Introduction to Complexity Theory

Mahesh Viswanathan

Fall 2018

We will now study the time and space resource needs to solve a computational problem. The running time and memory requirements of an algorithm will be measured on the Turing machine model we introduced in the previous lecture. Recall that in this model, the Turing machine has a designated read-only input tape, and finitely many read/write worktapes. Unless we talk about function computation in the context of reductions, the output tape does not really play a role, and so we can assume for most of the time that these machines don't have an output tape. We will consider both the deterministic and nondeterministic versions of such Turing machines.

Computational resources needed to solve a problem depend on the size of the input instance. For example, it is clearly easier to compute the sum of two one digit numbers as opposed to adding two 15 digit numbers. The resource requirements of an algorithm/Turing machine are measured as a function of the input size. We will only study time and space as computational resources in this presentation. We begin by defining time bounded and space bounded Turing machines, which are defined with respect to bounds given by functions $T : \mathbb{N} \to \mathbb{N}$ and $S : \mathbb{N} \to \mathbb{N}$. Our definitions apply to both deterministic and nondeterministic machines.

Definition 1. A (deterministic/nondeterministic) Turing machine M is said to run in *time* $T(n)$ if on any input u, *all* computations of M on u take at most $T(|u|)$ steps; here $|u|$ refers to the length of input u.

A (deterministic/nondeterministic) Turing machine M is said to use *space* $S(n)$ if on any input u, *all* computations of M on u use at most $S(|u|)$ cells of the worktapes. In this context, a worktape cell is said to be used if it is written to at least once during the computation. Notice that, if a worktape cell is written multiple times during a computation, it counts as only one cell when measuring the space requirements; thus, worktape cells can be reused without adding to the space bounds.

It is worth examining Definition 1 carefully. Our requirement for a Turing machine running within some time or space bound applies to *all* computations, whether they are accepting or not. Notice also that the definition is the same for both deterministic and nondeterministic models — in a deterministic machine the *unique* computation on a given input must satisfy the resource bounds, and in a nondeterministic machine, *all* computations on the input must satisfy the bounds. In particular, if a Turing machine (deterministic or nondeterministic) runs within a time bound, then it *halts* in every computation of every input.

Having defined time and space bounded machines, we can define the basic complexity classes which are collections of (decision) problems that can be solved within certain time and space bounds.

Definition 2. We define the following basic complexity classes.

- A language $A \in \text{DTIME}(T(n))$ iff there is a *deterministic* Turing machine that runs in time $T(n)$ such that $A = \mathbf{L}(M)$.
- A language $A \in \text{NTIME}(T(n))$ iff there is a *nondeterministic* Turing machine that runs in time $T(n)$ such that $A = \mathbf{L}(M)$.
- A language $A \in \text{DSPACE}(S(n))$ iff there is a *deterministic* Turing machine that uses space $S(n)$ such that $A = \mathbf{L}(M)$.
- A language $A \in \text{NSPACE}(S(n))$ iff there is a *nondeterministic* Turing machine that uses space $S(n)$ such that $A = \mathbf{L}(M)$.

1 Linear Speedup and Compression

Our computational model of Turing machines, and our definitions of time and space bounded computations are robust with respect to constant factors. This observation is captured by two central results in theoretical computer science, namely, the linear speedup and compression theorems. It says that one can always improve the running time or space requirements for solving a problem by a constant factor.

Theorem 3 (Linear Speedup)**.** *If $A \in DTIME(T(n))$ (or $A \in NTIME(T(n))$) and $c > 0$ is any constant, then $A \in DTIME(cT(n) + n)$ ($A \in NTIME(cT(n) + n)$).*

Proof. Let $A = \mathbf{L}(M)$. We will describe a machine M' which will simulate k steps of M in 8 steps; if $k > \frac{8}{c}$, we will get the desired result. M' will have one more work tape, a much larger tape alphabet, and control states than M.

- M' copies the input onto the additional work-tape in compressed form: k successive symbols of M will be represented by one symbol in M'. Time taken is n. M' will maintain compressed worktapes as well.
- M' uses the additional worktape as "input tape". The head positions of M, within the k symbols represented by current cells, is stored in finite control.

One *basic move* of M' (consisting of 8 steps), will simulate k steps of M as follows.

- At the start of basic move, M' moves its tape heads one cell left, two cells right and one cell left, storing the symbols read in the finite control. Now, M' knows all symbols within the radius of k cells of any of M's tape heads. This takes 4 steps.
- Based on the transition function of M, M' can compute the effect of the next k steps of M.
- Using any additional (at most) 4 steps, M' updates the contents of its tapes as a result of the k steps, and moves the heads appropriately. □

Theorem 4 (Linear Compression)**.** *If $A \in DSPACE(S(n))$ (or $A \in NSPACE(S(n))$) and $c > 0$ is any constant then $A \in DSPACE(cS(n))$ ($A \in NSPACE(cS(n))$).*

Proof. Increase the tape alphabet size and store worktape contents in compressed form as in Theorem 3. □

Theorems 3 and 4 suggest that when analyzing the time and space requirements of an algorithm we can ignore constant terms. This leads to the use of the order notation.

Definition 5. Consider functions $f : \mathbb{N} \to \mathbb{N}$ and $g : \mathbb{N} \to \mathbb{N}$.

