THE P VS. NP PROBLEM

GLENN SUN

OLGA RADKO MATH CIRCLE ADVANCED 2

APRIL 10, 2021

1 Main Problems

In 2000, the Clay Mathematical Institute put a $1 million prize on 7 important problems in mathematics, called the Millennium Prize Problems. One of them is the P vs. NP question, a question in theoretical computer science: if a problem can be checked quickly by an algorithm, can it also be solved quickly by an algorithm? You might know from experience that it takes a lot longer to solve your homework by yourself than to copy the answers from a friend and just check that the answers are right. Hence, most people believe the answer to the question is no, but where would you even start to try proving this?

There are lots of words that we need to define first: what is a *problem*, what does it mean for an *algorithm* to *solve it* or *check answers*, and how fast do we mean by *quickly*?

An *algorithm* is just a set of unambiguous instructions to solve a problem (think like a robot). For example, consider 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 this problem $(x_1, \dots, x_n)$, the following could work as an algorithm:

```
1. Let $m = x_1$.
2. For each $x_i$ in $(x_2, \dots, x_n)$,
    (a) If $x_i > m$, update $m$ to equal $x_i$.
    (b) Otherwise, go on to the next element.
3. Output $m$ as the maximum number.
```

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 a list of distinct integers $(x_1, \dots, x_n)$, find the i such that x_i is the largest number in the list.
2. Given two lists of numbers $(a_1, \dots, a_n)$ and $(b_1, \dots, b_n)$, determine if the two lists have a common element.

Solution.

1. Same as example, except remember and update both m and i_{max}.
2. For every a_i and every b_j, check if $a_i = b_j$ and output true if the check passes.

   Students who know sorting may suggest to first sort the two lists, then go through both simultaneously. This is faster but not required for the solution. $\square$

We will very often look at problems involving graphs. Recall that a graph $G = (V, E)$ is a collection of vertices and edges. When a graph is the input to a problem, you should think that we have for every vertex, a list of the vertices it is directly connected to by an edge.

Problem 2. Give an algorithm for the following problems. Don't worry about how fast your algorithm is. Trying everything is a valid solution.

1. A graph is *connected* if there is a way to walk between any two vertices using edges. Given a graph, determine if it is connected.
2. A *coloring* of the graph is a way to assign colors to the vertices such that every edge has different colors on the endpoints. Given a graph, determine if it can be colored using just 3 colors.

Solution.

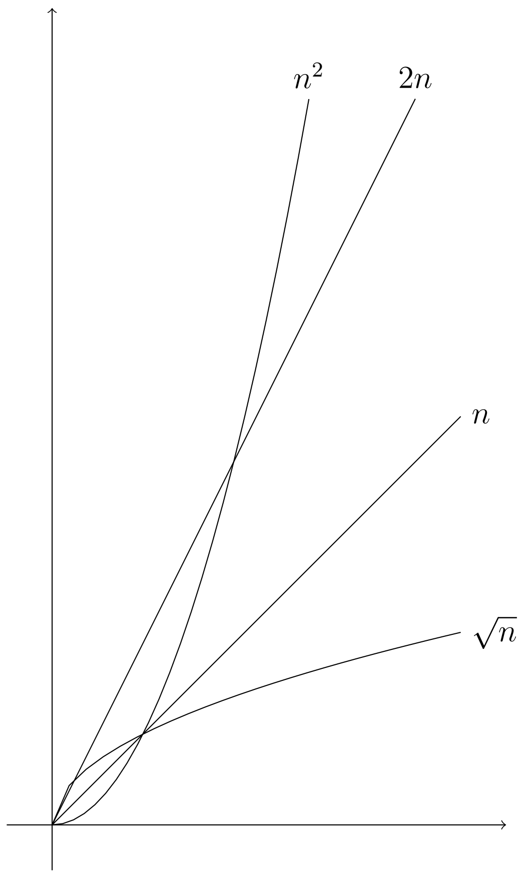
1. Choose an arbitrary starting vertex and keep some way to keep track of whether a vertex is already visited, discovered but not visited, or undiscovered (i.e. a list whose elements take 3 different values). Choose an arbitrary vertex, set it to visited, and discover all of its neighbors. At every step, choose a discovered but unvisited vertex, mark it as visited, and discover all of its neighbors. When there are no more discovered but unvisited vertices, the graph is connected if every vertex is visited and disconnected otherwise.

