

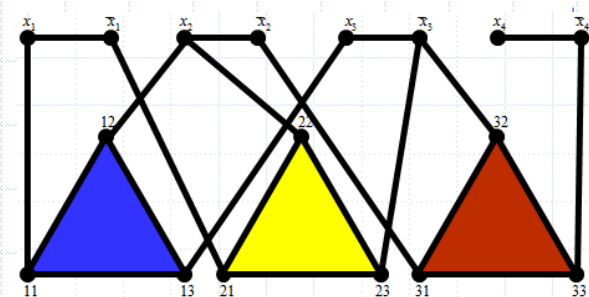
Lesson 15

NP-Complete Problems: *Handling Problems From the Field of All Possibilities*

Wholeness of the Lesson

Decision problems that have no known polynomial time solution are considered *hard*, but hard problems can be further classified to determine their degree of hardness. A decision problem belongs to NP if there is a polynomial p and an algorithm A such that for any instance of the problem of size n , a correct solution to the problem can be *verified* using A in at most $p(n)$ steps. In addition, the problem is said to be *NP-complete* if it belongs to NP and every NP problem can be polynomial-reduced to it.

Science of Consciousness: The human intellect can grasp truths within a certain range but is not the only faculty of knowing. The transcendental level of awareness is a field beyond the grasp of the intellect ("beyond even the intellect is he" -- Gita, III.42). And the field of manifest existence, from gross to subtle, is too vast and complex to be grasped by the intellect either ("unfathomable is the course of action" – Gita IV.17).

