Randomized Algorithms

Deterministic Algorithms

**Goal:** Prove for all input instances the algorithm solves the problem correctly and the number of steps is bounded by a function in the size of the input (the worst case complexity).

**Average Case Complexity:** For all input instances satisfying a probability distribution, the average complexity is computed.
Randomized Algorithms

- In addition to input, algorithm takes a source of random numbers and makes random choices during execution;
- Behavior can vary even on a fixed input.
- We compute the expected case result (similar to average case).

Randomized algorithms

- A **randomized algorithm** is just one that depends on random numbers for its operation
- These are randomized algorithms:
  - Using random numbers to help find a solution to a problem
  - Using random numbers to improve an approximate solution to a problem
- These are related topics:
  - Getting or generating “random” numbers
  - Generating random data for testing (or other) purposes
Pseudorandom numbers

- The computer is not capable of generating truly random numbers
  - The computer can only generate pseudorandom numbers—numbers that are generated by a formula
  - Pseudorandom numbers look random, but are perfectly predictable if you know the formula
    - Pseudorandom numbers are good enough for most purposes, but not all—for example, not for serious security applications
  - Devices for generating truly random numbers do exist
    - They are based on radioactive decay, or on lava lamps
- “Anyone who attempts to generate random numbers by deterministic means is, of course, living in a state of sin.” —John von Neumann

Generating random numbers

- Perhaps the best way of generating “random” numbers is by the linear congruential method:
  - \[ r = (a \cdot r + b) \mod m; \]
    - where \( a \) and \( b \) are large prime numbers, and \( m \) is \( 2^{32} \) or \( 2^{64} \)
  - The initial value of \( r \) is called the seed.
  - If you start over with the same seed, you get the same sequence of “random” numbers.
- One advantage of the linear congruential method is that it will (eventually) cycle through all possible numbers
- Almost any “improvement” on this method turns out to be worse.
  - “Home-grown” methods typically have much shorter cycles
(pseudo) random numbers in Java

- `import java.util.Random;`
- `new Random(long seed) // constructor`
- `new Random() // constructor, uses System.currentTimeMillis() as seed`
- `void setSeed(long seed)`
- `nextBoolean()`
- `nextFloat(), nextDouble() // 0.0 < return value < 1.0`
- `nextInt(), nextLong() // all 2^32 or 2^64 possibilities`
- `nextInt(int n) // 0 ≤ return value < n`

Shuffling an array

- **Which one is right?**
- **Range of swapping goes from n to 1:**
  ```java
  static void shuffle(int[] array) {
      for (int i = array.length - 1; i > 0; i--) {
          int j = random.nextInt(i + 1);
          swap(array, i, j);
      }
  }
  ```
- **Range of swapping remains to be n:**
  ```java
  static void shuffle(int[] array) {
      for (int i = 0; i < array.length; i++) {
          int j = random.nextInt(array.length);
          swap(array, i, j);
      }
  }
  ```
Shuffle an Array (method 1)

Shuffle a deck of cards.
- In $i$th iteration, choose a random element from remainder of deck and put at index $i$.
  - choose random integer $r$ between 0 and $i+1$
  - swap values in positions $r$ and $i$

<table>
<thead>
<tr>
<th>Array index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>9</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>2</td>
<td>10</td>
<td>J</td>
</tr>
</tbody>
</table>

Shuffling an array

Which one is Good?
- **Good:** all permutations are equally likely
  ```java
  static void shuffle(int[] array) {
      for (int i = array.length-1; i > 0; i--) {
          int j = random.nextInt(i+1);
          swap(array, i, j);
      }
  }
  ```

- **Bad:** all permutations are not equally likely
  ```java
  static void shuffle(int[] array) {
      for (int i = 0; i < array.length; i++) {
          int j = random.nextInt(array.length);
          swap(array, i, j);
      }
  }
  ```
Shuffling an array

**Proof of Goodness:**
- **Good:** all permutations are equally likely
  ```java
  static void shuffle(int[] array) {
    for (int i = array.length-1; i > 0; i--) {
      int j = random.nextInt(i+1);
      swap(array, i, j);
    }
  }
  ```

**Induction on n:**
- **Basic case:** $n = 1$, there is only one output.
- **Inductive case:** For the last position, a number $a_j$ is chosen randomly among all $n$ numbers. The induction hypothesis says shuffle will shuffle $n-1$ numbers randomly.

