THE P VS. NP PROBLEM

BASED ON WORKSHEETS FROM GLENN SUN
OLGA RADKO MATH CIRCLE ADVANCED 3

FEBRUARY 21, 2021

Today, we will discuss algorithms and the Millennium Prize Problem of P vs. NP. If you can resolve the P vs. NP problem, not only will you be famous for the rest of your life, but you will also immediately get a million dollars!

An *algorithm* is just a set of unambiguous instructions to solve a problem. For example, consider the problem of finding the largest number from a list of numbers. That is, if you are given the list $(9, 9, 222, -10, 3)$, you are supposed to output 222. To have a correct algorithm, your instructions have to work *in general*, no matter what the list is.

For example, if we call the input to the above problem $(x_1, \ldots, x_n)$, the following could work as an algorithm:

```
$m \leftarrow x_1$          // Remember a number called $m$ and set its value to $x_1$
for each $x_i$ in $(x_1, \ldots, x_n)$ do
    if $x_i > m$ then
        $m \leftarrow x_i$      // Update the value of $m$ to whatever $x_i$ currently is
return $m$
```

Problem 1. Give an algorithm to solve the following problems. Don't worry about how fast your algorithm is. Trying everything is a valid solution.

1. Given two sets of integers $A, B \subset \mathbb{Z}$, determine if the two sets have any numbers in common. (This is the DISJ problem, for "disjoint".)
2. Define a *vertex cover* of a finite, undirected graph $G = (V, E)$ to be a subset V' of V such that every edge is connected to some element of V'. Given G, determine whether there is a vertex cover of size k. (This is the VC problem, for "vertex cover".)
3. For a finite, undirected graph $G = (V, E)$, a *connected component* of a graph is a maximal subset V' of V for which there is a path between any two elements of V'. Given G, determine the number of connected components in G. (This is called the CC problem, for "connected components".)

Definition 1. A function $f : \mathbb{N} \to \mathbb{N}$ is said to be $O(g(n))$ (big-O of $g(n)$ or order $g(n)$) if there exist c and N such that $n > N$ implies $f(n) \leq c \cdot g(n)$.

The function f is said to be $\Omega(g(n))$ (omega of $g(n)$) if there exist c and N such that $n > N$ implies $f(n) \geq c \cdot g(n)$.

The function f is said to be $\Theta(g(n))$ (theta of $g(n)$) if it is both $O(g(n))$ and $\Omega(g(n))$. That is, there exist c_1, c_2, and N such that $n > N$ implies $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$.

Problem 2. For each of the following functions $f : \mathbb{N} \to \mathbb{N}$, determine if they are O, Ω, and/or Θ of 1, $\log(n)$, n, n^2, 2^n, and $n!$.

1. $f(n) = 4n^2 + 2n$
2. $f(n) = 1{,}000{,}000$
3. $f(n) = 2\lceil \log n \rceil + 5$
4. $f(n) = 3^n$

Problem 3. Provide a simple, tight bound (meaning using Θ) for the running time of each algorithm you came up with in Problem 1 in terms of a measure n of the size of the input. For Problem 1.1, let $n = \max\{|A|, |B|\}$. For Problem 1.2 and Problem 1.3, let $n = |V|$.

Problem 4 (Challenge, may require calculus)**.** Show that if there exists a constant a such that

$$\lim_{n\to\infty} \left| \frac{f(n)}{g(n)} \right| \leq a,$$

then $f(n)$ is $O(g(n))$. But also, show that even if the limit does not exist, it is still possible that $f(n)$ is $O(g(n))$. (If you take advanced mathematics classes in college, you might learn that there is another kind of limit called lim sup, pronounced lim-soup, that always exists and makes this into an if and only if statement.)

Definition 2. A *decision problem* is a question where the answer is always either yes or no. The class of decision problems P are those that can be solved with an algorithm that runs in polynomial time, in other words, $O(n^k)$ time for some constant k.

Problem 5. Based on the algorithms you wrote, which of the examples in Problem 1 are guaranteed to be in P?

