Extended Introduction to Computer Science
CS1001.py

Lecture 14: Number Theoretic Algorithms: Factoring Integers, primality testing; Diffie Hellman Public Exchange of Secret Key

Instructors: Daniel Deutch, Amir Rubinstein
Teaching Assistants: Michal Kleinbort, Amir Gilad

Founding Instructor: Benny Chor

School of Computer Science
Tel-Aviv University, Fall Semester, 2017-8
http://tau-cs1001-py.wikidot.com

Lecture 10-13

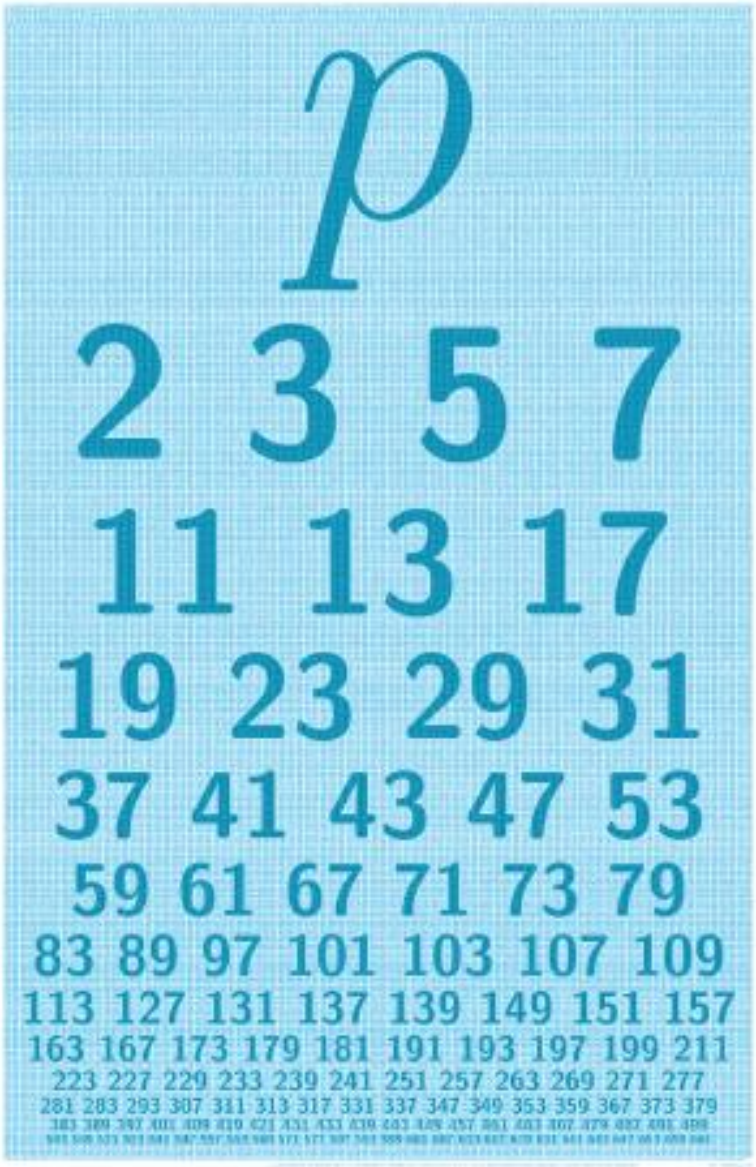
- Recursion, recurson, recursion...

Lecture 14: Plan

Number Theory algorithms

- Prime Numbers:
  - Trial division
  - Fermat's "little theorem", and randomized primality testing.
- Cryptography:
  - The discrete logarithm problem as a One-way function.
  - Diffie-Hellman scheme for secret key exchange over insecure communication lines.
- Maybe later in the course: Greatest Common Divisor

