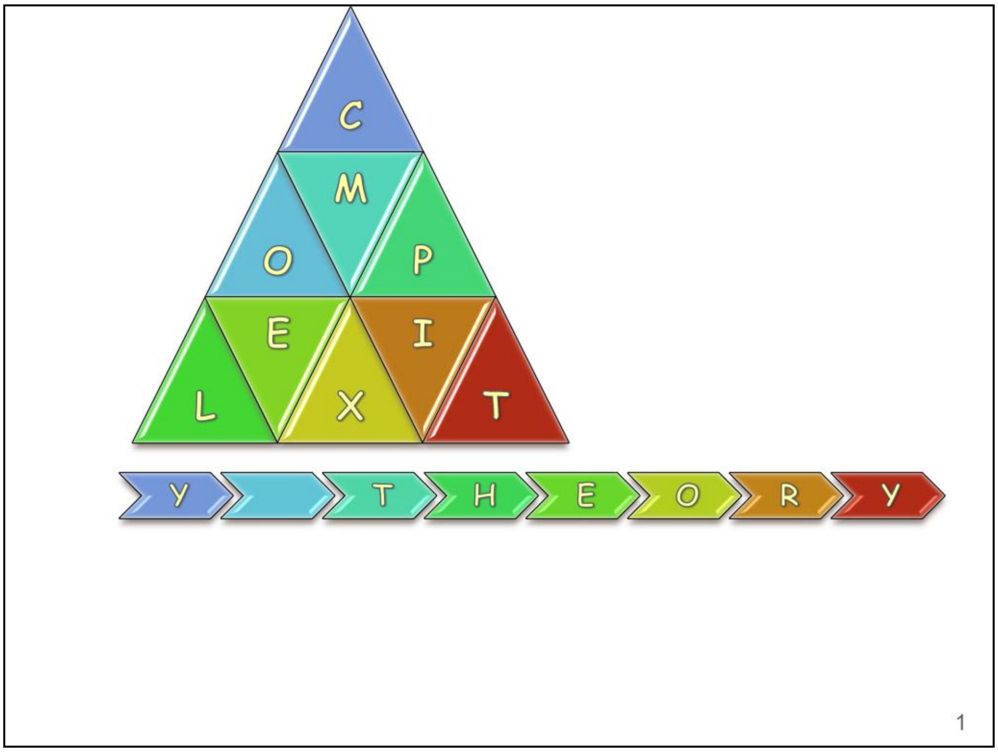
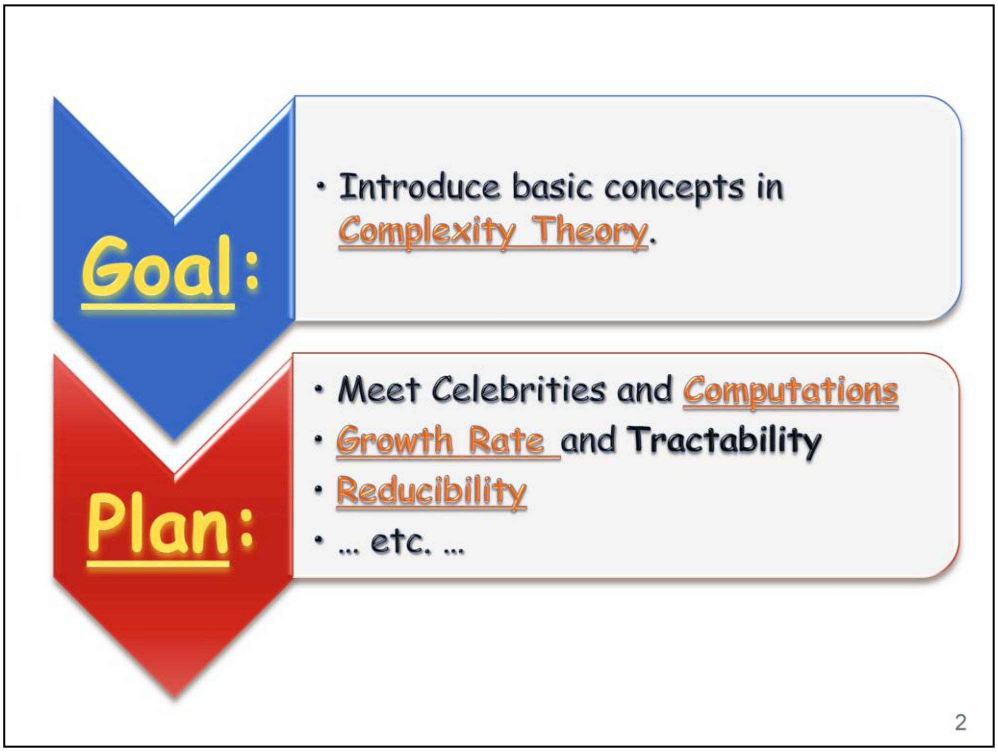
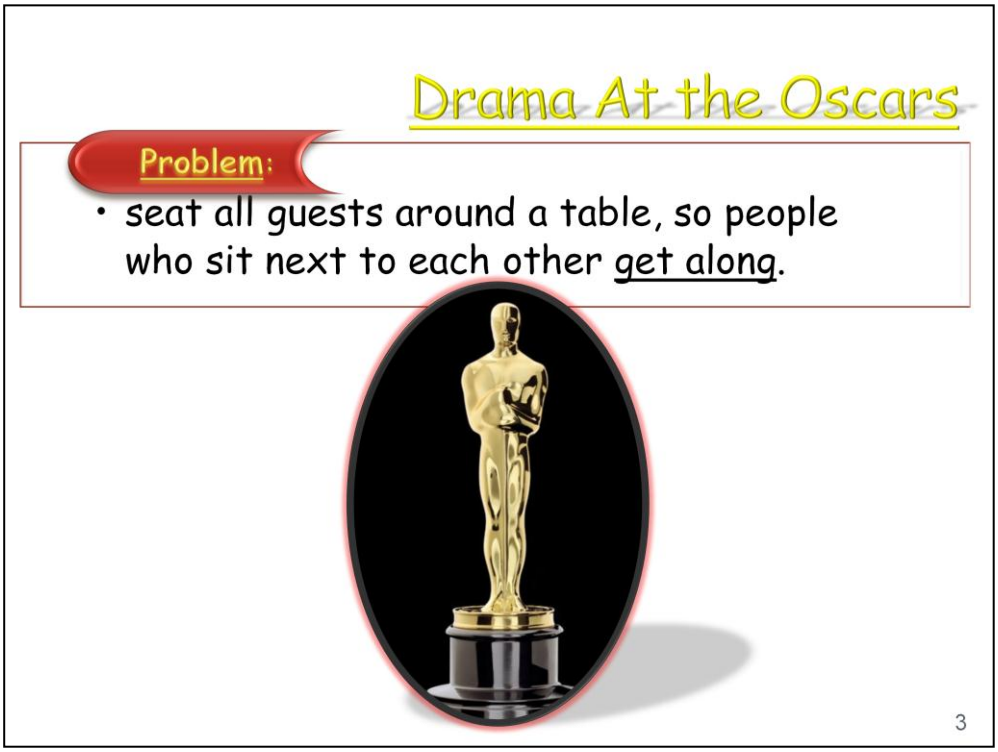


