

A large, abstract network graph occupies the left two-thirds of the slide. It consists of numerous small, semi-transparent circular nodes of varying sizes scattered across a dark purple-to-orange gradient background. A dense web of thin, light-colored lines connects these nodes, forming a complex web-like structure.

Foundations of Algorithm

SCS1308

Dr. Dinuni Fernando
PhD
Senior Lecturer



The theory of NP-completeness

- Tractable and intractable problems able to solve in polynomial time or not
- NP-complete problems able to verify in polynomial time

Classifying problems

- Classifying problems as tractable or intractable.
- Problem is *tractable* if there **exists at least one** polynomial bound algorithm that solves it
- An algorithm is *polynomial bound* if its worst case time complexity is bounded by a polynomial $p(n)$ in the size n of the problem

$$p(n) = a_n n^k + \dots + a_1 n + a_0 \text{ where } k \text{ is a constant}$$

Intractability

- Dictionary definition : Difficulty to treat or work
- Problem in computer science is intractable if a computer has difficulty solving it. [Vague]

Polynomial-time algorithm

- A polynomial-time algorithm is one whose worst-case time complexity is bounded above by a polynomial function of its input size. That is, if n is the input size, there exists a polynomial $p(n)$ such that

$$W(n) \in O(p(n))$$

Intractability

- Algorithms with the following worst-case time complexities are all polynomial-time.

$$2n$$

$$3n^3 + 4n$$

$$5n + n^{10}$$

$$nlgn$$

- Algorithms with the following worst-case time complexities are not polynomial-time.

$$-2^n$$

$$2^{0.01n}$$

$$2^{\sqrt{n}}$$

$$n!$$

Intractability

$$nlgn < n^2$$

$nlgn$ is not polynomial in n .

- It is bounded by a polynomial in n , which means that an algorithm with this time complexity satisfies the criterion to be called a polynomial-time algorithm.
- In computer science, a problem is called intractable if it is impossible to solve it with a polynomial-time algorithm. We stress that intractability is a property of a problem; it is not a property of any one algorithm for that problem.
- For a problem to be intractable, there must be no polynomial-time algorithm that solves it. Obtaining a non-polynomial-time algorithm for a problem does not make it intractable.

Example

2^n vs n^{10}

- We can create extreme examples in which a non-polynomial-time algorithm is better than a polynomial-time algorithm for practical input sizes.
- For example, if $n = 1,000,000$.

$$2^{0.00001} = 1024 \text{ whereas } n^{10} = (10^6)^{10} = 10^{60}$$

Problems in Intractability

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

Problems for which polynomial-time algorithms have been found

- Any problem for which we found a polynomial-time algorithm falls in this first category.
- $\Theta(n \lg n)$ algorithms for sorting
- a $\Theta(\lg n)$ algorithm for searching a sorted array
- a $\Theta(n^{2.38})$ algorithm for matrix multiplication
- a $\Theta(n^3)$ algorithm for chained matrix multiplication, and so on.
- Because n is a measure of the amounts of data in the inputs to these algorithms, they are all polynomial-time.

Problems that have been proven to be intractable

- Two types of problems here.
- **Type one problem** - require a non-polynomial amount of output.
 - The problem of determining all Hamiltonian Circuits. If there was an edge from every vertex to every other vertex, there would be $(n - 1)!$ such circuits. To solve the problem, an algorithm would have to output all of these circuits,
 - Determining all possible circuits are nonpolynomial amount of output.

Problems that have been proven to be intractable

- **Type two problem** : We can prove that the problem cannot be solved in polynomial time.
- Found relatively few such problems.
 - The first ones were undecidable problems (it can be proven that algorithms that solve them cannot exist)
 - Eg: The most well-known of these is the Halting problem. In this problem we take as input any algorithm and any input to that algorithm and decide whether or not the algorithm will halt when applied to that input.
 - The output for a decision problem is a simple “yes” or “no” answer. Therefore, the amount of output requested is certainly reasonable.
 - Eg: Presburger Arithmetic, which was proven intractable by Fischer and Rabin in 1974.