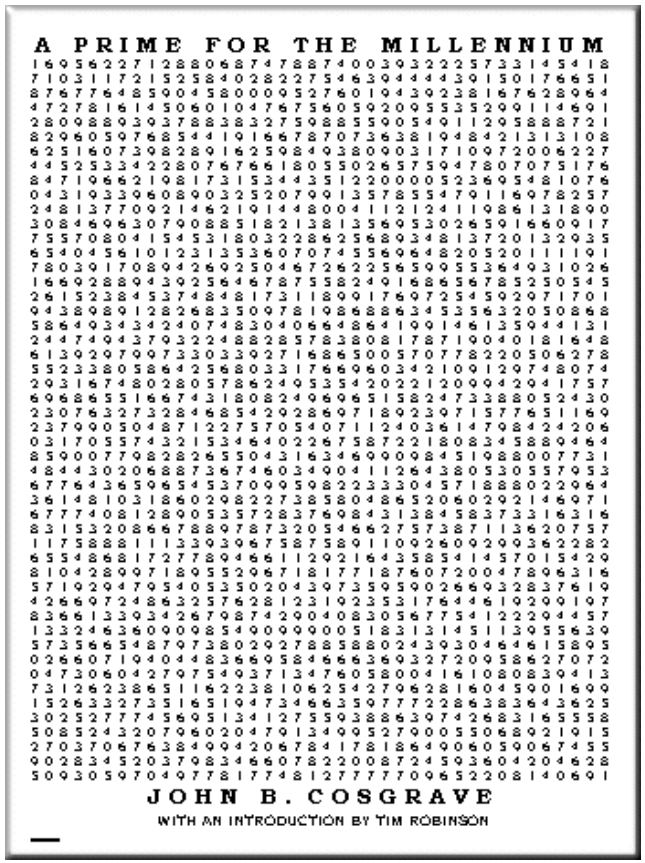
Prime Numbers and Randomized Primality Testing



(figure taken from unihedron site)

Prime Numbers and Randomized Primality Testing

A prime number is a positive integer, divisible only by 1 and by itself.
So $10,001 = 73 \cdot 137$ is not a prime (it is a composite number), but $10,007$ is.
There are some fairly large primes out there.


A PRIME FOR THE MILLENNIUM

JOHN B. COSGRAVE

WITH AN INTRODUCTION BY TIM ROBINSON

Published in 2000: A prime number with 2000 digits (40-by-50 table). By John Cosgrave, Math Dept, St. Patrick's College, Dublin, Ireland.

http://www.iol.ie/∼tandmfl/mprime.htm

Prime Numbers in the News: $p = 2^{57885161} - 1$

מספר חזק

17 מיליון ספרות: נחשף המספר הראשוני הכי גדול

2 בחזקת 57,885,161 פחות 1 - זה המספר הכי גדול שמתחלק רק בעצמו וב-1 שהתגלה עד כה. "זה כמו לטפס על האוורסט", אומר מדען על ההישג, שעל עולם המתמטיקה לא ממש ישפיע

Recommend 635

ynet פורסם: 08:15 , 07.02.13

2, 3, 5, 7, 11, 13, 17, 19... ו-2 בחזקת 57,885,161 פחות 1. השבוע חשפו מתמטיקאים אמריקנים את המספר הראשוני הגדול ביותר שהתגלה עד כה, והוא מורכב מלא פחות מ-17 מיליון ספרות. לחצו כאן כדי לראות את הגרסה המקוצרת אבל הארוכה להפליא של המספר העצום הזה.

עוד בערוץ החדשות

- מטוס-מסוק: צה"ל מבקש את ה-V22 "אוספרי"
- דרעי על שרה נתניהו: העיסוק בהופעתה אינו ראוי

המספר, שכמו כל מספר ראשוני מתחלק ללא שארית רק בעצמו וב-1, התגלה על ידי ד"ר כריס קופר מאוניברסיטת סנטרל מיזורי. יש לו 4 מיליון ספרות יותר מלשיאן הקודם, שנחשף ב-2008. למען הדיוק, במספר הראשוני החדש - 2 בחזקת 57,885,161 פחות 1 - יש למעשה 17,435,170 ספרות.

הבונוס על החשיפה: 3,000 דולר

בעיתון הבריטי "טלגרף" נכתב כי שני המספרים הראשוניים הענקיים הללו התגלו בעזרת רשת מורכבת של מחשבים שמכונה GIMPS. רשת זו משלבת 360 אלף מעבדים כדי לזהות את המספרים הראשוניים, והיא יכולה לבצע 150 טריליון חישובים לשנייה.

