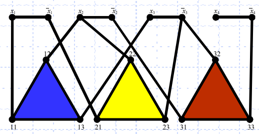
# Lecture 16:
# NP-Complete





## Goals of today's lecture

- Review NP and problem reduction
- Define classes NP-Hard and NP-Complete
- Describe why NPH and NPC are important concepts for computer scientists
- Explain why "P=NP?" is still an open question
- Describe a few approximation algorithms (tomorrow)



## Relationship between NP and Nondeterministic Algorithms

- Nondeterministic Algorithms have two phases
  - Write a guess
  - Check the guess
- The number of steps is the sum of the steps in the two phases
  - If both steps take polynomial time, then the problem is said to be a member of NP
- All problems become a search for a solution that verifies a yes answer!
  - We don't know how many times this process will have to be repeated before a solution is generated and verified
  - May need to repeat it exponential or factorial number of times (unless the problem is a member of class P since members of P can generate a definitive guess in polynomial time)



## Quiz

- Prove that Subset Sum is a member of NP:

  **Subset Sum**: Given a triple (S, *max*, *min*), where S is a set of positive integers and *max* and *min* are positive integers. Is there a subset T of S such that the sum of the integers in subset T is at most *max* and at least *min*?

**Step 1**: Randomly pick elements from S and put them in T
**Step 2**: Compute the sum of the elements in T
if the sum is between min and max then
return yes
else return no

Since Steps 1 & 2 run in O(n) time, Subset Sum is a member of NP



## Wholeness Statement

Complexity classes show the relationship between problems on the basis of their relative difficulty. Problems in the class P are considered "easy" (tractable) whereas problems in class NP-complete (NPC) are considered "hard" (intractable); there are several thousand problems in NPC. One of the attractions of Maharishi's programs is that they are easy, can be practiced by anyone, and are demonstrated to be powerful in their positive benefits to individual and society.



## Outline and Reading

- NP-completeness (§13.2)
  - Definition of NP-hard and NP-complete
  - The Cook-Levin Theorem

## Why should we care whether a problem is NP-hard or NP-complete?

- My claim: With this knowledge, we may now be able to better deal with problems that seem hard?
- For example,
  - What should we do if our boss asks us to implement something that seems like it will take a long time to compute and we can't seem to come up with an efficient (polynomial-time) algorithm?
  - What kinds of problems might fall into this category?
  - What should we do?



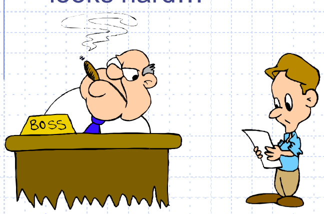
## Exercise/thought experiment:

- Suppose your professor (or an interviewer) asks you to solve the following problem:
  - Given a graph G=(V, E), design an algorithm to find the longest simple path between two vertices *u* and *v*.
- We'll come back to this problem later in today's lecture
- Which of the following would be your response?



## An Approach When Dealing with Hard Problems

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

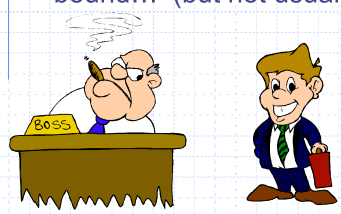



I couldn't find a polynomial-time algorithm;
I guess I'm not creative enough.



## An Approach When Dealing with Hard Problems

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

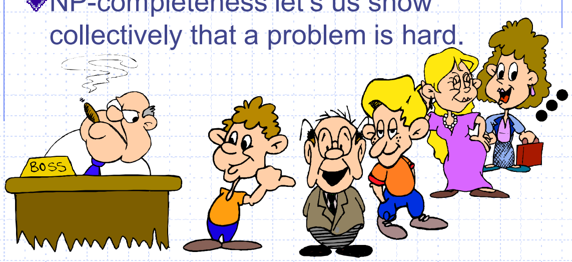



I couldn't find a polynomial-time algorithm,
because no such algorithm exists!



