Computer Science 1001.py

Lecture 12: Randomized Primality Testing (cont.)
Diffie Hellman Public Exchange of Secret Key
Integer gcd: Euclid's Algorithm

Instructors: Benny Chor, Amir Rubinstein
Teaching Assistants: Yael Baran, Michal Kleinbort

Founding Teaching Assistant (and Python Guru): Rani Hod

School of Computer Science
Tel-Aviv University
Spring Semester, 2014
http://tau-cs1001-py.wikidot.com

Lecture 11: Highlights

- The `Hanoi Towers` Monster: Finding individual moves.
- Binary search to the rescue.
- Tail recursion.

- Trial division and its computational complexity.
- Efficient modular exponentiation (a very brief reminder).
- Fermat's little theorem.
- Randomized primality testing.

Lecture 12: Plan

- Clarification: computational complexity wrt integer inputs.
- Randomized primality testing, recapping.

- Cryptography: Secure communication over insecure communication lines.
- The discrete logarithm problem.
- One-way functions.
- Diffie-Hellman scheme for secret key exchange over insecure communication lines.

- Integer greatest common divisor (gcd).
- Euclid's gcd algorithm.
- Approximating π using integer gcd.

Computation Complexity for Integer Inputs: Clarification

- We measure running time (or computational complexity) as a function of the input length.
- Input length is the number of bits in the representation of the input in the computer.
- In the computer, integers are represented in binary, and certainly not in unary.

- The number of bits in the representation of the positive integer M is not M.
- The number of bits in the representation of the positive integer M is $\lfloor \log_2(M) \rfloor + 1$.
- For example, the representations of both 10 and 15 are $\lfloor \log_2(10) \rfloor + 1 = \lfloor \log_2(15) \rfloor + 1 = 3 + 1 = 4$ bits long.

Computation Complexity for Integer Inputs, cont.

- We measure running time (or computational complexity) as a function of the input length.
- Suppose the positive integer M is n bits long.
- And we designed an algorithm whose running time is $\sqrt{M}$.
- Is this a polynomial time algorithm?

Computation Complexity for Integer Inputs, cont.

- We measure running time (or computational complexity) as a function of the input length.
- Suppose the positive integer M is n bits long.
- And we designed an algorithm whose running time is $\sqrt{M}$.
- Is this a polynomial time algorithm?

- No!, no!, and no!
- M is n bits long means $2^{n-1} \leq M \leq 2^n - 1$.
- So $2^{(n-1)/2} \leq \sqrt{M}$.
- $2^{(n-1)/2}$ is exponential in the input length, n. It is not polynomial in n.

Testing Primality/Compositeness (reminder)

- Now that we know, by the prime number theorem, that there are heaps of primes, we would like to efficiently test if a given integer is prime or composite.
- Given an n bits integer m, $2^{n-1} \leq m < 2^n$, we want to determine if m is composite or prime.

Testing Primality/Compositeness (reminder)

Basic Idea [Solovay-Strassen, 1977]: To show that m is composite, enough to find evidence that m does not behave like a prime.

Such evidence need not include any prime factor of m.

Fermat's Little Theorem

Let p be a prime number, and a any integer in the range $1 \leq a \leq p-1$.

Then $a^{p-1} = 1 \pmod{p}$.

Fermat's Little Theorem, Applied to Primality

By Fermat's little theorem, if p is a prime and a is in the range $1 \leq a \leq p-1$, then $a^{p-1} = 1 \pmod{p}$.

Suppose that we are given an integer, m, and for some a in in the range $2 \leq a \leq m-1$, $a^{m-1} \neq 1 \pmod{m}$.

Such an a supplies a concrete evidence that m is composite (but says nothing about m's factorization).

Randomized Primality Testing (reminder)

- The input is an integer m with n bits (namely $2^{n-1} < m < 2^n$)
- Repeat 100 times
  - Pick a in the range $1 \leq a \leq m-1$ at random and independently.
  - Check if a is a witness ($a^{m-1} \neq 1 \bmod m$)
    (Fermat test for a, m).
- If one or more a is a witness, output "m is composite".
- If no witness found, output "m is prime".

Remark: This idea, which we term the Fermat primality test, is based upon seminal works of Solovay and Strassen in 1977, and Miller and Rabin, in 1980.

Properties of Fermat Primality Testing

- Randomized: uses coin flips to pick the a's.
- Run time is polynomial in n, the length of m.
- If m is prime, the algorithm always outputs "m is prime".
- If m is composite, the algorithm may err and outputs "m is prime".
- So the algorithm may introduce a one sided error.

Properties of Fermat Primality Testing, cont.

- Miller-Rabin showed that if m is composite, then at least 3/4 of all $a \in \{1, \ldots, m-1\}$ are witnesses.
- To err, all random choices of a's should yield non-witnesses. Therefore,

$$\text{Probability of error} < \left(\frac{1}{4}\right)^{100} \lll 1 \ .$$

- Note: With much higher probability the roof will collapse over your heads as you read this line, an atomic bomb will go off within a 1000 miles radius (maybe not such a great example in November 2011, or in April 2015), etc., etc.

Pushing Your Machine to the Limit

You may try to verify that the largest known prime (so far) is indeed prime. But do take it easy. Even one witness will push your machine way beyond its computational limit.

It is a good idea to think why this is so.

```
>>> m=2**57885161-1
>>> pow(56,m-1,m)==1
     # patience, young lads!
     # and even more patience!!
```

Hint: Think of the complexity of computing $a^b \bmod c$ where all three numbers are ℓ bit long. And recall that for this large prime, $\ell = 57,885,161$.

Trust, But Check!

We said (quoting Miller and Rabin) that if if m is composite, then at least 3/4 of all $a \in \{1, \ldots, m-1\}$ are witnesses.

This is almost true.

There are some annoying numbers, known as Carmichael numbers, where this does not happen.

However:

- These numbers are very rare and it is highly unlikely you'll run into one, unless you really try hard.
- A small (and efficient) extension of Fermat's test takes care of these annoying numbers as well.
- If you want the details, you will have to look it up, or take the elective crypto course.

Primality Testing: Practice and Theory

For all practical purposes, the randomized algorithm based on the Fermat test (and various optimizations thereof) supplies a satisfactory solution for identifying primes.

Still the question whether composites / primes can be recognized efficiently without tossing coins (in deterministic polynomial time, *i.e.* polynomial in n, the length in bits of m), remained open for many years.

