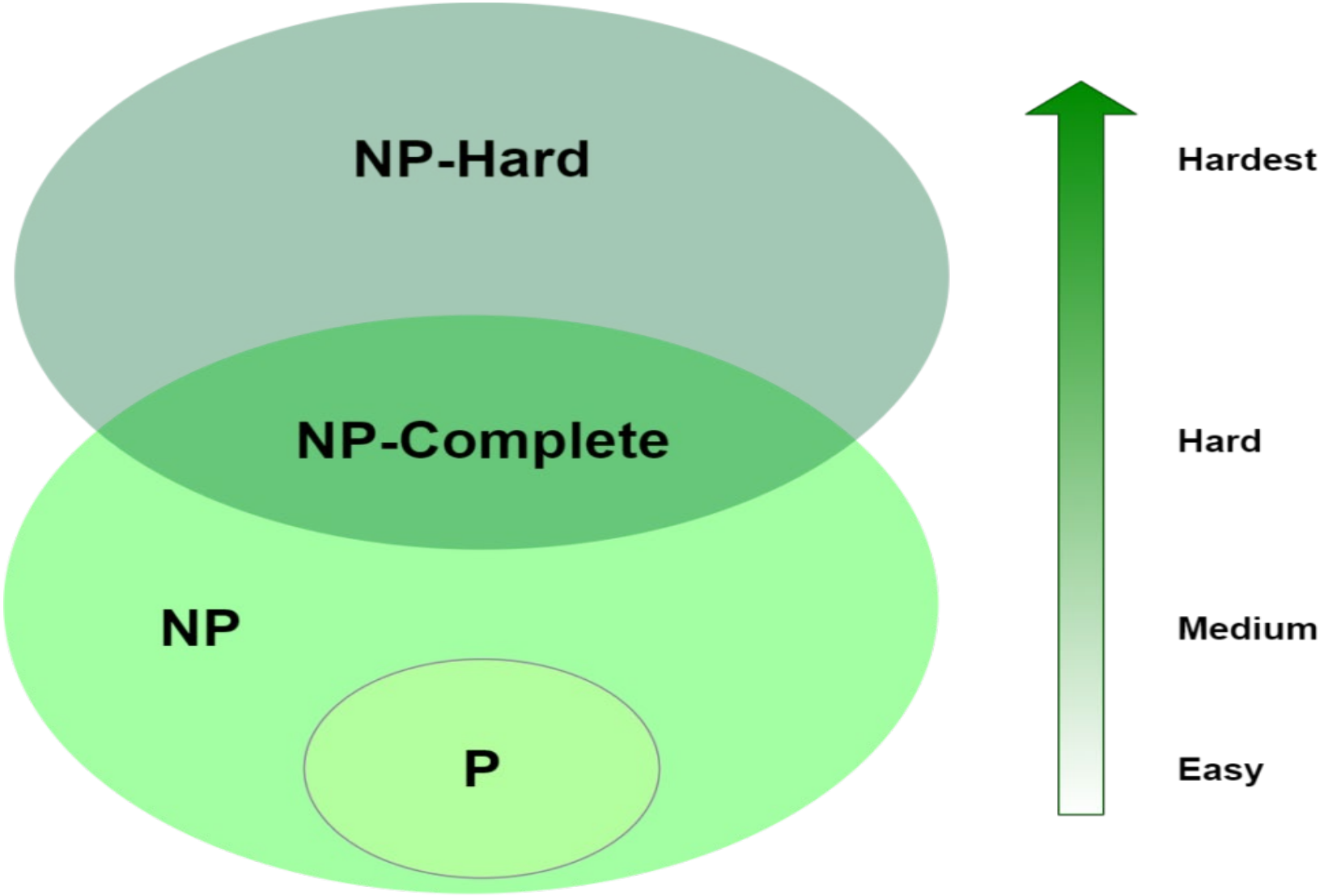
The Complexity Class P

- **Given a input x, we can construct a complement algorithm B, that simply runs A for p(n) steps** *where n is the size of x* **terminating A if it attempts to run more than p(n) steps**

- **If A outputs "yes" then B outputs "no" or if A runs for at least p(n) steps without outputting any thing, then B outputs "yes"**

- **If a language L representing some decision problem, is in P, then the complement of L is also in P.**

P, NP, NP-Complete and NP-Hard Problems in Computer Science



Hard problems, easy problems

Hard problems (**NP**-complete)	Easy problems (in **P**)
3SAT	2SAT, HORN SAT
TRAVELING SALESMAN PROBLEM	MINIMUM SPANNING TREE
LONGEST PATH	SHORTEST PATH
3D MATCHING	BIPARTITE MATCHING
KNAPSACK	UNARY KNAPSACK
INDEPENDENT SET	INDEPENDENT SET on trees
INTEGER LINEAR PROGRAMMING	LINEAR PROGRAMMING
RUDRATA PATH	EULER PATH
BALANCED CUT	MINIMUM CUT

Solving NP complete Problem

Class NP-complete

- Definition: A decision problem *D1* is said to be ***polynomially reducible*** to a decision problem *D2* if there exists a function *t* that transforms instances of *D1* to instances of *D2* such that
  - *t* maps all yes instances of *D1* to yes instances of *D2* and no instances of *D1* to no instances of *D2*;
  - *t* is computable by a polynomial-time algorithm.
- If a problem D1 **polynomially reducible** to some problem D2 that can be solved in polynomial time, then problem D1 can also be solved in polynomial time.

NP-Complete Definition

- Definition: A decision problem *D* is said to be ***NP-complete*** if

  1. It belongs to class *NP*;
  2. Every problem in *NP* is polynomially reducible to *D*.

- **NP**-complete problems are problems which are in **NP** but are also **NP**-hard:
- Definition: A language *L* is ***NP-hard*** if every problem *L'* in **NP** is poly-time mapping reducible to *L*. If in addition, *L* is in **NP**, *L* is said to be ***NP-complete***.

Important Points: NPC

- If we can find a polynomial time algorithm for an NP-Complete problem, then all problems in NP have polynomial time algorithms.
- If we can prove that one problem in NP can't be solved in polynomial time, then all NP-Complete problems are not solvable in polynomial time.
- All NP-Complete problems appear to be difficult.
- We are not aware of any polynomial time algorithms to solve them.
- However, there is no proof that polynomial time algorithms do not exist!

NP-Complete Problems

Here is a list of some **NP**-complete problems:

1. SAT
2. CSAT
3. 3SAT
4. Punch-Card Puzzle
5. Clique
6. Discrete Linear Algebra
7. Traveling Salesperson
8. Hamiltonian Path

See many more from Garey and Johnson's book.[1]

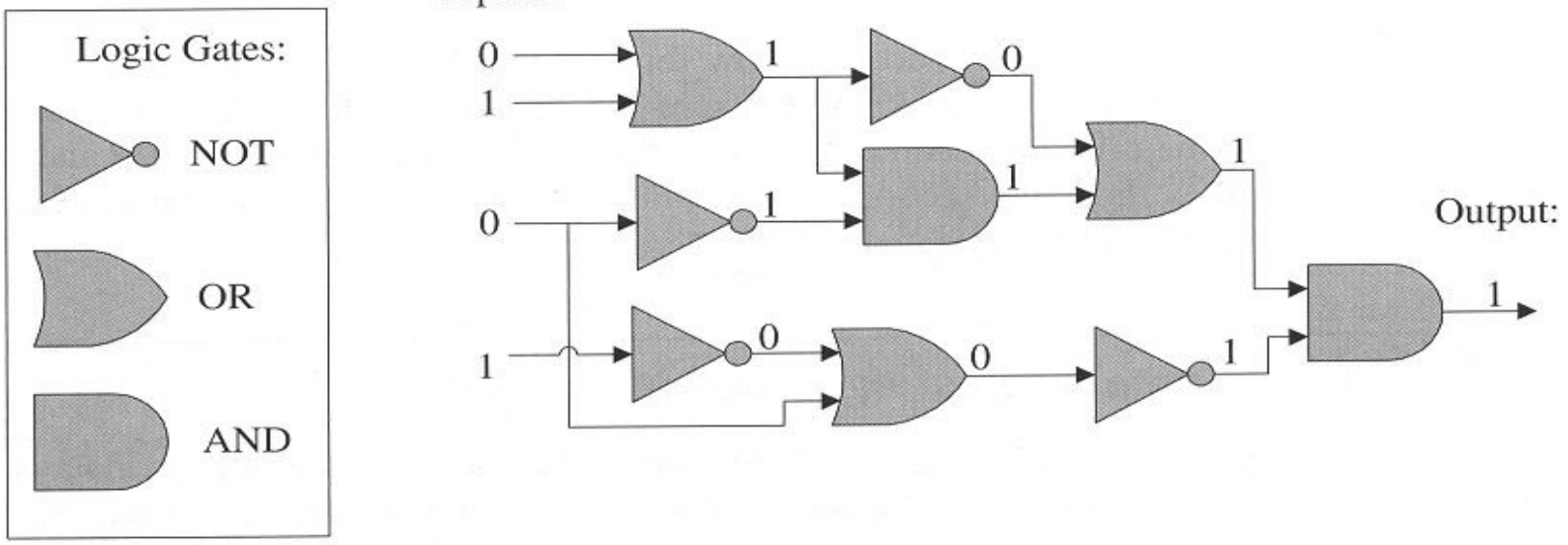
- [1]Computers and Intractability: A Guide to the Theory of NP-Completeness. Michael Garey and David Johnson. W.H. Freeman & Co. **1979**. ISBN: 07167-10455

NP-Complete Problems

- **Hamiltonian circuit** Determine whether a given graph as a Hamiltonian circuit (a path that starts and ends at the same vertex and passes through all the other vertices exactly once).
- **Traveling salesman** Find the shortest tour through n cities with known positive integer distances between them.
- **Graph coloring** For a given graph, find its chromatic number (the smallest number of colors that need to be assigned to the graph's vertices so that no two adjacent vertices are assigned the same color).

Boolean Circuit

- A **Boolean circuit** is a directed graph where each node, called a logic gate corresponds to a simple Boolean function *AND*, *OR*, or *NOT*
- The incoming edges for a logic gate correspond to inputs for its Boolean function and the outgoing edges correspond to the outputs



The First NP-Complete Problem

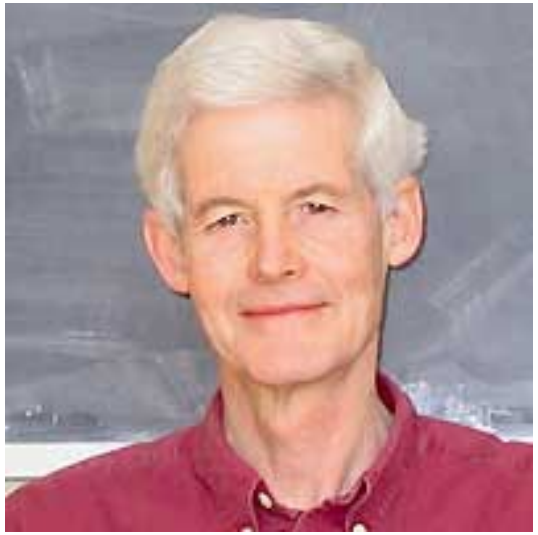
- Are there any NP-Complete problems? Yes.
- The first one identified was the "**Satisfiability** problem or **Circuit-Sat** ".
  - **Circuit-Sat** is the problem that takes an input a Boolean circuit with a single output node, and asks whether there is an assignment of values to the circuit's inputs so that its output value is *1*
  - Example: (a or b) and ((not b) or c)
- If a **non-deterministic algorithm** chooses an assignment of input bits (represented by a sequence of non-deterministic bits) and then it checks deterministically whether this input generates output *1*

Cook's theorem

In 1971 Cook published "The Complexity of Theorem Proving Procedures," a seminal paper that laid the foundations for the theory of NP-complete

NP = P iff the satisfiability problem is a P problem.

