I'd like to provide a little background, and a little tease, for question 1.

First of all you have to recognize that factoring integers is hard, much harder than multiplying. If you have to multiply two 100-digit numbers the way you learned in grade school, that will mean multiplying each of the 10,000 pairs of digits (each pair contains one digit from each of the numbers you're trying to multiply), simply using your times tables, and then adding up the (up to 100) numbers in each of the (up to 200) columns. That's a lot of little steps, maybe some tens of thousands of them, but each can be done in one clock tick on a computer, so those big numbers get multiplied in a fraction of a second. Of course it takes longer to multiply bigger numbers: 200-digit numbers would take about 4 times as long.

By contrast, factoring a 100-digit number N by the grade-school algorithm would take much longer. You look for factors using trial division, testing whether N is an even multiple of 2, then 3, then 4, then .... You may know you only need to test with divisors up to $\sqrt{N}$, and you only need to test the divisors that are themselves prime, but still, if N is a 100-digit number, that would mean something on the order of 10^{47} trial divisions. Even if each test could be accomplished in a single computer clock tick, this task could not be computed on my computer in less than the age of the universe!

This is not to say one HAS to use the grade-school algorithm, and indeed much faster algorithms are known. It's not unreasonable these days to expect the factorization of a 100-digit number in a few hours! But we still say factorization is very hard; the amount of time needed to factor a big number doubles every time we add a few digits, even with the best algorithms and the best machines. Attacking a 200-digit number is still pretty hopeless.

More precisely, what we are discussing here is an estimate of how long it takes to perform these two tasks with n-digit numbers. For multiplication, the time needed is roughly $C\,n^2$ for some constant C; for factorization it's something like 10^{Cn}. We say the first task can be computed "in polynomial time" but the second *seems* to require "exponential time". The holy grail in this particular business is to find a polynomial-time factorization algorithm.

Look up the Euclidean Algorithm. It's a simple way to take *two* integers and find their greatest common divisor. And it runs very quickly; the time needed is no more than a polynomial in the number of digits in the two integer inputs.

So here's an idea for factoring a big number N. First compute $\gcd(N, 6)$ to see whether N is divisible by any 1- or 2-bit primes. Then compute $\gcd(N, 35)$ to see if it has any 3-bit prime factors; then $\gcd(N, 143)$ checks for 4-bit prime divisors, and so on. Each one of these steps requires the Euclidean Algorithm, but I just told you that algorithm is fast, and if N itself is an n-bit number, then you would discover all its prime divisors by the time you had computed the gcd of N with the product of all $n/2$-bit primes. That is: I am asking you to run a polynomial-time algorithm, a polynomial number of times; thus the total run time would be a polynomial in n. See? There you go! a polynomial-time algorithm to factor any big integer. (I'm lying a little; there are issues related to repeated factors, and to pairs of prime factors that have the same numbers of digits (or bits). But there are ways around those issues, and maybe no one cares about those issues anyway.)

There's only one catch here: if you wanted to encode all this into a computer program, you would have to pre-compute the numbers 6, 35, 143, etc. that I describe in the previous paragraph., Already the product of the 5-bit primes (17,19,23,29,31) is getting pretty large, and there doesn't seem to be a pretty way to write it down or even to compute it, short of making a long list of all prime numbers.

A possible rescue is indicated in Problem 1 of this week's problems: we don't need literally to compute the product P_n of all the n-bit primes; it's sufficient to find a number divisible by all these primes, and now you know one such number: it's $C_k = \binom{2k}{k}$, where $k = 2^{n-1}$. So our computer program to factor n-bit numbers only needs to compute the first n numbers in the sequence

$$\binom{4}{2}, \binom{8}{4}, \binom{16}{8}, \ldots$$

It's just a select group of numbers down the middle column of Pascal's Triangle. Imagine — we could factor thousand-bit numbers N (which no one else can do yet!) by running our polynomial-time algorithm that computes $\gcd(N, C_k)$ for the first thousand or so numbers C_k in the the list above. (And don't be put off by the size of these numbers C_k : since we're only using them to compute a gcd with N, it will be sufficient to replace C_k with its reduction modulo N, which is then another thousand-digit number, nothing larger. Your computer can handle this!)

So, OK, here's the challenge: find a fast way to compute the above number C_k (modulo N). I only want a small bunch of these numbers (yes, 1000 is "small"!) but the problem is that they are pretty sparse in Pascal's triangle; the thousandth one is in row 2^{1000} of Pascal's triangle, so I require a means of getting these C_k that does not require computing the entire central column of Pascal's Triangle!