- $f(n) = O(g(n))$ if there are constants c, n_0 such that for $n > n_0$, $f(n) \leq cg(n)$. $g(n)$ is an *asymptotic upper bound*.
- $f(n) = \Omega(g(n))$ if there are constants c, n_0 such that for $n > n_0$, $f(n) \geq cg(n)$. $g(n)$ is an *asymptotic lower bound*.

2 Robust Complexity Classes

The complexity classes identified in Definition 2 is a very fine classification of problems. They include complexity classes whose classification of problems is sensitive to our use of Turing machines as a model of computation. Ideally we would like to study complexity classes such that if a problem is classified in a certain class then that classification should be "platform independent". That is, whether we choose to study complexity on Turing machines or Random Access Machines, our observations should still hold. They should also be invariant under small changes to the Turing machine model, like changing the number of worktapes, alphabet, nature of the tapes, etc. There is a strengthening of the Church-Turing thesis, called the *invariance thesis* articulated by Church, that underlies our belief in the robustness of the Turing machine model, subject to small changes in the time and space bounds. It says that

Any effective, mechanistic procedure can be simulated on a Turing machine using the same space (if space is $\geq \log n$) and only a polynomial slowdown (if time $\geq n$)

In addition to complexity classes be robust to changes to the computational platform, we would like the classes to be closed under function composition — making function/procedure calls to solve subproblems is a standard algorithmic tool, and we would like the complexity to remain the same as long as the subproblems being solved are equally simple. Finally, we would like our complexity classes to capture natural, "interesting", real-world problems. For these reasons, we typically study the following complexity classes that provide a coarser classification of problems than that provided in Definition 2.

Definition 6. Commonly studied complexity classes are the following.

$\mathrm{L} = \mathrm{DSPACE}(\log n)$	$\mathrm{NL} = \mathrm{NSPACE}(\log n)$
$\mathrm{P} = \cup_k \mathrm{DTIME}(n^k)$	$\mathrm{NP} = \cup_k \mathrm{NTIME}(n^k)$
$\mathrm{PSPACE} = \cup_k \mathrm{DSPACE}(n^k)$	$\mathrm{NPSPACE} = \cup_k \mathrm{NSPACE}(n^k)$
$\mathrm{EXP} = \cup_k \mathrm{DTIME}(2^{n^k})$	$\mathrm{NEXP} = \cup_k \mathrm{NTIME}(2^{n^k})$

In addition to the above classes, for any class $\mathcal{C}$, co-$\mathcal{C} = \{A \mid \overline{A} \in \mathcal{C}\}$. Please note that co-$\mathcal{C}$ is *not* the complement of $\mathcal{C}$ but instead is the collection of problems whose complements are in $\mathcal{C}$.

3 Relationship between Complexity Classes

We begin by relating time and space complexity classes.

Theorem 7. *$DTIME(T(n)) \subseteq DSPACE(T(n))$ and $NTIME(T(n)) \subseteq NSPACE(T(n))$*

Proof. A Turing machine can scan at most one new worktape cell in any step. Therefore, the number of worktape cells used during a computation cannot be more than the number of steps. □

Theorem 8. *$DSPACE(S(n)) \subseteq DTIME(n \cdot 2^{O(S(n))})$ and $NSPACE(S(n)) \subseteq NTIME(n \cdot 2^{O(S(n))})$. In particular, when $S(n) \geq \log n$, we have $DSPACE(S(n)) \subseteq DTIME(2^{O(S(n))})$ and $NSPACE(S(n)) \subseteq NTIME(2^{O(S(n))})$.*

Proof. Consider a problem A (in $\mathrm{DSPACE}(S(n))$/$\mathrm{NSPACE}(S(n))$) and a $S(n)$-space bounded (deterministic/nondeterministic) Turing machine M computing A. Without loss of generality, we can assume that M has a single worktape, in addition to a read-only input tape. Recall that a configuration of a Turing machine consists of the state, the position of the input head, and contents of the worktape including the worktape head position. On an input of length n, the input head position is going to be a number between 0 and n. If at most $S(n)$ worktape cells are used, then the worktape contents with the head position is of the form $u\sqcup^\omega$, where $|u| \leq S(n) + 2$. If there are q states, and t tape symbols, then the number of different configurations is $\leq q(n+1)t^{S(n)+2} = n2^{O(S(n))}$.

Any computation of M is a sequence of configurations, and so if a computation has more that $n2^{O(S(n))}$ steps, then some configuration repeats. If M is deterministic and if it has a computation of length $\geq n2^{O(S(n))}$, that means the computation is looping and it will never terminate. If M is nondeterministic, and if it has a terminating computation of length $\geq n2^{O(S(n))}$ then since some configuration repeats in the computation, we can remove the computation between the repeats and get a shorter computation. Thus, whether M is deterministic or nondeterministic, if M accepts the input, then M accepts the input using a computation of length $\leq n2^{O(S(n))}$.

This suggests the following time-bounded algorithm M' to solve A — simulate M for $n2^{O(S(n))}$ steps and accept only if M accepts within that many steps. Remember this algorithm will require a little extra storage to count $n2^{O(S(n))}$ steps. The extra storage is $\log(n2^{O(S(n))}) = (\log n)O(S(n))$; if $S(n) \geq \log n$ then this is just $O(S(n))$ storage, and so the time-bounded algorithms is also $S(n)$-space bounded.

