

## 1 Randomized Algorithms

We have seen a few randomized algorithms, such as *Finding the Median* and *Randomized Quicksort*. While it is still debatable whether real randomness exists, for our purposes, we assume the existence of a primitive that generates a random bit (0 or 1 with probability 0.5) in unit time.

## 2 Probabilistic Turing Machine (PTM)

A *Probabilistic Turing Machine (PTM)* is an extension of a standard Turing machine with two transition functions:  $\delta_0$  and  $\delta_1$ . At each computational step, the machine randomly chooses between  $\delta_0$  and  $\delta_1$  with equal probability (i.e., each with probability  $\frac{1}{2}$ ).

The input string is considered accepted if the machine reaches an accepting state with a high probability after multiple steps of computation.

## 3 Equivalent definition of a PTM in terms of probability over a given random string $r$

An equivalent way to define a *Probabilistic Turing Machine (PTM)* is by viewing it as a deterministic Turing machine  $M(x, r)$ , where  $x$  is the input and  $r$  is a random string, representing the probabilistic choices made by the machine.

For an input  $x$ , the PTM accepts if the deterministic machine  $M(x, r)$  accepts for a large fraction of possible random strings  $r$ . Formally, the input string  $x$  is accepted with high probability if:

$$\Pr_r[M(x, r) = 1] \geq 1 - \epsilon$$

for some small  $\epsilon > 0$ , where  $\Pr_r$  denotes the probability over the random choices of  $r$ .

In other words, the PTM simulates the behavior of a deterministic machine  $M(x, r)$  for various random strings  $r$ , and the input is accepted if  $M(x, r)$  accepts for most values of  $r$ .

## 4 Bounded-error Probabilistic Polynomial time (BPP)

A language  $L$  is said to be in the class *BPP* if there exists a probabilistic Turing machine (PTM)  $M$  that runs in polynomial time (on all possible paths) such that for all inputs  $x$ :

$$\Pr[M(x) = L(x)] \geq \frac{2}{3}.$$

This can be interpreted as:

$$x \in L \Rightarrow \Pr[M(x) = 1] \geq \frac{2}{3} \quad \text{and} \quad x \notin L \Rightarrow \Pr[M(x) = 1] \leq \frac{1}{3}$$

Here, polynomial time means that the expected running time of  $M$  is bounded by a polynomial in the size of the input. If necessary, we can artificially stop the machine after a polynomial number of steps.

Alternatively, BPP can be defined in terms of a deterministic Turing machine  $M$  with input  $x$ , a random string  $r$ , and a polynomial function  $q(|x|)$ , which bounds the length of the random string. Specifically, for all inputs  $x$ :

$$x \in L \Rightarrow \Pr_{r \in \{0,1\}^{q(|x|)}} [M(x, r) = 1] \geq \frac{2}{3}$$

$$x \notin L \Rightarrow \Pr_{r \in \{0,1\}^{q(|x|)}} [M(x, r) = 1] \leq \frac{1}{3}$$

In this case,  $r$  is chosen uniformly at random from the set of binary strings of length  $q(|x|)$ , where  $q$  is a polynomial function of the length of  $x$ .

## 5 Homework: Error Bounds in BPP

1. Prove that polynomial time on all inputs  $x$  is equivalent to expected polynomial time for BPP algorithms. Specifically, show that a BPP algorithm, which runs in expected polynomial time, can be artificially stopped after polynomially many steps without significantly affecting its error probability.
2. Demonstrate that the success probability  $\frac{2}{3}$  in the definition of BPP can be replaced with any constant greater than  $\frac{1}{2}$ . Show that by repeating the algorithm multiple times and using the Chernoff bound, the error probability can be made arbitrarily small, effectively amplifying the success probability.
3. Generalize the error bounds  $(\frac{2}{3}, \frac{1}{3})$  in BPP to any pair of constants  $(a, b)$  such that  $a > b$ . Prove that the class BPP remains unchanged when the constants  $a$  and  $b$  satisfy  $a > b$ , as long as  $a - b$  is a constant or even  $1/\text{poly}(n)$ .

## 6 Primality Testing (1970s)

### 6.1 Solovay-Strassen Primality Test

The Solovay-Strassen test is an algorithm for primality testing, with a time complexity of  $O(n^2)$ , where  $n$  is the number of bits required to represent the number being tested. Below is an outline of the method.

First let us try a simple idea. Given a number  $n$  (in binary) and we generate a random number  $a$ , and use Euclid's algorithm to check if  $\text{gcd}(a, n) = 1$ . If GCD is 1, we output prime, otherwise composite. The success probability of this algorithm depends on the probability that we will hit a number that is not coprime with  $n$ . We will argue that this probability can be very low. Assume  $n = p_1 \cdot p_2$ , where  $p_1$  and  $p_2$  are prime factors of  $n$ .

