Complexity classes
P and NP

Lecture 08.01

by Marina Barsky

Design and analysis of algorithms

Main focus: practical algorithms + supporting theory for solving fundamental computational problems:

- **Sorting**
- **Searching**
- **Shortest paths**
- **Sequence alignment**
- **Spanning trees**
- **…**

You might feel that now you can solve any problem efficiently, and always should try to do even better

Design and analysis of algorithms

Main focus: practical algorithms + supporting theory for solving fundamental computational problems:

- **Sorting**
- **Searching**
- **Shortest paths**
- **Sequence alignment**
- **Spanning trees**
- **…**

You might feel that now you can solve any problem efficiently, and always should try to do even better

Bad news: many important practical problems that you will encounter in your projects **do not have known efficient solutions**!

We need to know how:

- **Classify problems by hardness**
- **Identify problems that cannot be efficiently solved**
- **Deal with such problems**

Complexity class P

We say that the problem is *tractable* if there is an algorithm which solves it in time $O(n^k)$ for some constant k, and where n represents the input size [More precisely - the number of bits or keystrokes needed to describe the input]

Tractable:

$O(n)$, $O(n^2)$, $O(n^{1000})$, $O(n^{10,000,000})$

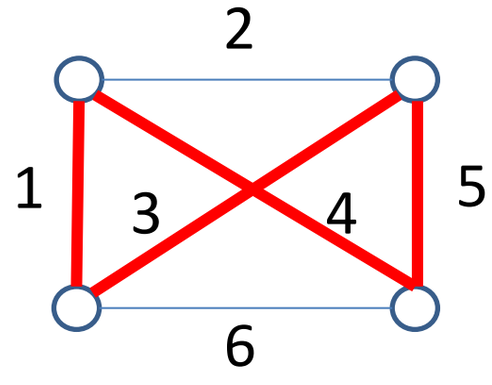
Class P: set of all problems solvable in polynomial time

All the algorithms we designed and implemented in this course belong to class P

Not in P?

Traveling Salesperson Problem (TSP)

Input: complete undirected graph with non-negative edge costs
Output: a min-cost tour – a cycle that visits each vertex exactly once



TSP path: 13

Solution (exponential):

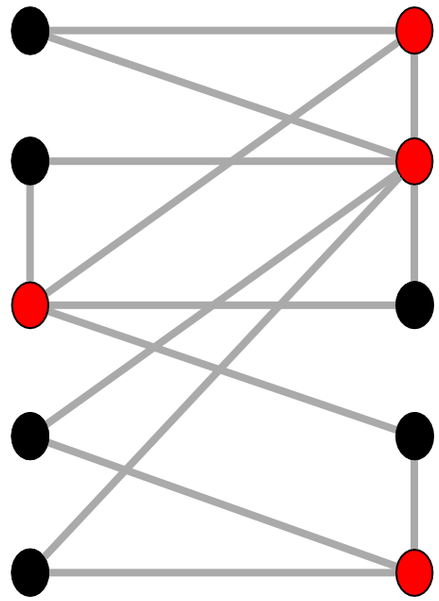
- Try all permutations of vertices
- Select the tour with the cheapest cost

We solved shortest paths, min spanning trees, why not TSP?
60 years of research – and there is no polynomial-time algorithm?

Not in P?

Min Vertex Cover

Input: graph G (V, E)
Output: a minimum-size subset C of vertices such that for each edge (v,w), we have v ∈ C or w ∈ C (C *covers* all the edges)



Vertex Cover of size 4

Solution (exponential):

- Try every subset of V
- For each subset check if it covers all the edges.
- Keep the subset of a minimum size

Not in P?

Knapsack 01 without repetitions

Input: set of n items with their weights and values and the knapsack capacity W
Output: maximum value of knapsack filled with items that fit into W (each item can be used only once)

Solution (exponential):

- Check 2^n different subsets of items
- Verify if items fit into a knapsack and compute total value. Each verification takes O(n)

Time complexity **$O(n2^n) = 4*16 = 64$**
[exponential in n]

What about our earlier DP solution?

subset	total weight	total value
Ø	0	$0
{1}	7	$42
{2}	11	$12
{3}	9	$40
{4}	5	$25
{1,2}	18	$54
{1,3}	16	$82
{1,4}	12	$67
{2,3}	20	$52
{2,4}	16	$37
etc. …		

Example:
items =
{(7 lbs, $42),
(11 lbs, $12),
(9 lbs, $40),
(5 lbs, $25)}

n=4 (4 items)
W = 20

KnapsackDP(*W, n* items)

initialize all *maxvalue* [0, *i*] ← 0
initialize all *maxvalue* [*w*, 0] ← 0
for *i* **from** 1 **to** *n*:
    for *w* **from** 1 **to** *W* : 1,2,3,….W
        maxvalue [*w, i*] ← *maxvalue* [*w, i* -1]
        if $w_i \leq w$:
            $val \leftarrow maxvalue\ [w - w_i, i - 1] + v_i$
            if *val* > *maxvalue* [*w, i*]:
                maxvalue [*w, i*] ← *val*
return *maxvalue* [*W, N*]

For n=4, W=20

Exhaustive – running time: $O(n2^n)$ 4*16 = 64

DP – running time $O(nW)$ 20*4 = 80

Running Time of DP Knapsack: *closer look*

- The running time is $O(nW)$
- *W* is not the **size** of the input - after all, the input consists of a single number (total knapsack capacity) - not *W* knapsacks
- For example for W=1,125,899,906,842,624 we will perform 1,125,899,906,842,624 loop iterations, while the input still consists of a single number W
- We loop over all possible values between 0 and W, and the time is **not proportional** to the size of the input, which is in fact n+1 (n items and 1 number W)

DP Knapsack is not polynomial!

