Time Complexity (1965-) [Sip7]

Oded Regev

April 18, 2009

A complexity class is a collection of functions that can be solved with some given resources. We will typically be interested in Boolean functions, i.e., languages. There are already 462 known complexity classes! Check out the Complexity Zoo

Definition 0.1 *For a function* $T : \mathbb{N} \to \mathbb{N}$ *define* $\mathrm{DTIME}(T(n))$ *as the set of Boolean functions that can be computed in time* $O(T(n))$ *(i.e., there exists a* $c > 0$ *such that they can be computed in time* $c \cdot T(n)$*).*

Note that this definition depends on the details of the machine. A program in C can do more than a Turing machine in the same time.

Definition 0.2 $\mathrm{P} = \bigcup_{c \geq 1} \mathrm{DTIME}(n^c)$

We usually think of P as the class of problems having an efficient solution.

Remarks about the definition of P:

- Unlike DTIME, the definition of P does not depend on the details of the machine. Whether we define P using Turing machines, C programs, Java programs, the resulting complexity class is exactly the same. We call this *robustness.*
- Not everything in P is really efficient. A running time of n^{100} is completely impractical. However, almost all problems in P have reasonably efficient algorithms. Usually, problems shown to be in P with a bad running time, were later shown to be solvable with a truly efficient running time.
- Does P really represent what we can do in polynomial time?
  - We can allow our algorithms to use *randomness* by using, e.g., coin tosses, radioactive decay, diode noise, etc. This ability is not captured by P and leads to the class BPP. There are a few problems in BPP that are not known to be in P, and some people believe that P = BPP.
  - We can allow our algorithms to use *quantum mechanics.* This leads to the class BQP. There are several problems in BQP that are not known to be in BPP, most important of which is the factoring problem. See Shor's algorithm.
- In all the above classes and in most of the course we will insist on the algorithm solving *every instance* of the problem. One can also consider algorithm that solve *most instances* of the problem. This leads to an area calls *average-case complexity* (as opposed to the traditional *worst-case complexity* which we will work with)
- One can also consider algorithm that solve a problem *approximately.* This leads to the area of *approximation algorithms* which we'll discuss later in the course.

1 The Class NP (1971) [AB2]

Does this Sudoku have a solution?

	2			3		9		7
	1							
4		7				2		8
		5	2				9	
			1	8		7		
	4				3			
				6			7	1
	7							
9		3		2		6		5

Figuring out if a given Sudoku has a solution seems like a hard question to solve (actually, it's NP-complete when extended to $n \times n$ Sudokus). But, I can easily convince you that the answer is 'yes' if I show you the solution. Finding proofs of mathematical theorems is hard (also NP-complete when properly defined), but verifying proofs is easy process: each line of the proof should follow from previous lines using one of a small set of rules. There are people working on writing proofs in a way that a computer can verify; see here. This leads to the definition of NP – these are yes/no questions for which one can efficiently get convinced that the answer is yes. The P versus NP question asks whether problems that can be verified efficiently can also be solved efficiently. This is the most central question in theoretical computer science, and mathematics in general. It was first mentioned in a 1956 letter from Gödel to von Neumann:

> If SAT has a quadratic time algorithm then this would have consequences of the greatest magnitude ... it would indicate that the mental effort of the mathematician ... would be completely replaced by machines ... this seems to me, however, within the realm of possibility.

Von Neumann died the following year, and the letter was only found in the 1980s.

We will see two equivalent definitions of NP. The first one uses the notion of a *witness*. The notion of a witness is very influential, and people in computer science look at many types of interactive proofs.

Definition 1.1 *We say that a language $L \subseteq \{0,1\}^*$ is in* NP *if and only if there exists a polynomial $p : \mathbb{N} \to \mathbb{N}$ and an algorithm M (often called* verifier*) such that for all $x \in \{0,1\}^*$,*

$$x \in L \Leftrightarrow \exists w \in \{0,1\}^{p(|x|)} \text{ s.t. } M(x,w) = 1.$$

We call such a w a witness of x.

Think of w as a 'defence witness' trying to convince the judge (M) that x is in the language. The judge listens to w and should decide if to believe her or not. If $x \in L$, we require that there is a way for the witness to convince the judge to accept. If $x \notin L$ then we require that no matter what the witness tells the judge, the judge will reject.

Question: what would happen if we change $M(x,w) = 1$ to $M(x,w) = 0$ in the definition? Answer: nothing. The definition still defines the class NP. That's because the existence of $x \in L$ is still determined by the *existence of a witness satisfying some polynomial time condition*. The fact that we switched the output of the judge doesn't make any difference (think of it as a Bulgarian judge).