Deterministic Primality Testing

In summer 2002, Prof. Manindra Agrawal and his Ph.D. students Neeraj Kayal and Nitin Saxena, from the India Institute of Technology, Kanpur, finally found a deterministic polynomial time algorithm for determining primality. Initially, their algorithm ran in time $O(n^{12})$. In 2005, Carl Pomerance and H. W. Lenstra, Jr. improved this to running in time $O(n^6)$.

Agrawal, Kayal, and Saxena received the 2006 Fulkerson Prize and the 2006 Gödel Prize for their work.

Fermat's Last Theorem (a cornerstone of Western civilization)

You are all familiar with Pythagorean triplets: Integers $a, b, c \geq 1$ satisfying

$$a^2 + b^2 = c^2$$

e.g. $a = 3, b = 4, c = 5$, or $a = 20, b = 99, c = 101$, etc.

Conjecture: There is no solution to

$$a^n + b^n = c^n$$

with integers $a, b, c \geq 1$ and $n \geq 3$.

In 1637, the French mathematician Pierre de Fermat, wrote some comments in the margin of a copy of Diophantus' book, Arithmetica. Fermat claimed he had a wonderful proof that no such solution exists, but the proof is too large to fit in the margin.

The conjecture mesmerized the mathematics world. It was proved by Andrew Wiles in 1993-94 (the proof process involved a huge drama).

Finding the Next Prime

```
def next_prime(start):
    """ find the first prime >= start, and the shift from start """
    for i in range(start,2*start):
        if is_prime(i):
            return i,i-start

>>> next_prime(2**100)
(1267650600228229401496703205653,277)
>>> next_prime(2**200)
(1606938044258990275541962092341162602522202993782792835301611,235)
>>> next_prime(2**300)
(2037035976334486086268445688409378161051468393665936250636140449
 354381299763336706183397533,157)
>>> next_prime(2**1000-1)
(1071508607186267320948425049060001810561404811705533607443750388370
...83720566806967 3, 298)
```

We hope you now believe the prime number theorem (and us :-).

שַׂחֲקִי, שַׂחֲקִי עַל הַחֲלוֹמוֹת,
זוּ אֲנִי הַחוֹלֵם שָׂח.
שַׂחֲקִי כִּי בָאָדָם אַאֲמִין,
כִּי עוֹדֶנִּי מַאֲמִין בָּךְ.

Primality Testing: Modified Python Code

If its input is composite, this code reveals if compositess was discovered by the small sieve (the input is divisible by a prime smaller than 30), or how many random witnesses were used.

```
import random  # random numbers package

def is_prime_show(m):
    """ probabilistic test for m's compositeness
    adds a trivial sieve to quickly eliminate divisibility
    by small primes. Prints number of witnesses. """
    for prime in [2,3,5,7,11,13,17,19,23,29]:
        if m % prime == 0:
            print("discarded by sieve")
            return None
    for i in range(0,100):
        a = random.randint(1,m-1) # a random integer in [1..m-1]
        if pow(a,m-1,m) != 1:
            print("no. attempts till compositeness proof =",i+1)
        return None
    return True  # True means m is prime
```

Running the Modified Code

Let us now run this code on some fairly large numbers:

```
>>> for i in range(14):
        m=random.randint(2**10000,2**10001-1) # 10001 bit numbers
        is_prime(m)

discarded by sieve
discarded by sieve
discarded by sieve
discarded by sieve
discarded by sieve
discarded by sieve
discarded by sieve
discarded by sieve
discarded by sieve
no. attempts till compositeness proof = 1
discarded by sieve
discarded by sieve
discarded by sieve
discarded by sieve
```

Most composites are (naturally) sieved out.

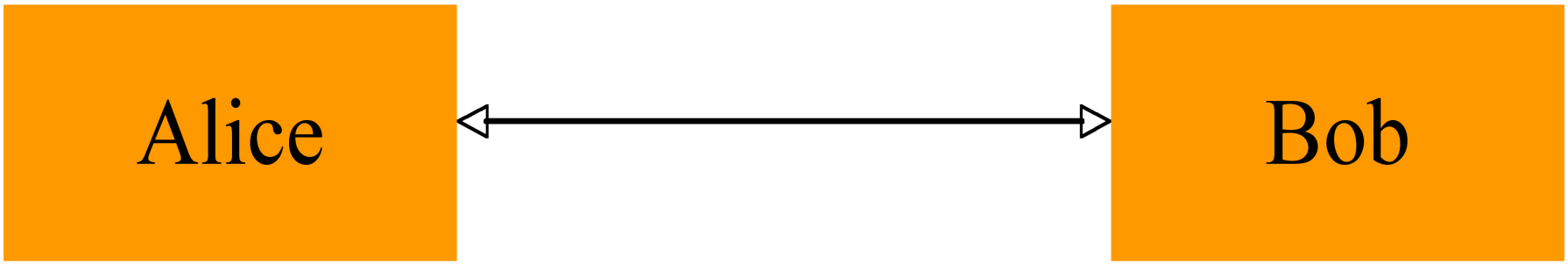
Most others are determined by just one random witness! (Finding one that requires more witnesses will take some effort.)

And Now For Something Completely Different: Encryption

taken at south west corner of Ramon crater, April 2015

Encryption: Basic Model

Let us welcome the two major players in this field, Alice and Bob (audience applauds and whistles).