- The running time of an algorithm is defined as a function of the input size
- In normal O(n) complexity we assume that reading each number takes a constant time (each number is using a constant number of bits)
- Say, we use m=log W bits to represent number W
- We need to loop from 0 to $W=2^m$
- The complexity is $O(n2^m)$: we need to check 2^m imaginary knapsacks!
- The algorithm is exponential in the input size – the number of bits used to represent the capacity: if we add just one more bit – we double W and double the run time

DP knapsack is exponential:

$O(n2^m)$

Input size: number of bits to represent number W

Pseudo-polynomial running time

- The complexity of an algorithm refers to the **number of input elements**, **not a value of a single element** in the input
- More precisely, the input size n is a number of keystrokes (alternatively number of bits) needed to describe the input
- Thus the complexity of the knapsack remains exponential in input size even with dynamic programming
- The DP knapsack algorithm is ***pseudo-polynomial***. That means: if W is O(n) then the algorithm is polynomial. However if W is > O(n), then algorithm is exponential in m – number of bits encoding W
- NP-hard problems with pseudo-polynomial solutions are called *weakly NP-complete*

Polynomial or pseudo-polynomial?

- Is prime (num)
- Naïve GCD (a,b)
- Money change (target)
- Subset sum (A of size n, target sum)
- Edit distance (S1 of size m, S2 of size n)
- Bellman-Ford (G(V,E), source s)

wikipedia link

Polynomial or pseudo-polynomial?

- Is prime (num) – pseudo-polynomial
- Naïve GCD (a,b) – pseudo-polynomial
- Money change (target) – pseudo-polynomial
- Subset sum (A of size n, target sum) – pseudo-polynomial
- Edit distance (S1 of size m, S2 of size n) –polynomial
- Bellman-Ford (G(V,E), source s) – polynomial

Intractable problems

Not all problems are tractable=can be solved in polynomial time

What is common to all the above problems: they can be solved via exhaustive search

Problem types

Most existing computational problems belong to one of three types:

- **Decision problems: return Boolean answer Yes/No**
- **Optimization problems: return min/max of some function [subject to constraints]**
- **Construction problems: return a structure with desired properties**

Problem types: examples

- **Decision problems: return Boolean answer Yes/No**
  - Is there a subset with sum = k? Yes or no?
  - Is there a cycle in the graph which passes through all vertices and visits every edge exactly once?

- **Optimization problems: return min/max of some function**
  - What is the value of the min-cost path from s to t?
  - What is the max possible value in a knapsack?

- **Construction problems: return a structure with desired properties**
  - Produce a shortest path from s to t
  - Produce a sequence of items in knapsack of maximum value

Complexity theory considers decision problems only

Decision problems: return Boolean answer Yes/No

They are the most fundamental computational problems.

Problem of any other type can be reduced to a decision problem or a sequence of decision problems

Example 1: Optimization problem

Problem p1: Max value of knapsack (optimization problem)
Problem p2: Is there a knapsack of value at least k (decision problem)

Reduction of p1 to p2:

- We ask: is there a knapsack with value V = (v_i+v_2+v_3+…v_n)?
  - If the answer is yes, this is the max value – we fit all the available items
  - If the answer is no, next decision problem: is there knapsack with value V/2?
- Binary search until we find the max value

Example 2: Construction problem

Problem p1: Sequence of items in max-valued knapsack (construction)
Problem p2: Max value of knapsack (optimization)
Problem p3: Is there a knapsack of value at least k (decision)

Reduction of p1 to p2 (which in turn reduces to p3)

- After we found max value V_{max} (see previous slide), we start removing one item i at a time and ask: is there still knapsack with value V_{max}?
- If the answer is no, the solution has to include item i
- We check for all n items in turn

Complexity class NP

The complexity class NP is defined to include all the decision problems from class P but allows for the inclusion of problems that may not be in P

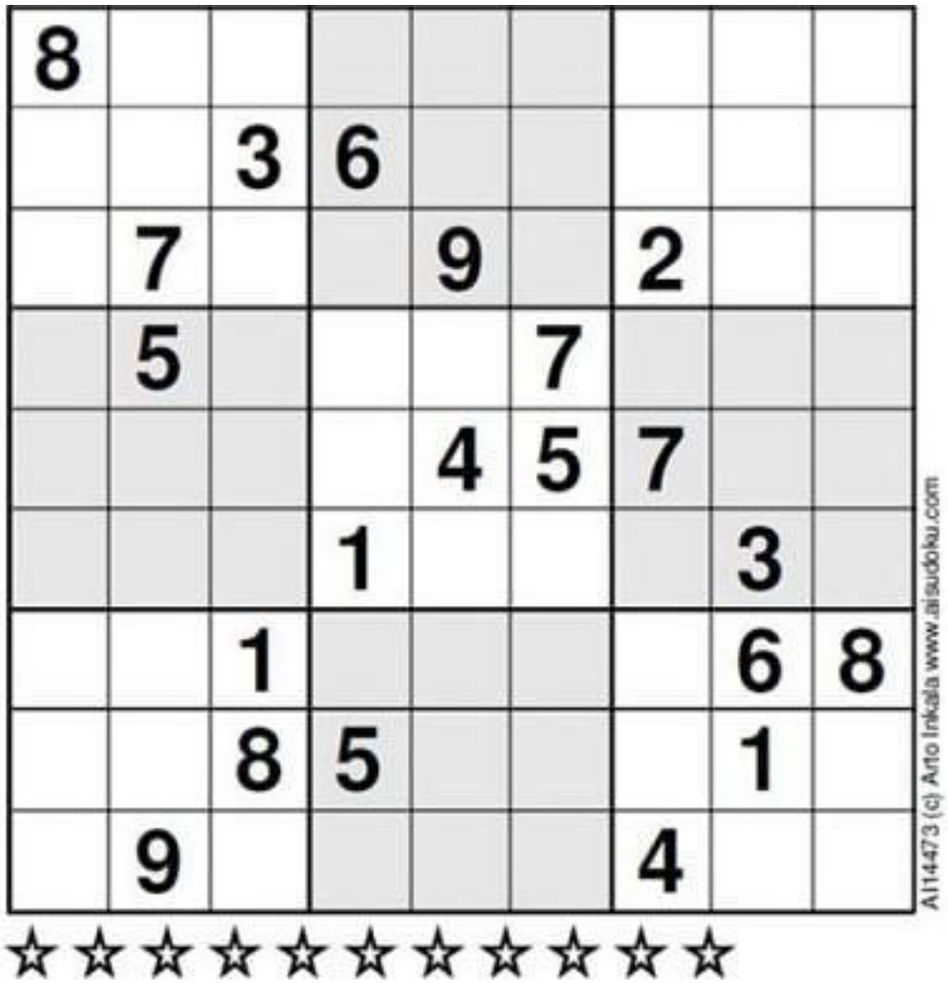
Every problem in NP can be solved in exponential time via Exhaustive Search

The solution must be efficiently verifiable:

- **Solutions (certificates) always have length polynomial in input size**
- **Proposed solution can be verified in polynomial time**

Checking a given solution is polynomial,
number of candidates to check can be exponential!

