CS 325 Recitation Session 7

Chirag Kaudan

June 2025

1 Introduction

- Survey of how many people have grading questions/complaints
- Are people confused about some conceptual stuff from lecture? **How many people recall the definitions of class P and class NP as discussed in lecture?**
- Goal for today: Motivate the study of complexity theory, open problems

2 Motivating Complexity Theory

Up until this point, we have seen examples of us solving seemingly difficult computational questions with slick, efficient algorithms. Armed with general algorithmic paradigms, it seems like we can conquer almost any problem out there efficiently. Unfortunately (but fortunately for the job safety of some), this does not seem to be the case for many problems.

Complexity theory studies how "hard" different problems are, for some definition of hardness. It investigates why certain problems appear to be "harder" than others. On some level, roughly speaking, "hardness" of a problem is related to how "efficient" the best algorithm to solve that problem is.

3 Defining Efficient Algorithms

What does it mean to solve a problem **efficiently**? It is worth doing a deep dive of what this should mean.

> **Major Question:** How should we make the notion of an "efficient" algorithm into something concrete?

We will attempt to answer this question as naturally as we can, each step of the way proposing a new answer, then discovering some limitations with our answer and refining it.

3.1 A Practical Definition?

Here's perhaps the most natural starting point to answer our big question:

> **Answer 1:** An algorithm is efficient if it can be *implemented* to run quickly on *actual inputs* of the problem it solves.

From a purely pragmatic viewpoint, this is a compelling answer to our major question. After all, at the end of the day, we care about quickly solving real problems and this working definition seems to focus in on just that.

But this definition is missing some things. For example, if all we cared about was how fast an algorithm, when implemented, runs on a computer for some actual inputs, we would be deceived by patently terrible algorithms running quickly on small size inputs. We would also be deceived by terrible algorithms running

on mind-bogglingly fast processors. Should an algorithm that used to be bad 20 years ago be considered good now that it runs quickly on the much faster processors of today?

It's very common to have two different algorithms that both seem equally efficient on inputs of size a couple hundred. But when you multiply the input size by 10, one algorithm is clearly more efficient and the other slows to a crawl...you saw this when you implemented various algorithms for Max Subarray and plotted their running times.

What we want is a notion of efficiency that is independent of the hardware and the instances encountered. We also want our definition of efficiency to stay consistent when we encounter larger and larger input sizes.

The discussion above highlights the following

- We should focus on how the efficiency of an algorithm scales with some natural "size" parameter of the problem
- This is why we use asymptotic analysis all the time when analyzing running times!
- So this suggests are more mathematical approach over a practical, pragmatic viewpoint

3.2 Comparison to Brute Force

In light of the above section, let's say you accept this mathematical practice of bounding the running time of algorithms with functions of the size of inputs and using asymptotic analysis to quantify those bounds. We ask ourselves a relevant mini-question: **How do we know whether a running time for an algorithm is good or bad? Are there any useful benchmarks?**

A common theme with all the problems we have studied so far, and indeed in many practical problems one can study, is the following scenario: given an algorithmic problem, even when the input size is relatively small, the *search space* of the problem is massive. That is, the set of all possible choices an algorithm can return to answer the problem, among which the optimal solution lies, is exceedingly large.

For a concrete example of this theme, consider the Minimum Spanning Tree problem. There are graphs for which the search space (the number of possible spanning trees on that graph) is as large as n^{n-2} (see Cayley's Theorem in graph theory, see also the Matrix Tree Theorem). This is a stupendously large number for even small values of n. The brute force solution to the Minimum Spanning Tree problem would entail searching over every possible spanning tree of a given weighted graph to find the minimum one, which could potentially take time n^n. That is terribly slow.

Brute-force search is bad not only because often times it is way too slow to be practical, but also because it does not give us any insight into the **structure** of the problem we are solving.

This motivates the following definition of efficiency:

Answer 2 An algorithm is efficient if it achieves qualitatively better performance than brute-force search.

