CS 341: ALGORITHMS (S18) — LECTURE 23
BONUS LECTURE I: ON HARD PROBLEMS

ERIC BLAIS

1. Impossible problems

We spent quite a few lectures in the **NP**-completeness part of the course showing how there are many natural problems that we do not know how to solve in polynomial-time (and that many people believe we simply can't solve in polynomial time). But all of the problems that we saw can easily be solved if we allow our algorithms to run in time that is *exponential* in the size of their inputs.

There are problems that are even harder than that. In fact, there are problems that cannot be solved by any algorithm whatsoever—even if we were to let the algorithm run for some ridiculous amount of time (like $2^{2^{2^{2^n}}}$ steps on inputs of length n, or any other function you can come up with). Such problems are called *undecidable*. And it would be reasonable to expect such problems to look rather formidable, and probably rearlly unnatural and contrived. That's not the case. In fact, there is an undecidable problem already hiding in the definition of algorithms itself.

Back in lecture 2, we defined an algorithm as a description of a process that is:

- effective (= built of sequence of basic steps),
- unambiguous (= does not require interpretation), and
- finite (= requires only finite number of steps to be written down); and that
- takes in some input and
- halts and generates some output after a finite number of steps.

With this description, it seems pretty easy to recognize algorithms. You should be able to verify, for example, that all the pseudocode that we discussed in this class satisfy this definition of algorithms. However, we probably all encountered a situation where we programmed what we believed to be an algorithm, only to find that because of a bug our code enters some infinite loop and never terminates. According to the definition above, therefore, the code that we implemented when this happens does not implement an actual algorithm. And even if you don't happen to care about the definition of algorithms itself, infinite loops are usually something we never want in any code that we use in the real world, so this situation begs the question: can't we design a piece of software to *check* whether we have any infinite loops in an algorithm? This problem is known as the *halting problem*.

Definition 23.1. In the Halting problem, we are given as input the code (in binary) for a (possibly non-halting) algorithm; we must determine whether the code halts after a finite number of steps on every input.

As a first step, we can even consider an easier variant of the Halting problem where we only consider inputs that represent code for functions that don't even take in any input, and we must determine if the code terminates after a finite amount of time. Even with this

simplification, the situation is grim: there is no algorithm that can solve either this variant or the original version of the HALTING problem.

Theorem 23.2 (Turing 1936)**.** *There is no algorithm that can solve the* HALTING *problem.*

This theorem is obtained with a beautiful application of the *diagonalization* proof technique. But if you're like me, even knowing that this theorem is true, it still doesn't seem possible. After all, how hard can it really be to determine if natural algorithms (as opposed to rather contrived examples that you might cook up just to prove this result) halt or not? Again, however, we really don't need to look far before we see just how hard the problem really is. Take the following simple procedure that takes in a positive integer x as input.

Algorithm 1: COLLATZ(x)

```
while x > 1 do
    if x is even then
        x ← x/2;
    else
        x ← 3x + 1;
```

Does this algorithm halt after a finite number of steps on every input x? We don't know! Lothar Collatz introduced the problem in 1937 and conjectured that the procedure does always halt, but nobody has been able to make any significant progress on the question apart from trying it experimentally on (many, many) specific integers. My favourite quote about the problem is from Paul Erdős, who stated that, quite simply, "mathematics may not be ready for such problems."

Or maybe you have heard of Goldbach's conjecture from 1742: can every even number be represented as the sum of two primes? That's a conjecture that we could resolve if we could determine whether the following procedure halts.

Algorithm 2: GOLDBACH()

```
n ← 4;
counter-example ← False;
while !counter-example do
    n ← n + 2;
    counter-example ← True;
    for x = 3, 5, 7, ..., n − 1 do
        if IsPrime(x) and IsPrime(n − x) then
            counter-example ← False;
return "n is a counter-example to the Goldbach conjecture!"
```

(There is a polynomial-time algorithm that solves ISPRIME, though for the task of settling Goldbach's conjecture it would suffice to implement the function with a simple naïve test that checks whether any number in the range $2, 3, \ldots, x-1$ divides x.)

