CS 341: ALGORITHMS (S18) — LECTURE 18
POLYNOMIAL-TIME ALGORITHMS

ERIC BLAIS

1. Efficient algorithms and tractability

What is an *efficient* algorithm? That depends on your situation. In various settings, a requirement for an algorithm to be efficient might be that the algorithm

- Has time complexity $O(n)$;
- Runs in $O(n)$ time *and* only needs to scan the input once (streaming algorithm);
- Can be run in $o(n)$ time using distributed algorithms (e.g., MapReduce);
- Runs in $o(n)$ time by sampling the input;
- Has time complexity $O(n \log n)$;
- Has time complexity $O(n^2)$;
- Has time complexity $O(n^3)$ *and* space complexity $O(\log n)$;
- etc., etc., etc.

For this class, we won't explore any of those definitions. Instead, we will take a much more lenient notion of efficiency: we'll say that an algorithm is (reasonably) efficient if it is a *polynomial-time algorithm*.

Definition 18.1. An algorithm A is a *polynomial-time algorithm* if there exists a constant k such that on any input of size n, the algorithm A runs in time $O(n^k)$.

Remark. For this definition, we measure the size of the input in terms of the number of bits required to encode the input (as opposed to, say, the number of entries in a matrix).

And then we get to the main question that we will study in this part of the course:

Which computational problems can be solved by polynomial-time algorithms?

We will call such problems *tractable*. Most of the problems we have covered in this class—and indeed most of the problems you see in all CS courses combined!—are tractable. The only exceptions in this class are the problems of the last lecture and the problems that can be solved with *pseudo-polynomial-time algorithms* we saw in the dynamic programming section, but it turns out that there are a *ton* of seemingly reasonable problems that, to the best of our knowledge today, are intractable. In this section, our task is to learn how to recognize these problems.

1.1. Why polynomial time?

But before we continue, there's one rather puzzling question that we should examine right away: *why* polynomial-time algorithms? After all, we listed many other perfectly reasonable possible definitions of efficient algorithms, and you may even argue that one (or any!) of them would be much better at capturing the idea of efficiency or tractability, so why didn't we pick one of them instead? I can think of three reasons.

(1) **Robustness.** We believe that the class of problems that can be solved by polynomial-time algorithms is *independent* of the way we define algorithms. This belief is known as the *strong Church-Turing thesis*, and this is something that would *not* be true if we defined tractability with any of the other candidates we listed earlier. This means that if you show that a problem is intractable and are faced with the criticism that your algorithm model is not the right one, you can reply that it doesn't matter! Any reasonable modification to the model won't affect the tractability or intractability of the problem. (See CS 360/365 for all the details.)
(2) **Strength of the conclusion.** The main point of this section is that we will want to be able to argue that some problems are *not* tractable. It turns out that, in this case, it is good to have a very lenient notion of tractability, because by showing that a problem has no polynomial-time algorithm, we will also imply at the same time that the problem can't be solved by an efficient algorithm under any of the other alternative definitions of efficiency above either! (And many others as well.) So instead of ruling out algorithms in one setting after another in a long and tedious process, we do so all at once.
(3) **Applicability.** All of the discussion above is very nice, but wouldn't be of interest if most problems could be solved by polynomial-time algorithms. But as we're about to see, that's far from the case. In fact, I would be extremely surprised if most of you don't end up running up against intractable problems multiple times throughout your careers. So knowing how to show that a problem is intractable is not just something that's of purely intellectual interest; it is something that will be useful in practice.

2. The class **P**

We saw in the last lecture two types of computational problems:

Decision problems: : problems where the possible outputs are `True` or `False` (or, equivalently, 1 or 0; `Yes` or `No`; etc.); and
Optimization problems: : problems where we are trying to find the value of the *best* solution.

There is also a third large class of problems that we can call

Search problems: : problems where the goal is to *find* a valid (or the best) solution itself.

We will now focus exclusively on *decision problems*. This makes everything we do from now on much easier, and it's still useful even if the problem you care about is an optimization or a search problem because we can often obtain analogous decision problems very easily. For example, we can define:

Mult: : Given two n-bit positive integers x and y and a coordinate $i \in \{1, 2, \ldots, 2n\}$, determine if the ith bit of the product xy is 1.
IntSched: : Given a set S of n intervals $(s_1, f_1), \ldots, (s_n, f_n)$ and a positive integer k, determine if there is a set of k non-overlapping intervals in S.
LCS: : Given two strings $x_1, \ldots, x_m$ and $y_1, \ldots, y_n$ and a positive integer k, determine if the longest common subsequence of x and y has length at least k.

