COMP 314 Class 19: The complexity class PolyCheck

The motivation for defining complexity classes Poly and Exp is reasonably clear: problems in Poly are usually "easy" and problems in Exp (but not in Poly) are usually "hard." In contrast, there is no such obvious motivation for the next complexity class we will look at—a class we'll call PolyCheck. The true motivation behind PolyCheck becomes clear only after we have seen what we can and cannot prove about the problems in this class. It will turn out that PolyCheck contains most of the genuinely useful, yet apparently "hard" problems—the hard problems that crop up in the computational sciences such as mathematics, biology, and engineering. Infuriatingly, however, no one has yet proved that any of the problems in PolyCheck really is "hard." But computer scientists believe that by studying this special and peculiar class, PolyCheck, we will one day gain a clearer understanding of the difference between "hard" and "easy" problems.

Hopefully, this gives you enough motivation to accept the following definitions of PolyCheck. Let's start with an extremely informal definition: PolyCheck is the class of problems for which it might take a long time to compute a solution, but once you have a solution, it's easy for a separate computer program to verify that solution quickly. FACTOR provides a concrete example of this (see Figure 1 for a formal definition of the problem). As far as anyone knows, there's no efficient way of finding the prime factorization of a large number; the best-known existing algorithms take exponential time. However, once you have a number's prime factors, there's an extremely efficient way of checking that the answer is correct: simply multiply the alleged factors together, and check the result equals the original input. Hence, at least according to our initial informal definition, FACTOR is in PolyCheck.

Problem FACTOR

- **Input:** A positive integer M in decimal notation
- **Output:** A list of all prime factors of M.

Figure 1: Description of the computational problem FACTOR.

With this example in mind, we're ready to tackle a more rigorous definition.
Definition of PolyCheck: A computational problem is in PolyCheck if you can check that its output is correct in polynomial time (as a function of the input size)—but with two caveats:

1. (the "hint" caveat) The process of checking the output is allowed to use a hint if desired, as long as the length of the hint is bounded by a polynomial (as a function of the input size).

2. (the "no" caveat) If the output happens to be the special value `no`, the process of checking the output can take as long as you want.

Let's agree to call a problem instance *positive* if the answer is something other than "`no`". And an instance is *negative* if its answer is "`no`". Then we can reformulate the definition of PolyCheck yet again: A computational problem is in PolyCheck if you can verify the output for positive instances in polynomial time, possibly with the help of a polynomial-sized hint. This definition includes both caveats in a single sentence, since it restricts to positive instances and mentions the possibility of a hint.

Let's reassure ourselves that FACTOR satisfies the formal definition of PolyCheck. The input is an integer M, which is n decimal digits long, and we know that $n \approx \log_{10} M$. Now suppose someone tells us an alleged prime factorization of M, and we want to check that it really is a prime factorization. How long might it take to do this, in the worst case? To keep the analysis simple, we will use some very sloppy upper bounds (after all, we only need a method that is polynomial in n). Firstly, how many factors could M have? The number of factors will be highest when the value of the factors is lowest, and the lowest possible value of a non-trivial factor is 2. So the number of factors is at most $\log_2 M$, which is $O(n)$. So we need to multiply together only $O(n)$ factors. How long could just one of these multiplications take? Even in the worst case, a factor can't be longer than the n digits of the input, and multiplying two n-digit numbers by the naïve gradeschool method is $O(n^2)$. Thus, we perform $O(n)$ multiplications that take $O(n^2)$ time each, for a total time of $O(n^3)$. Finally, we also need to check each of the factors is actually prime. But it turns out there's a polynomial-time algorithm for checking primality. So the entire checking process takes polynomial time, showing that FACTOR really is in PolyCheck.

Note that neither of the caveats in the definition applies to FACTOR. We didn't need any hint to check the correctness of the factorization, so the "hint" caveat is irrelevant. And the problem FACTOR never outputs the special value `no` (since its output is always a list of numbers), so the "no" caveat is irrelevant also.

Let's now look at some more examples, to understand how the two caveats can come into play.

The "hint" caveat

The "hint" caveat—that we might need a polynomial-sized hint to help us check the solution—usually applies when the computational problem has been defined in an artificially narrow way. For example, considering the following silly variant of FACTOR:

Problem FACTORSENDINGIN7

- **Input:** A positive integer M in decimal notation
- **Output:** A list of all prime factors of M that end in the digit 7.

