Lecture 11: Prime Numbers And Discrete Logarithms

Lecture Notes on "Computer and Network Security"

by Avi Kak (kak@purdue.edu)

February 25, 2021
12:20 Noon

Goals:

- Primality Testing
- Fermat's Little Theorem
- The Totient of a Number
- The Miller-Rabin Probabilistic Algorithm for Testing for Primality
- **Python and Perl Implementations for the Miller-Rabin Primality Test**
- The AKS Deterministic Algorithm for Testing for Primality
- Chinese Remainder Theorem for Modular Arithmetic with Large Composite Moduli
- Discrete Logarithms

CONTENTS

Back to TOC

11.1 PRIME NUMBERS

- **Prime numbers are extremely important to computer security.** As you will see in the next lecture, public-key cryptography would not be possible without prime numbers.

- As stated in Lecture 12, an important concern in public-key cryptography is to test a randomly selected integer for its **primality**. That is, we first generate a random number and then try to figure out whether it is prime.

- An integer is prime if it has exactly two **distinct** divisors, the integer 1 and itself. That makes the integer 2 the **first prime**.

- We will also be very interested in two integers being **relatively prime** to each other. Such integers are also called **coprimes**. Two integers m and n are coprimes if and only if their Greatest Common Divisor is equal to 1. That is if $gcd(m, n) = 1$. Therefore, whereas 4 and 9 are coprimes, 6 and 9 are not. [See Lecture 5 for *gcd*.]

- Much of the discussion in this lecture uses the notion of

coprimes, as defined above. The same concept used in earlier lectures was referred to as **relatively prime**. **But as mentioned above, the two mean the same thing.**

- Obviously, the number 1 is **coprime** to every integer.

Back to TOC

11.2 FERMAT'S LITTLE THEOREM

- Our main concern in this lecture is with testing a randomly generated integer for its primality. As you will see in Section 11.5, a widely-used computationally-efficient test for primality is based directly on Fermat's Little Theorem. [This theorem also plays an important role in the derivation of the famous RSA algorithm for public-key cryptography that is presented in Section 12.2.3 of Lecture 12. Yet another application of this theorem will be in the speedup of the modular exponentiation algorithm that is presented in Section 12.5 of Lecture 12.]

- Fermat's Little Theorem states that when p is a **prime**, then for **every** integer a that is **coprime** to p, the following relationship must hold:

$$a^{p-1} \equiv 1 \pmod{p} \tag{1}$$

Note that we are not allowed to use a's for which $a \equiv p (\text{mod } p)$. **That is, $a = 0$ and a's that are multiples of p are excluded specifically.** [Recall from Section 5.4 of Lecture 5 that $gcd(0, n) = n$ for all n, implying that 0 cannot be a coprime vis-a-vis any number n. Another way of arguing the same point is that, in *mod* n arithmetic, 0 is the same thing as n. Therefore, 0 cannot be coprime to n in just the same way that n cannot be coprime to n.] Another way of stating the theorem in Eq. (1) is that for

every prime p and every a that is coprime to p, $a^{p-1} - 1$ will always be divisible by p.

- The relationship in Eq. (1) is shown more commonly as

$$a^{p-1} \; mod \; p \quad = \quad 1 \tag{2}$$

- To prove the theorem, let's write down the following sequence assuming that p is prime and a is a non-zero integer that is coprime to p:

$$a,\ 2a,\ 3a,\ 4a, ..., (p-1)a \tag{3}$$

It turns out that if we reduce these numbers modulo p, we will simply obtain a **rearrangement** of the sequence

$$1,\ 2,\ 3,\ 4, ..., (p-1)$$

In what follows, we will first show two examples of this and then present a simple proof.

- For example, consider $p = 7$ and $a = 3$. Now the sequence shown in the expression labeled (3) above will be
3, 6, 9, 12, 15, 18 that when expressed modulo 7 becomes
3, 6, 2, 5, 1, 4.

- For another example, consider $p = 7$ and $a = 8$. Now the sequence shown in the expression labeled (3) above will be 8, 16, 24, 32, 40, 48 that when expressed modulo 7 becomes 1, 2, 3, 4, 5, 6.

- Therefore, we can say

$$\{a,\ 2a,\ 3a, ..., (p-1)a\} \quad mod\ p \quad = \\ some\ permutation\ of\ \{1,\ 2,\ 3, ..., (p-1)\} \qquad (4)$$

  for every prime p and every a that is coprime to p.

- The above conclusion can be established more formally by noting first that, since a cannot be a multiple of p, it is impossible for $k \cdot a \equiv 0 \pmod{p}$ for k, $1 \leq k \leq p-1$. The product $k \cdot a$ cannot be a multiple of p because of the constraints we have placed on the values of k and a. Additionally note that $k \cdot a$ is also not allowed to become zero because a must be a non-zero integer and because the smallest value for k is 1. Next we can show that for any j and k with $1 \leq j, k \leq (p-1)$, $j \neq k$, it is impossible that $j \cdot a \equiv k \cdot a \pmod{p}$ since otherwise we would have $(j-k) \cdot a \equiv 0 \pmod{p}$, which would require that either $a \equiv 0 \pmod{p}$ or that $j \equiv k \pmod{p}$.

- Hence, the product $k \cdot a \pmod{p}$ as k ranges from 1 through

$p-1$, both ends inclusive, must yield some permutation of the integer sequence $\{1, 2, 3, \ldots, p-1\}$.

- Therefore, multiplying all of the terms on the left hand side of Eq. (3) would yield

$$a^{p-1} \cdot 1 \cdot 2 \cdots p-1 \quad \equiv \quad 1 \cdot 2 \cdot 3 \cdots p-1 \qquad (\text{mod } p)$$

  Canceling out the common factors on both sides then gives the Fermat's Little Theorem as in Eq. (1). (The common factors can be canceled out because they are all coprimes to p.)

- We therefore have a formal proof for Fermat's Little Theorem as stated in Eq. (1).

- **Do you think it is possible to use Fermat's Little Theorem directly for primality testing?** Let's say you have a number n you want to test for primality. So you have come up with a small *randomly selected* integer a for use in Fermat's Little Theorem. Now let's say you have a magical procedure that can efficiently compute a^{n-1} *mod* n. If the answer returned by this procedure is NOT 1, you can be sure that n is NOT a prime. **However, should the answer equal 1, then you cannot be certain that n is a prime.** You see, if the answer is 1, then n may either be a composite or a prime. [A non-prime number is also

referred to as a composite number.] That is because the relationship of Fermat's Little Theorem is also satisfied by numbers that are composite. For example, consider the case $n = 25$ and $a = 7$:

$$7^{25-1} \ \ mod \ \ 25 \quad = \quad 1$$

For another example of the same, when $n = 35$ and $a = 6$, we have

$$6^{35-1} \ \ mod \ \ 35 \quad = \quad 1$$

- So what is one to do if Fermat's Little Theorem is satisfied for a given number n for a random choice for a? One could try another choice for a. [**Remember, Fermat's Little Theorem must be satisfied by every a that is coprime to n.**] For the case of $n = 25$, we could next try $a = 11$. If we do so, we get

$$11^{25-1} \ \ mod \ \ 25 \quad = \quad 16$$

which tells us with certainty that 25 is not a prime.

- In the examples described above, you can think of the numbers 7, 6, and 11 as **probes** for primality testing. The larger the number of probes, a's, you use for a given n, with all the a's satisfying Fermat's Little Theorem, the greater the probability that n is a prime. You stop testing as soon you see the theorem

not being satisfied for some value of a, since that is an iron-clad guarantee that n is NOT a prime.

- Note that Fermat's Little Theorem does NOT require that the probe a itself be a prime number. If the number n you are testing for primality is indeed a prime, every randomly chosen probe a between 1 and $n-1$ will obvoiusly be coprime to that value of n. On the other hand, should n actually be a composite, any choice you make for a may or may not be coprime to n. Let's say you are testing $n = 9633197$ for primality and a random selection for the probe throws up the value $a = 7$. For this pair of n and a, we have

$$7^{9633197-1} \;\; mod \;\; 9633197 \quad = \quad 117649$$

  implying that 9633197 is definite NOT a prime. As it turns out, the value of $a = 7$ in this test is a factor of 9633197.

- I will show in Section 11.5 how the above logic for primality testing is incorporated in a computationally efficient algorithm known as the Miller-Rabin algorithm.

- Before presenting the Miller-Rabin test in Section 11.5, and while we are on a theory jag, I want to get two more closely related things out of the way in Sections 11.3 and 11.4: the totient function and the Euler's theorem. We will need these in the presentation of the RSA algorithm in Lecture 12.

Back to TOC

11.3 EULER'S TOTIENT FUNCTION

- An important quantity related to positive integers is the Euler's Totient Function, denoted $\phi(n)$.

- As you will see in Lecture 12, the notion of a totient plays a critical role in the famous RSA algorithm for public key cryptography.