- SAT is NP-complete.
- It is the first NP-complete problem.
- Every NP problem reduces to SAT.



Stephen Arthur Cook

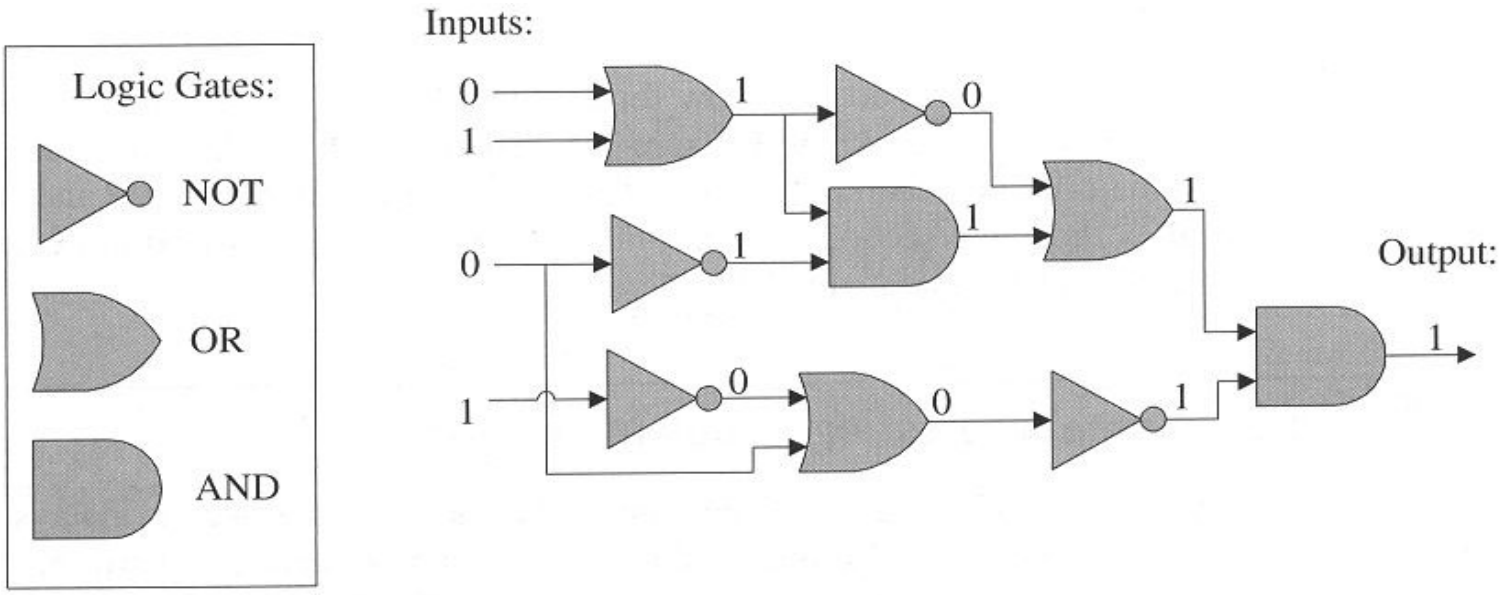
(1939~)

First NP-complete problem—Circuit Satisfiability (problem definition)

- Boolean combinational circuit
  - Boolean combinational elements, wired together
  - Each element, inputs and outputs (binary)
  - Limit the number of outputs to 1.
  - Called *logic gates*: NOT gate, AND gate, OR gate.
  - *true table*: giving the outputs for each setting of inputs
  - true assignment: a set of boolean inputs.
  - satisfying assignment: a true assignment causing the output to be 1.
  - A circuit is satisfiable if it has a satisfying assignment.

Satisfiability

- It appears difficult because there are 2^N possible solutions if there are *N* Boolean variables. This is exponential.
- But a possible answer is easy to check in polynomial time, so satisfiability is NP.
- It can be shown that every problem in NP can be reduced to satisfiability.



Problem 1: Satisfiability

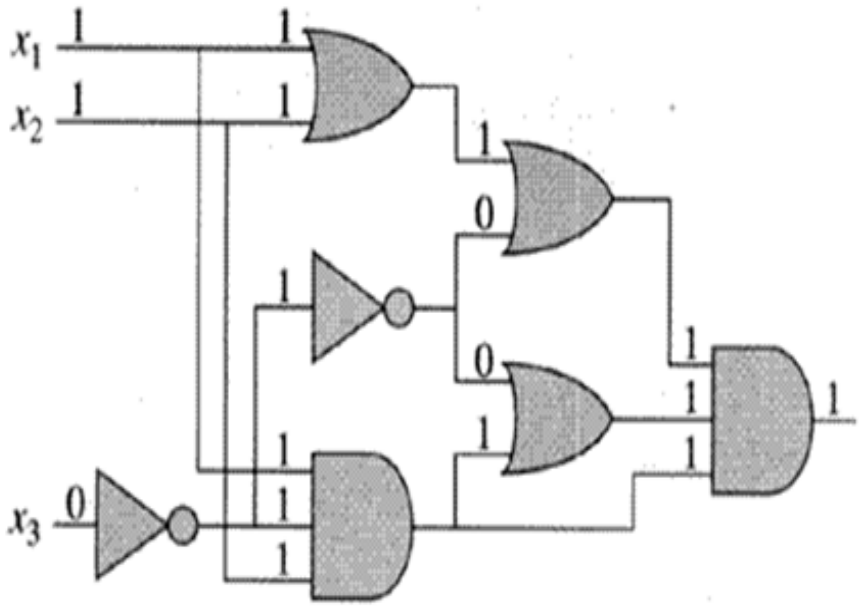
- Determine whether any given logical formula is satisfiable, that is, whether there is a way of assigning the values TRUE and FALSE to the variables so that the result of the formula is **TRUE**.

Example

- Consider Boolean variables A,B, and C and their complements A', B', and C'.
- The following formula is **satisfiable**:
  - A AND (B OR C) AND (C' OR A')
- The following formula is **not satisfiable**:
  - A AND (B OR C) AND (C' OR A') AND (B')

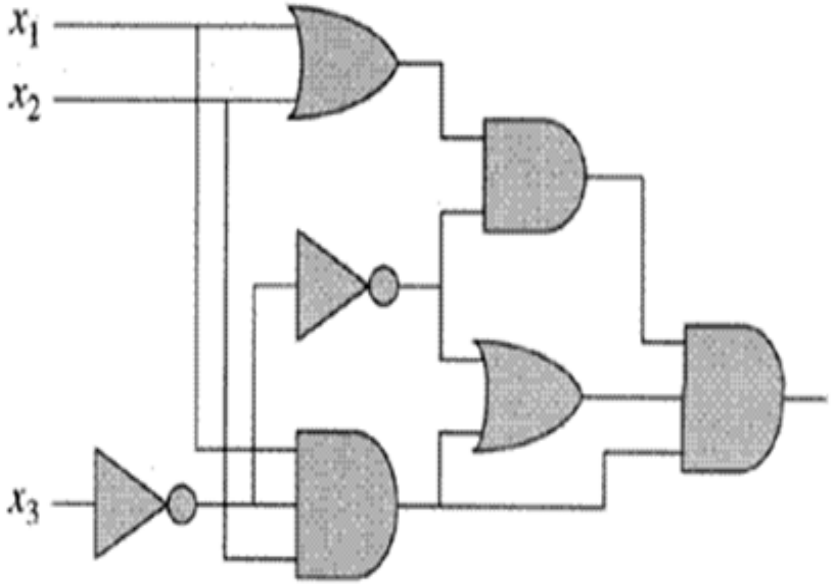
Two instances of circuit satisfiability problems

satisfiable



(a)

unsatisfiable



(b)

Two instances of the circuit-satisfiability problem. **(a)** The assignment $\langle x_1 = 1, x_2 = 1, x_3 = 0 \rangle$ to the inputs of this circuit causes the output of the circuit to be 1. The circuit is therefore satisfiable. **(b)** No assignment to the inputs of this circuit can cause the output of the circuit to be 1. The circuit is therefore unsatisfiable.

Circuit Satisfiability Problem: definition

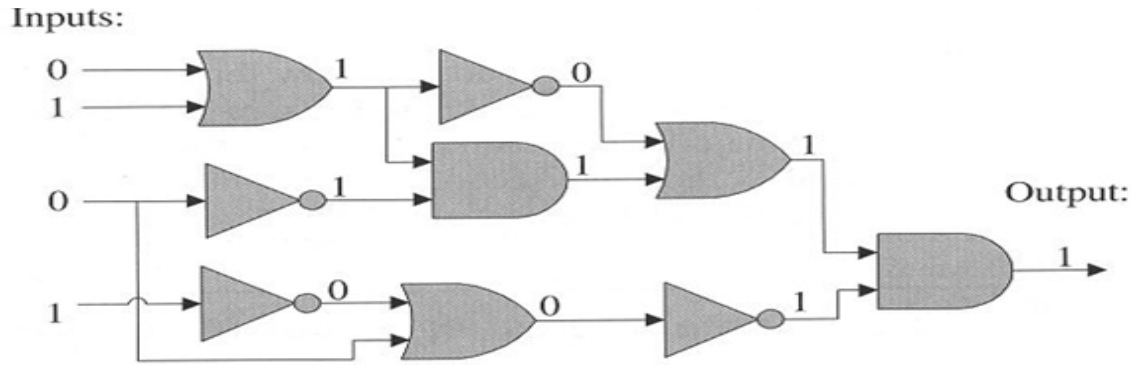
- Circuit satisfying problem: given a boolean combinational circuit composed of AND, OR, and NOT, is it stisfiable?
- CIRCUIT-SAT={<C>: C is a satisfiable boolean circuit}
- Implication: in the area of computer-aided hardware optimization, if a sub-circuit always produces 0, then the subcircuit can be replaced by a simpler sub-circuit that omits all gates and just output a 0.

Satisfiability or SAT

SAT looks like:

$$(x \vee y \vee z)\,(x \vee \overline{y})\,(y \vee \overline{z})\,(z \vee \overline{x})\,(\overline{x} \vee \overline{y} \vee \overline{z}).$$

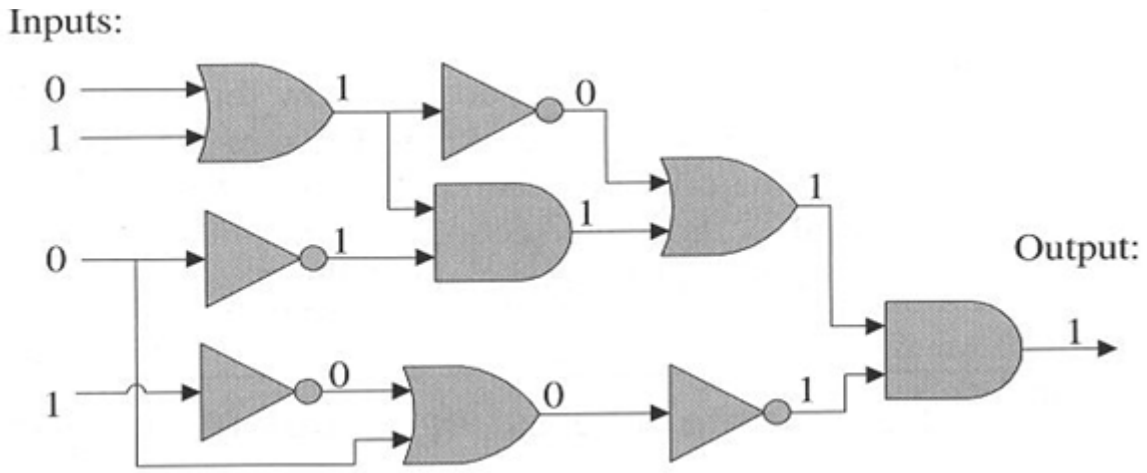
This is a *Boolean formula in conjunctive normal form (CNF)*. It is a collection of *clauses* (the parentheses), each consisting of the disjunction (logical *or*, denoted $\vee$) of several *literals*, where a literal is either a Boolean variable (such as x) or the negation of one (such as $\overline{x}$).



