A Visual Companion to Complexity Theory Basics

Casey Pancoast*

December 2020

Abstract

A companion for the visually inclined as they learn the very basics of computational complexity theory, written by a particularly visual undergraduate. This humble guide, replete with footnotes, is intended as a supplementary perspective to learning from a more established (and rigorous) source. Topics covered include problems, instances, **P**, **NP**, hardness and completeness, **PSPACE**, **EXPTIME**, and the duality between a class of problems and a computer capable of solving them.

1 Introduction

Computational complexity is the study of how difficult problems are. In the same way that physics is not about building better rockets, but understanding the natural principles behind rocket construction, [1] computational complexity is not about building better algorithms. Rather, it's about studying problems themselves as separate, natural entities. [8] Broadly speaking, we assign problems to classes denoted by inscrutable series of capital letters [2] (**P**, **NP**, **PSPACE**, **EXPTIME**, **BQP**) depending on how quickly the problem becomes stupidly difficult to solve as you try more and more complicated versions of it. That difficulty is given in terms of the amount of some resource that some computer has access to that it has to use in solving this problem. There are many different kinds of resources (like number of queries to another more powerful computer, or the depth of a boolean gate circuit) but here we'll just think about time and space.

Before long, a formal definition of a problem will be given, along with definitions of all of the complexity classes given above and what a complexity class is in the first place. Several nice visualizations are given for problems and instances in different complexity classes. This guide also covers problems that are "complete" or "hard" for their class, and what that means. To finish off, it presents how the topics of this guide fit in with the big picture of complexity

*Candidate for B.S. in Computer Science and Physics at Northeastern University.

[1] Among other things, I'm told.

theory and provides the reader (you!) with a manageable amount of references for their (your) trouble.

2 Problems and Complexity Classes

2.1 The class P

Let's look at a simple example problem: "Given a number, is that number prime?"

This problem, which we'll give the name PRIME[2], becomes only a *little* more difficult as the value of the number to be decided increases. That is, determining whether 1736598123657 is prime is certainly difficult, but it doesn't take much longer than determining whether 1736598123653 is prime.

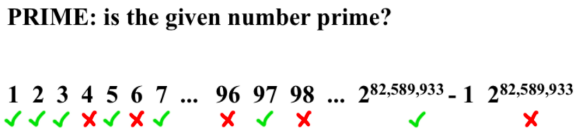

Figure 1: The problem of determining whether a given number is prime.

Problems like this we say are in **P**, which stands for polynomial time — formally, if a problem is in the class **P**, solving the problem takes polynomially more time as the input increases. Here's a formal definition of **P**:

$$\mathbf{P} = \bigcup_k \mathbf{TIME}(n^k)$$

TIME$(f(n))$ is just the set of the problems where, as n gets bigger, solving them takes more time proportionally to the way $f(n)$ gets bigger.

You might ask, "time for *what computer?*" The short answer is a *deterministic Turing machine*, and the longer answer is that it doesn't matter, as long as the machine is deterministic and has no "special powers".[3] We will introduce another kind of computer later which does.

2.2 A necessary infinity

I propose to you that solving chess is easy. Why? Well, it'll take a while, but eventually you can figure out what moves you need to take at every given position. Because there's only a finite number of positions, you'll figure it all out sometime!

[2]The convention in computational complexity theory is to write problems in small caps.

[3]If you're interested in understanding this on a deeper level, [8] is an excellent place to start, but otherwise assume the computer we're talking about is your laptop, or, just as good, the aging PC at your local library.

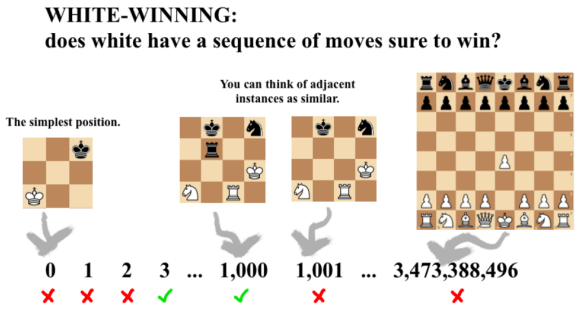

Figure 2: A mapping of instances of the problem WHITE-WINNING to the natural numbers.

