Complexity Classes

Katherine Taylor, Alexa Wingate

December 8, 2019

Abstract

This is a brief overview of computational complexity theory and some of the more well-known complexity classes. We define what we mean by problems and classes and explore the Turing machine model of computation. We explain the concepts of reduction, hardness/completeness, and diagonalization, and provide descriptions of lots of complexity classes.

1 Introduction: Why are all my classes so complex?

Definition 1.1 (Complexity Class)**.** *A complexity class is a set of functions with a common property.*

This common property is usually that they can all be computed within some time or space restraint. From here on, the restraints will typically be considered as a function of the input length.

Definition 1.2 (Decision Problem)**.** *A decision problem, or language, is a function that inputs a binary string of* 0*s and* 1*s and outputs either* 0 *or* 1*. It can be defined as the set of inputs for which it "accepts", or outputs* 1*.*

In other words, it's a problem with a yes-or-no answer. Lots of classes are defined with this sort of problem in mind, but there are exceptions (see More Classes).

2 Computational Models and the Turing Machine

Before we can start computing things, we need to discuss our computational model. This is something called a Turing machine, akin to a very basic computer. We present one version of a Turing Machine, but there are other versions (for example, we could use multiple tapes instead of one). We won't do so here, but it can be shown that such differences don't make a big difference in efficiency.

A Turing machine consists of a *tape* made of discrete cells. Each cell can be filled with one of the letters in a discrete alphabet Γ, for example, the set $\{0, 1\}$. There is also a *head*, which moves along the tape, editing the contents of the cells according to a fixed set of rules. The head is always in one of a finite number of *states*, Q.

A Turing Machine's head moves along in steps. At each step, the head:

1. reads the symbol contained in the cell currently under the head.
2. consults its *transition function* $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$. Given the head's current state and the symbol just read, the transition function tells the head (1) which state to switch into, (2) which symbol to write to the current cell, and (3) whether to move one cell to the left or one cell to the right.
3. follows the instructions of the transition function. If the state it has switched into is one of the set of *final states* $F \subset Q$, the Turing machine halts its computation.
4. Otherwise, it begins the cycle again in the cell it has just moved to.

Since there are different versions of Turing machines, there are different ways to manage "inputs" and "outputs". In our version, we'll make our input be the initial contents of the tape and our output be determined by which of the members of F we halted in. Most of the time, we'll be solving decision problems, so F will consist only of an "accept" state and a "reject" state (or equivalently, 1 and 0).

A Turing machine is completely determined by Γ, Q, and δ. The output of a Turing machine is completely determined by the input (the initial contents of the tape). As such, a Turing machine can be represented as a series of bits (a "string"). This string can be fed as the input to another Turing machine. The Turing machine that can take any other Turing machine's string representation as input and simulate the corresponding Turing machine is called the *Universal Turing Machine*.

While Turing machines can be implemented in real life, their use is mostly theoretical. The rationale behind using the Turing machine model is that, as well as having the same computational ability as a modern computer, it's simple enough that it can be used to rigorously prove stuff. Our belief in the usefulness of Turing machines can be encapsulated in the Church-Turing Thesis:

Thesis 2.1 (Church-Turing Thesis)**.** *Any computational algorithm can be implemented on a Turing machine.*

It's not completely settled whether this is an unproven hypothesis or if it really only serves as a definition for what we consider "computation".

Definition 2.1 (Turing Complete)**.** *A model of computation is Turing complete if it can simulate a Turing machine.*

Pretty much all modern computers and programming languages are Turing complete. Also, Turing machines are powerful enough to simulate real-life computers, and it seems that they do so without a big loss of efficiency. As a result of this, we strongly believe that all of our computational models are basically equivalent.

Notation/definition check before we move on: note that "problem", "language", and "function" are all essentially used interchangeably. Problems are distinct from Turing machines, which are used to implement algorithms for solving problems. Since Turing machines can be represented as strings, we denote by M_x the Turing machine whose string representation is x.