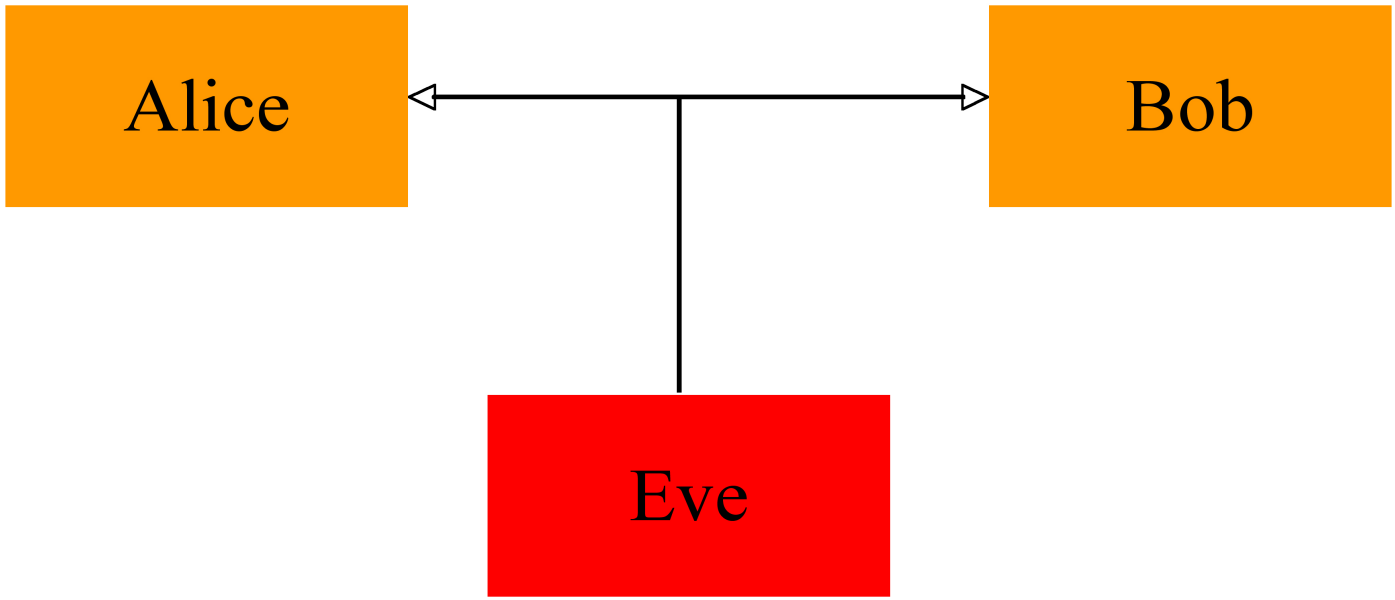


1. Two parties – Alice and Bob.
2. Reliable communication line.
3. Encryption algorithm, E.
4. Decryption algorithm, D.
5. Shared, secret key: $k_{A,B}$ (used both for encryption and decryption).

6. Goal: send a message M confidentially.

Adversarial Model: Passive Eavesdropper

Enters our third major player, Eve (claps again!).



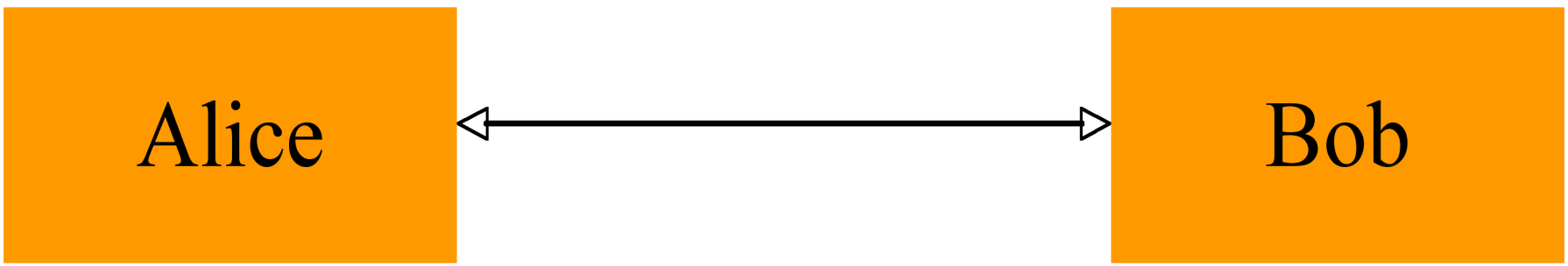
- Eve attempts to discover information about M
- Eve knows the algorithms E, D
- Eve knows the message space
- Eve has intercepted $E_{k_{A,B}}(M)$
- Eve does not know $k_{A,B}$

Additional Definitions (to complete the picture)

- Plaintext – the message prior to encryption ("attack at dawn", "sell short 6.5 billion £")
- Ciphertext – the message after encryption ("$\Im\partial\text{Æ}\perp\xi\varepsilon\beta\Xi\Omega\Psi\mathring{A}$", "jhhfo hjklvhgbljhg")

Classical, Symmetric Ciphers

- Alice and Bob share the same secret key, $k_{A,B}$.
- $k_{A,B}$ must be secretly generated and exchanged prior to using the insecure channel.



Major question, esp. at the internet era: How can Alice and Bob secretly generate and exchange $k_{A,B}$ if they have never physically met, they live on antipodal sides of the globe, and all communication lines are subject to eavesdropping?

New Directions in Cryptography (1976)

"We stand today on the brink of a revolution in cryptography. The development of cheap digital hardware has freed it from the design limitations of mechanical computing . . .

. . . such applications create a need for new types of cryptographic systems which minimize the necessity of secure key distribution . . .

. . . theoretical developments in information theory and computer science show promise of providing provably secure cryptosystems, changing this ancient art into a science."

– W. Diffie and M. Hellman, IEEE IT, vol. 22, no. 6, Nov. 1976.

(figures from Wikipedia)

Diffie and Hellman: New Directions in Cryptography (reference only)

In their seminal paper "New Directions in Cryptography", Diffie and Hellman suggest to split Bob's secret key k to two parts:

- k_E, to be used for encrypting messages to Bob.
- k_D, to be used for decrypting messages by Bob.
- k_E can be made public and be used by everybody.

This is public key cryptography, or asymmetric cryptography.

Diffie and Hellman suggested the notion of PKC, but had no concrete implementation.

Public key cryptography is surely not very intuitive at first sight.

However, we will not elaborate on it further. We refer the interested parties to the elective course in foundations of modern cryptography.

Diffie and Hellman: Public Exchange of Keys

Diffie and Hellman also proposed public exchange of keys.

Here, they did have a concrete implementation, based on the discrete logarithm problem.

Public Exchange of Keys

- Two parties, Alice and Bob, do not share any secret information.
- They execute a protocol, at the end of which both derive the same shared, secret key.
- Shared, secret key is $k_{A,B}$ (used both for encryption and decryption in a classical crypto system).
- A computationally bounded eavesdropper, Eve, who overhears all communication, cannot obtain the secret key or any new information about it.
- We assume Eve is passive (only listens).

Public Exchange of Keys: A Solid Metaphore

(Image from Dr. Seuss' Cat in the Hat.)

Discrete Log modulo p and One Way Functions

- Let p be a large prime (say 1024 bits long).
- Let g be a random integer in the range $1 < g < p - 1$.
- Let $x = g^i \bmod p$ for some $1 \leq i < p - 1$.

