1 An Overview

In this course, we develop algorithms to solve a variety of problems and analyse the complexity of these algorithms, usually by bounding their (worst case) running time.

You performed similar analyses of algorithms for sorting, selection, graph searching, and other problems in 308-252. Many of the tools developed in that course will be used in this course. (eg. big O notation, various data structures, and the sorting, selection and graph searching algorithms themselves).

We use the order of growth of the running time as a measure of the speed of an algorithm. We can compare the performance of algorithms using this measure. Thus we say one algorithm is *faster* than another if its time complexity has smaller growth rate.

We would also like to have a way of comparing the difficulty of the problems we consider. A natural measure would be the order of growth of the running time of the fastest algorithm for the problem. Unfortunately, it is very difficult to get a handle on this measure. Indeed, it is hard to prove any lower bound on the running time of the fastest algorithm for a problem, as this necessitates dealing with all of the (possibly infinitely many) algorithms which solve it.

In this course, we introduce a technique for comparing the difficulty of problems directly which does not require us to compute the measure discussed in the last paragraph. This is an important new direction which gives this course a flavour very different from earlier courses. This technique is based on reducing one problem to another. If we have a fast algorithm to reduce problem π to problem π' and a fast algorithm to solve problem π' then we have a fast algorithm to solve problem π. So a fast reduction allows us to use a fast algorithm for one problem to efficiently solve a different problem. You saw examples of such reductions in 308-252, e.g. when we reduced 2-SAT to the problem of computing the strong components of a directed graph. Note however, that if we have such a fast reduction and there is no fast algorithm for problem π then there is no fast algorithm for problem π'. Thus fast reductions also allow us to use a proof that some problem is hard to prove another problem is hard.

Another new topic is verifying that the output of an algorithm is correct. Obviously it is trivial to check whether the output of a program for sorting integers is correct: we just scan the output and make sure each number is less than or equal to its successor. But how could we check the correctness of, say, the output of a program that computes a shortest path between two nodes in a network, or a minimum weight spanning tree of a weighted graph? For more complicated problems we will see that the addition of some extra data, known as a *certificate*, can facilitate the checking process. We will see that the existence, or apparent non-existence, of certificates is intimately related to whether or not a problem can be solved efficiently.

This course also differs markedly from 308-252 in that we consider algorithms for hard problems. However, we want to stay away from those which are too hard. For example, in 308-330 you will see/have seen that there are problems which are undecidable, i.e. for which no algorithm exists at all. Certainly we would not want to work on such a problem. It turns out that the notion of a certificate is also a useful tool for deciding if it is reasonable to attempt to solve a given problem.

These new topics will be fully discussed in the second half of the course. The first half is devoted to the more familiar topic of algorithm design. Specifically developing a variety of techniques for designing algorithms and deciding which technique to use on a given problem. We will however stress two parts of this process which foreshadow the discussion in the second half of the course. These are *problem formulation* and the *verification* that the output of an algorithm is correct.

These two parts of the process are crucial even if one does not design algorithms oneself but rather relies on algorithms developed by others. To illustrate this point, we begin our discussion by considering Linear Programming, instances of which we will solve using a canned software package.