Advanced Complexity Exam 2019

All written documents allowed. No Internet access, no cell phone.

1 The class $\mathbf{P}/\log$

A language L is in the class $\mathbf{P}/\log$ if and only if there is a family of so-called *advice words* $(adv_n)_{n\in\mathbb{N}}$, where adv_n is of size $O(\log n)$ (and is not necessarily computable), and a polynomial time Turing machine $\mathcal{M}$, such that for every input x of size n, $x \in L$ if and only if $\mathcal{M}(x, adv_n)$ accepts.

1. Show that $\mathbf{P}/\log$ contains undecidable languages.

   As for $\mathbf{P}$/poly, fix an undecidable language H (say, the halting problem) of natural numbers. Let $L = \{1^n \mid n \in H\}$. This is again undecidable, otherwise H would be easily decided. The advice string adv_n is 1 if $n \in H$, 0 otherwise (constant size !), and $\mathcal{M}(x, adv_n)$ accepts if and only if x consists of all ones (that takes polynomial time !) and $adv_n = 1$.

   Alternative solution : the same argument shows that all unary languages are in $\mathbf{P}/\log$, but there are uncountably many unary languages and only countably many decidable languages.

2. In this question, we assume that $\mathbf{NP} \subseteq \mathbf{P}/\log$. Imagine we are given a language L in $\mathbf{NP}$: there is a language $D \in \mathbf{P}$ and a polynomial $p(n)$ such that $x \in L$ if and only there is a string y of size $p(n)$ (exactly) such that $(x, y) \in D$. As usual, n is the size of x. We assume y written on the alphabet $\{0, 1\}$. Using a self-reducibility argument, describe a polynomial time algorithm which :
   - on input x (of size n) and a (of size $O(\log n)$), computes a string y of size $p(n)$;
   - for some well-chosen advice string a, depending only on n and not on x, for every $x \in L$, the obtained string y satisfies $(x, y) \in D$.

   This will involve the creation of another language L' in $\mathbf{NP}$, of which a will be the advice string for inputs of a suitable size. Please describe L' explicitly.

   We let L' be the language of triples $(x, y', 1^{p(n)-|y'|})$ where y' is a prefix of some string y of size $p(n)$ such that $(x, y) \in D$. This is in $\mathbf{NP}$, hence in $\mathbf{P}/\log$. Let adv_m be advice strings ($m \in \mathbb{N}$) for that language, and $\mathcal{M}$ be the corresponding polynomial time Turing machine with advice.

We are taking triples, not pairs (x, y') so that all the triples have the same size $m = n + p(n) + \text{constant}$. The well-chosen a will be adv_m.

The algorithm is a (tail-)recursive one : $f(y')$ will return a string y of size $p(n)$ of which y' is a prefix (provided y' has size at most $p(n)$) such that $(x, y) \in D$, provided $x \in L$; otherwise it returns any string of size $p(n)$. Then we call $f(\epsilon)$. Note that if $x \in L$, then there is such a y extending ϵ.

The algorithm $f(y')$ works as follows. If y' already has size $p(n)$, then we return y'. Otherwise, we test whether $\mathcal{M}((x, y'0, 1^{p(n)-|y'0|}), a)$ accepts, in polynomial time (if $a = adv_m$, this tests whether $(x, y'0, 1^{p(n)-|y'0|})$ is in L'). If so, we return $f(y'0)$, otherwise we return $f(y'1)$.

3. Show that if $\mathbf{NP} \subseteq \mathbf{P}/\log$, then $\mathbf{P} = \mathbf{NP}$. Hint : can we enumerate the possible advice strings ?

The inequality $\mathbf{P} \subseteq \mathbf{NP}$ is obvious, let us prove the converse. Let L be a language in $\mathbf{NP}$. We build L' as in the previous question, and reuse the same notations. That has logarithmic-sized advices string, say of size $k \log n$.

On input x of size n, we enumerate all the possible advice strings a of size $k \log n$ and run the algorithm of the previous question, obtaining a string y. If $(x, y) \in D$, then we accept, otherwise we go the next a. If we have gone through all possible strings a without accepting, we reject.

This only takes polynomial time, because there are at most $2^{k \log n} = n^k$ possible strings a.

If $x \in L$, then we will eventually come across $a = adv_m$, and then (x, y) will be in D, so the algorithm accepts.

If $x \notin L$, then whatever y is computed at each turn of the loop, (x, y) cannot be in D, so we will eventually reject.