In other words, there exists some program, even if it's really hard for you to write, that can solve any 8 x 8 chess position almost instantly. This certainly doesn't tell us anything deep about chess!

We'll now introduce some jargon, to make things more complicated in such a way that we can understand them better.[4]

We say that a problem has some number of *instances*. In the case of PRIME, each instance is a number, and there are an infinite amount of them. In chess, our problem might be called WHITE-WINNING, which asks "given this initial position, does white have a *sure-fire* way to win the game?" An *instance* of WHITE-WINNING would be an arrangement of pieces on the board.

If we want to know something deep about how hard chess is, we must generalize our chessboard from 8 squares by 8 to any number n squares by n. This allows us to learn more about chess in the 8 by 8 case by learning how much harder chess gets when you make it bigger. Our computer is forced to do away with its list of solutions — now it must have an *algorithm*.[5]

WHITE-WINNING is not in **P** — we'll see what it is in later on.

2.3 Some formal definitions

Let's get concrete.

[4] Welcome to research.

[5] Someone (perhaps just the collective unconscious of computer scientists) once said that "an algorithm is a finite solution to an infinite problem".

A **problem** is any mapping we can make from the natural numbers to a boolean (true or false, yes or no).[a]

[a]Actually, a problem can be a mapping of the natural numbers to *anything*. Most of the time, complexity theorists look at *decision problems*, which take in a problem instance and determine whether it has some property. (That is, it maps to a boolean.) We'll only consider these in what follows, but keep this in mind if you want to learn more.

To talk about a problem (informal) in terms of complexity theory, we turn it into a problem (formal, given above) by first mapping instances of the problem to natural numbers, and then mapping those natural numbers to booleans.

A **problem instance** is one configuration of the problem (informal) that we are trying to solve that we have mapped to one of these natural numbers.

Above, an instance of PRIME was a number, while an instance of WHITE-WINNING was a board position.

Most of the time, it doesn't matter *how* you map from instances to natural numbers — it just matters that you can in such a way that as the natural numbers increase, the instances generally get harder.[6]

We're also ready for another one, straight from Wikipedia:

A **complexity class** is a set of problems of related resource-based complexity. [9]

We've already seen a couple complexity classes, and the resources they keep track of — both **P** and **TIME** use time. Well see some classes that use other resources later.

Now for the relation between problems, classes, and algorithms — if you've ever taken an algorithms course (and particularly if you haven't), you could *easily* design an algorithm for PRIME that takes more than polynomial time. For this reason,

A problem has membership in a complexity class depending on the resource usage of its *best possible algorithm*.

There are subtleties here involving some problems having an infinite amount of algorithms that approach some limit in terms of their resource usage, but we'll let such special cases escape our attention.

[6]For WHITE-WINNING, we might naturally map larger board sizes to larger values of natural numbers.

3 Solitaire and the class NP

Let's take another look at the class **P**. Remember PRIME? Well, one way of drawing the instances is as is shown in figure 3, where the length of the lines indicate the approximate length of the computation (of the best possible algorithm) to figure out whether the number is prime.[7]

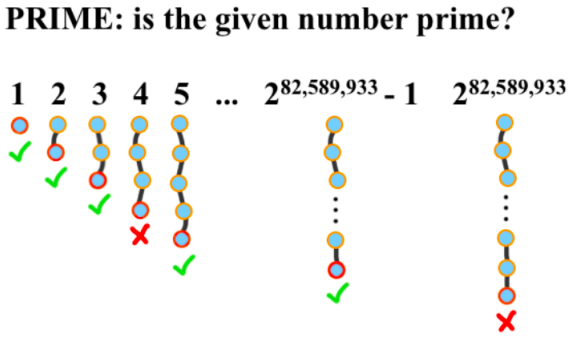

Figure 3: The amount of time it takes for a computer to solve each instance of PRIME, represented as a chain of computational steps.