3 Some complexity classes

This is a quick list of the most familiar complexity classes. which are organized by how much time or space they take up as a function of input length. They all deal with decision problems. Before we start, there's an important distinction to make between "deterministic" and "nondeterministic" computation: a deterministic machine follows the same set of instructions each time to reach an answer, while a nondeterministic machine has to make arbitrary choices at certain points along the way. We judge nondeterministic machines on the optimistic assumption that they always make the right choice, which is why, for example, "solvable by a nondeterministic machine in polynomial time" and "can be verified by a deterministic machine in polynomial time" are equivalent: if we can solve a problem with a nondeterministic machine in polynomial time, then it will only take polynomial time for a deterministic machine can check that the path to the solution the nondeterministic machine came up with is valid. Also, if we can verify a solution in polynomial time, then a nondeterministic machine could correctly guess each step of the solution as we verify it in order to solve the problem in nondeterministic polynomial time. Now let's introduce some classes.

Definition 3.1 ($DTIME$)**.** *Solvable in a finite amount of time for every input (i.e., the amount of time to solve is some function of input length).*

Definition 3.2 (P)**.** *The amount of time to solve is (at most) some polynomial function of the input length for every input.*

The class P is generally considered to be the class of "easy-to-solve" problems.

Definition 3.3 ($EXPTIME$)**.** *The amount of time to solve is (at most) some exponential function of the input length for every input.*

Classes that start with the letter N are for Nondeterministic machines. Remember that solving by a nondeterministic machine in x amount of time is equivalent to checking by a deterministic machine in x amount of time.

Definition 3.4 ($NTIME$)**.** *Solvable by a nondeterministic machine in a finite amount of time for every input.*

Definition 3.5 (NP)**.** *The amount of time to solve for a nondeterministic machine is (at most) some polynomial function of the input length for every input.*

Another way to think about the class NP is that it can be *verified* in polynomial time. For example, say you were given a list of arbitrary numbers $\{-5, -3, -2, 1, 6\}$ and you wanted to check if there exists a subset of these numbers that add to 0; in this instance, $-5-2+1+6=0$. You could design an algorithm to figure it out by generating all subsets of the set and test each one to see if their sum is 0. The time of this method would increase exponentially depending on the size of the set. However, given one possible subset (known as a *witness*), you can check its validity within polynomial time. Therefore, this problem is in NP.

Definition 3.6 ($NEXPTIME$)**.** *The amount of time to solve for a nondeterministic machine is (at most) some exponential function of the input length for every input.*

Classes can also be defined by how much space is needed to solve a problem, though time-based complexity is more important. Here are a few space classes anyway, starting with the ones for deterministic machines:

Definition 3.7 ($DSPACE$)**.** *Solvable in a finite amount of space for every input.*

Definition 3.8 (L)**.** *The amount of space to solve is some logarithmic function of the input length for every input.*

L is interesting because it is contained in P. In $O(\log n)$ space, there are only $2^{O(\log n)} = O(n)$ possible states, so if our algorithm takes more than $O(n)$ time it repeats itself.

Definition 3.9 ($PSPACE$)**.** *The amount of space to solve is some polynomial function of input length for every input.*

Definition 3.10 ($EXPSPACE$)**.** *The amount of space to solve is some exponential function of input length for every input.*

Definition 3.11 ($NSPACE$)**.** *Solvable by a nondeterministic machine in finite space for every input.*

Definition 3.12 (NL)**.** *The amount of space to solve for a nondeterministic machine is some logarithmic function of input length for every input.*

Definition 3.13 ($NPSPACE$)**.** *The amount of space to solve for a nondeterministic machine is some polynomial function of input length for every input.*

Definition 3.14 ($NEXPSPACE$)**.** *The amount of space to solve for a nondeterministic machine is some exponential function of input length for every input.*

4 Diagonalization and Reduction

The notion of "reduction" is an important one. It means to show that one problem is a transformation of another. The two main versions of reduction are (1) a bijection between instances/solutions of problem A and instances/solutions of problem B (many-one reduction), and (2) an algorithm for solving A, given an oracle for B (Turing reduction).