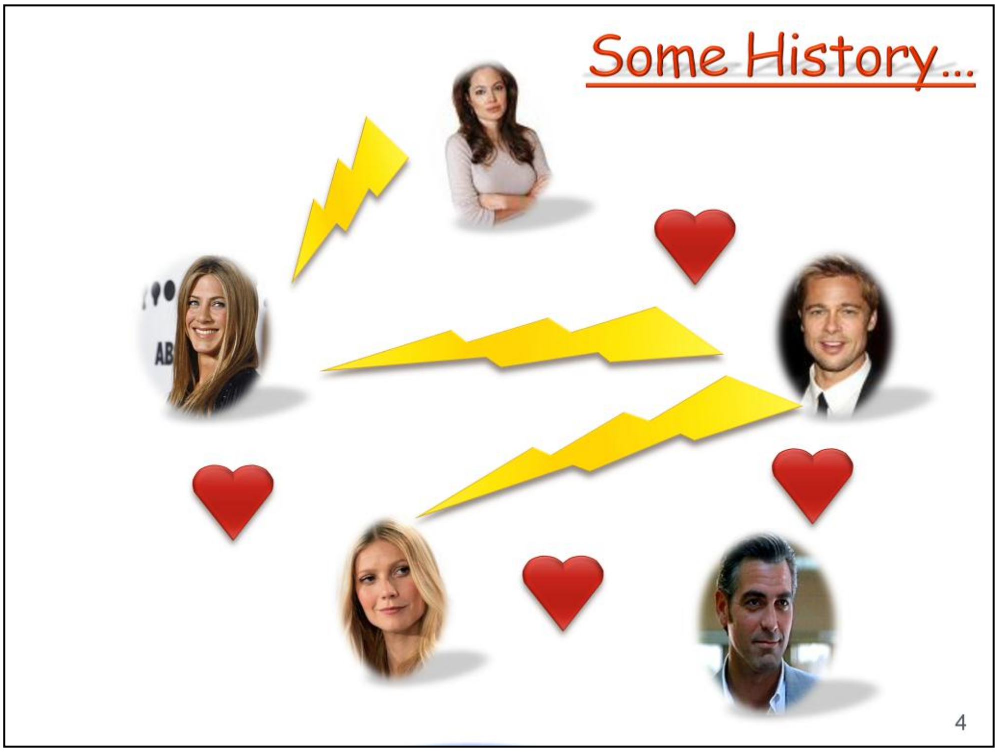

This course is about Complexity Theory,
in which we categorize computational problems to various classes
according to resources required for their solution.




This is the introductory lecture in which we will consider the basic motivations and methodology of the field.

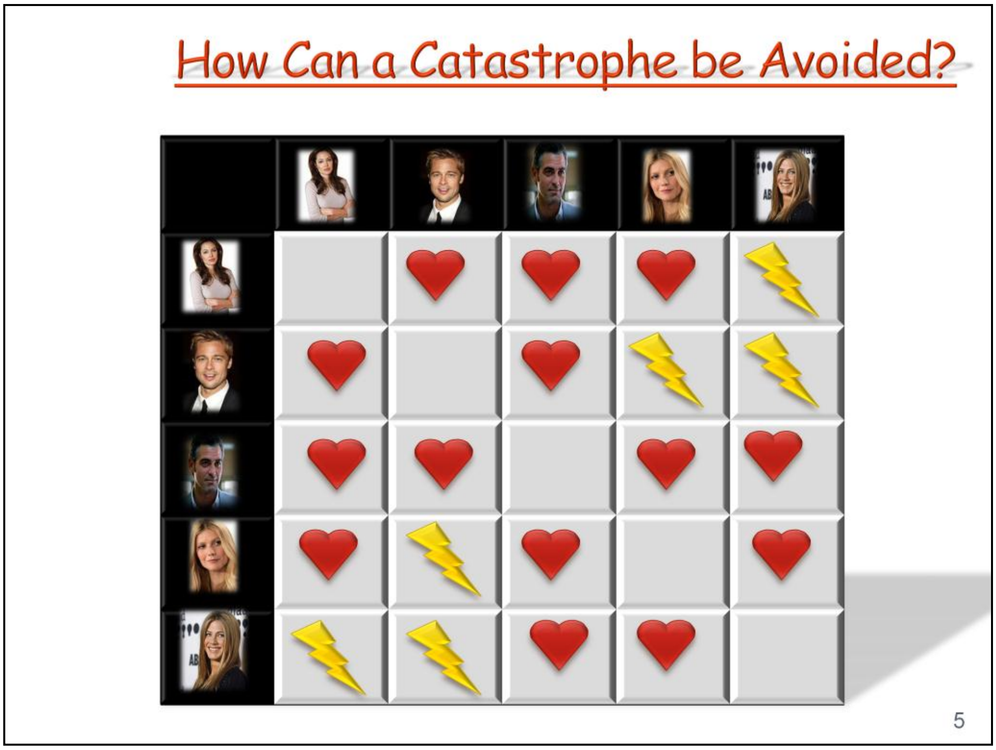



Say you're given a list of guests
who are to attend an event,
and the goal is to organize them
so they get along with each other.
You may use a computer for that purpose.




Here's an example:

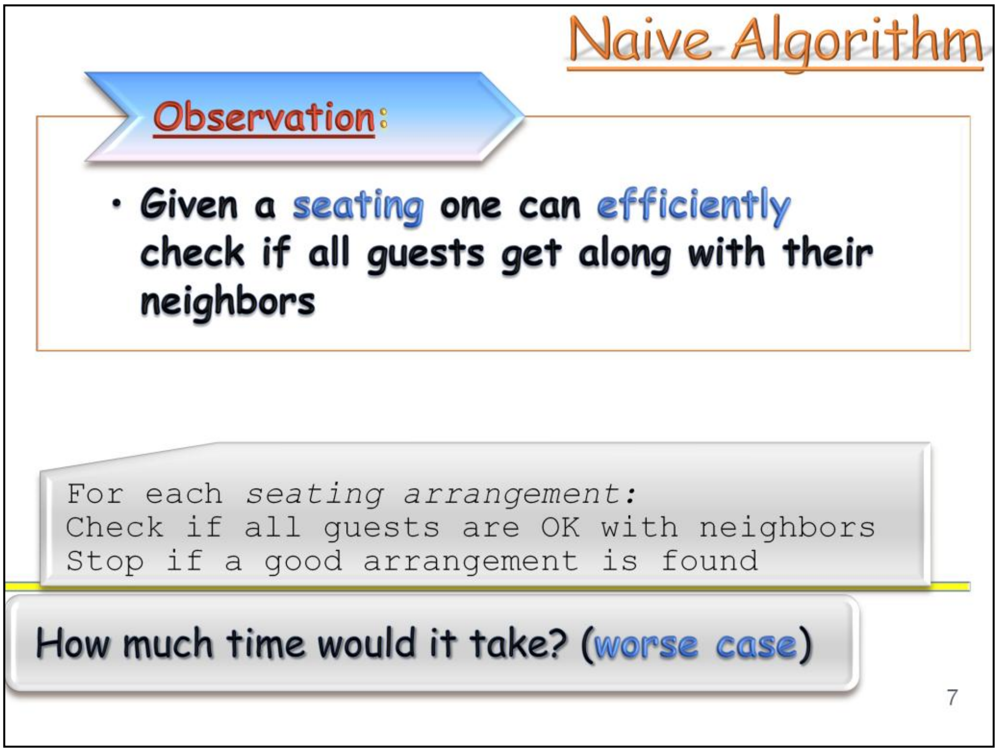
Every two guests may or may not get along with each other.




One can represent their relationship in a table,
which is essentially a 0,1 matrix.


Here is one way to organize guests so that they get along.

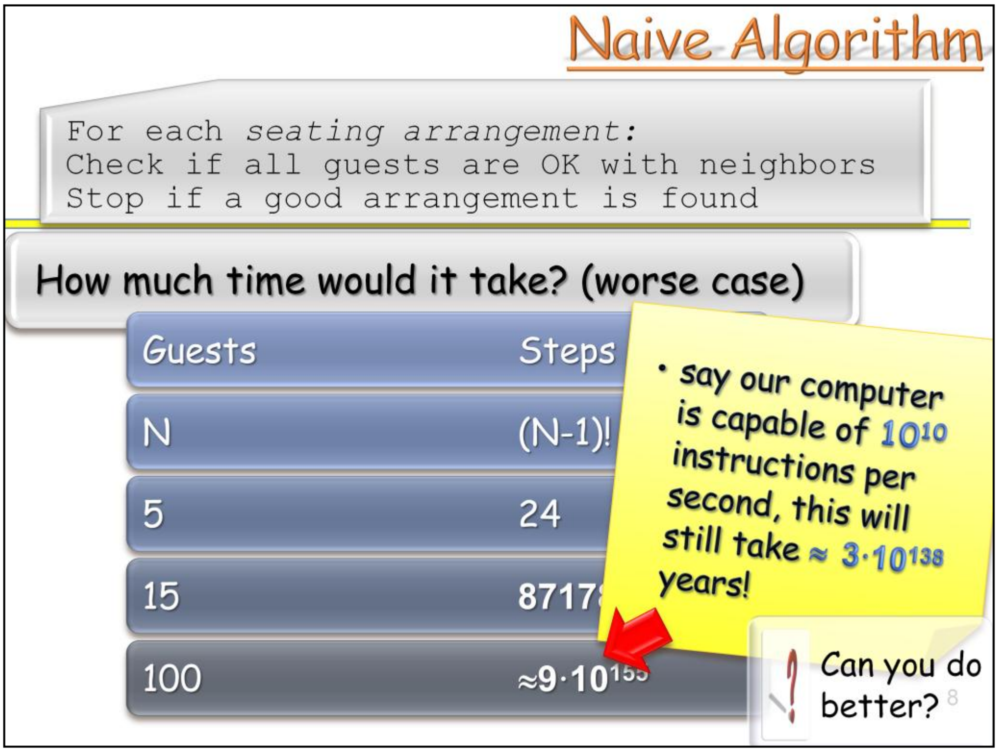
The question is what could be an organized, algorithmic method is to find such a seating if it exists.