## An Approach When 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.



## Tractable vs. Intractable for Deterministic Algorithms

- *Tractable* problems:
  - Standard working definition: *polynomial time*
    - On an input of size $n$ the worst-case running time is $O(n^k)$ for some constant $k$
    - Polynomial time: $O(n^2)$, $O(n^3)$, $O(1)$, $O(n \log n)$
- *Intractable* problems:
  - As the input grows large, finding a solution takes an unreasonable amount of time
  - Not in polynomial time: $O(2^n)$, $O(4^n)$, $O(n^n)$, $O(n!)$
- *Noncomputable* problems:
  - There can be no algorithm for finding a solution

## Tractable vs. Intractable for non-deterministic algorithms

- All problems (solvable and unsolvable) are simplified to a corresponding decision problem
- The problem then becomes a decision about whether or not a guess is a valid solution
  - **Tractable (feasible) problems**:
    - a valid guess can be deterministically generated and checked in polynomial time, i.e., the problems in complexity class P
  - **Intractable (infeasible) problems**:
    - no polynomial time algorithm to deterministically generate a valid guess has **yet** been found (or it may NOT be possible to generate a guess in polynomial time )
    - NP-Complete and NP-Hard problems are considered intractable, but we are not sure
    - Includes problems in NP and others not in NP
  - **Undecidable problems**:
    - noncomputable problems for which there can be no algorithm to validate a guess
    - must be proven mathematically, e.g., the halting problem



## Example Exam Questions

- Is the problem to enumerate the permutations of a set tractable or intractable?
  - Is it a member of NP?
- Is the problem to enumerate the set of all subsets tractable or intractable?
  - Is it a member of NP?
- Is the Halting Problem tractable, intractable, or a member of NP?



## What kinds of problems might take a long time to compute?

- Search problems such as
  - 0-1 Knapsack, Subset-Sum, Traveling SalesPerson (TSP), Hamiltonian Cycle, Circuit-Sat, Scheduling, Register Allocation, Factoring, etc.
- Why do other search problems, such as LCS, have a polynomial-time solution?
  - It is not well understood why!!!!
  - Many search problems seem to require searching the entire search space, which becomes difficult, "like trying to find a needle in a haystack"



## Van Gogh



## Requires thinking "out of the box"

- What if we had a really strong magnet?
  - Then we wouldn't have to search through the whole haystack!
- There is a movie called "Traveling Salesman" about a world where P=NP, i.e., all NPC computing problems are feasible (can be computed quickly)
  - Today we want to understand why if P=NP, then all problems in NP would be quickly/easily solvable
  - And if P≠NP, then NP-complete problems are necessarily going to continue to take a long time to calculate



## Testing for primality

- Is number x a prime number?
- Could search through the numbers less than x for its factors by division
  - If x is represented in binary and |x| = n, then searching takes $O(2^n)$ time
- However, we have a magnet for finding the needle in the haystack (Fermat's Little Theorem)
  - Let 0<a<x. If $a^{x-1}$ mod x = 1, then x is prime (actually prime with high probability, i.e., very few composite numbers have this property and we can narrow it down further by eliminating even numbers)
  - Proven to be in P in 2002 using the AKS Primality Test

## Why non-deterministic decision algorithms?

- Simplifies reduction from one problem to another
  - Since the output from each problem is yes/no
  - So only have to convert instances of one problem into instances of the other
    - (but both instances must give the same yes/no answer)



## Decidable vs. Undecidable

- Some problems solvable in polynomial time
  - Almost all algorithms we've studied provide a polynomial-time solution to some problem
  - **P** is the class of problems solvable in polynomial time
- 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
  - Such problems are clearly intractable, not in **P**



## Main Point

1. Many important problems such as job scheduling, TSP, the 0-1 knapsack problem, K-coloring, and Hamiltonian circuits have no known efficient algorithm (with a polynomial time bound).

   When an individual projects his intention from the state of pure awareness, then the algorithms of natural law compute the fulfilment of those intentions with perfect efficiency.

