Why should I bother?

This chapter gives an introduction to complexity theory.

In the previous sections, we studied problems which didn't require brute force search to find solutions. However, there are problems whose solutions can't be found without examining the entire possible search space. Take the case of prime factorization, for example. Below are the prime factors for 21:

$$7 * 3 = 21$$

It is easy to verify the solution; we just multiply 7 and 3, and check if the product equals 21. However, when we want to do the reverse, i.e. find the prime factors for 21, we need to go through trial division starting from 1, till we finally find the prime factors.

$$? * ? = 21$$

For smaller numbers like 21 a brute force algorithm would finish in reasonable time, but what about really large numbers like the one below?

$$? * ? = 213423567,890182215,329123767,110140235$$

A brute force algorithm would not be able to come up with prime factors for the above number in a reasonable amount of time, especially with today's computer technology. You may wonder why this matters or is even important. Well, enter cryptography! Encryption algorithms make use of the fact that it is easy to multiply two very large prime numbers, but un-assembling the resulting product back into its constituent prime numbers isn't easy. If an easy

solution was found for prime factorization, then a lot of the heavily relied upon encryption algorithms used in the industry would fall apart.

The consequences aren't just limited to cryptography. Thus far, even while using existing algorithm design techniques, humans have been unable to find efficient solutions to many problems. If efficient solutions to these problems were found, it would result in major advancements in fields as far apart as medical science and AI.

Complexity theorists define various classes of complexity depending on how hard a problem is to solve. We'll briefly describe the important classes in the upcoming sections. However, before we embark upon our study of complexity classes, we'll discuss *decision problems* which help us in defining some of the complexity classes.

Decision Problems

A decidable problem is one for which we can answer yes or no. Algorithms aren't decidable problems. For instance, the breadth first search algorithm isn't decidable in any sense. Complexity classes that we'll discuss for the purposes of this course only include decidable problems. Fortunately, we can cast algorithms as decidable problems in order to reason about their complexity. For instance, for depth first search, we can ask: can all the nodes of a graph (with no cycles) be traversed by visiting at most each node once? We can answer this question by running DFS, and keeping a count of visited nodes and answering yes or no at the end.

Optimization Problems

Optimization problems are interesting problems from the perspective of complexity theory. For instance, find the shortest distance between two vertices in a given graph. There may be multiple paths between the two vertices, but the optimum one needs to be the shortest. Also, there could be multiple shortest paths. We can transform the shortest path question into a decision problem by asking if there is a path between two vertices of a graph consisting of less than k edges?

If we can answer the decision version of an optimization problem, we can also answer the optimization problem itself. Consider starting with an arbitrary value of k edges. The decision version of the problem will spit out either a yes or a no, and (depending on the answer) we can tweak the value of k and

recursively apply the binary search to zoom in on the optimum answer.

Essentially, we are wrapping our decision-problem solution with binary search to get to the optimized solution.