3.1 Computers with special powers

I expect you've played solitaire (or some variant) before. Perhaps when you were trying to organize a deck of cards, or perhaps you simply don't value your time. If you don't know what solitaire is or how it is played, what matters for this guide is that it is a game where you try to turn all the cards face-up, but you can only do so for certain cards at a time, and depending on the way the deck is shuffled solitaire can actually be impossible.[8]

Let's name the problem of whether or not an initial condition of solitaire is solvable SOLITAIRE-SOLVE. The initial condition will be our *problem instance*. There's lots of ways to generalize solitaire to have an infinite number of instances — let's say we let the number of suits increase past four, or maybe we let the number of ranks increase past 13. (2 through 10, Jack, Queen, King, Ace.) To check whether a game is solvable, we can't just do something simple like repeated division attempts, as in PRIME — we have to play through each game! The computational path might look like figure 4.[9]

[7] One way is to try to divide by every number less than it, which will take time proportional to the size of the number. Evidently, this grows polynomially in the size of the number.

[8] One use of solitaire is to teach life lessons.

[9] **Exercise 1 (and only):** Imagine, with your own two brain lobes, the instances of the solitaire games that are being mapped to each natural number. What might it look like when the number of cards changes? The ability to visualize things like this will be crucial for your success in understanding complexity theory basics.

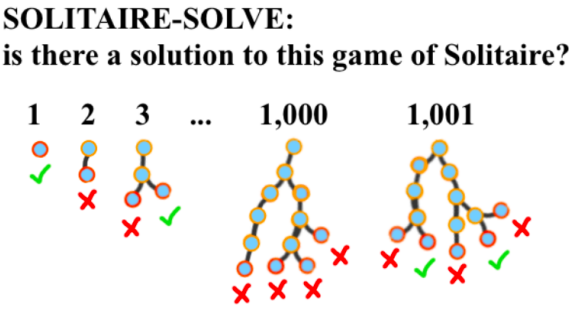

Figure 4: The computational steps a computer might have to take to solve a game of solitaire.

Solitaire-Solve asks, "does at least one of the ways of playing the game, starting from this instance, end in a solved game?" For instance 1437 in figure 4, this is true — for 1438, on the other hand, there is *no way* to play the game or make choices such that you win.

Now we can look at the formal definition of **NP**:

$$\mathbf{NP} = \bigcup_k \mathbf{NTIME}(n^k)$$

Remember $\mathbf{TIME}(f(n))$? Well, that's all of the problems whose problems get $f(n)$ harder for a deterministic computer to do as n increases. You can think of a deterministic computer as a computer that can only follow one path — just like the computer that you're reading this on. $\mathbf{NTIME}(f(n))$ is a class for computers with the godlike power to go down multiple paths at once, and then bring all the information back together at the end! Although these computers don't exist[10] we find it useful to study the problems that they might be able to solve.

However — and this is a critical point — while a deterministic computer can only solve Solitaire-Solve in exponential time (the number of nodes in the computational graph increases exponentially as the problem gets larger), it can *check solutions* to solitaire games in polynomial time. Why? Well, you can imagine that it simply starts at the end of the game (end of the path) and then works its way backwards, making sure that no rules were violated at each step.

Let's give this its own box:

> A deterministic computer can solve **NP** problems in exponential time, but it can check that the problems are correct in polynomial time.

[10] This is *not* what a quantum computer is — that's a whole 'nother story involving a class called **BQP**.

Anyway, this means that **NP** is the class of all problems that our godlike computer could solve in polynomial time, as it can "split". So, the time it takes to solve this problem is given by the *height* of the tree, not by each branch added together. Our regular, deterministic computers *can't* split, and so problems that are in **NP** (that aren't also in **P**) take more than polynomial time.

3.2 Relations between complexity classes

Note that our godlike computer could also solve all the problems in **P** easily (in polynomial time) as well. This means that any problem that's in **P** is also in **NP**. Or, more formally, $\mathbf{P} \subseteq \mathbf{NP}$.[11] You might have seen pictures which look like figure 5, and it's this concept that they are referring to.

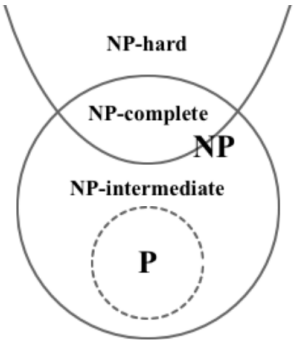

Figure 5: Relations between the sets of problems **P**, **NP**, **NP**-complete, and **NP**-hard.

So, when we want to talk about the hardness of a problem in relation to **NP**, how do we do that without leaving open the possibility that it's just part of the simpler class **P**?