Definition 4.1 (Oracle)**.** *An oracle for a problem A will, given any input x, correctly and quickly return A's output on x.*

One subtlety of reduction is that we need to keep track of how resources are taken up by the reductions themselves. For example, reducing one problem in P to another problem in P by means of an exponential-time reduction algorithm is not especially useful. For this reason, reductions within the time and space classes already mentioned are usually restricted to polynomial time and/or logarithmic space. Reduction is reflexive and transitive, but not symmetric. When problem B can be reduced to problem A, we know that A takes at least as much resources to solve as B does.

We will now deal with the use of reduction as a proof technique. It works like this: to show that problem A is impossible, we show that if we can solve problem A, we can also solve some problem that we already know is impossible, problem B. In the language of reductions, we show that some impossible problem B is reducible to A, which implies that A itself is impossible.

But before we get to reduction, we must introduce another very important proof technique, diagonalization.

Definition 4.2 (Computable)**.** *A function is computable if there exists an algorithm that can compute the output to the function for every possible input.*

If a function isn't computable, that means that any algorithm that attempts to solve it must either sometimes be wrong or loop forever on certain inputs.

Theorem 4.1. *There exists some decision problem that is not computable by any Turing Machine.*

Proof. For any Turing Machine M_α, denote by α the string representation of M_α. Here's a function f:

$$f(x) = \begin{cases} 0 & \text{if } M_x(x) = 1 \\ 1 & \text{otherwise} \end{cases}$$

Suppose there's some Turing machine M_β that computes f. Then what is $M_\beta(\beta)$? Suppose $M_\beta(\beta) = 1$. Then, looking at the definition of f, we must have that $f(\beta) = 0$. Since M_β computes f, we must have that $M_\beta(\beta) = f(\beta) = 0$. We have a contradiction - $M_\beta(\beta)$ cannot be 1. Okay, let's try $M_\beta(\beta) = 0$. This corresponds to "otherwise" in the definition of f, so $f(\beta) = 1$. Since M_β computes f, we must have that $M_\beta(\beta) = f(\beta) = 1$. We have a contradiction - $M_\beta(\beta)$ cannot be 0. There's nothing that $M_\beta(\beta)$ can be, so our only way out is to conclude that M_β, a Turing machine to compute f, does not exist. We have found an uncomputable function. □

The part of this proof that made it diagonalization was constructing f so that $f(x)$ was never equal to $M_x(x)$. More generally, diagonalization is about taking a set of things and then constructing a new thing that is different from everything else in the original set. Perhaps this doesn't seem especially "diagonal", but the name comes from other proofs (for instance, showing that the set $\mathbb{R}$ is uncountably infinite) where we can represent the proof using a diagram/list with a diagonal line through it.

We can use f to prove something else interesting using the reduction technique: that some problems are unsolvable by Turing machines.

Definition 4.3 (Halting Problem)**.** *The halting problems asks if an arbitrary computer program (or Turing machine) halts, i.e. whether or not it ever finishes running.*

In 1936, Alan Turing proved that the halting problem is *undecidable*, or that cannot be solved for one general algorithm. We will do a rigorous proof using the function f we introduced, but first, we present the concept of a more intuitive proof by contradiction:

Proof. Suppose there exists a total computable function halts(f) that returns true or false depending on whether f halts for some input. Then, let us define the pseudocode for a function g as follows:

```
def g() :
  if halts(g) :
    loop_forever()
```

If if halts(g) returns false, the function will end, thus creating a contradiction. However, if if halts(g) is true, loop_forever will be called and we have another contradiction. Therefore, the original assumption that halts(f) is a total computable function must be wrong. □

Now for a less intuitive but more rigorous proof. Define the function H (for Halting Problem) as follows: For every Turing machine M and input x, let $H(M, x) = 1$ if M eventually halts on input x, and $H(M, x) = 0$ if M loops forever on input x.

