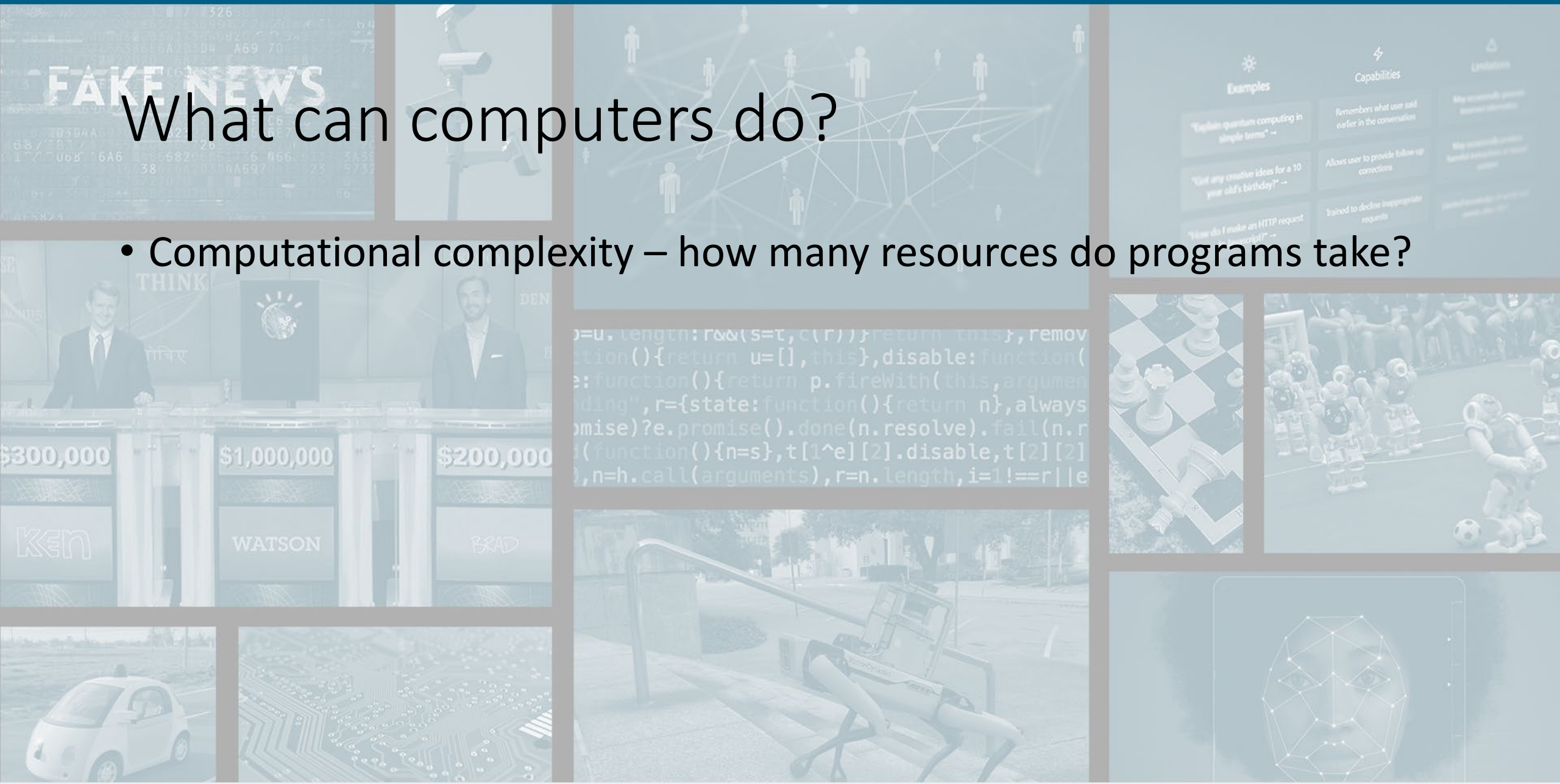
What can computers do?

