Basic ideas…

Class P (Polynomial time)

P denotes the class of languages, L, such that there exists a Turing machine that decides L in polynomial time – that is, whose run time on all input strings is in for some positive number k (where n denotes the length of the input string)

Similarly, we can discuss the computational complexity of “function problems” (algorithms that are designed to answer a specific question, rather than just returning “yes” or “no”). Such a problem is considered to be in the class P if it can be answered by an algorithm whose run time on all inputs of length n is in .

Note: it turns out that “enhancements” of Turing machines (extra tapes, extra read/write heads, etc.) can lower the value of “k,” but any language decided in polynomial time by such a machine can also be decided in polynomial time (possibly with a higher exponent, k) by some standard Turing machine. This is discussed in Chapter 14 of the text; we will simply go with it here. This assumption allows us to, as usual, conflate “algorithm” with “Turing machine” for the purposes of determining whether a language is in the class P.

Example: Consider the language . We describe a polynomial time algorithm for deciding this language.

Assume an input string of length n.

* Scan the prefix of 0’s (reject if string is empty or starts with a 1) – this takes at most n steps.
* Next scan the suffix of 1’s (reject if the string contains no 1’s, or has a 0 after a 1) – this also takes at most n steps.
(Note these first two parts of the algorithm *may* combine to take far less than 2n steps – for example, if the input string is not in , the algorithm will halt and return “NO” much more quickly. When measuring run time for an algorithm, though, we make sure to find an upper bound for the number of necessary steps.)
* Next, move right to left, copying the suffix of 1’s onto a second tape (at most 2n steps) – n steps of reading 1’s, n steps of writing 1’s on the other tape).
* Finally, simultaneously scan the string of 0’s on the first tape and the string of 1’s on the second tape (at most 2n steps). Accept if the lengths of these strings match; reject otherwise.
* Total: 8n steps. This means our algorithm’s run time is bounded by a constant multiple of the input length, so we can say the run time is in , or simply . Thus, this language is in the class P.

Note: the “8n steps” result may overlook the actual inner workings of the machine/algorithm that decides the language; perhaps, for example, it takes 2 or 3 steps rather than 1 to copy a character from one tape to another. If so, this is accounted for by the - at worst, the number of steps required to carry out a verbal “step” in the above description will be a maximum of M, for some finite number M – this would mean our algorithm’s run time is at most (8M)n steps rather than 8n steps. But this is still a constant multiple of n, so still applies.

Example: An example of a “function problem” that is in P is, given a binary input w, find the length of the longest substring of 1’s. (For example, on input 11000101010111110100111, the output would be 11111.)

Our algorithm here could be the following: Scan the input left-to-right. When a 1 is read, copy it and the following string of 1’s (if any) to a second tape, stopping when a 0 is encoutered. Each time a new string of 1’s is encountered, go back to the beginning of the second tape and overwrite the old substring of 1’s with the new one. (Note – this will leave the second tape unchanged unless the new string of 1’s is longer than the old one.) Once this algorithm terminates, the second tape’s contents will be the longest substring of 1’s in the original input.

The number of steps for this algorithm, “worst case,” can be thought of as follows:

\* Overall, the entire input string needs to be read – this takes steps.

\* Each string of 1’s is copied onto the second tape. Each time a string of 1’s is encountered, it will take, say, steps to copy this string onto the second tape. Since the total combined length of the strings of 1’s is at most , the copying will take at most steps.

\* Therefore, our algorithm’s run time is at most steps, which is in . Thus, the algorithm for solving the problem runs in polynomial time, so the problem is in class .

(Note: again, depending on the operation of the machine, each “step” above may actually take several Turing machine transitions. This is fine, though; let the constant M denote the maximum number of moves each “step” requires, and then our runtime is at most , which is still in . )

Example: A “function problem” that is (probably) NOT in class P is factorization. That is, given the positive integer N as input, find an integer d>1 such that N/d is an integer (or return “prime” if no such d exists).