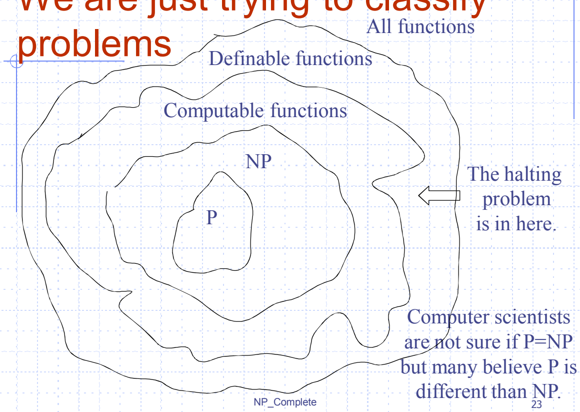


## Example

- What about the problem we did earlier for homework that outputs the permutations of a sequence of elements?
  - Is this problem tractable, intractable, or undecidable?
  - Is this problem a member of NP?



## We are just trying to classify problems






## Biggest Open Question in CS: Is P = NP?

- The *NP-Complete* problems are an important class of problems whose status is unknown
  - No polynomial-time algorithm has been discovered for them
  - No polynomial-time lower bound has been proved for any NP-Complete problem
- We call this the *P = NP question*
  - The biggest open problem in CS

## Problem Reductions

Let $V_B$ be an algorithm that correctly solves/verifies instances of B and let $V_A$ be an algorithm that solves/verifies instances of A.

If there exists an algorithm R that transforms any instance $a \in A$ into an instance $R(a) \in B$,
then R is a valid reduction if and only if $V_B(R(a))=V_A(a)$

- The key is that the transformation (reduction) R must preserve the correctness of the answer to A
- And the transformation must be easy (i.e., take polynomial time)



## Implications of Problem Reductions

- Reducing problem A to problem B means:
  - An algorithm to solve B can be used to solve A as follows:
    - Take input to A and transform it into input to B
    - Use algorithm that solves B to produce the answer for B which is also the answer for the input to A
  - Thus A cannot be harder than B if the transformation takes polynomial time
  - **Typically, instances of A are reduced to a small subset of the instances of B**
- Problems in P can be reduced to any other problem (Why?)



## Example reduction

- Consider the following decision problems:

  **Sorting**: Given a sequence S of elements and a comparator $\leq$. Is there a sequence T containing the same elements as S such that for all ranks r between 0 and n-2, $T_r \leq T_{r+1}$?

  **Subset Sum**: Given a triple (S, *max*, *min*), where S is a set of positive integers and *max* and *min* are positive integers. Is there a subset T of S such that the sum of the integers in T is at most *max* and at least *min*?



## Reduction of Sorting to Subset Sum

The transformation would use the following algorithm:

```
Algorithm reduceSortToSS(S, C)
  Input: a Sequence S of elements and a comparator C for sorting
     elements of S
  Output: a Sequence R of integers and the values of max and min for the
     Subset Sum problem
     R ← new empty Sequence
     R.insertLast( 2 )
     for i ← 0 to S.size()-1 do
         if ¬ C.isComparable(S.elemAtRank(i))
             then return (R, 1, 1)  {integers, max, min}
     return (R, 2, 2)           {integers, max, min}
```



## Main Point

2. A problem is in NP (nondeterministic polynomial) if there is a polynomial time algorithm for checking whether or not a proposed solution (guess) is a correct solution.

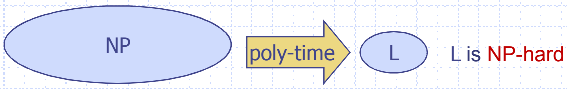
   Natural law always computes all possible paths to the goal and chooses the one with the least action and maximum positive benefit.