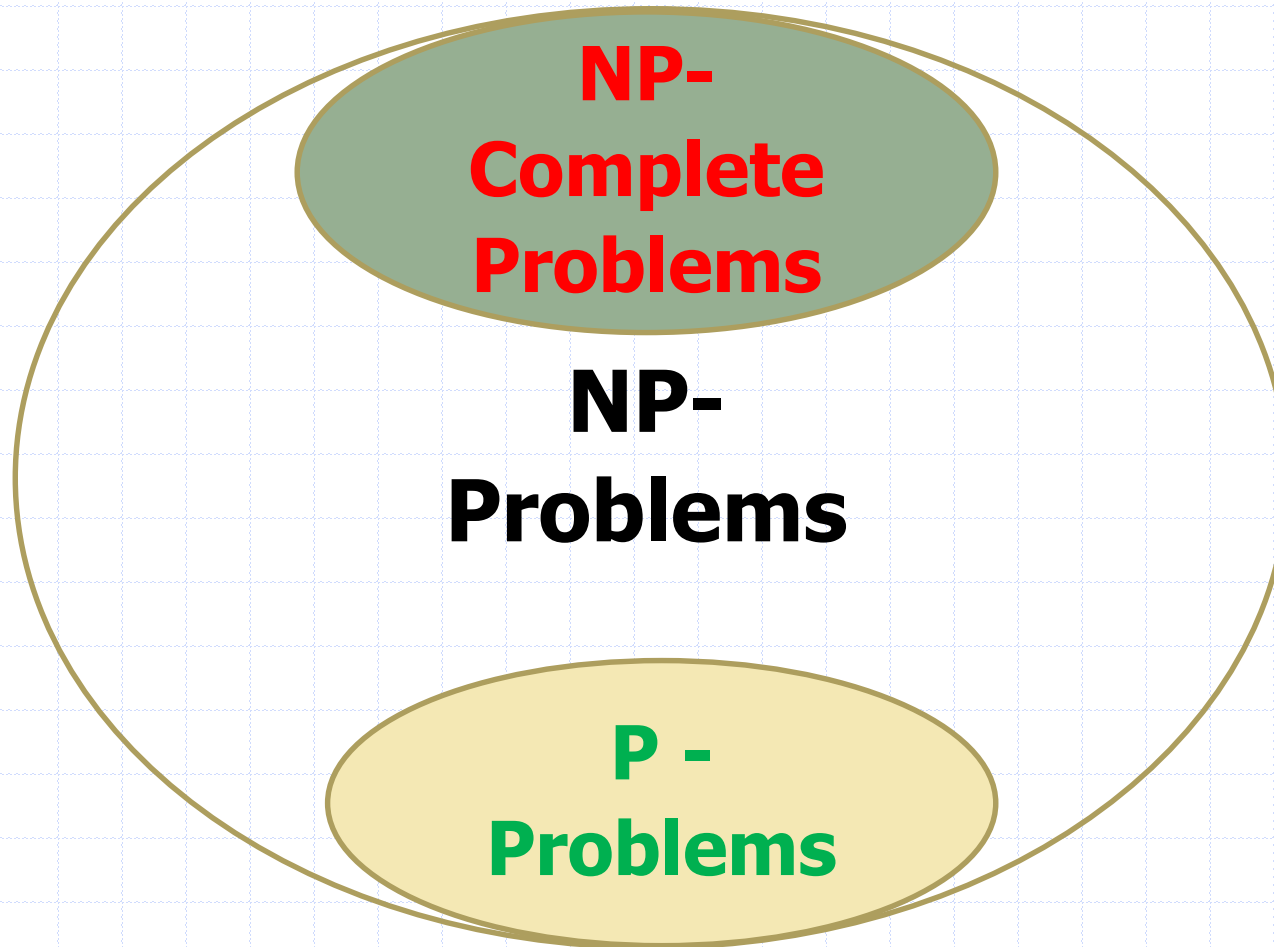


Overview

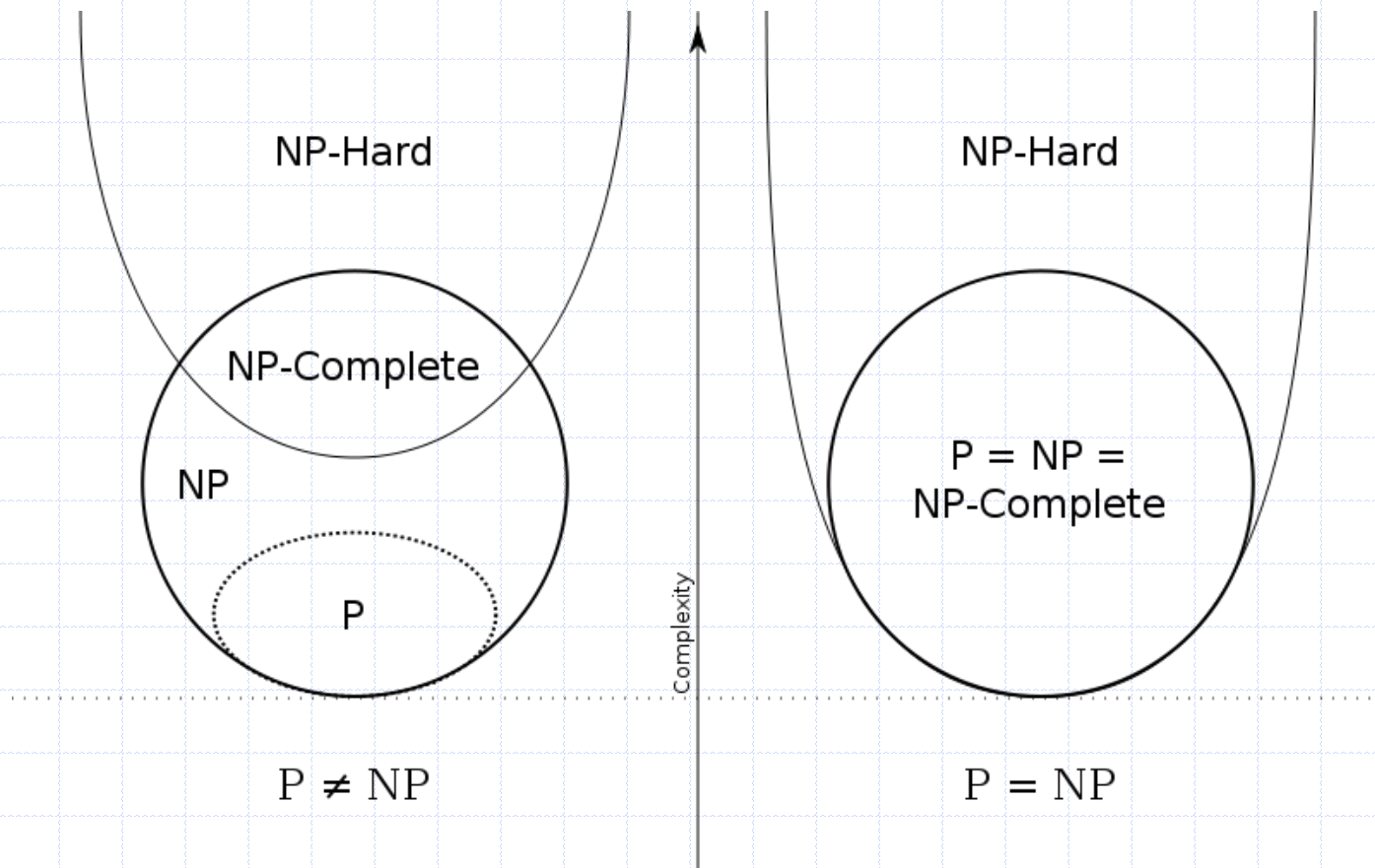
In this lesson:

- ◆ Review polynomial time decision problems and the class P
- ◆ Review the class NP of decision problems.
- ◆ Describe the class of “hard” NP problems – the *NP complete problems*.
- ◆ Demonstrating NP-completeness, with examples, and the $P=NP$ problem
- ◆ Techniques for handling NP-completeness in practice, with examples.

P versus NP problem



P versus NP problem



P versus NP problem versus Co-NP

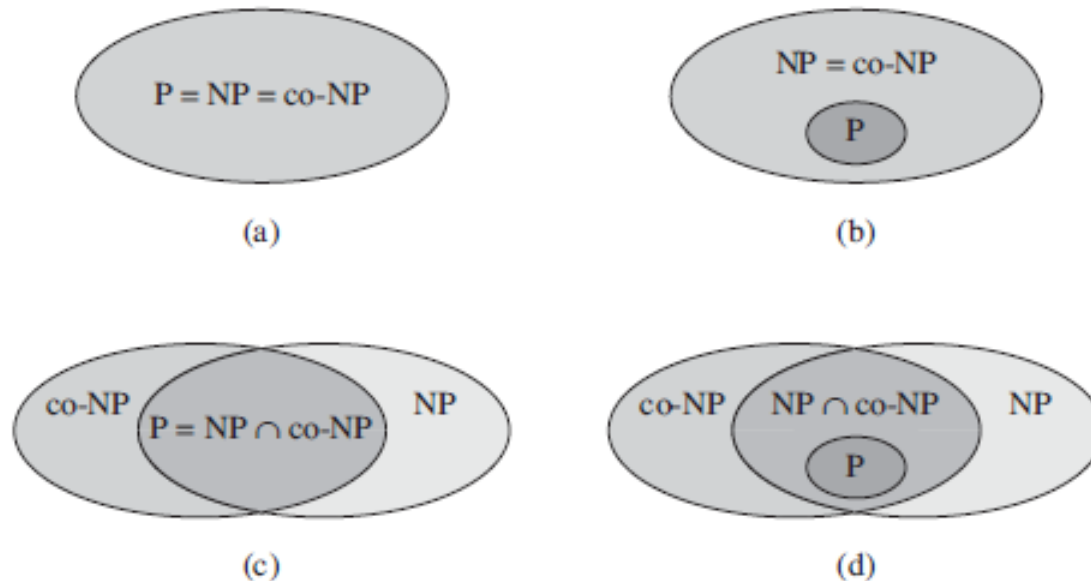


Figure 34.3 Four possibilities for relationships among complexity classes. In each diagram, one region enclosing another indicates a proper-subset relation. (a) $P = NP = \text{co-NP}$. Most researchers regard this possibility as the most unlikely. (b) If NP is closed under complement, then $NP = \text{co-NP}$, but it need not be the case that $P = NP$. (c) $P = NP \cap \text{co-NP}$, but NP is not closed under complement. (d) $NP \neq \text{co-NP}$ and $P \neq NP \cap \text{co-NP}$. Most researchers regard this possibility as the most likely.

Polynomial-Time Algorithms

- Are some problems solvable in polynomial time?
 - Of course: many algorithms we've studied provide polynomial-time solutions to some problems
- Are all problems solvable in polynomial time?
 - No: Turing's "Halting Problem" is not solvable by any computer, no matter how much time is given
- Most problems that do not yield polynomial-time algorithms are either optimization or decision problems.

How Do We Know When a Problem Does Not Belong to P ?

- ◆ Hard to know for sure because even if there is no known polynomial time algorithm today, tomorrow someone may come up with one.
- ◆ Modern-day example: The **IsPrime** problem. Before 2002, all known deterministic algorithms to solve this problem ran in superpolynomial time.
- ◆ **AKS Primality Test** was the first polynomial-time solution. Its fastest known implementation runs in $O(\text{length}(n)^6 * \log^k(\text{length}(n)))$ for some k . (AKS stands for Agrawal–Kayal–Saxena)
- ◆ *EXPTIME-Complete problems*, like $n \times n$ chess, *do require* superpolynomial time and therefore do not belong to P

The class NP

- ◆ To understand decision problems that may not belong to P , one approach is to see how hard it is to *check* whether a given solution is correct. Typically easier to check a solution than to obtain a solution in the first place.
- ◆ The class of decision problems with the property that a solution can be verified in polynomial time is denoted NP .

Optimization/Decision Problems

- Optimization Problems
 - An optimization problem is one which asks, “What is the optimal solution to problem X?”
 - Examples:
 - 0-1 Knapsack
 - Fractional Knapsack
 - Minimum Spanning Tree
- Decision Problems
 - A decision problem is one with yes/no answer
 - Examples:
 - Does a graph G have a MST of weight $\leq W$?

Optimization/Decision Problems

- An **optimization problem** tries to find an optimal solution
- A **decision problem** tries to answer a yes/no question
- Many problems will have decision and optimization versions
 - Eg: Traveling salesman problem
 - optimization: find hamiltonian cycle of minimum weight
 - decision: is there a hamiltonian cycle of weight $\leq k$
- Some problems are decidable, but **intractable**: as they grow large, we are unable to solve them in reasonable time
 - *Is there a polynomial-time algorithm that solves the problem?*

Decision Problems and Instances of Problems

- ◆ Example of a Decision Problem (VertexCover)

Given a graph $G = (V, E)$ and a positive integer k , is there a vertex cover for G of size at least k ?