Here is an algorithm for this problem:

It is easy to check whether a seating arrangement is a good one!

One can go over them one by one and check for each if it is good.




How long would this process take?

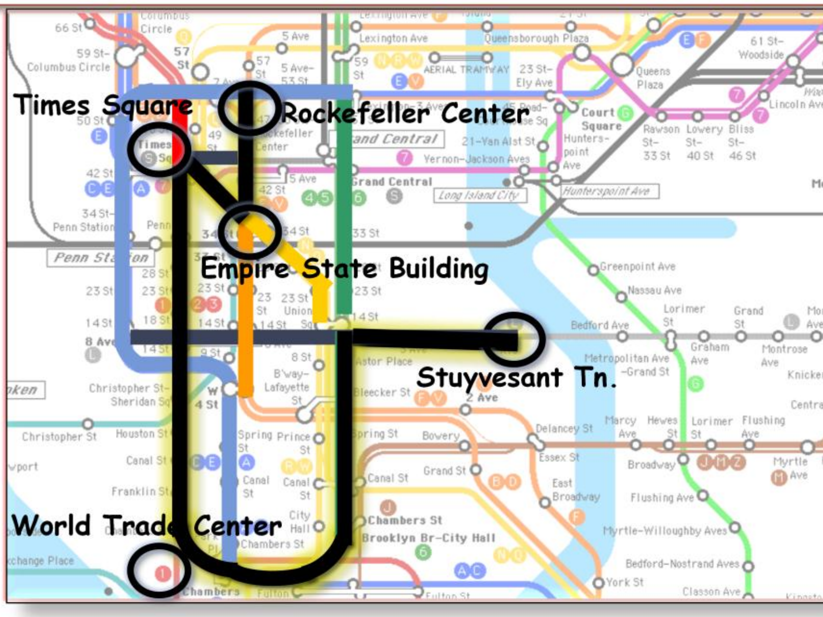
It is a function of the number of guests.

For a tiny number it may still be OK.

For anything but tiny number of guests,
the number of possible seating arrangements is huge.

## *Tour* Problem

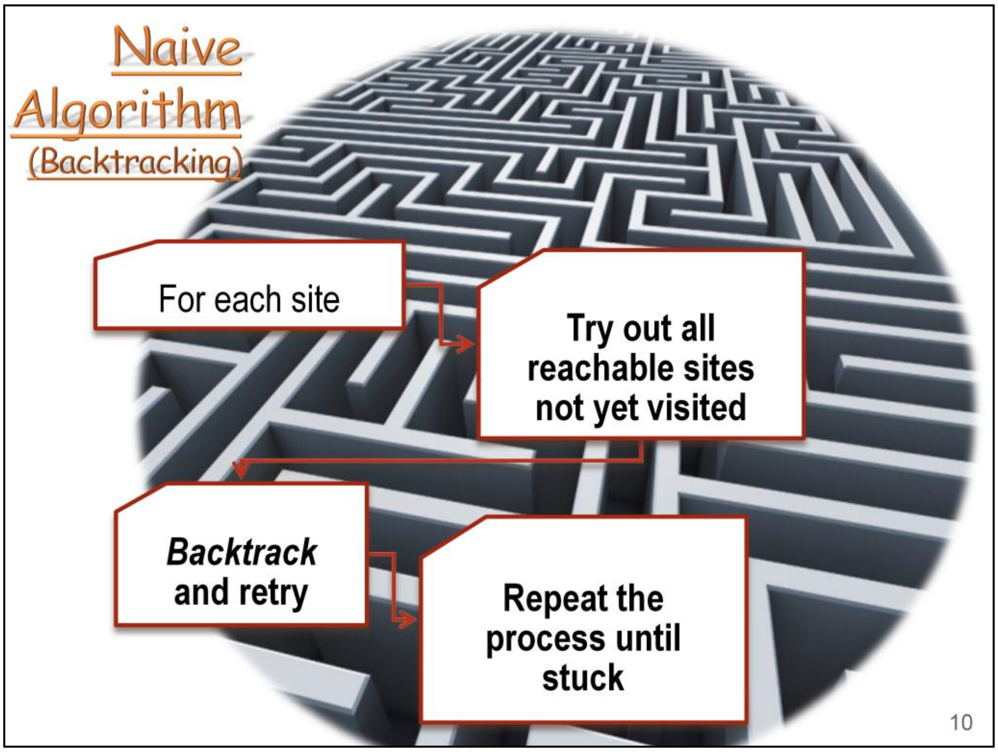
- Plan a trip that visits every location exactly once.




9

Here is another problem:

Say you are given a list of locations you need to visit and a map indicating between which locations there is a direct connection.