A *satisfying truth assignment* is an assignment of `false` or `true` to each variable so that every clause contains a literal whose value is `true`. The SAT problem is the following: given a Boolean formula in conjunctive normal form, either find a satisfying truth assignment or else report that none exists.

Satisfiability or SAT

- SAT is a typical *search problem*.
- Given an *instance* I (that is, some input data specifying the problem at hand, in this case a Boolean formula in conjunctive normal form), and asked to find a *solution* S (an object that meets a particular specification, in this case an assignment that satisfies each clause).



- A search problem must have the property that any proposed solution S to an instance I can be *quickly checked* for correctness.
- S must at least be concise (quick to read), with length polynomically bounded by that of I.

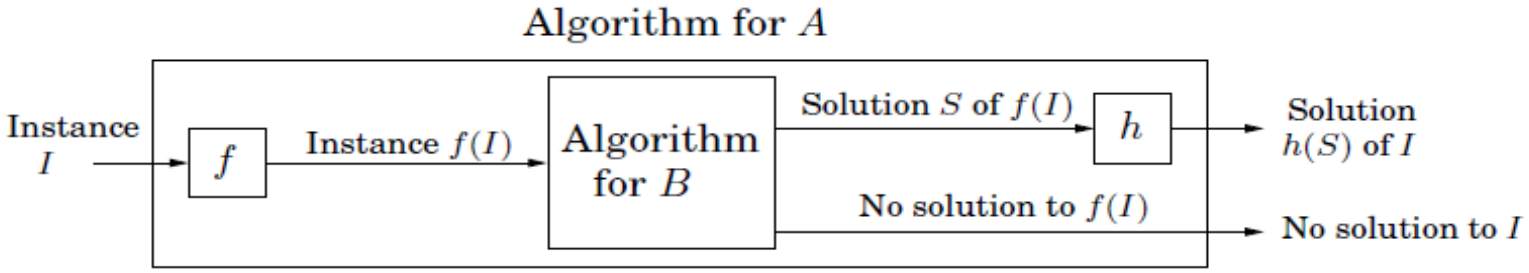
Satisfiability or SAT

- This is clearly true in the case of SAT, for which S is an assignment to the variables.
- To formalize the notion of quick checking, we will say that there is a polynomial-time algorithm that takes as input I and S and decides whether or not S is a solution of I.
- For SAT, this is easy as it just involves checking whether the assignment specified by S indeed satisfies every clause in I.

A *search problem* is specified by an algorithm C that takes two inputs, an instance I and a proposed solution S, and runs in time polynomial in $|I|$. We say S is a solution to I if and only if $C(I, S) = \texttt{true}$.

A search problem is NP-complete

- A *reduction* from search problem A to search problem B is a polynomial-time algorithm f that transforms any instance I of A into an instance f(I) of B, together with another polynomial-time algorithm h that maps any solution S of f(I) back into a solution h(S) of I;



- If f(I) has no solution, then neither does I. These two translation procedures f and h imply that any algorithm for B can be converted into an algorithm for A by bracketing it between f and h.

The class of the hardest search problems

- *A search problem is* **NP**-*complete if all other search problems reduce to it.*
- For a problem to be **NP**-complete, it must be useful in solving every search problem in the world! It is remarkable that such problems exist.

Reducibility

Reducibility

- We say that problem Q “reduces” to problem R if we can transform Q to R in polynomial time (and still produce the same answers).
- Thus if we have a polynomial time algorithm for R we also have one for Q.
- A reduction is **a way of converting one problem to another problem**, so that the solution to the second problem can be used to solve the first problem.
- Finding the area of a rectangle, reduces to measuring its width and height.
- Solving a set of linear equations, reduces to inverting a matrix.

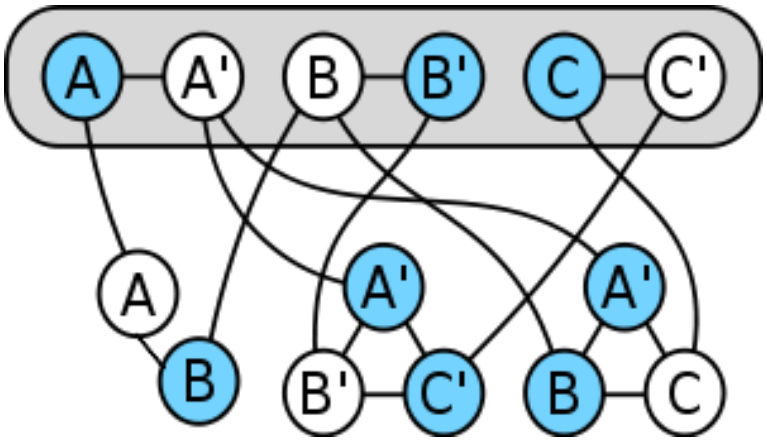
Reducibility

- In computability theory and computational complexity theory, a reduction is an **algorithm for transforming one** problem into another problem.
- A sufficiently efficient reduction from one problem to another may be used to show that the second problem is at least as difficult as the first.
- Intuitively, problem ***A*** is **reducible** to problem ***B***, if an algorithm for solving problem ***B*** efficiently (if it existed) could also be used as a subroutine to solve problem ***A*** efficiently.

Reducibility

- When this is true, solving *A* cannot be harder than solving *B*. "Harder" means having a higher estimate of the required computational resources in a given context (e.g., higher time complexity, greater memory requirement, expensive need for extra hardware processor cores for a parallel solution compared to a single-threaded solution, etc.).
- The existence of a reduction from ***A*** to ***B***, can be written in the shorthand notation $A \leq_m B$, usually with a subscript on the ≤ to indicate the type of reduction being used (**m** : mapping reduction, **p** : polynomial reduction).
- The mathematical structure generated on a set of problems by the reductions of a particular type generally forms a preorder, whose equivalence classes may be used to define degrees of unsolvability and complexity classes.

Example of a reduction



- Example of a reduction from the boolean satisfiability problem $(A \vee B) \wedge (\neg A \vee \neg B \vee \neg C) \wedge (\neg A \vee B \vee C)$ to a vertex cover problem.
- The blue vertices form a minimum vertex cover, and the blue vertices in the gray oval correspond to a satisfying truth assignment for the original formula.

Polynomial-Time Reducibility

- We say that a language *L*, defining some decision problem, is **polynomial-time reducible** to a language *M*, if
  - there is a function *f* computable in polynomial time, that
  - takes an input *x* to *L*, and transforms it to an input *f(x)* of *M*, such that,
  - $x \in L$ if and only if $f(x) \in M$
- We use notation $L \xrightarrow{poly} M$ to signify that language *L* is polynomial-time reducible to language *M*

NP-hardness

- We say that a language M, defining some decision problem, is **NP-hard** if every other language L in NP is *polynomial-time reducible* to M, i.e.,
- M is **NP-hard**, if for every L $\in$ NP, $L \xrightarrow{poly} M$
- If a language M is NP-hard and it belongs to NP itself, then M is **NP-complete**
- NP-complete problem is, in a very formal sense, one of the **hardest problems** in NP, as far as polynomial-time reducibility is concerned

How do we prove a problem is NP complete

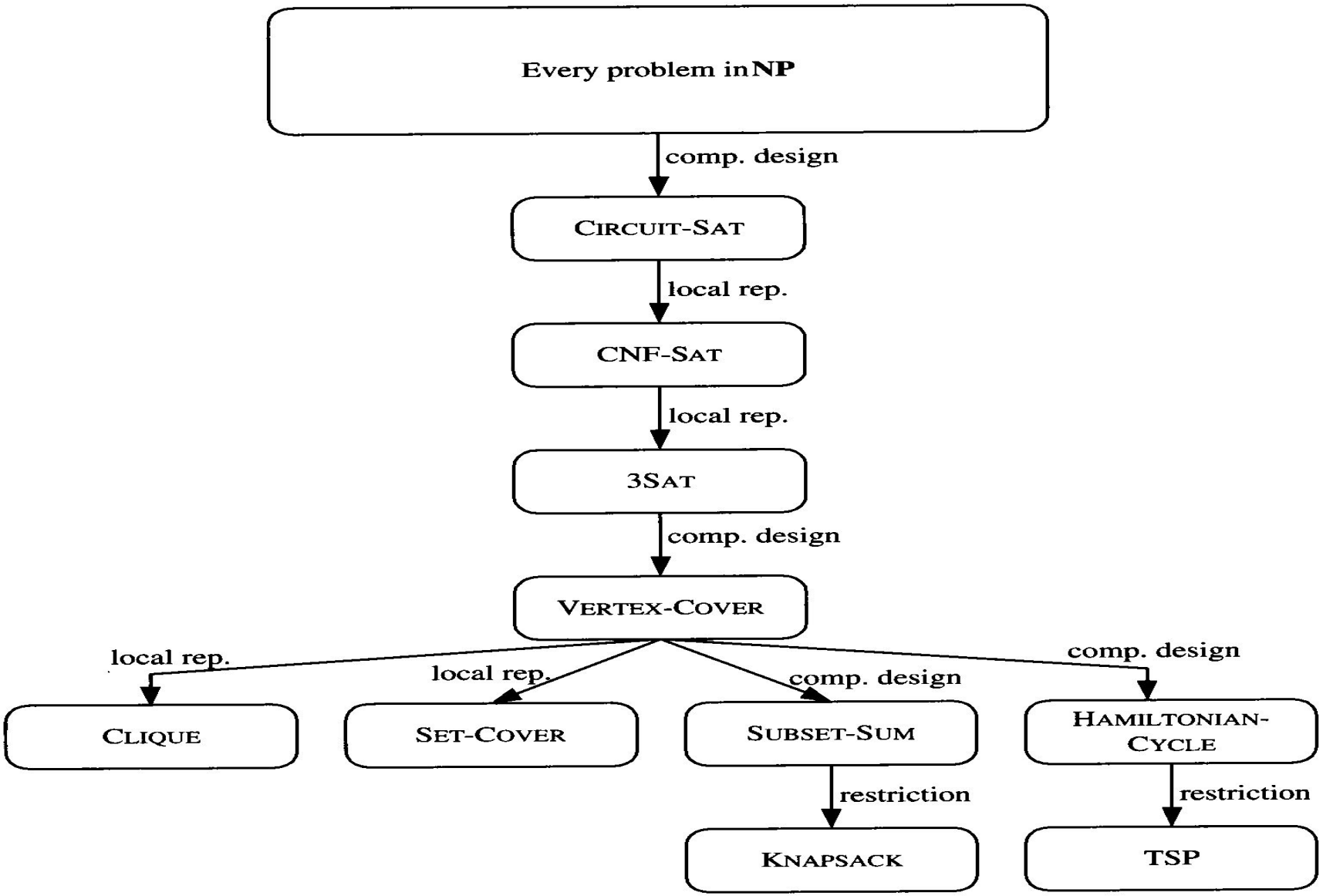
- Given a problem U, the steps involved in proving that it is **NP-complete** are the following:
- Step 1: Show that U is in NP.
- Step 2: Select a **known NP-complete** problem V.
- Step 3: Construct a **reduction** from **V** to **U**.
- Step 4: Show that the reduction requires **polynomial time.**

NP-Complete Problems

- To show other problems R are NP-Complete you merely have to show that SAT **reduces** to R.
  - Examples are the Traveling Salesman Problem, the Hamiltonian Circuit Problem, Bin Packing, the Knapsack Problem, Graph Coloring, Clique, etc.