- The inverse operation, $x = g^i \mod p \mapsto i$ (called discrete log) is believed to be computationally hard.
- We say that the mapping $i \longrightarrow g^i \mod p$ is a one way function.
- This is a computational notion. With unbounded (or even just exponential) resources, one can invert this function (compute discrete log).

Diffie and Hellman Key Exchange

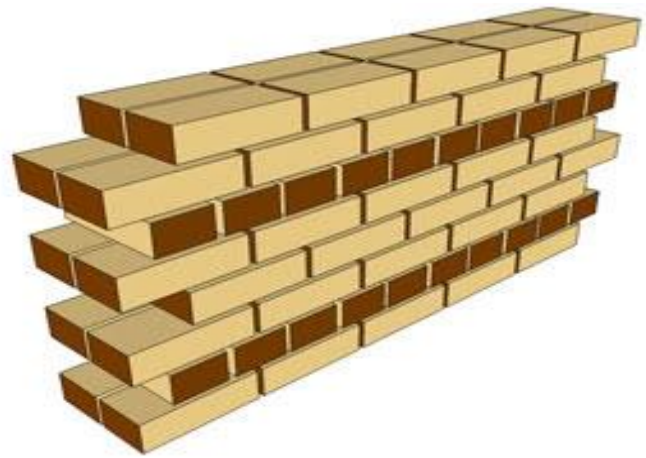
- Public parameters: A large prime p (1024 bit long, say) and a random element g in in the range $1 < g < p - 1$.
- Alice chooses at random an integer a from the interval $[2..p-2]$. She sends $x = g^a \pmod p$ to Bob (over the insecure channel).
- Bob chooses at random an integer b from the interval $[2..p-2]$. He sends $y = g^b \pmod p$ to Alice (over the insecure channel).

- Alice, holding a, computes $y^a = (g^b)^a = g^{ba} \pmod p$.
- Bob, holding b, computes $x^b = (g^a)^b = g^{ba} \pmod p$.
- Now both have the shared secret, $g^{ba} \pmod p$.
- An eavesdropper cannot infer the key, $g^{ba} \pmod p$ after seeing "only" $p,\ g,\ x = g^a \pmod p$ and $y = g^b \pmod p$ (under the assumption that discrete log is intractable).

- We have just witnessed a small miracle !

Diffie and Hellman Key Exchange: Artwork

Public: Large prime p, large g $(1<g<p)$



Alice

Secret: random a
$(1< a <p)$

Bob

Secret: random b
$(1< b <p)$

$x = g^a \bmod p$

computation (one pow each)

$y = g^b \bmod p$

communication over insecure channels

communication (one mssg each)

$y^a \bmod p$
$= (g^b \bmod p)^a \bmod p$
$= g^{ab} \bmod p$

computation (one pow each)

$x^b \bmod p$
$= (g^a \bmod p)^b \bmod p$
$= g^{ab} \bmod p$

Diffie and Hellman Key Exchange: Code (Centralized)

```
import random

def DH_exchange(p):
    """ generates a shared DH key """
    g=random.randint(1,p-1)
    a=random.randint(1,p-1)
    b=random.randint(1,p-1)
    x=pow(g,a,p)
    y=pow(g,b,p)
    key_A=pow(y,a,p)
    key_B=pow(x,b,p)
    return key_A,key_B,key_A-key_B  # y^a=x^b iff difference is 0

>>> p=next_prime(2**100)[0]
>>> DH_exchange(p)
(39077862285029579141502813208 3,39077862285029579141502813208 3, 0)
>>> DH_exchange(p)
(95694611970787670535133543470 1,95694611970787670535133543470 1, 0)
>>> DH_exchange(p)
(85683374601415666548159670601 1,85683374601415666548159670601 1, 0)
```

Modular Exponentiation: Clarifications

Questions about the order of exponentiation and mod p operations are often raised.

Well, all the following hold

- $((a \mod p) + (b \mod p)) \mod p = (a + b) \mod p$.
- $((a \mod p) \cdot (b \mod p)) \mod p = (a \cdot b) \mod p$.
- $(g^a \mod p)^b \mod p = (g^a)^b \mod p$.
- $(g^a \mod p)^b \mod p = (g^a)^b \mod p = g^{ab} \mod p$.

In fact, all these mod p operations are best viewed in the context of the finite field Z_p^*. But not being familiar with (mathematical) groups or fields, we have to think anew about mod p each time.

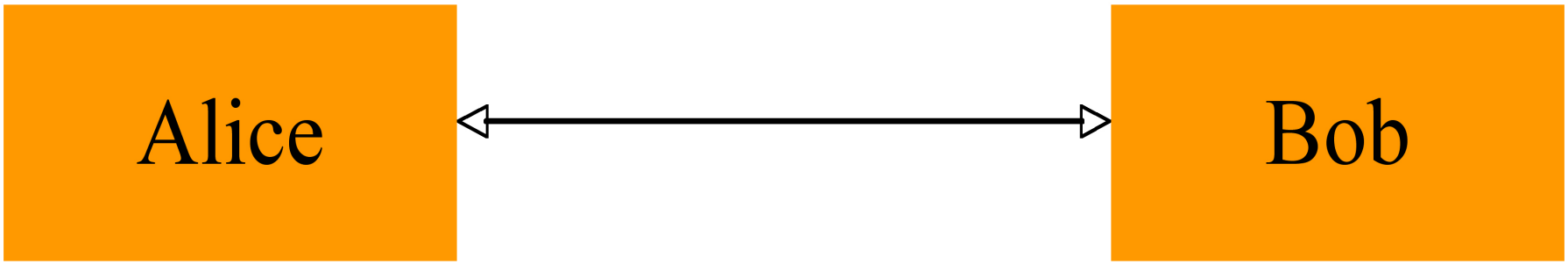
Diffie and Hellman – Final Remarks

- Recall that the length of the prime p in bits is $\lceil \log_2 p \rceil$.
- Computation time for exchanging the key is $O(\log_2^3 p)$ bit operations.

- DH key exchange is at most as secure as discrete log.
- Formal equivalence between DH (Diffie-Hellman key distribution) and DL (discrete logarithm problem) has never been proved, though some partial results are known.
- Over the last 36 years there were many attempts to crack the scheme. None succeeded, and DH key exchange (with an appropriately large prime p, *e.g.* 1024 bits) is considered secure.