There is a subtle point we buried in the argument above, which is that in order to count $n2^{O(S(n))}$ steps, we need to *compute* this number *within time* $n2^{O(S(n))}$ so that the total running time remains the same.

For the functions $S(n)$ used to define the complexity classes in Definition 6, one can show $n2^{O(S(n))}$ can be computed quickly. However, it turns out that this assumption of computability is not needed because of a useful trick. Let us start with simple case when M is nondeterministic. In this case, M' will guess a number S (the space bound in an accepting computation), compute $n2^S$, and guess a computation of length $\leq n2^S$. M' will check that during the computation the space bounds never exceed S, and that the guessed computation is indeed an accepting computation of M. If either of these checks fail, M' will reject. Notice that if M accepts the input within the space bound $S(n)$, then there are appropriate guesses M' can make, which will cause M' to accept.

Let us now consider the case when M is deterministic. In this case, the deterministic time-bounded simulation of M, namely M', will proceed as follows. M' will first start with space bound $S = 1$. It will start simulating M, making sure that the computation does not exceed 2^S steps. If during the simulation of M with space bound S, M uses more than S worktape cells, then M' will increase the space bound to $S + 1$ and restart M's simulation on the input from the beginning. One can argue that if M uses $S(n)$ space, then the total time taken by M' to carry out the multiple restarted simulations of M does not exceed $n2^{O(S(n))}$. □

An immediate consequence of Theorems 7 and 8 are the following relationships between the complexity classes.

Corollary 9.

$$L \subseteq P \subseteq PSPACE \subseteq EXP$$
$$NL \subseteq NP \subseteq NPSPACE \subseteq NEXP$$

We will establish relationships between deterministic and nondeterministic complexity classes.

Theorem 10. *$DTIME(t(n)) \subseteq NTIME(t(n))$ and $DSPACE(s(n)) \subseteq NSPACE(s(n))$.*

Proof. This follows from the fact that, by definition, every deterministic Turing machine is a special nondeterministic Turing machine, namely, those that have exactly one transition enabled from every non-halting configuration. □

Nondeterministic complexity class can also be related to deterministic complexity classes. In fact, we now prove a result that subsumes the containment results for nondeterministic classes established in Theorems 7 and 8.

Theorem 11. *$NTIME(T(n)) \subseteq DSPACE(T(n))$ and $NSPACE(S(n)) \subseteq DTIME(n2^{O(S(n))})$.*

Proof. Let us begin by proving the first inclusion. Consider $A \in \text{NTIME}(T(n))$ and let M be $T(n)$-time bounded nondeterministic machine recognizing A. On any input w of length n, the computations of M can be organized as a tree, and since M runs in time $T(n)$, this tree has height $T(n)$. Now the deterministic algorithm D to solve A will perform a *depth first search* (DFS) on this computation tree of M, constructing this tree as it is explored, and accepting if some node in this computation tree corresponds to an accepting configuration. The space needed by D to perform this DFS is the memory needed to store the call stack. The stack during a DFS keeps track of the path being currently explored in the tree to enable backtracking. Since the computation tree of M is of height $T(n)$, the height of the call stack is also $T(n)$. A naïve implementation of the DFS algorithm will store the sequence of tree vertices on the current path; since in this case each vertex is a configuration of M, these can be represented by strings of length $T(n)$ (as worktape cells cannot exceed $T(n)$ as in Theorem 7). This gives us a space bound of $T(n)^2$ for algorithm D. However, instead of storing the actual configurations in the computation being currently explored, D can just store the sequence of nondeterministic choices made by M in the current computation. With this information about the nondeterinistic choices, D can *reconstruct* the configuration at the end of a sequence of steps, by resimulating M from the beginning — this increases the running time of D, but reduces the space requirements of D which is what we care about for this result. If M has k choices at each step, the stack of D during DFS is simply a k-ary string of length $\leq T(n)$, which means that D is $T(n)$-space bounded.

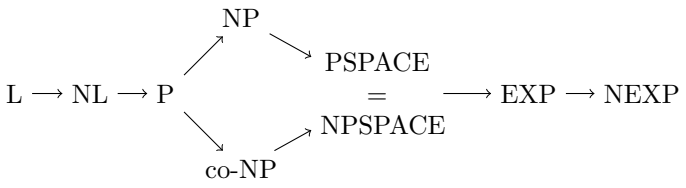

Figure 1: Relationship between Complexity Classes. $\rightarrow$ indicates containment, though whether it is strict is unknown.

For the second result, let us consider $A \in \text{DSPACE}(S(n))$ and a nondeterministic Turing machine M that recognizes A in $S(n)$ space. On a given input w of the length n, it is useful to define the notion of a *configuration graph* of M. The configuration graph is a directed graph that has as vertices, configurations of M, and has an edge from c_1 to c_2, if M can move from configuration c_1 to configuration c_2 in one step given input w. Observe that M accepts w if an accepting configuration is reachable from the initial configuration in this configuration graph. Notice also that since M is $S(n)$-space bounded, the total number of vertices in this graph is $\leq n2^{O(S(n))}$ (see proof of Theorem 8). Now we can run our favorite graph search algorithm (depth first search or breadth first search) on this configuration graph to see if an accepting configuration is reachable; the graph will be constructed on-th-fly as it is being explored. Such an algorithm (which deterinistic), takes time that is linear in the size of the graph, which gives us a $n2^{O(S(n))}$ deterministic algorithm for A. □

Our new observations relating deterministic and nondeterministic complexity classes gives us the following relationships.