Shuffling an array

**Bad:** all permutations are *not* equally likely
```java
static void shuffle(int[] array) {
  for (int i = 0; i < array.length; i++) {
    int j = random.nextInt(array.length);
    swap(array, i, j);
  }
}
```

Let $n = 3$, there are 27 cases for the output of random number: 000, 001, ..., 222. If we assume three consecutive random numbers are distinct, then there are only 6 sequences of \{0, 1, 2\}: 012, 021, 102, 120, 201, 210. Checking the output of the above algorithm for each random sequence will reveal the problem.
Motivation for Randomized Algorithms

- Simplicity;
- Performance;
- Reflects reality better (Online Algorithms);
- For many hard problems it helps obtain better complexity bounds when compared to deterministic approaches;

Types of Randomized algorithms

Las Vegas    Monte Carlo
Monte Carlo

- The time is limited by an upper bound.
- It may produce incorrect answer!
- We are able to bound its error by probability.
- By running it many times on independent random variables, we can make the failure probability arbitrarily small at the expense of running time.

Monte Carlo Example

- Suppose we want to decide a n-place function always returns zero, i.e.,
  \[ F(x_1, x_2, \ldots, x_n) = 0 \] for all \( x_i \)?
- We may randomly choose values for \( x_i \) and see if \( F(x_1, x_2, \ldots, x_n) = 0 \).
- It’s impossible to exhaust all values of \( x_i \).
- However, if we have checked enough times and \( F(x_1, x_2, \ldots, x_n) \) is always 0, then we have high probability the answer is true.
Monte Carlo Algorithms

**Goal:** Prove that the algorithm
- with high probability solves the problem correctly;
- for every input the number of steps is bounded by a polynomial in the input size.

**Note:** The expectation is over the random choices made by the algorithm.

Las Vegas

- Always gives the true answer.
- Running time is random (only expected time).
- Running time is bounded (the worst case is known).
- Randomized Quicksort is a Las Vegas algorithm.
Randomized QuickSort

Randomized-Partition($A, p, r$)
1. $i \leftarrow \text{random}(p, r)$
2. exchange $A[i] \leftrightarrow A[p]$
3. return Partition($A, p, r$)  // partition $A$ by the pivot $A[p]$

Randomized-Quicksort($A, p, r$)
1. if $p < r$
2. then $q \leftarrow \text{Randomized-Partition}(A, p, r)$
3. Randomized-Quicksort($A, p, q-1$)
4. Randomized-Quicksort($A, q+1, r$)

Las Vegas Algorithms

Goal: Prove that for all input instances the algorithm solves the problem correctly and the expected number of steps is bounded by a polynomial in the input size.

Note: The expectation is over the random choices made by the algorithm.
In order to analyze this, and other randomized algorithms, we need to use probability.

A **probability space** is a sample space \( S \) together with a probability function, \( \Pr \), that maps subsets of \( S \) to real numbers between 0 and 1, inclusive.

Formally, each subset \( A \) of \( S \) is an event, and we have the following:

1. \( \Pr(\emptyset) = 0 \).
2. \( \Pr(S) = 1 \).
3. \( 0 \leq \Pr(A) \leq 1 \), for any \( A \subseteq S \).
4. If \( A, B \subseteq S \) and \( A \cap B = \emptyset \), then \( \Pr(A \cup B) = \Pr(A) + \Pr(B) \).

**Independence and Conditional Probability**

Two events \( A \) and \( B \) are **independent** if

\[
\Pr(A \cap B) = \Pr(A) \cdot \Pr(B).
\]

A collection of events \( \{A_1, A_2, \ldots, A_n\} \) is **mutually independent** if

\[
\Pr(A_{i_1} \cap A_{i_2} \cap \cdots \cap A_{i_k}) = \Pr(A_{i_1}) \Pr(A_{i_2}) \cdots \Pr(A_{i_k}),
\]

for any subset \( \{A_{i_1}, A_{i_2}, \ldots, A_{i_k}\} \).

