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

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

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

   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

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?

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

s