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

- This category includes any problem for which a polynomial-time algorithm has never been found, but yet no one has ever proven that such an algorithm is not possible.
- Eg: 0-1 Knapsack problem, the Traveling Salesperson problem, the Sum-of-Subsets problem, the m -Coloring problem for $m \geq 3$, the Hamiltonian Circuits problem, and the problem of abductive inference in a Bayesian network all fall into this category.
- There is a polynomial in n that bounds the number of times the basic operation is done when the instances are taken from some restricted subset. However, no such polynomial exists for the set of all instances. To show this, we need only find some infinite sequence of instances for which no polynomial in n bounds the number of times the basic operation is done.

Intractable problems

- Problem is *intractable* if it is not tractable.
- **1st Category:** All algorithms that solve the problem are not in polynomial bound.
- It has a worst case growth rate $f(n)$ which cannot be bound by a polynomial $p(n)$ in the size n of the problem.
- For intractable problems the bounds are:

$$f(n) = c^n, \text{ or } n^{\log n}, \text{ etc.}$$

Another set of intractable problems

- **2nd category:** Undecidable problems
 - Cannot give a “yes” or “no” answer
 - E.g., **Halting problem**
 - *No algorithm can be devised to solve the halting problem*

Halting problem

- Input: A string P and a string I . Consider P as a program and I as input to P .
- Output: 1 if P halts on I ; 0 if P does not halt on I (infinite loop)
- **Theorem (Turing 1940): There is no program to solve the halting problem.** See next slide for proof.

Proof: Halting problem is undecidable

- Proof: To reach a contradiction, assume that there exists a program $\text{Halt}(P, I)$ that solves the halting problem. $\text{Halt}(P, I)$ returns true if and only if P halts on I . Otherwise, it returns false. Using $\text{Halt}(P, I)$, we construct the following program Z :

```
program (string x)
```

```
begin
```

```
  If Halt (x, x) then
```

Program x halts when given itself as input : returns true

```
    while(1) printf ("ha ha ha ");
```

Enters an infinite loop : contradicts the claim program
doesn't halt.

```
  Else exit(0)      returns false
```

Terminates immediately. Contradicting the claim that program
does not halt.

```
end
```

Halting problem

```
program (string x)
begin
    If Halt (x, x) then
        while(1) printf ("ha ha ha ");
    Else exit(0)
end
```

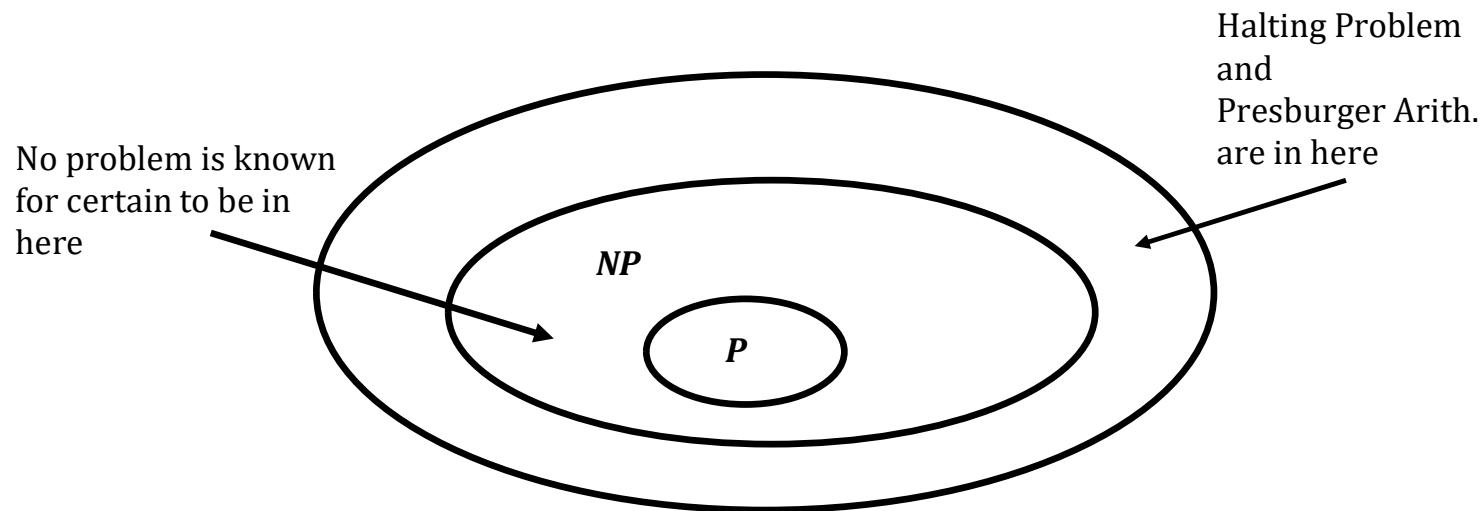
- Case 1: Program Z halts on input Z. By the correctness of Halt, Halt(Z, Z) returns true. Thus, program Z loops forever on input Z, printing “ha ha ha” **Contradiction**.
- Case 2: Program Z does not halt on input Z. Halt(Z, Z) returns false. Hence, program Z halts. **Contradiction**.