Question: why do we insist on the witness being of polynomial size? can't we simply remove this restriction? it seems that since the verifier is running in polynomial time, it anyway cannot read witnesses longer than polynomial. Answer: we cannot remove this restriction. The reason is that the running time of the verifier (like all other algorithms) is measured in terms of its input. When we say that the verifier runs in polynomial time, we mean that it's polynomial in $|x|+|w|$. So if we allowed w to be very big, the verifier could also run for a very long time. We *could*, however, remove the restriction on the size of w and instead require the verifier to run in time polynomial in $|x|$.

2 Examples of Some Interesting Problems

- **Independent Set:** How would you define the maximum independent set problem as a decision problem? what would be a reasonable definition?

  Given an (undirected) graph G and a number k, does G contain an independent set of size k?

  This definition seems 'reasonable' because being able to solve this decision problem implies a solution to the question of *finding* the maximum independent set. Why? Proof idea: First figure out the size of the maximum independent set by solving the decision problem for $k = 1, 2, \ldots, n$ and looking for the largest k that still returns yes. Once this is done, look for vertices that can be removed without reducing the size of the maximum independent set. After removing those vertices, we end up with an independent set of size k. What we did here is known as a Cook reduction, which is a more powerful notion of reduction than the notion of Karp reduction, which is the one we will usually work with. This is not the only possible definition (see below).

  This problem is in NP. Why? We can construct a verifier algorithm that expected to receive a list of vertices $v_1, \ldots, v_k$ that form an independent set. The verifier algorithm needs to check that: (1) there are k distinct vertices (2) there are no edges between the vertices. Notice that the witness is of polynomial size and also the verifier runs in polynomial time (where "polynomial" always means "polynomial in the input size"). Notice that there might be other valid verifiers.

  Is the problem also in P? You might be tempted to say 'no' but in complexity we usually cannot say 'no'. The problem is known to be NP-complete, which means that it is *highly unlikely* to be in P (since otherwise P = NP), but we don't know how to rule out this possibility.

- **Maximum Independent Set:** given a graph G and a number k, is the size of the maximum independent set *exactly* k?

  Notice that, as before, if we can solve this problem, we can also find the maximum independent set. So this decision problem also seems reasonable.

  Is it in NP? This is no longer so clear! A witness can show us an independent set of size k, but how can a witness convince us that *there is no* independent set of size larger than k? There are $\binom{n}{k+1}$ possibilities for an independent set of size $k+1$ and this can be exponential in n so we cannot try all options. Checking that the given set of size k cannot be enlarged shows that the set is *maximal*, but not necessarily *maximum*! In fact, we do not believe that this problem in NP. It is known to be in a class called Σ_2^p, forming part of the so-called polynomial-time hierarchy.

- **Hamiltonian path:** given a graph G, does it contain a path that visits each node exactly once?

  Can you see a way to *find* the Hamiltonian path given an algorithm that solves this decision problem?

This problem is also in NP: Here we can construct a verifier that expects to get the list of vertices in the path. The verifier checks that each vertex appears exactly once in the list, and that there is an edge from vertex i to vertex $i+1$ in the list. It is easy to check that the witness is of polynomial size and the verifier runs in polynomial time.

This problem is also known to be NP-complete, as well as the closely related problem of Hamiltonian cycle.

- **Linear Programming:** Is there a solution to a given a set of linear inequalities? For instance, is there a solution to the following list of inequalities:

  $$\begin{aligned} 2x - 4y + 9z &\geq 4 \\ 6x - 2y - z &\leq 2 \\ x + 3y + 3z &\geq 8 \\ 3x - 7y + 2z &\leq 2 \end{aligned}$$

  What's the size of the input? Let's assume all coefficients are integer (this is without loss of generality). If we have m equations and n variables, and the largest coefficient is at most B in absolute value, then the size of the input is $O(nm \log B)$.

  Solving such linear programming is extremely important in practice, used in everything from economics and engineering to biology.

  Is this problem in NP? Yes, but it's not absolutely obvious! We construct a verifier that expects to get as a witness the satisfying assignment. Then all the verifier has to do is simply check that all inequalities are satisfied. What is missing in this argument? We need to show that the witness size is polynomial in the input size and also that the running time of the verifier is polynomial. The verifier indeed runs in polynomial time; no problems here. The delicate issue here is with the size of the witness: we need to show that there is a satisfying assignment whose representation size is polynomial in the input size. It *might* be that any satisfying solution involves huge numbers that cannot be represented in polynomial size. Luckily, the story has a happy ending: it can be shown that there is always a solution whose representation is polynomial in the size of the input. It then follows that the problem is in NP.

  Is this problem also in P? A very important and famous algorithm from 1947, called the Simplex algorithm, works very well in practice. However, somewhat surprisingly, this algorithm is *not* a polynomial time algorithm. There are instances for which the running time of the algorithm is exponential. These instances are rare but they still exist. In 1979 Khachiyan developed the Ellipsoid algorithm, a very important polynomial time algorithm for solving linear programs (and more generally, convex optimization problems). This shows that the problem is in P.