- ◆ Example of an *instance* of a Decision Problem:

Is there a vertex cover of size at least 3 for the complete graph on 5 vertices?

- ◆ An instance of a Decision Problem is said to *have a solution* or be a *solvable instance* if “true” is the correct answer to the problem.

- ◆ *Example:* “Is there a vertex cover of size at least 3 for the complete graph on 5 vertices?” *has a solution* (so “true” is the correct answer)
- ◆ *Example:* “Is there a vertex cover of size at least 3 for the complete graph on 8 vertices?” *does not* have a solution (so “true” is *not* correct)

(continued)

- ◆ An instance I of a problem has *input data* (e.g. (V, E, k) for VertexCover); any *candidate solution* for an instance also consists of data (e.g. a subset W of V for VertexCover) called *solution data*.
- ◆ Need to know the *size* of input data. For graphs, the convention is that the number of vertices is the size that is used

NP-Complete Vrs. NP-Hard

- ◆ **NP-complete** problems are a set of problems that any other **NP**-problem can be reduced to in polynomial time, but retain the ability to have their solution verified in polynomial time. Alternatively, A problem (in NP) is NP-complete if any problem in NP is reducible to it

Examples: Traveling Salesman Problem & Hamiltonian Cycle problem

- ◆ **NP-hard** problems are those at least as hard as **NP**-complete problems, meaning all **NP**-problems can be reduced to them, but not all **NP-hard** problems are in **NP**, meaning not all of them have solutions verifiable in polynomial time.

Another Definition of P

◆ **Definition** A decision problem Q is said to *belong to P* if there exists a polynomial $p(y)$ (“polynomial witness”) and an algorithm A such that for any instance I of Q that has a solution and that has input data X of size n , when A is run on input X , A outputs “true” in $O(p(n))$ time.

We say $A, p(y)$ *witness* that Q belongs to P .

Another Definition of NP

◆ **Definition.** A decision problem Q is said to belong to NP if there exists a polynomial $p(y)$ (“polynomial witness”) and an algorithm B such that for any instance I of Q that has a solution, if X is input data of size n and Y is a solution data (called a *certificate*), then when B is run with input (X, Y) , B outputs the message “verified” in $O(p(n))$ time. (Time includes the effort to read the certificate Y .)

We say that $B, p(y)$ *witness* that Q belongs to NP

Yet Another Definition of NP

NP: the class of decision problems that are solvable in polynomial time on a *nondeterministic* machine (or with a nondeterministic algorithm)

- ◆ (A *deterministic* computer is what we know)
- ◆ A *nondeterministic* computer is one that can “guess” the right answer or solution
 - Think of a nondeterministic computer as a parallel machine that can freely spawn ***an infinite number*** of processes (to some extent similar to **Quantum** Computing).
- ◆ Note that *NP* stands for “Nondeterministic Polynomial-time”

Quantum Computing and NPC

- ❖ Uses Qbit that can super impose inputs in a probabilistic way.
- ❖ a qubit is the superposition $|\text{Sai}\rangle = \alpha |0\rangle + \beta |1\rangle$ with probabilities α^2 and β^2 etc. α, β are complex numbers.
- ❖ Can store 2^n in a probabilistic way in n qbits etc.
- ❖ Thus, bits, algorithms, results are all probabilistic.
- ❖ At this time only some classes of problems can be solved fast in a quantum machine.
- ❖ Quantum computer cannot solve NPC or NP-Hard problems in Polynomial time (in general, may be some specific problems can be solved in P time)

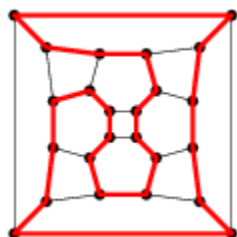
Quantum Computing Complexity

- ❖ **BQP** - : **bounded error quantum polynomial time** is the class of decision problems solvable by a quantum computer in polynomial time, with an error probability of at most $1/3$ for all instances.
- ❖ **QMA - Quantum Merlin Arthur**, is the quantum analog of the nonprobabilistic complexity class NP or the probabilistic complexity class MA.
- ❖ **QMA is related to BQP** in the same way NP is related to P, or MA is related to BPP.
(**BPP** is **bounded-error probabilistic polynomial time**)

NP Examples

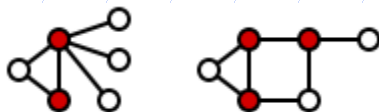
- ◆ The HamiltonianCycle and VertexCover problems have already been shown to belong to *NP*.

HamiltonianCycle: Given $G=(V,E)$, does G have a Hamiltonian Cycle? Solution data: a subset of E



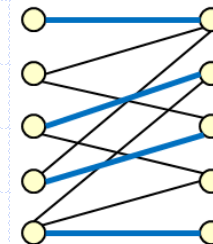
Hamiltonian path is a **path** in an undirected or directed graph that visits each vertex exactly once. A **Hamiltonian cycle** (or **Hamiltonian circuit**) is a **Hamiltonian path** that is a **cycle**

VertexCover: Given $G=(V,E)$ and k , does G have a vertex cover of size at least k ? Solution data: subset of V



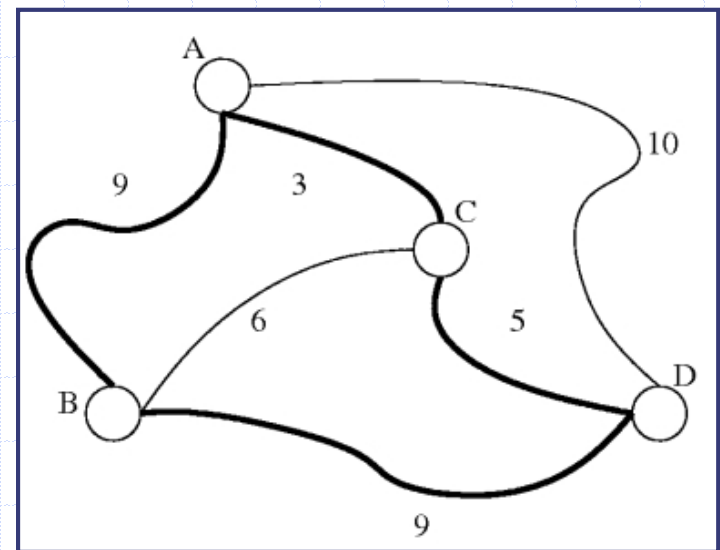
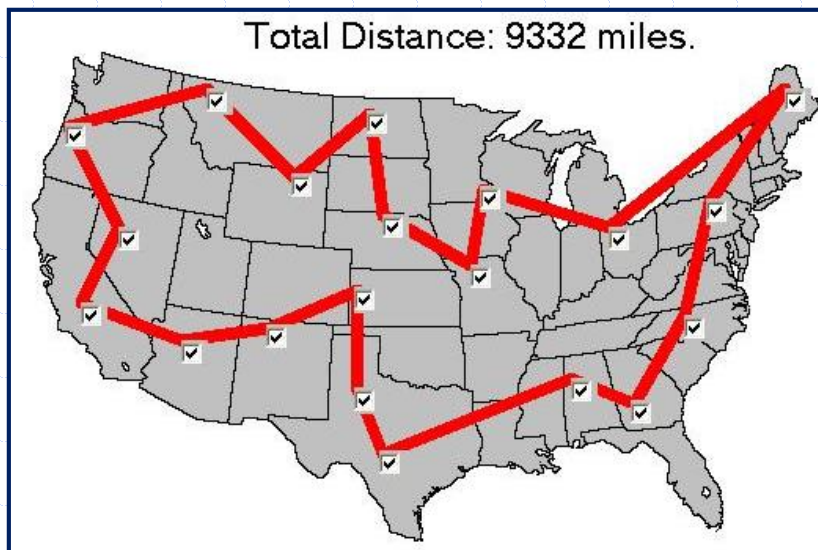
A matching in a graph is a set of edges no two of which share an endpoint.

