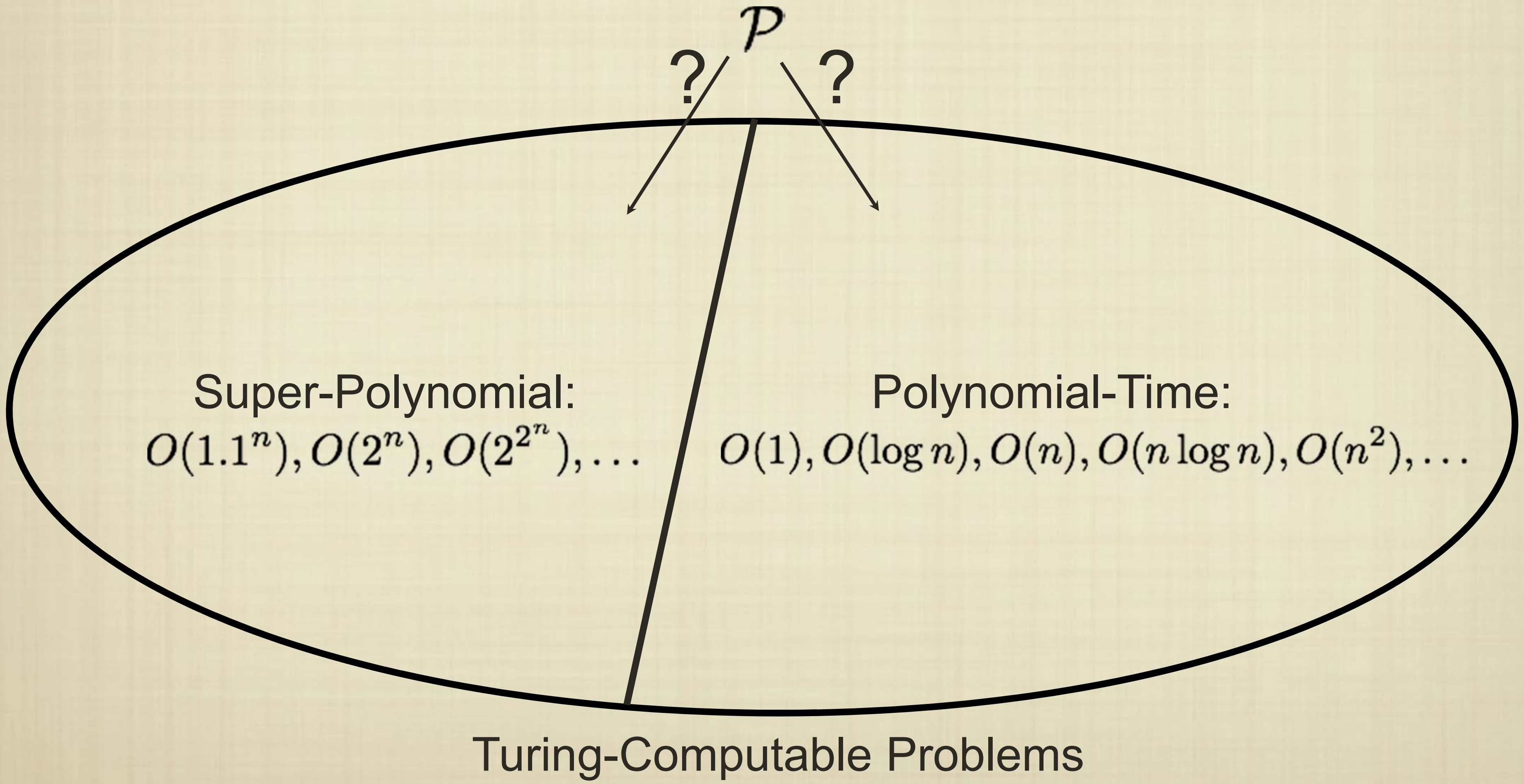
Theory and Frontiers of Computer Science II

Fall 2013

Carola Wenk

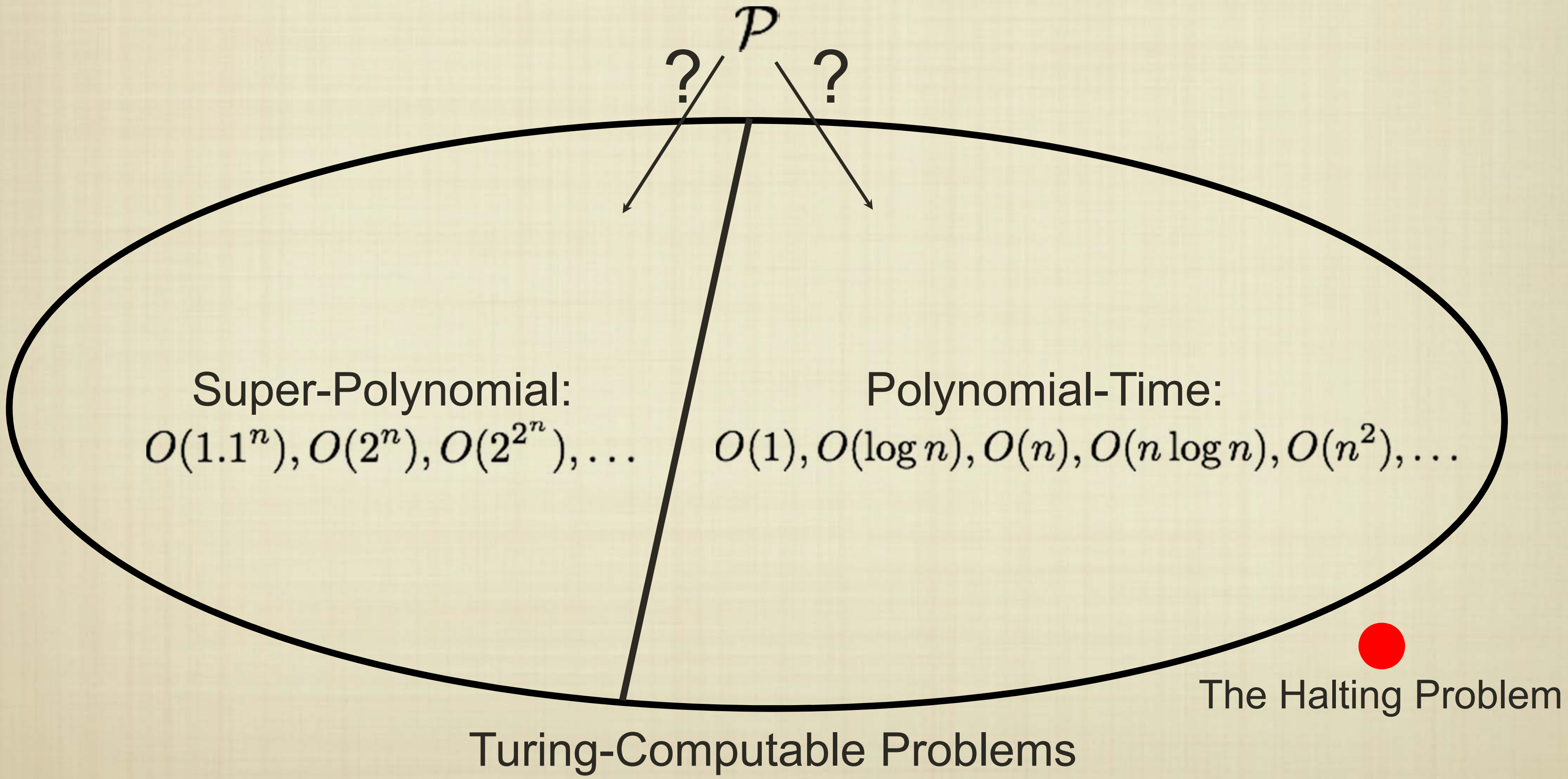
Computational Complexity

The field of “computational complexity” tries to categorize the difficulty of computational problems. It is a purely theoretical area of study, but has wide-ranging effects on the design and implementation of algorithms.


$\mathcal{P}$
?
?
Super-Polynomial:
$O(1.1^n), O(2^n), O(2^{2^n}), \ldots$
Polynomial-Time:
$O(1), O(\log n), O(n), O(n \log n), O(n^2), \ldots$
Turing-Computable Problems

Computational Complexity

The field of “computational complexity” tries to categorize the difficulty of computational problems. It is a purely theoretical area of study, but has wide-ranging effects on the design and implementation of algorithms.


$\mathcal{P}$
?
?
Super-Polynomial:
$O(1.1^n), O(2^n), O(2^{2^n}), \ldots$
Polynomial-Time:
$O(1), O(\log n), O(n), O(n \log n), O(n^2), \ldots$
The Halting Problem
Turing-Computable Problems

Upper and Lower Bounds

If we can come up with an algorithm that correctly solves a particular problem $\mathcal{P}$, then its worst-case running time is an upper bound.

What would be more useful though, is evidence that $\mathcal{P}$ cannot be solved in a given amount of time. In other words, to establish difficulty we need a lower bound on the running time of any algorithm for $\mathcal{P}$.

Upper Bound

Algorithm A for $\mathcal{P}$

$$\downarrow$$

$\mathcal{P}$ can be solved in $T_A(n)$ time

Lower Bound

Regardless of the algorithm, the problem $\mathcal{P}$ cannot be solved in less than $T^*(n)$ time.

Upper and Lower Bounds

If we can come up with an algorithm that correctly solves a particular problem $\mathcal{P}$, then its worst-case running time is an upper bound.

What would be more useful though, is evidence that $\mathcal{P}$ cannot be solved in a given amount of time. In other words, to establish difficulty we need a lower bound on the running time of any algorithm for $\mathcal{P}$.

Upper Bound

MergeSort for sorting a list

$\downarrow$

Sorting can be done in $O(n \log n)$ time

Lower Bound

Every sorting algorithm requires at least ??? time.

Upper and Lower Bounds

If we can come up with an algorithm that correctly solves a particular problem $\mathcal{P}$, then its worst-case running time is an upper bound.

What would be more useful though, is evidence that $\mathcal{P}$ cannot be solved in a given amount of time. In other words, to establish difficulty we need a lower bound on the running time of any algorithm for $\mathcal{P}$.

Upper Bound

MergeSort for sorting a list

$\downarrow$

Sorting can be done in $O(n \log n)$ time

Lower Bound

Every sorting algorithm requires at least cn time.

Can we match the lower bound to the upper bound?

Lower Bound for Sorting

We came up with an algorithm for sorting that took $O(n \log n)$ time, can we be sure that this is the fastest possible?

Given a list of distinct elements, consider what any algorithm for sorting actually does:

$$\begin{tikzcd}
\text{Unsorted List} \arrow[d] & L & \text{How many possible orderings?} \\
\mathcal{A} \arrow[d] & & \\
\text{Sorted List} & L & \text{How many possible outputs?}
\end{tikzcd}$$

Lower Bound for Sorting

We came up with an algorithm for sorting that took $O(n \log n)$ time, can we be sure that this is the fastest possible?

Given a list of distinct elements, consider what any algorithm for sorting actually does:

$$\begin{tikzcd}
\text{Unsorted List} \arrow[dd] & L & \text{How many possible orderings?} \\
 & & n! \\
\mathcal{A} \arrow[d] & & \\
\text{Sorted List} & L & \text{How many possible outputs?} \\
 & & 1
\end{tikzcd}$$

Lower Bound for Sorting

$$\begin{tikzcd} \text{Unsorted List} \arrow[d] & L \\ \mathcal{A} \arrow[d] & \\ \text{Sorted List} & L \end{tikzcd}$$

How many possible orderings?

$n!$

How many possible outputs?

1

Any correct sorting algorithm must be able to permute any input into a uniquely sorted list. Therefore any sorting algorithm must be able to “apply” any of the $n!$ possible permutations necessary to produce the right answer.

Lower Bound for Sorting

Any sorting algorithm must be able to “apply” any of the $n!$ possible permutations necessary to produce the right answer.