The class of tractable decision problems is of such fundamental importance that we give it a name.

Definition 18.2. The class **P** is the set of all decision problems that can be solved by polynomial-time algorithms.

We have already seen that the problems 3Sum, IntSched, LCS, Connectivity,... are all tractable by designing polynomial-time algorithm for them. In mathematical notation, we write this as

$$\begin{aligned} \text{3Sum} &\in \mathbf{P} \\ \text{IntSched} &\in \mathbf{P} \\ \text{LCS} &\in \mathbf{P} \\ \text{Connectivity} &\in \mathbf{P}. \end{aligned}$$

3. Reductions

How can we show that a new problem is in **P**? We can design a polynomial-time algorithm that solves the problem. Or we can use the idea of *reduction*.

Definition 18.3. The decision problem A is *polynomial-time reducible* to the decision problem B, written

$$A \leq_{\mathbf{P}} B,$$

if there is a polynomial-time algorithm F that, on any input I_A to the problem A, produces an input I_B to the problem B that has the same answer.

In other words, A is polynomial-time reducible to B if there is a polynomial-transformation of the inputs of A to those of B that maps all `Yes` inputs of A to the `Yes` inputs of B and all the `No` inputs of A to the `No` inputs of B. You can see a picture of this view of polynomial-time reductions in the class textbook.

You have already seen one example of a polynomial-time reduction in a tutorial earlier this term.

Definition 18.4. In the LIS (*longest increasing subsequence*), you are given a sequence $x_1, \ldots, x_n$ of n positive integers in the range $\{1, 2, \ldots, \ell\}$ and a positive integer k, and you must determine if the longest increasing subsequence of x has length at least k.

Theorem 18.5. LIS $\leq_{\mathbf{P}}$ LCS.

Proof. Given an input to the LIS problem, let F be the algorithm that sorts x to obtain the sequence y and returns the sequences x and y and the integer k. The algorithm F runs in polynomial-time and its result corresponds to an input of the LCS problem which has a longest common subsequence of length at least k if and only if the original sequence x had a longest increasing subsequence of length at least k. □

Now here is an observation that will prove to be extremely powerful: we can show polynomial-time reductions between A and B *even when we don't know of any polynomial-time algorithm for either problem*! For example, take the following two problems that we have not yet considered.

- Clique: Given a graph $G = (V, E)$ and a positive integer k, determine if G contains a clique of size k;
- IndepSet: Given a graph $G = (V, E)$ and a positive integer k, determine if G contains an independent set of size k.

We have not yet seen any polynomial-time algorithms for either of these problems, but we can still show polynomial-time reductions between the two problems.

Theorem 18.6. CLIQUE $\leq_{\mathbf{P}}$ INDEPSET *and* CLIQUE $\leq_{\mathbf{P}}$ INDEPSET.

Proof. To show that CLIQUE $\leq_{\mathbf{P}}$ INDEPSET, let F be the algorithm that takes in a graph $G = (V, E)$ and generates the complement graph $\overline{G} = (V, \overline{E})$ with $\overline{E} = \{(u, v) \notin E\}$. The construction takes polynomial time, and the resulting graph $\overline{G}$ has an independent set of size k if and only if the original graph has a clique of size k.

To show that INDEPSET $\leq_{\mathbf{P}}$ CLIQUE, we also use the same algorithm F, and we observe that every independent set of G becomes a clique in $\overline{G}$. □

What does this mean? The definition of polynomial-time reductions and the result CLIQUE $\leq_{\mathbf{P}}$ INDEPSET means that if someone designs a polynomial-time algorithm for INDEPSET, then there is also a polynomial-time algorithm for CLIQUE. This also means that if someone shows that there is *no* polynomial-time algorithm for CLIQUE, then there can be no polynomial-time algorithm that solves INDEPSET either. More generally, our interest in polynomial-time reductions stems from the following main lemma.

Lemma 18.7. *If A and B are two decision problems that satisfy $A \leq_{\mathbf{P}} B$, then*

- *If $B \in \mathbf{P}$ then also $A \in \mathbf{P}$; and*
- *If $A \notin \mathbf{P}$ then also $B \notin \mathbf{P}$.*

For most of the hard problems that we care about, we do not know how to show that they are not in **P**. Starting in the next lecture, we will use the theory of **NP**-completeness to do the next-best thing: show that if *one* of those hard problems is tractable, then they *all* are.