2 Finding prime numbers

The various known ways Arthur can test whether a number p is prime are collected in the Appendix. In this part, we show that one can find prime numbers under scarce resources. All numbers are represented in binary.

We will admit that all arithmetic operations $+$, $-$, $\times$ modulo an m-bit prime p take polynomial time in m, and $O(m)$ space.

4. Given a prime number p, show how Arthur can draw a number x at random uniformly between 1 and $p-1$, in *average* time $O(\log p)$ and using an average number of $O(\log p)$ random bits. (In particular, Arthur terminates with probability one.)

Arthur draws a random sequence of $|p|$ bits, interprets it as a number x, and loops until $1 \leq x < p$. This terminates in an average of less than 2 loop iterations, hence in at most $2 \log p$ time (plus the time to check $1 \leq x < p$, which is also $O(\log p)$) and using at most $2 \log p$ random bits on average.

Stopping the loop after any number of iterations would bias the distribution, so we cannot (easily) make the time and number of random bits be $O(\log p)$ in the worst case.

5. Arthur wishes to obtain a prime number of length m, where m is some parameter. Recall that such prime numbers exist, by Bertrand's postulate (explicitly : there are at least $2^{m-1}/(3m)$ prime numbers of length m). How does he do it algorithmically, in such a way that he uses only $O(m)$ average space and $O(m)$ average random bits ? Arthur must terminate with probability 1, may return a non-prime number with small probability (say, less than some fixed $\epsilon > 0$), and may collaborate with Merlin, using polynomially many rounds, with a constant-space communication tape between Merlin and Arthur. (You are not forced to use all these options.)

   Arthur may ask Merlin for such a prime p. Arthur then checks that its length is exactly m. It remains to check that p is prime, then.

   The Kayal-Agrawal-Saxena test is unusable, since it would run in time $O(m^6)$, hence also in space polynomial in $O(m^6)$. This is not $O(m)$, hence is the not the right way to do it.

   Pratt's test is unusable as well, since Arthur would need to store the list of prime divisors of $p-1$ provided by Merlin, and we have no guarantee that it can be stored in space $O(m)$.

   One can use the Solovay-Strassen test, but one should beware to implement the computation of the Jacobi symbol $\left(\frac{x}{p}\right)$ iteratively instead of recursively, to save stack space. (This is tail-recursion, not general recursion.) The space used is linear in $|p| = O(m)$, it uses logarithmically many random bits (those used to draw x), but may fail with probability at most $1/2$ in case p is not prime.

   The simplest is for Arthur to use the Rabin-Miller test. This only requires space of the order of $|p| = O(m)$, but one has to be careful how we compute modular exponentiations. This is by repeated squaring, and we note that the bit-pattern of p can itself be used as a stack. (Alternatively, we can write the repeated squaring algorithm iteratively.) We do not store the successive values of $x^{2^j q}$, but compute them one after the other. This is logarithmic space, uses only logarithmically many random bits (those used to draw x), but may fail with probability at most $1/4$ in case p is not prime.

   We iterate this procedure a constant number of rounds to reach an error bounded by ϵ. The bounds on time and number of random bits are only in the average case rather than the worst case, because of our need to draw x at random (see Question 4).

   A variant would be for Arthur to draw numbers of length m at random until one is prime, instead of asking Merlin. Bertrand's postulate is not enough to justify that this will terminate in a constant number of rounds on average. In fact, it does not, since the number of prime numbers of length m is about $2^{m-1}/(3m)$, meaning that the probability of finding one in one round is about $1/(3m)$. A simple computation

then shows that the average number of rounds N is $3m$. The space used is still $O(m)$, but the average number of random bits is $O(m).N = O(m^2)$.

3 Lipton's Fingerprint

Let $[x_1, x_2, \ldots, x_k]$ and $[y_1, y_2, \ldots, y_{k'}]$ be two lists of integers between 0 and $p-1$, with $k, k' < p$. We will also admit without proof that these two lists are equal up to permutation if and only if the polynomial :

$$\prod_{i=1}^{k}(X - x_i) - \prod_{j=1}^{k'}(X - y_j) \tag{1}$$

in $\mathbb{Z}/p\mathbb{Z}[X]$ is the zero polynomial.