- **Integer programming:** given a set of inequalities as before, the goal now is to decide if there is a solution *using integer numbers.*

  This problem is also known to be in NP. The proof is similar to the earlier one, and again the main part is to prove that there is a satisfying solution whose numbers are sufficiently small so that it can be represented in polynomial space.

  Somewhat surprisingly, this problem is NP-complete and therefore not believed to be in P.

- **Compositeness:** given a number N, is it composite (i.e., not a prime)?

  This problem is very important in cryptography since almost all cryptographic systems rely on prime numbers. For instance, in order to set up an HTTPS connection, the browser has to choose some prime numbers. One way to do this is to choose a random number and check if it's prime or not (this works because there are lots of prime numbers: the number of prime number between 1 and N is roughly $N/\log N$ so a random number will be prime with probability $1/\log N$ which means we only have to try $O(\log N)$ times before we find a prime number with good probability).

  At first, it might seem this problem is trivially in P: just consider the algorithm that tries for all $k = 2, 3, \ldots, N-1$ to see if k divides N or not. However, this already is *not* a polynomial time algorithm. The running time of the algorithm is around $O(N)$ but the input is of size $O(\log N)$!

  So, is this problem in NP? Yes. The verifier expects to get as a witness a nontrivial divisor K. The verifier checks that $2 \le K < N$ and that K divides N. The witness is of size at most $\log N$, and the verifier runs in time polynomial in $\log N$.

  Before we describe what else is known about this problem, let's talk a bit about the complement problem.

- **Primality:** given a number N, is it prime?

  This is the complement of the above problem, and therefore it is in coNP, i.e., a witness can convince the judge that an input is *not* in the language (simply by giving a nontrivial divisor).

  Is the problem also in NP? In other words, how do you convince someone that a certain number is prime? In 1867, a mathematician named Landry needed to prove that the number 1133836730401 is prime. But after supposedly proving it, he was unable to convince other people of this fact. Here's a quote from that paper:

  > At this point we are, if not uneasy, then at least embarrassed.
  >
  > Indeed, when one has succeeded in factoring a number, and has given its factors, this can be verified immediately. But it is a different matter when the methods used fail to discover any factor, and one then asserts that the number is prime. How could one then transmit to another such a totally personal conviction? Who could be convinced, without having redone all calculations, and without having understood the principles on which those calculations were used?

  Somewhat surprisingly, in 1975 Pratt proved that the problem *is* in NP. So there is an efficient way to convince someone that a number is prime! This shows that both primality and compositeness are in NP $\cap$ coNP, a class that contains P, and is believed to be bigger than P.

  A year or two later, several *randomized* polynomial time algorithms were discovered that solve the primality problem. This showed that primality (and hence also compositeness) is in BPP. These randomized algorithms have a small probability of error, but this probability can be made extremely small by repetition, as we will see later in the course.

  Finally, in 2002, Agrawal, Kayal, and Saxena presented for the first time a *deterministic* algorithm for solving the primality problem. This showed that primality (and hence also compositeness) is in P. Although running in polynomial time, their algorithm is significantly slower than the known randomized algorithms, and hence in practice people use randomized algorithms for testing primality.

- **Factoring:** the factoring problem is usually formulated as a search problem – given a number N, output its prime factors (e.g., given 15 output 3 and 5). There are several ways to formulate a decision problem related to factoring. Consider, for instance, the following: given numbers N and k, does N have at least k distinct prime factors? Although this is a valid formulation, it does not seem such a good one because even if we can solve this decision question, it is not clear how to actually factor N (and probably it's still hard).

  Instead, let us consider the following decision problem: given numbers N and K, does N have a prime divisor in the range $\{2,3,\ldots,K\}$? This formulation is better since if can solve it, we can also find the factors of N (use binary search to find the largest prime divisor, divide N by it, and continue by recursion).

  Is this problem in NP? Yes. We can take a verifier that expects to get as a witness a prime divisor L of N. It checks that L divides N, and that $2 \le L \le K$. What about checking that L is prime? We could do that too, but actually we don't need to: even if L is not prime, it has a prime factor p, and since L divides N, p divides N, and p is in the range $\{2,3,\ldots,K\}$. To convince yourself that this works, try to prove carefully the correctness of this verifier.

  The surprising thing is that this problem is also in coNP. We can construct a verifier that gets as a witness a list of numbers $p_1,\ldots,p_k$ that are supposed to be the factorization of N into prime factors. The verified checks that

  1. the product $p_1 \cdot p_2 \cdots p_k$ is equal to N,
  2. none of the p_i's is in the range $\{2,3,\ldots,K\}$, and
  3. all the p_i's are prime.