Proof. Suppose there's some Turing machine M_α that computes H. Since Turing machines can simulate each other, we can make another Turing machine M_z to simulate M_α and then do some additional stuff. Note that since $H(x)$ takes a pair (M, x) as input, so does M_α. M_z does the following on an input x:

1. Constructs M_x from x, or does whatever it needs to in order to simulate M_x later. (If you find this problematic, remember that since the Universal Turing Machine exists, there's an M_z that can always do this.)
2. simulates M_α running on input (M_x, x).
3. if $M_\alpha(M_x, x) = 0$, then we know M_x loops forever on input x. We return 1.
4. if $M_\alpha(M_x, x) = 1$, then simulate M_x running on input x.
5. If $M_x(x) = 0$, return 1. If $M_x(x) = 1$, return 0.

But wait, look closely. If M_x loops forever on x, we return 1. If $M_x(x) = 0$, we return 1. If $M_x(x) = 1$, we return 0. Sound familiar? Go back and look at the definition of $f(x)$ from before. $M_z(x)$ will always be the same as $f(x)$, so M_z computes f. If we can compute H, then we can use that to compute f. In other words, f reduces to H. Since f is uncomputable, we are forced to conclude that H is uncomputable too. □

Hopefully these proofs have demonstrated the power of diagonalization and reduction and helped us get an appreciation for how useful the Turing machine model can be.

5 I Feel So Complete

Two important concepts that rely on reduction are hardness and completeness.

Definition 5.1 (Hardness)**.** *Problem A is at least as hard as problem B if problem B can be reduced to problem A. A problem is hard for a class if all other problems in the class can be reduced to it.*

Also, sometimes "A is harder than B" is taken in the usual sense of "hard", i.e., meaning only that A requires more resources to compute.

Definition 5.2 (Completeness)**.** *A problem is complete for a class if it is hard for the class and the problem is in the class.*

The notion of completeness/hardness gives us an easy way to talk about all the problems in a class. For instance, a quick solution to a hard problem implies a quick solution to every problem in the class it is hard for. Thus, the existence of a polynomial-time algorithm for any problem that is NP-hard implies the existence of a polynomial-time algorithm for every problem in NP. In other words, if we wish to prove that $P = NP$, a good way of doing so is to pick an NP-hard problem and prove that that problem is actually in P. Note that it is widely believed that $P \neq NP$, so if we try this we are unlikely to be successful.

It's not intuitively clear that any problem can be complete. But in fact there are lots of them, including the most well-known group of them, the NP-complete problems. Here's one such problem $TMSAT$: given a Turing machine M and a time limit t, return 1 if there's an input y such that M returns 1 on input y within t steps.

Theorem 5.1. *$TMSAT$ is NP-complete.*

Proof. To be NP-complete, $TMSAT$ must be NP-hard and in NP. First observe that $TMSAT$ is checkable in polynomial time: if we claim that y is a solution, we can simply simulate M running on input y for a maximum of t steps. If M returns 1 during that time, then y really is a solution. Furthermore, To show that $TMSAT$ is NP-hard, let's first take any other problem $A \in NP$. Our goal is to show that A can be reduced to $TMSAT$, or that given an algorithm for solving $TMSAT$, we can solve A. Since A is in NP, A can be verified in polynomial time. Thus, there's a Turing Machine M_α that returns 1 on an input (x, y) if and only if A outputs y on input x, and does so in polynomial time.

Assuming we have an algorithm for solving $TMSAT$, we can hijack it to solve A instead. First, we make a version of M_α with the x part of the input built-in, call it $M_{\alpha,x}$. ($M_\alpha(x, y)$ and $M_{\alpha,x}(y)$ have the same output.) Our input to the $TMSAT$-solving algorithm is the Turing machine $M_{\alpha,x}$ and a time limit t for which $M_{\alpha,x}$ is guaranteed to finish. (Given the length of the input x, we can find such a t, since $M_{\alpha,x}$ runs in polynomial time.) If the $TMSAT$-solving algorithm works, it will output y such that $M_{\alpha,x}(y) = 1$. Using this method, we can always find y given x, which means we can always compute A. Since A could have been any problem in NP, every problem in NP reduces to $TMSAT$. □