Now here's where I get sneaky. There is, at least, a way to compute the central column of PT without filling in all the other entries. It turns out that the numbers we need are precisely the coefficients of the Taylor Series of $1/\sqrt{1-4x}$! Here are the first few terms:

$$\begin{aligned}1 + 2t + 6t^2 + 20t^3 + 70t^4 + 252t^5 + 924t^6 + 3432t^7 + 12870t^8 + 48620t^9 + 184756t^{10} + 705432t^{11}\\ + 2704156t^{12} + 10400600t^{13} + 40116600t^{14} + 155117520t^{15} + 601080390t^{16} + 2333606220t^{17} + \ldots\end{aligned}$$

Our focus is on $C_2 = 6$, $C_4 = 70$, $C_8 = 12870$, $C_{16} = 601080390$, and the other coefficients of t^{2^k}. (I invite you to check that C_{16} is indeed divisible by all the 5-bit primes, each to the first power, and by no larger primes. It is divisible by $2 \cdot 3^2 \cdot 5$ as well, but that does not really present a problem for our proposed factorization algorithm.

To summarize everything so far, we see that (modulo a few technical details that are not usually important) *it is possible to factor any n-bit number N in an amount of time which is polynomial in n, if we can first compute (quickly) the n numbers C_k ($k = 2, 4, 8, \ldots 2^n$) modulo N.*

Well, here's another surprise: I can do that! Sort of. What I can do is describe an iterative process that computes the above power series; with each iteration we get twice as many of the leading coefficients correct. So the initial iteration gives us $C_2 = 6$; the next iteration has more correct terms including $C_4 = 70$; the next iteration has twice as many correct terms, including $C_8 = 12870$, etc. Obviously we only need to carry out the iteration n times, which is to say the iteration process is "fast". Oh and as a bonus we can do all the calculations mod N along the way, so that none of our coefficients will have more than n bits while we compute with it.

Here's the recipe. You remember Newton's Algorithm from Calculus I ? It iteratively solves equations of the form $f(x) = 0$, starting with an initial approximation x_0. Each iteration simply replaces an approximation x_k with a new one defined by

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

You remember that formula, right? In Calc-I you apply this to (approximately) solve equations like $x^2 - 5 = 0$. Well, how about if you use it to solve the equation $f(x) = 0$ where $f(x) = (1/x^2) - a$ for some number a? You already know the solution to this equation: it's $x = 1/\sqrt{a}$. But the point is that Newton's method allows you to approximate this solution numerically. And (drumroll) for this particular function f, the Newton iteration formula simplifies into something very striking:

$$x_{k+1} = x_k - \frac{1/(x_k)^2 - a}{-2/(x_k)^3} = x_k + x_k \frac{1 - a x_k^2}{2} = x_k(3 - a x_k^2)/2$$

which is striking because it involves no division (except the "2", which is a "bit-shift").

Well, (in a way that can be justified using some funny mathematical frameworks) we can use precisely this recursion, starting with $x_0 = 1$, to get better an better approximations to $1/\sqrt{a}$, *even when a is a polynomial like $1-4t$!* Each approximation x_k will then be a polynomial in t, and as it turns out each x_k will have *twice as many* correct coefficients as its predecessor. I invite you to work this out too: we get

$$\begin{aligned}
x_0 &= 1\\
x_1 &= 1 + 2t\\
x_2 &= 1 + 2t + 6t^2 + 20t^3 + 16t^4\\
x_3 &= 1 + 2t + 6t^2 + 20t^3 + 70t^4 + 252t^5 + 924t^6 + 3432t^7 + 8496t^8 + \dots
\end{aligned}$$

Note that as a bonus the calculations can all be done modulo N, for any odd modulus N.

To summarize: here is all you need to do to factor an n-bit number N. Start with $x_0 = 1$. Then, for $k = 1, 2, \dots, n$, compute x_k by the above recursive formula (all done modulo N if you like), pick out C_k, the coefficient of $t^{2^{k-1}}$ in x_k, then compute $\gcd(N, C_k)$, which will simply be the product of all the k-bit primes that divide N.

Well then, is this a polynomial-time algorithm to factor big integers? Turns out there's a little technical issue running this algorithm. I invite you to try to discover what goes wrong (that part I know) and how to fix it (that part I can't do).

So, we're not *quite* there yet: it's still hard to factor big integers. But it was only a few years ago that they were saying it's even hard to determine for sure whether a number is prime, but then a polynomial-time primality-testing algorithm was invented (by some high school students!) so who knows, perhaps there are enough ideas here to allow *you* to develop the first polynomial-time factorization algorithm!