We can visualize the behavior of any sorting algorithm as a sequence of decisions based on comparing pairs of items:

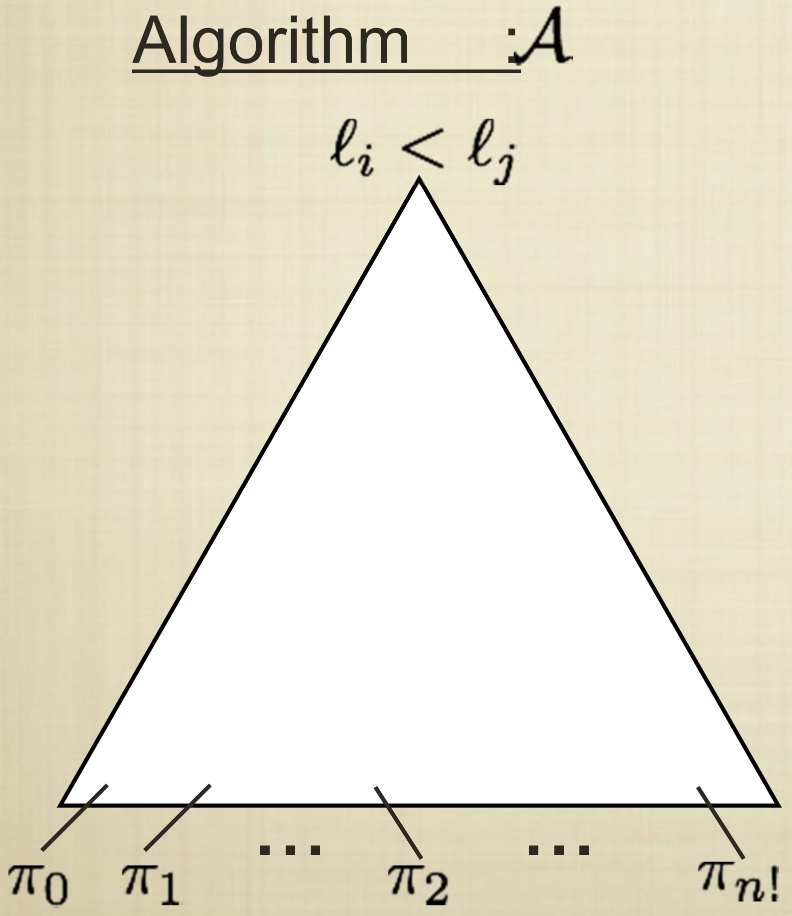
$$L = [\ell_0, \ell_1, \ell_2]$$

<u>Algorithm</u> : $\mathcal{A}$

$$\begin{tikzcd}
 & & & & \ell_0 < \ell_1 \arrow[dll, "\text{Yes}"'] \arrow[drr, "\text{No}"] & & & & \\
 & & \ell_0 < \ell_2 \arrow[dl, "\text{Yes}"'] \arrow[dr, "\text{No}"] & & & & \ell_1 < \ell_2 \arrow[dl, "\text{Yes}"'] \arrow[dr, "\text{No}"] & & \\
 & \ell_1 < \ell_2 \arrow[dl, "\text{Yes}"'] \arrow[dr, "\text{No}"] & & {} & \vdots & {} & & \langle 2, 1, 0 \rangle & \\
\langle 0, 1, 2 \rangle & & \langle 0, 2, 1 \rangle & & & & & &
\end{tikzcd}$$

Lower Bound for Sorting

For a list L with n items, let the possible permutations be $\pi_0, \pi_1, \ldots, \pi_{n!}$. Any sorting algorithm must be able to "reach" all of these permutations by making a sequence of comparisons. The corresponding decision tree is:



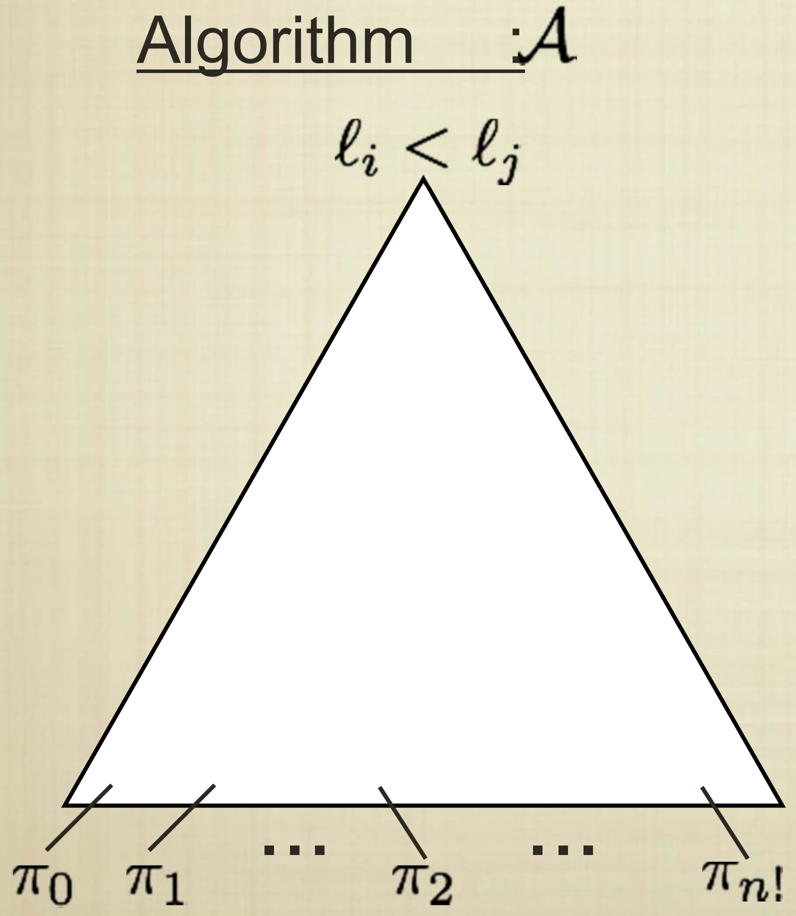
What does any of this tell us about the running time?

This decision tree is a binary tree, and its height is a lower bound on the running time of $\mathcal{A}$.

What is the minimum height of any binary decision tree?

Lower Bound for Sorting

For a list L with n items, let the possible permutations be $\pi_0, \pi_1, \ldots, \pi_{n!}$. Any sorting algorithm must be able to "reach" all of these permutations by making a sequence of comparisons. The corresponding decision tree is:



$n! \leq$ # leaves $\leq 2^{\text{height}}$

So, $n! \leq 2^{\text{height}}$

This is equivalent to:

$\log n! \leq$ height

Lower Bound for Sorting

For a list L with n items, let the possible permutations be $\pi_0, \pi_1, \ldots, \pi_{n!}$. Any sorting algorithm must be able to "reach" all of these permutations by making a

The corresponding decision tree is:

$n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 1$

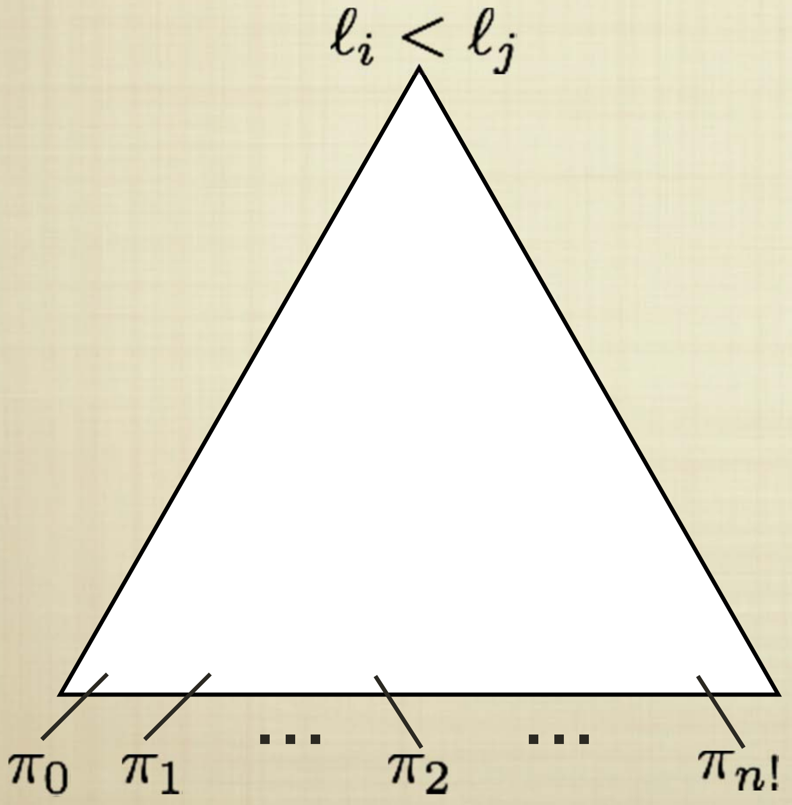
$= n \cdot \ldots \cdot (n/2+1) \cdot n/2 \cdot (n/2-1) \cdot \ldots \cdot 1$

$\geq n/2 \cdot \ldots \cdot n/2 \cdot n/2 \cdot 1 \cdot \ldots \cdot 1$

$\geq (n/2)^{n/2}$

So, $n! \geq (n/2)^{n/2}$

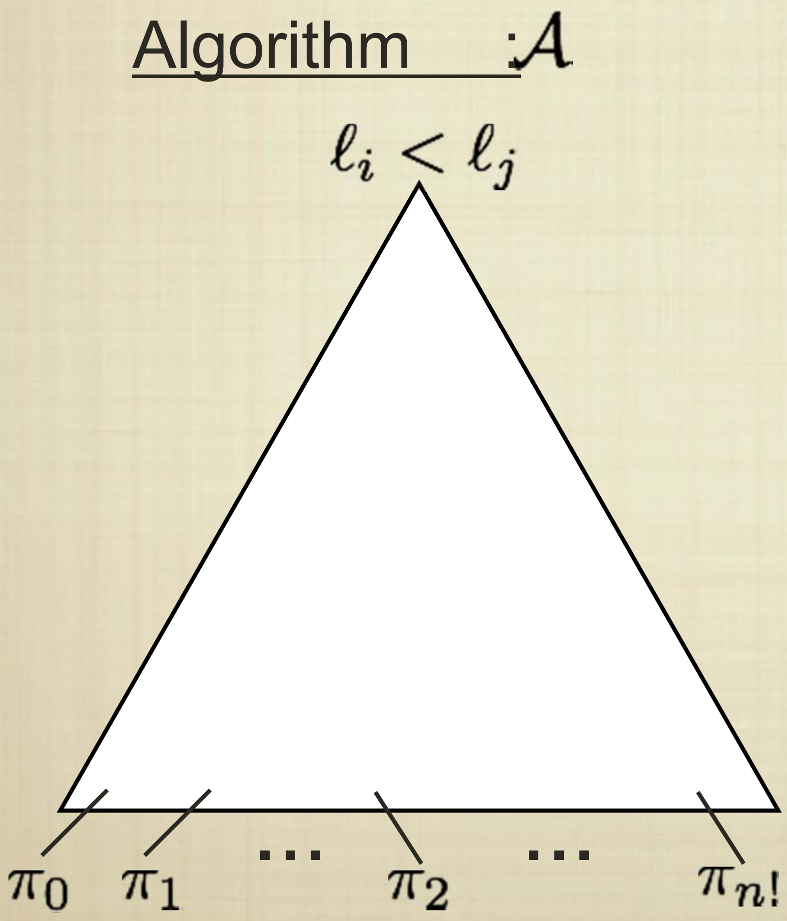
Algorithm : $\mathcal{A}$



Lower Bound for Sorting

So, $n! \geq (n/2)^{n/2}$

For a list L with n items, let the possible permutations be $\pi_0, \pi_1, \ldots, \pi_{n!}$. Any sorting algorithm must be able to "reach" all of these permutations by making a sequence of comparisons. The corresponding decision tree is:



$n! \leq$ # leaves $\leq 2^{\text{height}}$

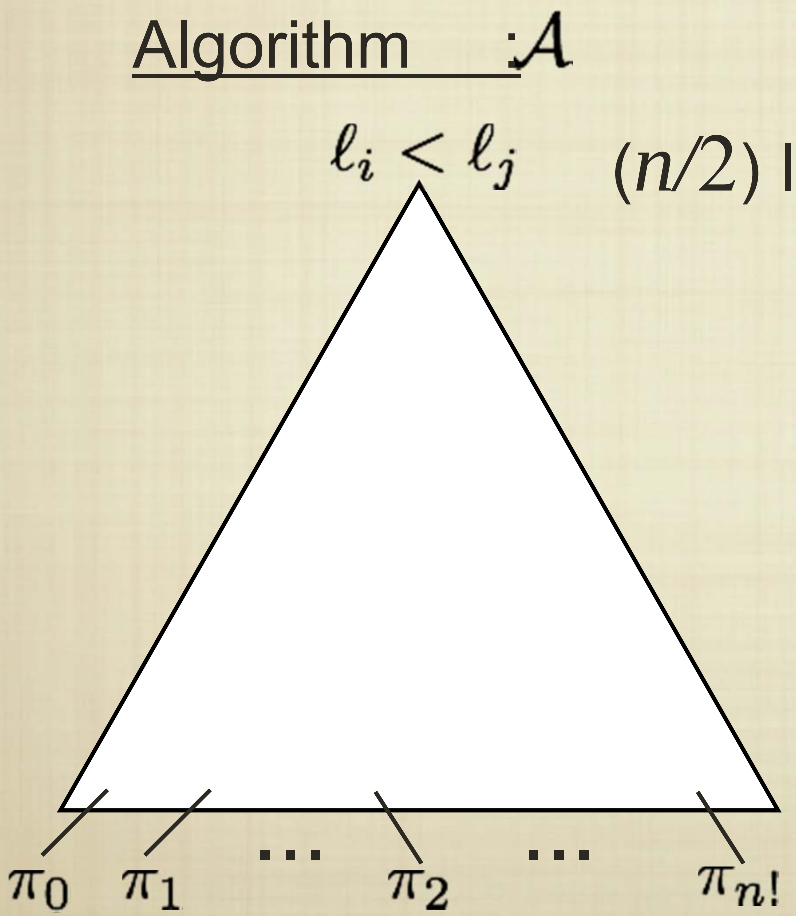
So, $n! \leq 2^{\text{height}}$

This is equivalent to:
$\log n! \leq$ height

So:
$\log (n/2)^{n/2} \leq \log n! \leq$ height

Lower Bound for Sorting

For a list L with n items, let the possible permutations be $\pi_0, \pi_1, \ldots, \pi_{n!}$. Any sorting algorithm must be able to "reach" all of these permutations by making a sequence of comparisons. The corresponding decision tree is:



So:

$$(n/2) \log (n/2) = \log (n/2)^{n/2} \leq \log n! \leq \text{height}$$

Therefore:

$$(n/2) \log (n/2) \leq \text{height}$$

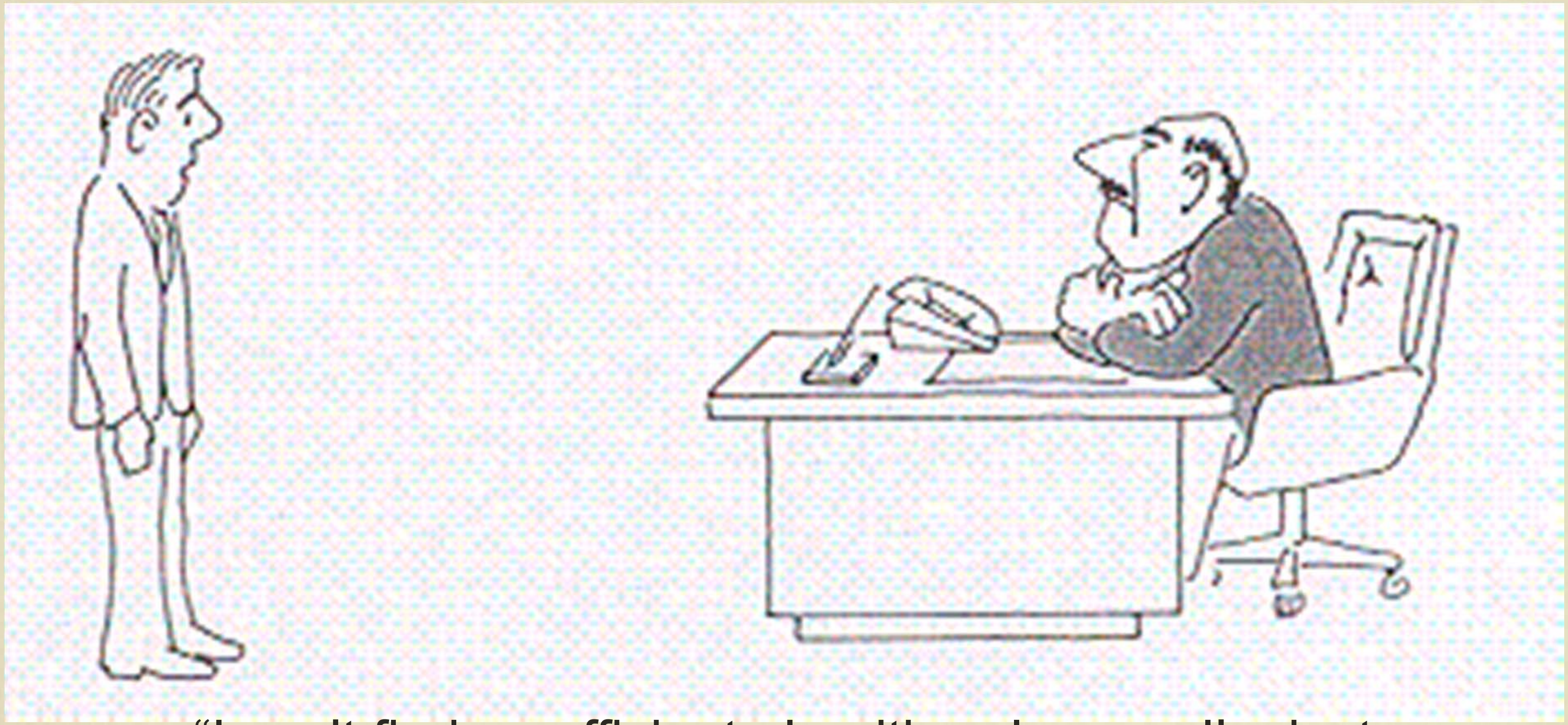
Or equivalently:

$$(1/2)\, n \log n - (n/2) \leq \text{height}$$

What does this tell us about Merge Sort?

The Power of Lower Bounds

Exponential-time Algorithm, Trivial lower bound



“I can’t find an efficient algorithm, I guess I’m just dumb.”

[Garey and Johnson ’79]

The Power of Lower Bounds

Matching Exponential-time bounds

“I can’t find an efficient algorithm, because no such algorithm is possible.”

[Garey and Johnson ’79]

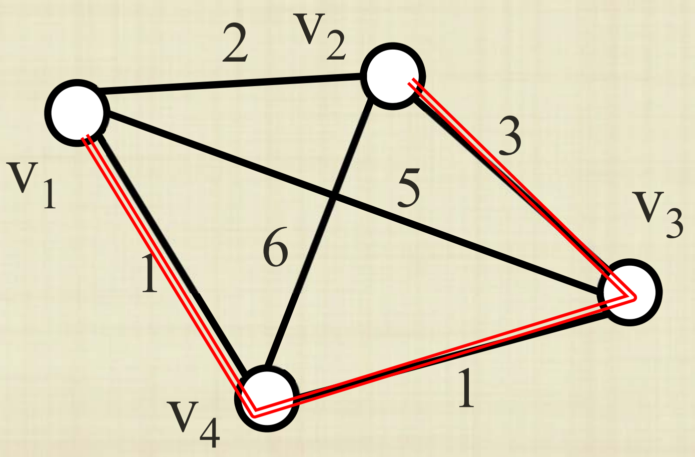
Some Interesting But Hard Problems

- Traveling Salesperson Problem (TSP)
- Satisfiability Problem (SAT)
- Independent Set (IS)

Traveling Salesperson (TSP)

- Suppose you are given a graph G on n vertices, with weighted edges, and a constant C.

- Can you find a path of total length at most C that visits every vertex exactly once?

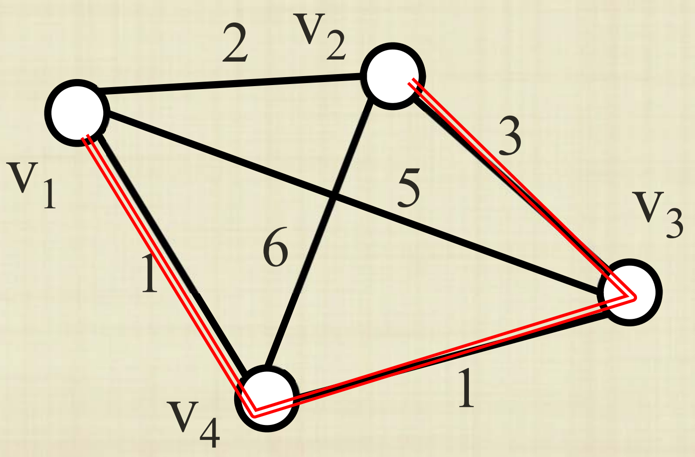


C=5

This problem has many real-world applications in planning and logistics, and it has many applied variants as well.

Traveling Salesperson (TSP)

- Suppose you are given a graph G on n vertices, with weighted edges, and a constant C.
- Can you find a path of total length at most C that visits every vertex exactly once?

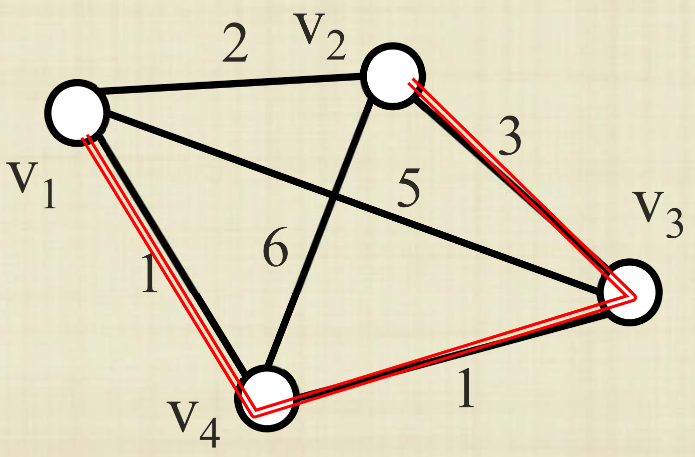


C=5

The simple algorithm would be to just check the length of every possible path. How many are there?

Traveling Salesperson (TSP)

- Suppose you are given a graph G on n vertices, with weighted edges, and a constant C.
- Can you find a path of total length at most C that visits every vertex exactly once?

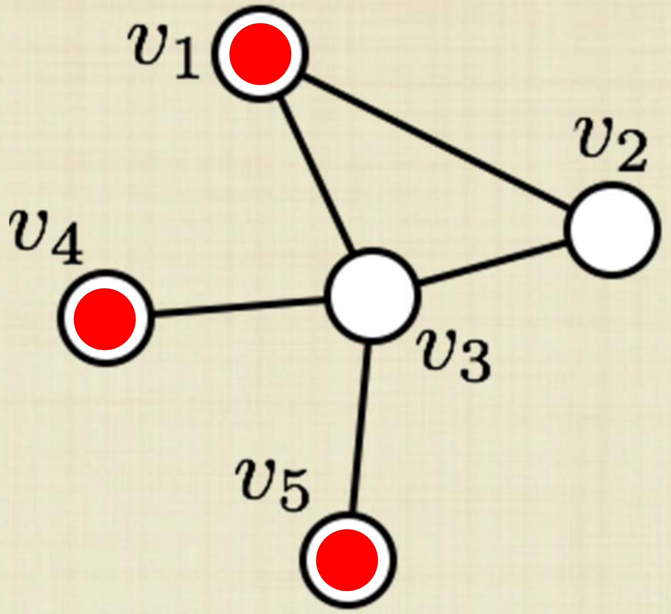


C=5

Sadly, we don't know how do much better than to check the n! possible paths; we don't have matching lower bounds either.

Independent Set (IS)

- Suppose you are given a graph G on n vertices, and a constant k. An <u>independent set</u> in G is a set of vertices that do not share an edge.



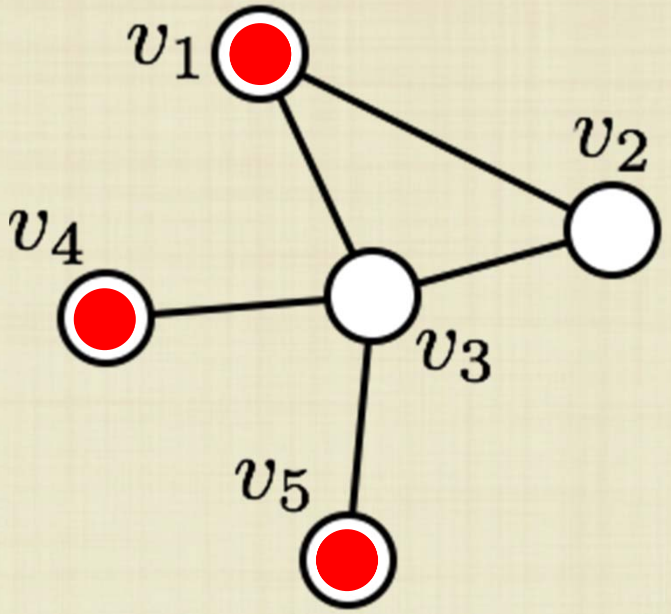
k=3

Does G have an independent set of size at least k?

This problem has applications to problems where we need to avoid collisions (e.g. scheduling, coloring).

Independent Set (IS)

- Suppose you are given a graph G on n vertices, and a constant k. An <u>independent set</u> in G is a set of vertices that do not share an edge.