   This problem should be intuitively clear how to do but students might have trouble being more formal. Don't worry too much about the formality.
2. For every 3-coloring of V, loop through all edges and check if the colors at the endpoints are the same. $\square$

Next, we analyze how fast algorithms run.

Definition 1. A function $f : \mathbb{N} \to \mathbb{N}$ is said to be $O(g(n))$ (big-O of $g(n)$) if for large n, $f(n)$ is at most a constant multiple of $g(n)$.

We think of $g(n)$ as a loose upper bound of $f(n)$. In the below picture, $\sqrt{n}$ is $O(n)$ because it is eventually always less than n. Although $2n$ is always greater than n, it is bounded by constant multiple of n, so it is also $O(n)$. The n^2 line is not $O(n)$ because it grows uncontrollably compared to n: for any c, n^2 is eventually greater than cn (when $n > c$).



Problem 3. Are the following functions $O(1)$? $O(n)$? $O(n^2)$? $O(2^n)$? $O(n!)$?

1. $f(n) = 1,000,000$
2. $f(n) = \sqrt{n}$
3. $f(n) = 4n^2 + 2n$
4. $f(n) = 3^n$

Solution.

1. The function $f(n) = 1,000,000$ is all of them.
2. The function $f(n) = \sqrt{n}$ is $O(n)$, $O(n^2)$, $O(2^n)$, and $O(n!)$.
3. The function $f(n) = 4n^2 + 2n$ is $O(n^2)$, $O(2^n)$, and $O(n!)$.
4. The function $f(n) = 3^n$ is $O(n!)$. □

We use big-O notation to analyze algorithms because algorithms can get complicated, and we want an easy way to roughly talk about how fast they are without all the constants. For example, in our algorithm to find the largest number in a list,

- Steps 1 and 3 take a constant amount of time to do.

- The inside of step 2 takes at most some constant amount of time to do every iteration.
- The inside of step 2 is repeated n times.

So the total amount of time it takes is at most something like $c_1 + nc_2$, where c_1 and c_2 are some constants. Therefore, we say that this algorithm is $O(n)$.

Problem 4. For each algorithm you wrote in Problems 1 and 2, give a rough upper bound on how many steps are taken on an input of size n using big-O notation. (For graphs, $n = |V|$. Remember that there are at most $\binom{n}{2}$ edges.)

Solution. With the example solutions, 1.1 takes $O(n)$ time, 1.2 takes $O(n^2)$ time with the first example solution, 2.1 takes $O(n^2)$ time, and 2.2 takes $O(n^2 3^n)$ time. □

Definition 2. An algorithm runs in *polynomial time* if its running time is $O(n^k)$ for some constant k. (Polynomial time is what we meant by "quickly.")

For example, $n^4 + 3n^2 + 1$ is polynomial time, but 2^n is not polynomial time. (A proof of the second fact requires calculus, so it's in the challenge problems.)

Problem 5. Based on the algorithms you wrote, which of the examples in Problems 1 and 2 run in polynomial time?

Solution. 1.1, 1.2, and 2.1. □

Definition 3. A decision problem is informally, a question to which the answer is always yes (T) or no (F). Formally, a decision problem is a function $f : X \to \{\mathsf{T}, \mathsf{F}\}$, where X is the set of all possible inputs. We say that $x \in X$ is a T-instance if $f(x) = \mathsf{T}$ and similarly F-instance if $f(x) = \mathsf{F}$.

An algorithm solves a problem f if for all inputs $x \in X$, the algorithm outputs $f(x)$. The set P is the set of all decision problems that can be solved in polynomial time.

Problem 6. Based on the algorithms you wrote, which of the examples in Problems 1 and 2 belong to P?

Solution. 1.2 and 2.1. □

The last notion we have to define is what it means to check a solution.