This is a useful working definition, since it turns out that algorithms that improve on brute-force search almost always contain genuinely novel structural insights. That is, such algorithms contain some useful ideas that make them work and they dig into the underlying reason behind what makes the problem easy or difficult. Good, we are coming closer to developing a deeper, more accurate definition of "efficiency".

But there are still some problems with this answer. What does "qualitatively better performance" look like? We should actually be looking at the specific running time bounds of algorithms in order to delineate which ones are impressive running time bounds and which ones are bad.

3.3 The Focus on Polynomial Time

Let's say n is the size of your problem. We want a running time of an algorithm to have the desirable scaling property that increasing n by a constant factor only slows down the algorithm by a constant factor. This is in opposition to a scaling property that says, for example, if you increase the size of n by 1, it results in a exponential slow down of the running time.

More formally, if your algorithm's running time is bounded by cn^k primitive computational operations, for constants $c, k > 0$ then it has the desired scaling property. We say an algorithm with such a running time bound runs in *polynomial time* or is a *polynomial time algorithm*.

Notice that if n, the size of your problem, doubles then the running time becomes $c(2n)^k = c2^k n^k$ where 2^k is simply a constant since k is a constant. So this would slow down the algorithm by a constant factor. Of course, we prefer this constant to be small in practice, which is why for example, linear time algorithms are generally more desirable than quadratic time algorithms.

The discussion above leads us to the following answer to our major question.

Answer 3: An algorithm is efficient if it runs in polynomial time.

This definition is mathematically precise, but it begs the question: should an algorithm whose running time is $\Theta(n^{1000})$ be considered efficient? Of course not. Conversely, should an algorithm with running time $O(n^{1+0.0001\log(n)})$, which is not polynomial, be considered inefficient? Surely we would be happy with this running time in practice.

In other words, our objection now is not whether our definition is too vague, as it was for previous definitions, but whether or definition is too strict.

3.4 In Defense of Polynomial Time

The issue with the above objection is that polynomial time as a definition for efficiency actually works well in practice. For almost every algorithm that runs in polynomial time, it happens to be that they are indeed quite tame polynomials. Polynomials of the form n^2 or $n \log n$ or n^3 and not behemoths like n^{1000}. And conversely, problems for which we don't know of any polynomial time algorithm tend to be very difficult to solve in practice (with some notable exceptions, there is an entire subfield of algorithms that focuses on problems that may be hard in theory but not too bad in practice). All this to say, this definition of polynomial time really does seem to work in practice.

The other, **fundamental benefit** of our definition is that we can negate it. It is now completely defined what a "non-efficient" algorithm is; one that has a non-polynomial running time. This allows us to tie the efficiency (or lack thereof) of an algorithm into a formal study of the difficulty of the computational problem it solves. That is, it becomes possible to express the notion that there is no efficient algorithm that can solve a given problem. Contrast this objectivity to our subjective (in varying senses) previous definitions.

The discussion in this section provides a justification for why we are interested in polynomial time as our benchmark for an "efficient" algorithm. In other words, for practical purposes, you could say "polynomial time = efficient and everything else is not".

4 The Big Picture

What does the general stratification of problems look like? Which ones are solvable in polynomial time (and therefore "efficient"), and which ones are not? Which ones require exponential time? Which ones can be solved in finite time?

The diagram looks a bit like Figure 1 below.

Here, the class R represents, informally, the set of problems solvable in finite time. What may surprise you is that there are problems that are not in R. That is, they are not solvable by any algorithm that runs in finite time! In fact, **most** problems are not solvable at all. If you take CS 321, you will talk much more about these types of problems, as well as build up the necessary theory to discuss these topics deeper than we are here.

(As a side note, for those of you that have taken CS 321, you may recognize the class R as being the class of decideable problems/formal languages. They are the same! The class is called R for "recursive" problems, which is another term used for decideable problems).