Konig's Theorem. In a bipartite graph, the number of edges in a maximum matching is the same as the number of vertices in a minimum vertex cover.



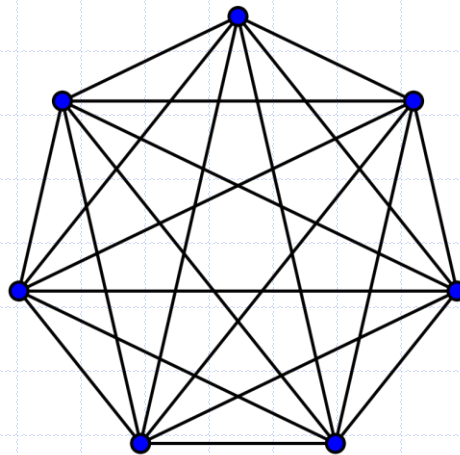
Traveling Salesman Problem

- ◆ *Traveling Salesman Problem (TSP)*: Given a **complete** graph G with cost function $c: E \rightarrow \mathbb{N}$ and a positive integer k , is there a Hamiltonian cycle C in G so that the sum of the costs of the edges in C is at most k ? Solution data: a subset of E .



Complete Graph, Induced Graph and Clique

A complete graph is a graph where every pair of distinct vertices are connected by a unique edge. Example: K_7 - a complete graph with 7 vertices



A subgraph, H , of a graph, G , is said to be **induced** (or **full**) if, for every pair of vertices x and y of H , xy is an edge of H if and only if xy is an edge of G .

A **clique** is subset of vertices of an undirected graph, such that its induced subgraph is complete.

TSP Belongs to NP

- ◆ Given an instance I of TSP with input data a complete graph G with n vertices, a cost function $c: E \rightarrow \mathbb{N}$ and a positive integer k , and given a certificate E_0 , a subset of E , the algorithm B checks that E_0 is a Hamiltonian cycle for G and then computes the sum of $c(e)$ over the edges e in E_0 to verify that it is at most k .

Example: Not in *NP*

- ◆ *PowerSet problem.* Given a set X of size n , a kind of optimization problem concerning the power set of X is to generate all subsets of X ("find the largest possible collection of subsets of X without duplicates"). Whatever method is used, just writing out the output requires at least 2^n steps. A corresponding decision problem is: Given a set X and a collection P , is $P = P(X)$? Any algorithm that solves this problem must check every element of P to see if it is a subset of X , so the algorithm requires at least 2^n steps in the worst case. So this problem does not belong to P .

Moreover, verifying correctness requires checking that each set that is output is a subset of X , and this has to be done 2^n times, so it doesn't belong to NP either.

[A Power Set is a set of *all the subsets of a set*. E.g. for set $\{a,b,c\}$, the Power set is $\{ \{\}, \{a\}, \{b\}, \{c\}, \{a,b\}, \{a,c\}, \{b,c\}, \{a,b,c\} \}$]

- ◆ *Finite Halting Problem and $N \times N$ Chess.* Finite Halting Problem: Given a program P and positive integers n, k , does P , when running with input n , produce an output after executing k or fewer steps? Neither of these problems belong to P , but it is not known whether they belong to NP .

Is $P = NP$?

- ◆ The answer is not known
- ◆ Many thousands of research papers have been written in an attempt to make progress in solving this problem.
- ◆ If it were true, then thousands of problems that were believed to have infeasible solutions would suddenly have feasible solutions
- ◆ **Clay Mathematics Institute** (CMI, Rhode Island, Providence, USA) in 2000 has announced \$1M award if one can prove that $P = NP$ or $P \neq NP$.

[In fact, there are 6 other problems, each for \$1M.
These are called Millennium Prize Problems]

A Remarkable Fact About **NP**

- ◆ Many of the problems that are known to be in NP can also be shown to be *NP-complete*. Intuitively, this means they are the hardest among the NP problems.
- ◆ It can be shown that if someone ever figures out a polynomial-time algorithm to solve an NP-complete problem, then *all problems in NP will also have polynomial time solutions*.
- ◆ One consequence: If anyone finds a polynomial time solution to an NP-complete problem, then $P = NP$.
- ◆ These points are elaborated in the next slides

Reducibility

- ◆ *Intuitively:* Q is *polynomial reducible* to R if, in polynomial time, you can transform a solvable problem of type Q into a solvable one of type R so that a solution to one yields a solution to the other.
- ◆ *Formally:* A problem Q is *polynomial reducible* to a problem R if there is a polynomial $p(y)$ and an algorithm C so that when C runs on input data X of size $O(n)$ for an instance I_Q of Q, C outputs input data Y of size $O(p(n))$ for an instance I_R of R in $O(p(n))$ steps so that

I_Q has a solution iff I_R has a solution

In this case, we say that C, $p(y)$ *witness* that Q is polynomial reducible to R.

- ◆ We write $Q \xrightarrow{\text{poly}} R$

VertexCover ^{poly} → HamiltonianCycle

It can be shown that VertexCover is polynomial reducible to HamiltonianCycle. What does this mean?

- ◆ *Formally:* There is a polynomial $p(y)$ and an algorithm C that does the following: Using an instance G, k of the VertexCover problem (where G has n vertices) as input to C , C outputs, in $O(p(n))$ time, a graph H with $O(p(n))$ vertices so that:
 G has a vertex cover of size at most k iff
 H has a Hamiltonian cycle
- ◆ *Intuitively:* the VertexCover problem can be turned into a Hamiltonian Cycle problem in polynomial time, so if you can solve the Hamiltonian Cycle problem, you can solve the VertexCover problem without doing much more work.
- ◆ *Note:* The details for this algorithm C are tricky – not given here, but we make use of this result later.