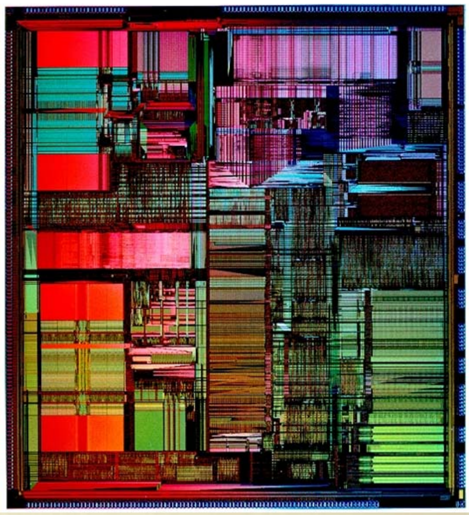


k=3

Does G have an independent set of size at least k ?

Unfortunately, we don't know how to check this, other than to examine all vertex sets of size at least k. How many sets are there?

Satisfiability (SAT)



- Suppose you are given a logical formula:

$$F(x, y, z) = (x \vee \neg y \vee z) \wedge (x \vee y \vee z)$$

Can we find an assignment of the variables that makes F true?

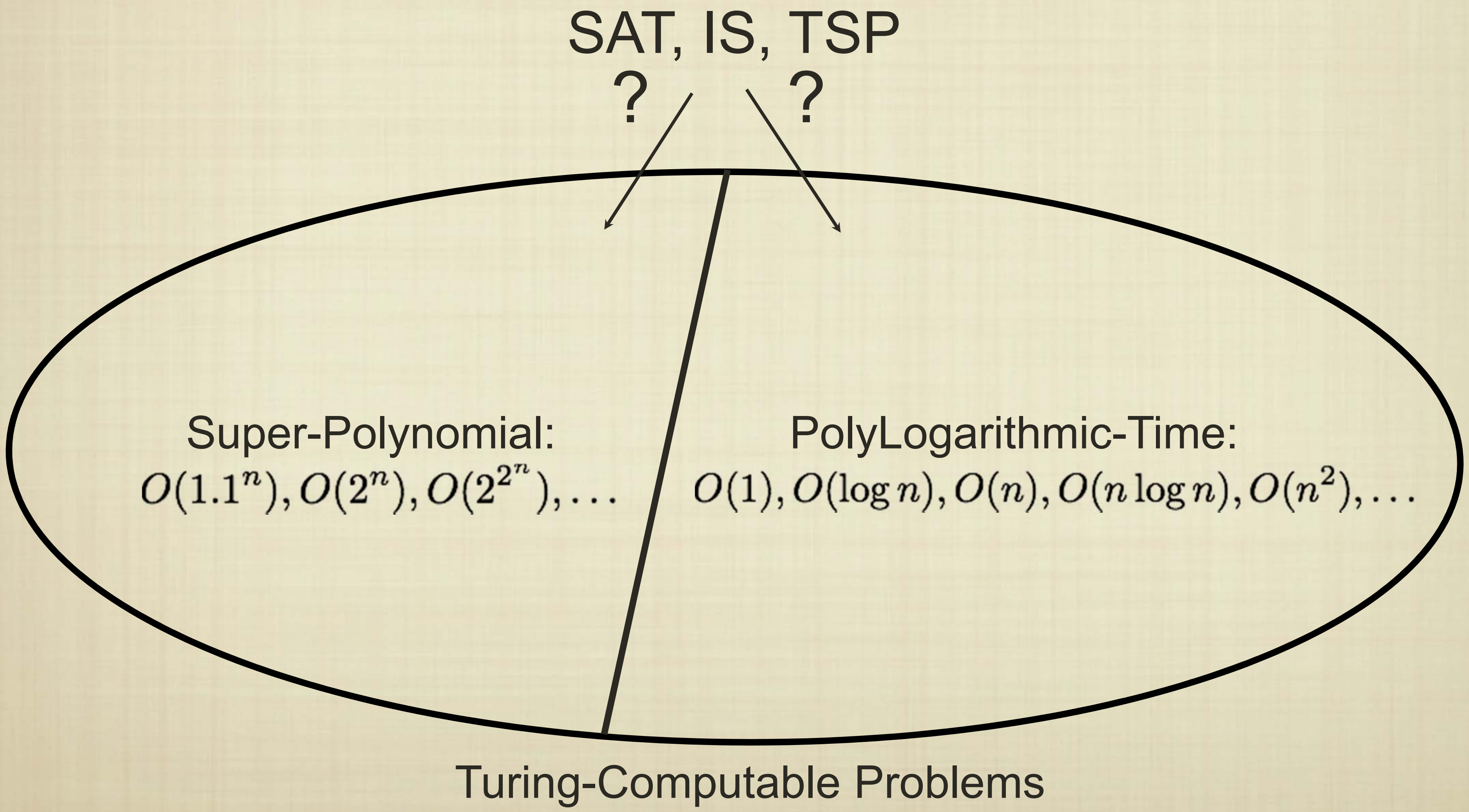
This has important applications in computer chip verification.

We can think of F as a circuit; one way to test whether this circuit ever produces a “1” is to just try all inputs.

Sadly, the fastest algorithms essentially do this. Is there a matching lower bound for SAT?

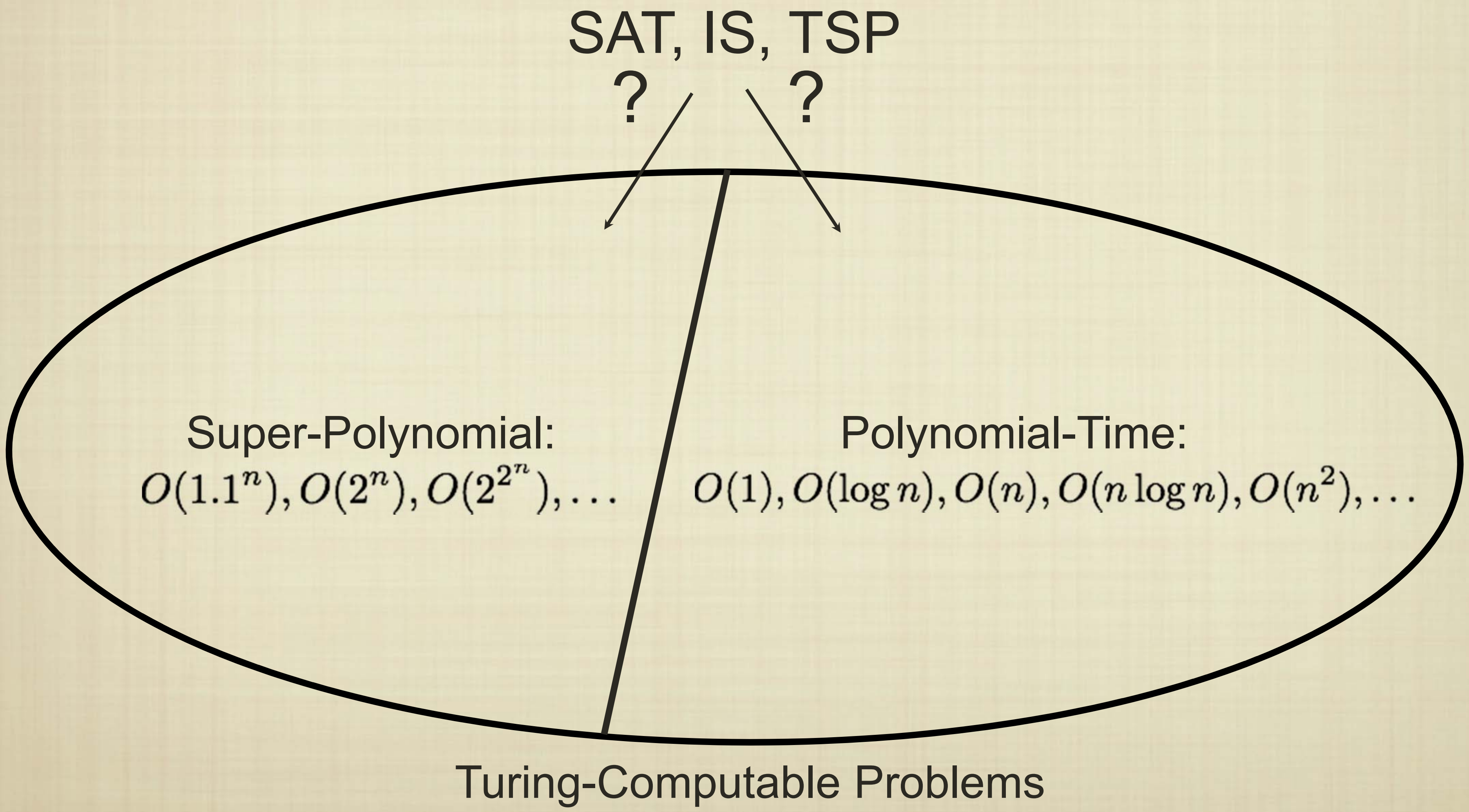
Difficulty of SAT, IS and TSP

- Where do SAT, IS and TSP fall into our two categories of problems? Are we just “dumb”?


SAT, IS, TSP
?
?
Super-Polynomial:
$O(1.1^n), O(2^n), O(2^{2^n}), \ldots$
PolyLogarithmic-Time:
$O(1), O(\log n), O(n), O(n \log n), O(n^2), \ldots$
Turing-Computable Problems

Difficulty of SAT, IS and TSP

- SAT, IS and TSP have evaded efficient algorithms for about 40 years. What have we been doing all this time?


SAT, IS, TSP
?
?
Super-Polynomial:
$O(1.1^n), O(2^n), O(2^{2^n}), \ldots$
Polynomial-Time:
$O(1), O(\log n), O(n), O(n \log n), O(n^2), \ldots$
Turing-Computable Problems

Is Checking Easier than Solving?