Why is this classification useful?

- If problem is intractable, no point in trying to find an *efficient* algorithm that solves the problem with polynomial time complexity in the worst case
- All algorithms will be too slow for **large inputs.**

Intractable problems

- Turing showed some problems are so hard that no algorithm can solve them (undecidable).
- Other researchers showed some decidable problems from automata, mathematical logic, etc. are intractable: Presburger arithmetic is doubly exponential.

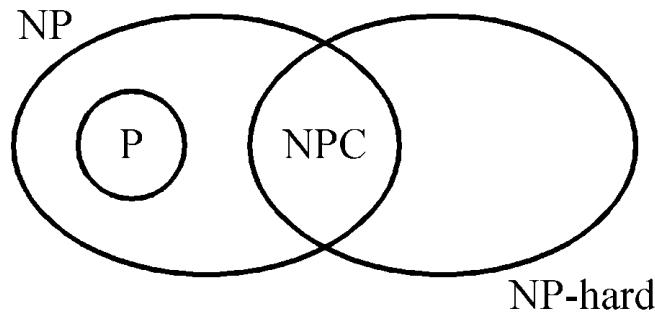


Problems Proven to be Intractable

- All Hamiltonian circuits: For a complete undirected graph, there are $(n-1)!$ Circuits
- Halting problem: Undecidable
- Presburger Arithmetic
- ...

Problems not proven to be intractable but no polynomial time algorithm

- 0-1 knapsack
- Traveling salesperson
- Sum of subsets
- M-coloring for $m \geq 3$
- ...



- **NP** is the class of decision problems (yes/no problems) for which a solution can be verified in polynomial time by a deterministic Turing machine.
- **P**: the class of problems which can be solved by a deterministic **polynomial** algorithm.
- **NPC** - are hardest problems in NP
- A problem X is NP-complete if:
 - X is in NP.
 - Every problem in NP can be reduced to X in polynomial time.
- **NP Hard** : problems are at least as hard as the hardest problems in NP but not necessarily belong to NP.

Coping with NP-Complete/NP-Hard Problems

- Rely on approximation algorithms, heuristics, etc.
- Sometimes we need to solve only a restricted version of the problem.
- If the restricted problem is tractable, design an algorithm for the restricted version

Nondeterministic algorithms

- A non-deterministic algorithm consists of
 - phase 1: guessing
 - phase 2: checking
- If the checking stage of a nondeterministic algorithm is of polynomial time-complexity, then this algorithm is called an NP (nondeterministic polynomial) algorithm.
- NP problems: (must be decision problems)
 - e.g. searching, MST, sorting
satisfiability problem (SAT)
traveling salesperson problem (TSP)

Nondeterministic operations and functions

- Choice(S) : arbitrarily chooses one of the elements in set S
- Failure : an unsuccessful completion
- Success : a successful completion
- Nondeterministic searching algorithm:

```
j ← choice(1 : n) /* guessing */  
if A(j) = x then success /* checking */  
else failure
```

- A nondeterministic algorithm terminates unsuccessfully iff there exist no set of choices leading to a success signal.
- The time required for $\text{choice}(l : n)$ is $O(1)$

Hard practical problems

- There are many practical problems for which *no one has yet* found a polynomial bound algorithm.
- Examples: 3-SAT, traveling salesperson, 0/1 knapsack, sum of subsets, graph coloring, bin packing etc.
- Most design automation problems such as testing and routing.
- Many OS, networks, database and graph problems.