- If we can find a polynomial time algorithm for any of these problems, we can find polynomial time algorithms for all!

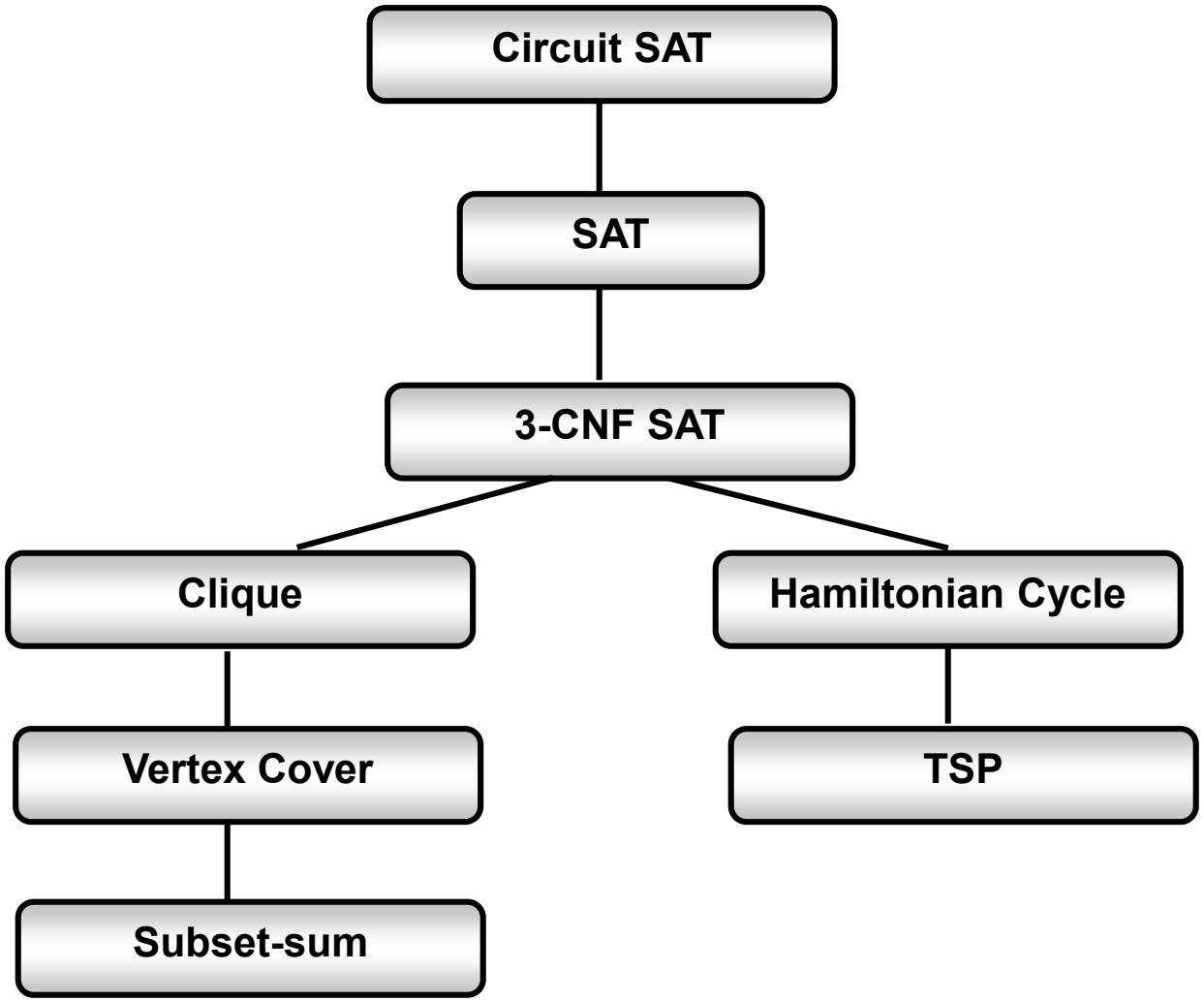
Reductions between search problems.



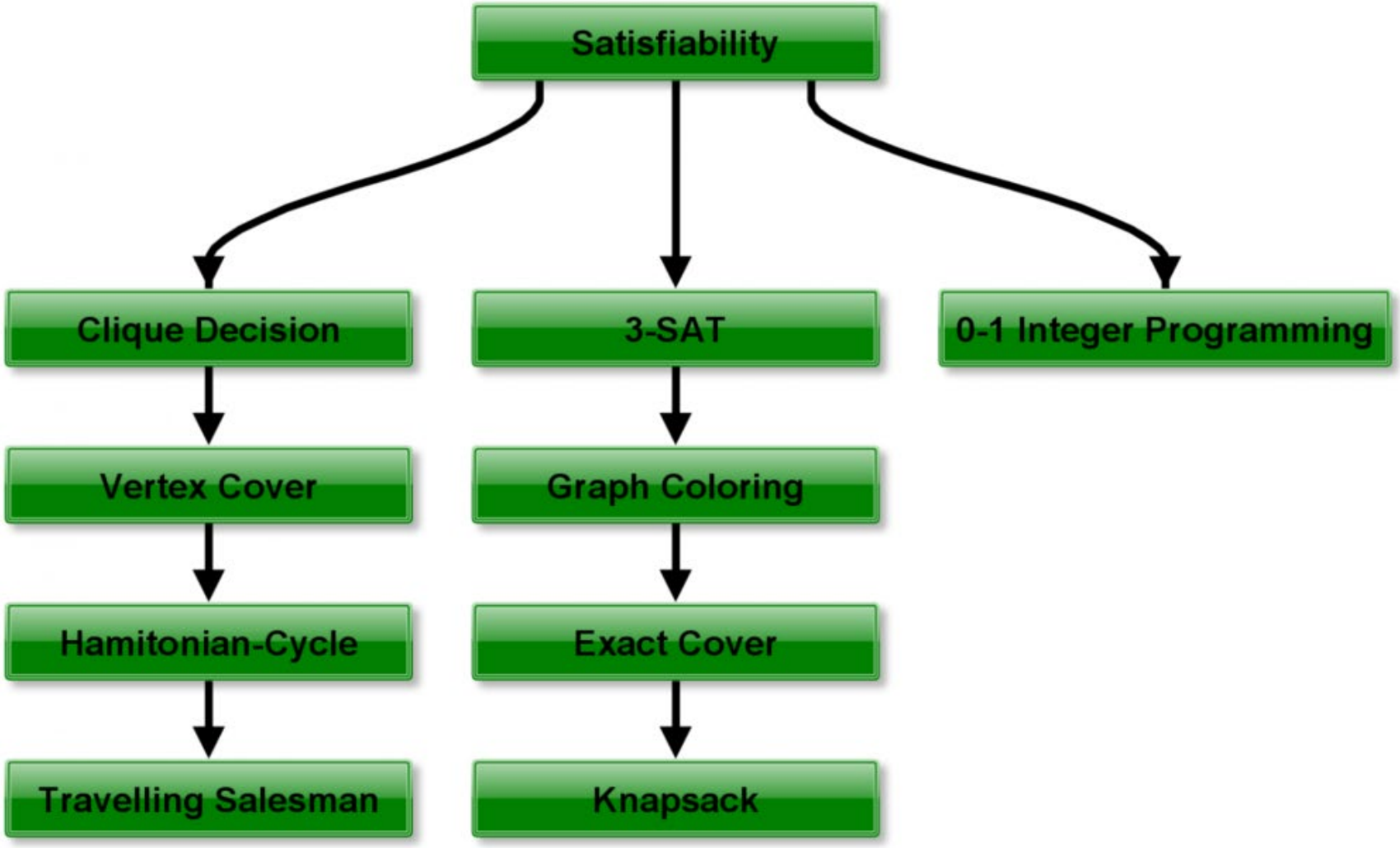
Types of reduction

- Let M be known NP-complete problem. The types of reductions are:
  - **By restriction:** noting that known NP-complete problem is a special case of our problem L
  - **Local replacement:** dividing instances of M and L into basic units, and then showing how each basic unit of M can be locally converted into a basic unit of L
  - **Component design:** building components for an instance of L that will enforce important structural functions for instances of M

Important NP-complete Problems



NP Completeness



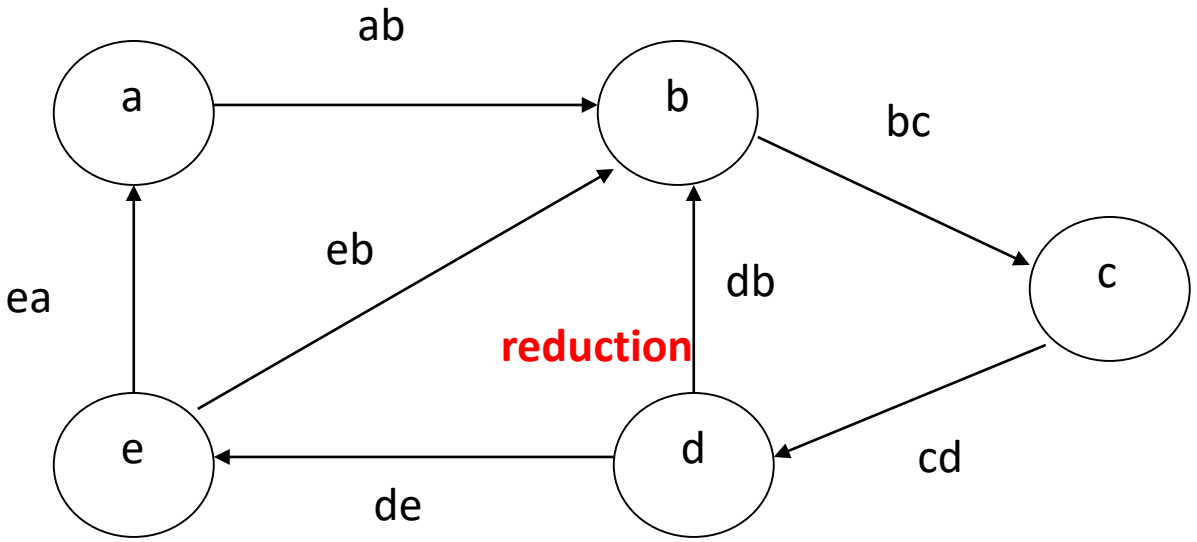
How do we prove a problem is NP complete

A new problem can be proven NP-complete by reduction from a problem already known to be NP-complete.