  The correctness of this verifier follows from the unique factorization theorem in number theory. Also the witness is of polynomial size since each factor p_i of N can be represented in at most $\log N$ bits and moreover, there are at most $\log N$ factors (because each factor is at least 2). What about the running time of the verifier? The first two steps are easy to perform in polynomial time. In order to perform the third step we need to use the algorithm by Agrawal et al.

  This works fine, but before 2002, we didn't know that there is a deterministic polynomial time algorithm for checking primality. So how did people show that the factoring problem is in coNP before 2002? We cannot use the randomized algorithms for checking primality, since an NP verifier is required by definition to be a deterministic machine (if we allow the verifier to be a randomized machine we obtain a class called MA). The idea is to ask the witness to convince us that these numbers are really prime. This can be done using Pratt's 1975 result. So now the witness not only includes numbers $p_1,\ldots,p_k$ but also includes for each number a proof that the number is prime. The verifier uses these proofs in the third step of verification.

  What does it mean that a problem is in $\text{NP} \cap \text{coNP}$? It is *not* known to imply that the problem is in P. In fact, most people do not believe that the factoring problem is in P, nor that it is in BPP. In fact, almost all our cryptographic systems rely on the assumption that the factoring problem is hard. If we could efficiently factor numbers, almost all our cryptographic systems would be broken. It is interesting to mention that in 1996 Peter Shor showed that factoring can be done efficiently on a *quantum* computer, and hence factoring is in the class BQP. This result is one of the main reasons many groups and organizations are interested in constructing quantum computers.

The fact that the problem is in NP $\cap$ coNP *does* have one important implication. It shows that the problem is unlikely to be NP-hard (since if it is NP-hard, we obtain that NP = coNP which is not believed to be true).

- **Graph Isomorphism**: given two graphs G_1 and G_2, are they isomorphic? In other words, is there a permutation $\pi : [n] \to [n]$ such that the graphs are the same up to renaming of the vertices according to π? Exercise: show that using a solution to this problem you can efficiently *find* the isomorphism π between any two given isomorphic graphs.

  This problem is clearly in NP, since the verifier can expect to get as a witness the permutation. Moreover, there are strong indications that the problem in not NP-complete. So is it in P? There are several algorithms that seem to solve this problem very well on lots of instances. However, there is no known algorithm that solves the problem in polynomial time on all instances, and therefore the problem is not known to be in P (although it might be!).

- **The halting problem:** This problem is not computable, and is therefore far far away from P and NP. All the problems we will encounter in this course are computable.

3 The Relation between NP and DTIME

Define EXP := $\bigcup_{c \geq 1}$ DTIME(2^{n^c}) as the class of all decision problems that can be solved in exponential time (where exponential means an exponential in a polynomial of the input size).

Claim 3.1 P $\subseteq$ NP $\subseteq$ EXP

Proof: We prove the two containments.

- P $\subseteq$ NP: Let L be any language in P. Our goal is to show it is in NP. Since L is in P, there is a polynomial time algorithm M that solves L. The same algorithm can be used as a polynomial time verifier that simply ignores the witness. The size of the witness in the definition of NP can be taken to be 0.

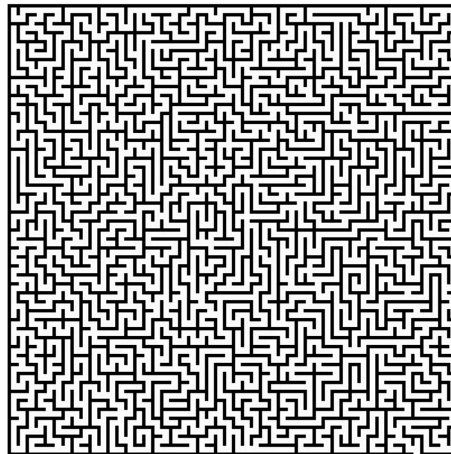
- NP $\subseteq$ EXP: Let L be any language in NP. By definition, there exists a polynomial p indicating the witness size and a polynomial-time verifier M. Construct a polynomial time algorithm M' that on input x, runs the verifier M on x and all $2^{\text{poly}(|x|)}$ possible witnesses w, and accepts if and only if at least one of those calls accepts. It follows from the correctness of the verifier M that M' computes L correctly, and moreover, it is easy to see that its running time is $O(2^{p(n)}\text{poly}(n))$, which is exponential, as required.

■

We don't know of any stronger relation between NP and DTIME classes. Most researchers believe that P $\neq$ NP but we cannot prove it (so it might be that P = NP). Similarly, most researchers believe that EXP $\neq$ NP, but again we don't know how to prove it (so it might be that EXP = NP). There's one thing we *do* know: P $\neq$ EXP. This follows from the time hierarchy theorem which we'll prove later.

