P vs NP

So far, we've seen:

- problems solvable efficiently by *simple* algorithms (greedy);
- problems that cannot be solved efficiently by simple techniques, but that can be solved efficiently by more complex algorithms (network flows, dynamic programming, linear programming);
- problems that cannot be solved efficiently by any known algorithm (the only known solutions involve exhaustive search).

We want to find a general method to prove some problems has no efficient solution.

Running times and input size:

- What is the input size? The amount of space we need to save the input.
- consider *bit-size*, i.e., each integer a requires size $\lceil(log_2(a+1))\rceil$ approximately $log_2 a$. For example, bit size of list $[a_1, ..., a_n]$ is $log_2 a_1 + ... + log_2 a_n \leqslant n log_2(max(a_1, ..., a_n))$. With additional assumption that integers take up constant number of bits, this is equivalent to n within a constant factor – but that assumption does not always hold!
- One exception: representing integers in *unary* notation (i.e., integer k is represented using k many 1's) requires exponentially many more characters than binary (or any other base), so we rule this out.
- As it turns out, different models of computation and different ways of measuring input size can affect running time by up to polynomial factors (e.g., doubling, squaring, etc.) So simply using big-Oh notation to measure running time is too "precise": e.g., problems solved in time $\Theta(n)$ on one model may require time $\Theta(n^3)$ on another model.
- Use coarser scale: ignore polynomial differences. Luckily, any polytime algorithm as a function of informal *size* is also polytime as a function of bitsize, as long as it uses only *reasonable* primitive operations – comparisons, addition, subtraction – other operations (multiplication, exponentiation) are **NOT** considered *reasonable* because repeated application of the operation can make the result's size grow exponentially. This does not mean that we cannot use multiplication, just that we cannot count it as a primitive operation.

Search problems vs. Optimization problems vs. Decision problems:

For a problem X:

- Search problem: we need to find a solution that satisfies X.
- Optimization problem: we need to find a solution that satisfies X and minimizes (maximizes) an objective function.
- Decision Problem: we need to determine if there is any solution for X. In fact the output of a decision problem is a single yes/no answer. e.g., "Given graph G, vertices s,t, is there a path from s to t?" or "Given weighted graph G, bound b, is there a spanning tree of G with weight $\leqslant b$?" We usually use the notation D to show a decision problem.
- Why limit ourselves to decision problems? We want to prove negative results (that problems have no efficient solution). Intuitively, decision problems are "easier" than corresponding optimization/search problems, so if we can show that a decision problem is hard (has no efficient algorithm), this would imply that the more general problem is also hard. We'll make this precise.

P, and NP:

- The class P: All decision problems for which solutions can be efficiently (*i.e.*in polynomial time) found.
- The class NP (non-deterministic polynomial): All decision problems that can be "verified" in polytime. A problem D can be "verified" in polytime if there is an algorithm V (the "verifier") that takes two inputs (x, c) such that:
  - for all yes-instances x (inputs to D for which the answer is yes), there is some string c such that $V(x, c)$ outputs "yes" in polytime (c is called a "certificate");
  - for all no-instances x, for all strings c, $V(x, c)$ outputs "no".

  In other words, for all inputs x, the answer is yes iff there is a certificate c such that $V(x, c)$ outputs "yes" in polytime (measured as a function of $n = |x|$ only, ignoring the size of c).
- Equivalently: decision problem D is in NP iff there is an algorithm to solve D that has the following form:

```
On input x:
            generate all "candidates" c, and for each one:
                verify some property of x and c
```

  where the last "verification" step runs in worst-case polynomial time, as a function of $size(x)$. The time required to generate all candidates is ignored.
- Why the complicated definition? Doesn't correspond to any practical notion of computation. However, turns out to exactly capture the computational complexity of the vast majority of "real-life" problems.

Example 1: Show that the decision problem COMPOSITE (given positive integer x, does x have any factors?) belongs to NP.
Consider the verifier $V(x, c)$:

- check that $1 < c < x$,
- check that c divides x,
- output *yes* if both checks succeed; *no* otherwise.

If x is composite, $V(x, c)$ will output *yes* for some value of c (pick c to be any factor of x). If x is not composite (*i.e.*, x is prime), $V(x, c)$ will output *no* for every value of c. Moreover, $V(x, c)$ runs in polytime as a function of $|x|$ because basic arithmetic operations are polytime.
Note: the obvious algorithm (try all prime numbers between 2 and $\sqrt{x}$) does NOT run in polytime because there are at least $\sqrt{x}/lnx$ many such numbers to try. Expressed as a function of $n = \log_2 x$ (the bit-size of x), this is $2^{n/2}/n$, which is exponential.

