In this week's Putnam practice I asked a hard number theory question:

10. Suppose N is an integer of at most 1000 decimal digits. Describe a way to compute the central binomial coefficient

$$\binom{2^{1000}}{2^{999}} \pmod{N}$$

using at most one billion arithmetic operations on integers of at most 1000 digits each.

This is a very hard question and as far as I know, no one knows the answer. I'd like to explain the relevance of this question.

It is easy to multiply numbers, even big numbers. If I give you two 1000-digit numbers and ask you to multiply them, you could do it in a day. After all, there's only one million pairs of digits that have to be multiplied, and that will make for a couple million additions and carries, but at the end of the day you will have a 2000-digit product.

By contrast, factoring big numbers is hard. If I hand you a 1000-digit number N and ask you to factor it, you will probably test for divisibility by 2, then by 3, then by 5 and 7 and 11 and so on. You won't be able to stop until you have factored it completely, which might require you to test all the primes up to $\sqrt{N}$. But that's about 10^{500} candidates! Even if you could test a trillion primes per second, across the entire age of the universe you would not ever come close to testing the hundred-digit primes, let alone the 200-digit ones and 300-digit ones and so on. Factoring is hard.

This difference between multiplying and factoring is at the basis of a popular encryption technique, the RSA Algorithm. (Look it up.) That's what's used every time you log in to a secure web site (the ones with `https` in the front). You get two large primes and multiply them together to get your personal encryption key N; there's a fast protocol for encryption using N so anyone can easily send you secret messages. And there's a fast protocol for decryption too, but it requires knowing more than just N: to decrypt, you need the prime factors of N. So the encrypted messages are secure, even if an enemy gets hold of them: the enemy cannot decrypt them precisely because factoring is hard.

Or is it? We actually have some techniques that can factor numbers faster than the naive method. Combining those algorithms together with specialty hardware, it's not really a challenge any more to factor 120-digit numbers. But even using these tools, 200-digit numbers are still extremely difficult; 1000-digit numbers are basically impossible. (Maybe the NSA can factor big integers somehow, but if so, they are not revealing their secret!)

But there is something we could try. Although it's hard to factor *one* number, it's easy to find the *common* factors shared by two numbers. That's the Euclidean Algorithm. Look it up: it's very easy to implement on a computer. Given another integer M we can compute the gcd of N and M easily, even if N is big and M is even bigger than N is. Start by computing the reduction of M modulo N, that is, find an $M' < N$ with $M \equiv M'$ modulo N. The Euclidean algorithm simply uses the fact that $\gcd(N, M') = \gcd(r, M')$ where r is the remainder from dividing the smaller number into the larger one. Just keep iterating this observation and you replace the two integers with smaller and smaller pairs,

reducing the total number of digits at least once every few iterations. So after only a couple thousand iterations you will find the gcd of N and M. That's fast.

That's also useful, if M has prime divisors you are interested in, and this is where the problem I posed originates. You might organize a factorization algorithm by checking your number N for larger and larger prime divisors, but not one prime at a time. Instead, first try all the 2-bit primes (the ones which are expressed in binary with just two bits) by computing gcd(N, 6). Then move on to the 3-bit primes by computing gcd(N, 35). Then comes gcd(N, 143), etc. A 1000-digit composite nuumber will definitely have a factor less than 10^{500}, which is about 2^{1661}, so it will only take a couple thousand iterations of this process, at worst, to discover all prime factors of N. That's fast!

There is the teensy little problem that we need to precompute these numbers 6, 35, 143, etc. We need a couple thousand terms of this sequence of numbers a_n each of which is the product of all the n-bit primes. Don't get overconfident about this: the very next term is $a_5 = 6,678,671$, and after that they escalate quickly! And in order to create these numbers, we will have to precompute all the 10-bit primes, and all the 107-bit primes, and all the 1661-bit primes and so on. That cannot be done in less than the age of the universe. And even if we compute the a_n, how could we write them down and incorporate them into a computer program? They have zillions of digits!