- Computational complexity – how many resources do programs take?



Computational Complexity

- Can solvable (decidable) problems really be solved in practice?
- Maybe it would take millions of years of computer time, or ridiculous amounts of memory, to solve them.
- Chess and go for example can in principle be solved by just looking at all the games that could be played forward from a given position and seeing which move leads to the best outcome.
  - 10^{100} and 10^{300} choices
  - Cannot be done with anything close to reasonable resources

Complexity

- Pick a computer model, how about a Turing Machine?
  - That's good because it's as powerful as any other real computer in principle, so it can do what other more practical computers can do.
- Pick a resource you want to measure, like *time* or *space*
  - How long will it take? How much memory will it use?
- Pick a problem type, e.g. *decision problems*, which just answer yes or no to a given question on a given input.
- Consider the size of the input, i.e. the number of input items, call that **N**
- Now, determine how much time (how many steps) a program would need to solve the problem on the worst case input of size **N** to produce an answer.

Complexity

- Does an input string of characters contain at least 10 “A”s?
- Easy, just keep a counter, initialized to 0, and increment it each time you read in an “A”.
- Worst case – there are no ”A”s in the input until the last 10 characters, so you have to read in all **N** characters to get the answer. The best case is 10, if the first 10 characters are “A”s, you don’t have to read past the 10^{th} one to answer the question.

Complexity

Given a Boolean formula, is there some way to assign TRUE or FALSE values to its variables to make the formula true? (Boolean satisfiability)

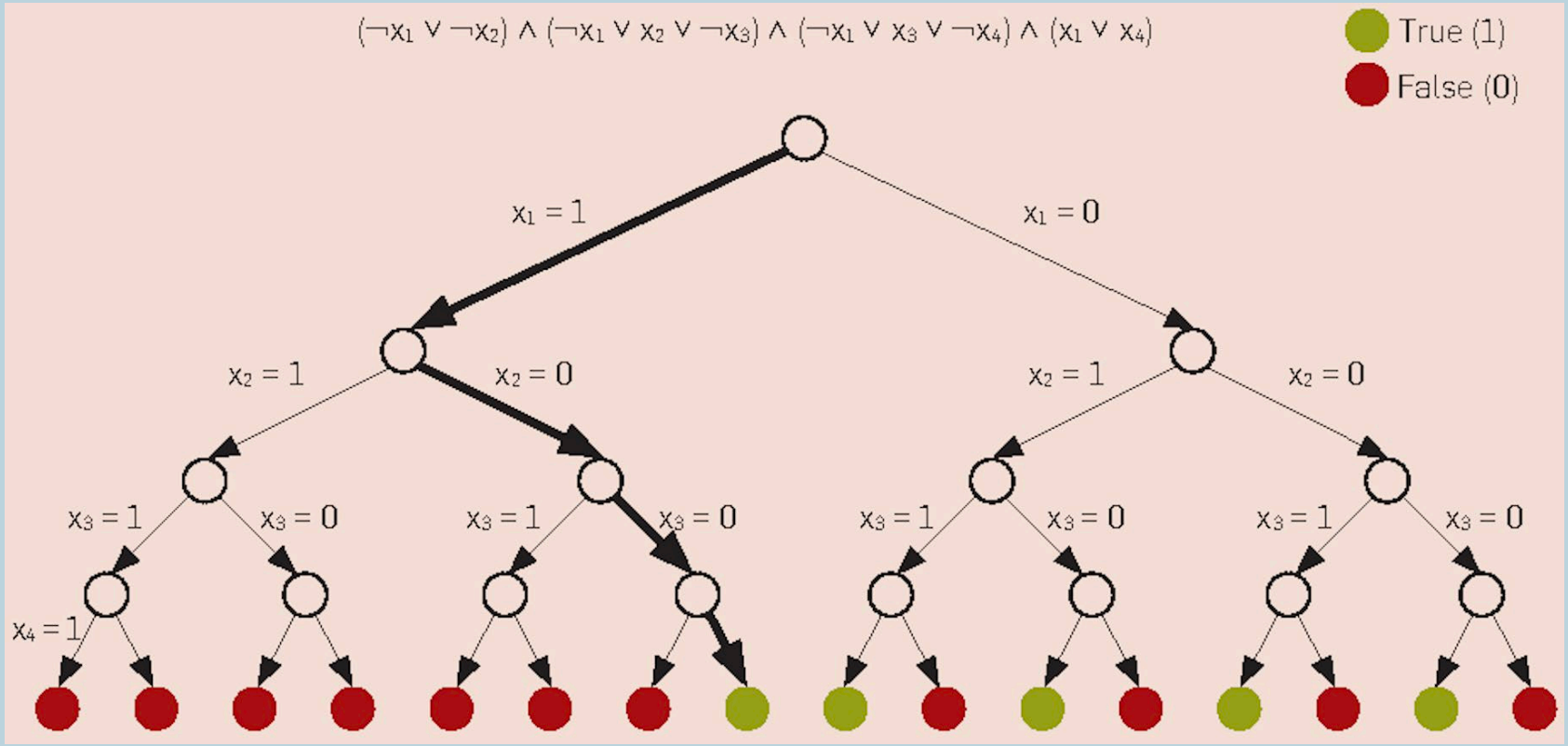
$$(\neg x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee x_3 \vee \neg x_4) \wedge (x_1 \vee x_4)$$