- U.S. Patent 4,200,770, now expired, describes the algorithm and credits Hellman, Diffie, and Merkle as inventors.
- And the three of them have joined the Hall of Fame.

Classical Encryption and Diffie Hellman



1. Two parties – Alice and Bob.
2. Reliable communication line.
3. Encryption algorithm, E.
4. Decryption algorithm, D.
5. Shared, secret key: The shared key $y^a = x^b \pmod p$ generated by the Diffie Hellman protocol is used as $k_{A,B}$ in a classical, secret key crypto system (for both decryption and encryption).

6. Comment: To learn how $k_{A,B}$ is employed in a classical, secret key crypto system, we refer you to the elective crypto course.
7. We did not explain or exemplify how classical crypto works.

And Now For Something Completely Different: Integer Greatest Common Divisor (gcd)

View from Har Ardon, looking south, April 2015

Integer Greatest Common Divisor

Computing the integer greatest common divisor (gcd) is maybe the oldest computational problem we have a record for.

Integer gcd (and some ramifications) is tightly related to arithmetic of large integers, *e.g.* computing inverses modulo a large prime, p, which are important in modern cryptography.

Integer Greatest Common Divisor

The greatest common divisor, or gcd, of two positive integers k, ℓ is the largest integer, g, that is a divisor of both of them. Since 1 always divides k, ℓ , the gcd g is well defined.
If one of these two integers is zero, we define $\gcd(k, 0) = k$.

For example,

gcd(28,32)=4,

gcd(276,345)=69,

gcd(1001,973)=7,

gcd(1002,973)=1.

If $\gcd(k, \ell) = 1$, we say that k, ℓ are relatively prime.

Computing Greatest Common Divisor Naively

The naive approach to computing $\gcd(k, \ell)$ is similar to the trial division approach: Start with $\min(k, \ell)$, and iterate, going down, testing at each iteration if the current value divides both k and ℓ.

How far do we go? Till the first divisor is found.

What is the (worst case) running time of this naive method? When relatively prime, the number of trial divisions is exactly $\min(k, \ell)$. If the minimum is an n bit number, the running time is $O(2^n)$. Hence this method is applicable only to relatively small inputs.

Alternatively, we could also go up, starting with 1. It won't make much of a difference in the worst case.

Slow GCD Code

```
def slow_gcd(x,y):
    """ greatest common divisor of two integers -
    naive inefficient method """
    assert isinstance(x,int) and isinstance(y,int)
     # type checking: x and y both integers
    x,y=abs(x),abs(y) # simultaneous assignment to x and y
            # gcd invariant to abs. Both x,y now non-negative
    if x<y:
        x,y=y,x    # switch x and y if x < y. Now y <= x
    for g in range(y, 0, -1): # from x downward to 1
        if x%g == y%g == 0:  # does g divide both x and y?
            return g
    return None # should never get here, 1 divides all

>>> from clock import elapsed
>>> elapsed("slow_gcd(2**50,2**23+1)")
2.294258
>>> elapsed("slow_gcd(2**50,2**27+1)")
36.838267
```

Computing GCD – Euclid's Algorithm

Euclid, maybe the best known early Greek mathematician, lived in Alexandria in the 3rd century BC. His book, *Elements*, lays the foundation to so called Euclidean geometry, including an axiomatic treatment. The book also deals with number theory and describes an efficient gcd algorithm.

(drawing from Wikipedia)

Euclid's gcd algorithm is iterative, and is based on the following invariant (an invariant is a property that remains true, or a value that is unchanged, before and after applying some transformation): Suppose $0 < \ell < k$, then $\gcd(k, \ell) = \gcd(k \mod \ell, \ell)$.

The algorithm replaces the pair (k, ℓ) by $(\ell, k \pmod{\ell})$, and keeps iterating till the smaller of the two reaches zero. Then it uses the identity $\gcd(h, 0) = h$.

Computing GCD – Euclid's Algorithm (cont.)

Euclid's gcd algorithm is based on the following invariant:
Suppose $0 < \ell < k$, then $\gcd(k, \ell) = \gcd(\ell, k \pmod{\ell})$.

Notice that after taking the remainder, $k \pmod{\ell}$ is strictly smaller than ℓ. Thus one iteration of this operation reduces both numbers to be no larger than the original minimum.

Consider a specific example:

```
>>> k,l = 6438, 1902
>>> k,l = l,k%l ; print(k,l)# simultaneous assignment; then print
1902 732
>>> k,l = l,k%l ; print(k,l)
732 438
>>> k,l = l,k%l ; print(k,l)
438 294
>>> k,l = l,k%l ; print(k,l)
294 144
>>> k,l = l,k%l ; print(k,l)
144 6
>>> k,l = l,k%l ; print(k,l)
6 0
```

The gcd of 6438 and 1902 is 6.

Computing GCD – Euclid's Algorithm (cont.)

Euclid's gcd algorithm is based on the following invariant:
Suppose $0 < \ell < k$, then $\gcd(k, \ell) = \gcd(\ell, k \pmod{\ell})$.

It can be shown (proof omitted) that two iterations of this operation make both numbers smaller than half the original maximum.

Example:
k_0=4807526976, ℓ_0=2971215073
k_1=2971215073, ℓ_1=1836311903
k_2=1836311903, ℓ_2=1134903170
k_3=1134903170, ℓ_3=701408733
k_4=701408733, ℓ_4=433494437
$\vdots$

Suppose that originally k is an n bit number, namely $2^{n-1} \leq k < 2^n$. On every second iteration, the maximum number is halved. So in terms of bits, the length of the maximum becomes at least one bit shorter. Therefore, the number of iterations is at most $2n$.

Python Code – Euclid's Algorithm, Displaying

The following code computes `gcd(x,y)` using Euclid's algorithm. In addition, it prints all intermediate pairs.

```
def display_gcd(x,y):
    """ greatest common divisor of two integers, Euclid's algorithm.
    This function prints all intermediate results along the way. """
     assert isinstance(x,int) and isinstance(y,int)
            # type checking: x and y both integers
     x,y=abs(x),abs(y) # simultaneous assignment to x and y
              # gcd invariant to abs. Both x,y now non-negative
     if x<y:
         x,y=y,x    # switch x and y if x < y. Now y <= x
     print(x,y)
     while y>0:
         x,y=y,x%y
         print(x,y)
     return x
```

Python Code – Euclid's Algorithm, Displaying (Execusion)

```
>>> display_gcd(10946,6765)
10946 6765
6765 4181
4181 2584
2584 1597
1597 987
987 610
610 377
377 233
233 144
144 89
89 55
55 34
34 21
21 13
13 8
8 5
5 3
3 2
2 1
1 0
1     # final outcome -- gcd(10946,6765)
```

Python Code – Euclid's Algorithm, Displaying (Execution)

```
>>> display_gcd(6774,4227)
6774 4227
4227 2547
2547 1680
1680 867
867 813
813 54
54 3
3 0
3  # final outcome: gcd(6774,4227)=3
```

Non trivial question: Which pairs of n bit integers, `x,y`, cause a worst case performance (maximal number of iterations) for Euclid's gcd?