There's two common bits of jargon for making these kind of distinctions.

> A problem is **NP-complete** if the problem is as hard as the hardest problem in **NP**.[a]
>
> [a] **NP-intermediate** contains those humdrum problems in **NP** which are not the hardest problems, but also are not part of a simpler class.

Solitaire-Solve, for example, is **NP**-complete.[12]

[11] But does $\mathbf{P} = \mathbf{NP}$? The dotted line in figure 5 indicates this uncertainty, despite the intuition I have just given you implying that it is certainly not. Some would argue [2] that this is the largest unsolved problem in all mathematics.

[12] Intuition will be provided in this guide for why this is true. [6] is a great place to go if you want to understand this more rigorously.

A problem is **NP-hard** if the problem is *at least* as hard as the hardest problem in **NP**.

And now a natural question might arise — what's harder than **NP**?

We'll get there later, but here's one example. **NP** problems can be solved by a deterministic computer in exponential time and checked in polynomial time. One might imagine a problem for which even *checking* a solution to an instance took exponential time!

I now assert that *almost all puzzles are **NP**-complete* — **NP** contains things that are easy (polynomial time) to check the answer to, but hard (exponential time, as you have to check every path) to *find* the answer to, and that's pretty much what a puzzle *is*, right? [6]

4 Chess, PSPACE, and EXPTIME

Now for chess. Let's go back to the beginning, when we were introducing problems: our chess problem is named WHITE-WINNING, which says, "given such-and-such position, does white have a winning move"? As with the previous games, we must somehow generalize to an infinite amount of positions so we can't just make a big list. We do this, again, by allowing our chessboard to be any size n by n.

The puzzle we considered earlier, solitaire, was **NP**-complete, because puzzles take the form "make a move that solves this puzzle". In chess, you're no loner alone with your problem — you're now playing against an adversary[13]!

4.1 PSPACE

Chess is "**PSPACE**-complete", because the hardest problems in **PSPACE** are problems that take the form "make a move such that for every move your opponent makes, you can make a move such that every move your opponent makes..." It's going to take exponential time to go down every one of those paths, but it will only take polynomial space to keep track of where you are in the path.

For completeness's sake, here's the formal definition of **PSPACE**. (See if you can suss out the finer points, like what $\mathbf{SPACE}(f(n))$ is, from the definitions of **P** and **NP** given earlier.)

$$\mathbf{PSPACE} = \bigcup_k \mathbf{SPACE}(n^k)$$

[13] If you have some theoretical computer science experience, you would know that an adversary is an entity that tries to make your code fail, no matter what. If you happen to have some experience in theology, note that this adversary is distinct from the Adversary, however hellish it may feel when your code won't run.

In the same way that SOLITAIRE-SOLVE was **NP**-complete, WHITE-WINNING is **PSPACE**-complete — it's as hard as the hardest problem in **PSPACE**.[14]

Now, let's look at WHITE-WINNING with fresh eyes, and draw it in the same "computational path" way that we drew SOLITAIRE-SOLVE (although in a simplified manner to save space). For simplicity's sake, we'll just look at one instance in figure 6.

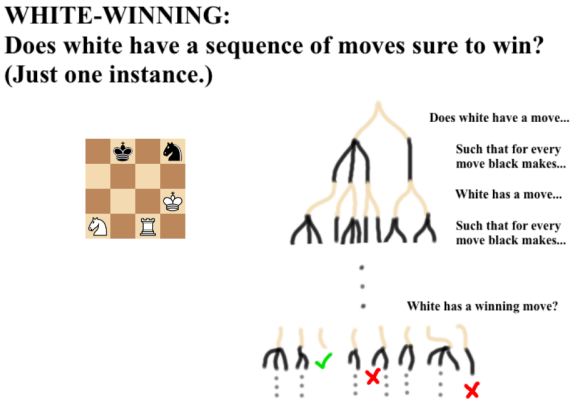

Figure 6: The computational path a computer might take to solve a game of chess. For this instance to return `true`, there could be no tree of black moves that only ended in an `false` (`X`) — failure for white.