Now let us turn out attention to NP. Here is a motivating example.

Problem 6. The subset sum problem is defined as follows: Given a list of positive integers S and another integer k, determine if there exists a sublist $S' \subseteq S$ that you can add up to get exactly k.

In other words, (S, k) is a yes-instance (meaning the answer should be yes) if and only if there exists a sublist $S' \subseteq S$ that you can add up to get exactly k.

In this question, we're *not* interested determining whether (S, k) is a yes-instance or no-instace. Instead write a polynomial time algorithm that takes (S, k, S') as input and satisfies the following properties:

- If (S, k) is a yes-instance, then there exists S' such that the algorithm returns "yes" when given (S, k, S') as input.
- If (S, k) is a no-instance, then for all S', the algorithm returns "no" when given (S, k, S') as input.

In other words, you just showed that subset sum can be checked in polynomial time, as long as you are provided a *certificate*. In the above case, the certificate was the subset that adds up to exactly k. In general, NP is the class of problems that can be *checked* in polynomial time, and we formalize the notion of checking as follows.

Definition 3. Consider a decision problem and call the input x. To say that the decision problem is in NP, we require that there exists a polynomial time algorithm, called a *verifier*, that takes (x, u) as input and does the following:

- If x is a yes-instance, the algorithm returns "yes" for some certificate u.
- If x is a no-instance, the algorithm returns "no" for all certificates u.

Sometimes, the certificate is also called a proof. The relationship with mathematical proofs is the following:

- If you are given a true statement, then for some proof of the statement, you can read it and verify that the statement is true.
- If you are given a false statement, then for all supposed proofs of the statement, you can find a flaw and be not convinced.

This is exactly what the verifier does for an NP problem.

Contrary to popular belief, the N in NP stands for "nondeterministic," not just "non." The sense in which NP has to do with nondeterminism is an advanced topic that is beyond the scope of today's discussion. But this certificate-verifier definition above will be sufficient for our purposes today.

Problem 7. Show that the following decision problems are in NP by finding a suitable certificate that can be verified in polynomial time.

1. Given a set of objects X, a bunch of subsets $S_1, \ldots, S_n$ of X, and an integer k, determine if you can contain at least one element of, or hit, every subset by picking at most k objects from X. (This is called the hitting set problem.)

2. Given a set of objects of potentially different weights and values, and a bag that can hold at most N units of weight, determine if you can fill the bag with objects having total value at least k. (This is called the knapsack problem.)

Problem 8. Explain why if a problem is in P, then it must also be in NP.

The P vs. NP conjecture simply says that the converse is false. That is, there are problems that can be checked in polynomial time, but cannot be solved directly in polynomial time.

Problem 9. Write an algorithm to solve the subset sum problem. Your algorithm should not run in polynomial time, unless you want to earn $1 million right now.

Problem 10. Try to come up with reasons why you think the subset sum problem cannot be solved in polynomial time. Most likely something about your reasoning will be wrong, so have your instructor or other students find the hole in your reasoning. You can stop when you are convinced that this is a difficult question.

The reason why the previous two problems asked about the subset sum problem is because the subset sum problem is what is called an NP-complete problem. You can think of NP-complete problems as the most difficult problems in NP.

Definition 4. A decision problem is NP-complete if the following two things are true:

1. The problem is in NP, and
2. If this problem could be solved in polynomial time, then any problem in NP can be solved in polynomial time.

It's amazing that NP-complete problems exist at all, but Stephen Cook and Leonid Levin proved in 1971 that they do exist, and in fact they found dozens of interesting NP-complete problems. It turns out that subset sum is one of these NP-complete problems.

NP-complete problems are relevant to the P vs. NP conjecture because if you can find a polynomial time algorithm to any NP-complete problem, it would disprove the conjecture, since it would imply that any problem in NP is solvable in polynomial time.

Problem 11. If you found a polynomial time algorithm to an NP problem that was not NP-complete, would it resolve the question of whether or not P is equal to NP?