Another notable NP-complete problem is $3SAT$, which asks, given a formula involving Boolean (true/false) variables and Boolean operators (and, or, not), can we assign values to those variables in such a way that the entire expression evaluates to true? $3SAT$ is a special case of the problem SAT, which deals with Boolean formulae in Conjunctive Normal Form (CNF). A formula in CNF is of the form

$$C_1 \wedge C_2 \wedge \cdots \wedge C_n$$

Each C_i is called a *clause* and is of the form $(l_1 \vee l_2 \vee \cdots \vee l_m)$, where $\wedge$ means and, and $\vee$ means or. The l_is are literals, which take the value of either a variable or its negation from the set of variables $\{x_1, \ldots x_k\}$. Here's an example of a Boolean formula in $3CNF$ form (3 literals per clause):

$$(x_1 \vee x_2 \vee x_3) \wedge (-x_4 \vee -x_3 \vee x_2) \wedge (x_4 \vee -x_1 \vee x_5) \wedge (x_3 \vee x_5 \vee x_2)$$

If the variables $\{x_1, \ldots x_k\}$ in a CNF formula can be assigned values so that the entire formula evaluates to true, we call that formula satisfiable. SAT asks whether a given CNF formula is satisfiable, and $3SAT$ asks whether a given 3CNF formula is satisfiable. Both problems are in NP, since we can check if a set of values is a solution by plugging in the set of values for the variables $\{x_1, \ldots x_k\}$. We'll show in a minute that SAT actually reduces to $3SAT$. The proof that $3SAT$ is NP-complete is a little more involved than either that proof or the proof that $TMSAT$ is NP-complete. It relies on showing that any NP problem can be expressed as a Boolean formula in 3CNF form.

Theorem 5.2. *Any SAT (CNF form) formula can be reduced to 3SAT (3CNF form).*

Proof. We break this proof into four cases to turn an arbitrary clause C into an equivalent set of clauses, each of which has 3 literals. This enables us to convert a CNF formula, where clauses can have any number of literals, into a 3CNF formula, where clauses are only allowed to have 3 literals.

Case 1: C has one literal. Define new variables z_1 and z_2 and replace C by the four clauses $(l \vee z_1 \vee z_2)$, $(l \vee \overline{z_1} \vee z_2)$, $(l \vee z_1 \vee \overline{z_2})$, and $(l \vee \overline{z_1} \vee \overline{z_2})$. Since adding $\wedge$s between these still makes $C = l$, C is now $3SAT$.

Case 2: C has two literals so that $C = l_1 \vee l_2$. This time, define z_1 and replace C with the clauses $(l_1 \vee l_2 \vee z_1)$ and $(l_1 \vee l_2 \vee \overline{z_1})$. Thus C is $3SAT$.

Case 3: In this case, C already has 3 literals, so it is already $3SAT$.

Case 4: C has more than 3 literals, so let $k > 3$ as $C = l_1 \vee l_2 \vee l_3 \vee \cdots \vee l_k$. Next, replace C with the $k - 3$ clauses: $(l_1 \vee l_2 \vee z_1), (l_3 \vee \overline{z_1} \vee z_2), (l_4 \vee \overline{z_2} \vee z_3), \ldots, (l_{k-2} \vee \overline{z_{k-4}} \vee z_{k-3}), (l_{k-1} \vee l_k \vee \overline{z_{k-3}})$. Notice that unlike the prior cases, this doesn't just simplify like before. However, it can be verified that

1. if there used to exist a way of assigning $x_1, \ldots, x_k$ so C is true, there is also a way of assigning $z_1, \ldots, z_{k-3}$ so that the new C is true.
2. if $x_1, \ldots, x_k$ make C false, it is impossible to assign $z_1, \ldots, z_{k-3}$ to make the new C true.

□

6 More classes

There are more classes than just the regular time and space ones!