- For a given **positive** integer n, $\phi(n)$ is the number of **positive** integers less than or equal to n that are coprime to n. Recall that two integers a and b are coprimes to each other if $gcd(a, b) = 1$; that is, if their greatest common divisor is 1. [See Lecture 5 for *gcd*.] **$\phi(n)$ is known as the totient of n.** [**Don't forget that 0 cannot be a coprime to any integer n since $gcd(0, n) = n \neq 1$ always.**]

- It follows from the definition that $\phi(1) = 1$. Here are some positive integers and their totients:

```
ints:      1   2   3   4   5   6   7   8   9   10   11   12  ....

totients:  1   1   2   2   4   2   6   4   6   4    10    4  ....
```

  To see why $\phi(3) = 2$: We know that 1 is coprime to 3. The number 2 is also coprime to 3 since their gcd is 1. However, 3 is

not coprime to 3 because $gcd(3,3) = 3$.

- If p is prime, its totient is given by $\phi(p) = p - 1$.

- Suppose a number n is a product of two primes p and q, that is $n = p \times q$, then

$$\phi(n) = \phi(p) \cdot \phi(q) = (p-1)(q-1)$$

This follows from the observation that in the set of numbers $\{1, 2, 3, \ldots, p, p+1, \ldots, pq-1\}$, the number p is **not** a coprime to n since $gcd(p, n) = p$. By the same token $2p$, $3p$, ..., $(q-1)p$ are **not** coprimes to n. By similar reasoning, q, $2q$, ..., $(p-1)q$ are **not** coprimes to n. That then leaves the following as the number of coprimes to n:

$$\begin{aligned} \phi(n) &= (pq - 1) - [(q-1) + (p-1)] \\ &= pq - (p + q) + 1 \\ &= (p - 1) \times (q - 1) \\ &= \phi(p) \times \phi(q) \end{aligned}$$

- [**An aside:** Euler's Totient Function and the Euler's Theorem to be presented next are named after Leonhard Euler who lived from 1707 to 1783. He was the first to use the word "function" and gave us the

notation $f(x)$ to describe a function that takes an argument. He was an extremely high-energy and rambunctious sort of a guy who was born and raised in Switzerland and who at the age of 22 was invited by Catherine the Great to a professorship in St. Petersburg. He is considered to be one of the greatest mathematicians and probably the most prolific. His work fills 70 volumes, half of which were written with the help of assistants during the last 17 years of his life when he was completely blind.

As to how he became blind is a story unto itself. Being intensely curious about the solar eclipse, the legend has it that he would try watching it directly without any eye protection. On the other hand, Galileo, who lived in the century previous to Euler's and who was even more intensely interested in astronomical phenomena, used to watch solar eclipses through their reflection in water.

Such are the stories of the greats of the past who have shaped us as we know ourselves today.]

Back to TOC

11.4 EULER'S THEOREM

- This theorem states that for **every** positive integer n and **every** a that is **coprime** to n, the following must be true

$$a^{\phi(n)} \equiv 1 \pmod{n}$$

where, as defined in the previous section, $\phi(n)$ is the totient of n.

- Note that **when** n **is a prime**, $\phi(n) = n - 1$. In this case, Euler's Theorem reduces to the Fermat's Little Theorem. However, Euler's Theorem holds for **all** positive integers n as long as a and n are coprime.

- To prove Euler's theorem, let's say

$$R = \{x_1, x_2, \ldots, x_{\phi(n)}\}$$

is the set of all integer less than n that are relatively prime (the same thing as co-prime) to n.

- Now let S be the set obtained when we multiply modulo n each element of R by some integer a co-prime to n. That is

$$S \;\; = \;\; \Big\{a \times x_1 \bmod n,\; a \times x_2 \bmod n, \ldots,\; a \times x_{\phi(n)} \bmod n\Big\}$$

- We claim that S is simply a permutation of R. To prove this, we first note that $(a \times x_i \bmod n)$ cannot be zero because, as a and x_i are coprimes to n, the product $a \times x_i$ cannot contain n as a factor. Next we can show that for $1 \leq i, j \leq \phi(n)$, $i \neq j$, it is not possible for $(a \times x_i \bmod n)$ to be equal to $(a \times x_j \bmod j)$. If it were possible for $(a \times x_i \bmod n)$ to be equal to $(a \times x_j \bmod j)$, then $(a \times x_i \;-\; a \times x_j \;\equiv\; 0 \pmod n)$ since both $a \times x_i$ and $a \times x_j$ are coprimes to n. That would imply that either a is 0 mod n, or that $x_i \equiv x_j \pmod n$, both clearly violating the assumptions.

- Therefore, we can say that

$$S \;\; = \;\; \textit{merely a permutation of } R$$

  implying that multiplying **all** of the elements of S should equal the product of **all** of the elements of R. That is

$$\prod_i s_i \in S \bmod n \;\; = \;\; \prod_i r_i \in R \bmod n$$

- Looking at the individual elements of S, multiplying all of the elements of S will give us a result that is $a^{\phi(n)}$ times the

product of all of the elements of R. So the above equation can be expressed as

$$a^{\phi(n)} \times \prod_i r_i \in R \quad \equiv \quad \prod_i r_i \in R \pmod{n}$$

which then directly leads to the statement of the theorem.

Back to TOC

11.5 MILLER-RABIN ALGORITHM FOR PRIMALITY TESTING

- One of the most commonly used algorithms for testing a randomly selected number for primality is the Miller-Rabin algorithm.

- A most notable feature of this algorithm is that it only makes a **probabilistic assessment of primality**: If the algorithm says that a number is **composite** (the same thing as **not a prime**), then the number is definitely not a prime. On the other hand, if the algorithm says that a number **is** a prime, then with a very small probability the number may **not** actually be a prime. (*With proper algorithmic design, this probability can be made so small that, as someone has said, there would be a greater probability that, as you are sitting at a workstation, you'd win a lottery and get hit by a bolt of lightening at the same time.*)

- The algorithm is presented in detail in the next several subsections. However, before you delve into these subsections, keep in the mind the fact that, theoretically speaking, all that the Miller-Rabin test does is to check whether or not the equality $a^{p-1} \equiv 1 \, (mod \, p)$ is satisfied for a candidate prime p

and for a set of values for the probe a. What the next few subsections accomplish is to show how this test can be carried out in a computationally efficient manner by exploiting a factorization of the even number $p-1$. As to how many probes one should try for the test, we will address that issue in Section 11.5.6.

Back to TOC

11.5.1 Miller-Rabin Algorithm is Based on an Intuitive Decomposition of an Even Number into Odd and Even Parts

- Given any odd positive integer n, we can express $n-1$ as a product of a power of 2 and a smaller odd number:

$$n \; - \; 1 \;\; = \;\; 2^k \cdot q \quad for \;\; some \;\; k > 0, \;\; and \;\; \textbf{odd} \;\; q$$

  This follows from the fact that if n is odd, then $n-1$ is even. It follows that after we have factored out the largest power of 2 from $n-1$, what remains, meaning q, must be odd.

- In any programming language, finding the values for k and q is quite trivial. As you will see in the Python and Perl scripts shown in Section 11.5.5, all you have to do is to count the number of trailing zeros in the bit representation of the integer $n-1$. [In general, given an odd integer, its least significant bit (the rightmost bit in the most commonly used printed representation of the binary representations of integers) will be set to 1. Multiplying this integer by 2 amounts to shifting the bit pattern for the odd integer to the left by one position. So if an odd integer (which in our case would be q) is multiplied k times by 2, you would be shifting the bit pattern for q to the left by k positions. Reversing this argument, in order to discover how many times 2 can divide an arbitrary integer $n-1$, all we have to do is to count how many trailing zeros there are in the bit representation of $n-1$.]

Back to TOC

11.5.2 Miller-Rabin Algorithm Uses the Fact that $x^2 = 1$ Has No Non-Trivial Roots in Z_p

- When we say that $x^2 = 1$ has only trivial roots in Z_p for any prime p, we mean that only $x = 1$ and $x = -1$ can satisfy the equation $x^2 = 1$. [Z_p was defined in Section 5.5 of Lecture 5 as a **prime finite field**.]

- Let's first try to see what the negative integer -1 stands for in the finite field Z_p for any prime p.