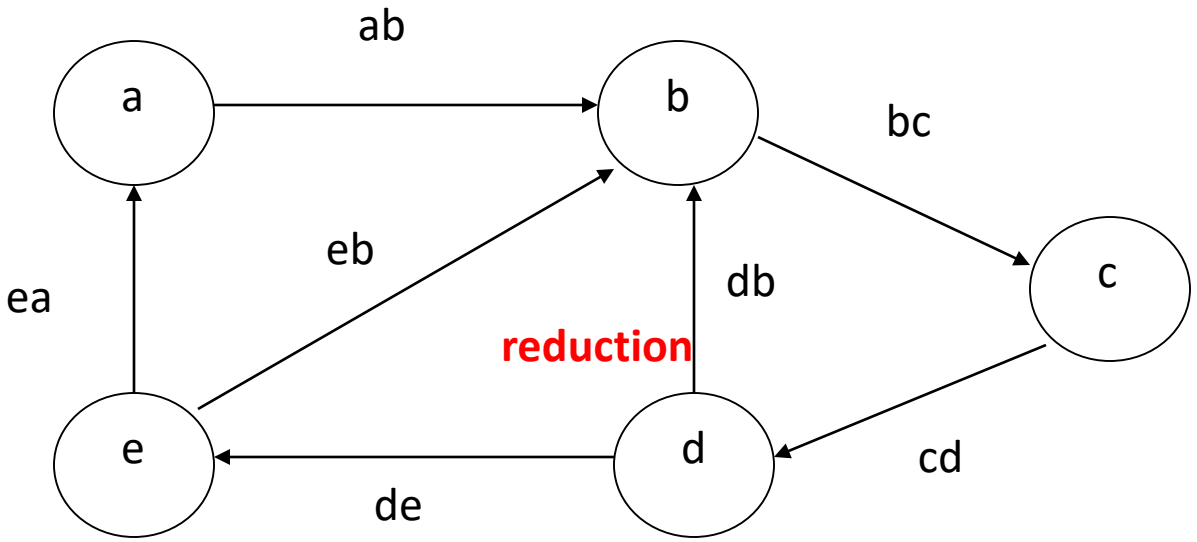
Hamiltonian circuit is NP-complete

- Determine whether a given graph as a Hamiltonian circuit (a path that starts and ends at the same vertex and passes through all the other vertices exactly once).

- To prove Hamiltonian circuit is NP-complete, we transform Hamiltonian circuit to Satisfiability



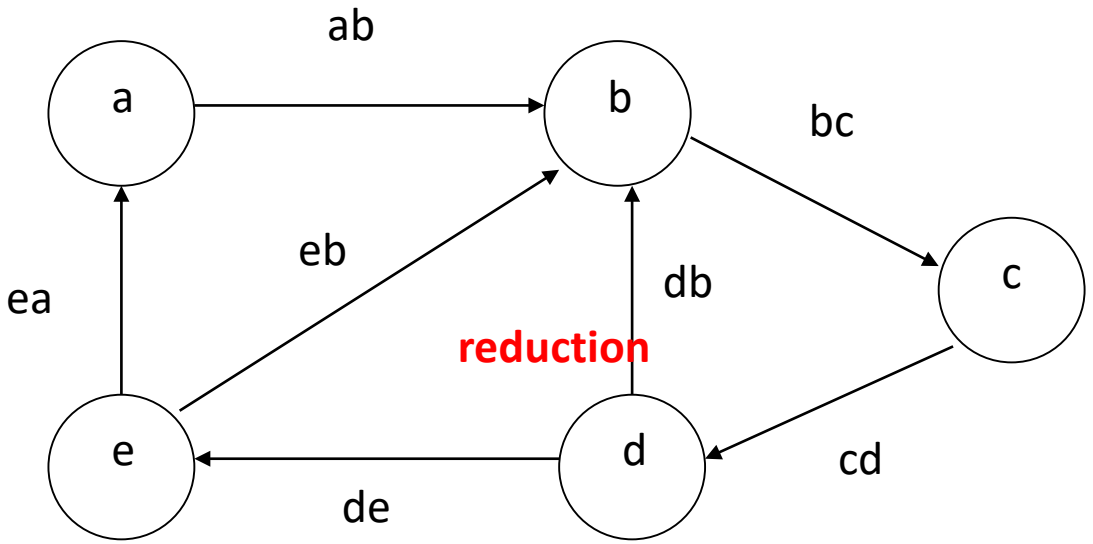
Example: Hamiltonian Circuit



Note that in a Hamiltonian circuit that one input edge and one output edge of every node is used.

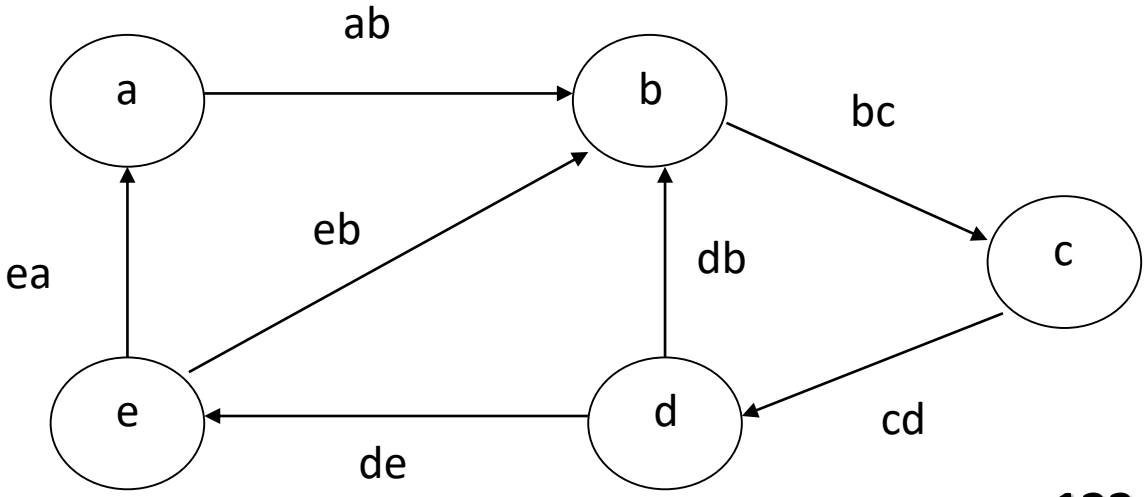
Transform to Satisfiability

- Each edge can be true or false – true if the edge is in the circuit and false if not.
- Since only one input edge and only one output edge for a node can be in the circuit we can write down equivalent Boolean expressions.
- For example, for node a: (ea and ab)



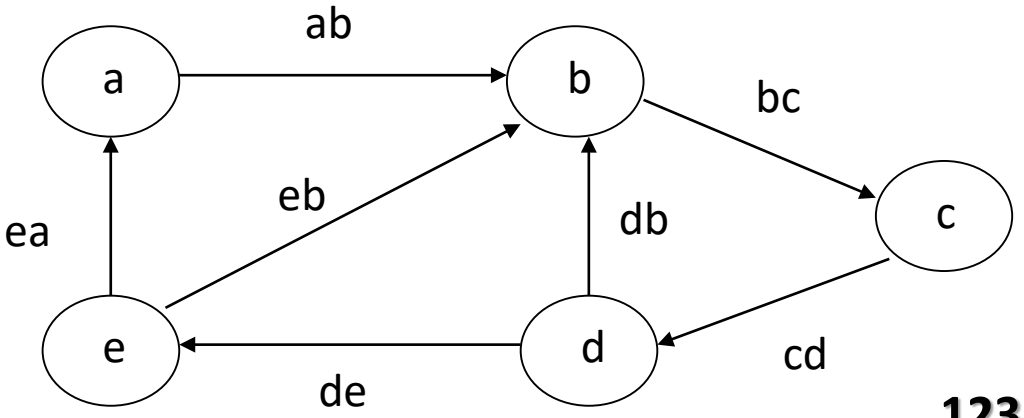
Transform to Satisfiability

- Each edge can be true or false – true if the edge is in the circuit and false if not.
- Since only one input edge and only one output edge for a node can be in the circuit we can write down equivalent Boolean expressions.
- For example, for node **a: (ea and ab)**



Transform to Satisfiability

- **Node b**: [(ab and (not db) and (not eb)) or ((not ab) and db and (not eb)) or ((not ab) and (not db) and eb)] and bc
- **Node c**: (bc and cd)
- **Node d**: cd and [(db and (not de)) or ((not db) and de)]
- **Node e**: de and [(ea and (not eb)) or ((not ea) and eb)]