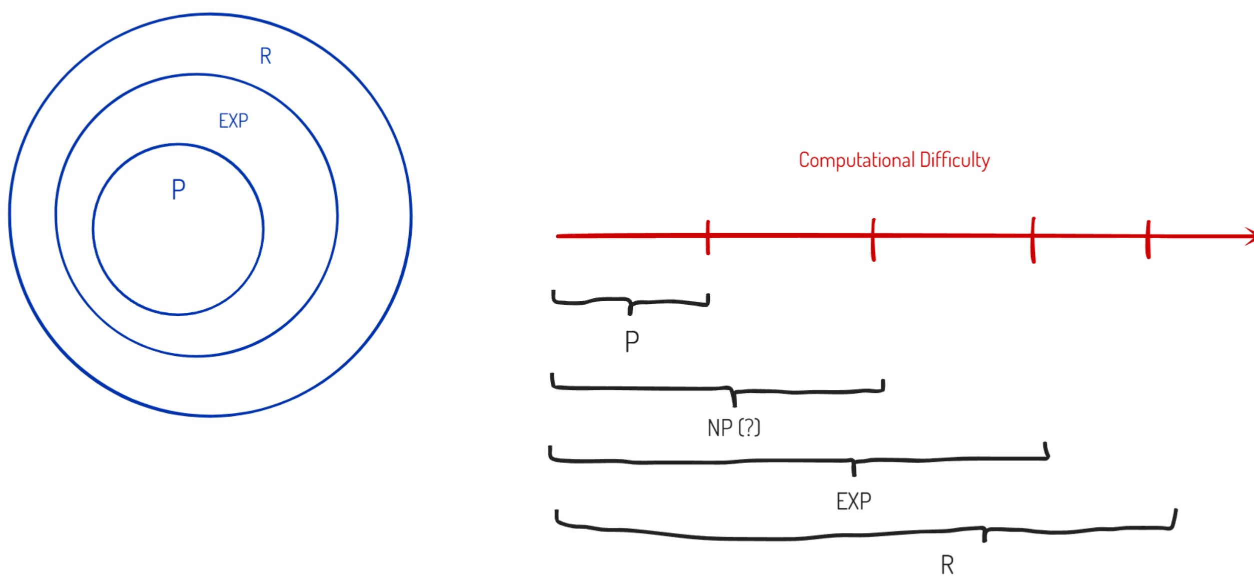

Figure 1: Rough layout of complexity classes arranged both in Venn Diagrams and in a line by their "computational difficulty". Whether the first region on the line is equal to the second region on the line is a big open problem

4.1 Theory of Computation Advertisement: Take CS 321!

The rest of the notes will be brief commentary about the general nature of what somebody would study in the theory of computation, why it is important, why it is relevant, and give you some brief exposure to it.

In theory of computation, one studies mathematical models of computation that help us rigorously define what it means to "compute" something, or what an "algorithm" means. This solid mathematical foundation allows us to explore the limits of our models. One would typically start off with weak models of computation, abstract machines, then successively give them additional powerups to see how the amount of stuff they can "compute" changes. This allows us to reason about computation done more effectively than is realistic, and in this sense, we can figure out which problems are "beyond computation".

I think people that are generally working in the career path that we are in should find it interesting to investigate the limits of computers. But even though it's a mathematical study at its core, it also has great applications throughout computer science. For example, studying a model of computation known as a *grammar* is extremely useful when you want to design a specialized programming language for your own uses. Any time you are dealing with string search and pattern matching algorithms/problems, models of computation known as finite automata and regular expressions will aid you greatly. Essentially all of modern cryptography relies on the mathematical principles you learn to develop in a study of the theory of computation. And any time you have a problem that seems to be taking too long even with your best algorithms, it's time to remember the theory of NP-completeness.

The one true way of knowing whether theory of computation (and by extension, theoretical computer science) interests you is by trying it and taking CS 321 as soon as possible!

5 Conclusion

We discussed the definitions of P and NP, justified why polynomial time can be used as a definition of an efficient algorithm, and gave a brief overview of the theory of computation.