You can see two things when the computational paths of WHITE-WINNING are drawn — first, you can see the "for every move my opponent makes, do I have a move..." recursive structure. Second, you can see that checking a *solution* indeed takes polynomial time — you simply follow the path back up![15]

4.2 EXPTIME

However, chess being in **PSPACE** assumes a rule in chess called the "fifty-move drawing rule". WHITE-WINNING is in **PSPACE** if there's a bottom to these paths, but if states are allowed to loop back on themselves, things can go on indefinitely, so we have to keep track of *all the past states of the path* to

[14] When talking about complexity classes, researchers will often pick a problem that is complete for that complexity class and designate it as the "canonical problem". Something not mentioned in this short introduction is the idea of *reduction*, that one can turn one problem into another! [8] is a great place to go for that one, as is [2] if you're short on libraries or cash.

[15] One might wonder — if solving a game of chess is so computationally intensive, how do computers do it regularly? The answer is that computers *don't* solve chess — rather, they produce approximate solutions. Most chess computer algorithms make use of machine learning, which, to gloss over a billion details, produces approximate solutions. Although it has nothing to do with complexity, I would highly recommend starting here [1] if you're interested in a visual introduction to machine learning.

make sure we're not repeating ourselves! [6] If you're looking for a visual here (albeit a very informal one), imagine the tree from figure 6, but the branches get thicker and thicker the further down the tree (the further into the game) you go. WHITE-WINNING still takes exponential time, but now it will take more than polynomial space to keep track of where we are.

WHITE-WINNING without this drawing rule is **EXPTIME**-complete. Here's the definition:

$$\mathbf{EXPTIME} = \bigcup_k \mathbf{TIME}(2^{n^k})$$

If a problem is **EXPTIME**-complete, it means that the time it takes for your laptop to solve an instance of the problem gets exponentially harder as the size of the problem increases.[16] This is *not* what you want to turn in in an algorithms course!

5 Discussion

5.1 Summary

We've come a long way. What have we done?

- We introduced (decision) *problems* as, rather unintuitively, mappings from *instances*, which are cases of the thing we're interested in solving, to natural numbers, and then to booleans. We then clarified that these problems are only interesting if they have an infinite amount of instances, as otherwise you can just store a big list.
- We introduced the complexity class **P**, which contains the kind of problems that take your laptop a polynomial amount of time longer to solve as the input gets bigger.
- We introduced the complexity class **NP**, which contains the kind of problems that take your laptop an exponential amount of time longer to *solve* as the input gets bigger, but only a polynomial amount of time longer to *verify answers to*. An **NP**-complete problem was presented, called SOLITAIRE-SOLVE, which asks "given this initial configuration of solitaire, is it possible to win the game?"
- We introduced the idea of a different type of computer, the godlike **NP** computer, which has the power to split itself and do multiple computations at once.[17]

[16] Mathematically, our choice of 2 to be the base for the exponent is for no other reason than it is the simplest number bigger than one. Allowing the base to be raised to a polynomial of any degree in the number n is sufficient to cover any change of base formula you might desire. Historically, computer science has shown great preference for the number 2 — if computational complexity was more the domain of physicists or pure mathematicians, I'm sure there would be an e right there.

[17] This computer could totally destroy you at solitaire.

- We introduced **PSPACE**, the class of problems for which your laptop needs polynomially more *space* as problems get harder. WHITE-WINNING, or "does white have a winning series of moves in such-and-such chess position" was introduced as an example of a **PSPACE**-complete problem.
- We introduced the class **EXPTIME**, motivated by the fact that a small change in the rules of chess to make it "unbounded" causes WHITE-WINNING to become **EXPTIME**-complete. We left off there.

5.2 A different take on complexity classes

The point at which I became truly interested in complexity theory was when I read [3]. In this blog post, the author makes the claim that complexity theory is the study of *quantitative theology* — in his words, it's "the mathematical study of hypothetical superintelligent beings such as gods".

This was the motivation for the "godlike **NP** computer" that I brought up in section 3.1. It raises[18] the question — what other kinds of godlike computers are there out there? Is there a **PSPACE** computer that can solve any **PSPACE** problem in polynomial time? How about an **EXPTIME** computer?

One complexity class we didn't discuss is called **BPP** — that class is often defined by the machine rather than by the problems.

> **BPP** is the class of problems that can be solved in polynomial time by a deterministic Turing machine[a] with the ability to flip a totally random coin.
>
> [a] Like your laptop.