Definition 4. The set NP is the set of all decision problems $f : X \to \{\mathsf{T}, \mathsf{F}\}$ with an polynomial-time algorithm that "checks proofs of $f(x) = \mathsf{T}$," i.e.

- The input consists of both some $x \in X$, and an attempted *proof* y.
- For all T-instances, there exists a proof for which the algorithm outputs T.

- For all F-instances, the algorithm outputs F for all proofs.

An example of a problem in NP is the subset sum problem. It says: given a set of positive integers S and another integer k, determine if there exists a subset $S' \subseteq S$ that you can add up to get exactly k.

Letting S' be the proof, all we need to do to check that S' is a valid proof is to check that $S' \subset S$ and the sum of the elements in S' is k. This is easy to do in polynomial time, and there exists a proof for x if and only if x is a T-instance, as we wanted.

Note that the original question of determining if there exists S' seems to require much more computation — there are 2^n subsets of S (where $n = |S|$), so it is not possible to try them all in polynomial time.

Problem 7. Show that the following problems belong to NP by identifying a proof for every T-instance x, and giving an algorithm to check the proof.

1. The 3-coloring problem from Problem 2.
2. Given a positive integer N that is n digits long, determine if N is composite. (The number of digits is the input length.)

Solution.

1. The proof is an assignment of colors to the vertices. To check that this is a valid proof, we loop through all the edges and check that the colors are different at the two endpoints. This takes up to $O(n^2)$ time.
2. The proof is a non-trivial factor of N. To check that this is a valid proof, divide N by the factor and check that the remainder is zero. But running time of the division algorithm might not be immediately obvious.

   Alternative, the proof is a nontrivial factorization of $N = pq$. To check that this is a valid proof, we can use schoolbook multiplication to compute pq. For n-digit p and q, schoolbook multiplication takes $O(n^2)$ time. □

Problem 8. Prove that $\mathsf{P} \subset \mathsf{NP}$.

Solution. The polynomial algorithm that "checks proofs" can ignore the proof and just compute the answer by itself. □

The P vs. NP problem asks if the reverse inclusion $\mathsf{NP} \subset \mathsf{P}$ is true. That is, are there problems that can be checked in polynomial time, but cannot be solved directly in polynomial time? Most people think this is *false*, but mathematicians and computer scientists still don't know for sure.

Definition 5. A *reduction* from f to g is a way to solve f assuming you know how to solve g.

Problem 9. In this problem, we will give a reduction from 3-coloring to 4-coloring.

1. Let G be a graph. Construct a graph H such that G can be 3-colored if and only if H can be 4-colored. (Hint: add a vertex.)
2. Use the above the give a reduction from 3-coloring to 4-coloring.

Solution.

- Let H be G with a vertex added that is connected to all other vertices in G by an edge. Then any proper coloring of H gives the new vertex its own color. The coloring of the remaining vertices is a 3-coloring of G.
- To solve 3-coloring, given a graph G, construct H as above. Run the 4-coloring algorithm to determine if H can be 4-colored, and output exactly what that algorithm says. □

Definition 6. A problem g is NP-complete if $g \in \mathsf{NP}$ and all other problems in NP can be reduced to g in polynomial time.

Problem 10.

1. It is a fact that 3-coloring is NP-complete. (See challenge problems for a proof.) Show that 4-coloring is NP-complete.
2. Explain why if g is an NP-complete problem, then to prove that $\mathsf{NP} \subset \mathsf{P}$, it suffices to show that $g \in \mathsf{P}$.

Solution.

1. Take any problem $f \in \mathsf{NP}$. First, we reduce f to 3-coloring because 3-coloring is NP-complete. Then we can solve 3-coloring with 4-coloring by the previous question.
2. To show that $\mathsf{NP} \subset \mathsf{P}$, let $f \in \mathsf{NP}$. Because g is NP-complete, there exists a polynomial time reduction from f to g. Because $g \in P$, there exists a polynomial time algorithm to solve g. Because addition/multiplication/composition of polynomials is a polynomial, it takes polynomial time to solve f using g, so $f \in \mathsf{P}$. □

Part 2 above is also the reason why we want to consider polynomial time as our definition of "quickly." Unlike, say, the set of problems that can be computed in $O(n^2)$ time, polynomial time is closed under composition and multiplication, which is very useful.