*Coprimality Check:* The number of elements that are not coprime with  $n$  and are greater than 1 is given by:

$$p_1 \cdot p_2 - (p_1 - 1)(p_2 - 1) - 1 = p_1 + p_2 - 2.$$

Note that in worst case  $p_1 + p_2 - 2$  is as small as  $O(\sqrt{n})$ . Thus, the success probability of randomly selecting an  $a$  that is coprime to  $n$  is at most:

$$\frac{\sqrt{n}}{n} = \frac{1}{\sqrt{n}} = \frac{1}{\text{exponential in the size of the input}}.$$

This probability is quite small, which is not ideal for testing primality.

### 6.2 Quadratic Residue and Jacobi Symbol

Solovay-Strassen algorithm uses the concept of a *quadratic residue*. A number  $a$  is called a quadratic residue modulo  $n$  if there exists some  $b$  such that:

$$a \equiv b^2 \pmod{n}.$$

We define the *Jacobi symbol*  $\left(\frac{a}{n}\right)$  as:

$$\left(\frac{a}{n}\right) = \begin{cases} 0, & \text{if } \gcd(a, n) \neq 1, \\ +1, & \text{if } a \equiv b^2 \pmod{n} \text{ for some } b, \\ -1, & \text{otherwise.} \end{cases}$$

**Properties for Prime  $n$ .** Suppose  $n$  is a prime number and  $g$  is a generator of the group of units modulo  $n$  (units means invertible elements. When  $n$  is a prime then every nonzero number has an inverse modulo  $n$ ). The Jacobi symbols follow the pattern:

$$g, g^2, g^3, \dots, g^{n-1} \Rightarrow -1, +1, -1, \dots, +1.$$

This is because,  $g$  cannot be expressed as  $b^2$  for any  $b$ , because if  $g = b^2$ , then:

$$g^{\frac{n-1}{2}} = b^{n-1} \equiv 1 \pmod{n},$$

which implies that  $g$  would have an order of  $\frac{n-1}{2}$ , which contradicts the assumption that its order is  $n-1$ . Thus, a number  $a$  is a quadratic residue modulo  $n$  if and only if:

$$a^{\frac{n-1}{2}} \equiv 1 \pmod{n}.$$

### 6.3 The Solovay-Strassen Primality Test

The primality test proceeds as follows:

1. Choose a random number  $a$ .
2. Compute the Jacobi symbol  $\left(\frac{a}{n}\right)$ .
3. Check if:

$$\left(\frac{a}{n}\right) = a^{\frac{n-1}{2}} \pmod{n}.$$

4. If the equality holds, declare  $n$  as a *probable prime*. If it does not hold, declare  $n$  as *composite*.

This test is correct with high probability, i.e., the likelihood of an incorrect classification is very low. Moreover, the algorithm has one-sided error, i.e., when it says that the number is composite then it is certainly composite.

Before bounding the error probability, first let us see how to compute Jacobi symbol (line 2 in the algorithm) efficiently. Jacobi symbol is computed recursively using the following properties:

- The Jacobi symbol satisfies  $\left(\frac{a}{n}\right) = \left(\frac{a \bmod n}{n}\right)$ .
- The relation between the Jacobi symbols of  $a$  and  $n$  is given by:

$$\left(\frac{a}{n}\right) \cdot \left(\frac{n}{a}\right) = (-1)^{\frac{(a-1)(n-1)}{4}},$$

provided that  $\gcd(a, n) = 1$  and  $a$  and  $n$  are odd.

•

$$\begin{aligned} \left(\frac{2}{n}\right) &= (-1)^{(n^2-1)/8} \\ \left(\frac{ab}{n}\right) &= \left(\frac{a}{n}\right) \cdot \left(\frac{b}{n}\right). \end{aligned}$$

Now we will show that the probability of randomly choosing an  $a$  such that the Jacobi symbol matches  $a^{\frac{n-1}{2}} \pmod{n}$  is at most  $\frac{1}{2}$ , ensuring that the algorithm correctly identifies composite numbers with high probability.

## 6.4 Proof of correctness

**Theorem 17.1.** *If  $n$  is not prime, then the probability that a randomly chosen  $a$  satisfies:*

$$\left(\frac{a}{n}\right) = a^{\frac{n-1}{2}} \pmod{n}$$

*is at most  $\frac{1}{2}$ .*

*Proof.* Let  $n$  be a composite number, and let us analyze the relationship between the Jacobi symbol  $\left(\frac{a}{n}\right)$  and the expression  $a^{\frac{n-1}{2}} \pmod{n}$  for randomly chosen  $a$ .