## NP-Complete

# NP-Complete

- A language M is polynomial-time **reducible** to a language L if an instance x for M can be transformed in polynomial time to an instance x' for L such that x is in M if and only if x' is in L.
  - Denote this by M→L.
- A problem L is **NP-hard** if every problem in NP can be reduced to L in polynomial time
  - That is, for each problem M in NP, we can take any input x for M, **transform** it in polynomial time to an input x' for L such that x is in M if and only if x' is in L
- L is **NP-complete** if L is in NP and is NP-hard






# NP-Hard Intuition

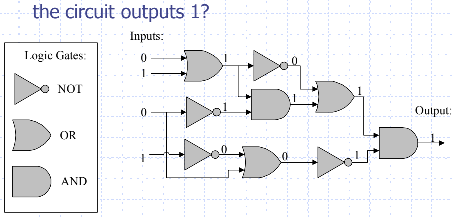
- If any NP-Hard problem can be solved in polynomial time, then all problems in NP can be solved in polynomial time
  - i.e., P=NP
  - Why?
- All problems in NP can be reduced in polynomial-time to the halting problem, so the halting problem is NP-Hard (we'll talk about why later)
  - Does this mean the halting problem is NP-complete?
- The halting problem is **NP-hard** but not **NP-complete** because it's not in NP.



# An Important Problem in NP

- A Boolean circuit is a circuit of AND, OR, and NOT gates
- The CIRCUIT-SAT problem is:

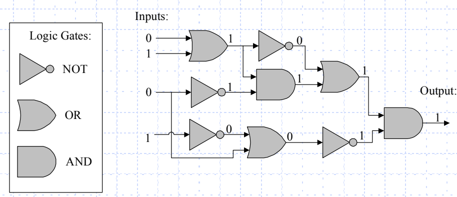
  Given a circuit with a single output node, is there an assignment of 0's and 1's to the circuit's inputs so that the circuit outputs 1?






# CIRCUIT-SAT is in NP

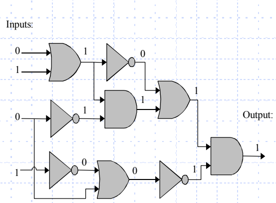
- Non-deterministically choose a set of inputs (flip a coin for each) and compute the outcome of every gate, then test the last gate's output value.






# CIRCUIT-SAT is NP-complete

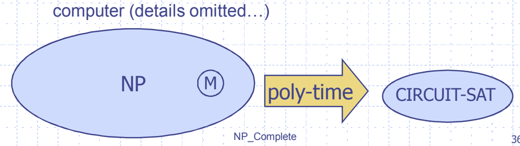
- Must prove:
  - CIRCUIT-SAT is in NP
  - For every problem M ∊ NP, M → CIRCUIT-SAT.





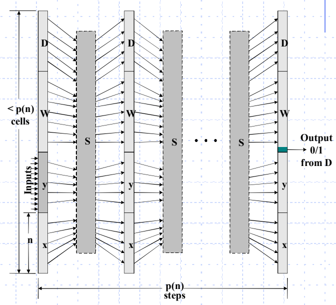

# Cook-Levin Theorem

- CIRCUIT-SAT is NP-complete.
  - We already showed it is in NP.
- To prove it is NP-hard, we have to show that every language (problem) in NP can be reduced to it.
  - Let M be in NP, and let x be an input for M.
  - Let y be a certificate that allows us to verify membership in M in polynomial time, p(n), by some algorithm D.
  - Let S be a circuit of size at most $O(p(n)^2)$ that simulates a computer (details omitted…)



## Cook-Levin Proof

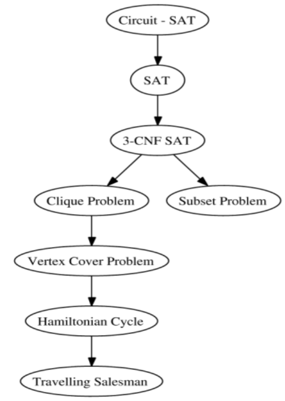
- We can build a circuit that simulates the verification of x's membership in M using y.
  - Let W be the working storage for D (including registers, such as program counter); let D be given in RAM "machine code."
  - Simulate p(n) steps of D by replicating circuit S for each step of D. Only input: y.
  - Circuit is satisfiable if and only if x is accepted by D with some certificate y
  - Total size is still polynomial: $O(p(n)^3)$.






