Proofs in Three Bits or Less

Math Circle

October 25, 2021

We started Math Circle today by introducing computational complexity, focusing on how 'complex' certain *algorithms* are. To do this, we introduced the idea of a *worst case analysis of the problem*; with any algorithm, we associate a function T, such that for each positive integer n, $T(n)$ gives the most number of steps an algorithm will take to compute an answer to a problem of 'size n' (the size depends on the problem; in the case of a list of numbers, we might take n to be the *size* of the list). To make our problems more insensitive to ambiguity, and to focus on the important parts of the function, we introduced the *computational complexity classes* $\Theta(1)$, $\Theta(n)$, $\Theta(n^2)$, etc, which give the 'highest order of growth' of the function. In these first set of questions, you'll find algorithms to solve problems, and investigate the complexity classes of the algorithms you create.

1 Complexity Theory

The world is a thing of utter inordinate complexity and richness and strangeness that is absolutely awesome.

Douglas Adams

1. In the beginning of the talk, we discussed two basic algorithms for determining whether a list of numbers (for simplicity, two digit numbers) contained a particular value. Assume that checking whether two digits numbers are equal takes two basic computational steps. Can you think of a list of numbers where the first algorithm is able to finish *faster* than the second algorithm (it finishes it's computation in fewer steps), and another list of numbers where the second algorithm is able to finish *faster* than the first algorithm?

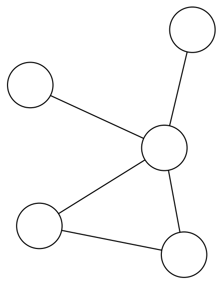
2. Again consider the problem of determining whether a list of numbers contained a particular value. The two algorithms above have complexity $\Theta(n)$, where n is the size of the list. Thus if you were given a list of 1000 numbers, in the worst case it would take approximately 1000 to 2000 steps to determine whether this list contained a particular number. Now assume that instead of an arbitrary list, we are given a *sorted* list of numbers. Can you find a faster algorithm to determine if a particular number is in the list? The fastest algorithm to determine this should take roughly 10 to 20 steps to finish in the worst case. If you apply your list to a sorted list of n numbers, what is the worst case number of steps your algorithm could take to conclude. Equivalently, what is the complexity class of your algorithm?

3. The *Fibonacci* numbers are the infinite sequence

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \dots,$$

   where each number in the sequence is obtained by adding the previous two numbers in the sequence. Find an algorithm to compute the nth Fibonacci number. What is the complexity class of this algorithm? Can you find a faster algorithm?

4. The next problem discussed a problem about *graphs*. A graph is a collection of *vertices*, together with *edges* connecting them. Two vertices are *adjacent* if there is an edge connecting them.



Graphs are super useful in many areas of mathematics, and many applications of mathematics to problems in the real world. You could model the streets of Madison as a graph, where each vertex is an intersection of roads, and two intersections are connected if there is a road leading directly from one intersection to the other. You could also use graphs to model relationships between people. Each vertex could correspond to a person, and two vertices could be connected if they correspond to people that are friends. Or you could form a family tree, where two edges connect parents to their children. Mathematical problems about graphs would then tell you about the streets of Madison, or the relationships between people.

A *colouring* of a graph associates with each vertex of the graph a particular colour, such that no adjacent vertices share the same colour. If we consider a graph where each vertex corresponds to a country in the world, and two countries are adjacent if they share a border with one another, then a colouring would be precisely a colouring you could use on a world map (it would be difficult to distinguish between two countries on a map bordering one another if they shared the same colour).

If the colouring only uses two colours (say, red and blue) then we say that the colouring is a *two colouring*. Can you find an algorithm to determine whether a graph is two colourable? What is it's complexity class? Can you find an algorithm to determine whether a graph is three colorable?

5. (Bonus Question) Given a graph, and two vertices in that graph, can you find an algorithm that finds the *shortest number of edges* that connect the two vertices? What is it's complexity class?