Important detail: for a positive integer N, the length of the input string is not N, but rather , where the base of the logarithm depends on the numeration system used… for example, if N is in binary, then its length in binary digits is the greatest integer less than ; for decimal notation, the number of digits is the greatest integer less than (Try a few examples to see why this is the case.) So for the purpose of determining computational complexity, we’ll use or equivalently . (This is a slight approximation, but this doesn’t affect our result significantly.)

The usual algorithm for this is to test all potential divisors from d=2 through ; if one divides , return its value; if none does, return “prime” since N is prime in that case. This process (described in class) will take up to, in the “worst case” (N prime), N(1/2+1/3+…+steps, which is approximately steps. This is before accounting for the number of moves needed for each divisibility check -- but that won’t affect the fact that we are now looking at exponential, rather than polynomial, run time for our algorithm…

(\* Note: from calculus, the approximation can be used as a rough approximation for for large values of .)

Class NP (will expand on this for tomorrow 5/12):

NP stands for “nondeterministic polynomial.” A language (or problem) is in the class NP if there exists a verification algorithm for that language (or for a solution to a problem) that can be run in polynomial time. This is equivalent to an decision/solution algorithm that runs on a nondeterministic (or, lucky guessing) Turing machine in polynomial time.

For example: the factoring algorithm described above would take a very long time to find a factor of, say, 5551. (Or, 1010110101111 in binary.) On the other hand, if we nondeterministically started our algorithm with d=61, we’d find pretty quickly that 5551/61=91. That was quick! Just one step, in fact. …OK, that’s an exaggeration. More specifically, a Turing machine would probably need to subtract 61 from 5551 91 times (this is essentially how long division works, except streamlined a bit.) Technically, there are several steps involved in division, but this number will scale linearly with the length of the input, not exponentially. Therefore, we have a polynomial-time verification algorithm for the function problem of factoring.

So, for example, the problem of factoring positive integers is in the class NP. As far as we know, it is not in the class P. We’re pretty sure it isn’t in P, but no proof of this fact exists! In fact, that’s the Million Dollar Question – **literally**, in fact!\*\*

Some important/famous examples of problems in the class NP are discussed in Section 14.5 of the textbook. In particular, 3SAT, Hamiltonian Path (and the closely related “Travelling Salesman problem”), and Clique are examples you should be familiar with. Make sure to read up on these if they are new to you!

Definition: The class “NP-Complete” consists of a set of problems in class NP such that each problem in the class is reducible (in polynomial time) to each of the others. That is, a polynomial-time algorithm to solve any of these problems can be converted into a polynomial time algorithm to solve ALL of them!

If we had another week or two, we could go into how to reduce one NP-complete problem to another… it is an interesting mental exercise, but covering it in a meaningful way would require much more time than we have. (Sorry about that!) Instead, for anyone interested in learning more, here are a few recommended links on the topic of computational complexity (P, NP, and related topics/classes of problems)…

* Clay Institute: <http://www.claymath.org/millennium-problems/p-vs-np-problem>

The question of whether P=NP (either proving it false, or – far less likely, in my opinion – proving it true) is central to computational theory; a resolution either way would be a major breakthrough. Therefore the Clay Institute has designated it as one of its “Millenium Prize” problems. A verified (peer-reviewed) solution to any of these problems comes with a one million dollar prize! (\*Like I said – literally a million dollar question.)

* MIT Open Courseware lecture: <https://www.youtube.com/watch?v=mr1FMrwi6Ew>

A great lecture by Prof. Erik Demaine of MIT covering the subject of computational complexity. There’s a lot of material online on this subject, but I’d recommend this lecture in particular if you’re interested in learning more.

* <http://www.travellingsalesmanmovie.com/> - This independent film came out about five years ago, and is available on Amazon Prime. It’s a very smart movie that discusses both the meaning and the ramifications of a solution to the P vs. NP problem. Highly recommended for all aspiring mathematicians and computer scientists!