HamiltonianCycle $\xrightarrow{\text{poly}}$ TSP

We show HamiltonianCycle is reducible to TSP

- ◆ Given a graph $G = (V, E)$ on n vertices (input for HamiltonianCycle) – notice G is a subgraph of K_n . Obtain an instance H, c, k of TSP as follows: Let H be the complete graph on n vertices (i.e. H is K_n), obtained by adding the missing edges to G . Let $c(e) = 0$ if $e \in E$, else $c(e) = 1$. Let $k = 0$.
- ◆ Need to show: G has a Hamiltonian cycle if and only if H, c, k has a Hamiltonian cycle with edge cost $\leq k$
- ◆ If G has Hamiltonian cycle C , C is Hamiltonian in H also. Since each edge e of C is in G , $c(e) = 0$. So cost sum $\leq k$. Converse: A solution C for H, c, k implies all edges of C have weight 0; therefore, every edge of C also is an edge in G . Therefore C is an HC in G .

NP-Complete Problems (Review)

A problem Q is ***NP-complete*** if Q belongs to NP , and for *every* problem R in ***NP***, R is polynomial reducible to Q .

Exptime (EXP) and Exptime-Complete Problems

- ❖ A problem Q is **EXPTIME-complete** if Q belongs to **EXP**, and for *every* problem R in **EXP**, R is polynomial time reducible to Q .
- ❖ Notice that although we don't know if NP is equal to P or not, we do know that EXPTIME-complete problems are not in P; it has been proven that these problems cannot be solved in polynomial time. These are harder than NP-Complete problems.

The First NP-Complete Problem

- ◆ Cook-Levin discovered the first NP-complete problem, known as the Satisfiability Problem (called SAT). The proof is complicated. We are skipping the proof (***use logic table to show the idea***).
- ◆ SAT is the following problem: Given an expression using boolean variables p, q, r, \dots joined together using connectives AND, OR, NOT, is there a way to assign values “true” or “false” to the variables so that the expression evaluates to true?

Example: Consider

$p \text{ AND NOT } (q \text{ OR } (\text{NOT } r))$

Assigning T to p and to r and assigning F to q causes the expression to evaluate to T (i.e. “true”).

The *Satisfiability* (SAT) Problem

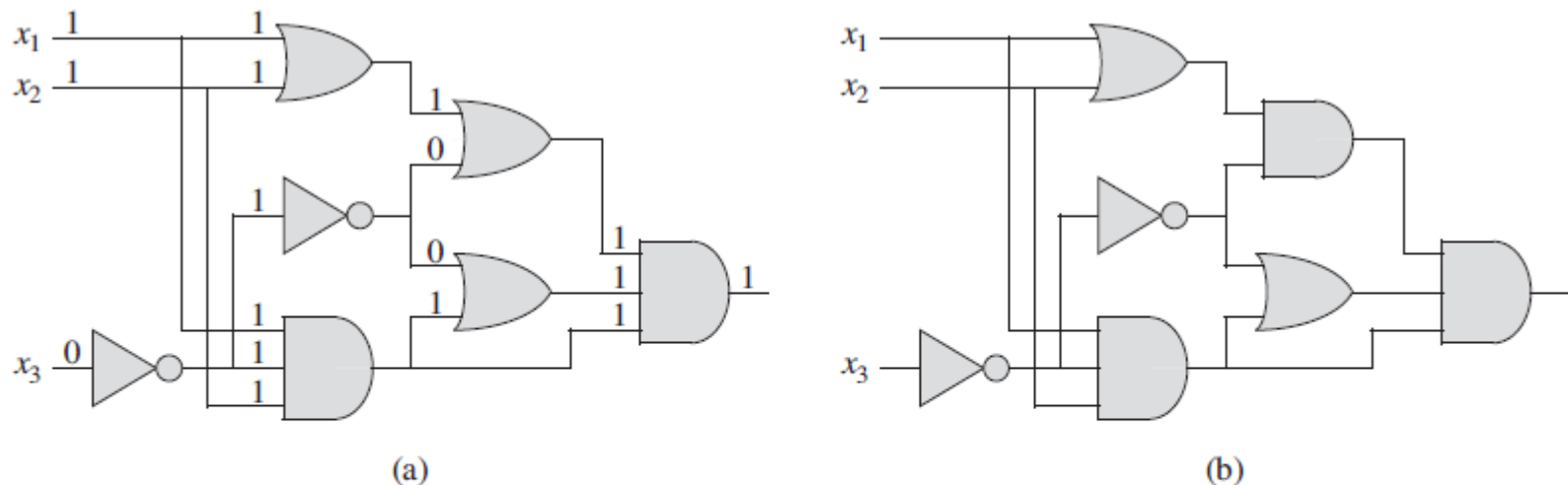


Figure 34.8 Two instances of the circuit-satisfiability problem. (a) The assignment $\langle x_1 = 1, x_2 = 1, x_3 = 0 \rangle$ to the inputs of this circuit causes the output of the circuit to be 1. The circuit is therefore satisfiable. (b) No assignment to the inputs of this circuit can cause the output of the circuit to be 1. The circuit is therefore unsatisfiable.

Other NP-Complete Problems

- ◆ Once a single NP-complete problem was discovered, others could be found by using the Cook-Levin result
- ◆ *Example:* VertexCover can be shown to be NP-complete. The main idea is to prove that SAT is polynomial reducible to VertexCover. Then: Given any NP problem Q, to show Q is polynomial reducible to VertexCover, observe:
 - Q is polynomial reducible to Sat
 - Sat is polynomial reducible to VertexCover
 - Therefore, Q is polynomial reducible to VertexCover(the last step requires verification: Exercise: if PROB1 is poly reducible to PROB2 and PROB2 is poly reducible to PROB3, then PROB1 is poly reducible to PROB3.)

More on Reducibility (From Corman)

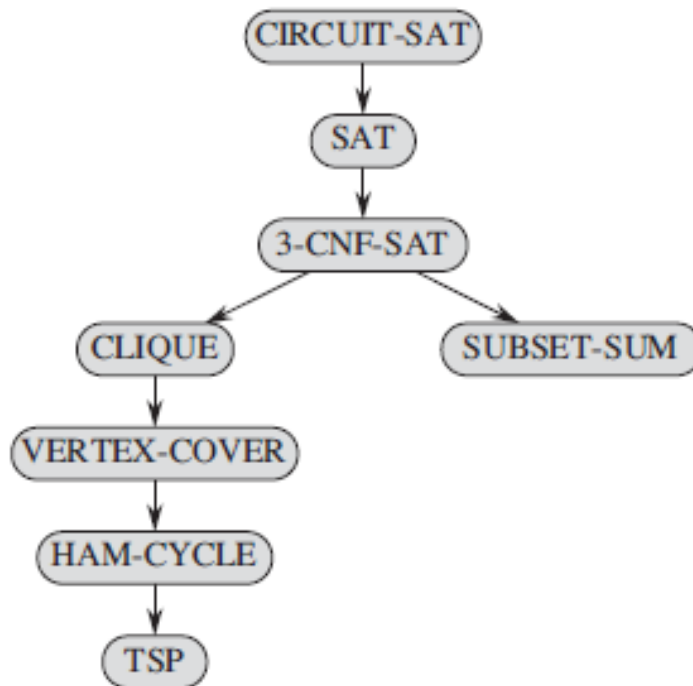


Figure 34.13 The structure of NP-completeness proofs in Sections 34.4 and 34.5. All proofs ultimately follow by reduction from the NP-completeness of CIRCUI-T-SAT.