So let's try to be a bit more efficient about this. First off, we don't really need to assemble lists of primes. It's OK to replace these a_n with numbers that are at least *divisible by* all the n-bit primes, if we have some control over what their other prime factors are. And this is where the binomial coefficients come in. From the factorial description of the binomial coefficients in general you can see that for every k,

$$b_k = \binom{2k}{k} = \frac{(2k)!}{(k!)^2}$$

will be divisible by every prime between k and $2k$ (because the numerator is and the denominator isn't), and is not divisible by any larger prime. There are usually some smaller factors too, but these will not concern us if we go through the process above, computing gcd(N, b_{2^n}) for each value of n in turn: when $n = 1$ we will find all the 2-bit prime divisors of N, which we could then factor out from N if we like; then when $n = 2$ the gcd will reveal only the 3-bit primes, since all the 2-bit prime factors have been removed. Marching through a thousand or so gcd computations in turn will separate out all the prime divisors of N in order of their size in bits.

These central binomial coefficients are somewhat larger than the simple products of the relevant primes that we proposed earlier, but they should be easier to compute because no factorization is involved. For example, you could imagine simply filling in Pascal's triangle one row at a time, using just addition! Down the middle of it you would see the numbers b_k. Actually it's faster (and uses MUCH less storage space) to use a recursion to compute these b_k:

$$\sum_{k=0}^{k=m} b_k b_{m-k} = 4^m$$

This allows us to determine each of the b_k in turn. Here are the first few:

$$\begin{aligned} b_0 &= 1 \\ b_1 &= 2 \\ b_2 &= 6 \\ b_3 &= 20 \\ b_4 &= 70 \\ b_5 &= 252 \\ b_6 &= 924 \end{aligned}$$

You might want to verify what I said about the prime divisors of b_k in these examples.

Another way to state this recurrence relation is that if you look at the power series

$$F := \sum_{k=0}^{\infty} b_k X^k = 1 + 2X + 6X^2 + 20X^3 + 70X^4 + 252X^5 + 924X^6 + \ldots$$

then $F^2 = \sum_{m=0}^{\infty} 4^m X^m$, which is the Taylor Series for $1/(1-4X)$. So F itself is the Taylor series of the function $1/\sqrt{1-4X}$. I'll come back to this observation later.

Another recurrence relation is equally fast: $b_{k+1} = \frac{4k+2}{k+1} b_k$. And this one requires no storage of previous values of the b_k. So computing these central binomial coefficients doesn't look hard. This starts to look very promising — except that the numbers we want in our gcd computations are very far down the list: we only want a few of these terms, namely the terms $a_n = b_{2^n}$. These numbers are $2, 6, 70, 12870$, 601080390, 1832624140942590534, ... The preceding display of the first few b_n above looks like a thin triangle because the number of digits seems to grow linearly, but a display of the a_n just picks out a very few of those; the number of digits in the a_n grows quite quickly. Indeed there is an approximation for factorials called Stirling's Approximation, which reveals the number of digits in them: it gives an expansion of the form

$$\log(n!) = n\,\log(n) - n + \frac{1}{2}\log(2\pi n) + O(1/n)$$

(It actually gives that final error term as a convergent series in $1/n$ with explicit coefficients involving the Bernoulli numbers.) From this we can estimate

$$\log(b_k) = 2k\,\log(2) - \frac{1}{2}\log(\pi k) + O(1/k)$$

which, in English, simply says b_k is a little less than $2k$ bits long when expressed in binary. That means our $a_n = b_{2^k}$ will roughly double in length with each term as we write down the terms of the series! My computer got as far as computing all 157824 digits of a_{18} before I relieved it of the task of continuing the sequence. Let me emphasize this: a_{1000} will not have thousands of digits; it will have something like 2^{1000} digits! These numbers a_n are uselessly huge!

In short, we have a recipe for computing the numbers a_n we want for our gcd computations, and the recipe does not involve factorization, but it does require picking out our numbers from an impossibly long list, and even if we could compute just those sparse terms of the sequence, we cannot even hope to write down the answers! Forget trying to use terabyte hard disks; if you could somehow use every single proton and neutron in our planet to store a separate bit of information, you'd only be able to write out a 10^{50} bit number, which is about how long a_{166} is; you'd need two Earths to store a_{167}, and over a thousand Earths just to store a_{177}, and so on. Storing the number a_{1000} into a computer program is obviously impossible even if we could somehow precompute it.

But wait! We don't actually need these numbers a_n to be written out in full. All we are going to do with them is to compute $\gcd(N,\, a_n)$ and as I said in my description of the Euclidean algorithm, the first step is to compute a number M' that is congruent to a_n modulo N (but, typically, smaller than N). This number M' we certainly CAN write down and compute with, because it is only a thousand digits or so — *if* we can figure out how to compute it modulo N.