Incredible! **BPP** is perhaps the most fundamental of the complexity classes that deal in *randomness*. Whether **BPP** = **P** is an open question — it asks, "is there any problem we can solve with randomness that we can't without it?"

There's more than one kind of randomness. Let's find another machine, and make a complexity class out of that one.

> **BQP** is the class of problems that can be solved in polynomial time by a quantum computer.

There's lots to be said about **BQP** (including how it is *not* equal to **NP** — that is, a quantum comptuer is not the godlike **NP** computer we considered earlier), and a lot of it is brought up in [2].

One more:

[18] It *raises* the question, not begs it. [5] I figure I might as well seize upon the opportunity to inform you of this fact, because it drives me insane, and even Roger Penrose has been known to make this common error. *Even you, Doctor Penrose!*

P/$f(|n|)$ is the class of problems that can be solved in polynomial time by a deterministic Turing machine if there is an all-powerful angel that gives the computer the best advice it can for each set of instances of a certain size $|n|$, the size of which is given by $f(|n|)$.

Okay, now this is just silly.[19]

5.3 Going on your way

Obviously, there's more to complexity theory than this. You can get a true sense of the scale from figure 7.

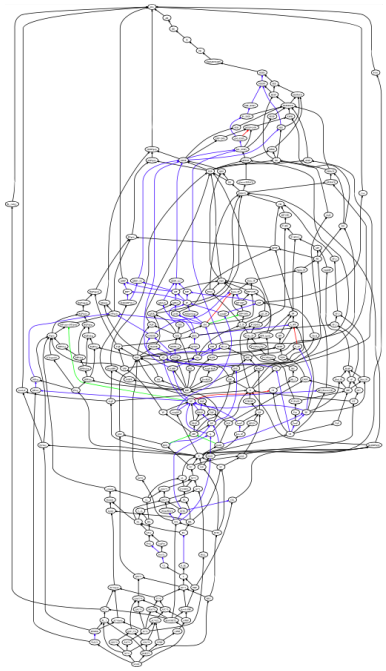

Figure 7: Known relationships between some complexity classes on [4]. (The point of this figure is more the sense of scale — go to [7] to zoom in.)

This humble guide provides one particularly visual undergraduate's view of complexity theory, with the understanding that looking at the same object from multiple angles can lead to a fuller understanding of its shape, but it by no means promises to be a complete guide to even the absolute basics.

If you're looking to keep learning, [2] is a free set of lecture notes by easily the funniest researcher in the field, [4] is a great reference guide he put together for orienting yourself in the morass of complexity classes (start at the Petting

[19] It's useful, I promise, but I'll let someone else show you that.

Zoo!), and [3] is a great post on his blog. If you for whatever reason find yourself sick of Scott Aaronson (he's pretty much how I learned complexity theory, and I haven't tired of him yet) and desirous of a good, hard textbook, [8] is the best I've come across.

References

[1] 3Blue1Brown. *Neural Networks*. `https://www.youtube.com/watch?v=aircAruvnKk`. 2017.

[2] Scott Aaronson. *PHYS771: Quantum Computing Since Democritus*. `https://www.scottaaronson.com/democritus/`. 2006.

[3] Scott Aaronson. *The Fable of the Chessmaster*. `https://www.scottaaronson.com/blog/?p=56`. 2006.

[4] Scott Aaronson and Greg Kuperberg. *Complexity Zoo*. `https://complexityzoo.uwaterloo.ca/Complexity_Zoo`.

[5] *Beg (Begging) the Question*. `https://www.merriam-webster.com/words-at-play/beg-the-question`.

[6] Erik D. Demaine. *Playing Games with Algorithms: Algorithmic Combinatorial Game Theory*. `http://erikdemaine.org/papers/AlgGameTheory_GONC3/paper.pdf`. June 2001.

[7] Greg Kuperberg. *Complexity Zoology Introduction*. `https://www.math.ucdavis.edu/~greg/zoology/intro.html`.

[8] Christopher Moore and Stephan Mertens. *The Nature of Computation*. Textbook. 2011.

[9] *Wikipedia: Complexity Classes*. `https://en.wikipedia.org/wiki/Complexity_class`.