The **conditional probability** that an event \( A \) occurs, given an event \( B \), is denoted as \( \Pr(A|B) \), and is defined as

\[
\Pr(A|B) = \frac{\Pr(A \cap B)}{\Pr(B)},
\]

assuming that \( \Pr(B) > 0 \).
Random Variables

- A **random variable** is a function $X$ that maps outcomes from some sample space $S$ to real numbers.
- An **indicator random variable** is a random variable that maps outcomes to the set $\{0, 1\}$.
- The **expected value** of a discrete random variable $X$ is defined as
  \[ E(X) = \sum_x x \Pr(X = x), \]
  where the sum is taken of the range of $X$.
- Two random variables $X$ and $Y$ are **independent** if
  \[ \Pr(X = x | Y = y) = \Pr(X = x), \]
  for all real numbers $x$ and $y$.
- If two random variables $X$ and $Y$ are independent, then we have $E(XY) = E(X)E(Y)$.

Linearity of Expectation

**Theorem 1.25 (The Linearity of Expectation):** Let $X$ and $Y$ be two arbitrary random variables. Then $E(X + Y) = E(X) + E(Y)$.

**Proof:**
\[
E(X + Y) = \sum_x \sum_y (x + y) \Pr(X = x \cap Y = y)
\]
\[
= \sum_x \sum_y x \Pr(X = x \cap Y = y) + \sum_x \sum_y y \Pr(X = x \cap Y = y)
\]
\[
= \sum_x \sum_y x \Pr(X = x \cap Y = y) + \sum_y \sum_x y \Pr(Y = y \cap X = x)
\]
\[
= \sum_x x \Pr(X = x) + \sum_y y \Pr(Y = y)
\]
\[
= E(X) + E(Y).
\]
Expectation

If a random variable $X$ has probability $p_i$ to be $a_i$, the expected value of $X$ is $E[X] = p_1a_1 + p_2a_2 + \ldots + p_na_n$.

- E.g., the expected value of a die is $E[X] = (1+2+3+4+5+6)/6 = 3.5$.

If an event has probability $p$ to happen, then the expected number of trials for that event to happen is $1/p$. Let $X$ be the number of trials needed for the event to happen for the first time. Then

$$P(X = k) = (1-p)^{k-1}p$$

$$E(X) = p + 2(1-p)p + 3(1-p)^2p + 4(1-p)^3p \ldots = \sum_{k=1}^{\infty} k(1-p)^{k-1}p = 1/p.$$  

If $X_1, X_2, \ldots, X_n$ are random variables, then

$$E\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} E[X_i]$$

Randomized QuickSort

**Randomized-Partition**($A, p, r$)

1. $i \leftarrow \text{random}(p, r)$
2. exchange $A[i] \leftrightarrow A[p]$
3. return **Partition**($A, p, r$)  // partition $A$ by the pivot $A[p]$

**Randomized-Quicksort**($A, p, r$)

1. if $p < r$
2. then $q \leftarrow \text{Randomized-Partition}(A, p, r)$
3. **Randomized-Quicksort**($A, p, q-1$)
4. **Randomized-Quicksort**($A, q+1, r$)
Randomized QuickSort

- The pivot element is equally likely to be any of input elements.
- *For any given input, the behavior of Randomized QuickSort is determined not only by the input but also by the random choices of the pivot.*
- We add randomization to QuickSort to obtain the expected performance of the algorithm to be good for any input.

Notation

- Rename the elements of $A$ as $z_1, z_2, \ldots, z_n$, with $z_i$ being the $i^{\text{th}}$ smallest element (Rank “i”): For each key $x$ in a set $X$,
  \[ \text{rank}(x) = |\{x' \in X \mid x' < x\}| + 1 \]
- Define the set $Z_{ij} = \{z_i, z_{i+1}, \ldots, z_j\}$ be the set of elements between $z_i$ and $z_j$, inclusive.
Expected Number of Total Comparisons in PARTITION

Let \( X_{ij} = 1 \) if \((z_i \text{ is compared to } z_j)\), 0 otherwise, where \(i < j\).
Let \( X \) be the total number of comparisons performed by the algorithm. Then

\[
X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}
\]

The expected number of comparisons performed by the algorithm is

\[
E[X] = E\left[ \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij} \right] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[X_{ij}]
\]

by linearity of expectation

\[
= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \text{Pr}\{z_i \text{ is compared to } z_j\}
\]