Let me clarify with an example. My estimate of the numbers b_k should be compared to the numbers $c_k = 4^k$, for which, obviously, $\log(c_k) = 2k\,\log(2)$. Compare that to the estimate I gave earlier:

$$\log(b_k) = 2k\,\log(2) - \frac{1}{2}\log(\pi k) + O(1/k)$$

and you see that the cs are larger than the bs. In particular, our horrible numbers $a_n = b_{2^n}$ are dwarfed by the even more horrible numbers $d_n = c_{2^n} = 4^{2^n}$. And yet, *these* huge numbers are very easily computed modulo N, since d_{n+1} is just the square of d_n, so we can simply start with $d_0 = 4$ and repeatedly square and reduce mod N (at each step). This would get us to d_{1000} with a thousand multiplications and a thousand reductions-modulo-N. I can't compute (or even display) d_{1000} but in less than a second I can determine that $d_{1000} \equiv 13277371$ modulo 123456789. That's fast.

So this is the point of the problem I posed: can you find a similarly fast algorithm which computes the slightly smaller numbers a_n, modulo N?

It's not unreasonable to hope for such a thing. Let me point out another example: the Fibonacci numbers $f_n = 1, 1, 2, 3, 5, \ldots$. You know you can compute these with a simple recursion (a bit simpler than the one I used for the bs !). You know that these things increase roughly linearly in length (as do the bs). What if you wanted to cut through to the powers of 2, just as I wanted to compute $a_n = b_{2^n}$, that is, how can you compute f_{2^n} quickly? The answer is known: use also the Lucas numbers $g_n = 1, 3, 4, 7, 11, 18, \ldots$ which use the same recurrence rule as the Fibonacci numbers but a different starting pair. These two sequences are intertwined in useful ways, including two identities

$$f_{2n} = f_n g_n \qquad g_{2n} = (5f_n^2 + g_n^2)/2$$

so we can start with the pair (f_2, g_2), then compute (f_4, g_4), (f_8, g_8), etc. After a thousand iterations we have the very large numbers $f_{2^{1000}}$ and $g_{2^{1000}}$. And since these recurrences are computed using only addition and multiplication (and division by 2) we can perform

all the arithmetic modulo N (as long as N is odd) and in particular compute $f_{2^{1000}}$ modulo N with just a few thousand arithmetic operations on thousand-digit numbers.

The same holds for the solution of a quadratic Diophantine equation such as a "Pell's equation", e.g. $x^2 - 2y^2 = 1$. The solutions listed in order have x and y coordinates that grow linearly in length, and there is a simple recurrence that gives each in terms of its predecessor, but we can cut through to the 1st, 2nd, 4th, 8th terms etc with a quadratic recurrence using both x and y. Like the previous example, this is because we're computing powers of certain algebraic numbers, and repeated multiplications can be abbreviated with repeated squarings.

I will toss out another example. In the theory of Elliptic Curves (look it up – it's another Diophantine problem) we might for example look for rational pairs (x, y) for which $y^2 = x^3+3x$. There are infinitely many solutions, starting with $x = 1/4, y = 7/8$, and when you list them out you notice that they steadily double in the length of their numerators and denominators, rather like our a_n. Yet they can be computed by a recursive formula, and there is a "group law" in the background that lets us cut through to the powers of two very quickly. (We will always have $x = X/Z^2, y = Y/Z^3$ for integers X, Y, Z computed by

$$X' = 12Z^4 + 49XZ^2 + 4X^2 - 28YZ, \qquad Z' = Z^2 - 4X$$
$$Y' = -42Z^6 - 168XZ^4 - 42X^2Z^2 + 145YZ^3 + 56X^3 + 204XYZ - 112Y^2$$

Whether any of these examples is similar enough to ours is not clear. I certainly know of no group law that applies to the central binomial coefficients like this. But it is not unreasonable to think we could construct perhaps a thousand sequences, one of which is our a_n, and each of which gets updated when n increases, using perhaps quadratic polynomial of all the others. That would enable us to compute a_{1000} modulo N in about a million steps. In our present context, I'd call that "fast".