Potential solution | Input

SAT

$x=0, y=1, z=1$

$x=1, y=1, z=1$

$x=1, y=0, z=1$

$$F(x,y,z) = (x \lor \neg y \lor z) \land (x \lor y \lor z)$$

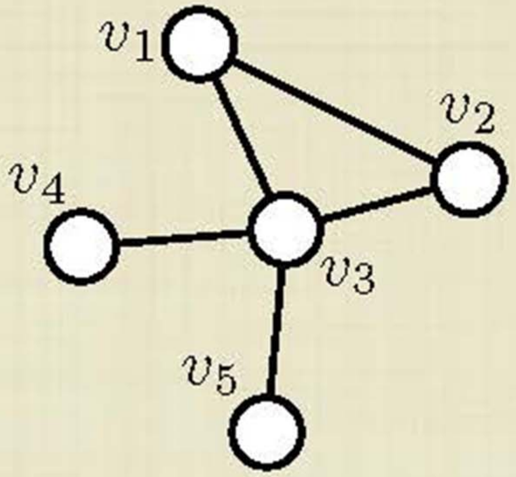
IS

$k=3$

~~$\{v_1, v_2, v_4\}$~~

$\{v_1, v_4, v_5\}$

$\{v_2, v_4, v_5\}$



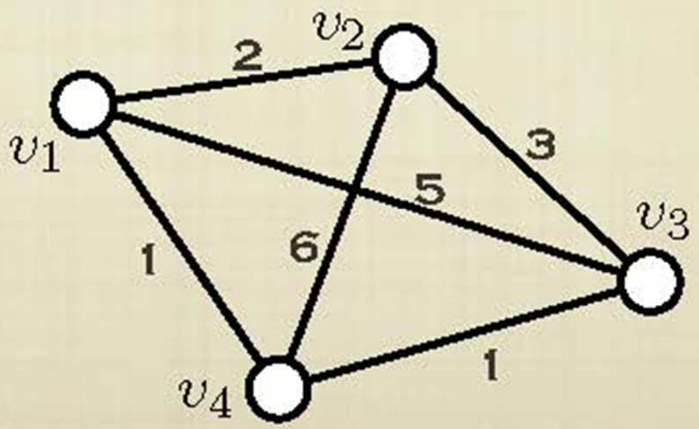
TSP

$C=5$

~~$\langle v_1, v_4, v_2, v_3\rangle$~~

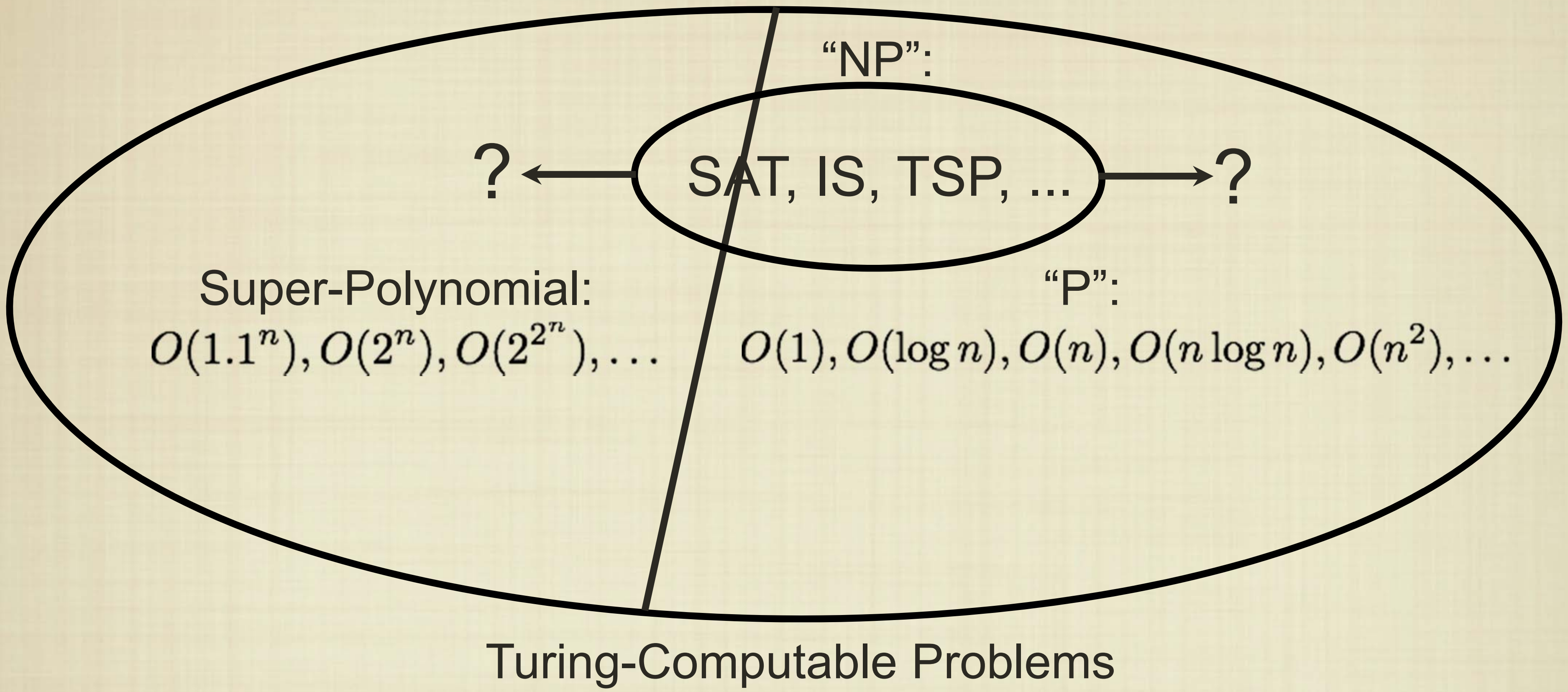
~~$\langle v_1, v_2, v_3, v_4\rangle$~~

$\langle v_1, v_4, v_3, v_2\rangle$



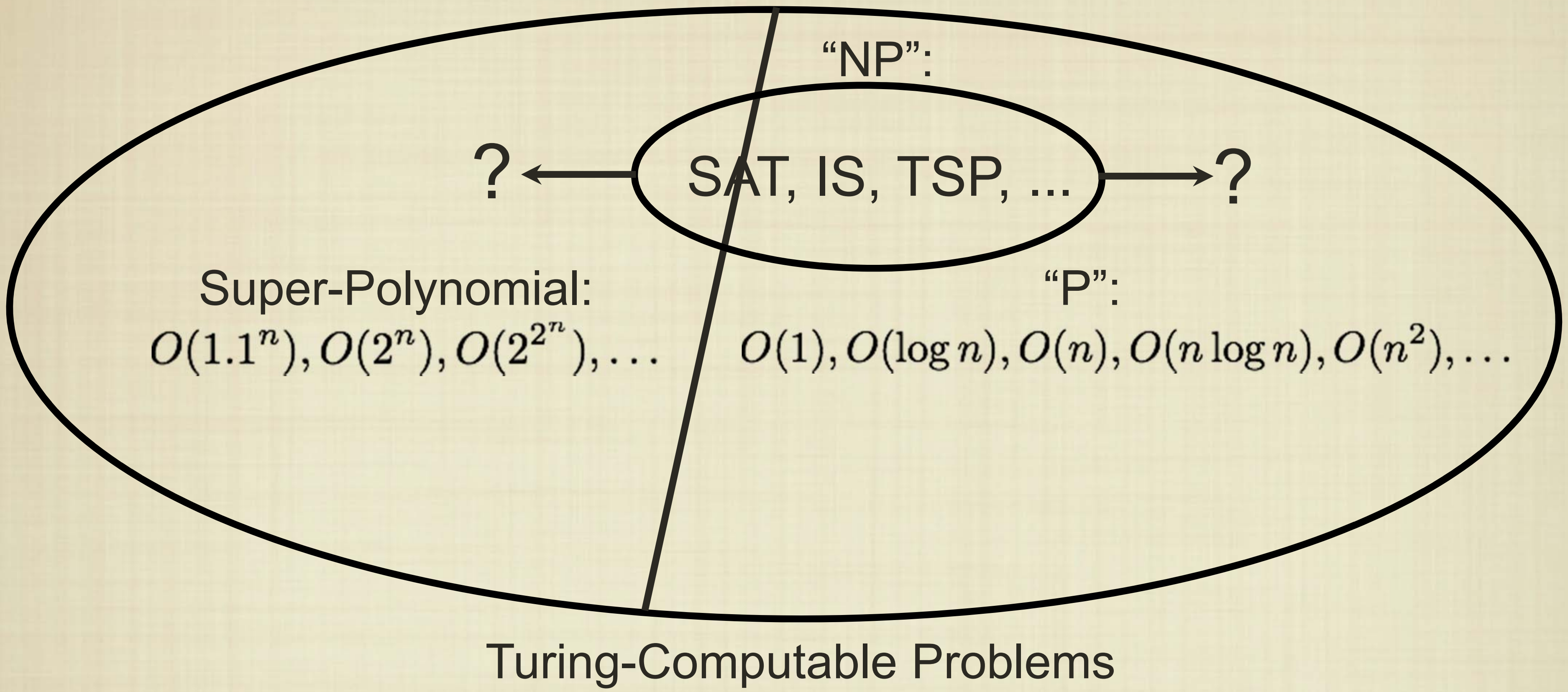
SAT, IS, and TSP solutions can be checked in linear time.

The P = NP Question


"NP":
SAT, IS, TSP, ...
?
?
Super-Polynomial:
$O(1.1^n), O(2^n), O(2^{2^n}), \ldots$
"P":
$O(1), O(\log n), O(n), O(n \log n), O(n^2), \ldots$
Turing-Computable Problems

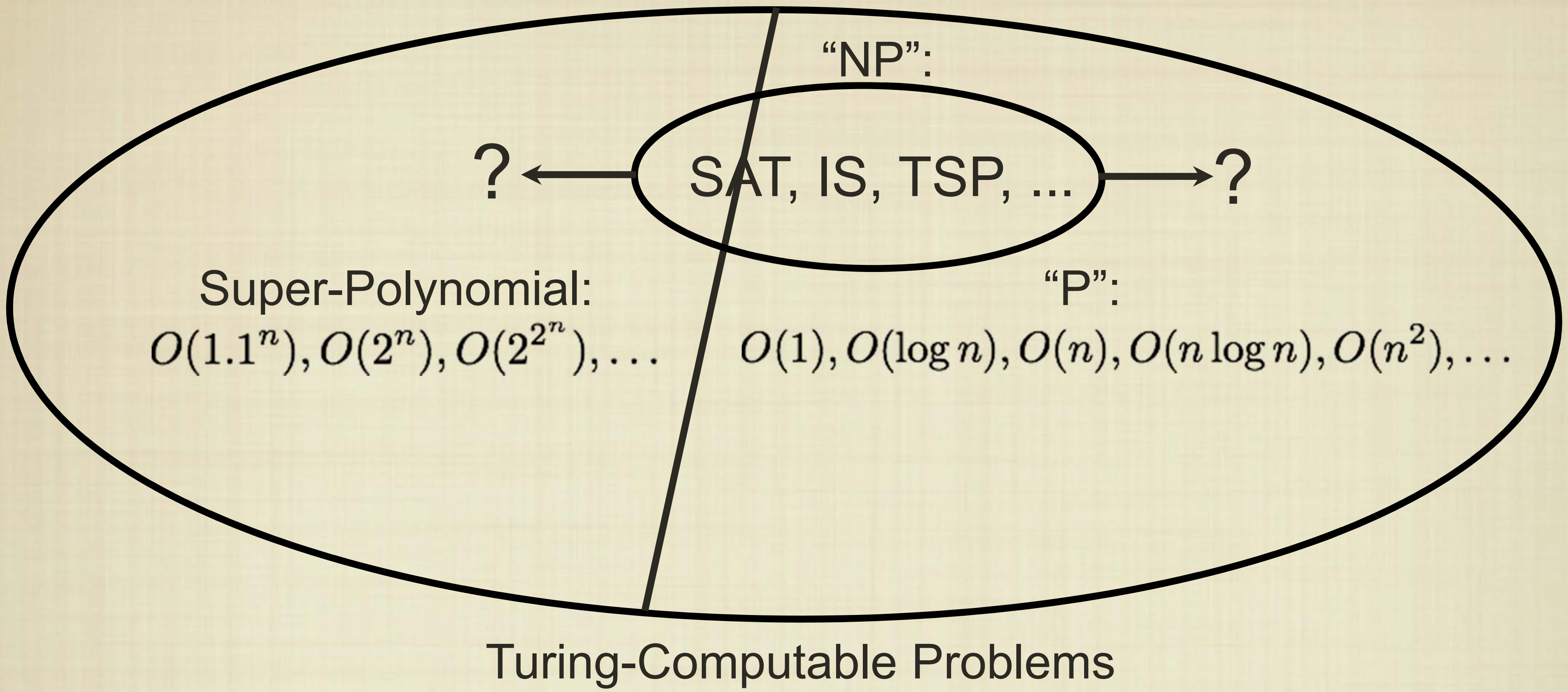
"NP" is defined as the class of problems that, if given a solution, it can be checked in polynomial time. There are thousands of such problems, but we don't know how to solve any of them in less than exponential time...

The P = NP Question


"NP":
SAT, IS, TSP, ...
?
?
Super-Polynomial:
$O(1.1^n), O(2^n), O(2^{2^n}), \ldots$
"P":
$O(1), O(\log n), O(n), O(n \log n), O(n^2), \ldots$
Turing-Computable Problems

Do problems that have polynomial-time checkable solutions, also have polynomial-time algorithms?

The P = NP Question

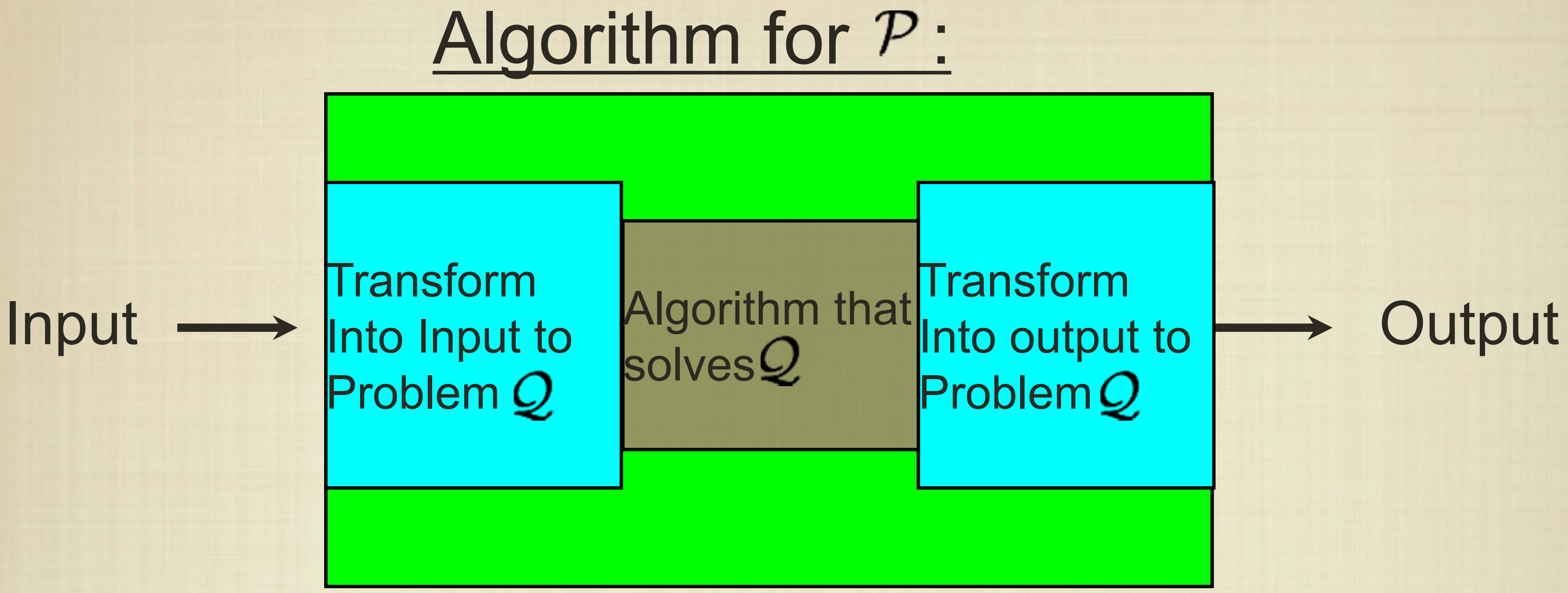


"NP":
SAT, IS, TSP, ...
?
?
Super-Polynomial:
$O(1.1^n), O(2^n), O(2^{2^n}), \ldots$
"P":
$O(1), O(\log n), O(n), O(n \log n), O(n^2), \ldots$
Turing-Computable Problems

Not a lot of progress has been made on this question...