Corollary 12.

$$L \subseteq NL \subseteq P \subseteq NP \subseteq PSPACE \subseteq NPSPACE \subseteq EXP \subseteq NEXP$$

An important result due to Savitch, relates nondeterministic and deterministic space complexity classes. Its proof is beyond the scope of this course.

Theorem 13 (Savitch)**.** *For $S(n) \geq \log n$, $NSPACE(S(n)) \subseteq DSPACE(S(n)^2)$. In particular, this means that $PSPACE = NPSPACE$.*

Putting all our observations together we get the relationships shown in Figure 1. It is worth observing that for any deterministic complexity class $\mathcal{C}$, $\mathcal{C} = \text{co-}\mathcal{C}$; this is because an algorithm for $\overline{A}$ is to run the deterministic algorithm for A and fli the final answer. Thus, P = co-P. Further, from Theorem 10, we have co-P $\subseteq$ co-NP, giving us the containment P $\subseteq$ co-NP. Finally, due the space hierarchy theorem, we also know that L $\neq$ PSPACE and NL $\neq$ PSPACE, and fromthe time hierarchy theorem, we know that P $\neq$ EXP and NP $\neq$ NEXP; the hierarchy theorems are beyond the scope of this course.

4 P and NP

The *Cobham-Edmonds Thesis*, named after Alan Cobham and Jack Edmonds, asserts that the only computational problems that have "efficient" or "feasible" algorithmic solutions are those that belong to P. In other words, P is the collection of *tractable* computational problems. There are many features of P that provide a justification for this view.

- The invariance thesis suggests that any problem in P can be solved in polynomial time on *any* reasonable computational model. Thus, the statement of a problem being efficiently computable is platform independent.

- Most encodings of an input structure are polynomially related in terms of their length. Thus, if a problem is in P for one encoding, it will be in P even if input instances are encoded in a different manner. Therefore, P is insensitive is problem encodings.
- Most natural problems in P have algorithms whose running time is bounded by a low-order polynomial. Thus, their running times are likely to be low for most problem instances.
- The assymptotic growth of polynomials is moderate when compared to the astounding growth of exponential functions. Thus, problems in P are likely to be feasibly solved even on large problem instances.

The crux of the Cobham-Edmonds thesis is that for problem to be solvable in practice, it should have a polynomial time algorithm. Therefore, must effort in the past 50 years has been devoted to understanding the class of problems in P. In particular, can we prove that the complexity classes containing P in Figure 1, like NP and PSPACE, also have efficient solutions, i.e., are contained in P? Or can we say for certain that some problems in NP and PSPACE *cannot* be solved efficiently?

4.1 Alternate characterization of NP

We defined NP as the collection of problems that can be solved in polynomial time on a nondeterministic Turing machine. In this section, we will give an alternate definition, namely, as those problems that are efficiently "verifiable".

Definition 14. A language A is *polynomially verifiable* if there is a $k \in \mathbb{N}$ and a deterministic Turing machine V such that

$$A = \{w \mid \exists p.\ V \text{ accepts } \langle w, p\rangle\}$$

and V takes at most $|w|^k$ steps on input $\langle w, p\rangle$, i.e., V running time is *independent* of the length of p. Here V is called a *verifier* for A, and for $w \in A$, the strings p such that $\langle w, p\rangle$ is accepted by V are called a *proof* of w (with respect to V).

The notion of a language A being polynomial verfiable says that when a string $w \in A$, there is a *proof* p (maybe even more than one) such that w augmented with p "convinces" V, i.e., causes V to accept. However, if $w \notin A$ then there is no proof string p that can convince V of w's membership in A. Notice also that V's running time on input $\langle w, p\rangle$ is independent of the length of p; it always runs in time $|w|^k$ no matter what p is. Now, since in $|w|^k$ steps, V cannot read more that $|w|^k - |w|$ bits of p, we can without loss of generality assume that p is a string whose length is bounded by a polynomial in the length of w. Thus, we could informally say that A is polynomially verifiable, if for any string $w \in A$ there is a "short" proof (of polynomial length) that can be efficiently checked (in polynomial time) by a verifier, and if $w \notin A$ there is no proof that can convince a verifier.

A language being polynomially verifiable is equivalent to a problem having a nondeterministic polynomial time verifier.

Theorem 15. *$A \in$ NP if and only if A is polynomially verifiable.*

Proof. Consider $A \in$ NP, and let M be nonterministic Turing machine recognizing A in time n^k for some k. We can assume without loss of generality that M has at most two choices at any given step. The verifier V for A will work as follows. On input $\langle w, p\rangle$, where p is a binary string, it will first copy w onto a work-tape, and compute n^k. It will then simulate M for n^k steps using the work-tape with w as the input tape, taking p to be the sequence of nondeterministic choices. V accepts $\langle w, p\rangle$ if M accepts w with p as the nondeterministic choices. Observe that V is a deterministic algorithm running in $O(n^k)$ time on $|w| = n$. Further $A = \{w \mid \exists p.\ V \text{ accepts } \langle w, p\rangle\}$.