4 An Alternative Definition of NP

An alternative way to define NP is using *nondeterministic algorithms*, and this is also the reason the class is called NP: Nondeterministic Polynomial time. Most textbooks define nondeterministic algorithms in terms of Turing machines, but let us try to define it first in terms of programs in C or Java. Nondeterministic programs are like normal programs, except they have one extra special command called NON_DET_CHOICE which allows the algorithm to branch simultaneously on both branches. We say that the algorithm accepts if *any* of the branches leads to acceptance. Otherwise, if all branches lead to rejection, we say that the algorithm rejects. The running time of the algorithm is the maximum running time over all branches. More precisely, we say that a nondeterministic algorithm runs in time $T(n)$ is for all $x \in \{0,1\}^*$ and *for all* nondeterministic choices, it stops within $T(|x|)$ steps.



To demonstrate the idea, consider the problem of finding if there is an exit from a maze. A 'nondeterministic mouse' can start at the beginning and each time there is a fork, split into two mice, one going left the other going right. A mice that gets to the cheese shouts 'ACCEPT' and the algorithm accepts. If no mouse finds the cheese, they all die and the algorithm rejects.

A more serious example is the following nondeterministic program to check if a given array is not all zero. Using standard deterministic programs would require scanning the whole array in time n, but a nondeterministic program can do this in time only $\log n$:

```
is_non_zero(int a[1,...,n])
{
    if n == 1 {
        if a[1] != 0 {
            accept
        } else {
            reject
        }
    }

    non_det_choice {
        is_non_zero(a[1,...,n/2])
```

```
    } or {
        is_non_zero(a[n/2+1,...,n])
    }
}
```

Definition 4.1 *We say that a langauge $L \subseteq \{0,1\}^*$ is in* $\mathrm{NTIME}(T(n))$ *if there exists $c > 0$ and a nondeterministic algorithm M running in time $c \cdot T(n)$ such that for all $x \in \{0,1\}^*$, $x \in L$ if and only if $M(x)$ accepts (i.e., there exists a nondeterministic choice of M leading to an accepting state).*

Theorem 4.2 $\mathrm{NP} = \cup_{c\geq 1}\mathrm{NTIME}(n^c)$

Proof: First let us show that $\mathrm{NP} \subseteq \cup_{c\geq 1}\mathrm{NTIME}(n^c)$. Let L be a language in NP. By definition, there exists an poly-time algorithm M and a polynomial p such that for all x,

$$x \in L \Leftrightarrow \exists w \in \{0,1\}^{p(|x|)} \text{ s.t. } M(x,w) = 1.$$

Consider the following non-deterministic algorithm M'. Given an input x, it starts by writing an arbitrary string w of length $p(|x|)$ using nondeterministic choices, and then runs the (deterministic) algorithm $M(x,w)$ and returns its answer. The first step can be implemented as

```
for i=1 to p(|x|) do {
    non_det_choice {
        w[i] = 0
    } or {
        w[i] = 1
    }
}
```

M' clearly runs in polynomial time. Moreover, it is easy to see that it accepts (i.e., has an accepting nondeterministic choice) if and only if there exists a w that makes $M(x,w)$ accept, which is equivalent to $x \in L$. Hence M' computes L, as required.

Now let us show that $\mathrm{NP} \supseteq \cup_{c\geq 1}\mathrm{NTIME}(n^c)$. Let L be a language in $\cup_{c\geq 1}\mathrm{NTIME}(n^c)$. This means that there exists a nondeterministic algorithm M running in time $p(n)$ for some polynomial p. Consider the following deterministic algorithm M' that on input x and $w \in \{0,1\}^{p(|x|)}$ runs the nondeterministic algorithm M by performing the nondeterministic choices using the bits in w. In more detail, we replace each nondeterministic choice with

```
if w[i++] == 0 {
    ...
} else {
    ...
}
```

It is easy to see that M' runs in polynomial time and that there exists a witness causing it to accept if and only if M has an accepting nondeterministic choice, which is to say that $x \in L$. Hence M' is a legal polynomial time verifier for L, and $L \in \mathrm{NP}$. ■

5 Reductions and NP-completeness

How would you define that a problem A is not harder than B? One intuitive attempt would be to say that the time it takes to solve A is not larger than the time it takes to solve B. This sounds like a good definition, but it is a big problem – we do not know how to argue about the time it takes to solve a problem. Even for seemingly difficult problems like independent set or Hamiltonian cycle, we cannot rule out the (very bizarre) possibility that there is a linear time algorithm solving them.

That is why we instead choose to work with the notion of *reductions*.

Definition 5.1 *A* polynomial-time (Karp) reduction *from a language $A \subseteq \{0,1\}^*$ to a language $B \subseteq \{0,1\}^*$ is a function $f : \{0,1\}^* \to \{0,1\}^*$ that is computable in polynomial time, and such that for all x,*

$$x \in A \Leftrightarrow f(x) \in B.$$

We use the notation $A \le_p B$ to denote that such a reduction exists. Remember that a reduction is 'from easy to hard'.