6. (Bonus Question) We say a problem is 'in **P**' if there exists an algorithm to compute the answer to a problem in time $\Theta(n^k)$ for some $k \geqslant 0$ (finding an element of a list is in **P** because we have an algorithm that takes $\Theta(n)$ time to compute an answer). Almost all of the problems in this section are 'in **P**'. Which problem were you not able to verify was in **P**.

2 Interactive Proofs

What is intuitively required from a theorem-proving procedure? First, that it is possible to "prove" a true theorem. Second, that it is impossible to "prove" a false theorem. Third, that communicating the proof should be efficient, in the following sense. It does not matter how long must the prover compute during the proving process, but it is essential that the computation required from the verifier is easy.

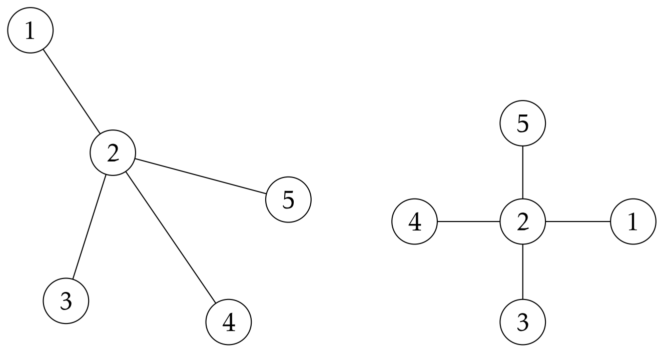
Goldwasser, Micali, Rackoff

In the last section, we considered algorithms which took in problems, and found solutions to those problems. In this section, we consider algorithms that *take in a solution as input*, and determine if this solution is a correct solution. We call this *proof verification*, since a solution is *proof that something is true*, and we are checking whether this proof is legitimate (we only care about checking proofs that something is true, since checking proofs that something is false can often be a very different problem). One way we can think of this is checking these solutions for mistakes. More cynically, we can think of the solver of the problem as perhaps maliciously lying about their knowledge. This is especially important nowadays, when computers verify the identities of one another by sending solutions to very difficult math problems, that could only be solved by a particular individual with possession of some help solving the problem (a password). Either with mistakes or with maliciousness, we should be thorough in how we check our solutions. We need our proof verification algorithms to be both:

- *Perfectly Complete*: If a solution is correct, we will verify the solution correctly.
- *Perfectly Sound*: If a solution is incorrect, we will throw out the solution.

Let's consider some examples of proof verification.

1. In the last set of problems, we considered the *three coloring problem*, which takes $\Theta(3^n)$ steps to find a coloring if it exists, or to show no coloring exists. Can you find an algorithm that takes in as input a three coloring for the problem, and then *checks* whether a three coloring is valid. What is it's complexity?
2. Consider an algorithm that takes in an input a proof that a particular number is in a list, and checks whether this proof is valid? What is it's complexity? What would a proof that a number is *not* in a list look like?
3. We say two graphs are *equivalent* if, after rearranging the vertices of the graph, you have the same graph. For instance, the following two graphs are equivalent:



   Given two graphs, what would a 'proof' that two graphs are equivalent look like? What would a 'proof' that two graph are *not* equivalent look like?

A problem is 'in **NP**' if there exists a way of checking solutions to a problem that have time complexity $\Theta(n^k)$ for some $k \geqslant 0$. In question 6 of the last section, we consider the class of problems 'in **P**'. Intuitively, it is easier to 'mark an exam' then to 'take an exam'. We even have phrases in english, such as 'it's easier to be a critic than a creator'. Thus one might conjecture that the class of problems **NP** contains more difficult problems than the class **P**. We might write this as $\mathbf{P} \neq \mathbf{NP}$. But despite being posed over 50 years ago, it is still completely unknown whether $\mathbf{P} = \mathbf{NP}$, or whether $\mathbf{P} \neq \mathbf{NP}$. At the turn of the millenium, a one million US dollar prize was offered to anyone who might be able to resolve this problem. But we are still nowhere close to resolving the problem. It's often the most fundamental problems in mathematics that remain unsolved for so long.