Conversely, suppose V is a polynomial time verifier for A. Suppose V runs in time $|w|^k$ on input $\langle w, p\rangle$. The nondeterministic algorithm M for A will work as follows. On input w, M will guess a string p of length $|w|^k$. Then M will simulate V on w and the guessed string p, accepting if and only if V accepts. It is easy to see that $\mathbf{L}(M) = A$ and M runs in time $O(n^k)$. □

Thus, NP is the collection of all problems A whose membership question has short, efficiently checkable proofs. The question of whether all problems in NP have polynomial time algorithms — whether P $\stackrel{?}{=}$ NP — is thus the question of whether every problem that has a short, efficiently checkable proofs also have the property that these proofs can be *found* efficiently. Phrased in this manner, the likely answer seems to be no. There are also results that seem to suggest that P is likely to be not equal to NP, though a firm resolution of this question has eluded researchers for the past 50 years.

4.2 Reductions, Hardness and Completeness

In an effort to resolve the P versus NP question, researchers have tried to identify cannonical problems whose study can helps address this challenge. The goal is to identify, in some sense, the most difficult problems in NP such that either (a) they are candidate problems that may not have polynomial time algorithms, or (b) finding a polynomial time algorithms for these problems will constructively demonstrate that P = NP. In order to identify such difficult problems, we need to be able to compare the difficulty of two problems. For this the most convenient technique is that of reductions. Unlike, many-one reductions introduced before, we will require that these not be computed in polynomial time.

Definition 16. A *polynomial time reduction* from A to B is a *polynomial time computable function* f such that for every input w,

$$w \in A \text{ if and only if } f(w) \in B.$$

In such a case we say that A is *polynomial time reducible* to B and is denoted by $A \leq_{\mathrm{P}} B$.

Example 17. Consider the following problems.

$$\text{SAT} = \{\langle\varphi\rangle \mid \varphi \text{ is in CNF and } \varphi \text{ is satisfiable}\}$$
$$k\text{-COLOR} = \{\langle G, k\rangle \mid G \text{ is an undirected graph that can be colored using } k \text{ colors}\}$$

Recall that in the lecture on compactness we showed that for any graph G and $k \in \mathbb{N}$ there is a set of clauses $\Gamma_{G,k}$ such that G is k-colorable if and only if $\Gamma_{G,k}$ is satisfiable. The number of clauses in $\Gamma_{G,k}$ is proportional to the number of vertices and edges in G, and the each clauses has at most k-literals. It is also easy to see that $\Gamma_{G,k}$ can be constructed from G in time that linear in the size of G. Thus, these observations together establish that k-COLOR $\leq_{\mathrm{P}}$ SAT.

Example 18. A formula φ in CNF is said to be in 3-CNF if every clause in φ has exactly 3 literals. For example, $(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_4) \wedge (x_4) \wedge (\neg x_2 \vee \neg x_3 \vee x_4)$ is not in 3-CNF, while $(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_4 \vee x_2)$ is in 3-CNF. Recall the SAT problem is one whether given a formula φ in CNF, we need to determine if φ is satisfiable. A special case of this problem is one where the input formula is promised to be in 3-CNF. Formally we have,

$$\text{3-SAT} = \{\langle\varphi\rangle \mid \varphi \text{ is in 3-CNF and is satisfiable}\}.$$

Since 3-SAT is a "special" version of SAT, the identity function is a reduction from 3-SAT to SAT; thus, 3-SAT $\leq_{\mathrm{P}}$ SAT. It turns out the one also has a reduction the other way around.

The reduction from SAT to 3-SAT is as follows. Consider a CNF formula φ; it will be convenient to this of φ as a set of clauses. Our reduction will convert (in polynomial time) each clause $c \in \varphi$ into a 3-CNF formula $f(c)$ such that c and $f(c)$ are satisfied by (almost) the same set of truth assignments. Then $f(\varphi) = \{f(c) \mid c \in \varphi\}$, and it will be the case that φ is satisfiable iff $f(\varphi)$ is satisfiable.

Let us now describe the translation of clauses. The translation of clause c will depend on how many literals c has. Let $c = \ell_1 \vee \vee \ell_2 \vee \cdots \vee \ell_k$. Depending on k, we have the following cases.

Case $k = 1$ Let u and v be "new" propositions not used before. Define $f(c)$ to be

$$(\ell_1 \vee u \vee v) \wedge (\ell_1 \vee u \vee \neg v) \wedge (\ell_1 \vee \neg u \vee v) \wedge (\ell_1 \vee \neg u \vee \neg v)$$

Case $k = 2$ Let u be a "new" proposition. $f(c)$ is given by

$$(\ell_1 \vee \ell_2 \vee u) \wedge (\ell_1 \vee \ell_2 \vee \neg u)$$

Case $k = 3$ In this case $f(c) = c$.

Case $k > 3$ Let $y_1, y_2, \ldots y_{k-3}$ be new propositions. Then $f(c)$ is

$$\begin{aligned}(\ell_1 \vee \ell_2 \vee y_1) \ \wedge \ (\ell_3 \vee \neg y_1 \vee y_2) \ \wedge \ (\ell_4 \vee \neg y_2 \vee y_3) \ \wedge \cdots \\ \wedge (\ell_{k-2} \vee \neg y_{k-4} \vee y_{k-3}) \ \wedge \ (\ell_{k-1} \vee \ell_k \vee \neg y_{k-3})\end{aligned}$$

It is easy to see that f can be computed in time that is linear in the size of φ. Moreover, φ is satisfiable iff $f(\varphi)$ is satisfiable (left as exercise). Thus, f is a polynomial time reduction showing SAT $\leq_{\mathrm{P}}$ 3-SAT.