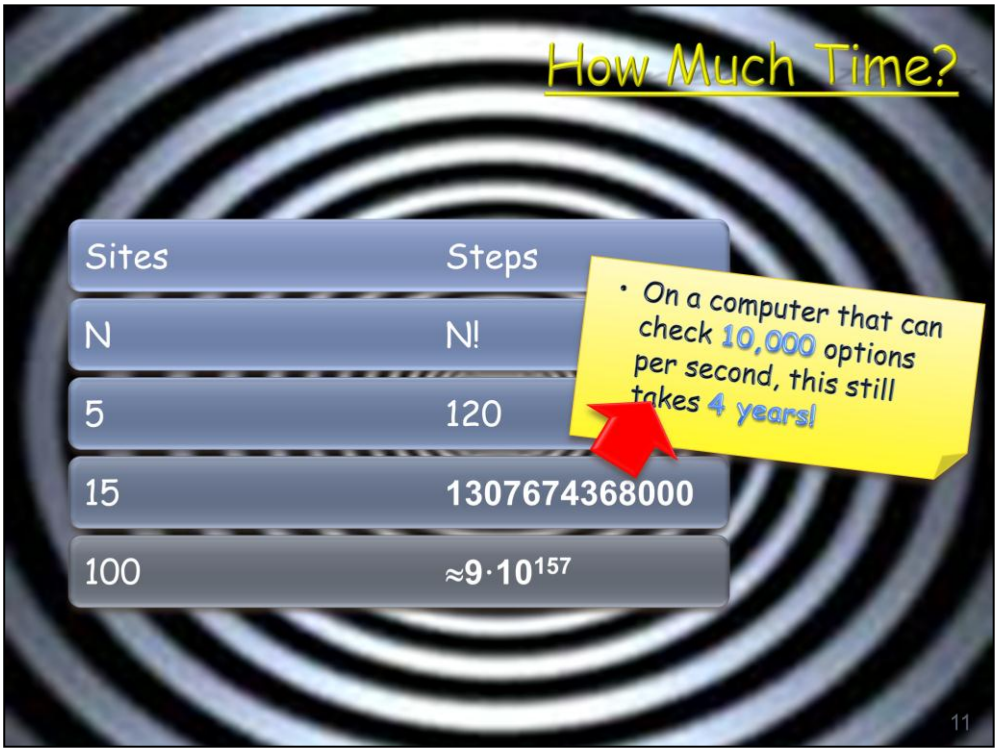



An algorithm for this problem would,

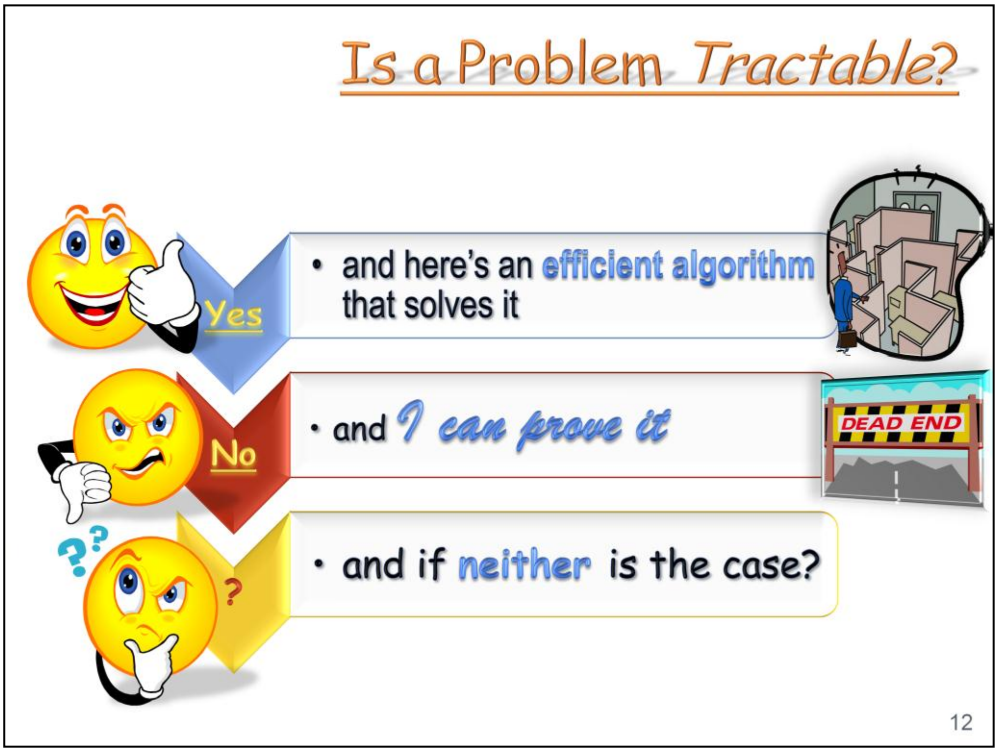
in every step,

go to the next connected location not yet visited.

If none exists, backtrack your steps and go to a yet not visited location.

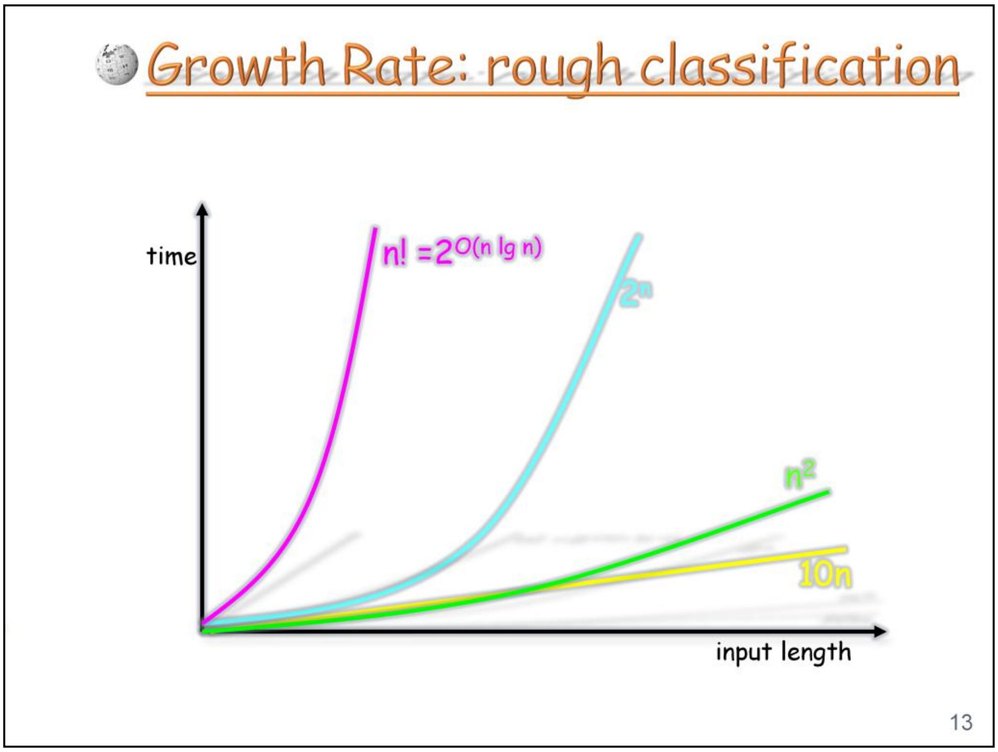



The time it will take this algorithm to figure out whether a traversal exists is even longer than the previous one.



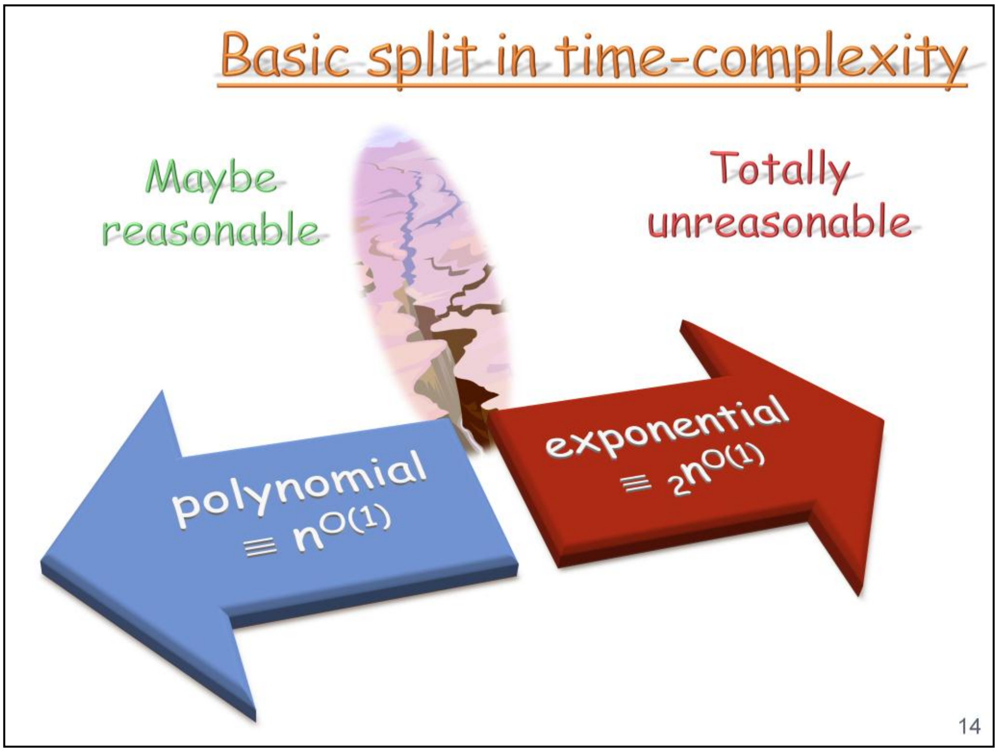

This brings us to the most fundamental question
one would like to know regarding a given computation problem:
Can it be efficiently solved?

The problem is that there are
almost no known techniques for
proving that a given problem
cannot be efficiently solved.

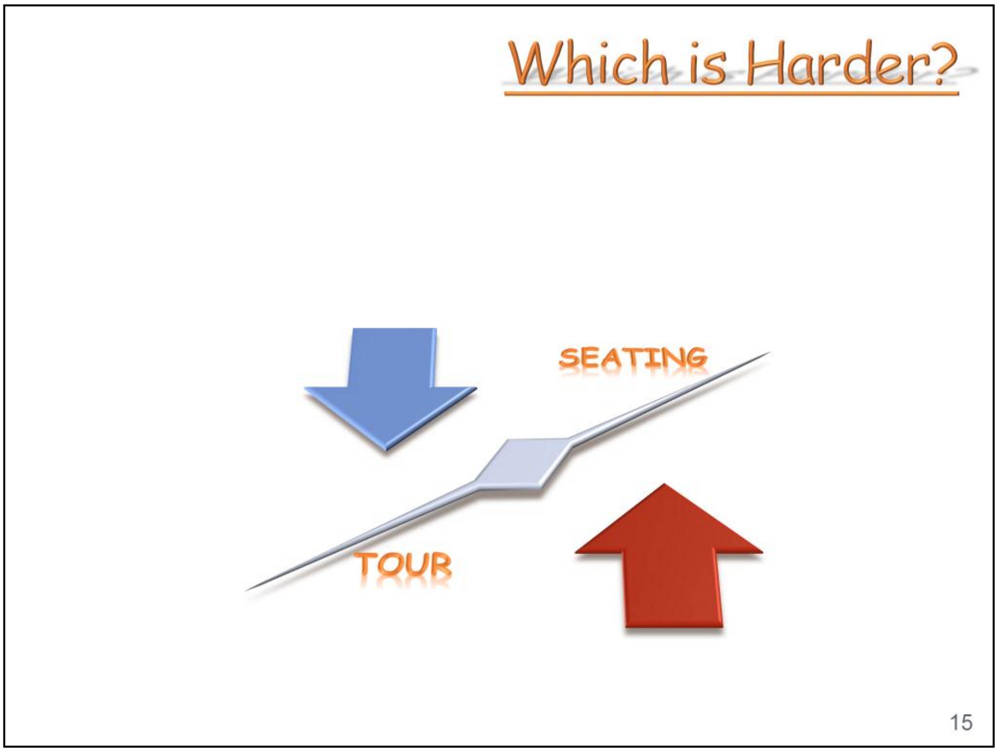



It's quite clear that the time it takes to solve a given problem is expected to grow as the input size grows.

