Solutions to Daily Due Sep 11 (AI generated)

Math 4440/5440, Coding and Cryptography, Fall 2026

Generated by Claude for Katherine E. Stange
(conventions follow the Fall 2024 lecture notes of Sept 23;
all computations computer-verified)

Problem 3: the parity of a discrete log

Throughout, p is an odd prime.

(1) If $ab \equiv 0 \pmod p$ then $a \equiv 0$ or $b \equiv 0 \pmod p$

$ab \equiv 0 \pmod p$ means $p \mid ab$. Since p is prime, p divides one of the factors (this is Euclid's lemma: a prime dividing a product divides one of the factors). If $p \mid a$ then $a \equiv 0 \pmod p$, and if $p \mid b$ then $b \equiv 0 \pmod p$.

Another way to say it: if $a \not\equiv 0 \pmod p$, then a is invertible modulo p (every nonzero residue is a unit modulo a prime), so we may multiply $ab \equiv 0$ by a^{-1} to get $b \equiv 0$.

(2) The only solutions to $x^2 \equiv 1 \pmod p$ are $x \equiv \pm 1$

If $x^2 \equiv 1 \pmod p$ then $x^2 - 1 \equiv 0$, that is, $(x+1)(x-1) \equiv 0 \pmod p$. By part (1), $x + 1 \equiv 0$ or $x - 1 \equiv 0 \pmod p$, so $x \equiv -1$ or $x \equiv 1 \pmod p$. Conversely both 1 and -1 do square to 1, and they are distinct residues because $p \neq 2$. So there are exactly two square roots of 1.

(This is where primality matters: modulo 15, for example, $4^2 = 16 \equiv 1$, so $x^2 \equiv 1 \pmod{15}$ has the four solutions $1, 4, 11, 14$.)

(3) If g is a primitive root then $g^{(p-1)/2} \equiv -1 \pmod p$

Let $y = g^{(p-1)/2}$. Then $y^2 = g^{p-1} \equiv 1 \pmod p$ by Fermat's little theorem, so by part (2), $y \equiv 1$ or $y \equiv -1$. But g is a primitive root, so its multiplicative order is exactly $p - 1$, and $(p-1)/2$ is a smaller positive exponent; hence $g^{(p-1)/2} \not\equiv 1$. The only remaining possibility is $g^{(p-1)/2} \equiv -1 \pmod p$.

(4) Deciding whether x is even or odd from $h \equiv g^x$

Raise $h \equiv g^x \pmod p$ to the power $(p-1)/2$:

$$h^{(p-1)/2} \equiv g^{x(p-1)/2} = \left(g^{(p-1)/2}\right)^x \equiv (-1)^x \pmod p,$$

using part (3). So compute $h^{(p-1)/2} \bmod p$:

- if the result is 1, then $(-1)^x = 1$ and x is **even**;

- if the result is $-1 \equiv p-1$, then $(-1)^x = -1$ and x is **odd**.

(No other value can occur, by part (2) applied to $y = h^{(p-1)/2}$, whose square is $h^{p-1} \equiv 1$.)

(5) Is x even or odd, where $2^x \equiv 5 \pmod{11}$?

Here $p = 11$, $(p-1)/2 = 5$, and $h = 5$. Compute 5^5 mod 11 by successive squaring:

$$5^2 = 25 \equiv 3, \qquad 5^4 \equiv 3^2 = 9, \qquad 5^5 = 5^4 \cdot 5 \equiv 9 \cdot 5 = 45 \equiv 1 \pmod{11}.$$

The result is 1, so x is **even**.

(6) Check by finding x

The powers of 2 modulo 11 are

$$2^0 = 1,\ 2^1 = 2,\ 2^2 = 4,\ 2^3 = 8,\ 2^4 = 16 \equiv 5,$$

so $x = 4$ (and, since 2 has order 10, the full solution set is $x \equiv 4 \pmod{10}$: all even). This agrees with part (5). (The same list continued, $2^5 \equiv 10$, $2^6 \equiv 9$, $2^7 \equiv 7$, $2^8 \equiv 3$, $2^9 \equiv 6$, $2^{10} \equiv 1$, confirms that 2 is a primitive root modulo 11.)