Polynomial time reductions satisfy properties similar to many-one reductions: they are transitive and if A reduces to B then $\overline{A}$ reduces to $\overline{B}$.

Proposition 19. *The following properties hold for polynomial time reductions.*

- *If* $A \leq_{\mathrm{P}} B$ *then* $\overline{A} \leq_{\mathrm{P}} \overline{B}$.
- *If* $A \leq_{\mathrm{P}} B$ *and* $B \leq_{\mathrm{P}} C$ *then* $A \leq_{\mathrm{P}} C$.

Proof. The detailed proof of these observations is left as an exercise. But the sketch is as follows. If f is a polynomial time reduction from A to B then f is also a polynomial time reduction from $\overline{A}$ to $\overline{B}$. And if f is a polynomial time reduction from A to B and g is a polynomial time reduction from B to C, then $g \circ f$ is a polynomial time reduction from A to C. □

Finally, polynomial time reductions do serve as a way to compare the computational difficulty of two problems. We show that if $A \leq_{\mathrm{P}} B$ and B is "easy" then A is easy.

Theorem 20. *If* $A \leq_{\mathrm{P}} B$ *and* $B \in P$ *then* $A \in P$.

Proof. Let f be a polynomial time reduction from A to B and let M be a deterministic polynomial time algorithm recognizing B. Then the polynomial time algorithm N for A does the following: On input w, compute $f(w)$ and then run M on $f(w)$. It is easy to see that N recognizing A from the properties of a reduction.

The tricky step is to argue that N runs in polynomial time. Let us assume that f is computed in time n^k and let M run in time n^ℓ. Since f can be computed in time n^k, it means that $|f(w)| \leq |w|^k$; this is because a single step in the computation of f can produce at most one bit of $f(w)$. Therefore, the total running time of N is $|w|^k$ (time to compute $f(w)$) + $(|w|^k)^\ell$ (time to run M on $f(w)$ which is a string of length $|w|^k$). This is bounded by $O(n^{k\ell})$ which is polynomial. □

In Theorem 20, we could have replaced P by any of the complexity classes in Figure 1 that contain P, and the proof would go through. Thus, polynomial time reductions are an appropriate lens by which measure the relative difficulty of problems that belong to complexity classes that contain P.

Definition 21 (Hardness and Completeness)**.** Let $\mathcal{C}$ be a complexity class in Figure 1 that contains P. A is said to be $\mathcal{C}$*-hard* iff for every $B \in \mathcal{C}$, $B \leq_{\mathrm{P}} A$.

A is said to be $\mathcal{C}$*-complete* iff $A \in \mathcal{C}$ and A is $\mathcal{C}$-hard.

In other words, informally, a problem is $\mathcal{C}$-hard if it is at least as difficult as any problem in $\mathcal{C}$. It is $\mathcal{C}$-complete if in addition it also belongs to $\mathcal{C}$. Fixing $\mathcal{C}$ to be NP, we could say that a problem is NP-complete if it is the "hardest" problem that belongs to NP. Because of their status as the most difficult problems in NP, they are candidate problems to study to help resolve the P versus NP question. This is captured by the following observation.

Proposition 22. *If A is NP-hard and $A \in P$ then NP = P.*

Proof. Consider any problem $B \in$ NP. Since $B \leq_{\mathrm{P}} A$ and $A \in$ *polytime*, by Theorem 20, we have $B \in$ P. □

In the absence of a firm resolution of the P versus NP questions, classifying a problem as NP-hard suggests that it is unlikely that the problem has a polynomial time algorithm given our belief that P $\neq$ NP.

Many natural problems are NP-complete. The historically (and pedagogically) first problem known to be NP-complete is SAT (see Example 17). Recall that

$$\mathrm{SAT} = \{\langle\varphi\rangle \mid \varphi \text{ is in CNF and } \varphi \text{ is satisfiable}\}$$

Theorem 23 (Cook-Levin)**.** *SAT is NP-complete.*

Proof. Any proof showing a problem to be NP-complete has two parts. First is an argument that the problem belongs to NP and the second that it is hard.

Membership in NP. The NP-algorithm for SAT is as follows — Guess a truth assignment, evaluate the formula on the truth assignment, and accept if the formula evaluates to true; otherwise reject. Guessing (nondeterministically) a truth assignment takes time which is linear in the number of propositions in the formula, which is linear in the size of the formula, and evaluating a formula on a truth assignment also takes time that is linear in the size of the formula, where the evaluation algorithm computes the value in a "bottom-up" fashion. Thus, the total running time is polynomial.

NP-hardness. Consider $A \in$ NP. Let M be a nondeterministic TM recognizing A in time n^ℓ. For an input x, we will construct (in polynomial time) a formula $f_M(x)$ in CNF such that M accepts x (i.e., $x \in A$) iff $f_M(x)$ is satisfiable. $f_M(x)$ will encode constraints on a computation of M on x such that a satisfying assignment to $f_M(x)$ will describe "how M accepts x". That is, $f_M(x)$ will encode that

- M starts in an initial configuration with input x
- Each configuration follows from the previous one in accordance with the transition function of M
- The accepting state is reached in the last step.