So a poly-time Karp reduction is simply a mapping that maps instances in A to instances in B and instances outside A to instances outside B. In order to show that $A \le_p B$, we need to

- Describe the reduction function f;
- Show that f can be computed in polynomial time;
- Prove that if $x \in A$ then $f(x) \in B$ (this is sometimes known as *completeness*);
- Prove that if $f(x) \in B$ then $x \in A$ (this is sometimes known as *soundness*).

Notice that having a poly-time Karp reduction from A to B means that if we could solve B, we could also solve A using only polynomial more time by simply running the algorithm for B on $f(x)$ where x is an input to A. We summarize this and other properties in the following theorem, whose proof is left as an exercise.

Theorem 5.2

- *If $A \le_p B$ and $B \le_p C$ then also $A \le_p C$ (this shows the* transitivity *of the $\le_p$ relation).*
- *If $A \le_p B$ and $B \in$ P then also $A \in$ P (this shows that the class* P *is* closed under poly-time reduction*).*
- *If $A \le_p B$ and $B \in$ NP then also $A \in$ NP (this shows that the class* NP *is* closed under poly-time reduction*).*

It is important to remember that there are other notions of reduction that we will encounter later. For instance, a polynomial-time Cook reduction from A to B is simply a polynomial time algorithm that solves A assuming it has the ability to solve B. Any polynomial-time Karp reduction is in particular also a polynomial-time Cook reduction. This is why showing a poly-time Karp reduction from A to B is a stronger result than showing a Cook reduction from A and B. Another 'problem' with Cook reduction is that NP is not believed to be closed under Cook reductions (as otherwise NP = coNP; think why!).

Another notion of reduction that we will work with later is logarithmic space reduction. This is a more restricted notion than poly-time Karp reduction and is important when working with space complexity classes.

Now we can finally define the notions of NP-hardness and NP-completeness.

Definition 5.3

- *We say that a language B is* NP-hard *if for all $A \in$* NP *it holds that $A \leq_p B$.*
- *We say that a language B is* NP-complete *if it is both in* NP *and* NP*-hard.*

Remember that "complete" always means "in the class + hard". The proof of the following theorem is left as an exercise.

Theorem 5.4

- *If A is* NP*-hard and is also contained in* P *then* P = NP.
- *Let A be an* NP*-complete language. Then $A \in P$ if and only if* P = NP.

It is important to remember that there is nothing "holy" about poly-time Karp reductions. We could also define NP-hard and NP-complete using other notions of reductions, and this could lead to different notions.

At this point one must stop and ask: are there any NP-complete problems at all? This is not at all obvious! There are classes that are not believed to have any complete problems. But as it turns out, there *are* NP-complete problems, and in fact *lots* of them. The Cook-Levin theorem which we will see in the next section proves that an important and quite natural problem known as SAT is NP-complete. But it is easy to show that the following (unnatural) problem is NP-complete:

$$\mathsf{TMSAT} = \{\langle \alpha, x, 1^n, 1^t \rangle : \exists u \in \{0,1\}^n \text{ s.t. } M_\alpha \text{ outputs 1 on input } \langle x, u \rangle \text{ within } t \text{ steps}\}$$

Theorem 5.5 TMSAT *is* NP*-complete.*

Proof: First we show that $\mathsf{TMSAT} \in$ NP. Consider the following verifier: given an instance $\langle \alpha, x, 1^n, 1^t \rangle$ and a witness $w \in \{0,1\}^n$, the verifier simulates M_α on the input $\langle x, w \rangle$ for t steps and checks that it outputs 1. The running time and the size of the witness are both polynomial in the size of the input (which is at least $t + n$).

Now let us show that TMSAT is NP-hard. Let L be any language in NP; our goal is to show a poly-time reduction from L to TMSAT. Since $L \in$ NP, there exists a poly-time verifier M_α that uses witnesses of length $p(|x|)$ and runs in time $q(|x|)$ for some polynomials p and q. Consider the following reduction from L to TMSAT: given an input x we output the instance $\langle \alpha, x, 1^{p(|x|)}, 1^{q(|x|)} \rangle$. This reduction can obviously be computed in polynomial time, and moreover, it is easy to check correctness. ■

6 The Cook-Levin theorem

The Cook-Levin theorem is probably the most fundamental theorem in theoretical computer science. It was proved independently by Cook in 1971 and Levin in 1973.

Recall that a *Boolean formula* is made of variables and their negations, connected by ANDs and ORs, e.g.,

$$(x_1 \wedge (x_9 \vee x_{10} \vee \overline{x_{19}})) \vee x_9 \vee \overline{x_1}.$$

We say that a formula is in *conjunctive normal form (CNF)* if it is the AND of ORs, e.g.,

$$(x_1 \vee x_9 \vee x_{10}) \wedge (x_9 \vee \overline{x_{11}}) \wedge (\overline{x_2} \vee x_9 \vee \overline{x_{10}}).$$

The SAT problem is the following: given a CNF formula, is it satisfiable?