Python Code – Euclid's vs. Slow

As noted before, on n bit numbers, Euclid's algorithm takes at most $2n$ iterations, while slow gcd takes up to 2^n iterations.
Let us put this theoretical analysis to the ultimate test – the test of the clock. We note that we now consider a version of Euclid's algorithm which does not display intermediate results (code omitted).

```
>>> from gcd import *
>>> elapsed("gcd(2**50,2**27+1)",number=1000000) #million runs
26.885727999999997
>>> elapsed("slow_gcd(2**50,2**27+1)",number=1)
37.548531000000004
>>> slow_gcd(2**50,2**27+1)  # sanity check
1
>>> gcd(2**50,2**27+1)
1
```

Euclid's algorithm is almost 1.4 million times faster than the naive one for this input. So theory and practice do agree here.

Euclid's gcd: Proof of Correctness Using an Invariant*

Suppose $0 < \ell \leq k$. We first show that $\gcd(k, \ell) = \gcd(\ell, k - \ell)$.

- Denote $g = \gcd(k, \ell)$, $h = \gcd(\ell, k - \ell)$.
- Since g divides both k and ℓ, it also divides $k - \ell$.
- Thus it divides both ℓ and $k - \ell$.
- Since h is the greatest common divisor of ℓ and $k - \ell$, every divisor of ℓ and $k - \ell$ divides h (think of primes' powers).
- As g is a divisor of both, we conclude that g divides h.
- A similar argument shows that any divisor of ℓ and $k - \ell$ is also a divisor of k.
- Thus h divides the gcd of k and ℓ, which is g.

- So g divides h and h divides g.
- They are both positive, therefore g equals h. ♣

*a (different) invariant was used to prove correctness of the iterated squaring algorithm to compute a^b.

Euclid's gcd: Proof of Correctness Using an Invariant (cont.)

Suppose $0 < \ell \leq k$. We just showed that $\gcd(k, \ell) = \gcd(k - \ell, \ell)$.

- If $k - \ell < \ell$, then $k \pmod{\ell} = k - \ell$, and we are done.
- Otherwise, $k - \ell \geq \ell$. $\gcd(k, \ell) = \gcd(k - \ell, \ell)$.
- Repeating the previous argument, $\gcd(k - \ell, \ell) = \gcd(k - 2\ell, \ell)$.
- There is a unique $m \geq 1$ such that $k \pmod{\ell} = k - m\ell$.
- By the argument above,
  $\gcd(k, \ell) = \gcd(\ell, k - \ell) = \gcd(\ell, k - 2\ell) = \gcd(\ell, k - 3\ell) =$
  $\ldots = \gcd(\ell, k - m\ell) = \gcd(\ell, k \mod \ell)$ ♠

Proof of Correctness Using an Invariant: Conclusion

In every iteration of Euclid's algorithm, we replace (k, ℓ) by $(\ell, k \mod \ell)$, until the smaller number equals zero.

The claim above means that at each iteration, the gcd is invariant. At the final stage, when we have $(g, 0)$, we return their gcd, which equals g.

By this invariance, g indeed equals the gcd of the original (k, ℓ) $\diamondsuit$

Note: In case you were wondering, $\spadesuit$, $\diamondsuit$, $\heartsuit$, $\clubsuit$ will frequently be used (by us, that is) to denote end of proof. Alternatively and more commonly, end of proof is denoted by QED, or quod erat demonstrandum, meaning, in Latin, "which was to be demonstrated".

Relative Primality and Multiplicative Inverses

gcd(28,31)=1
gcd(12,35)=1
gcd(527,621)=1
gcd(1002,973)=1

If $\gcd(k, m) = 1$, we say that k, m are relatively prime.

Suppose k, m are relatively prime, and $k < m$.
Then there is a positive integer, a, $a < m$, such that

$$a \cdot k = 1 \pmod{m} .$$

Such a is called a multiplicative inverse of k modulo m.

Relative Primality and Multiplicative Inverses, cont.

Suppose k, m are relatively prime, and $k < m$.
The there is a positive integer, a, $a < m$, such that

$$a \cdot k = 1 \pmod{m} .$$

Such multiplicative inverse, a, can be found efficiently, using an extended version of Euclid's algorithm (details not elaborated upon in class).

$$10 \cdot 28 = 1 \pmod{31}$$

$$3 \cdot 12 = 1 \pmod{35}$$

$$218 \cdot 527 = 1 \pmod{621}$$

$$817 \cdot 937 = 1 \pmod{1002}$$

Extended GCD (for reference only)

Claim: if $\gcd(x, y) = g$, then there are two integers a, b such that $ax + by = g$. For example,
$\gcd(1001, 973) = 7$, and indeed $35 \cdot 1001 - 36 \cdot 973 = 7$,
$\gcd(100567, 97328) = 79$. Indeed $601 \cdot 100567 - 621 \cdot 97328 = 79$.
$\gcd(10^7, 10^6 + 1) = 1$, and indeed $10^5 \cdot 10^7 - 999999 \cdot (10^6 + 1) = 1$.

A simple modification of Euclid's gcd algorithm enables to compute these coefficients a, b efficiently. This algorithm is termed extended Euclidian gcd.

If p is a prime and $1 \leq x \leq p - 1$ (this is also denoted $x \in Z_p^*$), then $\gcd(x, p) = 1$ (why?).

Therefore there are integers a, b such that $ax + bp = 1$. In particular, we have $ax = 1 \mod p$. Therefore a is the multiplicative inverse of x modulo p. (This establishes that Z_p^* with multiplication modulo p is a group, and has consequences in several cryptographic applications.)

And Now to Something Completely Different: A Mathematical Gedankenexperiment

The sun was shining on the sea,
Shining with all his might:
He did his very best to make
The billows smooth and bright –
And this was odd, because it was
The middle of the night.