FACTORSENDINGIN7 has no useful applications; we examine it purely to understand why a hint might be needed to check a solution. To see this, imagine that the input M is a

10,000-digit number, and the alleged solution which we have to check is `17,47`. Of course it is easy to check that 17 and 47 divide M, so we can efficiently verify this aspect of the solution. But how do we know the list is *complete*? It is certainly possible that M has some other prime factor ending in 7 too—imagine, for example, that there is a 5,000-digit prime factor of M that ends in 7, and therefore should also have been part of the solution. As far as anyone knows, it takes exponential time to verify that this 5,000-digit prime factor doesn't exist.

Hence, we need a hint. In this case, a good choice for the hint would be a list of all other prime factors of M. Once we have those, we can check in polynomial time that the list of prime factors is complete and correct (exactly as we did for FACTOR above), and then also verify that none of the other factors ends in 7 (this last check is $O(n)$, and therefore certainly polynomial). Finally, note that the hint itself has to be polynomial in length. In this case, it's easy to see this: from the analysis of FACTOR above, we know there are at most $O(n)$ prime factors, each of length $O(n)$; so the hint itself is $O(n^2)$ (this is a very sloppy bound, but good enough for our purposes). Thus, by employing the "hint" caveat, we have proved that FACTORSENDINGIN7 is in PolyCheck.

Note that we can assume the hint is provided to us for free, perhaps by an omniscient deity. To prove membership in PolyCheck using the "hint" caveat, it only matters that a suitable hint *exists*. It does *not* matter whether you could feasibly compute the hint. On the other hand, once we're given a hint, we do have to check that it really works. For example, we had to check that the allegedly prime factors in the hint above really were prime. You can't trust that the deity is telling the truth—everything has to be verified!

Do we really need the "hint" caveat?

Until you get used to it, the "hint" caveat makes the definition of PolyCheck somewhat messy and annoying to deal with. It's reasonable to ask if we really need this caveat at all—especially since the only example we have seen of a problem that needs a hint (FACTORSENDINGIN7) is a contrived and artificial one. The short answer is: yes, we do need the "hint" caveat—especially when dealing with decision problems, which are the traditional focus of complexity theory. Recall that the output of a decision problem is either `yes` or `no`. Suppose a sophisticated computer program analyzes a difficult problem for a long time and then outputs `yes`. To show membership in PolyCheck, our job is to efficiently check that the output is correct. But how can we do this, based only on a single bit of information ("yes"), without rerunning the whole computation? What we need is some kind of easily-checkable justification for *why* the output was `yes`. And this is exactly what a hint can provide. Let's stick with our example of factoring, to see this in action. A traditional text on complexity theory might present factoring as the following decision problem:

Problem FACTORDECISION

- **Input:** Three positive integers M, a, b in decimal notation
- **Output:** `yes` if M has a prime factor in the interval $[a, b]$, otherwise `no`.

The first thing to note is that this is a perfectly reasonable reformulation of FACTOR as a decision problem. Given a computer program *FD* that solves FACTORDECISION, we can use *FD* as a subroutine in another program F that solves FACTOR: by repeatedly calling *FD* with suitable values of a and b, we can use binary search to pin down the values of the factors one by one. And it's not hard to see that if *FD* runs in polynomial time, then so does F. So in some sense, the decision version of the problem captures all of the intrinsic difficulty of the full version: FACTORDECISION has an efficient solution if and only if FACTOR does.

Because the two problems have the same intrinsic difficulty, it seems sensible to define our complexity classes so that both problems are in the same classes. But if we didn't allow the "hint" caveat in our definition of PolyCheck, we would have the absurd situation of being able to prove that FACTOR is in PolyCheck, but not being able to prove the same thing for FACTORDECISION.

To check our understanding, let's now use the "hint" caveat to prove that FACTORDECISION does in fact belong to PolyCheck.

Claim: FACTORDECISION is in PolyCheck.

Proof of the claim: Suppose we've been given the inputs M, a, b (with total length n), and program *FD* has output `yes` on these inputs. We need to check that `yes` is the right answer, in polynomial time, and using a polynomial-sized hint. Hopefully, the necessary hint is obvious: we just need to know the value of the prime factor found by *FD*—let's call it K. The length of K is less than n, so the hint is of polynomial size, as required. We then need to check that (i) K is really a factor of M (use division for this); (ii) $a \leqslant K \leqslant b$; and (iii) K is prime. Each of these checks is polynomial in n, and hence FACTORDECISION is in PolyCheck. ■

The "no" caveat

Now we turn to the second wrinkle in the definition of PolyCheck: the "no" caveat, which says that a problem can be in PolyCheck even if there is no efficient way of checking its `no` answers.