x_1=0, x_2=1, x_3=1, x_4=1 works

Complexity

Program could work like this:

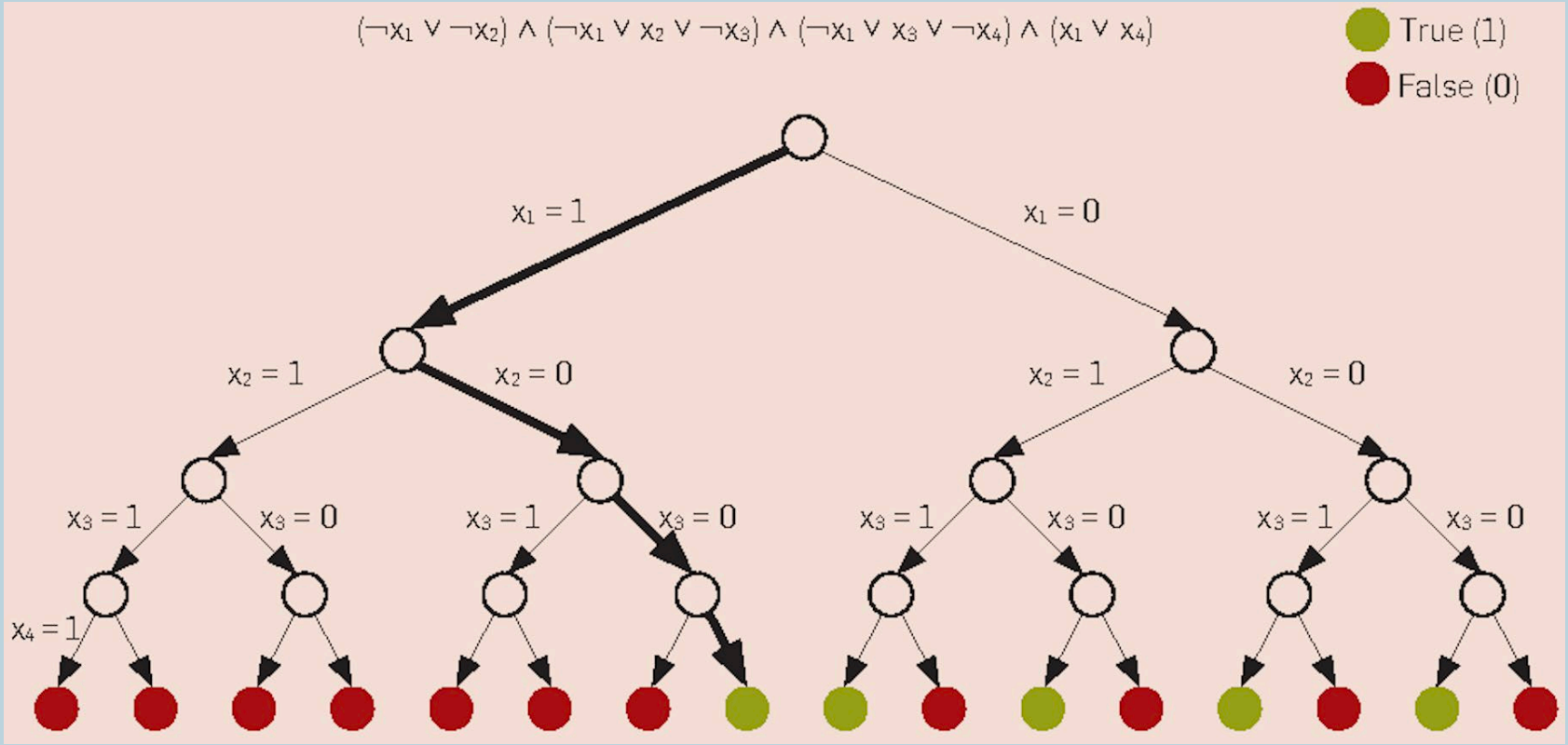
- 2^N choices
- That's too slow if **N** is large
- We don't know a way to do it that doesn't grow too large in the worst case.