Class NP—example



5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Checking solution to Sudoku can be done in polynomial time. So sudoku is in **NP**

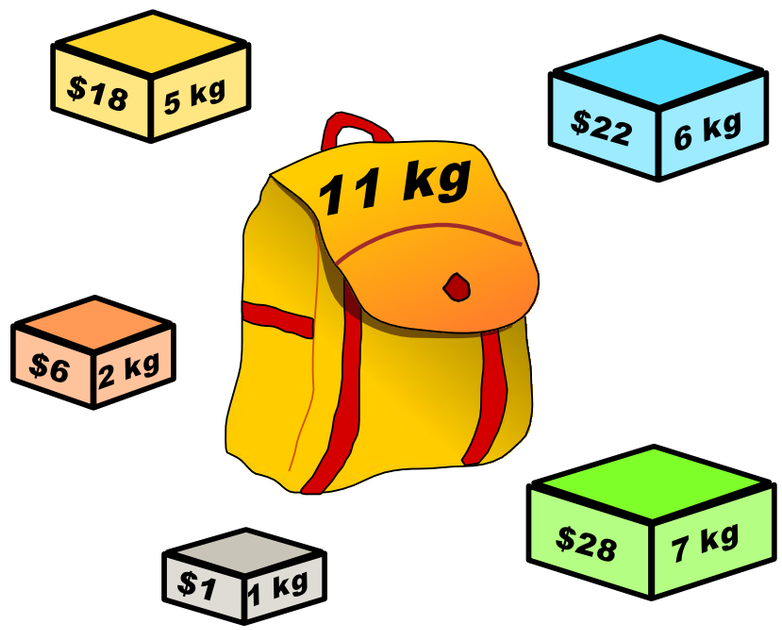
Class NP—example

Problem: is there a knapsack with value $40?

- {3, 4} has value $40 (and weight 11)

i	v_i	w_i
1	$1	1 kg
2	$6	2 kg
3	$18	5 kg
4	$22	6 kg
5	$28	7 kg

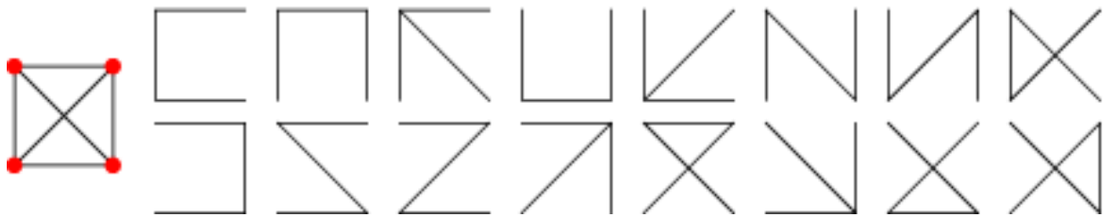
knapsack instance
(weight limit W = 11)



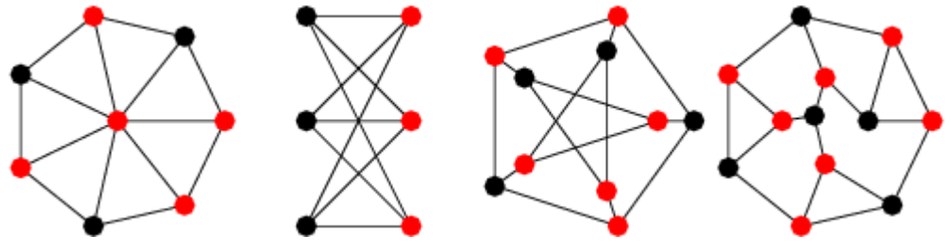
Checking the total value of a proposed knapsack can be done in polynomial time. So knapsack is in NP

Which problems in NP are tractable?