## NP-Complete

- Reducibility of some NP-complete problems
- Therefore, these decision problems are NP-hard (NP-complete since …)
- Decision problems are reducible, but not the optimization problem
- All NP-complete problems are reducible to each other, by definition






## Proving NP-Completeness

- *What steps do we have to take to prove a problem Q is NP-Complete?*
  - Pick a known NP-Complete problem A
  - Reduce A to Q
    - Define a transformation that maps instances of A to instances of Q
    - Prove the transformation works
      - i.e. "yes" for Q if and only if "yes" for A
    - Prove transformation runs in polynomial time
  - Also, prove Q ∈ **NP** (*if you can't, then … ?*)



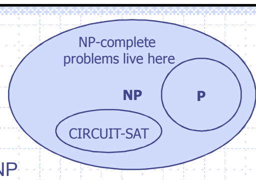
## Suppose problem A can be reduced to B in polynomial time

If $A \rightarrow_p B$,
- then B cannot be easier than A
  - Because A can be solved using the algorithm for B
- If A is NP-hard, then B is NP-hard
  - Since all problems in NP can be reduced to A
- If A is not computable, then B is not computable

Conclusions (review):
- **An easier problem can be reduced to a harder problem (or to one equally as hard)**
  - **This is why many textbooks use $\leq_p$ to indicate reduction in polynomial time (instead of $\rightarrow_p$)**
- NP-hard means at least as hard as any problem in NP, but not necessarily in NP
  - Thus not all NP-hard problems are NP-complete
- If there is a polynomial algorithm for any NP-hard problem, then all NP-complete problems can be solved in polynomial time, i.e., P=NP



## Some Thoughts about P and NP




- Belief: P is a proper subset of NP
- Implication: the NP-complete problems are the hardest in NP. Why?
  - Because if we could solve an NP-complete problem in polynomial time, we could solve every problem in NP in polynomial time
  - That is, if an NP-complete problem is solvable in polynomial time, then P=NP
  - Since so many people have attempted without success to find polynomial-time solutions to NP-complete problems, showing your problem is NP-complete is equivalent to showing that a lot of smart people have worked on your problem and found no polynomial-time algorithm



## Why Prove NP-Completeness?

- Nobody has proven that **P** ≠ **NP**,
  - However, if you prove a problem is NP-Complete, most people accept that it is probably intractable
- NP-complete problems are in a very formal, mathematical sense (as we have seen earlier) the hardest problems in NP
- Therefore it can be important to prove that a problem is NP-Complete
  - i.e., we don't need to waste time trying to come up with an efficient algorithm
  - We can instead work on an *approximation algorithm*

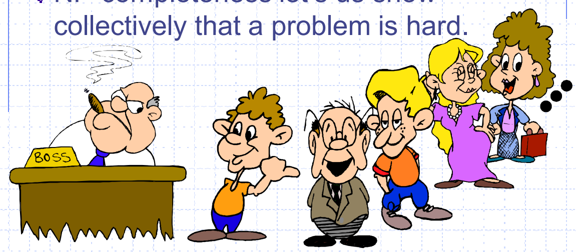
## Is the NPC concept useful?

- Suppose your professor (me or an interviewer) asks you to solve the following problem:
  - Given a graph G=(V, E), design an algorithm to find the longest simple path between two vertices *u* and *v*.
- This problem is going to be hard because the longest path would be similar to finding a Hamiltonian path between two nodes,
  - i.e., the longest path would go through as many nodes as possible
  - And Hamiltonian Path is known to be NP-Hard
  - So I knew fairly soon to stop (if I had to make a report to my boss, then I would need to define a reduction of some NPC problem to the longest path problem)
    - So I actually defined the reduction of Hamiltonian Path to Longest Path in a matter of minutes



## An Approach When 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.

 (cartoon inspired by [Garey-Johnson, 79]) 

## Homework

- Define a polynomial-time reduction from Hamiltonian Path to Longest Path
- First formulate the two problems as decision problems
  - **Hamiltonian Path**: Given a graph G=(V, E) and two vertices *u*, *v* ∈ V. Is there a simple path from u to v that visits every vertex in V?
  - **Longest Path**: Given a weighted graph G=(V,E), two vertices *u*, *v* ∈ V, and a positive number K. Is there a simple path between *u* and *v* with total weight at least K?