Comparisons in PARTITION

Observation 1: Each pair of elements is compared at most once during the entire execution of the algorithm
- Elements are compared only to the pivot point!
- Pivot point is excluded from future calls to PARTITION

Observation 2: Only the pivot is compared with elements in both partitions

\[
\begin{array}{c c c c c c c c c c}
7 & 9 & 8 & 3 & 5 & 4 & 1 & 6 & 10 & 2 \\
z_7, z_9, z_8, z_3, z_5, z_4, z_1, z_6, z_{10}, z_2
\end{array}
\]

\[
Z_{1,5} = \{1, 2, 3, 4, 5, 6\} \quad \{7\} \quad \text{pivot} \quad Z_{8,9} = \{8, 9, 10\}
\]

Elements between different partitions are never compared
Comparisons in PARTITION

Case 1: pivot chosen such as: \( z_i < x < z_j \)
- \( z_i \) and \( z_j \) will never be compared

Case 2: \( z_i \) or \( z_j \) is the pivot
- \( z_i \) and \( z_j \) will be compared
- only if one of them is chosen as pivot before any other element in range \( z_i \) to \( z_j \)

\[
\begin{array}{cccccccc}
  7 & 9 & 8 & 3 & 5 & 4 & 1 & 6 & 10 & 2 \\
\end{array}
\]

\[ Z_{1,6} = \{1, 2, 3, 4, 5, 6\} \quad \{7\} \quad Z_{8,9} = \{8, 9, 10\} \]

\[ \Pr\{z_i \text{ is compared to } z_j\}\]

Expected Number of Comparisons in PARTITION

\[ \Pr\{z_i \text{ is compared with } z_j\} = \Pr\{z_i \text{ or } z_j \text{ is chosen as pivot before other elements in } z_{i,j}\} = \frac{2}{j - i + 1} \]

\[ E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr\{z_i \text{ is compared to } z_j\} \]

\[ E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j - i + 1} = \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k + 1} < \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k} = \sum_{i=1}^{n-1} O(\log n) \]

\[ = O(n \log n) \]
Definition of Minimum Cut

- A cut, $C$, of a connected graph, $G$, is a subset of the edges of $G$ whose removal disconnects $G$.
- That is, after removing all the edges of $C$, we can partition the vertices of $G$ into two subsets, $A$, and $B$ such that there are no edges between a vertex in $A$ and a vertex in $B$.
- A **minimum cut** of $G$ is a cut of smallest size among all cuts of $G$.

Applications

- In several applications, it is important to determine the size of a smallest cut of a graph.
  - For example, in a communications network, the failures of the edges of a cut prevents the communication between the nodes on the two sides of a cut.
  - Thus, the size of a minimum cut and the number of such cuts give an idea of the vulnerability of the network to edge failures.

- Small cuts are also important for the automatic classification of web content.
  - Namely, consider a collection of web pages and model them as a graph, where vertices correspond to pages and edges to links between pages.
  - The size of a minimum cut provides a measure of how much groups of pages have related content. Also, we can use minimum cuts to recursively partition the collection into clusters of related documents.
Contracting Edges

- The simple randomized algorithm repeatedly performs contraction operations on the graph.
- Let $G$ be a graph with $n$ vertices, where we allow $G$ to have parallel edges. We denote with $(v, w)$ any edge with endpoints $v$ and $w$.
- The contraction of an edge $e$ of $G$ with endpoints $u$ and $v$ consists of the following steps that yield a new graph with $n - 1$ vertices, denoted $G/e$:
  1. Remove edge $e$ and any other edge between its endpoints, $u$ and $v$.
  2. Create a new vertex, $w$.
  3. For every edge, $f$, incident on $u$, detach $f$ from $u$ and attach it to $w$. Formally speaking, let $z$ be the other endpoint of $f$. Change the endpoints of $f$ to be $z$ and $w$.
  4. For every edge, $f$, incident on $v$, detach $f$ from $v$ and attach it to $w$.

Contracting Edges, An Example

Contracting the edge $(v_6, v_8)$
Contracting Edges, An Example

Contracting the edge \((v_2, v_3)\)

Contracting Edges, An Example

Contracting the edge \((v_7, v_9)\)
Contracting Edges, An Example

This final graph has the same minimum cut as the original graph.