Let us formallize this intuition by giving a precise construction. We begin by identifying the set of propositions we will use, and their informal interpretation.

Propositional Variables. The propositions of $f_M(x)$ will be as follows.

Name	Meaning if set to $\top$	Total Number
InpSymb@(b,p)	Input tape stores b at position p	$O(\lvert x\rvert)$
TapeSymb@(j,b,p,i)	Work-tape j stores b at position p at time i	$O(\lvert x\rvert^{2l})$
InpHd@(h,i)	Input head at position h at time i	$O(\lvert x\rvert \cdot \lvert x\rvert^{l})$
TapeHd@(j,h,i)	Work-tape j's head at position h at time i	$O(\lvert x\rvert^{2l})$
St@(q,i)	State is q at time i	$O(\lvert x\rvert^{l})$

Abbreviations. In order to give the precise definition of $f_M(x)$, the following abbreviations will be useful.

- $\bigwedge_{k=1}^{m} X_k$ means $X_1 \wedge X_2 \wedge \cdots \wedge X_m$
- $\nabla(X_1, X_2, \ldots X_m)$ will denote a subformula that is satisfiable iff exactly one of $X_1, \ldots X_m$ is set to true. In other words,

$$\nabla(X_1, X_2, \ldots X_m) = (X_1 \vee X_2 \vee \cdots \vee X_m) \wedge \bigwedge_{k \neq l} (\neg X_k \vee \neg X_l)$$

Observe all the above are in CNF.

Overall Reduction. The overall form of $f_M(x)$ will be as follows.

$$f_M(x) = \varphi_{\text{initial}} \wedge \varphi_{\text{consistent}} \wedge \varphi_{\text{transition}} \wedge \varphi_{\text{accept}}$$

where

- φ_{initial} says that "configuration at time 0 is the initial configuration with input x"
- $\varphi_{\text{consistent}}$ says that "at each time, truth values to variables encode a valid configuration"
- $\varphi_{\text{transition}}$ says that "configuration at each time follows from the previous one by taking a transition"
- φ_{accept} says that "the last configuration is an accepting configuration"

We now outline what each of the above formulas is.

Initial Conditions Let $x = a_1 a_2 \cdots a_n$

$$\begin{array}{rcll}
\varphi_{\text{initial}} & = & \text{St@}(q_0, 0) & \text{"At time 0, state is } q_0\text{"} \\
& & \wedge \text{InpSymb@}(\triangleright, 0, 0) \bigwedge_{j=1}^{k} \text{TapeSymb@}(j, \triangleright, 0, 0) & \text{"Leftmost cells contain } \triangleright\text{"} \\
& & \bigwedge_{p=1}^{n} \text{InpSymb@}(a_p, p, 0) & (*\ \text{"At time 0, cells 1 through } n \text{ hold } x\text{"}\ *) \\
& & \bigwedge_{j=1}^{k} \bigwedge_{p=1}^{n^\ell} \text{TapeSymb@}(j, \sqcup, p, 0) & \text{"At time 0, all work-tapes are blank"} \\
& & \text{InpHd@}(0, 0) \wedge \bigwedge_{j=1}^{k} \text{TapeHd@}(j, 0, 0) & \text{"At time 0, all heads at the leftmost position"}
\end{array}$$

Consistency Assume that the tape alphabet is $\Gamma = \{b_1, b_2, \ldots b_t\}$.

$$\begin{array}{l}
\varphi_{\text{consistent}} = \\
\quad \bigwedge_{i=0}^{n^\ell} \nabla(\text{St@}(q_0, i), \ldots \text{St@}(q_m, i)) \text{"state is unique"} \\
\quad \bigwedge_{j=1}^{k} \bigwedge_{i=0}^{n^\ell} \text{TapeSymb@}(j, \triangleright, 0, i) \text{"leftmost cell contains } \triangleright\text{"} \\
\quad \bigwedge_{j=1}^{k} \bigwedge_{i=0}^{n^\ell} \bigwedge_{p=0}^{n^\ell} \nabla(\text{TapeSymb@}(j, b_1, p, i), \ldots \text{TapeSymb@}(j, b_t, p, i)) \text{"work-tape cell contain unique symbols"} \\
\quad \bigwedge_{i=0}^{n^\ell} \nabla(\text{InpHd@}(0, i), \ldots \text{InpHd@}(n, i)) \text{"unique position of input head"} \\
\quad \bigwedge_{j=0}^{k} \bigwedge_{i=0}^{n^\ell} \nabla(\text{TapeHd@}(j, 0, i), \ldots \text{TapeHd@}(j, n^\ell, i)) \text{"unique position of each work-tape head"}
\end{array}$$

Transition Consistency Let

$$\delta(q, c_{in}, c_1, c_2, \ldots c_k) = \{(q^{i_1}, d_{in}^{i_1}, c_1^{i_1}, d_1^{i_1}, \ldots c_k^{i_1}, d_k^{i_1}), \ldots (q^{i_s}, d_{in}^{i_s}, c_1^{i_s}, d_1^{i_s}, \ldots c_k^{i_s}, d_k^{i_s})\}.$$