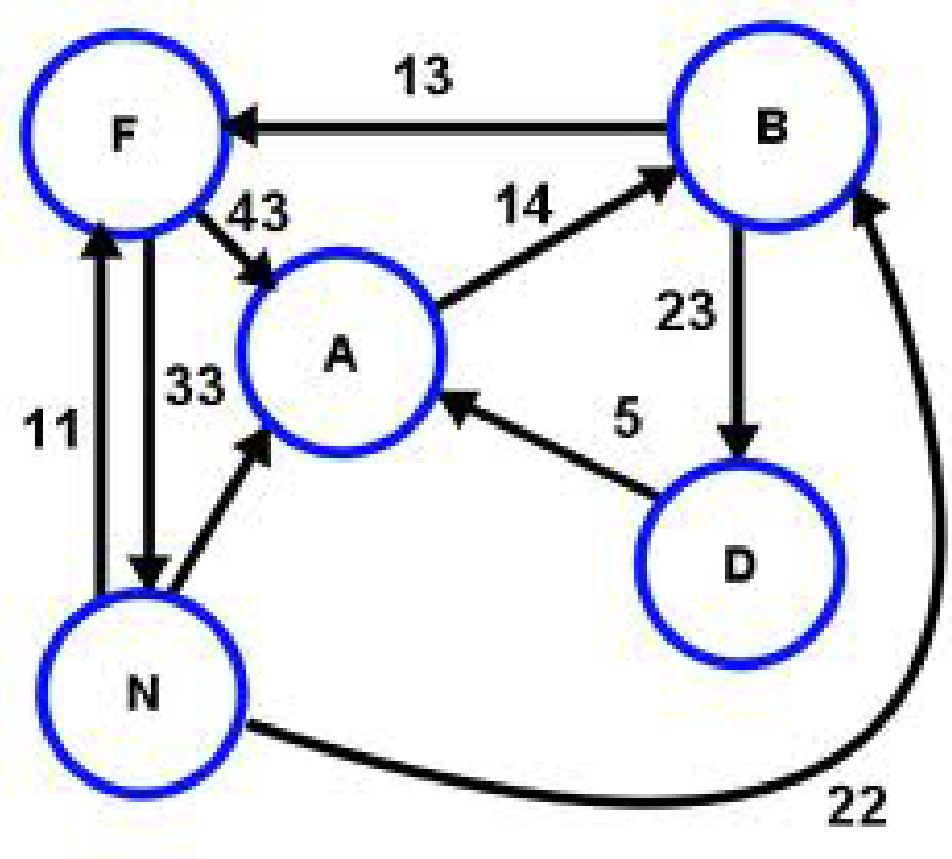
Transform to Satisfiability

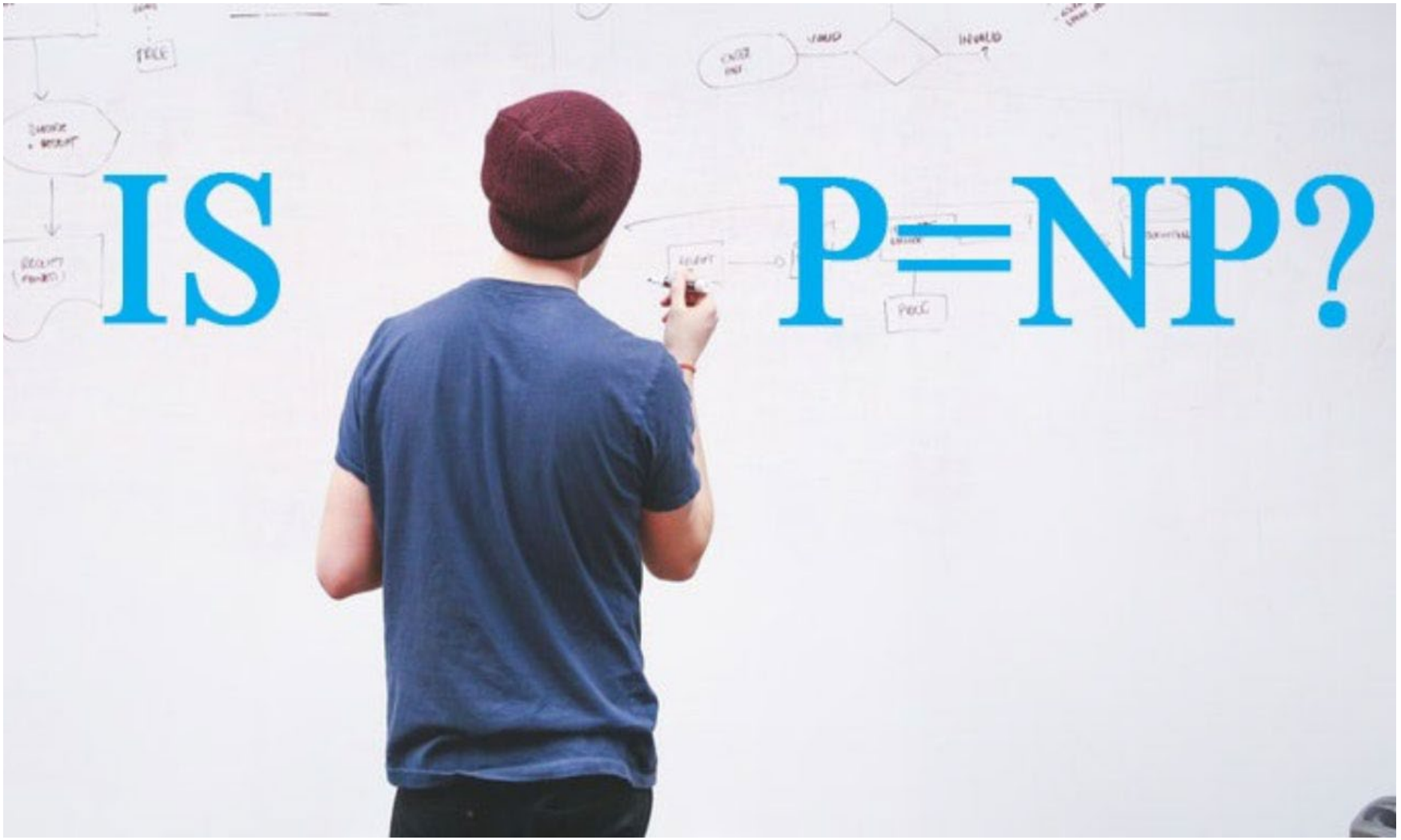
- Node b: [(ab and (not db) and (not eb)) or ((not ab) and db and (not eb)) or ((not ab) and (not db) and eb)] and bc
- Node c: (bc and cd)
- Node d: cd and [(db and (not de)) or ((not db) and de)]
- Node e: de and [(ea and (not eb)) or ((not ea) and eb)]

Transform to Satisfiability

- If you can find an assignment to the Boolean variables (the edges) such that all those Boolean expressions are true, you have found a Hamiltonian Circuit. [Note, this transformation isn't quite correct.]
- So, if we can solve a SAT problem in polynomial time, we can solve a HC problem in polynomial time, because the transformation takes polynomial time.

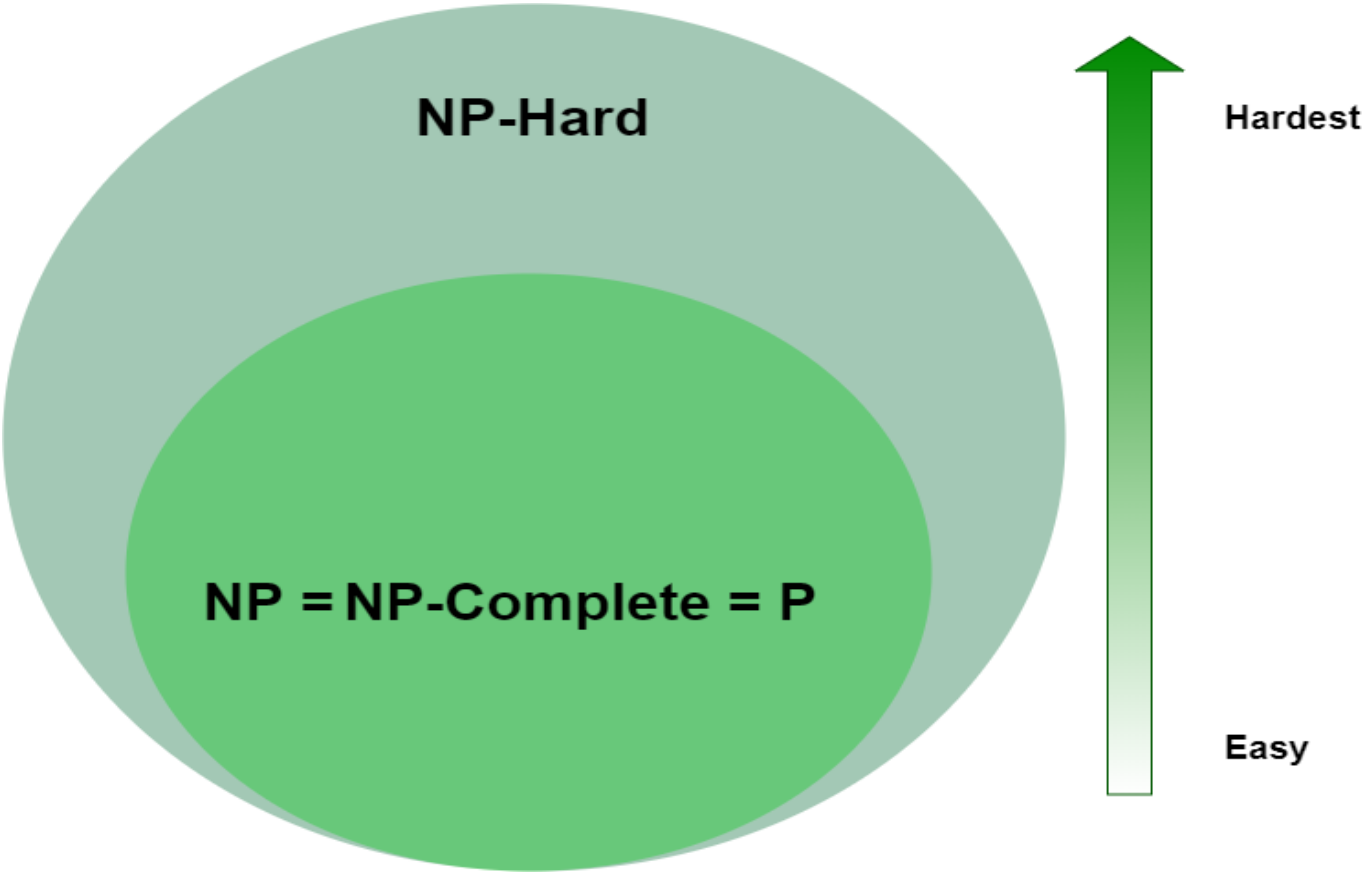
Proving NP-Complete




IS
P=NP?

Does P=NP?

- A question that's fascinated many computer scientists is whether or not all algorithms in **NP** belong to **P** :



Does P=NP?

- It's an interesting problem because it would mean, for one, that any **NP** or **NP-complete** problem can be solved in polynomial time.
- So far, proving that **P!=NP** as proven elusive. Because of the intrigue of this problem, it's one of the Millennium Prize Problems for which there is a $1,000,000 prize.
- For our definitions, we assumed that **P!=NP** , however, **P=NP** may be possible. If it were so, aside from **NP** or **NP-complete** problems being solvable in polynomial time, certain algorithms in **NP-hard** would also dramatically simplify. For example, if their verifier is **NP** or **NP-complete** , then it follows that they must also be solvable in polynomial time, moving them into **P = NP = NP-complete** as well.
- We can conclude that **P = NP** means a radical change in computer science and even in the real-world scenarios.
- Currently, some security algorithms have the basis of being a requirement of too long calculation time. Many encryption schemes and algorithms in cryptography are based on the number factorization which the best-known algorithm with exponential complexity. If we find a polynomial-time algorithm, these algorithms become vulnerable to attacks.

Summary

- The “hardest" problems in NP are the NP-complete
- problems: if you prove that one NP-complete problem can be solved by a polynomial-time algorithm, then it follows that all the problems in NP can be solved by a polynomial-time algorithm.
- Conversely, if you show that one particular problem in NP is intractable, then all NP-complete problems would be intractable

Summary

P problems are quick to solve

NP problems are quick to verify but slow to solve

NP-complete problems are also quick to verify, slow to solve and can be reduced to any other NP-problem

NP-Hard problems are slow to verify, slow to solve and can be reduced to any other NP-problem

As a final note, if has proof in the future, humankind has to construct a new way of security aspects of the computer era. When this happens, there has to be another complexity level to identify new hardness levels than we have currently.

Thanks for Your Attention!

References

- https://www.baeldung.com/cs/p-np-np-complete-np-hard
- https://en.wikipedia.org/wiki/Reduction_(complexity)

A Useful List of NP-Complete Problems

GRAPHS

- **Vertex Cover Decision Problem(VC)**: Given a graph G=(V,E) and a positive integer k, is there a subset V' of V of vertices which form a Vertex Cover for G with the size of V' no more than k. . (A Vertex Cover for G is a set of vertices with the property that every edge has at least one vertex from that set as an endpoint.)

- **Chromatic Number Decision Problem(CNDP):** Given a graph G=(V,E) and a positive integer m is it possible to assign one of the numbers 1, 2, ..., m to vertices of G so that for no edge in E is it true that the vertices on that edge have been assigned the same number. (This is called the Chromatic Number Decision Problem because if you think of 1, 2, ..., m as representing distinct colors, then we are saying that we can “color” the vertices so no two vertices of the same color are connected by an edge. One of the most famous recent theorems in Mathematics was that for a graph that can be drawn on a plane with no overlapping edges four colors are sufficient to color that graph. This is usually called the four-color map theorem because one can think of a map of countries as being represented by a graph where countries are vertices and edges are drawn between any two countries which touch each other. Then the problem becomes whether you can color that map with four colors. For a planar graph, 4 colors suffice. This was proven with a very large use of computers.)