Problem 11. The partition problem is: Given a set of positive integers S, determine if S can be split into two subsets that have equal sum. Using the fact that subset sum is NP-complete, we will prove that the partition problem is also NP-complete.

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 also be solved in polynomial time. (Hint: Let a denote the sum of the elements in S and consider $S \cup \{a - 2k\}$.)
3. Conclude that the partition problem is NP-complete.

Solution.

1. The two subsets form a proof, since you can add their elements and check that the sums are equal in polynomial time.
2. In order for two subsets of $S \cup \{a - 2k\}$ to have the same sum, they must each have sum $a - k$. One of the subsets must contain the number $a - 2k$. Hence the remaining elements of that subset, which are in S, sum to k.

   In other words, there exists a subset of S that sums to k if and only if the set $S \cup \{a-2k\}$ can be partitioned, so we run the partition algorithm on $S \cup \{a - 2k\}$ that exists by assumption and output the same thing.
3. For any problem in NP, we can first reduce it to subset sum by assumption, and then we can reduce the subset sum problem to the partition problem by above. So any NP problem can be reduced to partition, so the partition problem is NP-complete. $\square$

Many practical problems are NP-complete, so this is a very important class of problems. Here are some examples:

- The 3-coloring problem from Problem 2.
- The subset sum problem from the example above.
- Scheduling problems: Given a set of students who each want to take several classes and a limited number of timeslots, decide if the classes can be scheduled to avoid conflicts.
- Traveling salesman problem: Given a map and a list of cities, find the shortest route that visits all the cities. (Decision version: decide if a route of length $\leq k$ exists.)
- Knapsack problem: Given a knapsack with a weight limit and a list of items with various weights and values, determine which items to put in the knapsack to maximize the value. (Decision version: decide if there exists items to get value $\geq k$.)
- Generating crossword puzzles: Given a set of words and a desired crossword grid, determine if the white squares (not the black squares) can be filled such that every maximal horizontal and vertical sequence of white squares is a word from the set.

2 Challenge Problems

Problem 12. Prove that for any positive integer k,

$$\lim_{n\to\infty} \frac{n^k}{2^n} = 0.$$

Conclude that 2^n is not $O(n^k)$ for any constant k.

Solution. Use L'Hopital's rule k times to get $\lim_{n\to\infty} \frac{k!}{2^n \log(n)^k}$. The numerator is constant and the denominator is increasing in n, so this goes to 0. By definition of limit, this means that for all $\frac{1}{c} > 0$, we have $\frac{n^k}{2^n} < \frac{1}{c}$ for sufficiently large n. Rearranging gives $2^n > cn^k$ for all c when taking n sufficiently large, so 2^n is not $O(n^k)$. $\square$

Problem 13. Come up with a problem that is (provably) not in P. (Hint: use a diagonalization argument, like in the rational numbers worksheet. Ask for more hints if you're stuck.)

Solution. Given the program α and polynomial p, let $\pi_{\alpha,p}$ the program that on input x, simulates $\alpha(x)$ for $p(|x|)$ steps, and outputs F if the program doesn't halt in time.

Let f be the problem defined $f(\alpha, p) = \mathsf{T}$ if $\pi_{\alpha,p}(\alpha, p) = \mathsf{F}$, and $f(\alpha, p) = \mathsf{F}$ otherwise. Suppose for contradiction f can be solved in polynomial time by a program α. Then there exists p such that $\pi_{\alpha,p} = \alpha$.

Then if $\alpha(\alpha, p) = \mathsf{T}$ by running α, we needed $\alpha(\alpha, p) = \mathsf{F}$ by definition of f, a contradiction. Similarly, if $\alpha(\alpha, p) = \mathsf{F}$ by running α, we needed $\alpha(\alpha, p) = \mathsf{T}$ by definition of f, again a contradiction. $\square$

Problem 14. In this problem, we will prove from scratch that 3-coloring is NP-complete. This is a very difficult problem, so feel free to ask your instructor for hints.