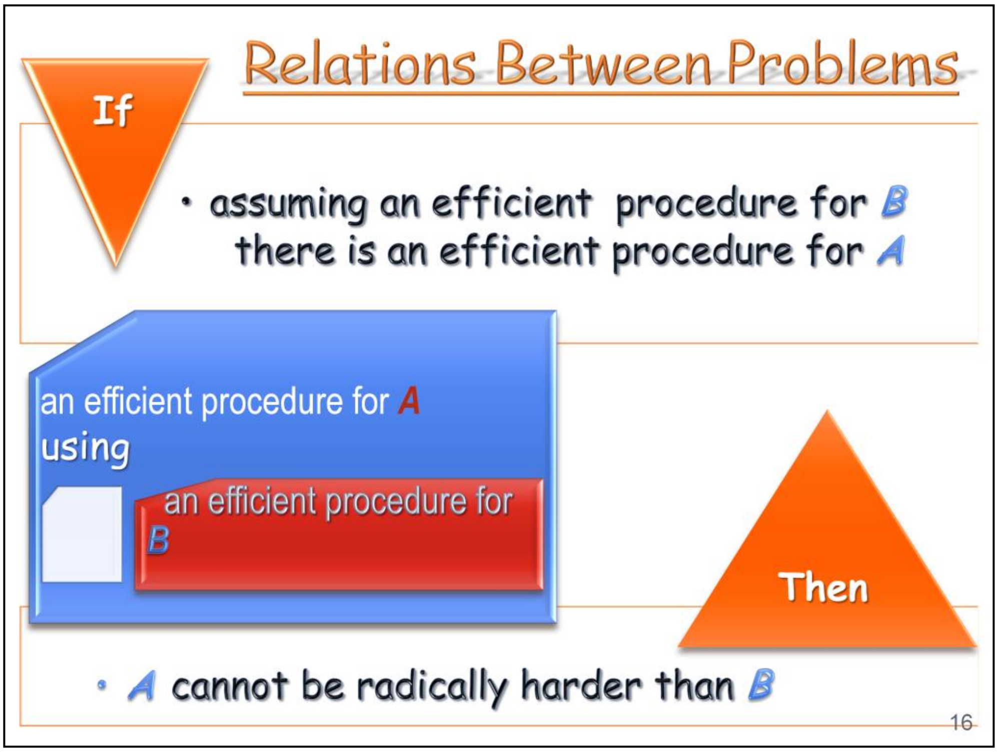
Some functions grow slowly as the input grows,
while other blowup very quickly.



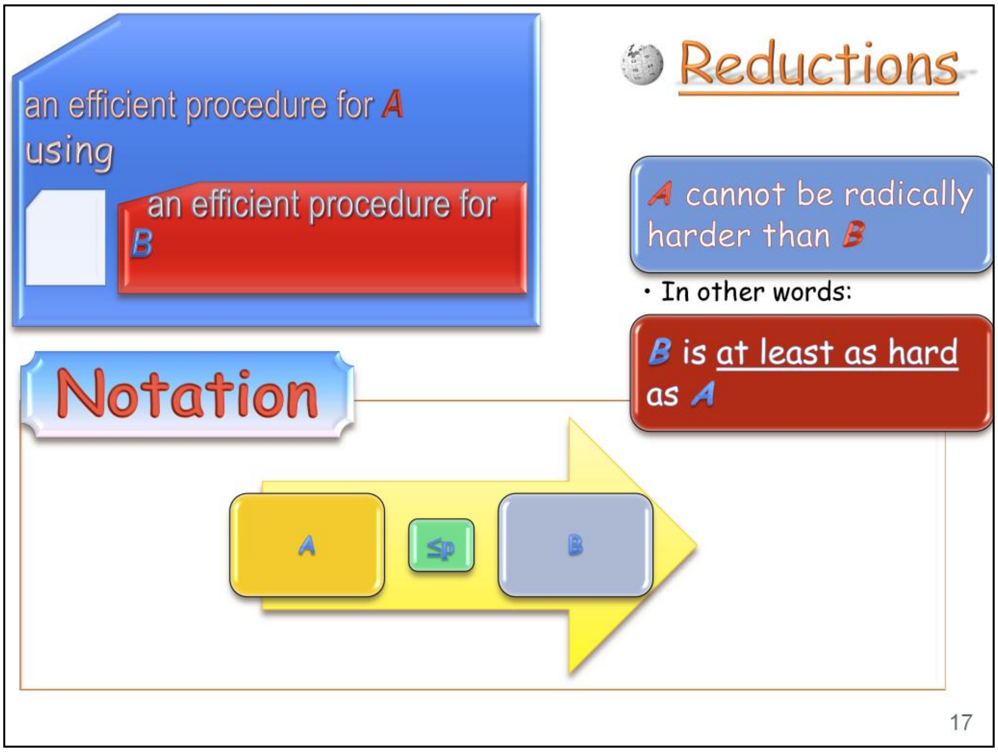

The most fundamental classification
we would like to apply to any
given computational problem
is the distinction between
problems whose growth rate in terms of time is *polynomial*
Vs. problems whose growth rate is *exponential.*



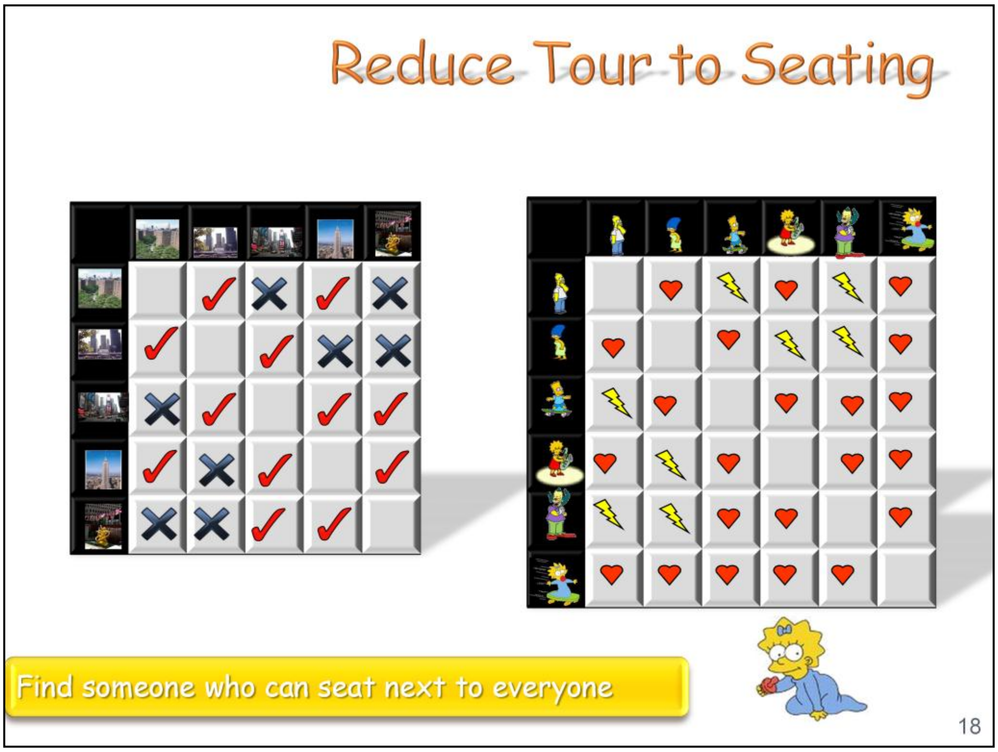

Once we have established
that the problem's complexity
can be measured by
a function of the time it takes to compute it for a given input size,
we can compare between problems' complexity.