Traveling Salesperson Problem

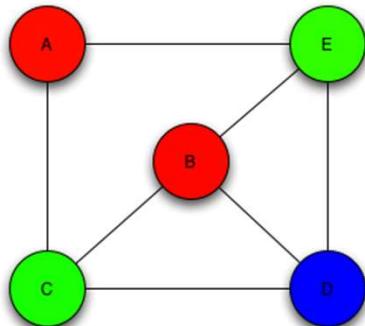
- Let a weighted, directed graph be given.
- A tour in such a graph is a path that starts at one vertex, ends at that vertex, and visits all the other vertices in the graph exactly once, and that the Traveling Salesperson Optimization problem is to determine a tour with minimal total weight on its edges.
- The Traveling Salesperson Decision problem is to determine for a given positive number d whether there is a tour having total weight no greater than d .

0-1 Knapsack problem

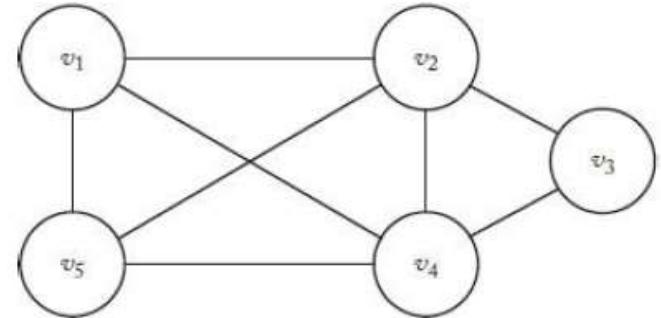
- **0-1 Knapsack Optimization problem** is to determine the maximum total profit of the items that can be placed in a knapsack given that each item has a weight and a profit, and that there is a maximum total weight W that can be carried in the sack.
- The **0-1 Knapsack Decision problem** is to determine, for a given profit P , whether it is possible to load the knapsack so as to keep the total weight no greater than W , while making the total profit at least equal to P .

Graph-Coloring Problem

- The Graph-Coloring Optimization problem is to determine the minimum number of colors needed to color a graph so that no two adjacent vertices are colored the same color.
- That number is called the chromatic number of the graph.
- The Graph-Coloring Decision problem is to determine, for an integer m , whether there is a coloring that uses at most m colors and that colors no two adjacent vertices the same color.



Clique Problem



- A clique in an undirected graph $G = (V, E)$ is a subset W of V such that each vertex in W is adjacent to all the other vertices in W .
- $\{v_2, v_3, v_4\}$ is a clique, whereas $\{v_1, v_2, v_3\}$ is not a clique because v_1 is not adjacent to v_3 . A maximal clique is a clique of maximal size. The only maximal clique in the graph in Figure is $\{v_1, v_2, v_4, v_5\}$.
- The Clique Optimization problem is to determine the size of a maximal clique for a given graph.
- The Clique Decision problem is to determine, for a positive integer k , whether there is a clique containing at least k vertices.

Traveling Salesperson Decision problem

- Suppose, for some instance, answer was “yes”
 - For some graph and number d, a tour existed in which weight was no greater than d.
 - We could write the algorithm that follows to verify whether what they produced was a tour with weight no greater than d.

```
Bool verify ( weighted digraph G, number d, claimed_tour  
S)  
{  
If(S is a tour && the weight of the edge in S is <= d)  
    return true;  
else  
    return false;  
}
```

- first checks to see whether S is indeed a tour. If it is, the algorithm then adds the weights on the tour.
- If the sum of the weights is no greater than d, it returns “true.”
- If weights exceeds d algorithm returns “false”.

Non-deterministic algorithm

1. Guessing (Nondeterministic) Stage: Given an instance of a problem, for some string S . The string can be thought of as a guess at a solution to the instance.
2. Verification (Deterministic) Stage: The instance and the string S are the input to this stage. This stage then proceeds in an ordinary deterministic manner either
 - (1) eventually halting with an output of “true,” which means that it has been verified that the answer for this instance is “yes,”
 - (2) halting with an output of “false,” or
 - (3) not halting at all (that is, going into an infinite loop). In these latter two cases, it has not been verified that the answer for this instance is “yes.”

Polynomial-time nondeterministic algorithm

- A polynomial-time nondeterministic algorithm is a nondeterministic algorithm whose verification stage is a polynomial-time algorithm.
- So what's set of NP
- NP is the set of all decision problems that can be solved by polynomial-time nondeterministic algorithms.