First, we define the SAT (satisfiability) problem. Let $\wedge$ denote "and," $\vee$ denote "or," and $\neg$ denote "not." A Boolean formula in CNF form looks something like:

$$(x_1 \vee \neg x_2 \wedge x_3) \wedge (\neg x_4 \vee x_1) \wedge (x_5 \vee x_6 \vee x_2 \vee x_3)$$

Formally, it is a $\wedge$ of *clauses*, each of which is the $\vee$ of possibly negated variables. The SAT problem asks: Given a Boolean formula φ, is there an assignment of T/F values to the x_i that makes the formula true?

1. Show that SAT is in NP.
2. Let $f \in \mathsf{NP}$. Assuming that you know how to solve SAT, sketch an argument to show how to solve f. This shows that SAT is NP-complete.
3. 3SAT is the version of SAT where every clause has as most 3 variables. Show that 3SAT is NP-complete by reducing SAT to 3SAT.

4. Show that 3-coloring is NP-complete by reducing 3SAT to 3-coloring.

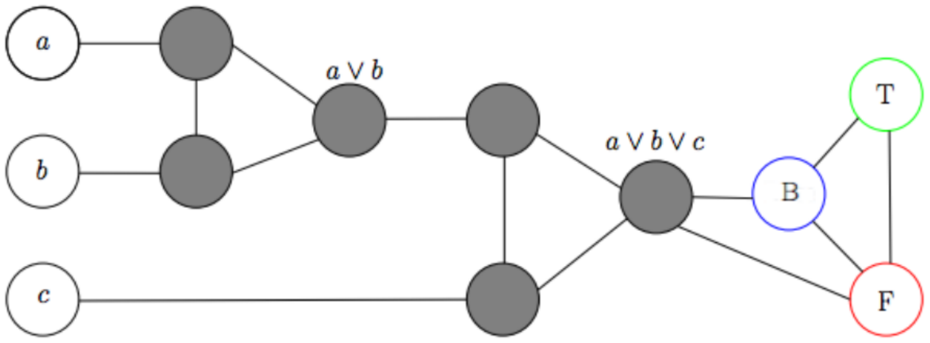
Solution.

1. The certificate is the assignment.
2. We are given an algorithm that checks proofs of "$f(x) = \mathsf{T}$". We write a Boolean expression that is satisfiable if and only if there exists a proof of $f(x) = \mathsf{T}$. We simply write an expression to check that the algorithm is always doing it job correctly at every step, and that the output of the algorithm is T. The expression being satisfiable means that the algorithm can output T, which means there exists a proof that $f(x) = \mathsf{T}$ by definition of NP.

   As you can probably tell, it takes a lot of tedious writing to spell out an expression to check that each step of the algorithm is doing the right thing. For example, you can think of a giant table where each row is the algorithm's memory at a particular point in time. Then, you can ask each memory cell to follow from the cells above it in a predictable way with a short formula, then take and over all cells.
3. Replace long clauses $x_1 \vee \cdots \vee x_m$ with

$$(x_1 \vee x_2 \vee z_1) \wedge (x_3 \vee \neg z_1 \vee z_2) \wedge (x_4 \vee \neg z_2 \vee z_3) \wedge \cdots \\ \cdots \wedge (x_{m-2} \vee \neg z_{k-4} \vee z_{k-3}) \wedge (x_{m-1} \vee x_m \vee \neg z_{k-3})$$

   This is not logically equal to the original clause, but there exists an assignment that satisfies it if and only if there exists an assignment that satisfies the original clause.
4. Given a 3SAT formula, first construct vertices $\mathsf{T}, \mathsf{F}, b$ and connect them to each other (so that they are different colors). Then construct vertices v_i and $v_{\neg i}$ for each variable x_i and connect them by an edge (so they will be assigned different colors), and connect them both to b (so they are assigned T or F). For each clause, construct the following gadget (source: University of Toronto):



□