Theorem 6.1 (Cook-Levin) SAT *is* NP*-complete.*

What would happen if we defined SAT with DNF formulas (where DNF formulas are ORs of ANDs)? We could also consider SAT with general formulas (as opposed to just CNF formulas); we choose to work with CNF formulas both because it leads to a stronger statement and because it requires no extra effort in the proof.

It is easy to see that SAT is in NP – just use a satisfying assignment as a witness. So in the rest of the proof we show that it is NP-hard.

Expressing constraints as CNFs: Before we get to the proof, let's start with a warm-up and try to express logical constraints as a CNF formula. For example, think how to express the constraint $x_1 = x_2$ as a CNF. This can be done using

$$(x_1 \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2).$$

In fact, it is not hard to prove that *any* constraint can be written as a CNF:

Claim 6.2 *Any Boolean constraint $f : \{0,1\}^k \to \{0,1\}$ can be expressed as a CNF formula of size at most $k2^k$ (where size is the total number of literals in the formula).*

The proof simply adds one clause (i.e., OR of literals) for each setting of the k variables that leads to 0. By combining all these clauses with ANDs, we end up with a CNF formula that expresses f.

One disadvantage of the claim is that it leads to exponential sized formulas. It turns out that for certain constraints we can do better. For instance, assume we have $2n$ Boolean variables and we want to express the constraint that $(x_1, \ldots, x_n) = (x_{n+1}, \ldots, x_{2n})$. Notice that this constraint is equivalent to the constraint that "$(x_1 = x_{n+1})$ and $(x_2 = x_{n+2})$ and ... and $(x_n = x_{2n})$". Since we already know how to express equality constraint between two bits, we obtain

$$(x_1 \vee \overline{x_{n+1}}) \wedge (\overline{x_1} \vee x_{n+1}) \wedge (x_2 \vee \overline{x_{n+2}}) \wedge (\overline{x_2} \vee x_{n+2}) \wedge \cdots \wedge (x_n \vee \overline{x_{2n}}) \wedge (\overline{x_n} \vee x_{2n}).$$

More generally,

> *any constraint that can be expressed as the AND of polynomially many "small constraints", each involving a constant number of bits, can also be expressed as a polynomial sized CNF.*

This is a crucial idea in the proof of the Cook-Levin theorem.

Illustration of the proof: Let L be an arbitrary language in NP. By definition, there exists a poly-time nondetereminstic algorithm M solving L. The core of the proof is a way to convert any given input x into a polynomial size CNF formula ϕ_x in such a way that ϕ_x is satisfiable if and only if $M(x)$ accepts. In other words, this is a poly-time reduction from L to SAT.

Before we describe this reduction formally, let us take as an example the NP problem of finding if there a way to get from the beginning to the end of a given maze (such as the maze on Page 8). Recall that the nondeterministic algorithm can be thought of as a mouse that starts walking from the beginning of the maze, and each time it gets to an intersection, split nondeterministically into all possible choices. A mouse that gets to the cheese at the end of the maze shouts 'accepts'. Now, given such a maze, how do we convert it into a CNF formula that checks whether there is a way to exit it? The idea is to use the Boolean variables to encode a movie showing how the mouse gets from the beginning to the end of the maze. In more detail, for each frame of the movie we have one Boolean variable for each location on the maze, indicating whether the mouse is there or not. The formula checks that: (1) in the first frame, the mouse sits in the beginning of the maze and there are no other mice anywhere; (2) in the last frame, the mouse is in the exit; and (3) for any i, the transformation from frame i to frame $i+1$ is legal. Crucially, these constraints can be expressed as the AND of polynomially many small constraints, each acting on a constant number of bits (for (1) and (2) this is obvious; for (3) we can check that for all 3×3 squares, the transformation from i to $i+1$ looks legal). This means that the entire formula can be encoded as a polynomial size CNF formula, as required.

Turing machines: The proof of the Cook-Levin theorem needs the algorithm M to be written in a programming language that is as simple as possible, since otherwise encoding it into a CNF formula becomes a nightmare. Therefore it is common to assume that M is given as a nondeterministic Turing machine. Recall that any (reasonable) programming language can be converted into the language of Turing machines, and that this transformation increases the running time by at most a polynomial factor; hence we our assumption that M is given as a nondeterministic Turing machine is without loss of generality.

To recall, a (deterministic) Turing machine is given by sets Q (internal machine state) and Γ (an alphabet), and a transition function $\delta : Q \times \Gamma \to Q \times \{Left, Right\}$ which describes the behavior of the machine given its internal state and what it sees in the cell under the head. A *nondeterministic* Turing machine is defined similarly, except that now we have two transition functions δ_0 and δ_1, and at each step the machine makes a nondeterministic choice of whether to use δ_0 or δ_1.