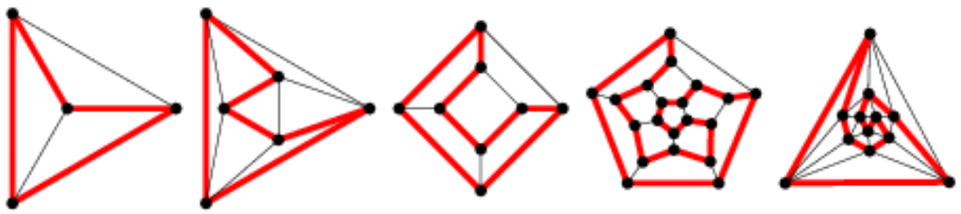
Spanning tree ✓



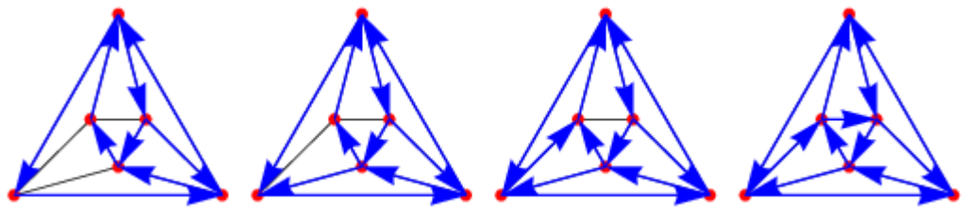
Min vertex cover



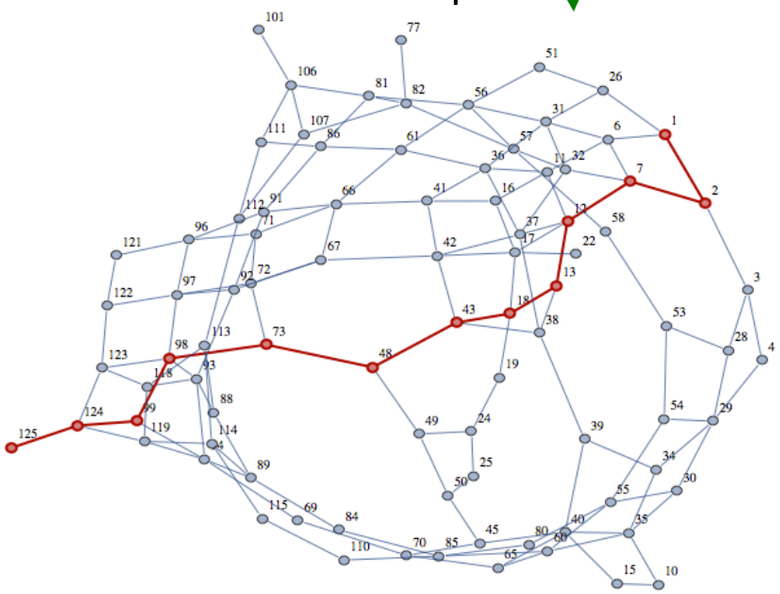
Hamiltonian Cycle



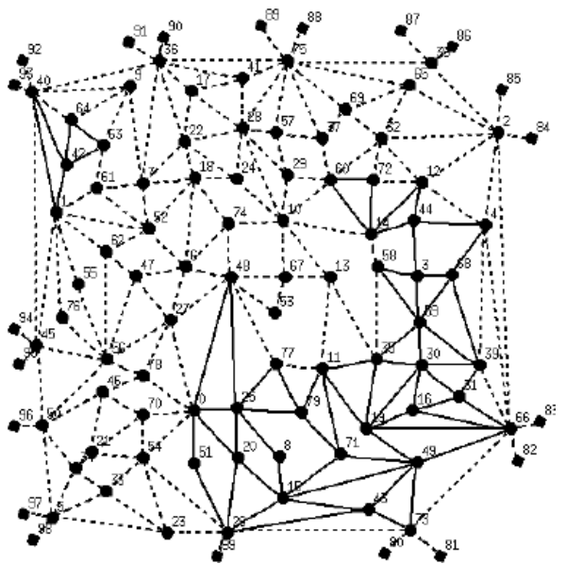
Eulerian cycle ✓



Shortest path ✓



Longest path



The definitions seem very similar…

Defining Intractability

- **Complexity class NP contains different problems: some of them we know to be *tractable* and others *intractable*: a polynomial-time algorithm is unknown**

- **Problems in NP are still decidable (solvable): if only we could magically guess the right solution, we could then quickly verify it**

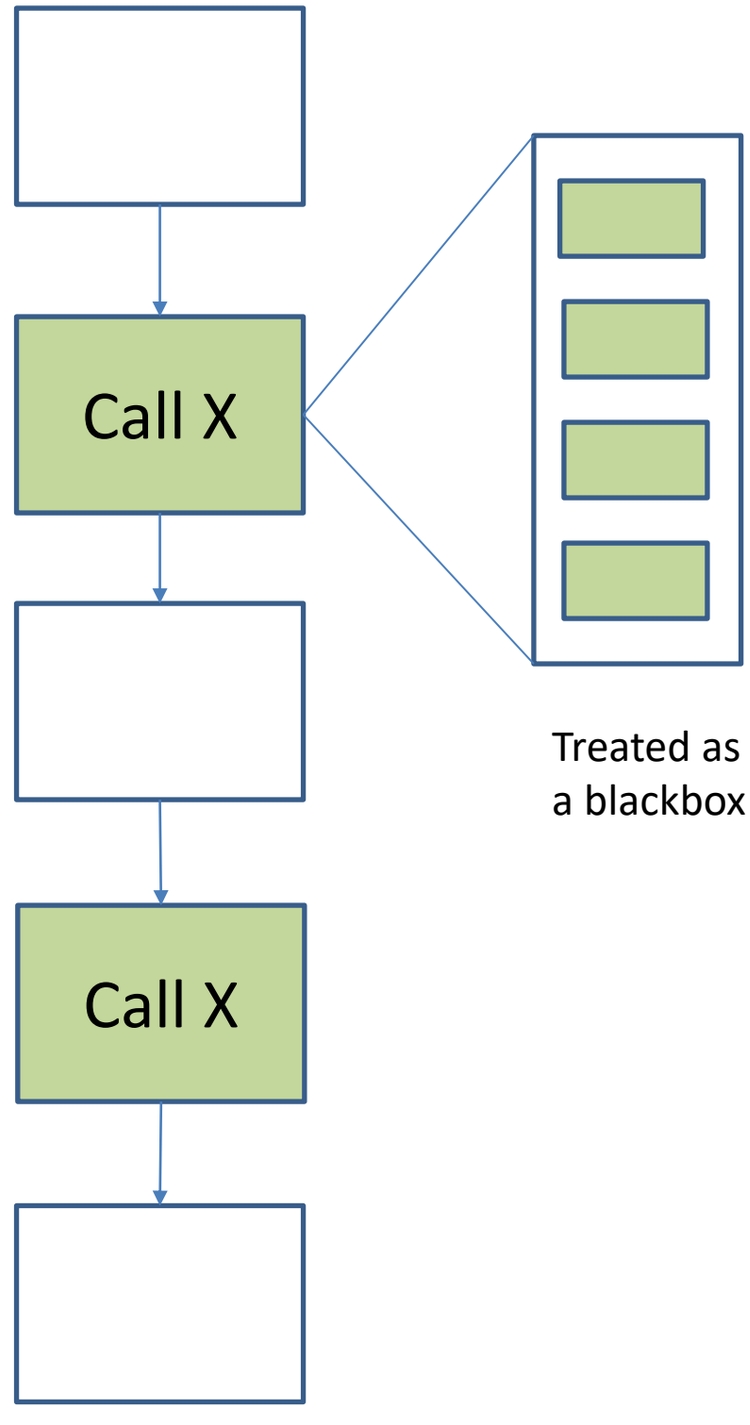
- **How do we formally define intractability?**

- **Evidence of intractability: relative difficulty**
  TSP is "at least as hard as" the list of really hard problems

Use of reductions

- **Reduction from one problem to another is a routine approach in algorithm design**

- **For any new problem Y we first ask:**
  - **Maybe I already know how to solve it? Maybe I can rephrase it as a shortest-path problem X?**
  - **Maybe I can invoke a known algorithm for X multiple times?**
  - **Or use a known algorithm to solve part of a new problem?**