Graphs

- **Clique Decision Problem (CDP):** Given graph G=(V,E) and a positive integer k, is there a subset V' of V such that V' contains at least k vertices and V' forms a clique. (A clique is a complete subgraph – that is, if you take that set of vertices V' then every pair of vertices in V' are connected by an edge in E. Thus, V' and those edges form a complete subgraph.)
- **Independent Set Decision Problem:** Given a graph G=(V,E) and a positive integer k, is there a subset V' of V of size at least k such that no two vertices in V' are connected by an edge in E. (Such a set of vertices is called an independent set for the graph.)
- **Hamiltonian Circuit (HCDP):** Given a graph G=(V,E), does G contain a Hamiltonian Circuit. (A Hamiltonian Circuit is a Path starting in one vertex and ending in that vertex - hence a Circuit - that visits every intermediate vertex exactly once and visits every vertex in V.)
- **Directed Hamiltonian Circuit (DHCDP):** Does directed graph G=(V,E) contain a Hamiltonian Circuit?
- **Hamiltonian Path Decision Problem (HPDP):** Does graph G=(V,E) contain a Hamiltonian Path. (A Hamiltonian Path is a path that starts in one vertex, ends in another, and visits every other vertex in V exactly once.)

Graphs

- **Directed Hamiltonian Path Decision Problem (HPDP):** Does directed graph G=(V,E) contain a Hamiltonian Path. (A Hamiltonian Path is a path that starts in one vertex, ends in another, and visits every other vertex in V exactly once.)
- **Traveling Salesman Decision Problem (TSDP):** Given a complete weighted graph G=(V,E) and a positive integer k, does graph G contain a Hamiltonian Circuit with total weight at most k.
- **Euclidean Traveling Salesman Decision Problem (ETSDP):** Given a complete weighted graph G=(V,E) which satisfies the triangle inequality and a positive integer k, is there a Hamiltonian Circuit for G with total weight less than or equal to k. (The triangle inequality says that for any three vertices v, w, and z the weight on <v,z> is less than or equal to the sum of the weights on <v,w> and <w,g>. This of course corresponds to the usual way things work in flat Euclidean Geometry.)

Graphs

- **Graph Isomomorphism Decision Problem (GIDP):** Given two graphs G=(V,E) and G'=(V',E'), is there a function f from V to V' such that f is one-to-one and onto and such that <v,w> is an edge in G if and only if <f(v),f(w)> is an edge in G'.
- **Subgraph Isomorphism Decision Problem:** Given two graphs G=(V,E) and G'=(V',E') and a positive integer k, is there a function f from a subset V'' of V to a subset of V' which is 1-1 and such that the subgraph formed by V'' and all edges in G connecting edges in V'' is isomorphic to a subgraph of G'.
- **Longest Path Decision Problem:** , Given a weighted graph G=(V,E) and a positive integer k, is there a simple path in G of length at least k? (Contrast this with shortest path!)
- **Longest Circuit Decision Problem:** Given a weighted graph G=(V,E) and a positive integer k, is there a simple circuit in G of length at least k?

SETS

- **Partition Decision Problem (Partition):** Given a sequence of positive integers $S=m_1, m_2, ..., m_n$, is there a subsequence S' of S such that the sum of the integers in S' is equal to the sum of the integers in S-S'? (That is, can you "partion" S into two equal sum subsequences?)
- **3-dimensional Matching (3DM):** Suppose you have three equal sized sets X, Y, and Z. Suppose we have a set of triples S a subset of XxYxZ. Is there a subset of S which contains each element of X, Y, and Z in a triple exactly once? (Notice that this is, essentially, like a 3-gender version of the marriage problem.)
- **Subset Sum:** Given a set S of integers (both positive and negative), is there a subset S' such that the sum of the elements in S' is 0?
- **Set Cover Decision Problem (SCDP):** Given a collection of subsets of positive integers S and a positive integer k, are there k or fewer subsets in S whose union is the union of all sets in S?
- **Exact Cover Decision Problem (Exact Cover):** Given a collection S of subsets of X is there a subcollection S' of S such that each element of X occurs in exactly one of the subsets in S'? (Another way of saying this is: Is there a subcollection S' of S where the subsets in S' are all mutually disjoint and the union of the subsets of S' is X?)

Sets

- **0/1 Knapsack Problem (Knapsack):** Given a set of positive integers $\{w_1,w_2,...,w_n\}$ (referred to as weights) and another set of positive integers $\{v_1,v_2,...,v_n\}$ and two positive integers W and V, can you find a subset S of the numbers 1 to n - say $S=\{i_1,i_2,...,i_n\}$ - such that the sum of the weights corresponding to the indices in S is no more than W and the sum of the values corresponding to the indices in S is at least V?
- **3-Partition:** Given a set S of positive integers of size 3*m, can you divide S into m sets of 3 elements such that the sum of the numbers in each subset is equal?
- **Hitting Set Decision Problem (Hitting Set):** Given a collection C of sets of positive integers and a positive integer k, can you find a single set S of positive integers of size at most k such that S has non-empty intersection with every set in C?

Data Storage:

- **File Allocation:** Given a set of m storage devices with capacties C_1, C_2, ..., C_n and a set of files of lengths L_1, L_2, ..., L_n, can you fit all of the files into those devices?
- **Bin packing:** Given a set of objects of size S_1, S_2, ..., S_n and a positive integer k and another positive integer V, can you fit the objects into k or fewer bins of size V?
- **2-tape problem:** Given two storage devices of size V and a set of files of length L_1, L_2, ..., L_n and a positive integer M, can you allocate the files to the two devices so that the difference in length of files on the two devices is at most M?

Sequences and Strings:

- **Longest Common Subsequence:** Given two sequences S1 and S2 and a positive integer k, Is there a subsequence of S1 whcih is also a subsequence of S2 whose length is at least k? (For example, for the sequences abcadac and ebefagac there is a common subsequence baac.)
- **Shortest Common Supersequence:** Given two sequences S1 and S2 and a positive integer k, is there a sequence of length at most k for which S1 and S2 are both subsequences. (For example, if the two sequences are abcadac and ebefagac, then a supersequence is aebcefagdac.)

Database:

- **Boyce-Codd normal form violation:** Given a set of tables in a relational database, do they violate Boyce-Codd normal form?
- **Conjunctive Boolean Query:** Given a set of boolean queries $R_1(t_1)$, ..., $R_n(t_n)$ on the tables forming database D where each R_i is a relation symbol and each t_i is a tuple of variables and constants, does the conjunction of those queries evaluate to true? (I.e., is $R_1(t_1)R_2(t_2)...R_n(t_n)$ true? Note this is at least as tough as SAT.)

Scheduling:

- **Job sequencing with deadlines:** Given n jobs where job i has a deadline D_i and a profit P_i and a number P. Each job takes one unit time to complete. Can you find a subset $i_1, i_2, ..., i_m$ of the numbers 1 to n such that each of the jobs $j_1, j_2, ..., j_m$ is processed by their deadline and such that the sum of the profits for those jobs is at least P?
- **Sequencing with start times and deadlines:** Given n jobs, each with a minimum starting time, a deadline for completion, and a processing time as well as a positive integer M, can you find a way to distribute those jobs to M processors so that they all finish by their deadlines and start after their start times?

Scheduling:

- **Multiprocessor scheduling:** Given N jobs, each requiring a given amount of time to run t_1, t_2, ..., t_N and P processors and a positive integer T, can you assign the jobs to the P processors in such a way that all jobs are done by time T?
- **Job-shop scheduling:** Given jobs 1,..., N such that job j has n_j activities which must be completed in order and where each activity requires 1 time unit to complete and M machines with capacity 1 job at a time and a positive integer T, can you assign the tasks for the jobs to the M machines in such a way that the order of the n_j tasks is maintained for each job and such that the total time required is at most T?
- **Deadlock avoidance:** Given a set of N jobs which share M variables, can you schedule the jobs in such a way as to avoid Deadlock?

Logic:

- **Conjunctive Normal Forrm Satisfiability (CNFSAT or SAT):** Given a logical expression in Conjunctive Normal Form (i.e., product of sums), is there an assignment to the variables which can make the expression true?
- **3-term Conjunctive Normal Form Satisfiability (3CNFSAT):** Given a logical expression in Conjunctive Normal Form where each sum contains exactly three terms, is there an assignment to the variables which can make the expression true?
- **Disjunctive Normal Form Tautology (DNFTAUT):** Is the logical expression in Disjunctive Normal Form (i.e., a sum of products) a tautology - i.e., true for every truth value assignment to the variables in the expression? (Note that this is the complement of SAT since the not of a CNF expression is automatically a DNF expression.)

Code Generation:

- **Feasible Register Assignment:** Suppose we have a program which we are compiling with N variables that require storage. Suppose we have m<N CPU registers. Suppose further we have a positive integer T. Can you determine a way to assign variables to registers throughout the running of the program so that the total running time is at most T? (Note that access and manipulation of registers is assumed to be faster than access and manipulation of things in memory.)

Exercises

Exercises

- List three problems that have polynomial-time algorithms. Justify your answer.
- Give a problem and two encoding schemes for its input. Express its performance using your encoding schemes.
- Write a non-deterministic algorithm to compute minimum spanning tree for a given full connected weighted Graph?

Exercises

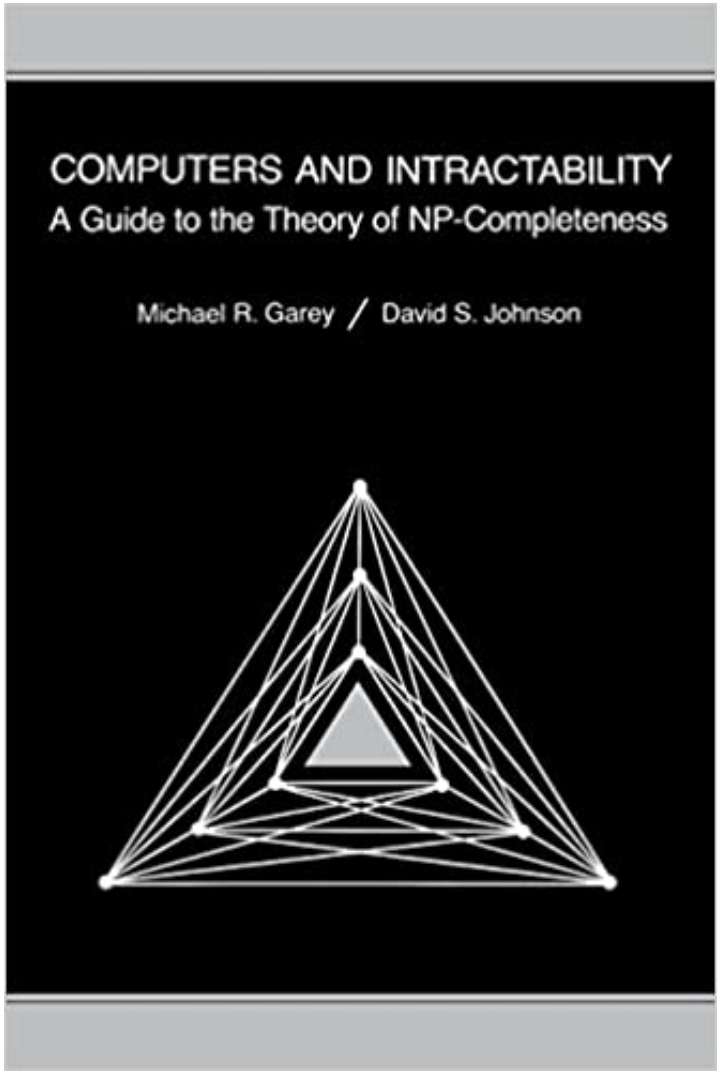
1. Show that a graph problem using the number of vertices as the measure of the size of an instance is polynomially equivalent to one using the number of edges as the measure of the size of an instance.
2. Write a polynomial-time verification algorithm for the Clique Decision problem.
3. Show that the reduction of the CNF-Satisfiability problem to the Clique Decision problem can be done in polynomial time.
4. Write a polynomial-time verification algorithm for the Hamiltonian Circuits Decision problem.
5. Show that the reduction of the Hamiltonian Circuits Decision problem to the Traveling Salesperson (Undirected) Decision problem can be done in polynomial time