We will first define the formula $\Delta_{q, c_{in}, c_1, \ldots c_k}^{i, p_{in}, p_1, \ldots p_k}$ says that at time i if the state is q and the symbols being read are $c_{in}, c_1, \ldots c_k$ at positions $p_{in}, p_1, \ldots p_k$, respectively, then at time $i+1$ the state, symbols written and new head position is one of the tuples described by the δ function.

$$\begin{array}{rl}
\Delta_{q, c_{in}, c_1, \ldots c_k}^{i, p_{in}, p_1, \ldots p_k} = & (\text{St@}(q, i) \wedge \text{InpHd@}(p_{in}, i) \wedge \text{InpSymb@}(c_{in}, p_{in}) \wedge \\
& \bigwedge_{j=1}^{k} (\text{TapeHd@}(j, p_j, i) \wedge \text{TapeSymb@}(j, c_j, p_j, i))) \rightarrow \\
& \nabla(\text{Ch}_{i, p_{in}, \ldots p_k}^{1}, \text{Ch}_{i, p_{in}, \ldots p_k}^{2}, \ldots \text{Ch}_{i, p_{in}, \ldots p_k}^{s})
\end{array}$$

where

$$\begin{array}{l}
\text{Ch}_{i, p_{in}, \ldots p_k}^{t} = \text{St@}(q^{i_t}, i+1) \wedge \text{InpHd@}(p_{in} + d_{in}^{i_t}, i+1) \wedge \\
\bigwedge_{j=1}^{k} \text{TapeSymb@}(j, c_j^{i_t}, p_j, i+1) \wedge \text{TapeHd@}(j, p_j + d_j^{i_t}, i+1)
\end{array}$$

Now Transition Consistency itself can be defined as follows.

$$\varphi_{\text{transition}} = \bigwedge_{i=0}^{n^\ell} \bigwedge_{p_{in}=0}^{n} \bigwedge_{p_1=0}^{n^\ell} \cdots \bigwedge_{p_k=0}^{n^\ell} \{$$

$$\bigwedge_{b \neq c} \bigwedge_{j=1}^{k} \neg \text{TapeHd@}(j, p_j, i) \rightarrow \neg(\text{TapeSymb@}(j, b, p_j, i) \wedge \text{TapeSymb@}(j, c, p_j, i+1))$$

"If head is not in some position, then symbol does not change"

$$\bigwedge_{q=q_0}^{q_n} \bigwedge_{c_{in}=b_1}^{b_t} \bigwedge_{c_1=b_1}^{b_t} \cdots \bigwedge_{c_k=b_k}^{b_t} \Delta_{q,c_{in},c_1,\ldots c_k}^{i,p_{in},p_1,\ldots p_k} \}$$

"If head is in some position, then a transition is taken"

Acceptance

$$\varphi_{\text{accept}} = \text{St@}(q_{\mathsf{acc}}, n^\ell)$$

We can argue that M accepts x if and only if $f_M(x)$ is satisfiable. Further $f_M(x)$ can be constructed in time that is polynomial in the size of x; the size of M also plays a role but that is fixed. □

From Example 18, Theorem 23, and the transitivity of reductions (Proposition 19), we can conclude that 3-SAT is also NP-hard. Since 3-SAT is a special form of SAT, membership in NP follows from the membership of SAT in NP. Therefore, we have

Corollary 24. *3-SAT is NP-complete.*

Let us consider the problem of determining if a propositional logic formula is a tautology. In other words,

$$\text{TAUT} = \{\langle\varphi\rangle \mid \varphi \text{ is a tautology}\}.$$

The Cook-Levin Theorem (Theorem 23) shows that TAUT is co-NP-complete.

Theorem 25. *TAUT is co-NP-complete.*

Proof. Observe that φ is a tautology if and only if $\neg\varphi$ is unsatisfiable. Thus, φ is *not* a tautology if and only $\neg\varphi$ is satisfiable. The NP algorithm in the proof of Theorem 23 works even if the input formula is not in CNF. Thus, $\overline{\text{TAUT}}$ is in NP. Therefore, by definition, TAUT $\in$ co-NP.

Let us take GENSAT $= \{\langle\varphi\rangle \mid \varphi \text{ is satisfiable}\}$. Since GENSAT is a generalization of SAT (which is the problem when the input formula is in CNF), we have SAT $\leq_{\text{P}}$ GENSAT. The NP-hardness of SAT and the transitivity of reductions (Proposition 19) allows us to conclude that GENSAT is NP-hard. The observations about the relationship between TAUT and GENSAT outlined in the previous paragraph, coupled with Proposition 19, allows us to conclude that TAUT is co-NP-complete. □

Observe that from Figure 1, we have P $\subseteq$ NP and P $\subseteq$ co-NP. From this we can conclude that P $\subseteq$ NP$\cap$co-NP. Related but independent of the P versus NP question is whether P $\stackrel{?}{=}$ NP$\cap$co-NP. This question also remains open. Many problems that were previously known to be in NP $\cap$ co-NP were proved to be in P years later. Two classical examples are Linear programming that was shown to be in P by Khachiyan in 1979 and testing whether a number is prime, which was proved by Agarwal-Kayal-Saxena in 2002 to be in P. However, there are some natural problems in NP $\cap$ co-NP whose status with respect to P is still unresolved. One is the problem of solving parity games, and the other is the factoring problem.