- Let's consider the finite field Z_7 for a moment:

```
Natural
nums: ... -8 -7 -6 -5 -4 -3 -2 -1  0  1  2  3  4  5  6  7  8  9  ...

Z_7 : ...  6  0  1  2  3  4  5  6  0  1  2  3  4  5  6  0  1  2  ...
```

We notice that -1 is congruent to 6 modulo 7. In general, we can say that for any prime p, we have in the finite field Z_p:

$$-1 \quad \equiv \quad (p \; - \; 1) \qquad (\text{mod } p)$$

- Getting back to the title of this section, an interesting thing about the prime finite field Z_p is that there exist only two numbers, -1 and 1, in the field that when squared give us 1. That is,

$$\begin{aligned} 1 \cdot 1 \;\; mod\; p &= 1 \\ -1 \cdot -1 \;\; mod\; p &= 1 \end{aligned}$$

- The relationship shown above also holds for any two integers a and b, with a congurent to 1 modulo p, and b congruent to -1 modulo p. That is, for *any* integer a with $a \equiv 1 \pmod{p}$ and *any* integer b with $b \equiv -1 \pmod{p}$, we must have:

$$a^2 \mod p = (a \mod p) \cdot (a \mod p) \mod p = 1$$

$$b^2 \mod p = (b \mod p) \cdot (b \mod p) \mod p = 1$$

Besides 1 and -1, there do **not** exist any other integers $x \in Z_p$ that when squared will return 1 mod p.

- We will prove the above assertion **by contradiction**:

  - Let's assume that there does exist an $x \in Z_p$, $x \neq 1$ and $x \neq -1$, such that

$$x \cdot x \mod p \quad = \quad 1$$

which is the same thing as saying that

$$x^2 \quad \equiv \quad 1 \pmod{p}$$

– The above equation can be expressed in the following forms:

$$\begin{aligned} x^2 - 1 &\equiv 0 \pmod{p} \\ x^2 - x + x - 1 &\equiv 0 \pmod{p} \\ (x - 1) \cdot (x + 1) &\equiv 0 \pmod{p} \end{aligned}$$

– Now remember that in our **proof by contradiction** we are not allowing x to be either -1 or 1. Therefore, for the last of the above equivalences to hold true, it must be the case that either $x - 1$ or $x + 1$ is congruent to 0 modulo the prime p. But we know already that, **when p is prime, no number is Z_p can satisfy this condition if x is not allowed to be either 1 or -1.** [Any x, which is neither 1 nor -1, satisfying the last of the equations above would imply that p possesses non-trivial factors. Remember, 0 is the same thing as p in arithmetic modulo p.] Therefore, the above equivalences must be false unless x is either -1 or 1. (As mentioned earlier, -1 is a standin for $p - 1$ in the finite field Z_p.)

- We summarize the above proof by saying that in Z_p the equation $x^2 = 1$ has only two **trivial** roots -1 and 1. There do

not exist any **non-trivial** roots for $x^2 = 1$ in Z_p for any prime p.

Back to TOC

11.5.3 Miller-Rabin Algorithm: Two Special Conditions That Must Be Satisfied by a Prime

- First note that for any prime p, it being an odd number, the following relationship must hold (as stated in Section 11.5.1)

$$p \;-\; 1 \;=\; 2^k \cdot q \qquad for\ some\ k > 0,\ and\ odd\ q$$

- Therefore, in terms of k and q, establishing that $a^{p-1} \equiv 1\ (mod\ p)$ boils down to showing that

$$a^{2^k \cdot q} \;\equiv\; 1 \qquad\qquad (mod\ p)$$

To understand the computational ramifications of this form, we will be interested in the following powers of a^q:

$$a^q,\quad a^{2q},\quad a^{2^2 q},\quad a^{2^3 q},\quad .....,\quad a^{2^k q}$$

Notice that each term in this sequence is a square of the previous term. **This observation creates a special computational advantage for the Miller-Rabin algorithm.** This is made evident by the arguments that follow:

- The Miller-Rabin algorithm is based on the observation that for any integer a in the range $1 < a < p-1$ (pay attention to the two inequalities; they say that a is **not** allowed to take on either the **first two values** or **the last value** of the range of the integers in Z_p and that all of the allowed values for a are coprime to p if p is truly a prime), **one of the following** conditions must be true when p is a prime:

  CONDITION 1: Either it must be the case that

$$a^q \equiv 1 \pmod{p}$$

  CONDITION 2: Or, it must be the case that one of the numbers a^q, a^{2q}, a^{4q}, ..., $a^{2^{k-1}q}$ is congruent to -1 modulo p. That is, there exists some number j in the range $1 \leq j \leq k$, such that

$$a^{2^{j-1}q} \equiv -1 \pmod{p}$$

- The rest of this subsection presents a proof for the Conditions 1 and 2 stated above. We must prove that when p is a prime, then either **Condition 1** or **Condition 2** must be satisfied.

- Since $p - 1 = 2^k \cdot q$ for some k and for some odd integer q, the following statement of Fermat's Little Theorem

$$a^{p-1} \equiv 1 \pmod{p}$$

can be re-expressed as

$$a^{2^k \cdot q} \equiv 1 \pmod{p}$$

for any positive integer a that is coprime to p. For prime p, that includes all values of a such that $1 \leq a \leq (p-1)$.

- We now restrict the range of a to $1 < a < (p-1)$ by excluding from the range specified for the Fermat's Little Theorem the values $a = 1$ and $a = p - 1$, the second being the same as $a = -1$. That is because Fermat's Little Theorem is always satisfied for these two values of a regardless of whether p is a prime or a composite.

- Choosing some a in the range $1 < a < (p-1)$, let's examine the following sequence of numbers

$$a^q \bmod p, \quad a^{2q} \bmod p, \quad a^{2^2 q} \bmod p, \quad a^{2^3 q} \bmod p, \quad ....., \quad a^{2^k q} \bmod p$$

Note that every number in this sequence is a square of the previous number. Therefore, on the basis of the argument presented in Section 11.5.2, either it **must** be the

case that the first number satisfies $a^q \bmod p = 1$, in which case every number in the sequence is 1; **or** it **must** be the case that one of the numbers in the sequence is -1 (**the other** square-root of 1), which would then make all the subsequent numbers equal to 1. This is the proof for **Condition 1** and **Condition 2** of the previous section. [You might ask as to why this proof does not include the following logic: If one of the members of the sequence after the first member is +1, that would also make all subsequent members equal to +1. To respond, let's say that the k^{th} member is the **first** member of the sequence that is +1. That, by Section 11.5.2, implies that the $(k-1)^{th}$ member must be -1. This $(k-1)^{th}$ member could even be the first member of the sequence. So we are led back to the conclusion that either the first member is +1 or one of the members (including possibly the first) before we get to the end of the sequence is -1.]

- In the logic stated above, note the role played by the fact that when $x^2 = 1$ in Z_p, then it must be the case that either $x = 1$ or $x = -1$. (This fact was established in Section 11.5.2.) Also recall that in Z_p, the number -1 is the same thing as $p - 1$.

Back to TOC

11.5.4 Consequences of the Success and Failure of One or Both Conditions

- The upshot of the points made so far is that if for a given number p there exists a probe a that is greater than 1 and less than $p-1$ and for which **neither** of the **Conditions 1 and 2** is satisfied, then the number p is definitely **not** a prime. However, if **either** of the two **Conditions** is true for a probe a, then p may be either a composite or a prime.

- Since we have **not** established a "if and only if" sort of a connection between the primality of a number and the two **Conditions**, it is certainly possible that a composite number may also satisfy the two **Conditions**.

- From experiments it is known that if either of the **Conditions** is true for a randomly selected $1 < a < (p-1)$, then p is likely to be prime with a very high probability. To increase the probability of n being a prime, one can repeat testing for the two **Conditions** with different randomly selected choices for the probe integer a.

- In Section 11.5.6 of this lecture we talk about how many probes

it may take for one to accept a candidate number n as a prime with a high level of confidence.

Back to TOC

11.5.5 Python and Perl Implementations for the Miller-Rabin Algorithm

- Shown on the next page is a Python implementation of the Miller-Rabin algorithm for primality testing. The names chosen for the variables should either match those in the earlier explanations in this lecture or are self-explanatory.

- You will notice that this code only uses for a the values 2, 3, 5, 7, 11, 13, and 17, as shown in line (A3). Researchers have shown that using these for probes suffices for primality testing for integers smaller than 341,550,071,728,321. [As you will see in the next lecture, asymmetric-key cryptography uses prime numbers that are frequently much larger than this. So the probe set shown here would not be sufficient for those algorithms.]

- As you should expect by this time, the very first thing our implementation must do is to express a prime candidate p in the form $p - 1 = q * 2^k$. This is done in lines (A6) through (A9) of the script. Note how we find the values of q and k by bit shifting. [This is standard programming idiom for finding how many times an integer is divisible by 2. Also see the explanation in the second bullet in Section 11.5.1.]

- What you see in lines (A10) through (A20) is the loop that tests the candidate prime p with each of the probe values. As shown in lines (A12) and (A13), a probe yields success if a^q is either equal to 1 or to $p-1$ (which is the same thing as -1 in $mod\ p$ arithmetic). If neither is the case, we then resort to the inner loop in lines (A16) through (A20) for squaring at each iteration a power of a^q. Should one of these powers equal $p-1$, we exit the inner loop.

- The part of the code in lines (M4) through (M10) exercises the testing function on a set of primes that have been diddled with the addition of a small random integer.