לתוצאה חשיבות קטנה מאוד בעולם המתמטיקה, אבל היא מהווה אות כבוד לחוקרים שמתחרים על מציאת מספרים ראשוניים גדולים ככל האפשר.

2 חסר כאן וגם 2 בחזקת 57,885,161 פחות 1. מספרים ראשוניים

שתף בפייסבוק
הדפסה
שלח כתבה
הרשמה לדיוור
תגובה לכתבה
עשו מנוי לעיתון

(screenshot from ynet, February 2013)

The Prime Number Theorem

- The fact that there are infinitely many primes was proved already by Euclid, in his Elements (Book IX, Proposition 20).
- The proof is by contradiction: Suppose there are finitely many primes $p_1, p_2, \ldots, p_k$. Then $p_1 \cdot p_2 \cdot \ldots \cdot p_k + 1$ cannot be divisible by any of the p_i, so its prime factors are none of the p_is. (Note that $p_1 \cdot p_2 \cdot \ldots \cdot p_k + 1$ need not be a prime itself, *e.g.* $2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdot 13 + 1 = 30,031 = 59 \cdot 509$.)
- Once we know there are infinitely many primes, we may wonder how many are there up to a given integer N of n bits,

- The prime number theorem: A random n bit number is a prime with probability $O(1/n)$.
- Informally, this means there are heaps of primes of any size, and it is quite easy to hit one by just picking at random.

Modern Uses of Prime Numbers

- Primes (typically small primes) are used in many algebraic error correction codes (improving reliability of communication, storage, memory devices, etc.).

- Primes (always huge primes) serve as a basis for many public key cryptosystems (serving to improve confidentiality of communication).

Trial Division

Suppose we are given a large number, N, and we wish to find if it is a prime or not.

If N is composite, then we can write $N = KL$ where $1 < K, L < N$. This means that at least one of the two factors is $\leq \sqrt{N}$.

This observation leads to the following trial division algorithm for factoring N (or declaring it is a prime):

Go over all D in the range $2 \leq D \leq \sqrt{N}$. For each such D, check if it evenly divides N. If there is such divisor, N is a composite. If there is none, N is a prime.

Trial Division

```
def trial_division(N):
    """ Check if integer N is prime """
    upper = round(N**0.5 + 0.5) # sqrt(N) rounded up
    for m in range(2,upper+1):
        if N%m == 0:    # m divides N
            print(m, "is the smallest divisor of", N)
            return False #N is composite
    # we get here if no divisor was found
    print(N, "is prime")
    return True
```

Trial Division: A Few Executions

Let us now run this on a few case (only printouts are shown):

```
>>> trial_division(2**40+15)
1099511627791 is prime
>>> trial_division(2**40+19)
5 is the smallest divisor of 1099511627795
>>> trial_division(2**50+55)
1125899906842679 is prime
>>> trial_division(2**50+69)
123661 is the smallest divisor of 1125899906842693
>>> trial_division(2**55+9)
5737 is the smallest divisor of 36028797018963977
>>> trial_division(2**55+11)
36028797018963979 is prime
```

Seems very good, right?

Seems very good? Think again!

Trial Division Performance: Unary vs. Binary Thinking

This algorithm takes up to $\sqrt{N}$ divisions in the worst case (it actually may take more operations, as dividing long integers takes more than a single step). Should we consider it efficient or inefficient?

Recall – efficiency (or lack thereof) is measured as a function of the input length. Suppose N is n bits long. This means $2^{n-1} \leq N < 2^n$.

What is $\sqrt{N}$ in terms of n?
Since $2^{n-1} \leq N < 2^n$, we have $2^{(n-1)/2} \leq \sqrt{N} < 2^{n/2}$.
So the number of operations performed by this trial division algorithm is exponential in the input size, n. You would not like to run it for $N = 2^{321} + 17$ (a perfectly reasonable number in crypto contexts).

So why did many of you say this algorithm is efficient? Because, consciously or subconsciously, you were thinking in unary.

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.

