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 \( a^{p-1} \mod p \)
- if \( \pm 1 \), 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 \( a^{p-1} \equiv 1 \mod p \), reject (p not prime)
- Compute sequence \( a^{2^t}, a^{2^t}, a^{2^t} \mod p \)
- If some element \( \neq 1 \), and last such element \( \neq -1 \), reject
- Repeat for k different values a \( \implies \) error probabiulity \( \leq 2^{-k} \)

Theorem: PRIMES is in BPP.
Primality Testing: Proof

- If number is rejected, it's composite
  - by Fermat's theorem, or
  - if number is accepted, very likely prime

Every 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 \( \geq \frac{1}{2} \), and strings not in the language are rejected with probability 1.

We showed \( \text{COMPOSITES} \in \text{RP} \)

Another primality test (Adleman-Huang) always rejects composites, and accepts primes with \( \Pr \geq \frac{1}{2} \), so \( \text{PRIMES} \in \text{RP} \)

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)

\[ E_{\text{ROBP}} = \{ \langle B_1, B_2 \rangle \mid B_1 \text{ and } B_2 \text{ are equivalent read-once branching programs} \} \]

\( E_{\text{ROBP}} \) 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 \( \frac{md}{f} \), for \( m \) variables, each of degree \( \leq d \), over field of size \( f \).

\[ \implies \text{Since degree of each variable is 1, choose } f > 3m. \]