One interesting fact we have established is that the easy-to-check problems all appear to have the same difficulty.

Reducibility:

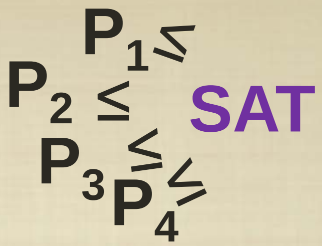

Algorithm for $\mathcal{P}$:
Input
Transform Into Input to Problem $\mathcal{Q}$
Algorithm that solves $\mathcal{Q}$
Transform Into output to Problem $\mathcal{Q}$
Output

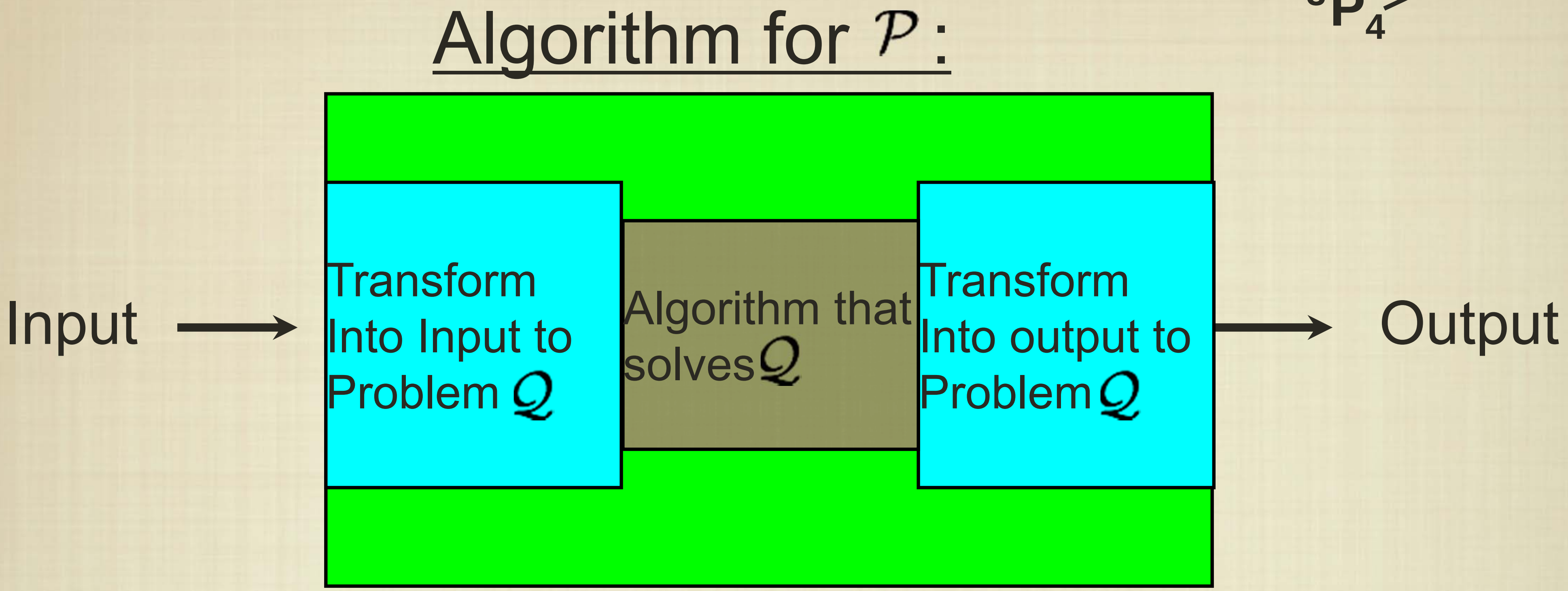
Compute Minimum $\mathcal{P} \leq \mathcal{Q}$ Sorting

One way to show that problem $\underbrace{\mathcal{Q} \text{ is at least as hard as problem } \mathcal{P}}$ is to show that we can use $\mathcal{Q}$ to solve $\mathcal{P}$.

The advantage is that we only need the "reduction", and not the full algorithm for $\mathcal{P}$.

Reducibility




Algorithm for $\mathcal{P}$:
Input
Transform Into Input to Problem $\mathcal{Q}$
Algorithm that solves $\mathcal{Q}$
Transform Into output to Problem $\mathcal{Q}$
Output

There are thousands of problems in NP that all “reduce” to **SAT**.

Cook (1971) and Levin (1973) showed that SAT is NP-complete; if you can solve SAT in polynomial-time, then all problems in NP can be solved in polynomial-time.

Independent Set is as hard as SAT

$P_1 \leq P_2 \leq P_3 \leq P_4 \leq$

SAT≤ IS

We can show that the Independent Set problems is as hard as solving SAT. To do this, we convert an input to SAT into an Independent Set problem.

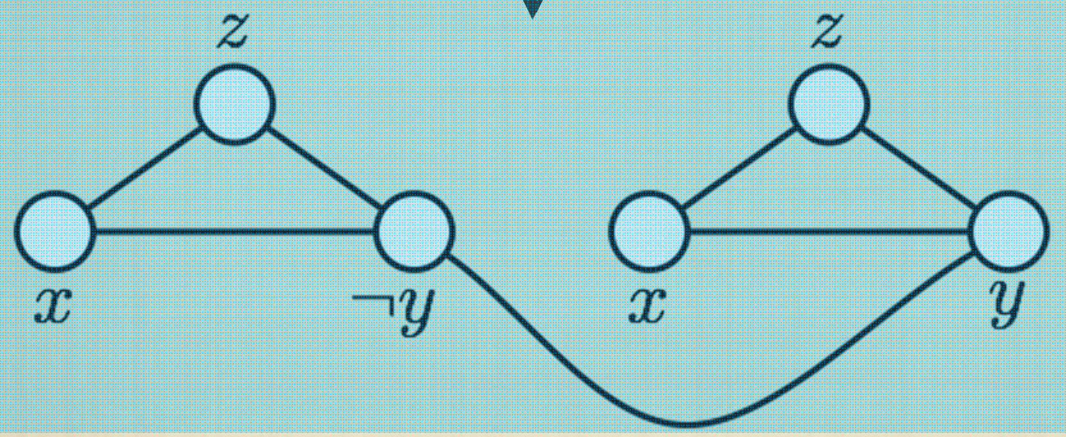
SAT

$$F(x, y, z) = (x \vee \neg y \vee z) \wedge (x \vee y \vee z)$$

Formula with n variables

$\big\downarrow$ polynomial-time

IS



Does this graph have an independent set with n vertices?

Given a formula F, we can construct an input to IS whose solution tells us which variables to assign to true to obtain a satisfying assignment.

The Power of Lower Bounds

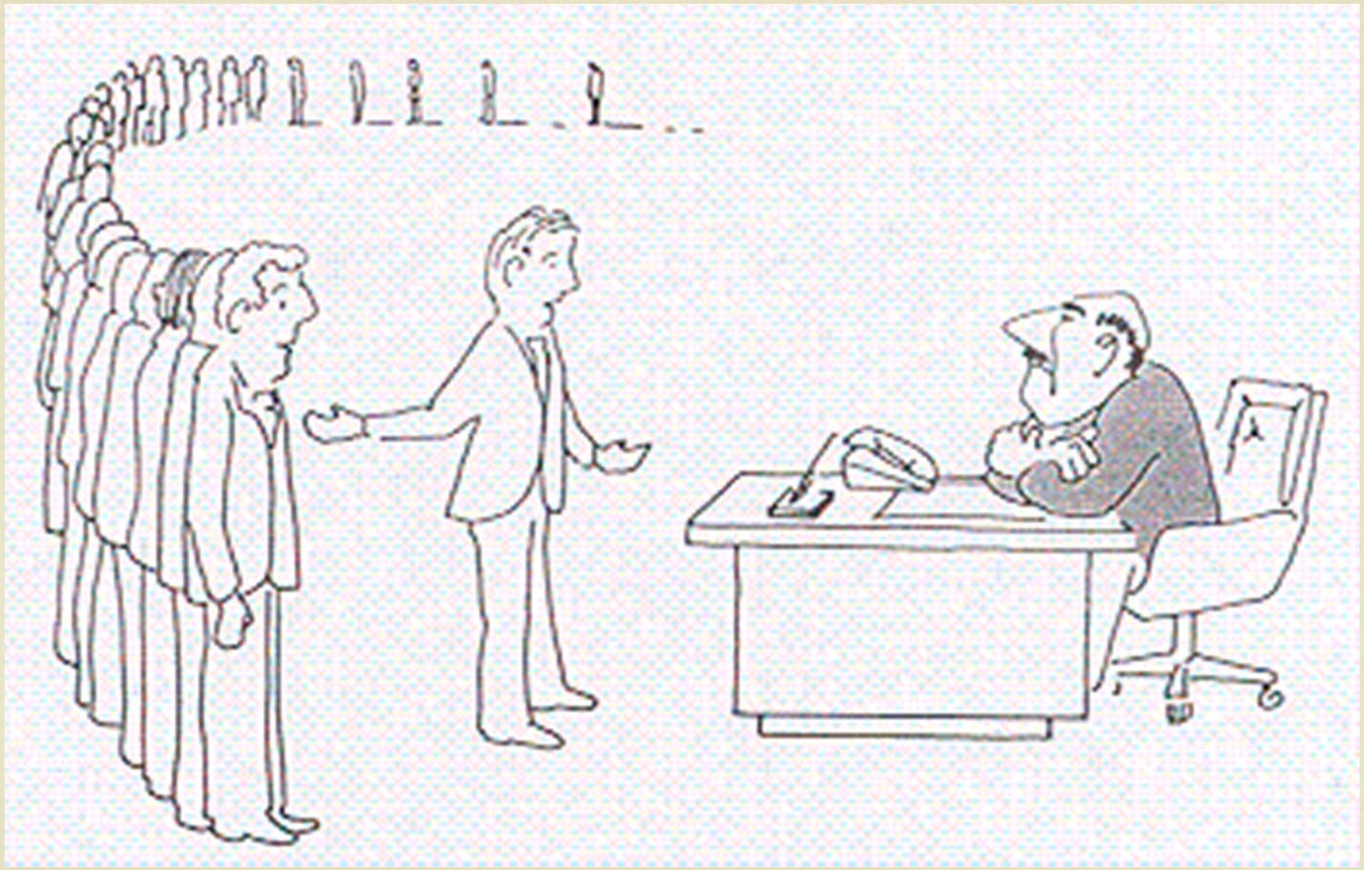
Matching Exponential-time bounds

“I can’t find an efficient algorithm, because no such algorithm is possible.”