6. Deduce that the following *Lipton protocol* correctly decides whether two lists $[x_1, x_2, \ldots, x_k]$ and $[y_1, y_2, \ldots, y_{k'}]$ of natural numbers between 1 and p (p prime) are permutations of each other with high probability : draw r at random uniformly between 0 and $p-1$, and compare the *fingerprints* $\prod_{i=1}^{k}(r - x_i) \bmod p$ of the first list and $\prod_{j=1}^{k'}(r - y_j) \bmod p$ of the second list ; if the two fingerprints are equal, then accept the two lists as being the same up to permutation ; otherwise, reject.

   Precisely, you shall show that the Lipton protocol :

   (a) does not err in case the two lists are equal up to permutation ;

   (b) errs with probability at most $\max(k, k')/p$ when they are not ;

   (c) runs in (randomized) space $O(\log p)$ and time $poly(\log p)$;

   (d) only needs $O(\log p)$ random bits on average.

   Errata : in (c), this should be *average* space $O(\log p)$ and *average* time $poly(k, k', \log p)$ (not $poly(\log p)$).

   This is Lagrange interpolation, or the one-dimensional variant of the Schwartz-Zippel Lemma. What we do here is just evaluating (1) on a random value r in $\mathbb{Z}/p\mathbb{Z}$.

   If the two lists are equal up to permutation, then we must get the same fingerprint, and accept (Claim a). Otherwise, (1) is not the zero polynomial, so the probability that we find equal fingerprints is at most d/p, where d is the degree of (1), namely $\max(k, k')$ (Claim b).

   Computing arithmetic operations mod p only requires space $O(\log p)$ and time polynomial in $\log p$. Evaluating each of the two polynomials can be done by keeping just one register which we mutiply by each factor $r - x_i$, hence we still only need space $O(\log p)$ for computing the two polynomials. The time required to evaluate these polynomials is $O((k+k')poly(\log p))$, which is polynomial in k, k' and $\log p$.

Together with our admitted result on the way we can draw r efficiently at random, computing the fingerprints only requires space $O(\log p)$ and time poly$(\log p)$ (Claim c).

Finally, we only need to draw a number r at random between 0 and $p-1$. This is of length $O(\log p)$ bits, hence uses $O(\log p)$ random bits on average.

7. Here is a strange way that Arthur and Merlin could collaborate to show that a 3-SAT formula such as $(x \vee y \vee z) \wedge (\neg w \vee \neg y \vee z) \wedge (w \vee \neg x \vee \neg z)$ is satisfiable. Recall that 3-SAT is **NP**-complete under logspace reductions.

   (a) In phase 1, Merlin sends an assignment of a truth value to each instance of a literal in the formula, *ordered by variable number*. Each single assignment is represented as a triple (L, i, v) where L is the literal, i is the number of the clause it appears in, and $v \in \{T, F\}$ is the truth value. In the example, Merlin might send :

   $$(\neg w, 2, F), (w, 3, T), (x, 1, T), (\neg x, 3, F), (y, 1, F), (\neg y, 2, T), (z, 1, F), (z, 2, F), (\neg z, 3, T)$$

   Arthur then checks that each instance of the literal is given the same value, and that complementary instances of literals are given opposite values.

   (b) In phase 2, Merlin sends an assignment again (which is purportedly the same, up to permutation), this time *in the order where the literals occur* in the formula. In the example, honest Merlin would send :

   $$(x, 1, T), (y, 1, F), (z, 1, F), (\neg w, 2, F), (\neg y, 2, T), (z, 2, F), (w, 3, T), (\neg x, 3, F), (\neg z, 3, T)$$

   Arthur then checks that each clause of the formula is satisfied with this assignment.

   (c) In phase 3, Arthur checks that the assignments given by Merlin in phases 1 and 2 are the same up to permutation.

   Show how you can implement phase 3 with high probability, using only logarithmically many random bits on average. (You can adapt what Arthur does in phases 1 and 2 slightly.) The whole three phases should only require logarithmic space and polynomial time on average from Arthur, and a constant-space communication tape between Merlin and Arthur (but polynomially many rounds). If the 3-SAT formula is satisfiable, then Merlin should have a way of making Arthur accept with probability 1 ; otherwise, whatever Merlin plays, Arthur should not accept with probability larger than 1/5.

   First, in order to simplify the presentation, we will pretend that Merlin can send data in chunks of up to polynomial size. This can be done by letting Merlin each such piece of data bit by bit, in polynomially many rounds.