Reductions: informally

***Informally*:**
Problem Y reduces to X if given an algorithm for solving X, we can use this algorithm as a subroutine for solving Y

Examples:

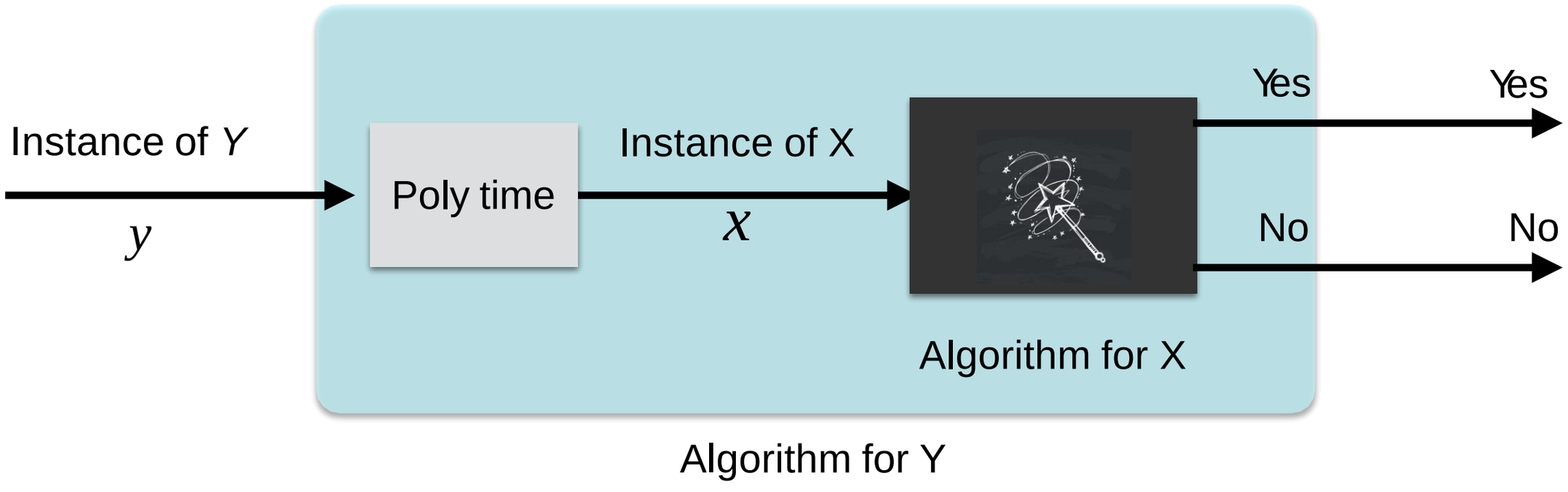
- **Computing median reduces to sorting**
- **All pair shortest paths reduces to n invocations of the single-source shortest path**

Polynomial-time reductions

Formal definition: Decision problem Y is *(polynomial-time) reducible* to decision problem X if we can convert any instance of Y into an instance of X and the following conditions hold:

- **The conversion of input of Y to input of X takes polynomial number of steps and the new input is polynomial in size of the original input**
- **All the additional operations are polynomial in the input size, and the calls to X can be done at most polynomial number of times**
- **For any instance of problem Y the reduction which uses X returns the correct decision**

Notation: $Y \leq_p X$



Completeness, or relative hardness

- **Suppose Y reduces to X in polynomial time: $Y \leq_p X$**
- **If X is tractable then Y is tractable (polynomial time is additive)**
- **If we know that Y cannot be solved efficiently in poly-time, then X cannot be solved in poly-time either**
- **Contrapositive use of reductions:**
  If Y is not in P then neither is X
  X is at least as hard as Y
- **To use this idea of comparative hardness, we need to have at least one problem that is not in P: this will be the hardest problem in NP, and we will call it NP-complete**

NP-completeness

- **By definition: solving 1 NP-complete problem in poly-time will provide a solution to all NP problems [P=NP]**
- **Interpretation: an NP-complete problem encodes simultaneously all problems for which the solution can be efficiently recognized (a “universal problem”)**
- **Can such a problem really exist?**

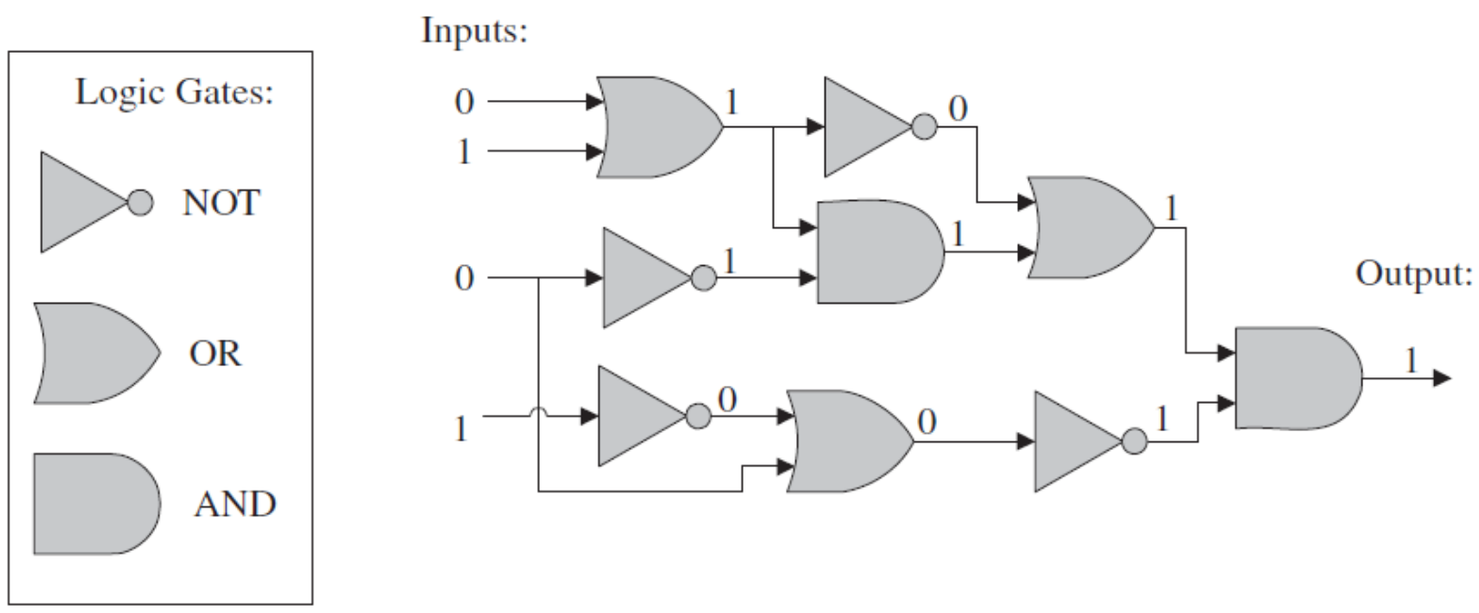
Cook-Levin theorem

(Cook 71, Levin 73)

