### Primality Testing

**Fermat’s Theorem:** If \( a \in \mathbb{Z}_p^+ \) (\( p \) prime), then \( a^{p-1} \equiv 1 \pmod{p} \)

Corollary: if \( a < n \), and \( a^{n-1} \not\equiv 1 \pmod{n} \), then \( n \) is not prime.

**Fermat test:** \( n \) passes test at \( a \) if \( a^{n-1} \equiv 1 \pmod{n} \)

Pseudoprime: \( n \) passes Fermat test for all \( 0 < a < n \). There are non-prime numbers that pass all tests (Carmichael numbers)

Can prove: if \( p \) is not pseudoprime, it fails at least half the tests. \( \implies \) run for \( k \) values \( a_i \), error bound \( 2^{-k} \).

### Miller-Rabin Primality Testing: Square Roots of 1

1 only has square roots -1 and 1 modulo any prime \( p \).

\( p \) prime candidate \( \implies \) odd \( \implies \) check \( a^{\frac{p-1}{2}} \)

if \( 0 \neq 1 \), \( p \) is not prime.

If 1, can keep dividing exponent as long as even.

Write \( p-1 = s \cdot 2^t \).

Choose \( a < p \).

If \( a^{p-1} \not\equiv 1 \pmod{p} \), reject (\( p \) not prime)

Compute sequence \( a^2, a^2, \ldots, a^2 \pmod{p} \)

If some element \( \not\equiv 1 \), and last such element \( \neq -1 \), reject

Repeat for \( k \) different values \( a \implies \) error probability \( \leq 2^{-k} \)

**Theorem:** PRIMES is in BPP.

---

### Probabilistic Turing Machines

**Def:** Nondeterministic TM with

- two branches at each nondeterminism step
- equal probabilities for both (coin flip) \( \implies \Pr[b] = 2^{-k} \) for branch \( b \) with \( k \) coin flips

Acceptance: \( \Pr[M \text{ accepts } w] = \sum_{\text{accepting } b} \Pr[b] \)

\( \Pr[M \text{ rejects } w] = 1 - \Pr[M \text{ accepts } w] \)

\( M \) decides language \( A \) with error probability \( \epsilon \) if

- \( w \in A \implies \Pr[M \text{ accepts } w] \geq 1 - \epsilon \)
- \( w \not\in A \implies \Pr[M \text{ rejects } w] \geq 1 - \epsilon \)

### BPP (Bounded Error Probabilistic Polynomial Time)

**BPP** is the class of languages decided by probabilistic polynomial Turing machines with error probability 1/3

Choice of 1/3 is arbitrary. Anything in \((0, 1/2)\) works.

We can make the error probability arbitrarily small.

**Amplification Lemma.** For any polynomial \( p(n) \), a PPTM \( M_1 \) with error probability \( < 1/2 \) has an equivalent \( M_2 \) with err. prob. \( 2^{-p(n)} = \text{arbitrary polynomial exponent, still in polynomial time} \)

How: \( M_2 \) simulates \( M_1 \) \( 2k \) times, takes majority decision.

Max. error prob. on one run: \( k \) right, \( k \) wrong, \( \epsilon^k (1-\epsilon)^k \).

Over all \( 2^k \) result sequences: \( 2^k \epsilon^k (1-\epsilon)^k = (4\epsilon(1-\epsilon))^k \).

We want \( (4\epsilon(1-\epsilon))^k \leq 2^{-p(n)} \). \( \implies k = -p(n)/\log(4\epsilon(1-\epsilon)) \)
Primality Testing: Proof

▶ If number is rejected, it’s composite
  ▶ by Fermat’s theorem, or
  ▶ $b^2 \equiv 1 \pmod{p} \implies (b+1)(b-1) = cp$
▶ If number is accepted, very likely prime
Each composite non-witness has a unique corresponding witness

The Class RP

Our test had one-sided error
  ▶ accept means likely prime
  ▶ reject means surely composite

Def. RP is the class of languages decided by PPTM, where strings in the language are accepted with probability $\geq 1/2$, and strings not in the language are rejected with probability 1.

We showed $COMPOSITES \in RP$
Another primality test (Adleman-Huang) always rejects composites, and accepts primes with $Pr \geq 1/2$, so $PRIMES \in RP$

Branching Programs

Take Boolean decision tree, merge equivalent nodes $\implies$ DAG

Def. Branching Program
  ▶ DAG, nodes are query nodes or two outputs, 0 and 1
  ▶ query nodes labeled by variables, 2 outgoing edges (yes/no)
  ▶ designated root (start) node (for evaluation)
Branching programs define the class L (can build poly-size branching program deciding any language in L)
read-once branching program: on each path from root to 0/1, a variable is queried at most once (not redundant)

$EQ_{ROBP} = \{\langle B_1, B_2 \rangle \mid B_1$ and $B_2$ are equivalent read-once branching programs}$

$EQ_{ROBP}$ is in BPP

Can’t randomly choose boolean vectors, since programs might differ only on one vector of $2^m$
Instead, compute polynomial, starting with 1 at root:
  ▶ YES branch on $x_i$: multiply with $x_i$
  ▶ NO branch on $x_i$: multiply with $(1 - x_i)$
For boolean vector inputs, value is always 0 or 1.
Take product going down on each branch.
Sum all branches entering 1 as the result polynomial.
Equivalent programs $\implies$ equivalent polynomials
If polynomials non-equivalent, probability of difference zero is at most $md/f$, for $m$ variables, each of degree $\leq d$, over field of size $f$.
$\implies$ Since degree of each variable is 1, choose $f > 3m$. 