Satisfiability (SAT) problem

- CNF-Satisfiability Decision problem is to determine, for a given logical expression in CNF, whether there is some truth assignment that makes the expression true.

Conjunctive Normal Form (CNF)

- A **literal** is a variable or the negation of a var.
 - Example: The variable x is a literal, and its negation, $\neg x$, is a literal.
- A **clause** is a disjunction (an OR) of literals.
 - Example: $(x \vee y \vee \neg z)$ is a clause
- A formula is in **Conjunctive Normal Form (CNF)** if it is a conjunction (an AND) of clauses.
 - Example: $(x \vee \neg z) \wedge (y \vee z)$ is in CNF.
- A CNF formula is a conjunction of disjunctions, i.e., a product (AND) of sums (OR)

Example

- What is the answer to CNF –satisfiability ?

A. $(x_1 \vee x_2) \wedge (x_2 \vee \neg x_3) \wedge \neg x_2$

- $X_1 = \text{true}$
- $X_2 = \text{false}$
- $X_3 = \text{false}$

How about B. $(x_1 \vee x_2) \wedge \neg x_1 \wedge \neg x_2$

Definition: A CNF formula is a **3CNF-formula** iff each clause has exactly 3 literals.

$$(x_1 \vee \neg x_2 \vee x_3) \wedge (x_4 \vee x_2 \vee x_5) \wedge \dots \wedge (x_3 \vee \neg x_2 \vee \neg x_1)$$

The diagram shows a CNF formula consisting of a conjunction of several clauses. Each clause is a disjunction of literals. The formula is structured as follows:

$$(x_1 \vee \neg x_2 \vee x_3) \wedge (x_4 \vee x_2 \vee x_5) \wedge \dots \wedge (x_3 \vee \neg x_2 \vee \neg x_1)$$

A blue bracket is placed under the first two clauses, with the word "clauses" written below it in blue. The individual clauses are separated by orange AND operators (\wedge), and the entire formula is a single black AND operator (\wedge) connecting the clauses.

- A literal is a variable or the negation of a var.
- A clause is a disjunction (an OR) of literals.
- A formula is in Conjunctive Normal Form (CNF) if it is a conjunction (an AND) of clauses.
- A CNF formula is a conjunction of disjunctions of literals.

Definition: A CNF formula is a **3CNF-formula** iff each clause has exactly 3 literals.

$$\phi = (x_1 \vee \neg x_2 \vee x_3) \wedge (x_4 \vee x_2 \vee x_5) \wedge \dots \wedge (x_3 \vee \neg x_2 \vee \neg x_1)$$



YES $(x_1 \vee \neg x_2 \vee x_1)$

NO $(x_3 \vee x_1) \wedge (x_3 \vee \neg x_2 \vee \neg x_1)$

NO $(x_1 \vee x_2 \vee x_3) \wedge (\neg x_4 \vee x_2 \vee x_1) \vee (x_3 \vee x_1 \vee \neg x_1)$