An informal reformulation of this is as follows. We can think of a problem in PolyCheck as being a search for a correct solution among a vast space of possible solutions. Although it's a tired cliché, this really is just like searching for a needle in a haystack. Think of searching for the factors of a 10,000-digit number: each factor is a needle, and each non-factor is a piece of hay. In this example, there are far more pieces of hay than atoms in the observable universe.

Pushing our cliché a little further, we could say that a problem is in PolyCheck if there is an efficient way of verifying needles: at a glance, we need to be able to say "yes, that's a needle, it's definitely not a piece of hay." This is where the "no" caveat comes in: if no needle is found, efficient verification isn't required. More specifically, suppose a computer program that solves a PolyCheck problem produces the output, "no, there are no needles anywhere in this haystack." Then it's okay to verify this output by painstakingly going through the haystack until we're sure that no needle is present.

You may not have noticed, but we actually used the "no" caveat in our last proof. Your job is to explain exactly where and how:

Compulsory exercise 1. Explain where and how the "no" caveat was used in the proof of the claim that FACTORDECISION is in PolyCheck, on page 4.

I should be completely honest and admit that this is actually a bad example of the "no" caveat. I included it anyway because we are already familiar with the FACTORDECISION problem, and have invested some effort in analyzing it. But it turns out that—if we're given the right hint—we can efficiently verify `no` outputs from FACTORDECISION. The hint we need is the same as for `yes` outputs: the entire prime factorization of input M. As discussed earlier (page 3), this hint is polynomial in length. And it's trivial to check that none of the prime factors lies in the range $[a, b]$ specified in the input. So we can verify the `no` output in polynomial time, and hence we didn't need to take advantage of the "no" caveat.

So let's instead look at a *good* example of the "no" caveat. For this, we'll need the PACKING problem defined in Figure 2. Informally, we are given a bunch of packages with weights, and we want to load them onto a delivery truck that needs at least weight L to make a delivery worthwhile, and can carry at most a total weight H. If this can't be done, the output is `no`. Otherwise, the output is a list of which packages can be used to produce a feasible packing of the truck. It's extremely easy to prove that PACKING is in PolyCheck—try to do this on your own:

Compulsory exercise 2. Prove that PACKING is in PolyCheck. Note that you won't need the "hint" caveat in the definition of PolyCheck, but you will need the "no" caveat.

Solutions to compulsory exercises

1. The "no" caveat was used in the first sentence of the proof on page 4. Specifically, we assume that the program *FD* has output `yes`, and go on to explain an efficient way of verifying this output. But we didn't consider the possibility that *FD* had output `no`, because the "no" caveat in the definition of PolyCheck guarantees we don't have to. Of course, *FD* frequently does output `no`, and it must be correct when it does so—but we don't need an efficient way of verifying this correctness.

2. To prove that PACKING is in PolyCheck, we need to show that any answer—except a "`no`" answer—can be verified in polynomial time (possibly with the help of a polynomial-sized

Problem PACKING

- **Input:**
  - A list of positive integers in decimal notation separated by whitespace. For example, this part of the input could be `12 4 6 24 4 16`. These integers are called the *weights*; we think of them as the weights are some objects to be loaded into a delivery truck. Denote the weights by $w_1, w_2, \ldots w_m$, where m is the number of weights.
  - Two integer thresholds L and H, also in decimal notation. The L stands for Low and H for High. We think of L as the minimum weight that makes it worthwhile for the truck to make a delivery, and H as the maximum weight that the delivery truck can carry. As a specific example, let's take $L = 20$ and $H = 27$.
- **Output:** A *feasible packing* is a subset of the input weights whose total weight W lies between the low and high thresholds: $L \leqslant W \leqslant H$. If no feasible packing exists, output `no`. Otherwise, output a subset S of the weights that represents a feasible packing. For the above example, one correct output is `2 3 6`. This means that a feasible packing of the truck is achieved by loading objects 2, 3, and 6. The total weight is therefore $W = w_2 + w_3 + w_6 = 4 + 6 + 16 = 26$, and $L \leqslant 26 \leqslant H$, as required.

Figure 2: Description of the computational problem PACKING. The decision version of this problem is known as SUBSETSUM.

hint). In this case, a positive answer consists of some indices, specifying which packages should be loaded onto the truck. So, we just need to add up the specified weights and check that the sum is between L and H. Each of these operations requires at most $O(n)$ time, where n is the length of the input. So the positive answer can be verified in linear time, which is certainly polynomial. For a negative answer, on the other hand, we seem to be stuck: there is no obvious way of verifying that the truck *can't* be loaded, other than trying all possible subsets of the input integers—and this will take exponential time. But fortunately, the "no" caveat tells us this doesn't matter, so we have completed the proof that PACKING is in PolyCheck.