Contracting the edge $(v_1, w_2)$

Contracting the edge $(v_4, v_5)$, yielding $w_5$, contracting $(w_4, w_5)$, yielding $w_6$, and contracting $(w_1, w_3)$, yielding $w_7$. This final graph has the same minimum cut as the original graph.
Karger’s Algorithm

A simple min-cut algorithm, which succeeds with high probability is to repeat the following procedure multiple times, keeping track of the smallest cut that it ever finds:

Algorithm ContractGraph(G):

Input: An undirected graph, G, with n vertices
Output: A cut of G that has minimum size with probability at least \( \frac{2}{n(n-1)} \)

while G has more than 2 vertices do
    pick a random edge, e, of G
    contract edge e
    G ← G/e
return the edges of G

Analysis

Let G be a graph with n vertices and m edges, and let C be a given minimum cut of G. We will evaluate the probability that the algorithm returns the cut C.

Since G may have other minimum cuts, this probability is a lower bound on the success probability of the algorithm.

Let G_i be the graph obtained after i contractions performed by the algorithm and let m_i be the number of edges of G_i. Assume that G_{i-1} contains all the edges of C. The probability that G_i also contains all the edges of C is equal to \( 1 - \frac{k}{m_{i-1}} \) since we contract any given edge of C with probability \( \frac{1}{m_{i-1}} \) and C has k edges.

Thus, the probability, P, that the algorithm returns cut C is

\[
P = \prod_{i=0,\ldots,n-3} \left( 1 - \frac{k}{m_i} \right).
\]
Analysis, part 2

- Since \( k \) is the size of the minimum cut of each graph \( G_i \), we have that each vertex of \( G_i \) has degree at least \( k \).
- Thus, we obtain the following lower bound on \( m_i \), the number of edges of \( G_i \):

\[
m_i \geq \frac{k(n-i)}{2}, \text{ for } i = 0, 1, \ldots, n-3.
\]

- We can then use these bounds to derive a lower bound for \( P \).

Analysis, part 3

- The following bound implies that \( P \) is at least proportional to \( 1/n^2 \):

\[
P = \prod_{i=0}^{n-3} \left(1 - \frac{k}{m_i}\right) \geq \prod_{i=0}^{n-3} \left(1 - \frac{2k}{k(n-i)}\right) = \prod_{i=0}^{n-3} \left(\frac{n-i-2}{n-i}\right) = \frac{2}{n(n-1)} \cdot \frac{1}{3} \cdot \frac{2}{4} \cdot \frac{n-4}{n-2} \cdot \frac{n-5}{n-3} \cdots \frac{1}{2} \cdot \frac{n-3}{n-1} = \frac{1}{(n-1)^2}.
\]
Analysis, part 4

- We can boost the probability by running the algorithm multiple times. In particular, if we run the algorithm for \( t \times n \times (n-1)/2 \) rounds, where \( t \) is a positive integer, we have that at least one round returns cut \( C \) with probability

\[
P(t) = 1 - \left( 1 - \frac{1}{\binom{n}{2}} \right)^{t}.
\]

- By a well-known property (Theorem A.4) of the mathematical constant \( e \), the base of the natural logarithm, \( \ln \), we obtain \( P(t) > 1 - 1/e^t \).

- If we choose \( t = c \ln n \), where \( c \) is a constant, then the success probability is at least \( 1 - 1/n^c \).

Running Time Analysis

- A contraction operation can be executed in \( O(n) \) time, if the graph is represented using an adjacency list.
- Thus, the running time of one round is \( O(n^2) \).
- We repeat the algorithm \( O(n^2 \log n) \) times.
- Thus, the total running time is \( O(n^4 \log n) \).
- This can be improved to \( O(n^2 \log^3 n) \). (See book)
Monte Carlo versus Las Vegas

- A Monte Carlo algorithm produces an answer that is correct with non-zero probability, whereas a Las Vegas algorithm always produces the correct answer.
- The running time of both types of randomized algorithms is a random variable whose expectation is bounded in general by a polynomial in terms of input size.
- These expectations are only over the random choices made by the algorithm independent of the input. Thus independent repetitions of Monte Carlo algorithms drive down the failure probability exponentially.