NO $(x_1 \vee \neg x_2 \vee x_3) \wedge (x_3 \wedge \neg x_2 \wedge \neg x_1)$

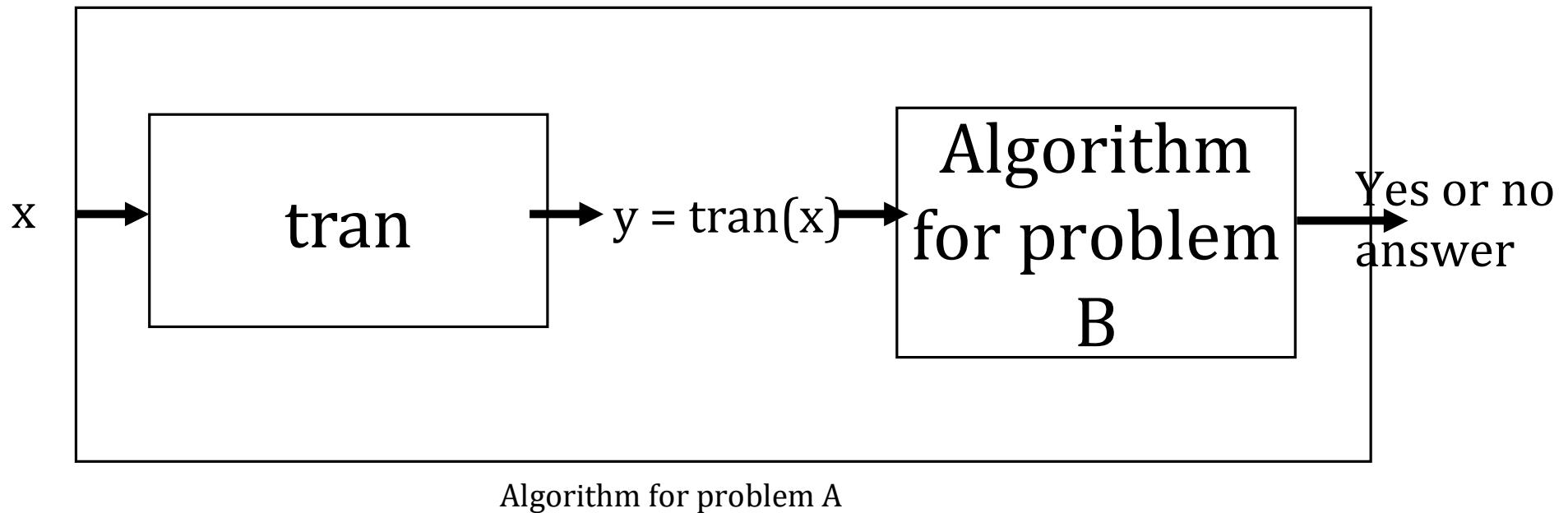
3SAT = { ϕ | ϕ is a satisfiable 3cnf-formula }

Boolean Basics: Literals, Clauses, CNF

- Boolean function on n variables is a mapping $\{0,1\}^n \rightarrow \{0,1\}$
- **Literal** = Boolean variable or its negation
- **Clause** = disjunction of literals (no complementary pair)
- **Conjunctive Normal Form (CNF)** = conjunction of clauses,
i.e., product-of-sums (Fact: Every Boolean function has a
CNF representation)

- It is easy to write a polynomial-time algorithm that takes as input a logical expression in CNF and a set of truth assignments to the variables and verifies whether the expression is true for that assignment.
- Therefore, the problem is in NP.
- No one has ever found a polynomial-time algorithm for this problem, and no one has ever proven that it cannot be solved in polynomial time.
- So, we do not know if it is in P.
- In 1971, Stephen Cook published a paper proving that if CNF-Satisfiability is in P, Then P = NP

Polynomial time Reducibility



- Algorithm tran is a transformation algorithm that maps each instance x of decision problem A to an instance y of decision problem B. Together with the algorithm for decision problem B, it yields an algorithm for decision problem A.

Polynomial time Reducibility

Transformation algorithm

- Suppose we want to solve decision problem A
- We have an algorithm that solves decision problem B.
- we can write an algorithm that creates an instance y of problem B from every instance x of problem A such that an algorithm for problem B answers “yes” for y if and only if the answer to problem A is “yes” for x.
- *A function that maps every instance of problem A to an instance of problem B*

How are they handled?

- A variety of algorithms based on backtracking, branch and bound, dynamic programming, etc.
- None can be shown to be polynomial bound (exponential in the worst case)

Cook's theorem

- SAT is NP-complete
- 3-SAT is NP-complete (1-SAT or 2-SAT is P)
- It is the first NP-complete problem
- Every NP problem reduces to SAT
- NP = P iff the SAT problem is a P problem

NP Complete – Definition

- A problem B is called NP-complete if both of the following are true:
 1. B is in NP.
 2. For every other problem A in NP,

$$A \propto B$$

Theory of NP completeness

- The theory of NP-completeness enables showing that these problems are at least as hard as *NP-complete* problems
- Practical implication of knowing a problem is NP-complete is that it is **probably** intractable (whether it is or not has not been proved yet)
- So any algorithm that solves it will probably be very slow for large inputs

We need to discuss

- Decision problems
- Converting optimization problems into decision problems
- The relationship between an optimization problem and its decision version
- The class P
- Verification algorithms
- The class NP
- The concept of polynomial transformations
- The class of NP-complete problems

Satisfiability (SAT) problem

- Is there a truth assignment to the n variables of a logical expression in CNF which makes the value of the expression true?
- The answer is yes, if all clauses evaluate to true
- Otherwise, the answer is “no”