*Step 1: Understanding the Jacobi Symbol*

Recall that the Jacobi symbol  $\left(\frac{a}{n}\right)$  is a generalization of the Legendre symbol for composite numbers  $n$ . It satisfies the following properties: - If  $\gcd(a, n) \neq 1$ , then  $\left(\frac{a}{n}\right) = 0$ . - If  $\gcd(a, n) = 1$ , then  $\left(\frac{a}{n}\right)$  behaves similarly to the Legendre symbol, which tells us whether  $a$  is a quadratic residue modulo  $n$ .

For prime  $n$ ,  $a^{(n-1)/2} \pmod{n}$  tells us precisely whether  $a$  is quadratic residue or not. However, for composite  $n$ ,  $a^{(n-1)/2} \pmod{n}$  does not necessarily coincide with whether  $a$  is a quadratic residue. This discrepancy is the key to the Solovay-Strassen test.

*Step 2: Comparing  $\left(\frac{a}{n}\right)$  and  $a^{\frac{n-1}{2}} \pmod{n}$*

For prime  $n$ , it is known that:

$$a^{\frac{n-1}{2}} \equiv \left(\frac{a}{n}\right) \pmod{n}.$$

However, when  $n$  is composite, this relationship no longer holds for all  $a$ . Specifically, for composite  $n$ , there exists a large fraction of elements  $a$  for which:

$$a^{\frac{n-1}{2}} \not\equiv \left(\frac{a}{n}\right) \pmod{n}.$$

*Step 3: Probability Analysis*

Clearly, if  $\gcd(a, n) > 1$  then  $\left(\frac{a}{n}\right) = 0$ , but  $a^{(n-1)/2}$  is cannot be zero modulo  $n$  if  $a < n$ . So, we only need to consider numbers which are coprime with  $n$ . For simplicity, we will do the proof only for the case when  $n = pq$  for two primes  $p, q$ . The general case is left as exercise.

First let us argue that there is a number  $a$  such that

$$a^{\frac{n-1}{2}} \not\equiv \left(\frac{a}{n}\right) \pmod{n}. \tag{1}$$

Recall the Chinese remaindering theorem, which states that every number  $0 \leq a < pq$  can be uniquely represented by the tuple  $(a \pmod{p}, a \pmod{q})$ . We know that half the numbers between 1 and  $p-1$  are not quadratic residues modulo  $p$ . Let  $b$  be such a number. That is,  $\left(\frac{b}{p}\right) = -1$ . Let us choose a number  $1 \leq a < pq$  such that

$$a \pmod{p} = b \text{ and } a \pmod{q} = 1.$$

Now, note that

$$\left(\frac{a}{p}\right) = \left(\frac{b}{p}\right) = -1 \text{ and } \left(\frac{a}{q}\right) = \left(\frac{1}{q}\right) = 1.$$

Hence, by the product rule

$$\left(\frac{a}{pq}\right) = \left(\frac{a}{p}\right) \left(\frac{a}{q}\right) = -1.$$

On the other hand,  $a^{(n-1)/2} \equiv 1 \pmod{q}$  and hence  $a^{(n-1)/2} \not\equiv -1 \pmod{q}$ . Thus, we can obtained a number that does not satisfy (1).

We want to show that for a composite  $n$ , the probability that a randomly chosen  $a$  satisfies:

$$\left(\frac{a}{n}\right) = a^{\frac{n-1}{2}} \pmod{n}$$

is at most  $\frac{1}{2}$ .

Let us examine the behavior of the Jacobi symbol for  $a$  modulo  $n$ . By quadratic reciprocity and properties of the Jacobi symbol, we have the following key result:

$$\left(\frac{a}{n}\right) \cdot \left(\frac{n}{a}\right) = (-1)^{\frac{(a-1)(n-1)}{4}}.$$

This relationship holds when  $\gcd(a, n) = 1$  and gives insight into the mismatch between the Jacobi symbol and the modular exponentiation for composite numbers.

When  $n$  is composite, a randomly chosen  $a$  is unlikely to satisfy the equivalence  $\left(\frac{a}{n}\right) = a^{\frac{n-1}{2}} \pmod n$  because the structure of  $\mathbb{Z}_n^*$  (the multiplicative group modulo  $n$ ) is different from the case when  $n$  is prime. In fact, half of the elements in  $\mathbb{Z}_n^*$  will not satisfy this equivalence for composite  $n$ .

*Step 4: Conclusion*

Thus, for composite  $n$ , the probability that a randomly chosen  $a$  satisfies:

$$\left(\frac{a}{n}\right) = a^{\frac{n-1}{2}} \pmod n$$

is at most  $\frac{1}{2}$ .