Trial Division Performance: Actual Measurements

Let us now measure actual performance on a few cases.

```
>>> elapsed("trial_division (2**40+19)")
5 is the smallest divisor of 1099511627795
0.002822999999999909
>>> elapsed("trial_division (2**40+15)")
1099511627791 is prime
0.16658700000000004
>>> elapsed("trial_division (2**50+69)")
123661 is the smallest divisor of 1125899906842693
0.022221999999999964
>>> elapsed("trial_division (2**50+55)")
1125899906842679 is prime
5.829111
>>> elapsed("trial_division (2**55+9)")
5737 is the smallest divisor of 36028797018963977
0.003503999999999075
>>> elapsed("trial_division (2**55+11)")
36028797018963979 is prime
29.706794
```

Trial Division Performance: Food for Thought

Question: What are the best case and worst case inputs for the `trial_division` function, from the execution time (performance) point of view?

Factoring by Trial Division - Summary

- We wanted to efficiently test if a given n bits integer N, ($2^{n-1} \leq N < 2^n$), is prime/composite.
- Trial division factors an n bit number in time $O(2^{n/2})$. The best algorithm to date, the general number field sieve algorithm, does so in $O(e^{8n^{1/3}})$. (In 2010, RSA-768, a "hard" 768 bit, or 232 decimal digits, composite, was factored using this algorithm and heaps of concurrent hardware.)

- The search problem, "given N , find all its factors" is believed to be intractable.

- Does this imply that the decision problem, "determine if N is prime", is also (believed to be) intractable?
  or is there a better way to check primality than by factoring N?

Beyond Trial Division

So how should we proceed with checking if a given integer is prime or not? Two possible directions:

- Find an alternative integer factoring algorithm, that is efficient.
  - This is a major open problem. We will not try to solve it.

- Find an efficient primality testing algorithm.
  - This is the road we will take.

Randomized Primality (Actually Compositeness) Testing

Basic Idea [Solovay-Strassen, 1977]: To show that N is composite, enough to find evidence that N does not behave like a prime.
Such evidence need not include any prime factor of N.

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, N, and for some a in in the range $2 \leq a \leq N-1$, we find that $a^{N-1} \neq 1 \pmod{N}$.

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

Fermat Test: Example

Let us show that the following 164 digits integer, N, is composite.
We will use Fermat test, employing the good old `pow` function.

```
>>> N = 57586096570152913699974892898380567793532123114264532903689671329
43152103259505773547621272182134183706006357515644099320875282421708540
995974523600877839218983091
>>> a = 65
>>> pow(a ,N-1, N)
28361384576084316965644957136741933367754516545598710311795971496746369
83813383438165679144073738154035607602371547067233363944692503612270610
976637261645893300588 2    # does not look like 1 to me
```

This proof gives no clue on N's factorization (but I just happened to bring the factorization along with me, tightly placed in my backpack: $N = (2^{271} + 855)(2^{273} + 5)$).

Randomized Primality Testing

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

It was shown by Miller and Rabin that if N is composite, then at least 3/4 of all $a \in \{1, \ldots, N-1\}$ are witnesses.

Randomized Primality Testing (2)

It was shown by Miller and Rabin that if N is composite, then at least 3/4 of all $a \in \{1, \ldots, N-1\}$ are witnesses.

If N is prime, the by Fermat's little theorem, no $a \in \{1, \ldots, N-1\}$ is a witness.

Picking $a \in \{1, \ldots, N-1\}$ at random yields an algorithm that gives the right answer if N is composite with probability at least 3/4, and always gives the right answer if N is prime.

However, this means that if N is composite, the algorithm could err with probability as high as 1/4.

How can we guarantee a smaller error?

Randomized Primality Testing (3)

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

Remark: This idea, which we term 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 N (why??).
- If N is prime, the algorithm always outputs "N is prime".

- If N is composite, the algorithm may err and outputs "N is prime".
- Miller-Rabin showed that if N is composite, then at least 3/4 of all $a \in \{1, \ldots, N-1\}$ are witnesses.
- To err, all random choices of a's should yield non-witnesses. Therefore,

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

Properties of Fermat Primality Testing, cont.

- 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 back in November 2011), an earthquake of Richter magnitude 7.3 will hit Tel-Aviv in the next 24 hours, etc., etc.