- Here is the Python implementation:

```
#!/usr/bin/env python

##  PrimalityTest.py
##  Author: Avi Kak
##  Date:   February 18, 2011
##  Updated:  February 28, 2016
##  An implementation of the Miller-Rabin primality test

### You can call this script with either no comamnd-line args or with just one
### command-line arg.  If you call it with no args, it returns primality results on a
### set of randomly altered 36 primes.  On the other hand, if you call it with just
### one arg, it returns the answer for that integer.

def test_integer_for_prime(p):                                              #(A1)
    if p == 1: return 0                                                     #(A2)
    probes = [2,3,5,7,11,13,17]                                             #(A3)
    if p in probes: return 1                                                #(A4)
    if any([p % a == 0 for a in probes]): return 0                          #(A5)
    k, q = 0, p-1        # need to represent p-1 as  q * 2^k                #(A6)
    while not q&1:                                                          #(A7)
        q >>= 1                                                             #(A8)
```

```
        k += 1                                                              #(A9)
    for a in probes:                                                        #(A10)
        a_raised_to_q = pow(a, q, p)                                        #(A11)
        if a_raised_to_q == 1: continue                                     #(A12)
        if (a_raised_to_q == p-1) and (k > 0): continue                     #(A13)
        a_raised_to_jq = a_raised_to_q                                      #(A14)
        primeflag = 0                                                       #(A15)
        for j in range(k-1):                                                #(A16)
            a_raised_to_jq = pow(a_raised_to_jq, 2, p)                      #(A17)
            if a_raised_to_jq == p-1:                                       #(A18)
                primeflag = 1                                               #(A19)
                break                                                       #(A20)
        if not primeflag: return 0                                          #(A21)
    probability_of_prime = 1 - 1.0/(4 ** len(probes))                       #(A22)
    return probability_of_prime                                             #(A23)

primes = [179, 233, 283, 353, 419, 467, 547, 607, 661, 739, 811, 877, \
          947, 1019, 1087, 1153, 1229, 1297, 1381, 1453, 1523, 1597, \
          1663, 1741, 1823, 1901, 7001, 7109, 7211, 7307, 7417, 7507, \
          7573, 7649, 7727, 7841]                                           #(A24)

if __name__ == '__main__':

    import sys                                                              #(M1)
    import random                                                           #(M2)

    if len(sys.argv) == 1:                                                  #(M3)
        for p in primes:                                                    #(M4)
            p += random.randint(1,10)                                       #(M5)
            probability_of_prime = test_integer_for_prime(p)                #(M6)
            if probability_of_prime > 0:                                    #(M7)
                print("%d is prime with probability: %f" %(p,probability_of_prime))
                                                                            #(M8)
            else:                                                           #(M9)
                print("%d is composite" % p)                                #(M10)
    elif len(sys.argv) == 2:                                                #(M11)
        p = int(sys.argv[1])                                                #(M12)
        probability_of_prime = test_integer_for_prime(p)                    #(M13)
        if probability_of_prime > 0:                                        #(M14)
            print("%d is prime with probability: %f" %(p,probability_of_prime))
                                                                            #(M15)
        else:                                                               #(M16)
            print("%d is composite" % p)                                    #(M17)
    else:                                                                   #(M18)
        sys.exit("""You cannot call 'PrimalityTest.py' with more """        #(M19)
                 """than one command-line argument""")
```

- When called without a command-line argument, the exact output of the above script will depend on how the prime numbers are modified in line (M5). A typical run without a

command-line argument will produce something like what is shown below:

```
181   is prime with probability:  0.999938964844
234   is composite
291   is composite
361   is composite
423   is composite
477   is composite
555   is composite
614   is composite
668   is composite
748   is composite
814   is composite
884   is composite
954   is composite
1025  is composite
1091  is prime with probability:  0.999938964844
1162  is composite
1231  is prime with probability:  0.999938964844
1306  is composite
1387  is composite
1456  is composite
1527  is composite
1603  is composite
1671  is composite
1742  is composite
1833  is composite
1911  is composite
7008  is composite
7119  is composite
7212  is composite
7308  is composite
7424  is composite
7512  is composite
7582  is composite
7657  is composite
7734  is composite
7844  is composite
```

- On the other hand, if you call the Python script shown above with an integer supplied as a command-line argument, it will report back the result for just that integer.

- Shown next is the Perl implementation of the same algorithm. The only significant difference between the Python code shown above and the Perl code shown next is regarding the modular exponentiation step implemented in lines (A18) through (A21) of the script that follows. [I am referring to implementing in Perl what was done by a single statement call in line (A11) of the Python code.] Unless you use the Perl's `Math::BigInt` library, you can be pretty certain that Perl will make errors even for seemingly small exponentiations like 3^{89}. The result of this exponentiation cannot be accomodated in Perl's native 4-byte representation for an unsigned integer. [The largest unsigned integer that Perl can fit in a 4-byte representation is $2^{32} - 1$.] So, at some point during the calculation of 3^{89}, Perl will switch to a floating point representation for the partial result whose conversion to int will not yield the correct answer. Try calculating `(3 ** 89) % 179` in Perl. And then try to do the same in Python by calling `pow(3,89,179)` or, for that matter, even by the less efficient `(3 ** 89) % 179`. Python will yield the correct answer of 1 in either case. On the other hand, Perl's answer will be incorrect — I get 8 on my machine. To get around this problem, the code in lines (A18) through (A21) is an implementation of the modular exponentiation algorithm that the built-in function `pow()` of Python is also based on.

```
#!/usr/bin/env perl

##  PrimalityTest.pl
##  Author: Avi Kak
##  Date:   February 28, 2016

##  An implementation of the Miller-Rabin primality test

### You can call this script with either no comamnd-line args or with just one
### command-line arg.  If you call it with no args, it returns primality results on a
```

```
### set of randomly altered 36 primes.  On the other hand, if you call it with just
### one arg, it returns the answer for that integer.

use strict;
use warnings;

unless (@ARGV) {
    my @primes = qw[ 179 233 283 353 419 467 547 607 661 739 811 877
                     947 1019 1087 1153 1229 1297 1381 1453 1523 1597
                     1663 1741 1823 1901 7001 7109 7211 7307 7417 7507
                     7573 7649 7727 7841 ];                                    #(M1)
    foreach my $p (@primes) {                                                  #(M2)
        $p += 1 + int(rand(10));                                               #(M3)
        my $probability_of_prime = test_integer_for_prime($p);                 #(M4)
        $probability_of_prime > 0 ?                                            #(M5)
          print "$p is prime with probability: $probability_of_prime\n" :      #(M6)
          print "$p is composite\n";                                           #(M7)
    }
} elsif (@ARGV == 1) {                                                         #(M8)
    my $p = shift;                                                             #(M9)
    die "Your number is too large for this script. Instead, try the " .
        "script 'PrimalityTestWithBigInt.pl'\n"
        if $p > 0x7f_ff_ff_ff;                                                 #(M10)
    my $probability_of_prime = test_integer_for_prime($p);                     #(M11)
    $probability_of_prime > 0 ?
      print "$p is prime with probability: $probability_of_prime\n" :
      print "$p is composite\n";                                               #(M12)
} else {                                                                       #(M13)
    die "You cannot call 'PrimalityTest.py' with more " .
        "than one command-line argument";                                      #(M14)
}

sub test_integer_for_prime {                                                   #(A1)
    my $p = shift;                                                             #(A2)
    return 0 if $p == 1;                                                       #(A3)
    my @probes = (2,3,5,7,11,13,17);                                           #(A4)
    my @in_probes = grep {$p == $_} @probes;                                   #(A5)
    return 1 if @in_probes;                                                    #(A6)
    my $p_mod_a = 1;                                                           #(A7)
    map { $p_mod_a = 0 if $p % $_ == 0 } @probes;                              #(A8)
    return 0 if $p_mod_a == 0;                                                 #(A9)
    my ($k, $q) = (0, $p - 1);                                                 #(A10)
    while (! ($q & 1)) {                                                       #(A11)
        $q >>= 1;                                                              #(A12)
        $k += 1;                                                               #(A13)
    }
    my ($a_raised_to_q, $a_raised_to_jq, $primeflag);                          #(A14)
    foreach my $a (@probes) {                                                  #(A15)
        my ($base,$exponent) = ($a,$q);                                        #(A16)
        my $a_raised_to_q = 1;                                                 #(A17)
        while ((int($exponent) > 0)) {                                         #(A18)
            $a_raised_to_q = ($a_raised_to_q * $base) % $p
                                          if int($exponent) & 1;               #(A19)
            $exponent = $exponent >> 1;                                        #(A20)
            $base = ($base * $base) % $p;                                      #(A21)
```

```
        }
        next if $a_raised_to_q == 1;                                    #(A22)
        next if ($a_raised_to_q == ($p - 1)) && ($k > 0);               #(A23)
        $a_raised_to_jq = $a_raised_to_q;                               #(A24)
        $primeflag = 0;                                                 #(A25)
        foreach my $j (0 .. $k - 2) {                                   #(A26)
            $a_raised_to_jq = ($a_raised_to_jq ** 2) % $p;              #(A27)
            if ($a_raised_to_jq == $p-1) {                              #(A28)
                $primeflag = 1;                                         #(A29)
                last;                                                   #(A30)
            }
        }
        return 0 if ! $primeflag;                                       #(A31)
    }
    my $probability_of_prime = 1 - 1.0/(4 ** scalar(@probes));          #(A32)
    return $probability_of_prime;                                       #(A33)
}
```