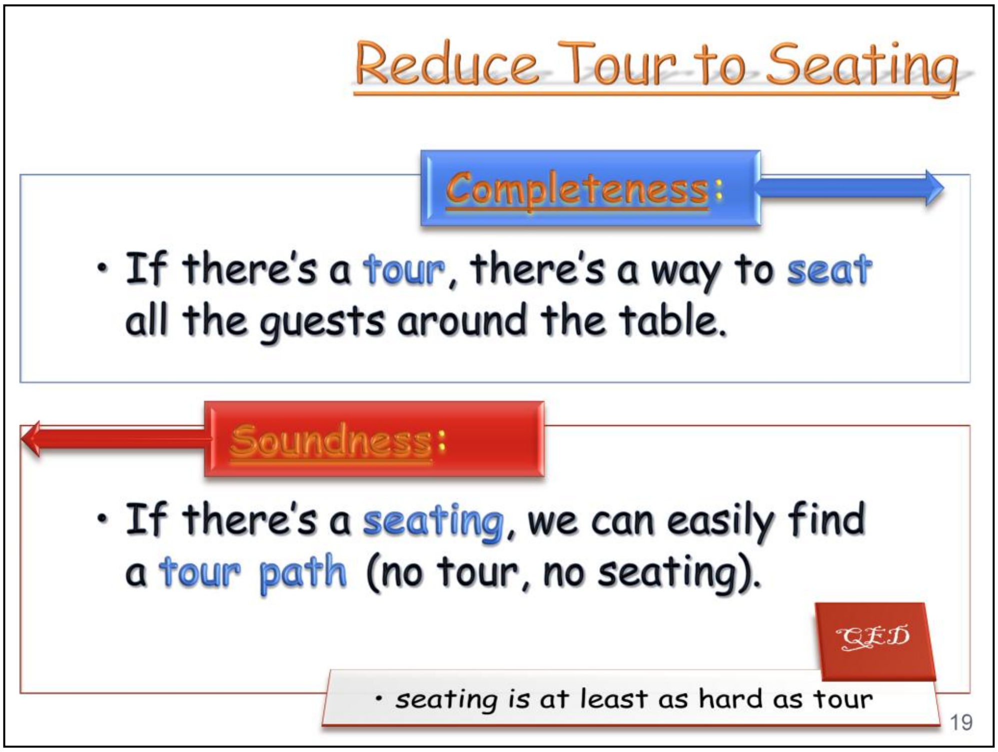

Assume that
we can come up with a procedure for problem A
that calls on a procedure for a problem B,
so that if B has an efficient procedure
then so does A;
it must then be the case that
A is not much harder than B,
or alternatively that
B cannot be much easier than A.




Here is how we denote such a notion:
we refer to it as "*reduction*";
the symbol we use to denote it is the "less than",
while the letter P implies the reduction is efficient.

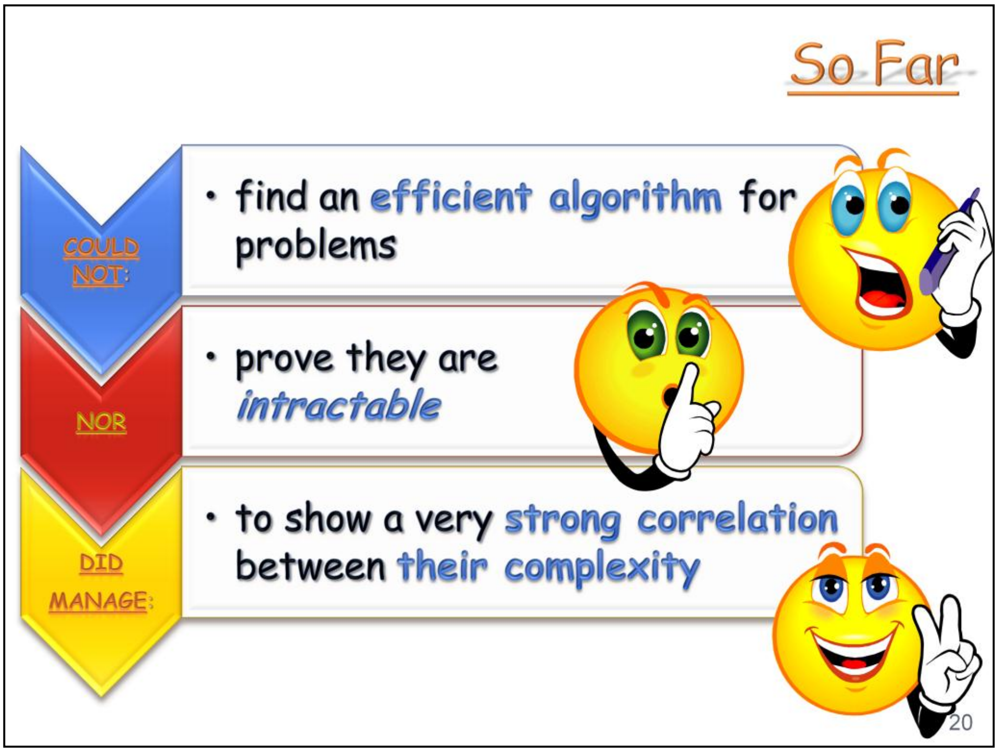



Here is a simple efficient reduction
from the tour problem to the seating problem:
think of every location as a guest
and now add an additional guest
that can be seated next to everyone.

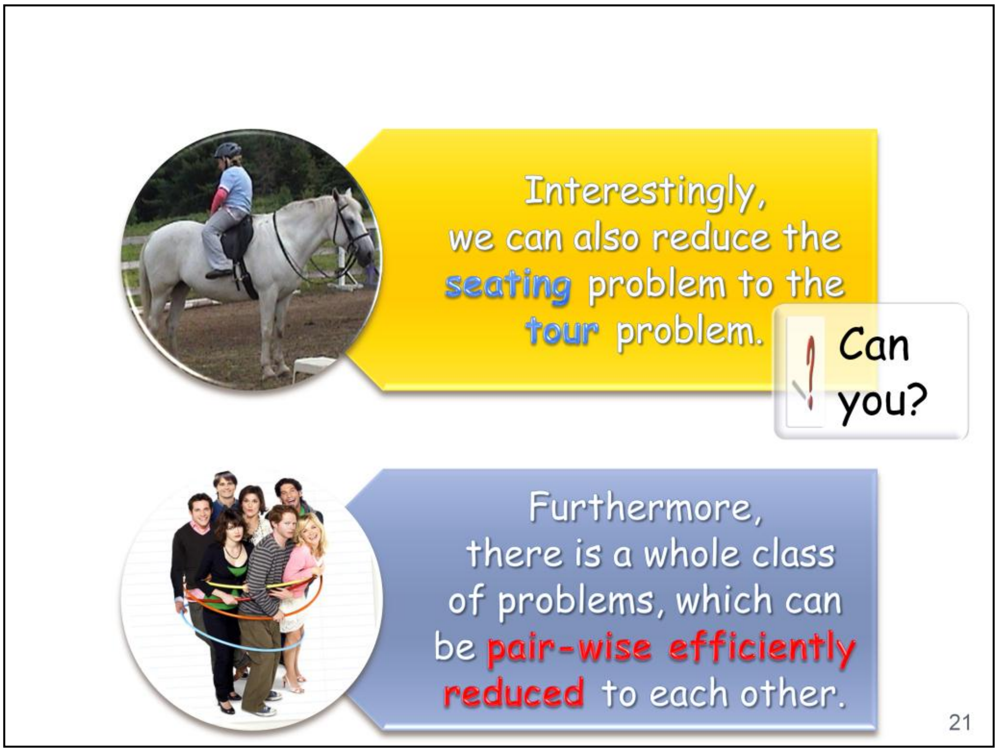



If there exists a tour,
seat guests accordingly
and seat the extra guest between the two ends of the tour.

The other side of the proof,
is proved in the counter positive form:
To prove that no tour implies no seating,
we prove that a seating implies a tour.
Given a seating, simply ignore the extra guest.