This concludes the proof. □

## 6.5 Note: Implications of SAT in BPP

*Note:* If SAT (Satisfiability) belongs to BPP, then the Polynomial Hierarchy (PH) collapses to  $\Sigma_2^P$ . This is because BPP is a subset of P/Poly.

*Explanation:*

– P/Poly refers to the class of languages that can be decided by polynomial-time machines with polynomial-sized advice strings.

– It is known that  $\text{BPP} \subseteq \text{P/Poly}$ , which means that any language in BPP can be simulated by a polynomial-time machine with non-uniform advice (polynomial-sized advice strings). This will be shown in the next lecture.

– The collapse of the Polynomial Hierarchy (PH) refers to the situation where a higher level of the hierarchy becomes equal to a lower level.

## 7 RP: Randomized Polynomial Time

In the primality testing algorithm above, we have an error if  $\left(\frac{a}{n}\right)$  (Jacobi symbol) does not equal  $a^{\frac{n-1}{2}} \pmod n$ . This is a *one-sided error*, meaning that if the test returns a failure, we are certain that  $n$  is composite.

To formalize algorithms with one-sided error, we define another complexity class known as RP (*Randomized Polynomial time*). An language  $L$  belongs to the class RP if there is a polynomial time probabilistic algorithm such that

- For all  $x \in L$ , the algorithm correctly accepts with probability at least  $\frac{1}{2}$ .
- For all  $x \notin L$ , the algorithm *never* incorrectly accepts.

coRP is the set of languages which are complement of those in RP. Thus, the primality testing algorithm falls into the class coRP due to its one-sided error behavior, where incorrect outputs only occur when falsely identifying composite numbers as primes, but never the reverse.

A language  $L$  is said to be in the class RP (*Randomized Polynomial time*) if there exists a probabilistic Turing machine  $M$  that runs in polynomial time such that:

$$\begin{aligned} x \in L &\Rightarrow \Pr[M(x) = 1] \geq \frac{1}{2}, \\ x \notin L &\Rightarrow \Pr[M(x) = 1] = 0. \end{aligned}$$

*Note:* The class RP is a subset of NP. This is because any problem that can be solved by an RP machine can also be solved by a nondeterministic polynomial-time machine that guesses the correct random string  $r$ .

## 8 Perfect Matching in Bipartite Graphs

**Goal: Check if there is a Perfect Matching** The decision problem is to determine whether a perfect matching exists in a bipartite graph. A perfect matching is a set of edges that connects every vertex in one set of the bipartite graph to exactly one vertex in the other set.

**Adjacency Matrix and Determinants** We represent the bipartite graph using its bi-adjacency matrix  $A$ , where:

- Rows represent vertices from one set (say  $U$ ) in the bipartite graph.
- Columns represent vertices from the other set (say  $V$ ).
- If there is an edge between vertex  $u \in U$  and vertex  $v \in V$ , the entry  $A[u, v]$  is set to 1. Otherwise, it is set to 0.

**Determinants and Matchings:** The determinant of the bi-adjacency matrix relates to matchings because:

$$\det(A) = \sum_{M \text{ is a perfect matching}} (-1)^{\text{Sgn}(M)},$$

where  $\text{Sgn}(M)$  is the sign of the permutation representing a matching. Clearly, if there is no perfect matching then the determinant is zero.

**Limitations of the Determinant Approach** If  $\det(A) = 0$ , it is possible that perfect matchings still exist, such as in the case of a complete bipartite graph. This approach can fail because multiple matchings may cancel out.

**Improved Randomized Approach Using Schwartz-Zippel Lemma** To improve this approach, we randomly assign numbers to the entries in the matrix instead of using 1's. This gives us a probabilistic algorithm:

- If  $\det(A) = 0$ , we conclude with high probability that there is no perfect matching.
- If  $\det(A) \neq 0$ , we conclude with certainty that a perfect matching exists.

This puts the problem in the complexity class  $RP$  (*Randomized Polynomial time*) because it has one-sided error.

**Finding a Perfect Matching Using RNC** There is an algorithm to find a perfect matching (not just determine its existence) in  $RNC$  (*Randomized NC*), which is a class of efficient parallel algorithms.

**Dynamic Programming with Weighted Edges** To find the perfect matching, we fill in the matrix entries with powers of 2:  $2^{w_1}, 2^{w_2}, \dots$ , where  $w_1, w_2, \dots$ , are randomly chosen weights. There is a theorem which says that under a random weight assignment, the minimum weight perfect matching will be unique. The uniqueness helps us to find the unique minimum weight perfect matching in parallel. For each edge in parallel, you can decide whether it is a part of the minimum weight perfect matching.