- As was the case with the Python script, the Perl script shown above can also be called with and without a command-line argumnet, and its behavior in both cases is the same as for the Python script — except when the number involved is too large to fit in a 4-byte representation that Perl uses for unsigned ints. Since you have already seen the without-command-line-argument behavior for the Python case, here is calling the Perl script shown above with an integer supplied through the command line:

  ```
  PrimalityTest.pl 1234567891
  ```

  and it comes back

  ```
  1234567891 is prime with probability: 0.99993896484375
  ```

- On the other hand, if you call the script with a larger number, as in

  ```
  PrimalityTest.pl 123456789123456789
  ```

you will get the following response from the script:

```
Your number is too large for this script. Instead, try the
script 'PrimalityTestWithBigInt.pl'
```

- As implied by the above error message, if you want to use Perl for primality testing of really large numbers, you'll have to import the `Math::BigInt` library into your script, as shown by the script that follows:

```
#!/usr/bin/env perl

##  PrimalityTestWithBigInt.pl
##  Author: Avi Kak
##  Date: February 28, 2016

use strict;
use warnings;
use Math::BigInt;

die "\nUsage:   $0  <integer> \n" unless @ARGV == 1;                          #(M1)

my $p = shift @ARGV;                                                         #(M2)
$p = Math::BigInt->new( "$p" );                                              #(M3)

my $answer = test_integer_for_prime($p);                                     #(M4)
if ($answer) {                                                               #(M5)
    print "$p is prime with probability: $answer\n";                         #(M6)
} else {
    print "$p is composite\n";                                               #(M7)
}

sub test_integer_for_prime {                                                 #(A1)
    my $p = shift;                                                           #(A2)
    return 0 if $p->is_one();                                                #(A3)
    my @probes = qw[ 2 3 5 7 11 13 17 ];                                     #(A4)
    foreach my $a (@probes) {                                                #(A5)
        $a = Math::BigInt->new("$a");                                        #(A6)
        return 1 if $p->bcmp($a) == 0;                                       #(A7)
        return 0 if $p->copy()->bmod($a)->is_zero();                         #(A8)
    }
    my ($k, $q) = (0, $p->copy()->bdec());                                   #(A9)
    while (! $q->copy()->band( Math::BigInt->new("1"))) {                    #(A10)
        $q->brsft( 1 );                                                      #(A11)
        $k += 1;                                                             #(A12)
```

```
    }
    my ($a_raised_to_q, $a_raised_to_jq, $primeflag);                          #(A13)
    foreach my $a (@probes) {                                                  #(A14)
        my $abig = Math::BigInt->new("$a");                                    #(A15)
        my $a_raised_to_q = $abig->bmodpow($q, $p);                            #(A16)
        next if $a_raised_to_q->is_one();                                      #(A17)
        my $pdec = $p->copy()->bdec();                                         #(A18)
        next if ($a_raised_to_q->bcmp($pdec) == 0) && ($k > 0);                #(A19)
        $a_raised_to_jq = $a_raised_to_q;                                      #(A20)
        $primeflag = 0;                                                        #(A21)
        foreach my $j (0 .. $k - 2) {                                          #(A22)
            my $two = Math::BigInt->new("2");                                  #(A23)
            $a_raised_to_jq = $a_raised_to_jq->copy()->bmodpow($two, $p);      #(A24)
            if ($a_raised_to_jq->bcmp( $p->copy()->bdec() ) == 0 ) {           #(A25)
                $primeflag = 1;                                                #(A26)
                last;                                                          #(A27)
            }
        }
        return 0 if ! $primeflag;                                              #(A28)
    }
    my $probability_of_prime = 1 - 1.0/(4 ** scalar(@probes));                 #(A29)
    return $probability_of_prime;                                              #(A30)
}
```

- If you call the script with a larger number, as in

  ```
  PrimalityTestWithBigInt.pl 123456789123456789123456789l
  ```

  you will get the following response from the script:

  ```
  123456789123456789123456789l is prime with probability: 0.99993896484375
  ```

Back to TOC

11.5.6 Miller-Rabin Algorithm: Liars and Witnesses

- When n is **known** to be composite, then the dual test

$$a^q \not\equiv 1$$

and

$$a^{2^i \cdot q} \not\equiv -1 \bmod n \quad for\ \ all\ \ 0 < i < k-1$$

will be satisfied by only a certain number of a's, $a < n$. All such a's are called **witnesses for the compositeness** of n.

- When a randomly chosen a for a known composite n does not satisfy the dual test above, it is called a **liar** for the compositeness of n.

- It has been shown theoretically that, in general, for a **composite** n, at least 3/4th of the numbers $a < n$ will be witnesses for its compositeness.

- It follows from the above statement that if n is indeed composite, then the Miler-Rabin algorithm will declare it to be

a prime with a probability of 4^{-t} where t is the number of probes used.

- In reality, the probability of a composite number being declared prime by the Miller-Rabin algorithm is significantly less than 4^{-t}.

- If you are careful in how you choose a candidate for a prime number, you can safely depend on the Miller-Rabin algorithm to verify its primality.

Back to TOC

11.5.7 Computational Complexity of the Miller-Rabin Algorithm

- The running time of this algorithm is $O(t \times \log^3 n)$ where n is the integer being tested for its primality and t the number of probes used for testing. [In the theory of algorithms, the notation $O()$, sometimes called the 'Big-O', is used to express the limiting behavior of functions. If you write $f(n) = O(g(n))$, that implies that as $n \rightarrow \infty$, $f(n)$ will behave like $g(n)$. More precisely, it means that as $n \rightarrow \infty$, there will exist a positive integer M and an integer n_0 such that $|f(n)| \leq M|g(n)|$ for all $n > n_0$. **(At Purdue, the theory of complexity is taught in ECE664.)**] A more efficient implementation can reduce the time complexity measure to $O(t \times \log^2 n)$.

- In the theory of algorithms, the Miller-Rabin algorithm would be called a **randomized algorithm**.

- A **randomized algorithm** is an algorithm that can make random choices during its execution.

- As a randomized algorithm, the Miller-Rabin algorithm belongs to the class **co-RP**.

- The class **RP** stands for **randomized polynomial time**. This is

the class of problems that can be solved in polynomial time with randomized algorithms provided errors are made on only the "yes" inputs. What that means is that when the answer is known to be "yes", the algorithm occasionally says "no".

- The class **co-RP** is similar to the class **RP** except that the algorithm occasionally makes errors on only the "no" inputs. What that means is that when the answer is known to be "no", the algorithm occasionally says "yes".

- The Miller-Rabin algorithm belongs to **co-RP** because occasionally when an input number is known to **not** be a prime, the algorithm declares it to be prime.

- The class **co-RP** is a subset of the class BPP. BPP stands for **bounded probabilistic polynomial-time**. These are randomized polynomial-time algorithms that yield the correct answer with an exponentially small probability of error.

- The fastest algorithms that behave deterministically belong to the class **P** in the theory of computational complexity. **P** stands for **polynomial-time**. All problems that can be solved in exponential time in a deterministic machine belong to the class **NP** in the theory of computational complexity.

- The class **P** is a subset of class **BPP** and there is no known direct relationship between the classes **BPP** and **NP**. In general we have

$$P \subset RP \subset NP$$
$$P \subset co-RP \subset BPP$$

Back to TOC

11.6 THE AGRAWAL-KAYAL-SAXENA (AKS) ALGORITHM FOR PRIMALITY TESTING

- Despite the **millennia old obsession** with prime numbers, until 2002 there did not exist a computationally efficient test with an unconditional guarantee of primality.

  – A deterministic test of primality (as opposed to a randomized test) is considered to be **computationally efficient** if it belongs to class **P**. That is, the running time of the algorithm must be a polynomial function of the size of the number whose primality is being tested. (**The size of** n **is proportional to** $\log n$**.** Think of the binary representation of n.)

  – If there was no concern about computational efficiency, you could always test for primality by dividing n by all integers up to $\sqrt{n}$. The running time of this algorithm would be directly proportional to n, which is **exponential** in the size of n.

  – Only very small integers can be tested for primality by such

a brute-force approach even though it is unconditionally guaranteed to yield the correct answer.