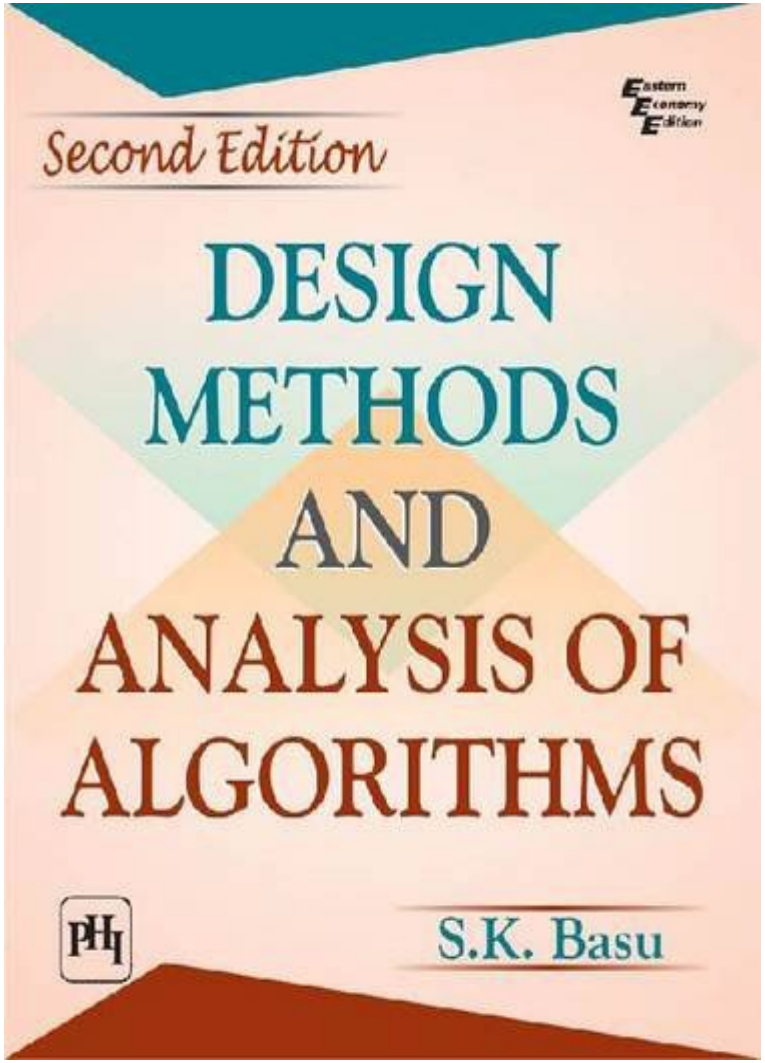
Exercises

1. Show that the reduction of the Traveling Salesperson (Undirected) Decision problem to the Traveling Salesperson Decision problem can be done in polynomial time.
2. Show that a problem is *NP-easy* if and only if it reduces to an *NP-complete* problem.
3. Suppose that problem *A* and problem *B* are two different decision problems. Furthermore, assume that problem *A* is polynomial-time many-one reducible to problem *B.* If problem *A* is *NP-complete,* is problem *B NP* complete? Justify your answer.
4. When all instances of the CNF-Satisfiability problem have exactly three literals per clause, it is called the 3-Satisfiability problem. Knowing that the 3-Satisfiability problem is *NP-complete,* show that the Graph 3-Coloring problem is also *NP-complete.*

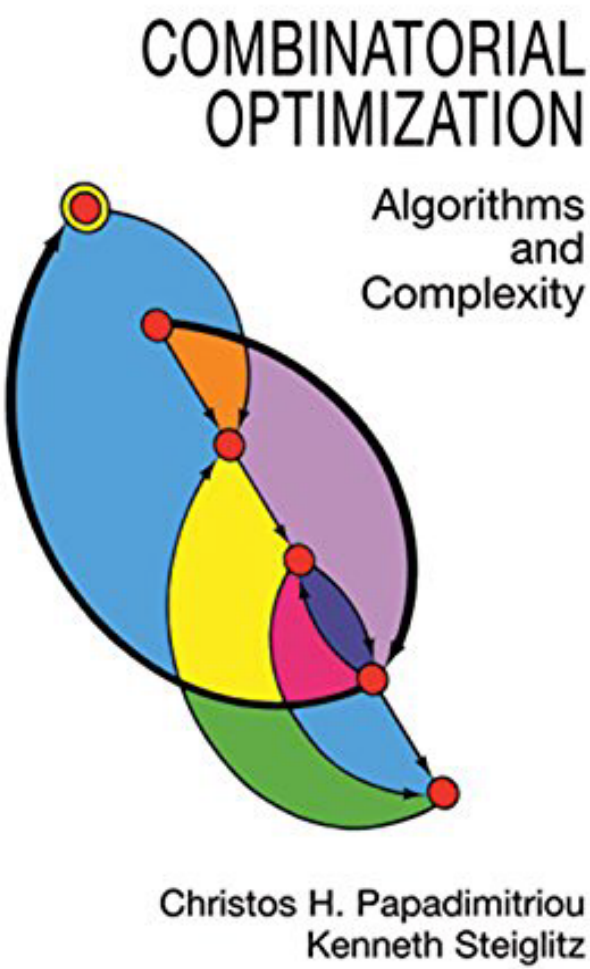

Thank You

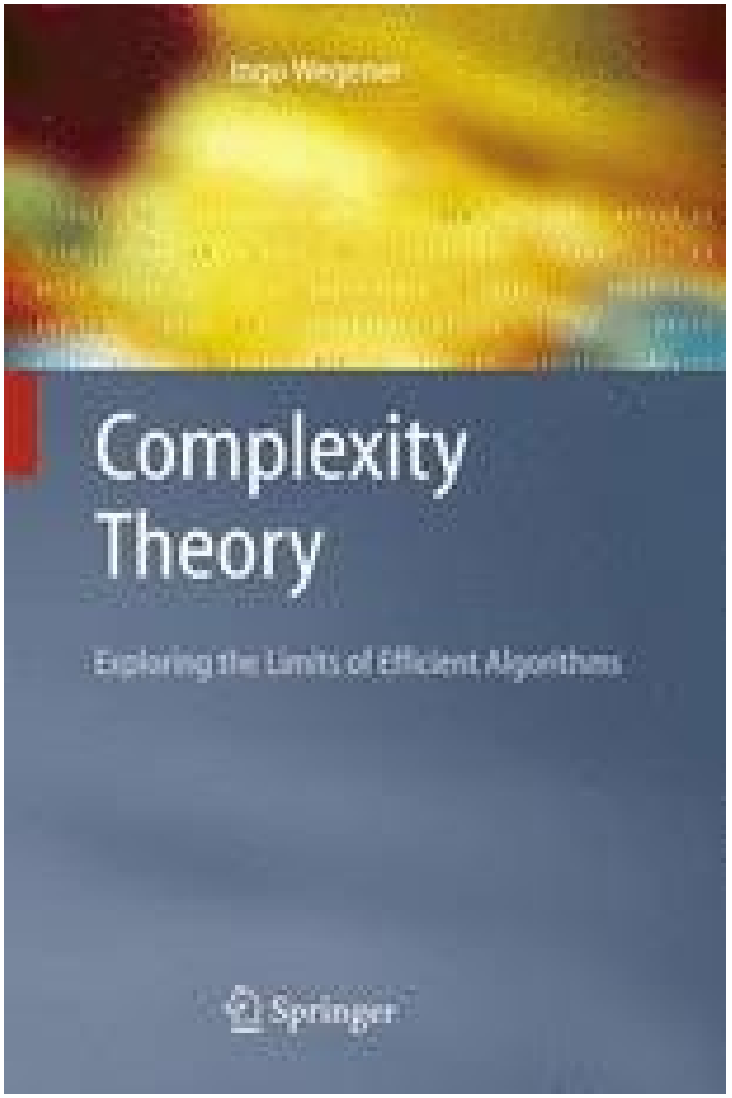
References


COMPUTERS AND INTRACTABILITY
A Guide to the Theory of NP-Completeness
Michael R. Garey / David S. Johnson


Second Edition
DESIGN METHODS AND ANALYSIS OF ALGORITHMS
S.K. Basu

References


COMBINATORIAL
OPTIMIZATION
Algorithms
and
Complexity
Christos H. Papadimitriou
Kenneth Steiglitz


Ingo Wegener
Complexity
Theory
Springer

Methods adopted to cope with NP-complete Problem

- Use dynamic programming, backtracking or branch-and-bound technique to reduce the computational cost. This might work if the problem size is not too big.
- Find the sub-problem of the original problem that have polynomial time solution.
- Use **approximation algorithms** to find approximate solution in polynomial time.
- Use **randomized algorithm** to find solutions is affordable time with a high probability of correctness of the solution.
- Use **heuristic** like greedy method, simulated annealing or genetic algorithms etc. However, the solutions produced cannot be guaranteed to be within a certain distance from the optimal solution.

Running times for different sizes of input.

n	$\log n$	n	$n \log n$	n^2	n^3	2^n
8	3 nsec	0.01 μ	0.02 μ	0.06 μ	0.51 μ	0.26 μ
16	4 nsec	0.02 μ	0.06 μ	0.26 μ	4.10 μ	65.5 μ
32	5 nsec	0.03 μ	0.16 μ	1.02 μ	32.7 μ	4.29 sec
64	6 nsec	0.06 μ	0.38 μ	4.10 μ	262 μ	5.85 cent
128	0.01 μ	0.13 μ	0.90 μ	16.38 μ	0.01 sec	10^{20} cent
256	0.01 μ	0.26 μ	2.05 μ	65.54 μ	0.02 sec	10^{58} cent
512	0.01 μ	0.51 μ	4.61 μ	262.14 μ	0.13 sec	10^{135} cent
2048	0.01 μ	2.05 μ	22.53 μ	0.01 sec	1.07 sec	10^{598} cent
4096	0.01 μ	4.10 μ	49.15 μ	0.02 sec	8.40 sec	10^{1214} cent
8192	0.01 μ	8.19 μ	106.50 μ	0.07 sec	1.15 min	10^{2447} cent
16384	0.01 μ	16.38 μ	229.38 μ	0.27 sec	1.22 hrs	10^{4913} cent
32768	0.02 μ	32.77 μ	491.52 μ	1.07 sec	9.77 hrs	10^{9845} cent
65536	0.02 μ	65.54 μ	1048.6 μ	0.07 min	3.3 days	10^{19709} cent
131072	0.02 μ	131.07 μ	2228.2 μ	0.29 min	26 days	10^{39438} cent
262144	0.02 μ	262.14 μ	4718.6 μ	1.15 min	7 mnths	10^{78894} cent
524288	0.02 μ	524.29 μ	9961.5 μ	4.58 min	4.6 years	10^{157808} cent
1048576	0.02 μ	1048.60 μ	20972 μ	18.3 min	37 years	10^{315634} cent

Note: “nsec" stands for nanoseconds, “ μ”is one microsecond and “cent" stands for centuries. The explosive running time (measured in centuries) when it is of the order 2^n.