NP-complete problems exist

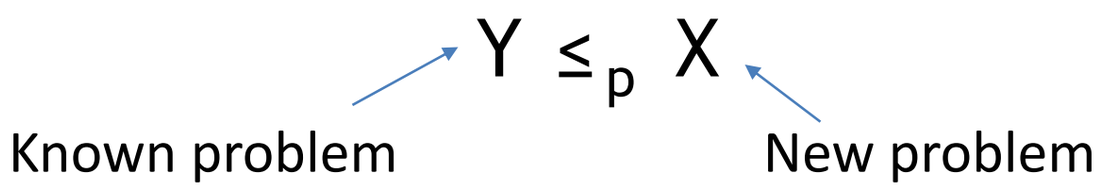
Any computer program can be represented by a circuit-SAT.

Circuit-SAT is NP-complete



You'll see the proof in CSCI 361

The logic of comparative hardness



Suppose we can reduce problem Y to problem X in polynomial time

- **If Y is in P then nothing is known about X – we can always encode the easy instance into a hard one**
- **If X is in P then Y is also in P: solve X in poly time + poly time of reduction**
- **If Y is not in P then X is not in P (contrapositive): suppose X is in P then Y should also be in P – but we know it is not**
- **If X is not in P then nothing known about Y – Y might still have a poly solution – we could have encoded an easy problem Y into a hard problem X**

To prove that a new problem X is NP-complete, reduce a known NP complete problem Y to X

Classic problem in NP: Satisfiability

Given a logical expression, can we assign "True" and "False" to the variables to *satisfy* the equation (make the expression True)?

SAT. Given a CNF formula φ, does it have a satisfying truth assignment?

3-SAT. A SAT formula where each clause contains exactly 3 literals (corresponding to different variables)

$$\varphi = (\overline{x_1} \vee x_2 \vee x_3) \wedge (x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee x_4)$$

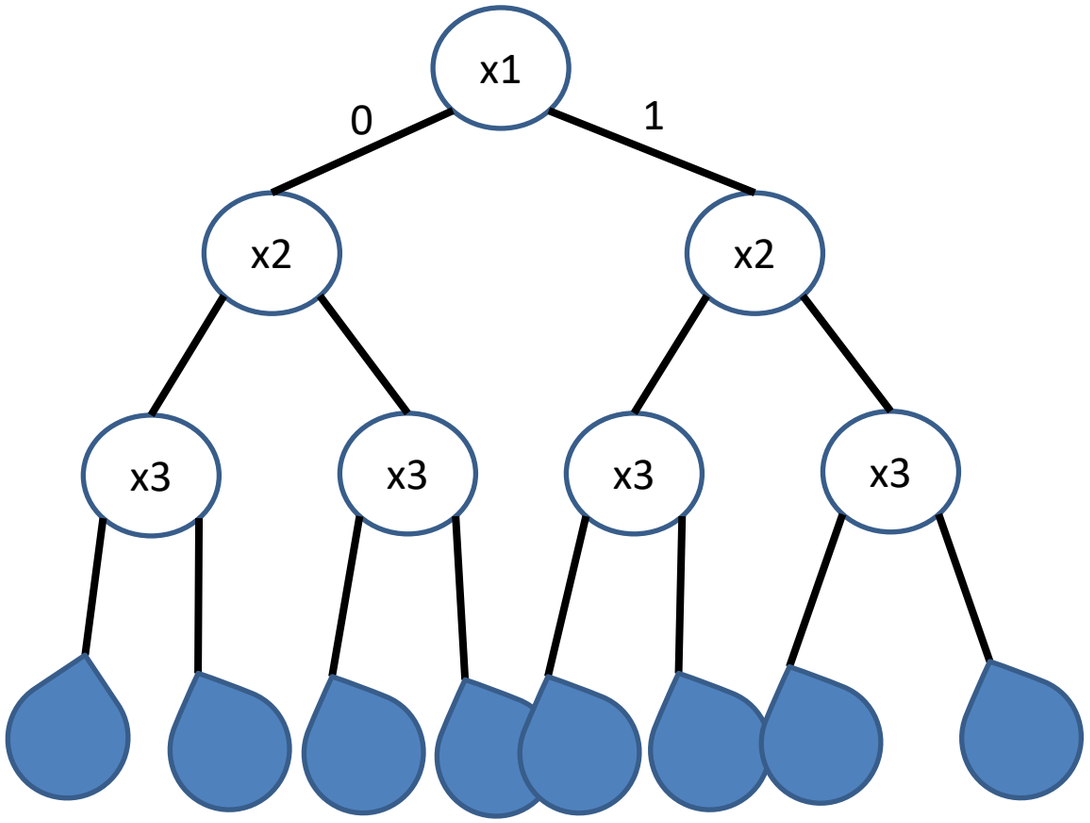
Satisfying instance: $x_1 = 1, x_2 = 1, x_3 = 0, x_4 = 0$

SAT, 3-SAT $\in$ NP

- **Certificate: truth assignment to variables (poly-size)**
- **Verification: check if assignment makes φ true (poly-time)**

Decision tree

Decision problems can be expressed as a tree of decisions or choices



$$\phi = (\overline{x_1} \vee x_2 \vee x_3) \wedge (x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee x_3)$$