[Garey and Johnson ’79]

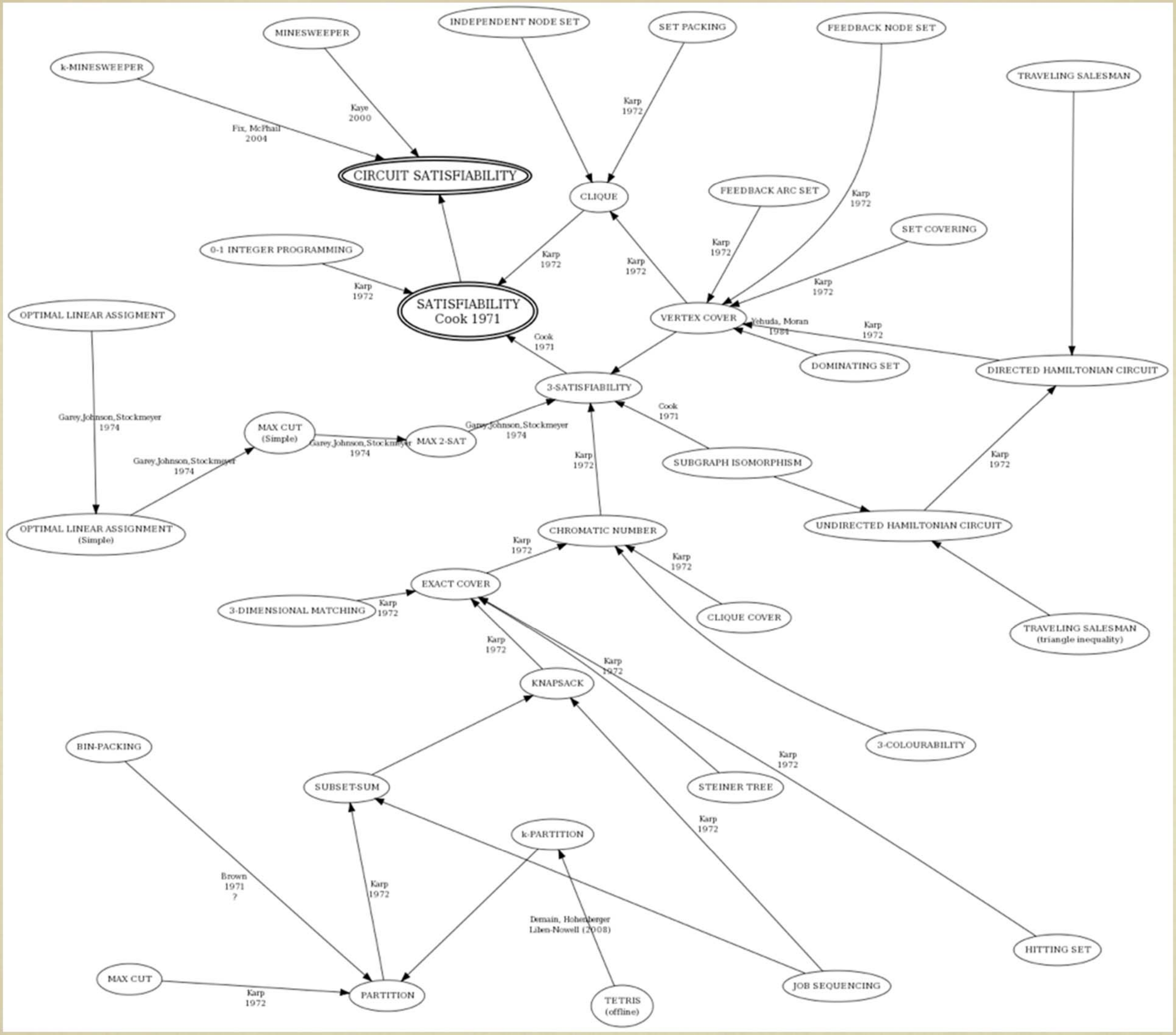
State Of The Art

Reductions and NP-Completeness



“I can’t find an efficient algorithm, but neither can all these smart people.”

[Garey and Johnson ’79]


MINESWEEPER
INDEPENDENT NODE SET
SET PACKING
FEEDBACK NODE SET
k-MINESWEEPER
TRAVELING SALESMAN
Kaye 2000
Karp 1972
Fix, McPhail 2004
CIRCUIT SATISFIABILITY
CLIQUE
FEEDBACK ARC SET
Karp 1972
SET COVERING
0-1 INTEGER PROGRAMMING
Karp 1972
Karp 1972
Karp 1972
Karp 1972
Karp 1972
SATISFIABILITY Cook 1971
OPTIMAL LINEAR ASSIGMENT
VERTEX COVER
Yehuda, Moran 1984
Karp 1972
Cook 1971
DOMINATING SET
DIRECTED HAMILTONIAN CIRCUIT
3-SATISFIABILITY
Cook 1971
Garey,Johnson,Stockmeyer 1974
MAX CUT (Simple)
MAX 2-SAT
Garey,Johnson,Stockmeyer 1974
Garey,Johnson,Stockmeyer 1974
Karp 1972
Karp 1972
Garey,Johnson,Stockmeyer 1974
SUBGRAPH ISOMORPHISM
UNDIRECTED HAMILTONIAN CIRCUIT
OPTIMAL LINEAR ASSIGNMENT (Simple)
CHROMATIC NUMBER
Karp 1972
Karp 1972
EXACT COVER
3-DIMENSIONAL MATCHING
Karp 1972
CLIQUE COVER
TRAVELING SALESMAN (triangle inequality)
Karp 1972
Karp 1972
KNAPSACK
BIN-PACKING
3-COLOURABILITY
Karp 1972
SUBSET-SUM
STEINER TREE
Karp 1972
k-PARTITION
Brown 1971 ?
Karp 1972
Demain, Hohenberger Liben-Nowell (2008)
HITTING SET
MAX CUT
JOB SEQUENCING
Karp 1972
PARTITION
TETRIS (offline)

State Of The Art

- Given the difficulty of these problems, researchers have looked for approximations and special cases that are solvable.
- We have also used distributed computing and “human computing” to help solve important optimization problems.
- Interestingly, difficult problems can be used for security purposes (factoring, CAPTCHAs, etc.).
- Researchers are also studying more powerful models of computation, such as quantum computing, to address these kinds of problems.

Using/Solving Hard Problems

Brute Force

A brute-force search can be easily "split up". Recruit the necessary computational resources and perform the search in parallel.

Approximation

We may not always need an exact answer. Devise algorithms which produce solutions that are not exact, but are guaranteed to be "close".

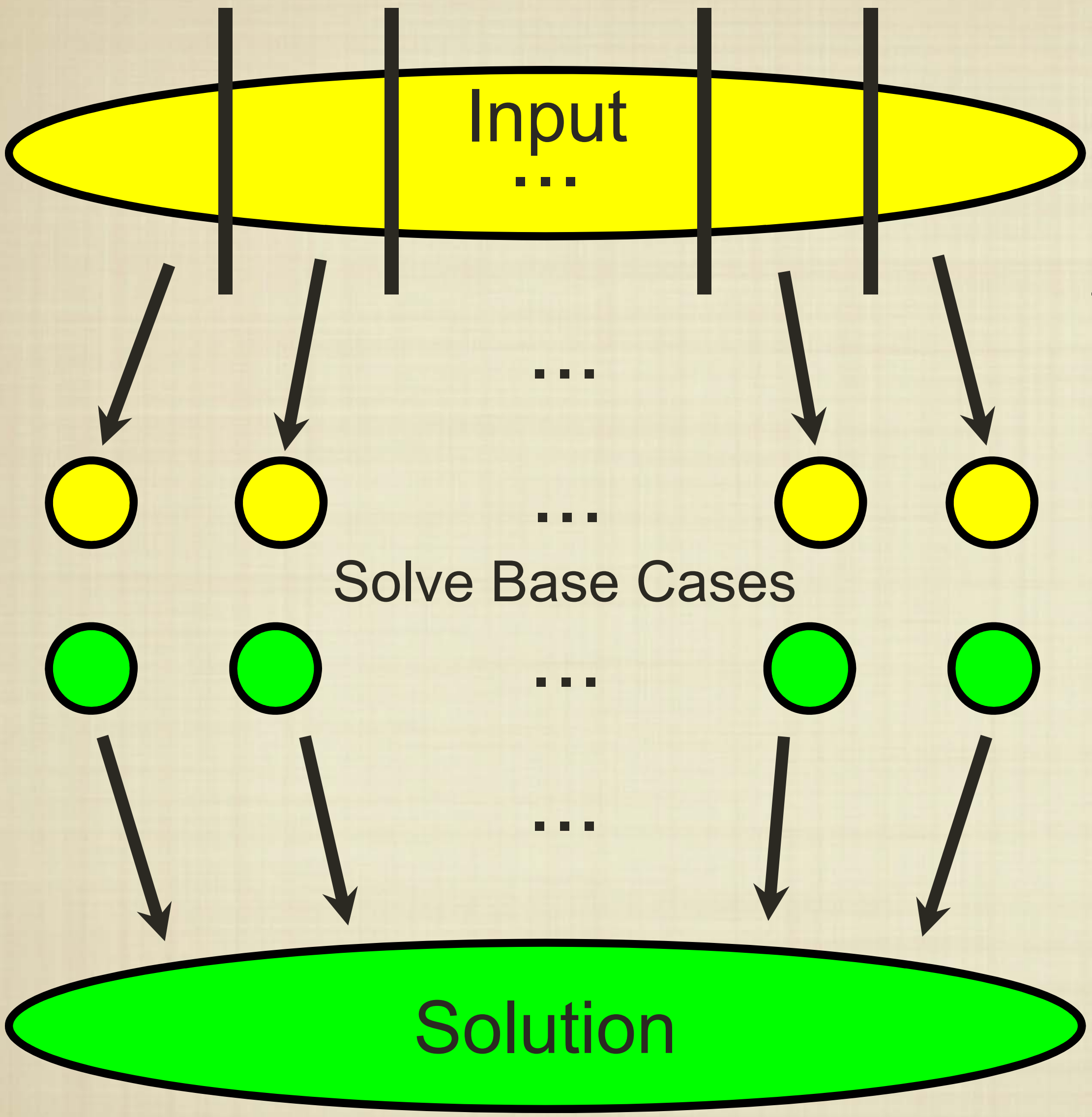
Security

Use intractability to protect sensitive information, by using the solution to a computationally difficult problem as a "key."

"Human Computing"

Humans appear to be good at coming up with solutions to some computationally difficult problems. Use human intuition to guide computational search.

“Divide-And-Conquer”



<u>Divide-and-Conquer</u>:

1. If the input is small enough, solve.

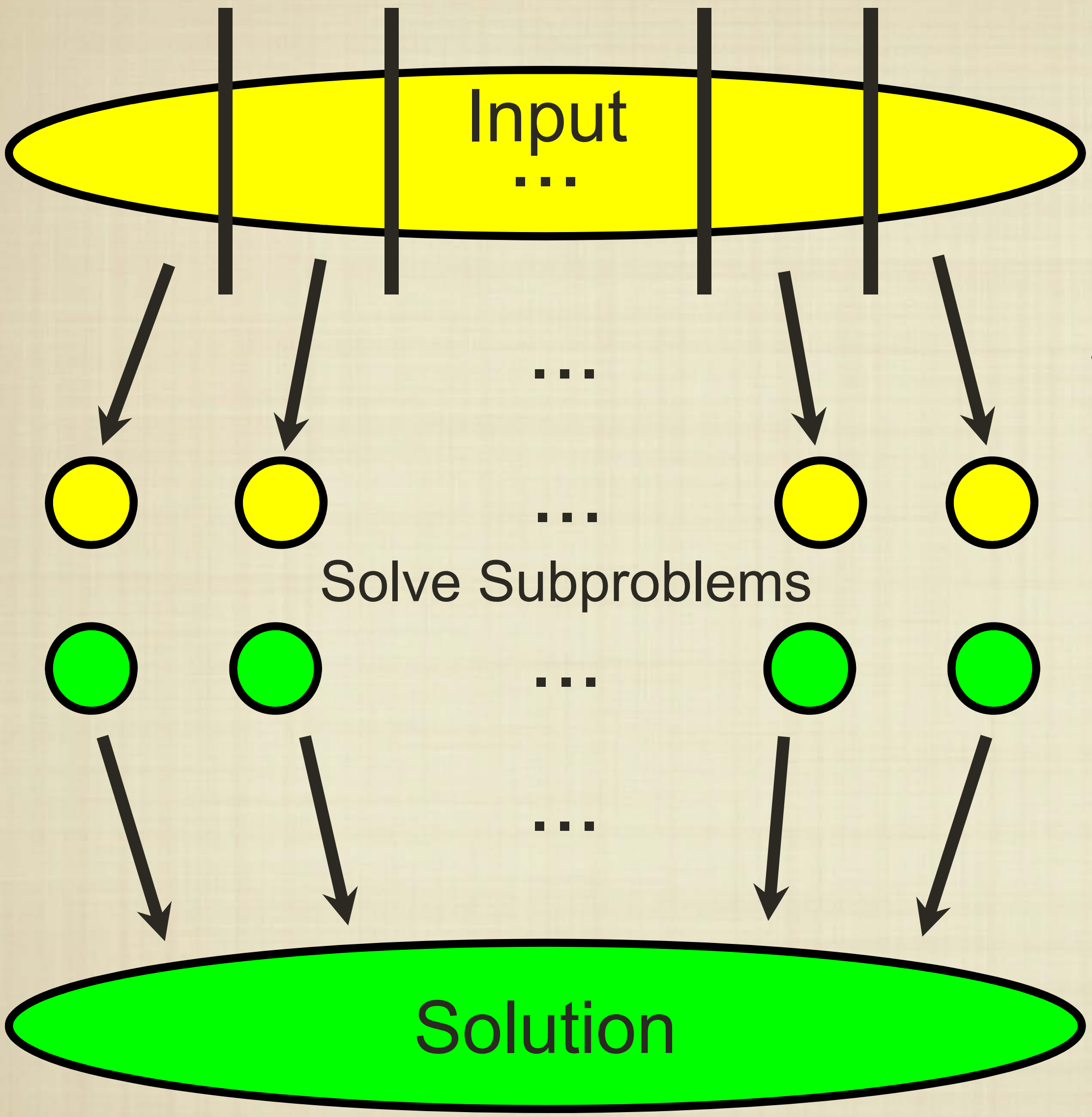
2. Otherwise, split input into parts.