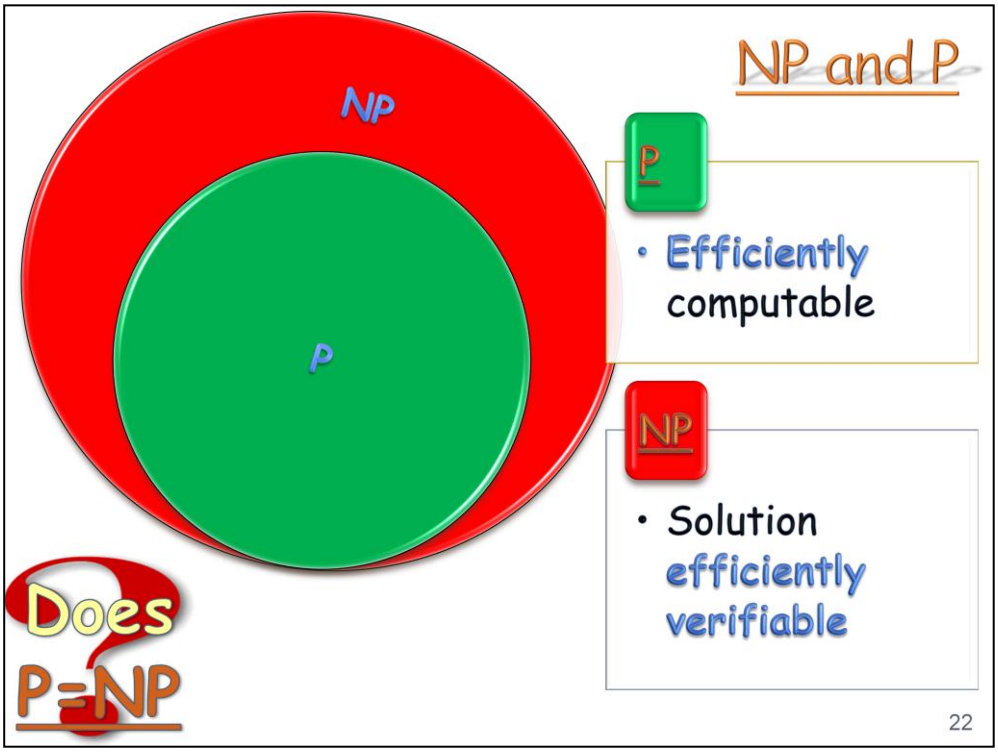



We have encountered some problems
whose complexity is quite unclear,
nevertheless,
we have managed to show
a relationship between their (unknown) complexities.



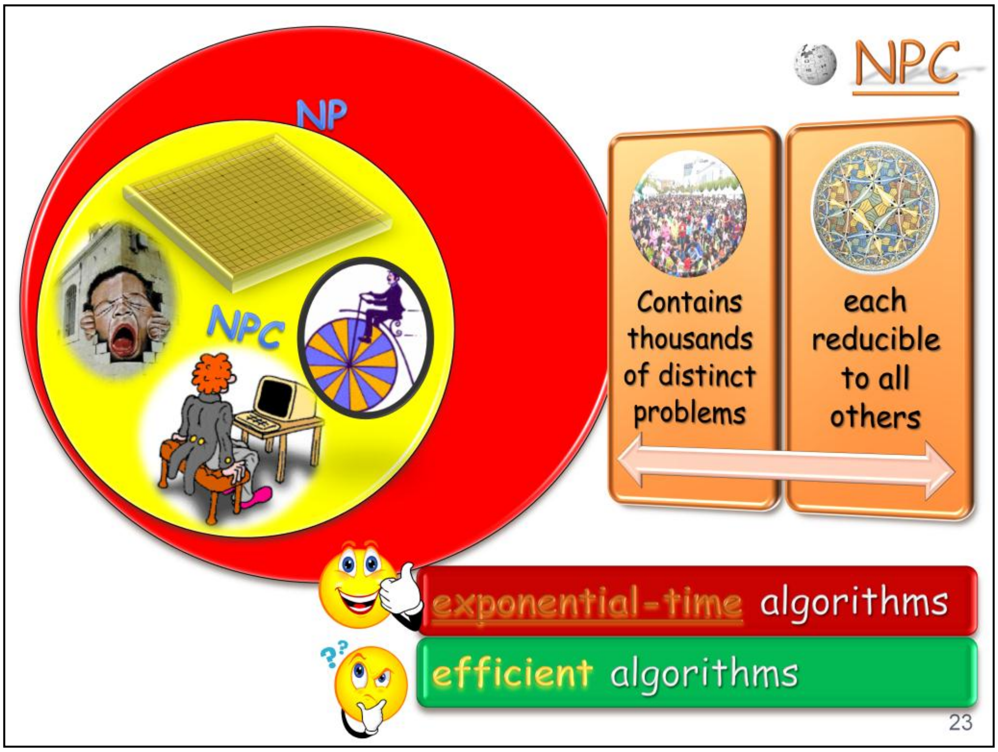

If we also show the reduction
in the other direction,
it would bound the complexity of the two
to be roughly the same.

It turns out that there is the
*class of problems*
whose complexity is bound
to the complexity of these two.

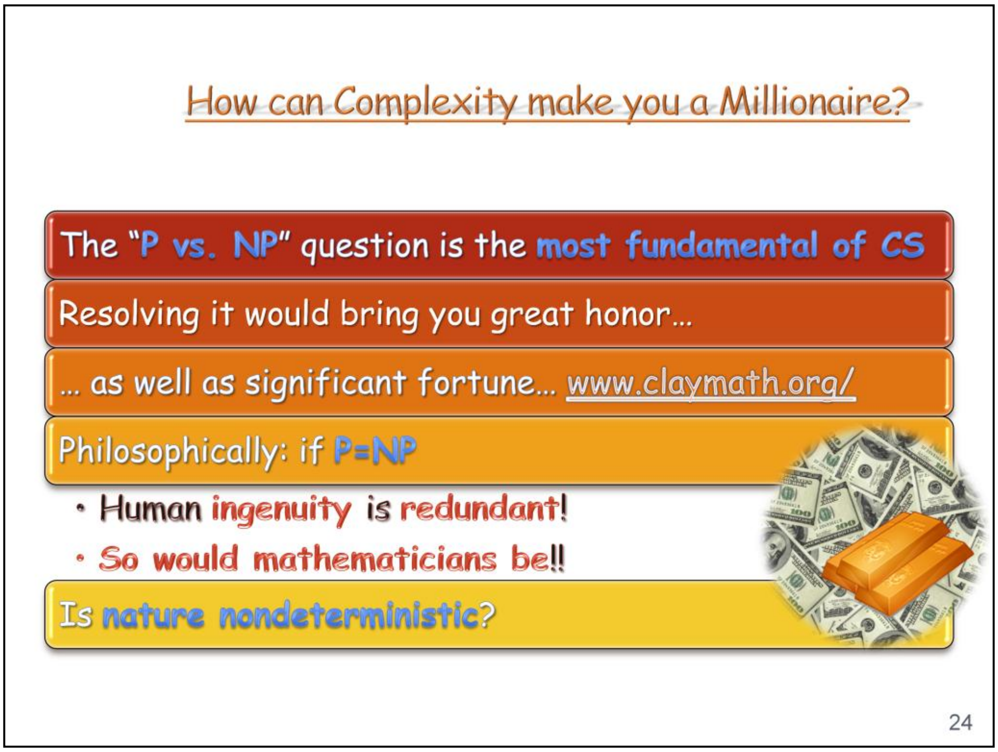



We can now informally introduce
two important classes of computational problems:
the class P,
which consist of all problems that can be efficiently computed,
and the class NP,
for which finding a solution can be very difficult
however checking the solution
can be carried out efficiently.

The $1,000,000 question is
whether the two classes are in fact the same.