– Hence the great interest by **all** (the governments, the scientists, the commercial enterprise, etc.) in discovering a computationally efficient algorithm for testing for primality that guarantees its result unconditionally.

- So when on **August 8, 2002** The New York Times broke the story that the trio of Manindra Agrawal, Neeraj Kayal, and Nitin Saxena (all from the Indian Institute of Technology at Kanpur) had found a computationally efficient algorithm that returned an unconditionally guaranteed answer to the primality test, it caused a big sensation.

Back to TOC

11.6.1 Generalization of Fermat's Little Theorem to Polynomial Rings Over Finite Fields

- The Agrawal-Kayal-Saxena (AKS) algorithm is based on the following generalization of Fermat's Little Theorem to polynomial rings over finite fields. [See Lecture 6 for what a polynomial ring is.] This generalization states that if a number a is coprime to another number p, $p > 1$, then p is prime **if and only if** the **polynomial** $(x + a)^p$ defined over the finite field Z_p obeys the following equality:

$$(x + a)^p \equiv x^p + a \pmod{p} \tag{6}$$

Pay particular attention to the '**if and only if**' clause in the statement above the equation. That implies that the equality in Eq. (6) is both a **necessary** and a **sufficient** condition for p to be a prime. It is this fact that allows the AKS test for primality to be deterministic. By contrast, Fermat's Little Theorem is only a necessary condition for the p to be prime. Therefore, a test based directly on Fermat's Little Theorem — such as the Miller-Rabin test — can only be probabilistic in the sense explained earlier.

- To establish Eq. (6), we can expand the binomial $(x + a)^p$ as follows:

$$(x + a)^p = \binom{p}{0} x^p + \binom{p}{1} x^{p-1} \cdot a + \binom{p}{2} x^{p-2} \cdot a^2 + \cdots + \binom{p}{p} a^p \quad (7)$$

  where the binomial coefficients are given by

$$\binom{p}{i} = \frac{p!}{i!(p-i)!}$$

- To prove Eq. (6) in the forward direction, suppose p is prime. Now the first and the last binomial coefficients in the expansion shown in Eq. (7) will both be 1. That is, $\binom{p}{0}$ and $\binom{p}{p}$ will both equal 1. [Note that 0! is by convention equal to 1 since it involves no factors for multiplication. (If you are curious about the convention, see the Wikipedia page on factorials.)] The other binomial coefficients, since they contain p as a factor, will obey

$$\binom{p}{i} \equiv 0 \pmod{p}$$

  Also that since by Fermat's Little Theorem we have $a^{p-1} \equiv 1 \ (mod\ p)$, the a^p in the last term in the binomial expansion reduces to just a. As a result, the expansion in Eq. (7) reduces to the form shown in Eq. (6).

- To prove Eq. (6) in the opposite direction, suppose p is composite. It then has a prime factor $q > 1$. Let q^k be the greatest power of q that divides p. Then q^k does NOT divide the binomial coefficient $\binom{p}{q}$. That is because this binomial coefficient has factored out of it some power of q and therefore the binomial coefficient cannot have q^k as one of its factors. [To make the same assertion contrapositively, let's assume for a moment that q^k is a factor of $\binom{p}{q}$. Then it must be the case that a larger power of q can divide p which is false by the assumption about k.] We also note that q^k must be coprime to a^{p-q} since we started out with the assumption that a and p were coprimes, implying that a and p cannot share any factors (except for the number 1). Now the coefficient of the term x^q in the binomial expansion is

$$\binom{p}{q} \cdot a^{p-q}$$

  We have identified a factor of p, the factor being q^k, that does **not** divide $\binom{p}{q}$ and and that is a coprime to a^{p-q}. For the coefficient of x^q to be 0 mod p, it must be divisible by p. But for that to be the case, the coefficient must be divisible by all factors of p. But we have just identified a factor, q^k, that divides neither $\binom{p}{q}$ not a^{p-q}. Therefore, the coefficient of x^q **cannot** be 0 mod p. This establishes the proof of Eq. (6) in the opposite direction, since we have shown that when p is **not** a prime, the equality in Eq. (6) does not hold.

- The generalization of Fermat's Little Theorem can be used directly for primality testing, but it would **not** be computationally efficient since it would require we check each of the p coefficients in the expansion of $(x + a)^p$ for some a that is coprime to p.

- There is a way to make this sort of primality testing more efficient by making use of the fact that if

$$f(x) \bmod p \quad = \quad g(x) \bmod p \tag{8}$$

then

$$f(x) \bmod h(x) \quad = \quad g(x) \bmod h(x) \tag{9}$$

where $f(x)$, $g(x)$, and $h(x)$ are polynomials whose coefficients are in the finite field Z_p. (But bear in mind the fact that whereas Eq. (8) implies Eq. (9), the reverse is **not** true.)

- As a result, the primality test of Equation (4) can be expressed in the following form for some value of the integer r:

$$(x + a)^p \bmod (x^r - 1) \quad = \quad (x^p + a) \bmod (x^r - 1) \tag{10}$$

with the caveat that there will exist some **composite** p for which this equality will also hold true. So, when p is known to be a prime, the above equation will be satisfied by all a coprime to p

and by all r. However, when p is a composite, this equation will be satisfied by some values for a and r.

- The main AKS contribution lies in showing that, when r is chosen appropriately, if Eq. (10) is satisfied for appropriately chosen values for a, then p is guaranteed to be a prime. **The amount of work required to find the value to use for r and the number of values of a for which the equality in Eq. (10) must be tested is bounded by a polynomial in $\log p$.**

Back to TOC

11.6.2 The AKS Algorithm: The Computational Steps

```
p  =  integer to be tested for primality
if ( p == a^b for some integer a and for some integer b > 1 ) :
        then return ‘‘p is COMPOSITE’’
r = 2

### This loop is to find the appropriate value for the number r:
while r < p:
    if ( gcd(p,r) is not 1 ) :                                  # (A)
        return "p is COMPOSITE"

    if ( r is a prime greater than 2 ):
        let q be the largest factor of r-1
        if ( q > (4 . sqrt(r) . log p) )   and
           ( p^{(r-1)/q} is not 1 mod r ) :
             break
    r = r+1

### Now that r is known, apply the following test:
for a = 1 to (2 . sqrt(r) . log p) :
    if ( (x-a)^p is not (x^p - a) mod (x^r - 1):                #(B)
        return "p is COMPOSITE"

return "p is PRIME"
```

There are two main challenges in creating an efficient implementation from the pseudocode shown above:

- For large candidate numbers, the number of iterations of the

`while` loop for finding an appropriate value for r may be large enough to require that you use the binary GCD algorithm in Section 5.4.4 of Lecture 5 — as opposed to the regular Euclid's algorithm also presented in the same section.

- Your main challenge is going to be to carry out what looks like computer algebra in line (B) where you are supposed to figure out whether, for the given value for a, the polynomial $(x-a)^p$ is congruent to the polynomial $x^p - a$ modulo the polynomial $x^r - 1$. Barring an implementation of this step as an exercise in computer algebra, how does one do that? One way to implement this step is by using logic that is similar to what was shown in Section 7.9 of Lecture 7 where we talked about polynomial multiplications modulo the irreducible polynomial for AES. Accordingly, as we raise $(x-a)$ to successively larger powers, the modulo $x^r - 1$ effect would come into play only when the exponent of $(x-a)$ is r or larger. Starting with $(x-a)^r$, its expansion has only one term to which the modulo operation needs to be applied and that term is x^r. So if we pre-calculate the value $x^r \; mod \; (x_r - 1)$, with the coefficients manipulated in the field Z_p, we can find out what $(x-a)^r \; mod \; (x^r - 1)$ is easily. If we now multiply this result by $(x-a)$ and use similar logic as in the previous step, we obtain $(x-a)^{(r+1)} \; mod \; (x^r - 1)$ easily; and so on.

Back to TOC

11.6.3 Computational Complexity of the AKS Algorithm

- The computational complexity of the AKS algorithm is

$$O\left((\log p)^{12} \cdot f(\log\log p)\right)$$

  where p is the integer whose primality is being tested and f is a polynomial. So the running time of the algorithm is **proportional** to the twelfth power of the number of bits required to represent the candidate integer times a polynomial function of the logarithm of the number of bits.

- There exist proposals for alternative implementations of the AKS algorithm for which the running time approaches the fourth power of the number of bits required to represent the number.

Back to TOC

11.7 THE CHINESE REMAINDER THEOREM (CRT)

- Discovered by the Chinese mathematician Sun Tsu Suan-Ching around 4^{th} century A.D. Particularly useful for modulo arithmetic operations on very large numbers with respect to large moduli.

- CRT says that in modulo M arithmetic, if M can be expressed as a product of n integers that are pairwise coprime, then every integer in the set $Z_M = \{0, 1, 2, ..., M-1\}$ can be reconstructed from residues with respect to those n numbers. [In all examples of modulo arithmetic so far in this lecture series, the modulus M has been prime. But now we are considering a modulus that is a composite. As you will see in the next lecture, in the famous RSA algorithm for public-key cryptography, the modulus M is a product of two primes, and therefore a composite.]