Complexity

If we could build a ”nondeterministic Turing Machine” that makes copies of itself every time it branches and runs them in parallel, we could do it easily.

Of course, no such thing is possible.



Complexity Classes

- We can group problems into “classes” according to how hard they are in this sense
- P is the class of problems that can be solved efficiently on a conventional deterministic computer (Turing Machine) i.e. worst-case time required is expressible as a polynomial in n ($\mathbf{N}^2$, $\mathbf{N}^3$, etc. but not $k^{\mathbf{N}}$ - that grows too fast)
- Similarly, NP is the class of problems that can be solved efficiently using a “nondeterministic” computer (which we cannot actually build). It is also the class of problems we for which we can check to see if an answer is correct efficiently even if we cannot actually compute the answer quickly.

Complexity Classes

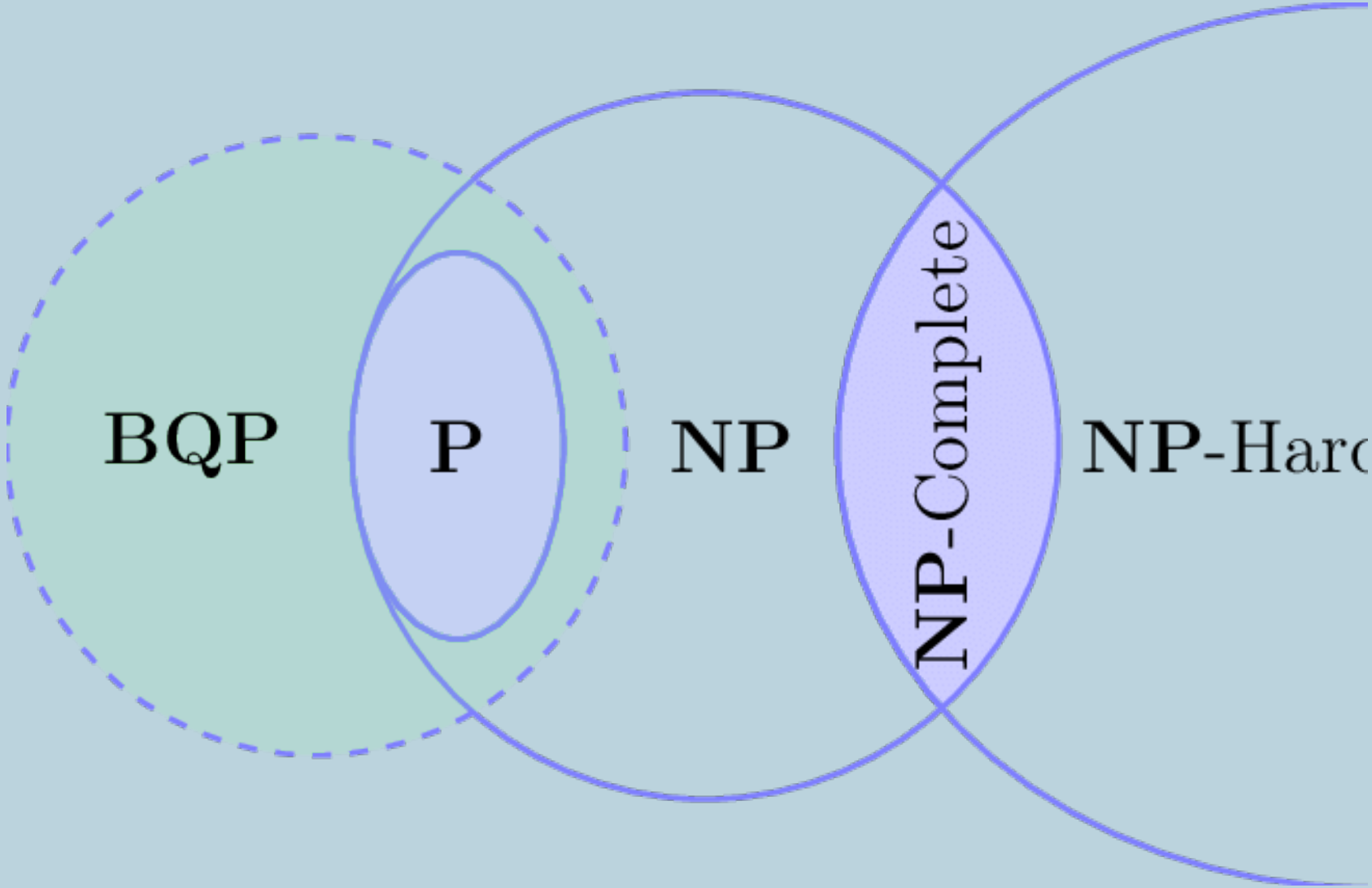
- We can also show that problems are essentially equally hard by showing how to transform one into another in polynomial time (reduction).
- In this sense, all problems in P are equally hard. For problems in NP, if they can all be reduced efficiently to a particular problem ***x***, then we say **x** is in the class of "NP-complete" problems. NP complete problems can all be transformed into each other efficiently, so they are all essentially equally hard, and they are all at least as hard as any other problem in NP.

P=NP?

- If even one NP-complete problem could be solved efficiently by an ordinary deterministic computer, then all problems in NP could also be solved efficiently.