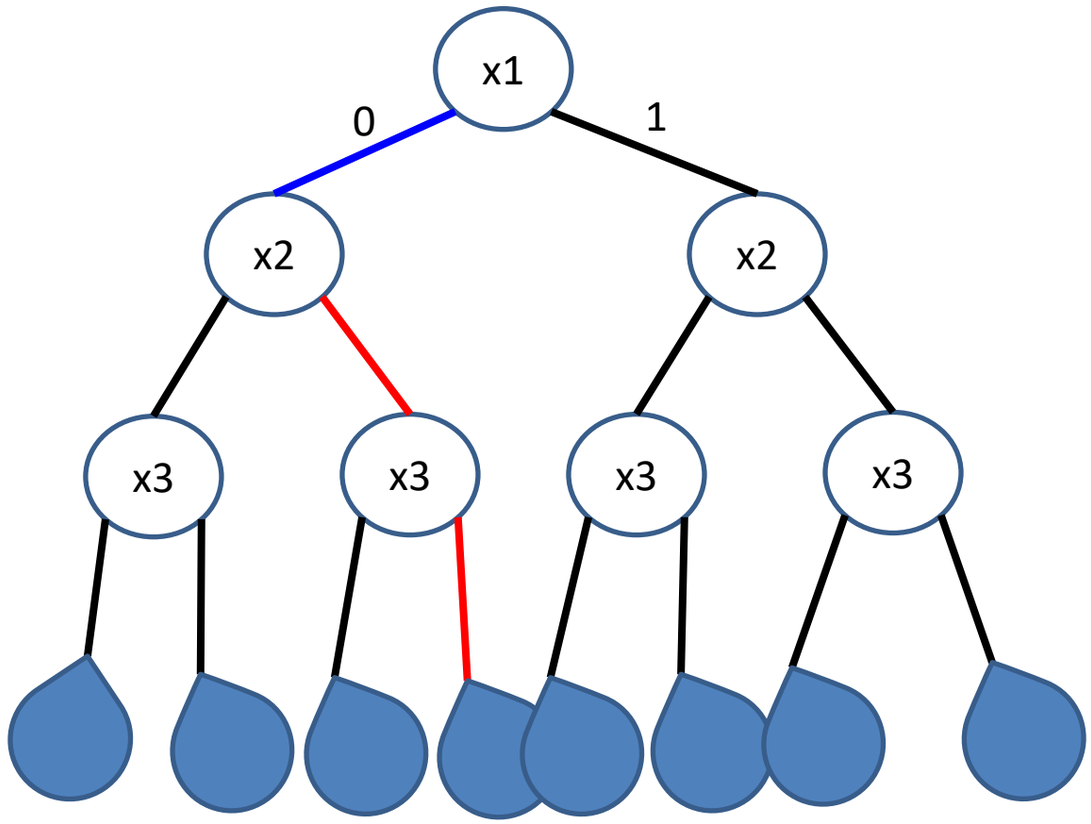
Alternative definition of class NP:

- **The decision tree may have exponential number of leaves**
- **The height of the decision tree must be polynomial in input size**
- **Each node of the tree must be encoded in polynomial number of bits**

Each leaf represents a solution (certificate) and can be reached in polynomial number of steps

Class NP: Non-deterministic Polynomial

- **When searching for a satisfying assignment, we allow to use function *choose(b)* which randomly (non-deterministically) chooses the next decision**
- **Once all the selections have been made – the solution can be easily verified**
- **We allow a random "guess". If we are lucky – we found the satisfying assignment**



$$\phi = (\overline{x_1} \vee x_2 \vee x_3) \wedge (x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee x_3)$$

Vast majority of natural computational problems are in NP

NP: name

Non-deterministic Polynomial (Knuth, Terminological Proposal, 1974)

Alternative name:

PET

Possibly Exponential Time (currently)
Provably Exponential Time (if proven that P ≠ NP)
Previously Exponential Time (if proven that P = NP)

P vs NP

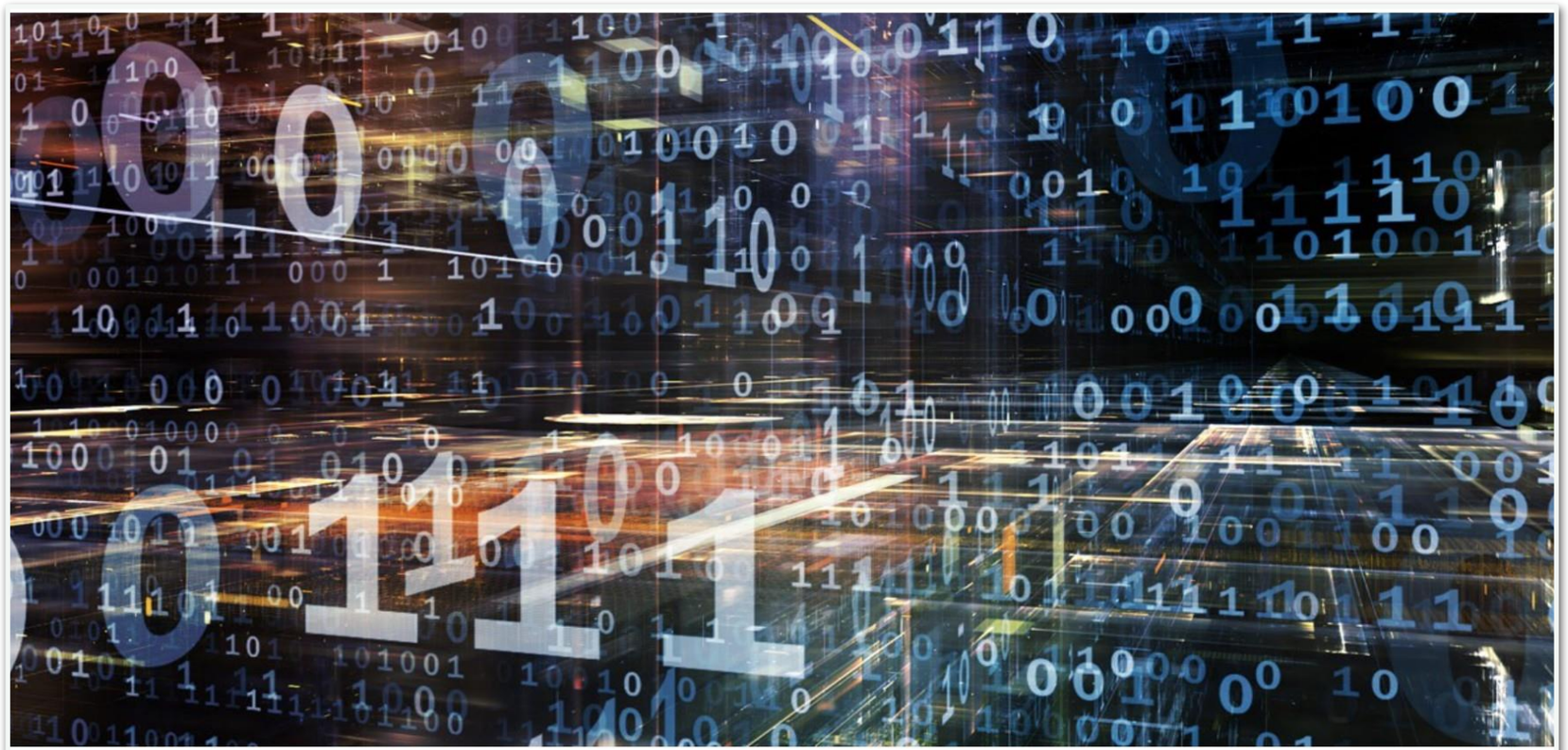
We know that every problem in P is also in NP