- For example, the prime factors of 10 are 2 and 5. Now let's consider an integer 9 in Z_{10}. Its residue modulo 2 is 1 and the residue modulo 5 is 4. So, according to CRT, 9 can be represented by the tuple (1, 4). As to why that's a useful thing to do, you'll soon see.

- Let us express a decomposition of M into factors that are

pairwise coprime by

$$M \quad = \quad \prod_{i=1}^{k} m_i$$

Therefore, the following must be true for the factors: $gcd(m_i, m_j) = 1 \quad \text{for } 1 \leq i, j \leq k \text{ and } i \neq j$. As an example of such a decomposition, we can express the integer 130 as a product of 5 and 26, which results in $m_1 = 5$ and $m_2 = 26$. Another way to decompose the integer 130 would be express it as a product of 2, 5, and 13. For this decomposition, we have $m_1 = 2$, $m_2 = 5$ and $m_3 = 13$.

- CRT allows us to represent any integer A in Z_M by the k-tuple:

$$A \equiv (a_1, a_2, \ldots, a_k)$$

where each $a_i \in Z_{m_i}$, its exact value being given by

$$a_i \quad = \quad A \bmod m_i \qquad for \;\; 1 \leq i \leq k$$

Note that each a_i can be any value in the range $0 \leq a_i < m_i$.

- CRT makes the following two assertions about the k-tuple representations for integers:

– The mapping between the integers $A \in Z_M$ and the k-tuples is a **bijection**, meaning that the mapping is one-to-one and onto. That is, there corresponds a **unique** k-tuple for every integer in Z_M and vice versa. (More formally, the bijective mapping is between Z_M and the Cartesian product $Z_{m_1} \times Z_{m_2} \times \ldots Z_{m_k}$.)

– Arithmetic operations on the numbers in Z_M can be carried out equivalently on the k-tuples representing the numbers. When operating on the k-tuples, **the operations can be carried out independently on each of coordinates of the tuples,** as represented by

$$
\begin{aligned}
(A \ + \ B) \bmod M \quad &\Leftrightarrow \quad ((a_1 \ + \ b_1) \bmod m_i, \ldots, (a_k \ + \ b_k) \bmod m_k) \\
(A \ - \ B) \bmod M \quad &\Leftrightarrow \quad ((a_1 \ - \ b_1) \bmod m_i, \ldots, (a_k \ - \ b_k) \bmod m_k) \\
(A \ \times \ B) \bmod M \quad &\Leftrightarrow \quad ((a_1 \ \times \ b_1) \bmod m_i, \ldots, (a_k \ \times \ b_k) \bmod m_k)
\end{aligned}
$$

where $A \Leftrightarrow (a_1, \ a_2, \ldots, \ a_k)$ and $B \Leftrightarrow (b_1, \ b_2, \ldots, \ b_k)$ are two arbitrary numbers in Z_M.

- To compute the number A for a given tuple $(a_1, \ a_2, \ldots, \ a_k)$, we first calculate $M_i = M/m_i$ for $1 \leq i \leq k$. Since each M_i has for its factors all the other prime moduli $m_j, \ j \neq i$, it must be the case that

$$M_i \equiv 0 \pmod{m_j} \qquad for \ \ all \ \ j \neq i$$

Very loosely speaking, you could say that each M_i is invisible in all other sets of residues. Let's now construct a sequence of numbers c_i, $1 \leq i \leq k$, in the following manner

$$c_i \quad = \quad M_i \times (M_i^{-1} \bmod m_i) \quad \quad for \;\; all \;\; 1 \leq i \leq k$$

Since M_i is coprime to m_i, there must exist a multiplicative inverse for M_i mod m_i. [The equation above is a bit disconcerting at first sight since it seems that the right hand side should equal 1 as we are multiplying M_i with M_i^{-1}. But note that we are interpreting the first operand M_i in modulo M arithmetic and not in modulo m_i arithmetic.]

- Now we can write the following formula for obtaining A from the tuple $(a_1,\; a_2, \ldots,\; a_k)$:

$$A \quad = \quad \left(\sum_{i=1}^{k} a_i \times c_i \right) \bmod M$$

To see the correctness of this formula, we must show that 'A mod m_i' produces a_i for $1 \leq i \leq k$. This follows from the fact that $M_j \bmod m_i = 0,\;\; j \neq i,$ implying that $c_j \bmod m_i = 0,\;\; j \neq i,$ and the fact that $c_i \bmod m_i = 1$.

Back to TOC

11.7.1 A Demonstration of the Usefulness of CRT

- CRT is extremely useful for manipulating very large integers in modulo arithmetic. We are talking about integers with over 150 decimal digits (that is, numbers potentially larger than 10^{150}).

- To illustrate the idea as to why CRT is useful for manipulating very large numbers in modulo arithmetic, let's consider an example that can be shown on a slide.

- Let's say that we want to do arithmetic on integers modulo 8633. That is, $M = 8633$. This modulus has the following decomposition into two pairwise coprimes:

$$8633 \quad = \quad 89 \times 97$$

  So we have $m_1 = 89$ and $m_2 = 97$. The corresponding M_i integers are $M_1 = M/m_1 = 97$ and $M_2 = M/m_2 = 89$.

- By using the Extended Euclid's Algorithm (see Lecture 5), we can next figure out the multiplicative inverse for M_1 modulo m_1 and the multiplicative inverse for M_2 modulo m_2. (These multiplicative inverses are guaranteed to exist since M_1 is

coprime to m_1, and M_2 is coprime to m_2.) We have [You could call the Python script `FindMI.py` in Section 5.7 of Lecture 5 to get the following MI values.]

$$\begin{aligned} M_1^{-1} \bmod m_1 &= 78 \\ M_2^{-1} \bmod m_2 &= 12 \end{aligned}$$

You can verify the correctness of the two multiplicative inverses by showing that $97 \times 78 \equiv 1 (\text{mod } 89)$ and that $89 \times 12 \equiv 1 (\text{mod } 97)$.

- Now let's say that we want to add two integers 2345 and 6789 modulo 8633.

- We first express the operand 2345 by its CRT representation, which is $(31,\ 17)$ since $2345 \bmod 89 = 31$ and $2345 \bmod 97 = 17$.

- We next express the operand 6789 by its CRT representation, which is $(25,\ 96)$ since $6789 \bmod 89 = 25$ and $6789 \bmod 97 = 96$.

- To add the two "large" integers, we simply add the two corresponding CRT tuples modulo the respective moduli. This gives us $(56,\ 16)$. For the second of these two numbers, we initially get 113, which modulo 97 is 16.

- To recover the result as a single number, we use the formula

$$a_1 \times M_1 \times M_1^{-1} \quad + \quad a_2 \times M_2 \times M_2^{-1} \qquad \text{mod } M$$

  which for our example becomes

$$56 \times 97 \times 78 \quad + \quad 16 \times 89 \times 12 \qquad \text{mod } 8633$$

  that returns the result 501. You can verify this result by directly computing $2345 + 6789 \bmod 8633$ and getting the same answer.

- For the example we worked out above, we decomposed the modulus M into its prime factors. In general, it is sufficient to decompose M into factors that are coprimes on a pairwise basis.

- In the next lecture, we will see how CRT is used in a computationally efficient approach to modular exponentiation, which is a key step in public key cryptography.

Back to TOC

11.8 DISCRETE LOGARITHMS

- First let's define what is meant by a **primitive root modulo a positive number** N.

- You already know that when p is a prime, the set of remainders, Z_p, is a finite **field**.

- We can show similarly that for **any positive** integer N, the set of all integers $i < N$ that are coprime to N form a **group** with modulo N **multiplication** as the **group operator**.

  [Note again we are talking about a group with a multiplication operator, and NOT a ring with a multiplication operator, NOR a group with an addition operator.]

- For example, when $N = 8$, the set of coprimes is $\{1, 3, 5, 7\}$. This set forms a group with modulo N multiplication as the group operator. What that implies immediately is that the result of multiplying modulo N any two elements of the set is contained in the set. For example, $3 \times 7 \bmod 8 = 5$. The identity element for the group operator is, of course, 1. And every element has its inverse with respect to the identity element within the set. For example, the inverse of 3 is 3 itself since $3 \times 3 \bmod 8 = 1$. (By the way, each element of

$\{1,\ 3,\ 5,\ 7\}$ is its own inverse in this group.)