3 Probabilistically Checkable Interactive Proofs

Suppose a mathematician circulates a proof of an important result, say Riemann Hypothesis, fitting several thousand pages. To verify it would take you and your doubting colleagues several years. Can you do it faster? Yes, according to the PCP Theorems. He can rewrite his proof so you can verify it by probabilistically selecting 3 bits to examine in it.

Sanjeev Arora

Suppose we don't want to be as thorough in our marking. If we only want to spot *most* mistakes, is the complexity of our problem reduced? Let's be precise about how we do this. It turns out that, amazingly, you can reduce the complexity of the problem *immensely*, such that every problem 'in **NP**' can be checked in *at most 3 steps*. In three steps, we can only really ask three 'yes or no' questions about the solution to the problem. So we can think of these proof verifiers as being almost like players of the game '20 questions', or 'Guess Who'. We must extract the maximal amount of information about a solutions to a problem in as minimal a number of questions as possible.

Let's be precise about the amount of error we are allowed in the problem. We consider a *random* algorithm, that chooses a set of questions to ask in a random manner, such that we have:

- *Imperfect Completeness*: If a statement is true, then we will accept the solution with probability $\geqslant 1/2$
- *Imperfect Soundness*: If a statement is false, then we will reject the solution with probability $\leqslant 1/2$.

The particular probability bound $1/2$ doesn't really matter here, since iterating the algorithm repeatedly will decrease the amount of error in the problem; if we run the algorithm m times in a row, then we will accept true solutions with probability $\geqslant 1 - 1/2^m$, and reject false solutions with probability $\leqslant 1/2^m$. We call such an algorithm a *probabilistically checkable proof*.

Here's an example of a probabilistically checkable proof, which examines a solution to the problem of when two graphs are *not equivalent*, as in Question 3. After puzzling aruond with this problem, you might have seen that a proof that two graphs are not equivalent with *perfect completeness* and *perfect soundness* (no errors) can be very very long. But we can check whether two graphs G and H are not equivalent with *a single question*. The idea is simple. We flip a coin, and pick one of the graphs G and H depending on the outcome of the coin toss. Then we shuffle all of the vertices and edges of the graph, thus hiding which graph we originally chose. We ask the prover of the non-equivalence statement to identify which of the graphs we began with as part of the solution they are submitting to us. If the prover is correctly able to identify the original graph, we accept their solution. If the prover incorrectly identifies a graph, we reject their solution.

This interactive proof has *perfect completeness*; if the prover gives a valid solution to the problem, we will always accept their solution. It also has *imperfect soundness*: Suppose that a person *claimed* to have shown the two graphs were not equivalent, but that the graphs were in actuality equivalent to one another. In the interactive proof above, since the two graphs are equivalent, the prover will have no way to tell which of the two graphs we began with, since the problem we gave them to solve is equally likely to have come from either of the two equivalent graphs. Thus the prover is bound to fail with probability at least $1/2$.

Let us try and consider a probabilistically checkable proof that two graphs are three colorable. We might ask the prover to submit a three coloring. We then select two adjacent vertices at random, and check whether they are the same color. If they are, we reject the solution. If they aren't we accept the solution. Since we only check the colors of *two of the vertices*, this probabilistically checkable proof only asks two questions. This algorithm has perfect completeness, since a correct solution will never be rejected. If the graph is not three colorable, there will be at least one pair of adjacent vertices which are colored the same, so our algorithm will reject this solution with probability $\geqslant 1/E$, where E is the number of edges in the graph. This doesn't quite give imperfect soundness, since $1/E$ can be much smaller than $1/2$, unless we had some guarantee that every coloring of the graph would fail on at least half of the adjacent edges. We call such a graph an *expander*. Thus we have a probabilistically checkable proof, at least for the class of expander graphs.