1. Impossible problems

We spent quite a few lectures in the \textbf{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 \textit{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 $2^{2^{2^n}}$ steps on inputs of length $n$, or any other function you can come up with). Such problems are called \textit{undecidable}. You have already seen such problems in earlier classes. One classic undecidable problem is the \textit{halting problem}.

\textbf{Definition 23.1} (Halting problem). Given as input the binary code for an algorithm, determine whether the code halts after a finite number of steps on every input.

\textbf{Theorem 23.2} (Turing 1936). \textit{There is no algorithm that can solve the halting problem.}

This theorem is obtained with a beautiful application of the \textit{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.

\begin{algorithm}
\textbf{Algorithm 1: Collatz($x$)}
\begin{algorithmic}
\State \textbf{while} $x > 1$ \textbf{do}
\hspace{1em} \textbf{if} $x$ is even \textbf{then}
\hspace{2em} $x \leftarrow x/2$;
\hspace{1em} \textbf{else}
\hspace{2em} $x \leftarrow 3x + 1$;
\end{algorithmic}
\end{algorithm}

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.”
We can even consider an easier variant of the Halting problem that only takes in algorithms which don’t even have any inputs (or the variant where we only care to know if the algorithm halts on a given input). Even with this simplification, the situation is grim: there is no algorithm that can solve this variant of the Halting problem either. And once again there are natural examples of algorithms for which we don’t even know whether they halt or not. For instance, you may 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 algorithm halts.

\[
\text{Algorithm 2: Goldbach}
\]
\[
\begin{align*}
&n \leftarrow 4; \\
&\text{counter-example } \leftarrow \text{False}; \\
&\textbf{while} \ \neg \text{counter-example} \ \textbf{do} \\
&\quad n \leftarrow n + 2; \\
&\quad \text{counter-example } \leftarrow \text{True}; \\
&\quad \textbf{for} \ x = 3, 5, 7, \ldots, n - 1 \ \textbf{do} \\
&\quad\quad \textbf{if} \ \text{IsPrime}(x) \ \text{and} \ \text{IsPrime}(n - x) \ \textbf{then} \\
&\quad\quad\quad \text{counter-example } \leftarrow \text{False}; \\
&\quad \textbf{return} \ "n \text{ is a counter-example to the Goldbach conjecture!}" \\
\end{align*}
\]

(There is a polynomial-time algorithm that solves \text{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.)

If you are currently considering a problem that you suspect is impossible to solve with an algorithm, you can prove that this is the case using the notion of reduction we have worked on extensively over the last few weeks. In this setting, the only difference is that your reduction does not need to run in polynomial time.

**Theorem 23.3.** If \(X\) is a decision problem for which the reduction Halting \(\leq X\) holds, then there is no algorithm that can solve \(X\).

2. **Very difficult problems**

Even when we restrict our attention to problems that can be solved with algorithms, we can still identify some very difficult problems—problems that are even more difficult than the \(NP\)-complete problems we have seen already.

One particularly notable difficult problem is \(TQBF\), which stands for “Totally Quantified Boolean Formula”.

**Definition 23.4** (TQBF problem). Given a totally quantified Boolean formula
\[
\exists x_1 \forall x_2 \exists x_3 \forall x_4 \cdots \phi(x_1, \ldots, x_n),
\]
determine if the formula is true or not.

This problem captures a wide variety of natural 2-player games, as we can see by noticing that trying to figure out a winning strategy for such a game consists of trying to answer the question “Is there a first move \(M_1\) that I can make such that no matter what move \(M_2\) my opponent makes, there exists a response \(M_3\) where for any possible next move \(M_4\) of the opponent I have a further move \(M_5\)…for which I end up in a winning position?”.
Note that unlike SAT, TQBF appears to be both hard to solve and hard to verify—after all, what certificate could you devise that would convince a polynomial-time verifier that a given formula is true? Whether or not TQBF can be solved in polynomial-time, however, remains open: it is in fact equivalent to the problem of determining whether all problems that can be solved by algorithms that use a polynomial amount of memory can also be solved by algorithms that use a polynomial amount of time. (In complexity theory terms: does \( P = \text{PSPACE} \)?)

There are also problems that are known to require more than polynomial time. One such problem is another variant of the Halting problem.

**Definition 23.5** (\( \text{HaltIn}^k \text{Steps} \) problem). Given an algorithm \( A \), an input \( x \) to \( A \), and a positive integer \( k \), does \( A \) halt on input \( x \) after at most \( k \) steps?

When we fix \( A \) and \( x \) and then consider \( k \to \infty \), then the input to the \( \text{HaltIn}^k \text{Steps} \) problem has size \( O(\log k) \). And this problem can be solved in time (roughly) \( O(k) \) by simulating the algorithm \( A \) for \( k \) steps. A modification of the proof of Turing’s theorem shows that this is also essentially best possible: all algorithms that solve the \( \text{HaltIn}^k \text{Steps} \) problem have time complexity \( \Omega(k) \), which is exponential in the input size.

Finally, for a problem that can’t even be solved in exponential time, consider the task of determining if two regular expressions are equivalent to each other: if we let these regular expressions have negations as well as the usual union and star operators, then this problem can be solved but has time complexity that is not bounded by \( O(2^{2^{2^{\cdots^{2^k}}}}) \) when this tower has any constant type.

These problems show that there is a wide variety of different difficulties for computational problems. And, once again, if you suspect that the problem \( X \) you are considering is at least as hard as any of them, a polynomial-time reduction from one of these problems to \( X \) is all you need to prove that fact.

### 3. Rather difficult problems

Let’s now turn back our attention to the more “reasonable” problems in \( \text{NP} \). 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. \( \text{NP} \) 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 23.6.** 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 \( \text{NP} \), as long as we consider all possible algorithms—the only stronger lower bounds that we have apply to specific models of computation.
4. SLIGHTLY DIFFICULT 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 23.7.** 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 23.8.** 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 23.9.** 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

\[
\frac{b^3 - a^3}{b - a} = \frac{c^3 - b^3}{c - b} \quad \iff \quad a^2 + b^2 + ab = b^2 + c^2 + bc
\]

\[
\iff \quad a^2 + ab + ac = c^2 + bc + ac
\]

\[
\iff \quad (a - c)(a + b + c) = 0
\]

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