Within the class NP,
we may consider the class of
what seemingly are the hardest problems,
whose complexities are all bound together:
this class is referred to as NP-complete

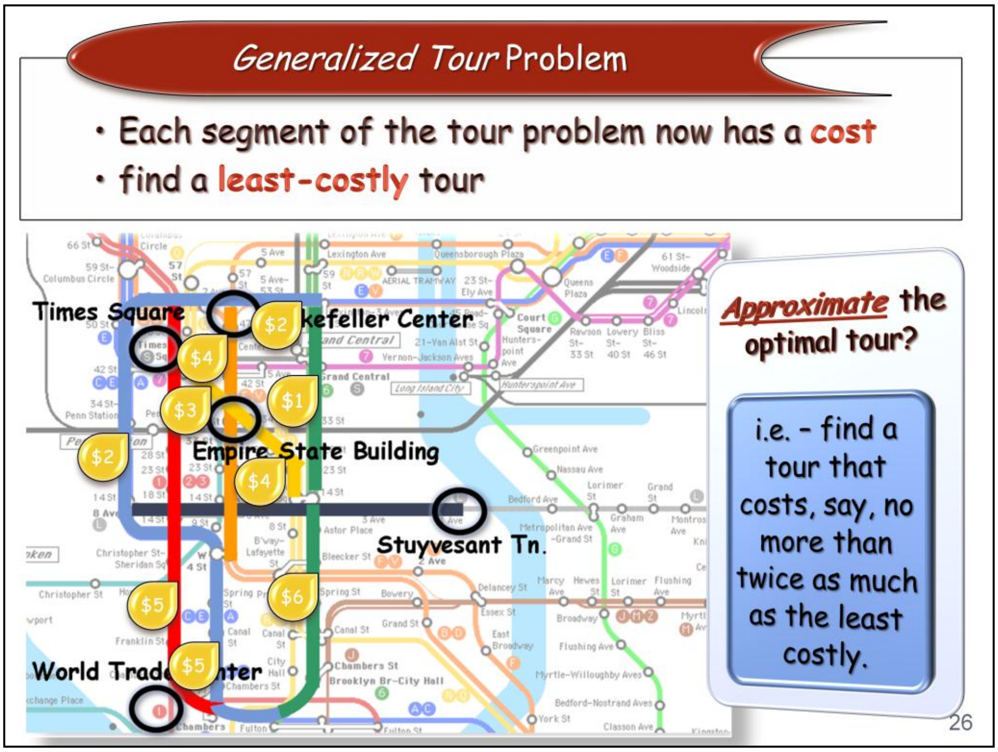



The P vs. NP problem
is the most fundamental question of computer science,
but it is also
one of the most important open questions in mathematics.

It is also a very deep philosophical question,
as if P is equal to NP
most human activities considered creative
may become mechanical.

It is also possible that some natural phenomena
utilized so far in computers suffer this distinction,
however, other natural phenomena may avoid this distinction.


Let us now briefly mention some other issues we will study in the course.




We can generalize the tour problem
assuming every direct connection
has a price attached to it.

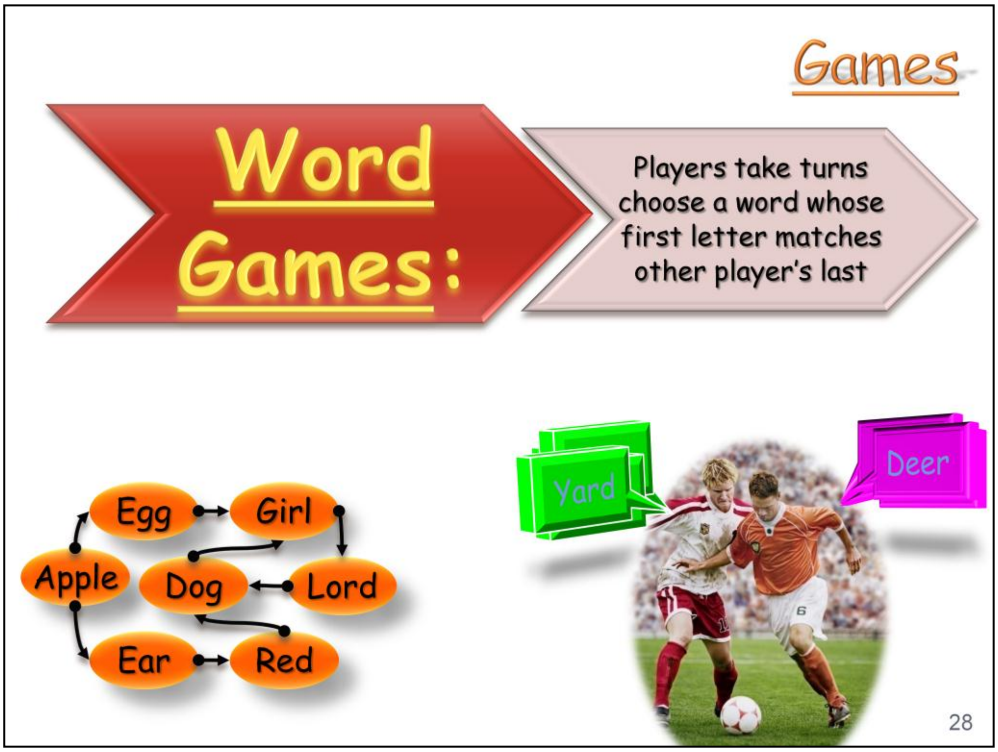
One would like to find the least expensive tour.

If that's impossible,
one would be content with a tour
that is not much more expensive
than the least expensive one.

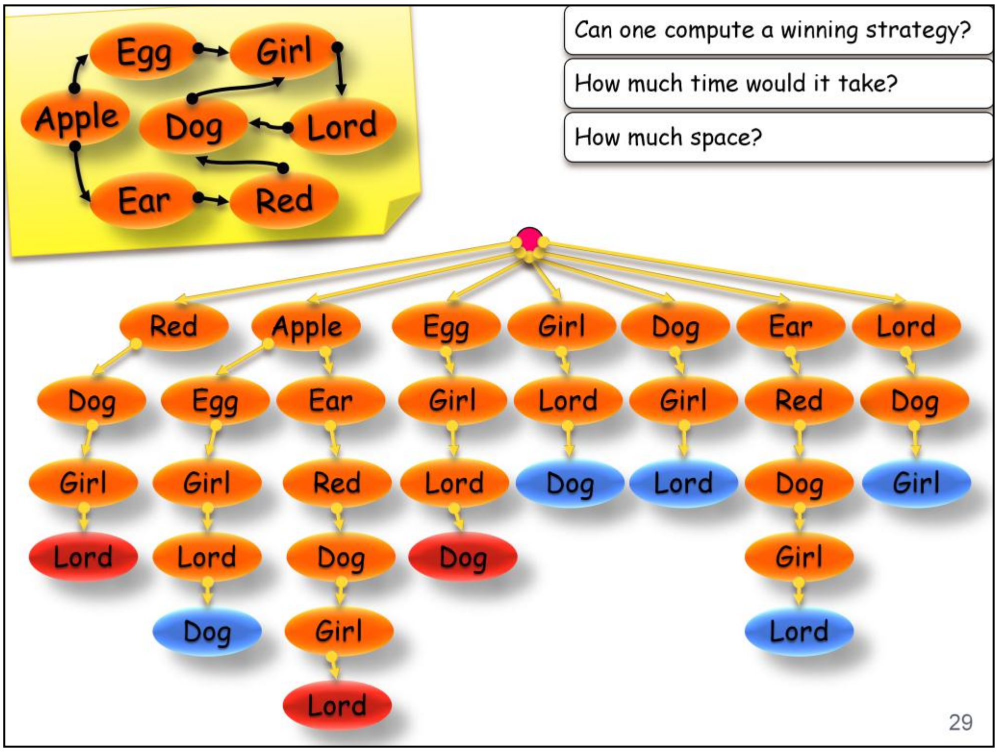
These types of problems are called
*approximation problems.*


So far we've measured
the complexity of problems
only according to the time it takes for their computation.

We will consider other resources,
in particular,
the size of memory it takes to solve them.




Here's an interesting example:
we're given the rules of
a game between two players
and are asked to decide
which of the players wins.




One can solve such a problem
by computing the game tree.
The size of that tree however
is potentially exponential
in the number of steps it takes
to get to the end of the game.

This is prohibitive!

Is there another way to solve this problem?

## Summary

We have introduced two problems:

1. **Seating** ≡ HAMILTONIAN-CYCLE
2. **Tour** ≡ HAMILTONIAN-PATH

Unable to settle their **complexity** we, nevertheless, showed strong **correlations** between them


These problems are representatives of a large **class** of problems:
NPC

## Prognosis

**Topics to be studied later:**

- Approximation
- Space-bounded computations

# WWindex

| | | |
|---|---|---|
| Complexity Theory | Computations | Completeness |
| Hamiltonian Path | Growth Rate | Completeness |
| Reducibility | | Soundness |
| Complexity Classes | P | NP |
| NPC | | |
| Exponential Time | www.claymath.org | Approximation |

32