Problem 12. Given a list of positive integers S, determine if S can be split into two sublists that have equal sum. This is called the partition problem. Using the fact that subset sum is NP-complete, prove that the partition problem is also NP-complete. (This kind of argument is called a *reduction*.)

1. Prove that the partition problem is in NP.
2. Show that if partition problem can be solved in polynomial time, then the subset

sum problem can be solved in polynomial time. Here are the steps to take:

   (a) Let a denote the sum of the elements in S. Show that $S \cup \{a - 2k\}$ (the disjoint union) can be split into two sublists of equal sum if and only if S has a sublist that sums to k.

   (b) Use the above fact to solve subset sum using partition.

3. Conclude that the partition problem is NP-complete.

These types of reductions are a common way of showing a problem is NP-complete. Once enough problems are shown to be NP-complete, we can try to find a reduction from a new problem to one of the known problems.

Let x_i denote a Boolean variable that takes values 0 (false) or 1 (true). We will write $\overline{x_i}$ for the negation of x_i. We will call a Boolean variable or its negation a *literal*. A *clause* is a sequence of literals connected by OR statements like $x_1 \vee x_2 \vee \overline{x_3}$. The *conjunctive normal form* of a Boolean formula is a sequence of clauses connected by AND statements. For example, $(x_1 \vee x_2 \vee \overline{x_3}) \wedge (x_1 \vee \overline{x_2} \vee x_3)$ is a Boolean formula in conjunctive normal form with two clauses. The `SAT` problem, which stands for Boolean satisfiablility, asks whether a Boolean formula in conjunctive normal form can ever output 1.

Problem 13. Show that the Boolean formula in conjunctive normal form

$$(\overline{x_1} \vee x_2 \vee x_3) \wedge (x_1 \vee \overline{x_2} \vee x_3) \wedge (x_2 \vee x_3) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_3})$$

is satisfiable.

Problem 14. Show that `SAT` is NP.

Theorem 5 (Cook-Levin)**.** `SAT` is NP-complete.

The proof to Cook-Levin Theorem is not extremely difficult, but it is beyond the scope of this worksheet. For an outline of the proof, see `http://pages.cs.wisc.edu/~shuchi/courses/787-F07/scribe-notes/lecture09.pdf`.

We can reduce `SAT` to many important NP problems. With Cook-Levin Theorem, these reductions imply NP-completeness. For instance, we will show that the following clique problem and `VC` are NP-complete.

Problem 15. Let $G = (V, E)$ be an undirected graph. A *clique* is a subset of vertices $V' \subseteq V$ such that each pair of vertices in V' is connected by an edge. The clique problem is to determine whether a clique of size k exists in the graph G. Show that the clique problem is NP.

In order to show the clique problem is NP-complete, we are going to reduce `SAT` to the clique problem. As a result, we will need some way to translate a Boolean formula with k clauses to a graph G. We want inputs that satisfy the formula to correspond to cliques of size k in G.

Problem 16. Construct a graph G for the formula in Problem 13 so that a satisfying input corresponds to a clique of size 4 in G. Label the clique of size 4 that represents the solution you found.

Problem 17. Given a Boolean formula in conjunctive normal form with k clauses, show how to construct a graph G in such a way that:

1. If the formula is satisfiable, then G has a clique of size k.
2. If G has a clique of size k, then the formula is satisfiable.
3. G can be constructed from the formula in polynomial time.

Now use the above to show that the clique problem is NP-complete.

Problem 18. Recall `VC` from the second part of Problem 1. Show that `VC` is NP.

Problem 19. Find a reduction from the clique problem to `VC`. Conclude that `VC` is NP-complete. (Hint: The complement $\overline{G}$ of a graph G is a graph with the same vertices and only the edges that do not occur in G.)

There are now hundreds, or perhaps thousands, of known NP-complete problems. People have even proved that generalized versions of games like Battleship and Sudoku are NP-complete! And remember, showing any one of the NP-complete algorithms is $O(n^k)$ for some k is worth a million dollars!