Topics in Midterm 2

Some Facts About Numbers

- **Prime number** $p$:
  - $p$ is an integer
  - $p \geq 2$
  - The only divisors of $p$ are 1 and $p$

- **Examples**
  - 2, 7, 19 are primes
  - −3, 1, 6 are not primes

- **Prime decomposition of a positive integer** $n$:
  $$ n = p_1^{e_1} \times \cdots \times p_k^{e_k} $$

- **Example**:
  - $200 = 2^3 \times 5^2$

**Fundamental Theorem of Arithmetic:**

The prime decomposition of a positive integer is unique.

Greatest Common Divisor

- The greatest common divisor (GCD) of two positive integers $a$ and $b$, denoted $\text{gcd}(a, b)$, is the largest positive integer that divides both $a$ and $b$
- The above definition is extended to arbitrary integers

- **Examples**:
  - $\text{gcd}(18, 30) = 6$
  - $\text{gcd}(0, 20) = 20$
  - $\text{gcd}(−21, 49) = 7$

- Two integers $a$ and $b$ are said to be **relatively prime** if $\text{gcd}(a, b) = 1$

- **Example**:
  - Integers 15 and 28 are relatively prime
Modular Arithmetic

- Modulo operator for a positive integer \( n \)
  \[ r = a \mod n \]
equivalent to
  \[ a = r + kn \]
and
  \[ r = a - \lfloor a/n \rfloor n \]

- Example:
  \[ 29 \mod 13 = 3 \]
  \[ 13 \mod 13 = 0 \]
  \[ -1 \mod 13 = 12 \]
  \[ 29 = 3 + 2 \times 13 \]
  \[ 13 = 0 + 1 \times 13 \]
  \[ 12 = -1 + 1 \times 13 \]

- Modulo and GCD:
  \[ \gcd(a, b) = \gcd(b, a \mod b) \]

- Example:
  \[ \gcd(21, 12) = 3 \]
  \[ \gcd(12, 21 \mod 12) = \gcd(6, 9) = 3 \]

Powers

- Let \( p \) be a prime
- The sequences of successive powers of the elements of \( Z_p \) exhibit repeating subsequences
- The sizes of the repeating subsequences and the number of their repetitions are the divisors of \( p - 1 \)
- Example \((p = 7)\)

<table>
<thead>
<tr>
<th>( x )</th>
<th>( x^2 )</th>
<th>( x^3 )</th>
<th>( x^4 )</th>
<th>( x^5 )</th>
<th>( x^6 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>6</td>
<td>4</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>6</td>
<td>2</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>6</td>
<td>1</td>
<td>6</td>
<td>1</td>
</tr>
</tbody>
</table>
**Fermat’s Little Theorem**

**Theorem**
Let $p$ be a prime. For each nonzero residue $x$ of $\mathbb{Z}_p$, we have $x^p - 1 \mod p = 1$

**Example ($p = 5$):**

\[
\begin{array}{c|c}
1^4 \mod 5 & 1^2 \mod 5 \\
3^4 \mod 5 & 4^4 \mod 5 \\
\end{array}
\]

**Corollary**
Let $p$ be a prime. For each nonzero residue $x$ of $\mathbb{Z}_p$, the multiplicative inverse of $x$ is $x^{p-2} \mod p$

**Proof**
\[
x(x^{p-2} \mod p) \mod p = xx^{p-2} \mod p = x^{p-1} \mod p = 1
\]

**Pseudoprimality Testing**

- The number of primes less than or equal to $n$ is about $n / \ln n$
- Thus, we expect to find a prime among randomly generated numbers with $b$ bits each
- Testing whether a number is prime (primality testing) is believed to be a hard problem
- An integer $n \geq 2$ is said to be a base-$x$ pseudoprime if
  - $x^{n-1} \mod n = 1$ (Fermat’s little theorem)
- Composite base-$x$ pseudoprimes are rare:
  - A random 100-bit integer is a composite base-2 pseudoprime with probability less than $10^{-13}$
  - The smallest composite base-2 pseudoprime is 341
- Base-$x$ pseudoprimality testing for an integer $n$:
  - Check whether $x^{n-1} \mod n = 1$
  - Can be performed efficiently with the efficient power algorithm
Compositeness Witness Function

Let $n$ be an odd integer that we want to test for primality, and let $witness(x, n)$ be a Boolean function of a random variable $x$ and $n$ with the following properties:

1. If $n$ is prime, then $witness(x, n)$ is always false.
2. If $n$ is composite, then $witness(x, n)$ is false with probability $q < 1$.

