Hodge Theory

Jacob Denson

October 25, 2021

This is a talk about *computational complexity*. Since you began learning mathematics, you've trained yourselves to compute various different quantities; at the beginning, you learned how to compute the sum $a + b$, the product $a - b$, and the quotient a/b of any two numbers a and b. Then you might have learned how to compute the roots of a quadratic polynomial $ax^2 + bx + c$. You might have also learned how to find the prime factors of an integer n. Or perhaps computing the nth Fibonacci number. The theory of computational complexity asks us to measure how *complex* these computations are. Let's start by thinking intuitively: which of these computations is *hardest*, and which one is *easier*. Why?

1 Complexity Theory

An *algorithm* is a sequence of steps that allow you to compute some quantity. Our goal is to figure out which algorithms are more complex than other algorithms. The main way of doing this is quantifying how many steps an algorithm takes to compute an answer.

To figure out how to do this precisely, let's consider a very simple example. Consider a list of numbers

$$81, 88, 62, 46, 58, 69, 94, 72, 64, 63, 71, 53, 26, 27, 25, 57, 44, 30, 31, 79, 66.$$

Is 83 in this list? After you've solved it, think about what *algorithm* you used to determine whether 83 was in this list?

- Check whether each element of the list is 83.
- If it is, then answer yes.
- If no, then there isn't.

We have to check each integer in the list, so since the list has 21 elements, it takes 21 comparisons of integers to solve the problem. Given any n element list, this algorithm will (*in the worst case*), takes $T(n) = n$ comparisons to perform (it might take fewer if 83 is actually in the list). The function T, which takes the size of a problem, and outputs the worst case number of steps to solve the problem, is known as the *complexity function* of the algorithm. The idea of complexity theory is that, the faster T grows, the more complex the problem is.

Another algorithm to solve the problem might be to quickly scan through the *first digits* of the numbers in the list, filter out the numbers with 8 as the first digit, then check those numbers to see if the second digit is a 3. In the worst case, this algorithm takes $T(n) = 2n$ comparisons to complete. But one might argue that checking the first digits of two numbers can be done faster than checking whether two digits numbers are completely equal (we only have to compare one digit, rather than two). If we viewed comparing double-digit numbers as taking two steps, rather than a single step, the first algorithm would also take $T(n) = 2n$ steps in the worst case. Thus the two algorithms have the same complexity function.

The first algorithm might perform faster on some list (which ones?) and the second algorithm might perform faster on other lists (which ones?). One way to determine which algorithm is faster is to have a competition. We could either have two people try and perform the algorithm on a 200 element list, and see who is faster. Or we could get two computers to perform the algorithm on a *very long list*, and see which is faster. Which is faster might depend on the list, which people are competing, or which computers are being used to perform the calculation. Thus it is *ambiguous* which problem is more complex. Theoretically, in complexity theory, we get around this by only caring about the *order of growth* of the complexity function. We view the complexity function $T_1(n) = n$ and $T_2(n) = 100n$ as being *equally complex* (though in practice we might want to use the algorithm associated with the first complexity function). We say that T_1 and T_2 both lie in the *complexity class* $\Theta(n)$ (they both have 'order of growth n'). This also gets us around the ambiguity of what we consider as a 'basic step' in the problem (we can view checking if two numbers are equal as a single step, or as a sequence of two steps) since the order of growths with respect to both measures is $\Theta(n)$. As other examples, $T(n) = 25n^4$ lies in $\Theta(n^4)$, as does $T(n) = 100n^4 + 20n^3 - 10n$.