   Phase 3 should consist for Arthur in checking that the two lists of triples sent by Merlin in phases 1 and 2 are equal up to permutation. To use Lipton fingerprinting, we need :

   — *to encode the triples (L, i, v) as natural numbers ;*

— *to modify phases 1 and 2 so that Arthur computes the fingerprints on-line, without storing the assignments sent by Merlin ; instead, Merlin sends his assignments one letter at a time, and Arthur modifies the current value of the fingerprint at each new triple (L, i, v) received, using only logarithmic space to store the current fingerprint and the first characters to the current triple ; (time is not an issue ;)*

— *we must finally pick the prime p so that the error probabilities turn out to be small enough.*

To encode the triples (L, i, v), we can encode them as bit strings on 1 bit (sign of L) + $\log n$ bits (variable number) + $\log n$ bits (clause number) + 1 bit (true/false). Then we read these $2 + 2\log n$-bit strings as binary numbers $\ulcorner L, i, v \urcorner$: so now we have lists of natural numbers, on which we can apply Lipton's fingerprinting technique.

To keep the space used by Arthur logarithmic, we must separate each triple by some fixed letter, say #, and reject Merlin's output if we ever see more than $2 + 2\log n$ bits without a # ; and we must compute our fingerprints incrementally, without storing all triples :

— *Before phase 1, Arthur must find a large enough prime p (we shall see how large below) ; this may involve Merlin's help ; next, Arthur draws r at random uniformly between 0 and $p - 1$.*

— *In phase 1, Arthur initializes the first fingerprint to $f_1 := 1$, and initializes $t := \bot$; each time Arthur reads a new triple (L, i, v) from Merlin :*

  — *it updates the fingerprint : $f_1 := f_1 \times (r - \ulcorner L, i, v \urcorner) \bmod p$;*

  — *if t is of the form (L, j, w) for some j, w, then Arthur checks that $v = w$ (else rejects) ; if t is of the form $(\neg L, j, w)$ for some j, w, then Arthur checks that v is the negation of w (else rejects) ; if t is of the form (L', j, w) for $L' \neq L, \neg L$, then Arthur checks that the number of the propositional variable in L is one plus the number of that in L' exactly : this is the phase 1 consistency check ;*

  — *Arthur sets $t := (L, i, v)$.*

  Let k be the number of (occurrences of) literals in the input clauses. We also make sure that Arthur reads at most k triples from Merlin. This is needed in the complexity analysis below, and can be implemented by letting Arthur read the input, and ask Merlin about a new triple only once Arthur has parsed a new literal.

— *In phase 2, Arthur initializes the second fingerprint to $f_2 := 1$, sets a clause counter to $c := 1$, and starts reading the clauses in the input, from left to right :*

  — *For each clause, say $L_1 \vee L_2 \vee L_3$, Arthur expects Merlin to send three triples (L_1, c, v_1), (L_2, c, v_2), (L_3, c, v_3) (with the middle number equal to*

c, and the first numbers matching the literals in the clause) ; Arthur stores all three, and checking that one of v_1, v_2, v_3 at least is true ; else Arthur rejects ; this checks that all clauses are made true by Merlin's assignment ;

— *after the clause is read,* $c := c + 1$, $f_2 := f_2 \times (r - \ulcorner L_1, c_1, v_1 \urcorner)(r - \ulcorner L_2, c_2, v_2 \urcorner)(r - \ulcorner L_3, c_3, v_3 \urcorner)$ *mod* p.

— *Finally, Arthur accepts if* $f_1 = f_2$, *and rejects otherwise.*

If the input formula is satisfiable, then Merlin has an obvious winning strategy. Otherwise, Merlin must cheat. We must evaluate the probability that it cheats without getting caught : this involves sending two lists of (encodings) of triples (L, i, v), *one that passes phase 1 consistency and one that satisfies all clauses. These two lists cannot be permutations of each other, since the formula is unsatisfiable. The above protocol makes sure that the two lists have* k *elements, where* k *is the number of (occurrences of) literals in the input clause set. By Question 6, the two lists will have equal fingerprints with probability at most* k/p.

We now need k/p *to be at most* $1/5$. *However, we shall also make errors in checking whether* p *is prime, so we add a little slop. We require* k/p *to be at most* $1/6$ *instead. Now* k *is at most the input size* n, *so we have to find* p *prime* $\geq 6n$ *(and of size* $O(\log n)$*). We use Question 5 for that with* $m = \lceil \log(6n) \rceil$, *and error* ϵ *at most* $1/5 - 1/6 = 1/30$.

