DS 320 Algorithms for Data Science **Complexity Classes**
Spring 2024 Prof. Kira Goldner

P vs. NP, NP Completeness, and Reductions

- A decision problem is in P if there is an algorithm which solves it in polynomial time. ("Easy to Solve")
- A decision problem is in NP if there exists a polynomial-time certifier algorithm (that takes polynomial-size certificates). ("Easy to Check")
  - Can always find a certificate to convince me of a yes input.
  - Can't convince me that no inputs are actually yes inputs.
- $P \subseteq NP$—solve the problem itself in poly-time and use that as the certificate.
- Not all problems are in NP.
- If $A \leq_P B$ (A poly-time reduces to B), then you used B to solve A in polynomial time, so B is *at least as hard as* A.
- A problem B is *NP-hard* if for all $A \in NP$, $A \leq_P B$. That is, B is *at least as hard* as every problem in NP. Note B is not necessarily in NP.
- A problem is *NP-complete* if it is NP-hard and also in NP. To prove a problem in NP-complete,
  - Prove it's NP-hard (reduce from a known NP-hard problem)
  - Prove it's in NP. (describe the certification algorithm)
- 3-SAT is NP-Hard. [Cook-Levin '71]
  - We showed that this implies 3-SAT is NP-complete, and thus so is Independent Set, Vertex Cover, and Knapsack.

Main Steps for a Reduction

Reductions tend to follow a consistent pattern. The following is for reducing A to B, that is, $A \leq_P B$.

Step 1: Choose a problem to reduce to. What are some problems that you (a.) know how to solve and (b.) have a similar structure to the one you want to solve? Pick one of these. You might have to try several options before one works. Hereafter, we'll call the problem you are trying to solve A and the problem you are reducing to B.

Step 2: Construct an instance of problem B. Given an instance of problem A, give a systematic way of representing that instance as an instance of problem B. Each part of the input to A should correspond to a part of the input for B.

Step 3: Solve your instance of problem B. We have assumed you know a method for solving problem B. Solve the instance you constructed using this method.

Step 4: Interpret your solution to problem B. Your algorithm needs to return something! You have solved the instance of problem B you have constructed; that should tell you what you wanted to know about your instance of problem A. For yes/no decision problems, you will often return true for A if the solution to B is true, and false if not. If you are asked to produce a solution to A beyond a True/False answer, you will derive this solution to A from your solution to B.

Step 5: Prove your algorithm is correct. To prove correctness, you should show how to turn solutions to A into solutions to B and vice versa. If your problem is a yes/no decision problem, this will imply that your algorithm returns "yes" if and only if the answer is actually "yes." If the problem requires you to return an explicit solution, this will show that the thing your algorithm returns is a solution, and that you find a solution whenever one exists. The two parts, more explicitly, are:

a. Turn a solution to B into a solution to A. If your algorithm returns something other than True or False, you probably have already described how to do this. In either case, you should prove that the solution you produce to A is indeed a solution to A - it is feasilble, and if your are maximizing (or minimizing) an objective function, has a high (or low) objective value.

   For decision problems, this shows that your algorithm returns "yes" only if the answer is truly yes. That is, it gives no false positives. For problems where you return a solution, this proves that the solution you return is feasible.

b. Turn your solution to A into a solution to B. As with the other direction, you should prove that this solution to B is feasible and if appropriate, has a high (or low) objective value.

   For decision problems, this shows that if the answer is truly yes, your algorithm will return "yes." That is, it gives no false negatives. For problems where you return a solution this proves that your algorithm will always find a solution if it exists.

Step 6: Runtime. We will only care about proving that the runtime of our algorithm is polynomial. To prove that this is the case, you only need to show that the size of the instance of B that you constructed is polynomial in the size of your input to A. Since your algorithm for solving B was a polynomial-time algorithm, the resulting runtime of the whole reduction will be polynomial.

If you were interested in the precise runtime of your algorithm, you would need to plug the size of your input to B into the runtime function for the algorithm you used to solve B