This is because the order of growth is the most important part of the complexity function *for large n*. Consider the two complexity functions $T_1(n) = 10n$ and $T_2(n) = n^2$. The 100 surely matters for small values of n. A human can perform about 3 steps of an algorithm per second. Thus in 10 seconds, a human can only compute the algorithm with inputs ranging up to size as most $n = 3$, whereas in the second algorithm, a human can compute the algorithm with input sizes up to $n = 5$. However, in an hour, a human could compute the first algorithm with n ranging to $n = 1080$, whereas the second can only compute values up to $n = 103$, much much smaller. This becomes even more prominant when we use computers to compute values, since, roughly speaking, a computer can compute $10,000,000$ steps of an algorithm per second. Thus in a single second, a computer could compute instances of the algorithm of size up to $n = 1,000,000$, whereas in the second, a computer could only compute instances of size at most $n = 3162$. Thus order of growth is really only important for large problems, and is especially only important when we use computers to follow through algorithms.

2 Your Turn

Now's your chance to find algorithms for problems, and to determine their complexity. Suppose that you were given a *sorted* list of 1000 numbers. The algorithms above would take $\approx$ 1000 steps to check if this list contains the number 82. Can you find an algorithm which takes much less than 1000 steps to check if this list contains the number 82 (say $\approx$ 10 steps)? If you apply your algorithm to a sorted list of n elements, what is the function $f(n)$ such that your algorithm has complexity class $\Theta(f(n))$?

The *Fibonacci numbers* are the sequence $1, 1, 2, 3, 5, 8, 13, \dots,$ where each number is the sum of the two previous numbers. Find an algorithm to compute the nth Fibonacci number. What is the complexity class of this algorithm? Can you do better than this?

Now let's consider a problem about *graphs.* A graph is a collection of *vertices*, together with *edges* connecting them. Two vertices are *adjacent* if an edge connects them.

DIAGRAM HERE

Graphs are super useful in many areas of mathematics, and many applications of mathematics in the real world. You could model Madison in this method, where each vertex is a building in the city, and each edge is a road connecting two buildings. You could also use graphs to model relationships between people, where each vertex is a person, and an edge connects two friends, or arch enemies.

A *two-coloring* of a graph associates with each vertex one of two different colors, such that no two adjacent vertices share the same color. Think of the vertices of the graph as seats in a building, with two vertices connected if the seats are within six feet. Then a two-coloring would give two ways to partition seats into those people can sit in while maintaining social distancing. Or maybe the vertices of the graph represent countries, with two vertices connected if the countries border one another. Then a two-coloring represents a coloring that could be applied to give a map of the world (the graph associated with earth is *not* two colorable).

Find an algorithm to determine if a graph is two colorable. What is it's complexity class? Can you find an algorithm to determine if a graph is *three colorable*? What is it's complexity class?

Given two vertices in a graph, can you find an algorithm that finds the *shortest* number of edges required to connect them? What is it's complexity class?

3 Interactive Proofs

As a student, you've often taken exams. you might have felt that *solving a problem* on an exam is much harder than *marking an exam.* Your teacher has it easy. Let's consider the mathematics of why this is true.

In the last section, we considered algorithms which took in problems, and found solutions to 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 of some knowledge*, and we are checking whether this proof is legitimate. 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. In either way, we have to 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.

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 *checks* whether a three coloring is valid. What is it's complexity?

4 Probabilistically Checkable Proofs, and Extensions

We can consider more interesting types of proof verification. One choice, which is quite magical, is a *zero knowledge proof verification system.* In these systems, we want to verify a solution is correct, *without ever being able to reconstruct the solution ourselves.* Thus we must ask questions that do not communicate enough information for us to know the solution, but *do* give enough information to determine if the solution is valid. Often in modern computing, people prove their identity via a password, which acts as a 'hint' which allows one to make a difficult math problem much easier to solve. Zero knowledge proofs force one to formulate difficult math problems which a password can make easier to solve, but which *do not* allow us to reconstruct a person's password. Can you think of a zero knowledge proof that checks whether a graph with m vertices is *perfectly complete*, and *sound* with probability $1/m$, but which does not allow us to reconstruct a 3-colorable graph regardless of how many times we repeat the proof?