8. Show that the protocol of Question 7 can be improved so that Arthur only uses logarithmically many random bits *in the worst case*, while keeping the error probability to at most $1/4$.

As in the **ZPP** = **RP** ∩ **coRP** *proof, replace all the rejection sampling loops (of* r, *of testing* p *for primality) by loops with a fixed upper bound of* t *loop turns, where* t *will be obtained later. The point is that the probability of hitting that upper bound should be at most* $1/40$ *for each of* r *and* p.

Each loop succeeds with some constant probability ($3/4$ *with Miller-Rabin and* $1/2$ *for Solovay-Strassen, for* p *; at least* $1/2$ *for* r *; in any case at least* $1/2$*), so the probability that we would hit the upper bound on the number of loop turns is at most* $1/2^t$. *We fix* t *so that this is at most* $1/40$ *:* $t = 6$ *is fine.*

Alternatively, we use Markov's inequality. Since each number is obtained in some time T *on average, we fix a timeout of* $40T$, *so that again the probability of hitting the timeout, both for* r *and* p, *is at most* $1/40$. *(This is less efficient.) We can also put a timeout on the combined computation of* r *and* p, *of* 20 *times the average time needed to compute both.*

If the timeout is exceeded, then let Arthur just accept (right away, without proceeding with the rest of the computation). This does not change anything in case the input formula is satisfiable, otherwise the probability of making a mistake is at most $1/5$ *plus twice* $1/40$ *(twice because we have two rejection sampling loops), namely* $1/5 + 1/20 = 1/4$.

9. Conclude that **NP** is included in the class **IP**(logspace, lograndbits) of interactive proofs with polynomially many rounds, restricted so that Arthur can use only logarithmic space and polynomial time, communication between Arthur and Merlin goes through a constant-size communication tape, and Arthur only draws a logarithmic amount of random bits.

   Erratum : I had forgotten to say that the error probability of that class is taken to be at most $1/2$ (instead of $1/2^{n^\ell}$ as in the lecture notes).

   First, we cannot repeat the protocol of the previous question n^ℓ times. This would indeed let us reach an error probability of at most $1/2^{n^\ell}$ but we would then need much more than a logarithmic number of random bits. Hence the erratum.

   We have shown that 3-SAT was in **IP**(*logspace*, *lograndbits*)*. To conclude, it suffices to show that* **IP**(*logspace*, *lograndbits*) *is closed under logspace reductions. This is done exactly as we have shown that logspace reductions compose, or that* **NL** *is closed under logspace reductions : Arthur simulates the logspace reduction by simulating the whole logspace reduction each times it needs one letter from its output, keeping only the letter at the desired position in memory during the simulation.*

   We even obtain an error bound of $1/4$ instead of the desired $1/2$.

4 Strategies

10. Given an **IP** protocol between Arthur and Merlin (with polynomially many rounds, say $p(n)$), a strategy for Merlin (on input x) is just a map from all public histories that can be played (starting from input x) to Merlin's answers. Show that, if Arthur uses only $O(\log n)$ random bits in the **IP** protocol and communication between Arthur and Merlin goes through a constant-size communication tape, then there is a concise way of representing Merlin's strategy on input x. By *concise*, we mean of polynomial size.

    On input x, any two plays played with the same strategy from Merlin and with the same sequences of random bits must be equal. Assume Arthur only uses $k \log n$ random bits. This makes at most $2^{k \log n} = n^k$ possible sequences of random bits, hence at most n^k different possible histories. Merlin only has to store up to $p(n)$ answers to the $p(n)$ questions asked in the run obtained by fixing the random bits. By building a table indexed by (r, i), where r is the sequence of random bits and i is the question number, Merlin only has to store $p(n)n^k$ (constant-size) answers. This is certainly of polynomial size.

11. Show that any language L in **IP**(logspace, lograndbits) has an **MA** protocol (an Arthur-Merlin protocol where Merlin plays first, Arthur plays second and there is no other round), with resource constraints as in the lectures, except that Arthur uses only logarithmically many random bits ; and where the error probability is at most $1/4$. We recall that, although the protocols of Section 3 had one-sided error, **IP**(logspace, lograndbits)

is meant to have two-sided error (Arthur may make mistakes when $x \notin L$, but also when $x \in L$.)