A snapshot of a Turing machine is described by (1) an element of Q, representing the internal state; (2) a list of elements from Γ, representing the contents of the tape; and (3) the location of the head. The reason Turing machine are so convenient to use is because so little changes in them from one step to the next (namely, the contents of the cell under the head can change, the head moves one location to the right or to the left, and the internal state changes; other than that, the entire state remains the same).

The proof: Let L be an arbitrary language in NP. By definition and our discussion above, there exists a nondetereminstic Turing machine M that solves L and runs in time $p(n)$ for some polynomial p. Let Q and Γ denote M's internal states and alphabet. Our goal is to show a reduction from L to SAT, i.e., our goal is to show a way to convert any input x to a CNF formula ϕ_x in such a way that ϕ_x is satisfiable if and only if $x \in L$. This is equivalent to showing that ϕ_x is satisfiable if and only if $M(x)$ has an accepting

path.

As before, we use the variables to encode a 'movie' of length $p(|x|)$, where now each frame is a snapshot of the Turing machine. Each such frame can be represented by $O(p(|x|))$ Boolean variables by using a constant number of variable to represent the status of each cell of the tape, where by status we mean both its contents (i.e., an element of Γ), whether the head is in that cell, and in case it is, the internal state of the machine. Hence for each cell we need $\lceil \log_2 \Gamma \rceil + 1 + \lceil \log_2 Q \rceil = O(1)$ Boolean variables. It is enough to store the status of the first $p(|x|)$ cells on the tape since the machine can never access any cells beyond those. So the total number of Boolean variables needed per frame is $O(p(|x|))$, and the total number of Boolean variable needed for the whole movie is $O(p(|x|)^2)$, which is polynomial in $|x|$.

$\downarrow q_0$ x_1	x_2	x_3	$\ldots$	x_n	$\perp$	$\ldots$	$\perp$
1	$\downarrow q_9$ x_2	x_3	$\ldots$	x_n	$\perp$	$\ldots$	$\perp$
$\downarrow q_{22}$ 1	0	x_3	$\ldots$	x_n	$\perp$	$\ldots$	$\perp$
$\vdots$							$\vdots$
1	0	1	$\ldots$	$\downarrow q_4$ 0	1	$\ldots$	0

It remains to describe a constraint that checks that the movie represents a correct run of the machine M. Crucially, the constraint should be made of polynomially many small constraints, each involving a constant number of variables. This will imply that it can be represented as a CNF of polynomial size. The constraint consists of the following checks:

- **Initialization:** The first frame contains the bits of x followed by spaces; the first cell contains the head indicating the machine is in internal state q_0, and all other cells do not contain a head.

- **Termination:** The last frame contains a "1" as the contents of the first cell. (Here we assume that the machine M indicates acceptance by writing 1 in the first cell; clearly this can be done without loss of generality.)

- **Progress:** For all $0 \le t < p(|x|)$, check that frame $t+1$ looks like a legal step of the nondeterministic machine applied to frame t. This requires checking that:

  - for any 2×1 rectangle, if there is no head in the top cell, then the contents of the top and bottom cell are equal;
  - for any 2×3 rectangle, if there is a head in the top center cell, then the state of the bottom three cells (contents, location of head, and internal state) corresponds to applying either δ_0 or δ_1 on the top three cells; and
  - for any 2×3 rectangle, if there a head in the middle bottom cell, then there is a cell in one of the three top cells.

It is not difficult to verify that this constraints can be written as the AND of polynomially many small constraints, each involving a constant number of variables, and hence it can be written as a polynomial size CNF formulas ϕ_x. It remains to prove that ϕ_x is satisfiable if and only if $M(x)$ has an accepting path. The 'completeness' direction is easy: if $M(x)$ has an accepting path, then the assignment that corresponds to the movie that shows the accepting run of the machine M passes all the tests above, and hence ϕ_x is satisfiable. The other direction (soundness) requires showing that if ϕ_x is satisfiable then $M(x)$ has an accepting path. In other words, we need to prove that a satisfying assignment to ϕ_x must correspond to a legal accepting run of $M(x)$. The proof of this is quite technical, and is left as an exercise. Notice that this proof is the one that shows that our constraints above are sufficient; to help you understand the proof, think why the proof fails if we remove any of the constraints above.

Exercise: Given a satisfying assignment to ϕ_x, show how to extract in polynomial time the list of nondeterministic choices that make $M(x)$ accept.

Exercise: Modify the proof so that it works with (deterministic) verifiers, instead of nondeterministic algorithms. (Of course a deterministic verifier can be transformed to a nondeterministic algorithm, but the intention here is to modify the proof so that it directly works with verifiers.)