## Example:

- Suppose I asked you to come up with an efficient algorithm for allocating registers in a compiler
  - (we have n variables and need to allocate the *k* CPU registers to minimize running time)
- One might notice that this problem seems a lot like the K-coloring problem
- Someone else noticed that 3-SAT had a similar structure to K-coloring
  - So proved that 3-SAT $\leq_p$ K-coloring
- So you prove
  - K-coloring $\leq_p$ K-register-allocation
- What can we conclude?
  - We should look for an approximation algorithm!!!
  - Brute force takes O(n $2^n$)
  - There are data structures and algorithms that improve this, but so far they are all exponential
- Sudoku can be viewed as a 9-coloring of a graph with 81 vertices (each puzzle is a different graph)



## What about these problems?

- Given a graph G and positive integer k, does G have a simple cycle consisting of k edges?
  - NPC since Hamiltonian Cycle can be reduced to this problem
- Given a graph G and positive integer k, does G have a spanning tree T such that every vertex in T has degree at most k?
  - NPC since Hamiltonian Path can be reduced to this problem
- What about finding a maximum spanning tree?
  - Is a member of class P. Why?



## NPC Graph Problems

- Hamiltonian Path
- Hamiltonian Cycle
- Longest Path
- TSP
- Vertex Cover
- Maximum Clique Problem
- Graph Coloring
- Minimum Degree Spanning Tree
- Shortest Total Path Length Spanning Tree
  - Given graph G=(V,E) and positive integer K, is there a spanning tree T =(V,E') such that the length of the path in T between every pair of vertices u,v∈V is less than or equal to K?
- K-minimum Spanning Tree
  - Given graph G=(V,E), positive integer K ≤ |V|, and positive weight W. Is there a tree that spans K vertices with total weight ≤ W?

## How to deal with hard optimization problems?

- Look for ways to reduce the number of computations that have to be done
  - Dynamic programming
  - Branch-and-Bound
- Look for NP-complete problems with a similar structure
  - Approximation



## Branch and bound

- At each node, calculate a bound that might lie farther on in the graph
- If that bound shows that going further would result in a solution necessarily worse than the best solution found so far, then we need not go on exploring this part of the graph, tree, or solution space
- Prunes branches of a tree or closes paths in a graph
- The bound is also used to choose the open path that is most promising
- 0-1 Knapsack problem can be solved in this way rather than through dynamic programming (in pseudo-polynomial time)



## Main Point

3. A problem M is said to be NP-hard if every other decision problem in NP can be reduced to M in polynomial time. M is NP-complete if M is also in NP. NP-complete problems are, in a very formal sense, the hardest problems in NP.
   Individual and collective problems are hard to solve on the surface level of the problem. However, if we go to the root, the source of creativity and intelligence in individual and collective life, we can enliven and enrich positivity on all levels of life.



## How to deal with NP-complete optimization problems?

- Apply an approximation algorithm.
  - Typically faster than an exact solution.
  - Assuming the problem has a large number of feasible solutions.
    - Also, has a cost function for the solutions.
    - Want to find a solution with minimum cost in a reasonable time (i.e. polynomial time).
- Apply Heuristic solution
  - Looking for “good enough” solutions.



## Connecting the Parts of Knowledge with the Wholeness of Knowledge

1. All problems for which reasonably efficient algorithms are known are grouped into the class P (polynomial-bounded). The class NP consists of problems that can be solved by non-deterministic polynomial-time algorithms. NPC problems are the “hard” problems in NP.

2. Algorithms have been improved through techniques like dynamic programming and branch and bound solutions. Since complexity theory has not been able to establish non-trivial lower bounds for any NPC problem, for all we know, NPC problems can be solved in polynomial time, i.e., P=NP.



3. **Transcendental Consciousness** is the field of all solutions, a taste of life free from problems.
4. **Impulses within Transcendental Consciousness**: The natural laws within this unbounded field are the algorithms of nature that efficiently solve all problems of the universe.
5. **Wholeness moving within itself:** In Unity Consciousness, one realizes the full dignity of cosmic life in the individual. We have the vision of possibilities – transcend to remove stress in the individual physiology and live our full potential.