> *Given an* $\mathbf{IP}(\mathit{logspace}, \mathit{lograndbits})$ *protocol* π *for* L*, we build the claimed* **MA** *protocol by letting Merlin output a polynomial-size strategy first. Then Arthur will play* π *alone, simulating Merlin's moves in* π *by consulting the strategy instead.*
>
> *If* $x \in L$*, then Merlin had a way of winning* π *with probability at least* $3/4$*. Playing the corresponding strategy, Arthur will necessarily accept with probability at least* $3/4$ *as well.*
>
> *If* $x \notin L$*, then whatever the polynomial-sized strategy Merlin plays in the* **MA** *protocol, playing* π *by letting Merlin use the published strategy gets him no better chance of winning than by any strategy, i.e., Merlin wins with probability at most* $1/4$*.*
>
> *Notice finally that it does not matter whether Arthur's coins are public (as required here) or private (as given in the original protocol* π*), because Merlin will never play after Arthur.*

12. Show that $\mathbf{IP}(\text{logspace}, \text{lograndbits}) \subseteq \mathbf{NP}$.

> *We derandomize Arthur's computations in the protocol of Question 11. Since Arthur has only* $k \log n$ *random bits, we can instead simulate all* $2^{k \log n} = n^k$ *possible runs of Arthur, by enumerating all* $k \log n$*-bit random bit strings, and counting how many lead Arthur to accept. Either at least* $3/4$ *of them will lead to acceptance, and we accept; or at most* $1/4$ *lead to acceptance, and we reject.*
>
> *This computation only takes polynomial time by Arthur. So we obtain an* **MA** *protocol where Merlin plays first, and Arthur decides in polynomial time. This is an* $\mathbf{M} = \mathbf{NP}$ *protocol.*

To sum up, we obtain the *Condon-Ladner Theorem* : $\mathbf{IP}(\text{logspace}, \text{lograndbits}) = \mathbf{NP}$.

A Primality testing

Here are the known ways Arthur can test whether a number p is prime. (Write $|p|$ for the size of p in binary.) We always assume that p is odd in the sequel.

1. Primality is in $\mathbf{P}$: Agrawal, Kayal, and Saxena's algorithm runs in time $O(|p|^6)$;
2. by a result of V. Pratt, primality is in $\mathbf{NP}$, so one can ask Merlin to provide a proof that p is prime and check it in polynomial time ; precisely, Merlin provides a number x between 1 and $p-1$, Arthur checks that the gcd of x and p equals 1, that $x^{p-1} = 1 \mod p$, and that for every prime factor of $p-1$, $x^{(p-1)/q} \neq 1 \mod p$. The "for every prime factor of $p-1$" part is checked by asking Merlin for a candidate list of prime factors, checking that their product equals $p-1$, and doing a recursive call to check that each element of the list is indeed prime. The total size of the numbers provided by Merlin is guaranteed to be polynomial in $|p|$.

3. the Miller-Rabin test : Arthur draws x at random uniformly between 1 and $p-1$, and (assuming p is odd, so $p = 1 + 2^k q$ with q odd) check that $x^q = 1 \bmod p$ or $x^{2^j q} = -1 \bmod p$ for some j between 1 and k (if p is prime, this must succeed always ; otherwise, it must fail with probability at least $3/4$) ;
4. the Solovay-Strassen test : Arthur draws x at random uniformly between 1 and $p-1$, and checks that $x^{(p-1)/2} = \left(\frac{x}{p}\right) \bmod p$, where the Jacobi symbol $\left(\frac{x}{p}\right)$ can be computed by the recursive equations (for n odd) :
   - $\left(\frac{m}{n}\right) = \left(\frac{m \bmod n}{n}\right)$; this is used when $m \geq n$
   - $\left(\frac{1}{n}\right) = 1$
   - $\left(\frac{2m}{n}\right) = \left(\frac{m}{n}\right)$ if n equals 1 or 7 mod 8, $\left(\frac{2m}{n}\right) = -\left(\frac{m}{n}\right)$ if n equals 3 or 5 mod 8
   - $\left(\frac{m}{n}\right) = \left(\frac{n}{m}\right)$ if m is odd and n or m equals 1 mod 4, $\left(\frac{m}{n}\right) = -\left(\frac{n}{m}\right)$ if m is odd and $n = m = 3 \bmod 4$; this is used when m is odd and $m < n$.

   If p is prime, this always succeeds ; otherwise, it must fail with probability at least $1/2$.