What about the reverse?

- **If a problem can be efficiently *verified*, does that mean it can be efficiently solved in the first place?**
- **Or, do there exist problems that can be verified quickly that are provably *impossible* to solve quickly?**

The answer: we do not know

Million Dollar Question:
P vs NP



P vs NP and the $1M Millennium Prize Problems

What's the most difficult way to earn $1M US Dollars?

https://medium.com/@mpreziuso/

Is P=NP?

Widely believed: P≠ NP
But this has not been proven!

Arguments:

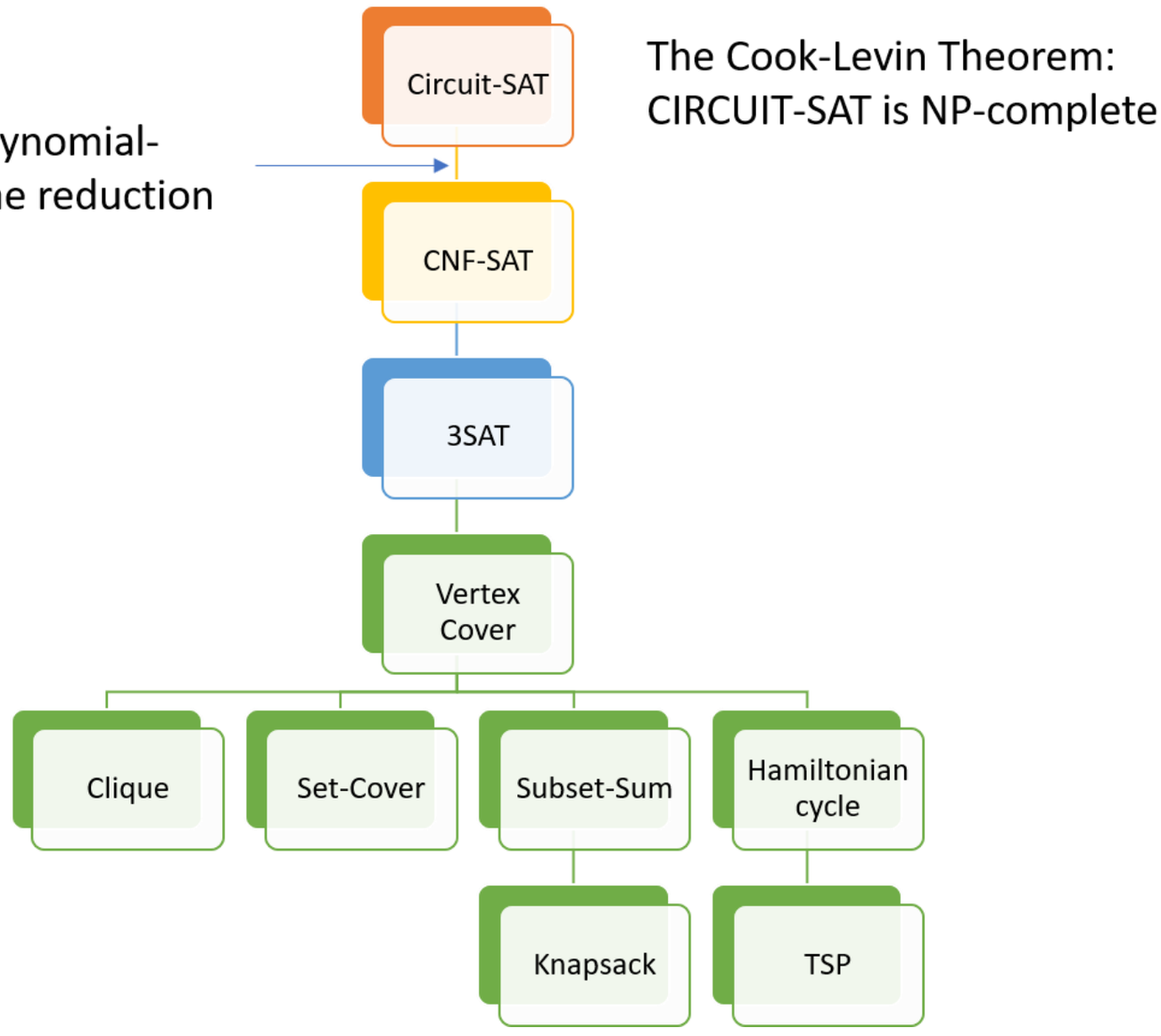
1. **P≠ NP (psychological). Many smart people tried to solve at least one NP-complete problem and never succeeded**

2. **P≠ NP (philosophical). To prove something is much more difficult than to verify somebody else's proof. Verifying in poly-time does not imply that we can solve in poly-time. Can mathematical creativity be automated?**

3. **P=NP (mathematical). There are surprisingly efficient polynomial-time algorithms (i.e. Number of inversions, Matrix multiplication) which seem counter-intuitive and difficult to discover. So maybe we just need to try harder?**

23 NP-complete problems

(Karp, 72)



The Cook-Levin Theorem:
CIRCUIT-SAT is NP-complete

All proven to be at least as hard as Circuit-SAT