HamiltonianCycle is NP-Complete

This is an outline of a proof that HamiltonianCycle is NP-Complete under the assumption that VertexCover is NP-complete:

- ◆ Recall that VertexCover is polynomial reducible to HamiltonianCycle.
- ◆ Therefore, since VertexCover is NP-complete, given any NP problem Q , Q is polynomial reducible to VertexCover
- ◆ Therefore, any such Q is polynomial reducible to HamiltonianCycle as well.

CSAT is *NP*-complete

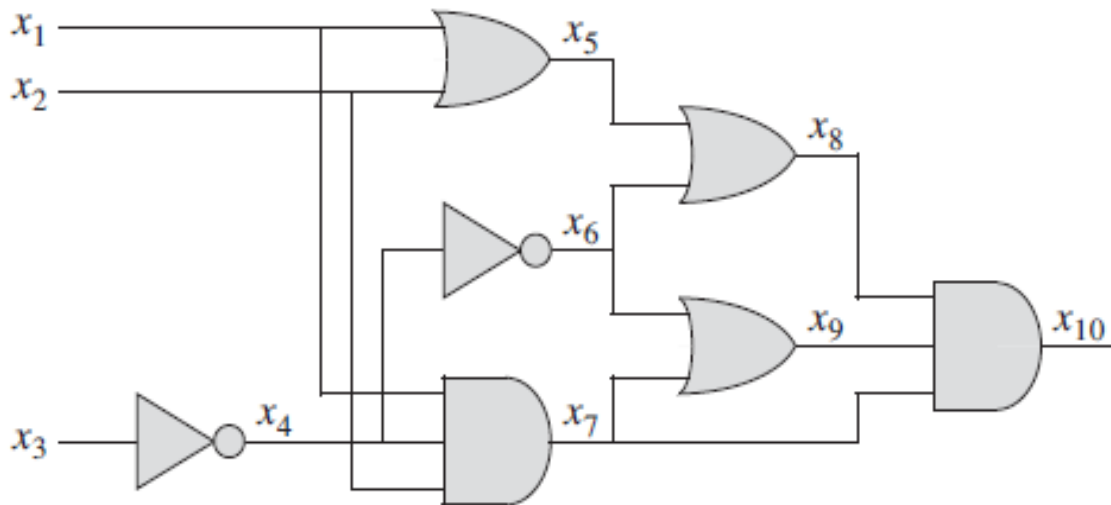


Figure 34.10 Reducing circuit satisfiability to formula satisfiability. The formula produced by the reduction algorithm has a variable for each wire in the circuit.

SAT is *NP*-complete

$$\begin{aligned}\phi = & x_{10} \wedge (x_4 \leftrightarrow \neg x_3) \\ & \wedge (x_5 \leftrightarrow (x_1 \vee x_2)) \\ & \wedge (x_6 \leftrightarrow \neg x_4) \\ & \wedge (x_7 \leftrightarrow (x_1 \wedge x_2 \wedge x_4)) \\ & \wedge (x_8 \leftrightarrow (x_5 \vee x_6)) \\ & \wedge (x_9 \leftrightarrow (x_6 \vee x_7)) \\ & \wedge (x_{10} \leftrightarrow (x_7 \wedge x_8 \wedge x_9)) .\end{aligned}$$

Given a circuit C , it is straightforward to produce such a formula ϕ in polynomial time.

3-CNF is NP Complete

- ◆ The following Boolean Formula is in 3-CNF (Conjunction of Disjunctions) form:

$$(x_1 \vee \neg x_1 \vee \neg x_2) \wedge (x_3 \vee x_2 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_4)$$

- ◆ Any Boolean Expression can be converted to CNF form (next page)

Conversion to CNF

$$B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$$

1. Eliminate \Leftrightarrow , replacing $\alpha \Leftrightarrow \beta$ with $(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$.
 $(B_{1,1} \Rightarrow (P_{1,2} \vee P_{2,1})) \wedge ((P_{1,2} \vee P_{2,1}) \Rightarrow B_{1,1})$
2. Eliminate \Rightarrow , replacing $\alpha \Rightarrow \beta$ with $\neg \alpha \vee \beta$.
 $(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge (\neg(P_{1,2} \vee P_{2,1}) \vee B_{1,1})$
3. Move \neg inwards using de Morgan's rules and double-negation:
 $(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge ((\neg P_{1,2} \wedge \neg P_{2,1}) \vee B_{1,1})$
4. Apply distributive law (\wedge over \vee) and flatten:
 $(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge (\neg P_{1,2} \vee B_{1,1}) \wedge (\neg P_{2,1} \vee B_{1,1})$

3-CNF is NP Complete

- ◆ Thus the following formula can be converted to 3-CNF in Polynomial time

$$\phi = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2. \quad (34.3)$$

- ◆ Thus, SAT can be reduced to 3-CNF in Polynomial time. Hence 3-CNF is NP-Complete as SAT is NP-Complete.

Solving One NP-Complete Problem Solves Them All

Suppose Q is an NP-complete problem and someone finds a polynomial-time algorithm A that solves it in $O(p(n))$ time.

Let R be any problem in NP. R is polynomial reducible to Q , with witness polynomial $q(y)$.

Polynomial-time algorithm to solve R : Given an instance I_R of R , create an instance I_Q of Q (in $O(q(n))$ time) that has a solution iff I_R does. Solve I_Q in $O(p(q(n)))$ time. Output solution to I_R . The algorithm runs in $O(q(n) + p(q(n)))$.

Main Point

The hardest NP problems are *NP-complete*. These require the highest degree of creativity to solve. However, if a polynomial-time algorithm is found for any one of them, then all NP problems will automatically be solved in polynomial time. This phenomenon illustrates the fact that the field of pure consciousness, the source of creativity, is itself a field of *infinite correlation* – “an impulse anywhere is an impulse everywhere”.

What To Do with NP-Complete Problems?

There are many thousands of NP-complete problems, and a large percentage of these would be very useful if they could be implemented in a feasible way. But the known algorithms are too slow. What can be done?

Handling NP-Hard Problems

- ◆ Find a more efficient algorithm somehow
- ◆ The IsPrime breakthrough of 2002
- ◆ The technique of dynamic programming – examples: SubsetSum (Also Knapsack as shown before)
- ◆ Sometimes exponential running time is good enough
- ◆ Approximation algorithms
- ◆ Probabilistic algorithms
- ◆ Use hard problems as an advantage - cryptosystems

Options: Find a Better Algorithm

In the case of NP problems not known to be NP-complete, this is at least a reasonable goal.

Example: a polynomial time solution to the IsPrime problem was discovered in 2002; all known (deterministic) algorithms before 2002 were exponential

Example: Subset Sum Problem using Dynamic Programming (DP is Covered Before)

◆ The Subset Sum optimization problem says: We have set $S = \{s_0, s_1, \dots, s_{n-1}\}$ of n positive integers and a non-negative integer k . Find a subset T of S so that the sum of the s_r in T is k .

$$\sum_{s_r \in T} s_r = k.$$

Note. A solution to the optimization problem gives a solution to the decision problem

Main Observation

- ◆ Given an instance of SubsetSum: $S = \{s_0, s_1, \dots, s_{n-1}\}$ and k . If T is a solution, then s_{n-1} is either in T or not.
- ◆ If s_{n-1} not in T , then T is a solution for $\{s_0, s_1, \dots, s_{n-2}\}, k$.
- ◆ If s_{n-1} is in T , then $T - \{s_{n-1}\}$ is a solution for $\{s_0, s_1, \dots, s_{n-2}\}, k - s_{n-1}$.