3. Recursively solve each part.

4. Merge solutions.

Implementing these algorithms is easy because they are recursive.

“Distribute-And-Conquer”



<u>Distribute-and-Conquer</u>:

1. Split input into subproblems and distribute.

2. Solve each subproblem.

3. Collect and merge solutions.

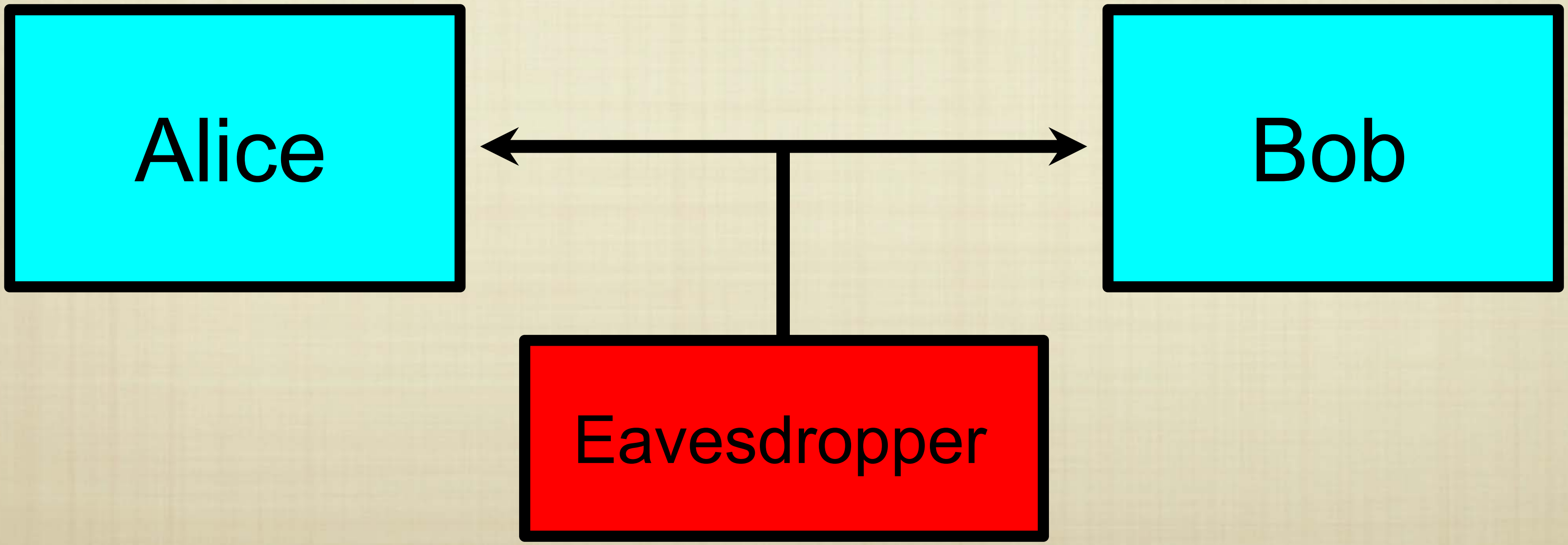
Implementing these algorithms is easy because intractable problems are often easy to parallelize. Why?

Using Difficulty for Security

Any hard-to-solve, but easy-to-check problem makes an ideal "security question".

Given a number n, the FACTORING problem asks whether n has a factor that is d or less.

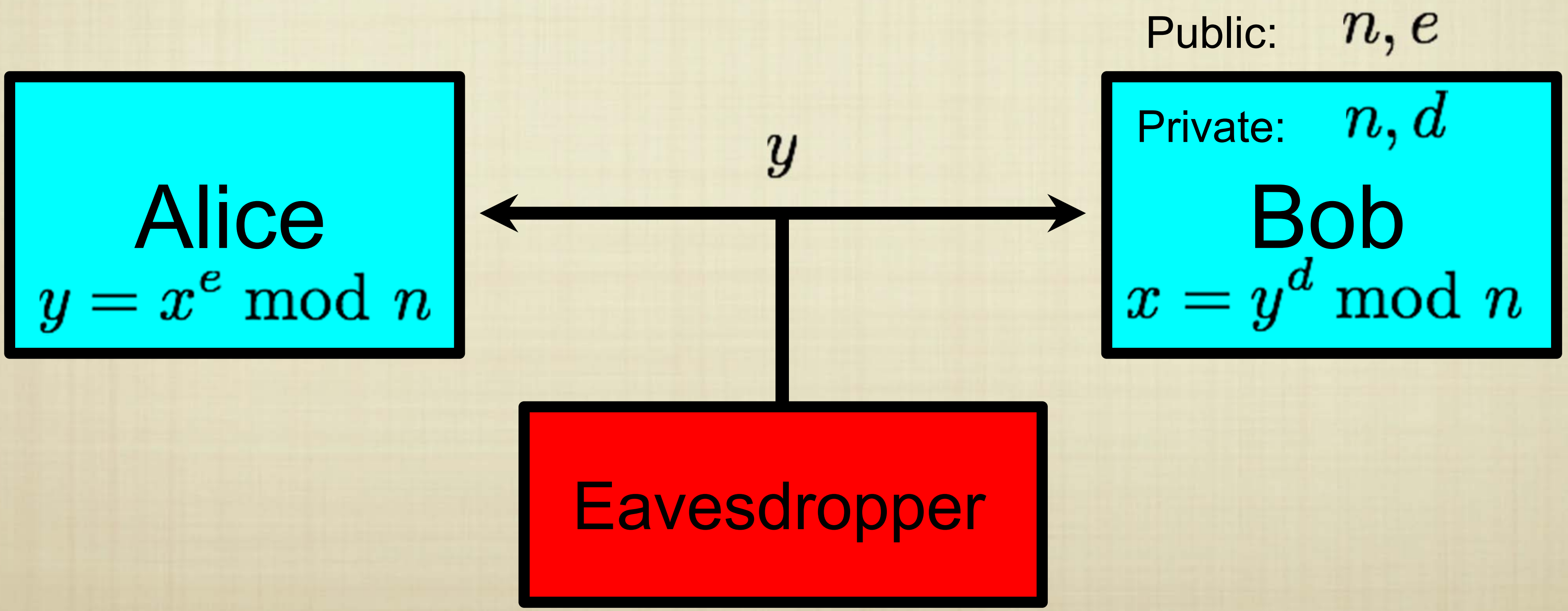
The difficulty of factoring is the basis for the RSA public-key cryptosystem.



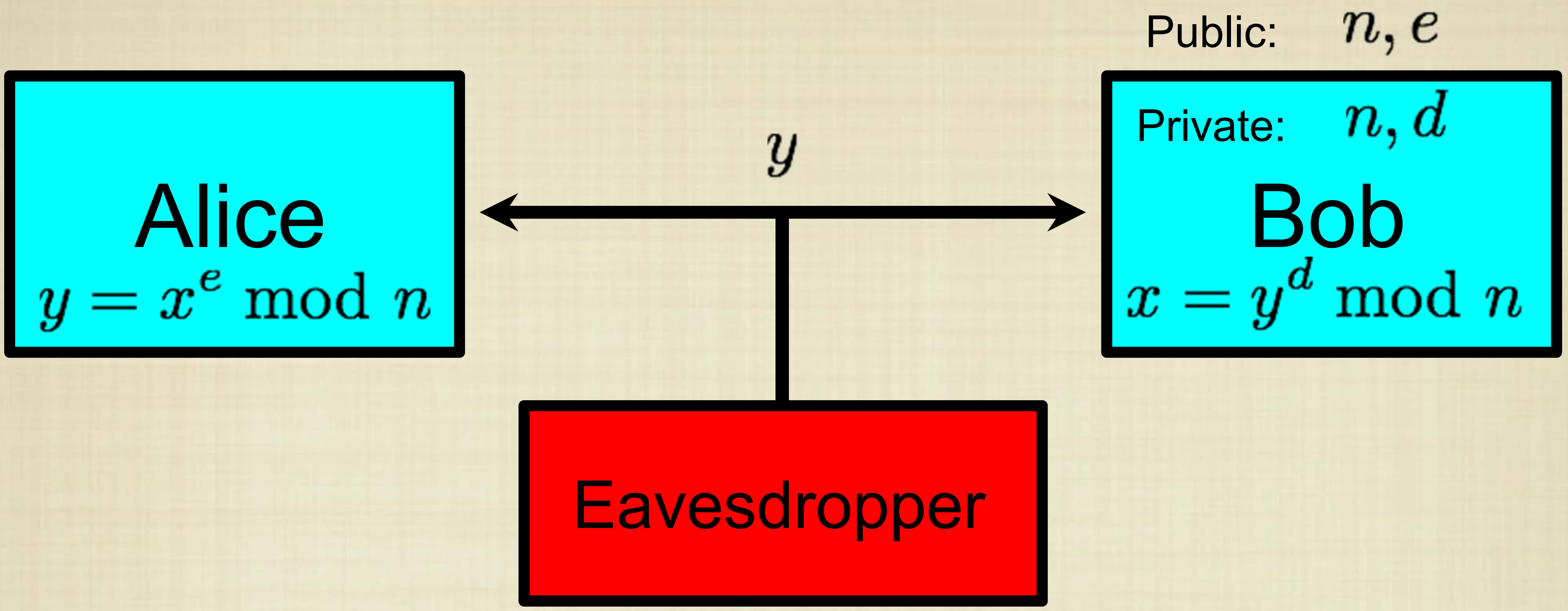
Public Key Cryptography

Suppose Alice wants to send $x \bmod n$ to Bob. First, Bob must compute two large primes p, q and $n = p \cdot q$. Then, Bob chooses a number e that shares no factors with $(p-1)(q-1)$. Finally, Bob computes d such that $d \cdot e = 1 \bmod (p-1)(q-1)$.

This scheme works because $x^{ed} = x \bmod n$.



Public Key Cryptography



To break this cryptosystem, we need to reconstruct the value of d . The only way to do this is to factor n .

If we could solve FACTORING, we could compute d efficiently. How?

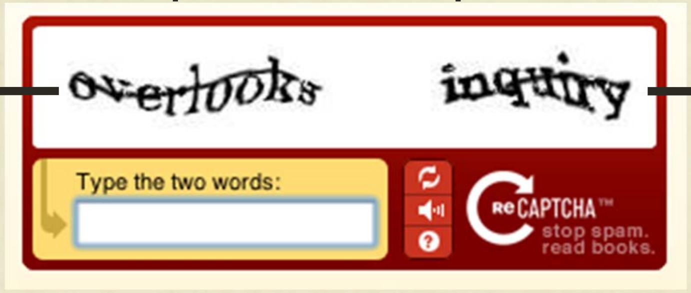
"Human Computing"

- Computationally intractable problems can be very useful, but what is we actually want to solve these problems, and not just use them for security?

- reCAPTCHAs combine these two goals:

Captcha/reCaptcha

Known ← → Unknown



- In 2003, humans spent over 9 billion hours playing video games. The "Games With a Purpose" (GWAPs) project seeks to solve intractable problem in artificial intelligence with games.

Recap: Frontiers of Computer Science

- How do we establish the difficulty of a given computational problem?
- How did we define the notion of an efficient algorithm? Why did we choose this definition?
- What is the ideal pair of facts we would like to know about a particular computational problem? Why? Is Merge Sort an optimal algorithm for sorting?
- What is the definition of NP? What is the P=NP question?
- What are the ways in which we deal with intractable problems?