- If we could prove that even one NP-complete problem could never be solved efficiently by a deterministic computer, then no NP-complete problem could be solved efficiently, so NP-complete problems are harder those in P.
- We believe that the latter is the case, but we cannot prove it (yet). There are many other complexity classes, and generally we do not know how to prove that they are different in this sense.

Quantum Complexity

- For quantum computers, we cannot prove that any problems can be solved faster (in a complexity sense) on a quantum computer than on an ordinary deterministic computer.
- We have evidence that there may be problems that are efficiently solvable on quantum computers but not on classical deterministic computers.
- E.g. integer factorization, Shor's algorithm can do this on a quantum computer in polynomial time
- Integer factorization is the basis for many cryptographic protocols



From: Dan Padilha https://www.researchgate.net/publication/322159605_Solving_NP-Hard_Problems_on_an_Adiabatic_Quantum_Computer

Heuristics

- Many problems cannot be decided algorithmically
- Others which can take too many resources to be computed in practice
- So maybe we can't a computer to produce exact answers to some problems.
- But what about approximate answers?
- Or what about getting it exactly right most of the time but not always?
- Programs to do this can be much more efficient (or even possible) than exact algorithms.
- Such programs can be called heuristics, i.e. using efficient rules of thumb to solve problems "well enough" even if not always perfectly.