Applying Dynamic Programming to Solve SubsetSum

There are only $(k+1) * n$ problems to solve, namely:

For $0 \leq i \leq n-1, 0 \leq j \leq k$,
find a subset $T \subseteq \{s_0, s_1, \dots, s_i\}$ so that
 $\sum_{s_r \in T} s_r = k$.

Build a solution for bigger values of i and j using stored solutions for smaller values of i and j .

The Goal

Obtain a 2-dimensional array (a matrix) A so that

$$A[i, j] = \begin{cases} T & \text{where } T \subseteq \{s_0, s_1, \dots, s_i\}, \sum_{s_r \in T} s_r = j \\ \text{NULL} & \text{if such a } T \text{ does not exist} \end{cases}$$

- ◆ If S contains values $> k$, we ignore them since they don't contribute to the solution (computations for which j is too big are skipped – see the implementation in code)
- ◆ Fill row $i = 0$ first, then fill later rows based on values of earlier rows.
- ◆ Each cell requires $O(1)$ time, so computation of $A[n-1, k]$ requires $O(kn)$ time.

Details

Row 0:

$$\begin{aligned} A[0, 0] &= \emptyset \quad \text{and} \quad A[0, s_0] = \{s_0\} \\ A[0, e] &= \text{NULL whenever } e \neq 0 \text{ and } e \neq s_0 \end{aligned}$$

Note: $\sum_{s_r \in \emptyset} s_r = 0$ and $\sum_{s_r \in \{s_0\}} s_r = s_0$.

Row i :

$$A[i, j] = \begin{cases} T = A[i-1, j] & \text{if } \sum_{s_r \in T} s_r = j \quad \text{if } s_i \text{ Not in } j \\ T = A[i-1, j - s_i] \cup \{s_i\} & \text{if } \sum_{s_r \in T} s_r = j \quad \text{if } s_i \text{ in } j \end{cases}$$

Note: In computation of $A[i, j]$, a value of NULL in both $A[i-1, j]$ and $A[i-1, j - s_i]$ means that $A[i, j] = \text{NULL}$.

Pseudo-polynomial time

The dynamic programming solution to SubsetSum runs in $O(kn)$. However, k may be much bigger than n , and even if k is $\Theta(n)$, the true running time is based on the number of bits in k , not on the value of k . So even this algorithm runs in exponential time.

Options: Sometimes Exponential Time is Good Enough

Sometimes the context in which an exponential algorithm is to be used does not require very large inputs. In such cases, exponential algorithms don't present difficulties.

Options: Approximation Algorithms

Use an approximation algorithm. There are many examples of NP-hard problems that have been “approximately solved” using a very fast algorithm. Sometimes an approximate solution is good enough.

An Approximation Algorithm for VertexCover

- ◆ An approximation algorithm outputs values that are within a specified tolerance of the correct outputs, but typically runs much faster than an algorithm that can produce completely correct outputs.
- ◆ The idea behind VertexCoverApprox is this: Loop through all the edges of G . For each edge $e = (u, v)$, include u, v in the cover and then delete all edges that are incident to u or v .

(continued)

Algorithm VertexCover Approx(G)

Input: A graph G

Output: A small vertex cover C for G

$C \leftarrow$ new Set

while G still has edges **do**

 select an edge $e = (v, w)$ of G

 add vertices v and w to C

for each edge f incident to v or w **do**

 remove f from G

return C

(continued)

- ◆ *C is a vertex cover.* Notice that the empty set covers any isolated vertex. Every edge was either *used* or *discarded*. Suppose $e = (v, w)$ was a used edge. Then both v and w are in C . Suppose $e = (v, w)$ was a discarded edge. Then one of v and w is in C , by the criterion for discarding.
- ◆ *C is at worst twice the size of an optimal vertex cover.* Let r be the number of edges that are *used* in VertexCoverApprox. The vertex cover C obtained from the algorithm will therefore have size $2r$. At least one endpoint of each of these r edges must occur in a minimal vertex cover, and no endpoint is ever shared between two such edges. Therefore, a minimal vertex cover U must have at least r vertices:
$$r \leq |U| \leq 2r = |C|$$
- ◆ Running time of VertexCoverApprox is clearly $O(m^2)$; with attention to implementation details, this can be improved to $O(m+n)$.

Options: Probabilistic Algorithms

- ◆ *Use a probabilistic (or “randomized”) algorithm.* Sometimes, randomization can be used in the construction of an algorithm to guarantee correct outputs with high probability, and also to guarantee fast running times. Example: The IsPrime problem again.

Solovay-Strassen Solution to IsPrime

Fact: There is a function f , which runs in $O(\log n)$ (that is, $O(\text{length}(n))$), such that for any odd positive integer n and any a chosen randomly in $[1, n-1]$, if $f(a,n) = 1$, then n is composite, but if $f(a,n) = 0$, n is “probably” prime, but is in fact composite with probability $< 1/2$.

The function f with range $\{0,1\}$ is defined by:

$$f(a,n) = 0 \text{ iff } a^{\frac{n-1}{2}} \equiv \left(\frac{a}{n}\right) \pmod{n} \text{ and } \left(\frac{a}{n}\right) \neq 0$$

When $f(a,n) = 1$, n must be composite. When $f(a,n) = 0$, n is “probably prime” but is composite with probability $< 1/2$.

(continued)

The **Solovay-Strassen** algorithm for determining whether an input natural number is prime is the following:

To conclude n is prime with probability $> 1 - (1/2)^k$,

- Perform **Single-Round IsPrime** k times and store outputs in a list L
- If at least one value in L is FALSE, return FALSE.
- Otherwise (if all values in L are TRUE), return TRUE

Algorithm **Single-Round IsPrime**:

Input: A positive integer n

Ouput: TRUE if n is probably prime, FALSE if n is composite

if $n = 2$ **return** TRUE

if $n \% 2 = 0$ **return** FALSE

$a \leftarrow$ a random number in $[1, n-1]$

if $f(a,n) = 1$

return FALSE

return TRUE

Options: Use hardness as an advantage

- ◆ Cryptosystems sometimes base their encryption algorithm on an NP-hard problem. When successful, a hacker would have to, in essence, solve the NP-hard problem in order to crack a code.
- ◆ Example: Merkle-Hellman attempted to base a cryptosystem on the (hardness of the) Knapsack problem. However, this cryptosystem was eventually hacked.
- ◆ These days, the problem of factoring large numbers is used as the hard problem hackers have to solve to crack cryptosystems.