(7) Runtime of the parity test

The test is a single modular exponentiation, $h^{(p-1)/2} \bmod p$, with an exponent of about $\log_2 p$ bits. By successive squaring this takes $O(\log p)$ multiplications modulo p, each costing $O((\log p)^2)$ bit operations, for a total of $O((\log p)^3)$ bit operations. That is polynomial in the input size $\log_2 p$, so the parity of a discrete log is easy to find, even though the discrete log itself is (we believe) hard. Compare baby-step giant-step, which needs about $\sqrt{p}$ operations, exponential in $\log_2 p$.

(8) Is the parity of a discrete log well-defined modulo n?

The discrete log x is only defined modulo the order of g; for a primitive root modulo p that means modulo $p-1$. Two exponents x and $x + (p-1)$ represent the same discrete log, and they have the same parity exactly because $p-1$ is *even* (adding an even number does not change parity). So "even or odd" is a well-defined property of the residue class x mod $(p-1)$, and this is why the test makes sense for odd primes p.

In general, parity of an exponent modulo n is well-defined only when n is even. If the order of g were odd, say g^x with x defined modulo an odd n, then x and $x+n$ would have opposite parities while being the same discrete log, so "the parity of x" would be meaningless. (For $p = 2$ the group of units is trivial and there is nothing to ask.)

Problem 4: Baby-Step Giant-Step for $51 = 2^x \pmod{83}$

Here $p = 83$, $g = 2$ and $h = 51$; 2 is a primitive root modulo 83, so x is determined modulo $p - 1 = 82$.

Step size. Take $N = \lceil \sqrt{p-1}\, \rceil = \lceil \sqrt{82}\, \rceil = 10$, since $9^2 = 81 < 82 \le 100 = 10^2$. Every x with $0 \le x < 100$ (in particular every x with $0 \le x < 82$) can be written as $x = j + 10k$ with $0 \le j \le 9$ and $0 \le k \le 9$; we list j, k from 0 to 10 for safety.

Baby steps $g^j = 2^j \bmod 83$: each entry is twice the previous one, reduced modulo 83.

j	0	1	2	3	4	5	6	7	8	9	10
$2^j \bmod 83$	1	2	4	8	16	32	**64**	45	7	14	28

For instance $2^7 = 128 = 83 + 45 \equiv 45$, and $2^{10} = 2 \cdot 14 = 28$.

The giant-step multiplier. We need $g^{-N} = 2^{-10} \pmod{83}$. From the baby list $2^{10} \equiv 28$, so $2^{-10} \equiv 28^{-1} \pmod{83}$, and $28 \cdot 3 = 84 \equiv 1$, so

$$g^{-N} \equiv 3 \pmod{83}.$$

Giant steps $h \cdot g^{-Nk} = 51 \cdot 3^k \bmod 83$: start at $h = 51$ and multiply by 3 repeatedly.

k	0	1	2	3	4	5	6	7	8	9	10
$51 \cdot 3^k \bmod 83$	51	70	44	49	**64**	26	78	68	38	31	10

For instance $51 \cdot 3 = 153 = 83 + 70 \equiv 70$, and $70 \cdot 3 = 210 = 2 \cdot 83 + 44 \equiv 44$.

The collision. The value 64 appears in both lists: as the baby step $j = 6$ and as the giant step $k = 4$. That means

$$2^6 \equiv 51 \cdot 2^{-10 \cdot 4} \pmod{83}, \qquad \text{so} \qquad 51 \equiv 2^{6+40} = 2^{46} \pmod{83}.$$

Hence $x = j + Nk = 6 + 10 \cdot 4 = \mathbf{46}$.