Any suggestions? A back-of-the-envelope calculations suggests that a_{n+1} is about $\sqrt{2^{n-1}\pi}$ times as big as a_n^2 so it is plausible that there could be a triangle of numbers — think of Pascal's Triangle — with the a_n running down the left edge, and the other terms in the nth row being comparable in size (the largest being perhaps approximately $a_n \log(a_n)$). Then, plausibly, each term in the nth row might be a sum of small multiples of products of two terms from the previous row. I just don't know how to define these other terms in row n and how they should be built from the previous row(s).

Here's a possible way of thinking about these. The number a_n, like any binomial coefficient, has a combinatorial description. We can describe it as the number of ways of selecting exactly half of the vertices of a box in $\mathbf{R}^{n+1}$. For example a box in the plane has four vertices, and the pairs among them consist of the four edges and the two diagonals, for a total of $a_1 = 6$ subsets of cardinality 2. There are $a_2 = 70$ ways to select four of the eight corners of a cube, but these 70 ways correspond to just six pictures: 6 of those subsets are called "faces"; 8 of them are called "corner tetrahedra", and so on. (The other four pictures don't have evocative names, sorry.) The point is that the things a_n is supposed to count can be broken down into "a few" subsets, and if we can find a clever geometric classification of those subsets, perhaps that would allow us in some simple way to write a_n as a sum of smaller numbers, each easily relatable to a product of couple of much smaller numbers. I tried to think geometrically in a couple of ways but didn't see anything obvious. How about you?

Let me make one last point about the central binomial coefficients. I *do* have what appears to be a way to compute them fast! It has to do with how Numerical Analysts compute numerical solutions quickly on a computer. You may recall from Calculus a technique called Newton's Method that allows you to solve an equation $f(x) = 0$ with a sequence of approximations x_n, computed iteratively as $x_{n+1} = x_n - f(x_n)/f'(x_n)$. In practice, each of these approximations will have twice as many correct decimal digits as its predecessor! In particular you can compute square roots this way, but not the way you think. Instead, imagine trying to solve the equation $f(x) = x^{-2} - a = 0$. You know the solution is $x = 1/\sqrt{a}$. But Newton's Method suggests you get approximations to the solution using the recurrence

$$x_{n+1} = x_n - f(x_n)/f'(x_n) = x_n - (x_n^{-2} - a)/(-2x_n^{-3}) = x_n + x_n(1 - ax_n^2)/2 = x_n(3 - ax_n^2)/2$$

The key observation is that these updates are computed using only multiplication and subtraction (and division by 2). In terms of manipulating real numbers that can be handy. But it's also very useful in our context. You may recall that the central binomial coefficients form the coefficients of the Taylor series for $1/\sqrt{1-4X}$. Why don't we use Newton's Method to compute this Taylor series? That is, we start with the series $F_0 = 1$ and then construct more and more series (polynomials, actually) using the recurrence

$$F_{n+1} = F_n(3 - (1 - 4X)F_n^2)/2$$

Here are the first few of these:

$$\begin{aligned}
F_0 &= 1 \\
F_1 &= 1 + 2X \\
F_2 &= 1 + 2X + 6X^2 + 20X^3 + 16X^4 \\
F_3 &= 1 + 2X + 6X^2 + 20X^3 + 70X^4 + 252X^5 + 924X^6 + 3432X^7 + 8496X^8 + \ldots
\end{aligned}$$

Notice that each iteration correctly computes more of the coefficients of F; in fact, we double the number of correct coefficients with each iteration! (That's provable.) In particular, we can compute a_n with only about n iterations. We can get a_{1000} with just a thousand iterations of Newton's Method! And the steps used are only addition and multiplication, which means we can carry out all the computations modulo N and determine all these binomial coefficients mod N !

Well, more precisely, we have only an efficient way to compute the *value* of the polynomials $F_n(x)$ for any starting value of x. What we really need is one of the *coefficients* of the polynomial. The naive approach would call for us to expand each F_n in full, and each multiplcation of *polynomials*, not of integers, takes exponentially more time and space with each step. This method computes ALL the central binomal coefficients b_k, not just the a_n that I want, and as you already know there are way, way too many of those b_k for us to really accomplish such a goal as far as $a_{1000} = b_{2^{1000}}$.

So what is needed is a way to pick out just one coefficient of a polynomial which is defined by a simple recurrence like this. I don't know how to do it. Perhaps you do?