Optional: Public-Key Cryptography

- ◆ Alice wants to send message M to Bob and wants assurance that
 - No one else will be able to view M (confidentiality)
 - The message Bob gets is the one sent by Alice (authorship)
- ◆ Idea:
 - Each person has a public key P and a secret key S .
 - P is available to anyone but only the owner of S knows S
 - P and S are inverses: $P(S(M)) = M = S(P(M))$

(cont)

◆ Usage:

- Alice can send a message to Bob by encrypting with Bob's public key: send $P_B(M)$. Then Bob can decrypt with his secret key $S_B(P_B(M)) = M$. This guarantees confidentiality but not authorship
- To ensure authorship, Alice can also digitally sign $P_B(M)$ using her secret key: $S_A(P_B(M))$. Then Bob can decrypt using P_A followed by S_B : $M = S_B(P_A(S_A(P_B(M))))$. If the result is "garbage" it means that the author is not Alice. Both confidentiality and authorship are ensured (though in practice more efficient protocols are followed)

RSA Cryptosystem

The Requirements

- ◆ $P(S(M)) = M$
- ◆ Computation of $P(M)$ and $S(M)$ is efficient (when P and S are known)
- ◆ Infeasible to compute S from P (can therefore make P public)
- ◆ $S(P(M)) = M$ (S , which is private, serves to *digitally sign* the message)

Making It Infeasible to Derive D from E

- ◆ Arrange it so that computation of S from P is an NP-complete (or at least *hard*) problem.
- ◆ Merkle-Hellman tried to do this using hardness of Knapsack problem, but relevant instances of Knapsack were not hard to solve
- ◆ RSA ties difficulty of factoring large numbers to problem of computing S from P.

RSA Setup

1. Pick at random two several hundred bit (~ 512 these days) primes p , q , and let $n = pq$.
2. Pick a small prime e relatively prime to $\phi(n) = (p-1)(q-1)$. Often $e = 3, 7, 65,537$ are used.
3. Compute d as modular inverse of $e \bmod \phi(n)$; that is, $ed \equiv 1 \bmod \phi(n)$
4. Public key is (e,n) . Private key is (d,n) .
5. Encryption of message M :
$$P(M) = M^e \bmod n$$
6. Decryption of message M :
$$S(M) = M^d \bmod n$$

Things To Verify

1. Encryption $P(M)$ and decryption $S(M)$ are fast when P and S are known [fast modular exponentiation]
2. $P(S(M)) = S(P(M)) = M$ [Euler's Theorem]
3. Setup portion is reasonably fast
 - ◆ Obtain two large primes p, q [probable primes]
 - ◆ Verify e is relatively prime to $\phi(n)$ [gcd algorithm]
 - ◆ Compute d from e given $\phi(n)$ [variation of gcd algorithm]
4. Argue that computing d from e without knowledge of $\phi(n)$ is hard. [If only e and n are known, need to find factors p, q of n . Best known algorithm is exponential. For 512 bit primes, infeasible.]

Complete RSA Example

◆ Setup:

- $p = 5, q = 11$
- $n = 5 \cdot 11 = 55$
- $\phi(n) = 4 \cdot 10 = 40$
- $e = 3$
- $d = 27$ ($3 \cdot 27 = 81 = 2 \cdot 40 + 1$)

◆ Encryption

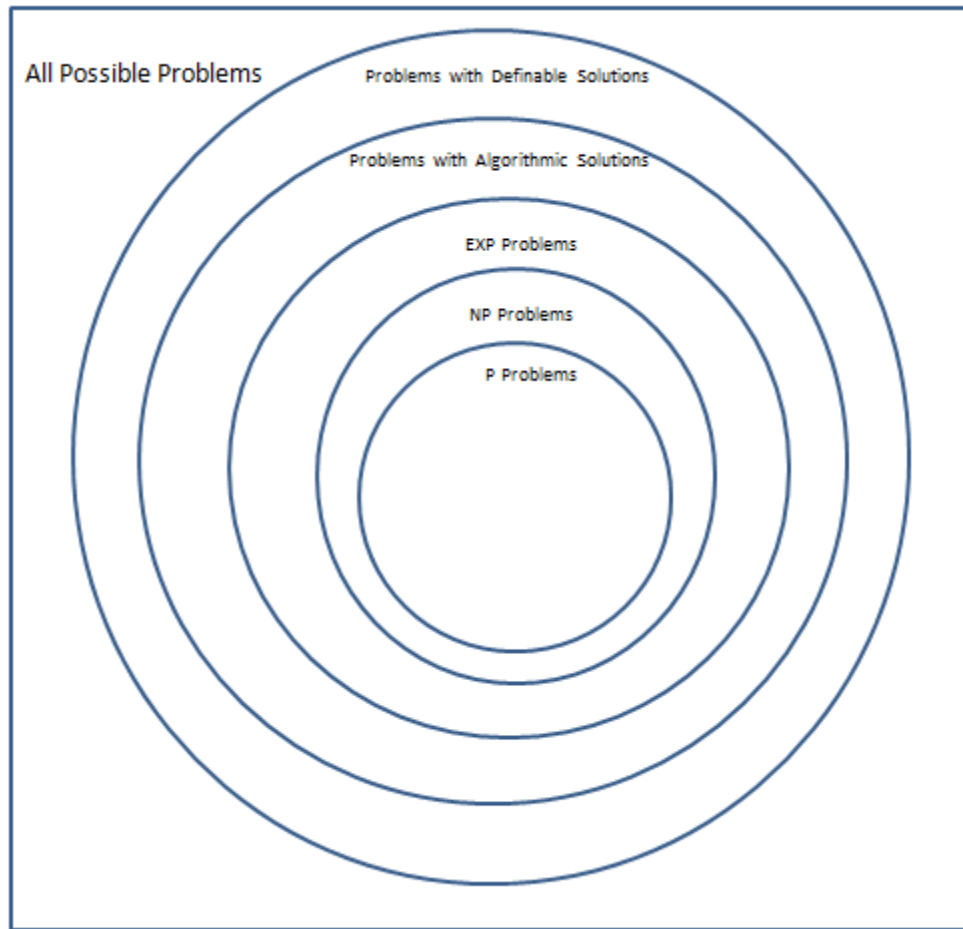
- $C = M^3 \bmod 55$

◆ Decryption

- $M = C^{27} \bmod 55$

<i>M</i>	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
<i>C</i>	1	8	27	9	15	51	13	17	14	10	11	23	52	49	20	26	18	2
<i>M</i>	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36
<i>C</i>	39	25	21	33	12	19	5	31	48	7	24	50	36	43	22	34	30	16
<i>M</i>	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54
<i>C</i>	53	37	29	35	6	3	32	44	45	41	38	42	4	40	46	28	47	54

Classes of Problems (Another Look)



Connecting The Parts of Knowledge With The Wholeness of Knowledge

1. There are many natural decision problems in Computer Science for which feasible solutions are needed, but which are NP-complete. Therefore, there is little hope of finding such solutions.
2. The hardness of certain NP-complete problems is being used to ensure the security of certain cryptographic systems.

3. *Transcendental Consciousness* is a field of all possibilities and infinite creativity.

4. *Impulses Within the Transcendental Field.*

Pure consciousness as it prepares to manifest is a “wide angle lens” making use of every possibility for creative ends

5. *Wholeness Moving Within Itself.* In Unity Consciousness, awareness does not get stuck in problems; problems are seen as steps of progress in the unfoldment of the dynamics of consciousness.