The moon was shining sulkily,
Because she thought the sun
Had got no business to be there
After the day was done–
"It's very rude of him," she said,
"To come and spoil the fun!"

Through the Looking-Glass and What Alice Found There:
Lewis Carroll, 1871.

Mathematical Gedankenexperiment

Choose two positive integers, x, y, independently at random. What is the probability that they are relatively prime, *i.e.* $\gcd(x, y) = 1$?
We note that uniform choice over the positive integers is not possible (think why!). But we could choose x, y uniformly over a large range [1,N], and then send $N \longrightarrow \infty$.

Let us perform a small Gedankenexperiment. The probability that 2 divides x is $1/2$, and so is the probability that 2 divides y. By independence, 2 divides both x and y with probability $1/4$.
Likewise, the probability that 3 divides both x and y is $1/9$, the probability that 5 divides both x and y is $1/25$, etc.

Mathematical Gedankenexperiment, cont.

Define the event $E_p = \{(x, y) \mid$ the prime p divides both x and $y\}$.

We argued that $Pr(E_p) = 1/p^2$, so $Pr(\overline{E_p}) = 1 - 1/p^2$.

$\gcd(x, y){=}1$ if and only if $(x, y) \in \overline{E_2} \cap \overline{E_3} \cap \overline{E_5} \cap \ldots = \bigcap_{p \text{ prime}} \overline{E_p}$.

With some leap of faith, all these events are mutually independent.

$$\begin{aligned} Pr(\gcd(x, y){=}1) &= \prod_{p \text{ prime}} \left(1 - 1/p^2\right) \\ &= \left(\sum_{n=1}^{\infty} 1 + 1/n^2\right)^{-1} \\ &= 6/\pi^2 \approx 0.6079271 \end{aligned}$$

(the last two equalities are from standard calculus class).

Experimental Math: Design

We can approximate $Pr(\gcd(x,y)=1)$ by random sampling: Generating many random pairs (x,y) (chosen uniformly over a large range), and count how many of them are relatively prime.

If `count` denotes the number of sampled pairs that are relatively prime, and `size` is the total number of pairs that were sampled, then `count/size` approximates $6/\pi^2$.

This implies that `size/(6*count)` approximates π^2, so `(size/(6*count))**0.5` approximates π. Our code will display both `count/size` and `(size/(6*count))**0.5` .

Note that our task here is much easier than that of the US/Israel/wherever election pollsters: Our integers always cooperate, they are not effected by changing demographies, and most importantly, they never lie :-).

Experimental Math: Code

Our code will display both `count/size` and `(size/(6*count))**0.5` .

```
import random

def samplegcd(n,size):
    """ repeats the following "size" times: Choose two
    random integers, n bits long each (between 1 to 2**n-1).
    Checks if they are relatively prime.
    Computes the frequency and the derived approximation to pi."""

    count=0
    for i in range(0,size):
        if gcd(random.randint(1,2**n-1),random.randint(1,2**n-1))==1:
            count += 1    # the dreaded +=
    return count/size, (6*size/count)**0.5
```

Experimental Math: Execution

We can approximate $Pr(\gcd(x,y)=1)$ by generating many random pairs (x,y) (over a large range), and count how many of them are relatively prime, and what approximation to π we get.

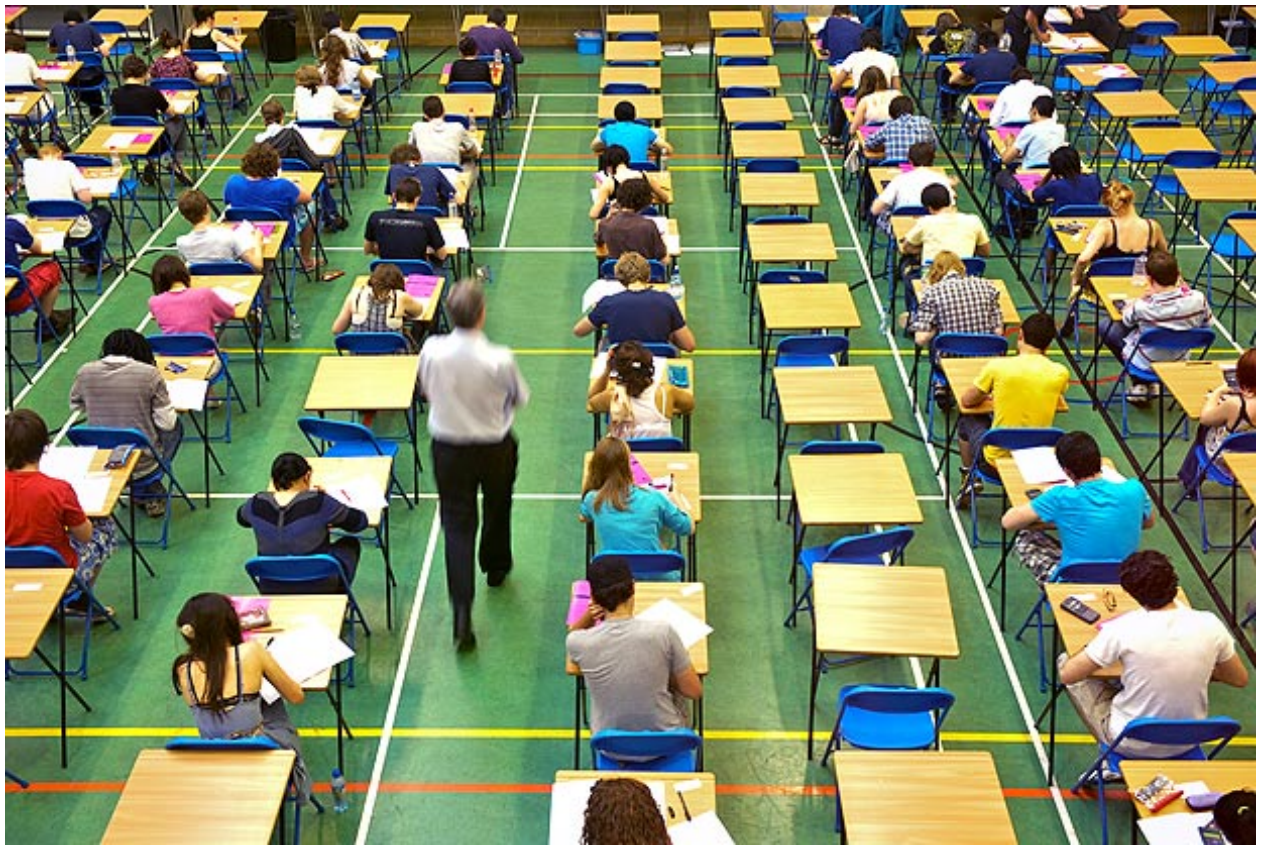
```
>>> for i in range(1,9): samplegcd(50,2**(15+i))

(0.6092529296875, 3.138172498115786)
(0.608650207519531 2, 3.139725917537844 2)
(0.607215881347656 2, 3.143431959265063)
(0.607948303222656 2, 3.141537873730807 7)
(0.608119964599609 4, 3.141094442490883 8)
(0.607780456542968 8, 3.141971632578730 6)
(0.608010292053222 7, 3.141377724120358 6)
(0.60796880722045 9, 3.14148489845641 7)
```