2. Rather hard problems

Let's now turn back our attention to the more "reasonable" problems in **NP**. Let's say that we're considering some problem like CLIQUE. We have seen that we can't hope to

show that there is no polynomial-time algorithm that can solve CLIQUE before we resolve the infamous **P** vs. **NP** problem. But what if we *really* want an unconditional lower bound on the amount of time required to solve the problem? This shouldn't be too unreasonable to ask for—if we don't expect even $O(n^{1000})$-time algorithms to be able to solve CLIQUE, perhaps we could at least hope for a lower bound of, say $\Omega(n^4)$, to say that it's at least "not very easy"!

Alas, even that is too much to ask for. The best lower bound we have for CLIQUE is the following.

Theorem 23.3. *Any algorithm that solves* CLIQUE *must have time complexity* $\Omega(n + m)$ *on input graphs with* n *vertices and* m *edges.*

Proof idea. Any deterministic algorithm that solves CLIQUE must at least read all of its input before producing the answer! □

Amazingly, this lower bound is essentially the only unconditional lower bound we have on the time complexity of *any* explicit problem in **NP**, as long as we consider all possible algorithms—the only stronger lower bounds that we have apply to specific models of computation. For example, the $\Omega(n \log n)$ lower bound that you saw in previous classes for sorting applies only to "comparison-based" algorithms, and indeed there are other types of algorithms like Radix Sort that run in linear time.

3. Slightly hard problems

The fact that we don't have any non-trivial unconditional lower bounds for CLIQUE is mitigated by the fact that we can at least show that the problem is **NP**-complete and thereby at least obtain some explanation/justification for the fact that we can't come up with polynomial-time algorithm for the problem. But what if we are considering a problem that *is* in **P**? If you're looking at a well-studied problem, like 3SUM, then the fact that many researchers have looked at the problem before you without finding better algorithms than the one you have is good information. In fact, for this problem, it has even been conjectured that (essentially) the best possible algorithm has already been found.

Definition 23.4. In the 3SUM problem, we are given a set $S \subseteq \mathbb{Z}$ of integers and we must determine whether there are three integers $a, b, c \in S$ such that $a + b + c = 0$.

Conjecture 1 (3SUM conjecture)**.** *For every* $\epsilon > 0$*, any algorithm that solves* 3SUM *has time complexity* $\Omega(n^{2-\epsilon})$*.*

But what if you are considering a new problem that, as far as you know, nobody has studied before? Take the following problem, for instance.

Definition 23.5. In the COLINEAR problem, we are given a set of n points in the plane, and we must determine if there is a line in the plane that passes through at least 3 of the points.

How efficiently can you solve this problem? There is a simple $O(n^3)$-time algorithm: enumerate all sets of 3 points and for each such set determine if they are in a line or not. Can you do better? With some work, we can obtain an $O(n^2 \log n)$ time algorithm and you may even find an algorithm with time complexity $O(n^2)$. But then no matter how hard you try, you can't do any better. Can we explain why we get stuck at this point? Indeed we can, using the idea of reductions!

Theorem 23.6. *If the* 3SUM *conjecture is true, then there is no algorithm that solves* COLINEAR *and has time complexity* $O(n^c)$ *for any constant* $c < 2$.

Proof. The idea is that we can construct a reduction from 3SUM to COLINEAR that takes only $O(n)$ time—this means that if we obtain an algorithm for COLINEAR that runs in time $O(n^c)$ for some $1 \leq c < 2$, we can apply the reduction and invoke this algorithm to solve 3SUM in time $O(n + n^c) = O(n^c)$.

What is the reduction? For each integer $a \in S$ in the 3SUM problem, add the point (a, a^3) to the instance of COLINEAR. Then three points (a, a^3), (b, b^3), and (c, c^3) are colinear if and only if

$$\begin{aligned} \frac{b^3 - a^3}{b - a} &= \frac{c^3 - b^3}{c - b} \\ \Leftrightarrow a^2 + b^2 + ab &= b^2 + c^2 + bc \\ \Leftrightarrow a^2 + ab + ac &= c^2 + bc + ac \\ \Leftrightarrow (a - c)(a + b + c) &= 0 \end{aligned}$$

But this can only hold when $a + b + c = 0$ since all the integers in S are disjoint and so $a - c \neq 0$. □