The function $witness$ is said to be a **compositeness witness function** with error probability $q$.

- $q$ bounds the probability that $witness$ will incorrectly identify a composite number as possibly prime.

By repeatedly computing $witness(x, n)$ for independent random values of the parameter $x$, we can determine whether $n$ is prime with an arbitrarily small error probability.

Randomized Primality Testing

- Compositeness witness function $witness(x, n)$ with error probability $q$ for a random variable $x$
  
  **Case 1:** $n$ is prime
  
  $witness(x, n) = false$

  **Case 2:** $n$ is composite
  
  $witness(x, n) = false$ with probability $q < 1$

Algorithm $RandPrimeTest(n, k)$ tests whether $n$ is prime by repeatedly evaluating $witness(x, n)$

A variation of this algorithm provides a suitable compositeness witness function for randomized primality testing (Rabin-Miller algorithm)

```
Algorithm RandPrimeTest(n, k)
Input integer n, confidence parameter k and composite witness function witness(x,n) with error probability q
Output an indication of whether n is composite or prime with probability $2^{-k}$

$t \leftarrow \frac{k}{\log_2(1/q)}$
for $i \leftarrow 1$ to $t$
    $x \leftarrow random()$
    if $witness(x,n)= true$
        return "n is composite"
return "n is prime"
```
Rabin-Miller Algorithm

- Uses Fermat’s Little Theorem and the following lemma:

**Lemma 19.6:** Let \( p \) be a prime number greater than 2. If \( x \) is an element of \( \mathbb{Z}_p \) such that
\[
x^2 \equiv 1 \pmod{p},
\]
then either \( x \equiv 1 \pmod{p} \) or \( x \equiv -1 \pmod{p} \).

- A **nontrivial square root of the unity** in \( \mathbb{Z}_n \) is an integer \( 1 < x < n - 1 \) such that \( x^2 \equiv 1 \pmod{n} \).

**Lemma 19.6** states that if \( n \) is prime, there are no nontrivial square roots of the unity in \( \mathbb{Z}_n \).

- The Rabin-Miller algorithm uses this fact to define a good witness \((x, n)\) function for primality testing.

Rabin-Miller Witness Function

**Algorithm witness** \((x, n)\):

Write \( n - 1 \) as \( 2^k m \), where \( m \) is odd.
Compute \( y \leftarrow x^m \pmod{n} \)
If \( y \equiv 1 \pmod{n} \) then
\[
\text{return false} \quad // n \text{ is probably prime}
\]
For \( i \leftarrow 1 \) to \( k - 1 \) do
If \( y \equiv -1 \pmod{n} \) then
\[
\text{return false} \quad // n \text{ is probably prime}
\]
y \leftarrow \( y^2 \pmod{n} \)
\[
\text{return true} \quad // n \text{ is definitely composite}
\]

- An important fact (presented without proof):

**Lemma 19.7:** Let \( n \) be a composite number. There are at most \((n - 1)/4\) positive values of \( x \) in \( \mathbb{Z}_n \) such that the Rabin-Miller compositeness witness function \( \text{witness}(x, n) \) returns true.
Analysis

- Given an odd positive integer \( n \) and a parameter \( k > 0 \), the Rabin-Miller algorithm determines whether \( n \) is prime, with error probability \( 2^{-k} \), by performing \( O(k \log n) \) arithmetic operations.
- The number, \( \pi(n) \), of primes that are less than or equal to \( n \) is \( (n/ \ln n) \). In fact, if \( n \geq 17 \), then \( n/ \ln n < \pi(n) < 1.26n/ \ln n \).
- A consequence of this fact is that a random integer \( n \) is prime with probability \( 1/ \ln n \).
- Thus, to find a prime with a given number \( b \) of bits, we can again use the pattern of repeated independent trials.