Check. By successive squaring, $2^{46} = 2^{32} \cdot 2^8 \cdot 2^4 \cdot 2^2$. Modulo 83: $2^8 \equiv 7$, $2^{16} \equiv 49$, $2^{32} \equiv 49^2 = 2401 = 28 \cdot 83 + 77 \equiv 77$. Then $2^{46} \equiv 77 \cdot 7 \cdot 16 \cdot 4$: $77 \cdot 7 = 539 = 6 \cdot 83 + 41 \equiv 41$, $41 \cdot 16 = 656 = 7 \cdot 83 + 75 \equiv 75$, $75 \cdot 4 = 300 = 3 \cdot 83 + 51 \equiv 51$. ✓

Problem 5: $2^{(3^{2026})} \pmod{33}$

Outer step. The base 2 is coprime to 33, so Euler's theorem applies: $2^{\varphi(33)} \equiv 1 \pmod{33}$ with $\varphi(33) = \varphi(3)\varphi(11) = 2 \cdot 10 = 20$. Therefore the exponent 3^{2026} may be reduced modulo 20:

$$2^{(3^{2026})} \equiv 2^{(3^{2026} \bmod 20)} \pmod{33}.$$

Inner step. To find $3^{2026} \bmod 20$, note that 3 is coprime to 20, so Euler's theorem applies again with $\varphi(20) = \varphi(4)\varphi(5) = 2 \cdot 4 = 8$: the exponent 2026 may be reduced modulo 8. Since $2026 = 253 \cdot 8 + 2$,

$$3^{2026} \equiv 3^2 = 9 \pmod{20}.$$

Finish. So $2^{(3^{2026})} \equiv 2^9 = 512 \pmod{33}$, and $512 = 15 \cdot 33 + 17$. Therefore

$$2^{(3^{2026})} \equiv \mathbf{17} \pmod{33}.$$

(Both coprimality conditions were checked: $\gcd(2, 33) = 1$ for the outer step and $\gcd(3, 20) = 1$ for the inner step. A computer confirms the answer 17.)

Problem 6: linear search versus binary search on the list $i^3 + i$

The input is the pair of integers N and t.

(1) Input size

N takes about $\log_2 N$ bits. For the problem to be interesting, t is at most about the last entry $N^3 + N$, so t takes at most about $\log_2(N^3 + N) \approx 3\log_2 N$ bits (a larger t can be rejected immediately). So the input size is

$$n \approx \log_2 N + 3\log_2 N = 4\log_2 N \text{ bits}, \qquad \text{that is, } n = O(\log N).$$

The essential point is that the input is only $O(\log N)$ bits long, even though the (implicit) list has N entries: the list is not part of the input.

(2) Runtimes

Each entry $i^3 + i$ is computed from an integer i of at most $\log_2 N$ bits: two multiplications and an addition of numbers of at most $3\log_2 N$ bits, costing $O((\log N)^2)$ bit operations with schoolbook multiplication. Comparing the entry with t costs a number of bit operations proportional to their length, $O(\log N)$. So one "compute and compare" step costs $O((\log N)^2)$ bit operations.

Linear search. In the worst case (for example t is the last entry, or is just past it) the algorithm computes and compares all N entries, for a total of

$$O\big(N(\log N)^2\big) \text{ bit operations.}$$

In terms of the input size $n \approx 4\log_2 N$, we have $N \approx 2^{n/4}$, so this is

$$O\big(2^{n/4}\, n^2\big) :$$

exponential in the input size.

Binary search. Each round computes one middle entry, compares it with t, and halves the range of indices, so there are at most $\lceil \log_2 N \rceil + 1 = O(\log N)$ rounds. (Because the entries $i^3 + i$ are increasing in i, discarding half the range is legitimate.) The total is

$$O\big(\log N \cdot (\log N)^2\big) = O\big((\log N)^3\big) = O(n^3)$$

bit operations: **polynomial** in the input size.

Moral. Measured against the actual input, a length-N list is exponentially large, so scanning it is an exponential-time algorithm, while binary search, which only ever looks at $O(\log N)$ entries, is polynomial. This is the same phenomenon as in the discrete logarithm problem: trying all exponents $x = 1, 2, \ldots, p-1$ is exponential in the input size $\log_2 p$, and the whole game is to find algorithms that avoid looking at most of the list.