- For any positive integer N, the set of all coprimes modulo N, along with modulo N multiplication as the group operator, forms a group that is denoted $(Z/NZ)^{\times}$. When $N = p$, that is, when N is a prime, we will denote this group by Z_p^*. [IMPORTANT: Z_p^* is NOT to be confused with Z_p. The two structures are very, very different. Whereas Z_p is a finite field in which every integer is represented. For example, all multiples of p are represented by 0 in Z_p. On the other hand, Z_p^* is merely a group that consist of just the $p-1$ integers in the set $\{1, 2, 3, \cdots, p-1\}$. Z_p^* is frequently referred to as a *multiplicative group of order* $p-1$. The order of a group is the number of elements in the group.] [With regard to the notation $(Z/NZ)^{\times}$, where the superscript is the multiplication symbol, the superscript is important for what we want this notation to stand for. Without the superscript, that is when your notation is merely Z/NZ, the notation is used by many authors to mean the same thing as Z_N, that is, the set of remainders modulo N along with the modulo N addition as the group operator.] In the previous example, we have $(Z/8Z)^{\times} = \{1, 3, 5, 7\}$. Choosing a prime for N, for another example we have $Z_{17}^* = \{1, 2, 3, \cdots, 16\}$.

- For some values of N, the set $(Z/NZ)^{\times}$ contains an element whose various powers, when computed modulo N, are all distinct and span the entire set $(Z/NZ)^{\times}$. Such an element is called the **primitive element** of the set $(Z/NZ)^{\times}$ or **primitive root modulo** N.

- Consider, for example, $N = 9$. We have

$$\begin{aligned} Z_9 &= \{0,\ 1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7,\ 8\} \\ (Z/9Z)^{\times} &= \{1,\ 2,\ 4,\ 5,\ 7,\ 8\} \end{aligned}$$

Now we will show that 2 is a **primitive element** of the group $(Z/9Z)^{\times}$, which is the same as **primitive root mod 9**. Consider the following **consecutive powers** of 2:

$$\begin{aligned} 2^0 &= 1 \\ 2^1 &= 2 \\ 2^2 &= 4 \\ 2^3 &= 8 \\ 2^4 &\equiv 7 \pmod 9 \\ 2^5 &\equiv 5 \pmod 9 \\ \cdots &\quad \cdots \quad \cdots \\ 2^6 &\equiv 1 \pmod 9 \\ 2^7 &\equiv 2 \pmod 9 \\ 2^8 &\equiv 4 \pmod 9 \\ &\vdots \end{aligned}$$

- It is clear that for the group $(Z/9Z)^{\times}$, as we raise the element 2 to all possible powers of the elements of Z_9, we recover all the elements of $(Z/9Z)^{\times}$. That makes 2 a primitive root mod 9.

- A primitive root can serve as the base of what is known as a **discrete logarithm**. Just as we can express $x^y = z$ as $\log_x z = y$, we can express

$$x^y \equiv z \pmod{N}$$

as

$$dlog_{x,N}\ z \ = \ y$$

- Therefore, the table shown on the previous page for the powers of 2 can be expressed as

$$\begin{aligned}
dlog_{2,9}\ 1 &= 0 \\
dlog_{2,9}\ 2 &= 1 \\
dlog_{2,9}\ 4 &= 2 \\
dlog_{2,9}\ 8 &= 3 \\
dlog_{2,9}\ 7 &= 4 \\
dlog_{2,9}\ 5 &= 5 \\
\ldots \quad &\ldots \quad \ldots \\
dlog_{2,9}\ 1 &= 6 \\
dlog_{2,9}\ 2 &= 7 \\
dlog_{2,9}\ 4 &= 8 \\
\ldots &
\end{aligned}$$

- It should follow from the above discussion that unique discrete logarithm mod N to some base a exists only if a is a primitive root modulo N.

Back to TOC

11.9 HOMEWORK PROBLEMS

1. What is the relationship between Euler's Theorem and Fermat's Little Theorem?

2. Intuitively speaking, primality testing seems trivial. Why? But, practically speaking, primality testing is extremely difficult for large numbers. Why?

3. Shown below is a naive approach to the implementation of the following primality test

$$a^{p-1} \equiv 1 \ (mod\ p)$$

where p is a candidate prime and a a probe:

```
p = int(sys.argv[1])
assert p > 17

probes = [2,3,5,7,11,13,17]

for a in probes:
    product = 1
    for _ in range(p-1):
        product *= a
    if product % p != 1:
        print "%d is NOT a prime" % p
        sys.exit(0)
print "%d is a prime" % p
```

Can this implementation really be used for testing a p that has so many decimal digits in it that it fills up half a page?

Assuming you have not yet heard of the Miller-Rabin test, how would you make the above code more efficient? And why would that not be efficient enough for practical applications?

4. The smarter way to implement the primality test takes advantage of the factorization:

$$p - 1 \quad = \quad 2^k \times q$$

where q is an odd integer. What's the commonly used programming idiom to find k and q for a given prime number candidate p?

5. You already know about the Fermat's Little Theorem (FLT) that is used for primality testing:

$$a^{p-1} \quad \equiv \quad 1 \;\; (mod\; p)$$

where p is a candidate prime and a a probe. If the test fails, we are sure that p is not a prime. However, if the test succeeds, with a probability of approximately 1/4, there is a chance that p is a composite.

Therefore, if the test succeeds, you choose another probe a and repeat the test. If this test fails, you are sure p is not a prime.

However, if the test succeeds, with a probability of $(1/4)^2$, there is a chance that p is a composite.

You continue in this manner until either the test fails or until the probability that a composite is masquerading as a prime is sufficiently small.

Considering that we have very fast algorithms for *gcd* computation, why can't our probabilistic testing strategy be based directly on the test that if p is a prime, then

$$gcd(p, a) \quad = \quad 1$$

for all values for the probe a, $1 \leq a < p$? As in the implementation of the test based on FLT, an implementation of this test based on *gcd* could conceivably use a set of randomly selected values for a.

6. The AKS primality test is based on what generalization of the Fermat's Little Theorem?

7. As a small illustration of the Chinese Remainder Theorem (CRT) that can all be solved mentally, say $M = 30$. Let's say we express this M as the product of the pairwise coprimes 2, 3, and 5. That is, $m_1 = 2$, $m_2 = 3$, and $m_3 = 5$. Given that the numbers involved are small, you should be able to fill the following table with just mental calculations. [As you know from

Section 11.7, the entries you place in the last column will be your reconstruction coefficients, c_1, c_2, c_3. Let's say (p_1, p_2, p_3) is your CRT representation of large integer I. That is, $p_i = I \ mod \ m_i$. You can recover I from its CRT representation by $I = (\sum_1^3 c_i p_i) \ mod \ M$.]

m_i	M_i	$M_i^{-1} \bmod m_i$	$M_i \times (M_i^{-1} \bmod m_i)$
2 3 5			

After you are done filling the table, calculate $(75 + 89)$ mod 30, (75×89) mod 30, etc., using the Chinese Remainder Theorem. Verify your answers by direct computations on the operands in each case.

8. What is difference between the notation Z_N and the notation $(Z/NZ)^\times$?

9. We say that the element 2 is a primitive root of the set $(Z/9Z)^\times$. What does that mean?

10. What is discrete logarithm and when can we define it for a set of numbers?

11. **Programming Assignment:**

Expand one of the primality testing scripts shown in Section 11.5.5 into a script for generating prime numbers whose bit representations are of specified size. The two main parts of such a script will be: (1) generation of an appropriate random number of the required bit-field width; and (2) testing of the random number with an appropriate script from Section 11.5.5. If you are doing this homework in Python, for the first part you can invoke `random.getrandombits( bitfield_width )` to give you a random integer whose bit-field is limited to size `bitfield_width`. Once you have gotten hold of such an integer, you would need to set its lowest bit, so that it is odd, and the highest bit to make sure that its bit field spans the full size you want. [As will become clear in Lecture 12, in some cases you may need to set the two highest bits, as opposed to just the highest bit.] Shown below is a code fragment that does all of these things:

```
candidate = random.getrandbits( bitfield_width )
if candidate & 1 == 0: candidate += 1
candidate |= (1 << bitfield_width - 1)
candidate |= (2 << bitfield_width - 3)
```

where you need the last statement only if you wish to set the two most significant bits. Subsequently, should this `candidate` prime prove to be a composite, you can increment it by 2 and try again. As you are dubugging your script, you may wish to print out the bit patterns generated by the calls shown above using a statement like:

```
print format(candidate, '064b')
```

assumign that you are generating 64 bit primes.

Should you choose to do this homework in Perl, the statements that have roughly the same behavior as shown above for Python would be:

```
@arr = map {my $x = rand(1); $x > 0.5 ? 1 : 0 } 0 .. $bitfield_width - 4;
push @arr, 1;
unshift @arr, (1,1);
$bstr = join '', split /\s/, "@arr";
$candidate = oct("0b".$bstr);
```

Use your script to generate 64-bit wide and 128-bit wide prime numbers. [**HINT:** The Python and Perl solutions to this problem are presented in the Homework Problems section of Lecture 12. However, try not to look at those solutions before creating your own solution.]