Comments:

- Returned values converge to the "true values", 0.6079271 and 3.1415926, but not monotonically.
- These executions are not instantaneous. For example, it took almost two minutes for `size`=$2^{23} = 8,388,608$ samples).

Intentionally Left Blank

Group Theory Background, and Proof of Fermat's Little Theorem (for reference only – not for exam)



(photo taken from the Sun)

Group Theory Background

The next slides describe some (rather elementary) background from group theory, which is needed to prove Fermat's little theorem.

For lack of time, nor did we cover this material in class, neither shall we cover it in the future.

If you are ready to believe Fermat's little theorem without seeing its proof, you can skip the next slides. (Don't worry, be happy: we will not examine you on this material :=)

If you wish to learn a bit about groups (a beautiful mathematical topic, which also plays fundamental roles in physics), you are welcome to keep reading. Hopefully, this material will be covered in more depth in some future class you'll take.

A (Relevant) Algebraic Diversion: Groups

A group is a nonempty set, G, together with a "multiplication operation", $*$, satisfying the following "group axioms":

- Closure: For all $a, b \in G$, the result of the operation is also in the group, $a * b \in G$ ($\forall a \forall b \exists c \, a * b = c$).
- Associativity: For all $a, b, c \in G$, $(a * b) * c = a * (b * c)$ ($\forall a \forall b \forall c \, (a * b) * c = a * (b * c)$).
- Identity element: There exists an element $e \in G$, such that for every element $a \in G$, the equation $e * a = a * e = a$ holds ($\exists e \forall a \, a * e = e * a = a$). This identity element of the group G is often denoted by the symbol 1.
- Inverse element: For each a in G, there exists an element b in G such that $a * b = b * a = 1$ ($\forall a \exists b \, a * b = b * a = e$).

If, in addition, G satisfies

- Commutativity: For all $a, b \in G$, $a * b = b * a$ ($\forall a \forall b \, a * b = b * a$).

then G is called a multiplicative (or Abelian) group.

A Few Examples of Groups

Non Commutative groups:

- $GL_n(\mathbb{R})$, the set of n-by-n invertible (non singular) matrices over the reals, $\mathbb{R}$, with the matrix multiplication operation .
- n-by-n integer matrices having determinant ± 1, with the matrix multiplication operation (unimodular matrices).
- The collection S_n of all permutations on $\{1, 2, \ldots, n\}$, with the function composition operation.

Commutative groups:

- The integers, $\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$, with the addition operation.
- For any integer, $m \geq 1$, the set $Z_m = \{0, 1, 2, \ldots\ m-1\}$, with the addition modulo m operation.
- For any prime, $p \geq 2$, the set $Z_p^* = \{1, 2, \ldots\ p-1\}$, with the multiplication modulo p operation. If p is composite, this Z_p^* is not a group (check!).

Sub-groups and Lagrange Theorem

- Let $(G, *)$ be a group. $(H, *)$ is called a sub-group of $(G, *)$ if it is a group, and $H \subset G$.
- Claim: Let $(G, *)$ be a finite group, and $H \subset G$. If H is closed under $*$, then $(H, *)$ is a sub-group of $(G, *)$.
- Question: What happens in the infinite case?

- Lagrange Theorem: If $(G, *)$ is a finite group and $(H, *)$ is a sub-group of it, then $|H|$ divides $|G|$.

Lagrange Theorem and Cyclic Subgroups

- Let a^n denote $a * a * \ldots * a$ (n times).
- We say that a is of order n if $a^n = 1$, but for every $m < n$, $a^m \neq 1$.

- Claim: Let G be a group, and a an element of order n. The set $\langle a \rangle = \{1, a, \ldots, a^{n-1}\}$ is a subgroup of G.
- Let G be a group with k elements, a an element of order n.
- Since $\langle a \rangle = \{1, a, \ldots, a^{n-1}\}$ is a subgroup of G, Lagrange theorem implies that $n \mid k$.
- This means that there is some positive integer ℓ such that $n\ell = k$.
- Thus $a^k = a^{n\ell} = (a^n)^\ell = 1^\ell = 1$.

Proof of Fermat's Little Theorem

- We just saw that Lagrange theorem, for every $a \in G$, the order of any element $a \in G$ divides $|G|$.
- And thus raising any element $a \in G$ to the power $|G|$ yields 1 (the unit element of the group).
- For any prime p, the order of the multiplicative group $a \in Z_p^* = \{1, \ldots, p-1\}$ is $p-1$.
- We thus get Fermat's "little" theorem: Let p be a prime. For every $a \in Z_p^* = \{1, \ldots, p-1\}$, $a^{p-1} \mod p = 1$.

Fermat's Little Theorem: Second Proof (more direct)

Let $p \geq 2$ be a prime. For every $a \in Z_p^*$, the mapping $x \mapsto ax \mod p$ is a one-to-one mapping of Z_p^* onto itself (this follows from the fact that Z_p^* is a group with respect top multiplication modulo p, thus every such $a \in Z_p^*$ has a multiplicative inverse).

This implies that $\{a \cdot 1, a \cdot 2, \ldots a \cdot (p-1)\}$ is a rearrangement of $\{1, 2, \ldots p-1\}$. Multiplying all elements in both sets, we get $a^{p-1} \cdot 1 \cdot 2 \cdot \ldots (p-1) = 1 \cdot 2 \cdot \ldots (p-1) \mod p$, implying $a^{p-1} = 1 \mod p$. ♠