Definition 6.1 (co-NP)**.** *A problem x is in the class co-NP if its complement, $\overline{x}$, is in NP. This means co-NP contains problems that can be verified "no" in polynomial time.*

One particular example of a co-NP problem is co-SAT. The regular SAT (boolean satisfiability) problem is to determine whether a boolean expression, say "A or not B", can be made true with A and B. In this case, A = true and B = false works. Therefore, co-SAT asks if a boolean expression is not satisfiable. Keep in mind that showing that simply one example that doesn't work doesn't necessarily put the entire problem in co-NP.

Definition 6.2 (RP)**.** *RP, meaning randomized complexity time, is a class containing the problems for which there exists a probabilistic Turing machine such that:*

1. *It runs in polynomial time*
2. *If the answer is "no", it always correctly returns "no"*
3. *If the answer is "yes", it correctly returns this answer at least 50% of the time*

Definition 6.3 ($\#P$)**.** *Problems in $\#P$ calculate the number of solution paths of an NP problem. They are not decision problems.*

Definition 6.4 (PP)**.** *PP is a large class of problems, containing other classes such as NP, BPP, and BQP. Problems in PP are solvable by a probabilistic Turing machine in polynomial time with less than 50% error.*

One interesting proof is showing that NP is in PP. Because of the theory of NP-completeness, all we have to do is show that SAT is in PP. Let F be the Boolean formula in question. Next, check if a random assignment of variables satisfies F. If so, the algorithm outputs 1, but if not, performs a coin flip. If $F \in SAT$ (meaning it is satisfiable), then the overall probability of our algorithm correctly outputting 1 is at least $1/2 + 2^{-m}$ (where m is the number of arguments in F, so we made m random decisions in the first step). If $F \notin SAT$, then the probability of the algorithm correctly answering -1 is at least $1/2$. Thus, SAT is in PP, which generalizes to show that the class NP is in PP.

Definition 6.5 (BPP)**.** *BPP stands for bounded-error probabilistic polynomial time, and is a more efficient subset of the PP complexity class. A problem belongs in BPP if there is a probabilistic Turing machine that runs in polynomial time. It returns the correct answer with at least $2/3$ probability (or the wrong answer with at most $1/3$ chance).*

A subtle difference between RP and BPP that is important to point out is that *if* the program outputs 'yes', it is always right. Although not separately defined in this paper, we also know that co-RP means the opposite: that when the answer 'no' is outputted, that answer is correct. Therefore, one wonders whether it is possible to have a problem that, when probabilistically solved, has some error for both 'yes' and 'no' answers.

There's not an easy answer to this question, but one possible example of a problem that is in BPP but not RP or co-RP is something along the lines of telling whether a number is a Perfect Number. A perfect number is a positive integer for which the sum of its proper divisors is equal to itself. Other people speculate problems like certain factorization algorithms or using linear programming to approximate the volume of a convex body, while some people believe that there are very few, if any, problems left in BPP but not P.

Definition 6.6 (ZPP)**.** *ZPP stands for zero-error probabilistic polynomial time, meaning that a problem exists in this class if there exists a probabilistic Turing machine such that it returns "yes" or "no" answers with perfect accuracy, and that this answer can always be found in polynomial time.*

Definition 6.7 (BQP)**.** *Explained informally, BQP is essentially the same thing as BPP except including quantum Turing machines. Meaning, there is a polynomial time quantum algorithm that has a maximum $1/3$ chance of error.*

As you may have heard, quantum algorithms are a lot more powerful than classical algorithms, enabling certain problems in mathematics to be computed in a much shorter amount of time. For example, Shor's Algorithm factors an integer n in polynomial time in terms of $\log n$ (the size of n), Specifically in $O((\log n)^2(\log\log n)(\log\log\log n))$. This is substantially faster than the general number field sieve, which is currently the fastest factoring algorithm we have and runs in sub-exponential time.

The first part of the algorithm, using regular classical computing, reduces the factoring problem into an order-finding problem. Then, the magic to Shor's Algorithm comes from the efficiency of the quantum Fourier transform. It's too complicated to explain with the scope of this paper, but the Fourier transform maps one quantum state to another.