Example 2: VERTEX-COVER
Input: Undirected graph G, positive integer k
Question: Does G contain a vertex cover of size k, *i.e.*, a set C of k vertices such that each edge of G has at least one endpoint in C?
VERTEX-COVER (VC) is in NP:
On input $< G, k, c >$:

- verify that c is a subset of exactly k vertices of G
- check that c forms a vertex cover

First step takes time $\mathcal{O}(kn)$; second step takes time $\mathcal{O}(mk)$; total is $\mathcal{O}(k(m+n))$ that is polynomial.
Notation: From now on, the notation "$x \in D$" means that "x is a yes-instance of the decision problem D" (similarly, "$x \notin D$" means that "x is a no-instance of D").
Note: P is a subset of NP; however it is not known whether P = NP or P ≠ NP. It is widely believed that P ≠ NP (but not proven yet).

Polytime reductions/transformations:
A technique to formalize this notion that one problem is not "harder" than another.

$D_1 \leqslant_p D_2$ if there is a function f computable in polytime such that for all inputs x, $x \in D_1$ iff $f(x) \in D_2$. In other words, inputs for D_1 can be transformed (in polytime) into inputs for D_2 such that the answers are the same for both inputs.

We have seen many concrete examples of reductions, when working with network flows and linear programming.

Theorem: If $D_1 \leqslant_p D_2$ and D_2 is in P, then D_1 is in P. (same for NP)
Proof: By definition, $D_1 \leqslant_p D_2$ means there is some polytime computable transformation f such that $x \in D_1$ iff $f(x) \in D_2$. D_2 in P means that there is some algorithm A such that $A(x) =$ yes iff $x \in D_2$. The following is a polytime algorithm for D_1:
On input x, compute $f(x)$ and call $A(f(x))$, returning same answer.
Since $f(x)$ computable in polytime, $|f(x)|$ (the size of $f(x)$) is a polynomial in $|x|$, so runtime of $A(f(x))$ is a polynomial function of a polynomial in $|x|$, which is still polynomial.
Same argument works with polytime verifier $V(x, c)$ in place of algorithm $A(x)$, to show the result for NP.

Corollary: If $D_1 \leqslant_p D_2$ and D_1 not in P, then D_2 not in P.

NP-completeness:
* Use $\leqslant_p$ to identify "hardest" problems in NP.
Decision problem D is "NP-complete" if:

1. D in NP
2. D is "NP-hard", *i.e.*, for all D' in NP, D' $\leqslant_p$ D.

Theorem: Suppose D is an NP-complete problem. Then D in P iff P = NP.
Proof:
(⇐) If P = NP, then D in NPc → D in NP → D in P.
(⇒) If D in P, then D in NPc → D in NP-hard → for all D' in NP, D' $\leqslant_p$ D so D' in P since D in P. Hence, NP subset of P so NP = P.

Corollary: If $P \neq NP$ and D is NP-complete, then D not in P.

Some well-known NP-complete problems:

- Circuit-SAT: Given a circuit with a single output gate, is there some setting of the inputs that will make the output equal to 1?
- SAT: Given a propositional formula F (written using propositional connectives negation, and, or, implication), is there some setting of the variables that will make F true (in which case F is said to be *satisfiable*)?
- CNF-SAT: Given a propositional formula F in Conjunctive Normal Form (also called product of sums), is F satisfiable? Note this means F has the form $C_1 \wedge C_2 \wedge \cdots \wedge C_k$, where each "clause" $C_i = a_1 \vee a_2 \vee ... \vee a_r$,

where each "literal" a_j is either a variable (x) or negated variable ($\sim x$). For example: $(x_1 \vee \sim x_2) \wedge \sim x_3 \wedge (\sim x_1 \vee x_2 \vee x_3 \vee x_2)$

- 3SAT: Given a propositional formula F in 3-CNF (CNF where each clause contains exactly 3 literals), is F satisfiable?

NP-hardness:
Theorem: To show that D is NP-hard, it is sufficient to find some known NP-hard problem D' and prove $D' \leqslant_p D$.

Proof: Since D' NP-hard, for all D'' in NP, $D'' \leqslant_p D'$. Moreover we know that if $A \leqslant_p B$ and $B \leqslant_p C$, then $A \leqslant_p C$ (you should show it in your assignment). Therefore, $D'' \leqslant_p D$ (since $D' \leqslant_p D$). This shows that D is NP-hard.

Cook's Theorem: SAT is NP-complete.

- SAT in NP:
  Given F,c, where c is a setting of values (True/False) for the variables of F:

  Output the value of F under the setting given by c.

  This can be carried out in polynomial time: given a formula F and a setting of its variables, just substitute the values for each variable and then evaluate each connective one-by-one, from the inside out.
  Moreover, if F is satisfiable, then there is some value of c that will make this verifier output yes (when $c =$ a setting that makes F true); and if F is not satisfiable, then this verifier will output *no* for every possible value of c (since no setting makes F true).

  The same reasoning shows that Circuit-SAT, CNF-SAT and 3SAT also belong to NP.

- SAT is NP-hard (next week)