Primality Testing: Simple Python Code

```
import random  # random numbers package

def is_prime(N, show_witness=False):
    """ probabilistic test for N's compositeness """
    for i in range(0,100):
        a = random.randint(1,N-1) # a is a random integer in [1..N-
        if pow(a,N-1,N) != 1:
            if show_witness:  # caller wishes to see a witness
                print(n,"is composite","\n",a,"is a witness, i=",i+
            return False
    return True
```

Let us now run this on some fairly large numbers:

```
>>> is_prime(3**100+126)
False
>>> is_prime(5**100+126)
True
>>> is_prime(7**80-180)
True
>>> is_prime(7**80-18)
False
>>> is_prime(7**80+106)
True
```

How to find an n-bit long prime number

```
def find_prime(n):
    """ find random n-bit long prime """
    while(True):
        candidate = random.randrange(2**(n-1),2**n)
        if is_prime(candidate):
            return candidate
```

while(True)??!

Can we be 100% sure we will not loop forever?
Can we be (100-ϵ)% sure?
What is the expected number of trials until we get a prime?

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.

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

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

Efficient Modular Exponentiation (reminder)

Goal: Compute $a^b \bmod c$, where $a, b, c \geq 2$ are all n bit integers. In Python, this can be expressed as `(a**b) % c`.
We should still be a bit careful. Computing a^b first, and then taking the remainder $\bmod c$, is not going to help at all.
Instead, we compute all the successive squares $\bmod c$, namely $\left\{a^1 \bmod c, a^2 \bmod c, a^4 \bmod c, a^8 \bmod c, \ldots, a^{2^{n-1}} \bmod c\right\}$.
Then we multiply the powers corresponding to in locations where $b_i = 1$. Following every multiplication, we compute the remainder. This way, intermediate results never exceed c^2, eliminating the problem of huge numbers.

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.

Efficient Modular Exponentiation: Complexity Analysis

Goal: Compute $a^b \bmod c$, where $a, b, c \geq 2$ are all n bit integers. Using iterated squaring, this takes between $n-1$ and $2n-1$ multiplications.

Intermediate multiplicands never exceed c, so computing the product (using the method perfected in elementary school) takes $O(n^2)$ bit operations.

Each product is smaller than c^2, which has at most $2n$ bits, and computing the remainder of such product modulo c takes another $O(n^2)$ bit operations (using long division, also studied in elementary school, but we did not see it in this course).

All by all, computing $a^b \bmod c$, where $a, b, c \geq 2$ are all n bit integers, takes $O(n^3)$ bit operations.

Modular Exponentiation in Python (reminder)

We can easily modify our function, power, to handle modular exponentiation.

```
def modpower(a, b, c):
    """ computes a**b modulo c, using iterated squaring """
    result = 1
    while b>0:       # while b is nonzero
        if b%2 == 1:       # b is odd
            result = (result * a) % c
        a = (a**2) % c
        b = b//2
    return result
```

A few test cases:

```
>>> modpower(2,10,100)    # sanity check:  2^10 = 1024
24
>>> modpower(17, 2*100+3**50, 5**100+2)
535179367519434237142526199613451017810131381775103207690859233912593 3
>>> 5**100+2    # the modulus, in case you are curious
78886090522101180541172856528278622967320643510902300477027893066406 27
>>> modpower(17, 2**1000+3**500, 5**100+2)
11198874511251598021191388421459035679739562823569349572111064482646 30
```

Built In Modular Exponentiation – pow(a,b,c)

Guido van Rossum has not waited for our code, and Python has a built in function, pow(a,b,c), for efficiently computing $a^b \bmod c$.

```
>>> modpower(17,2**1000+3**500,5**100+2)\  # line continuation
                          -pow(17,2**1000+3**500,5**100+2)
0
# Comforting: modpower code and Python pow agree. Phew...

>>> elapsed("modpower(17,2**1000+3**500,5**100+2)")
0.0026359999999542
>>> elapsed("modpower(17,2**1000+3**500,5**100+2)",number=1000)
2.280894000000046
>>> elapsed("pow(17,2**1000+3**500,5**100+2)",number=1000)
0.745319999999924
```

So our code is just three times slower than pow.

Does Modular Exponentiation Have Any Uses?

Applications using modular exponentiation directly (partial list):

- Randomized primality testing - just saw this.
- Diffie Hellman Key Exchange - coming up next.
- Rivest-Shamir-Adelman (RSA) public key cryptosystem (PKC) - in an elective crypto course.

Trust, But Check!

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

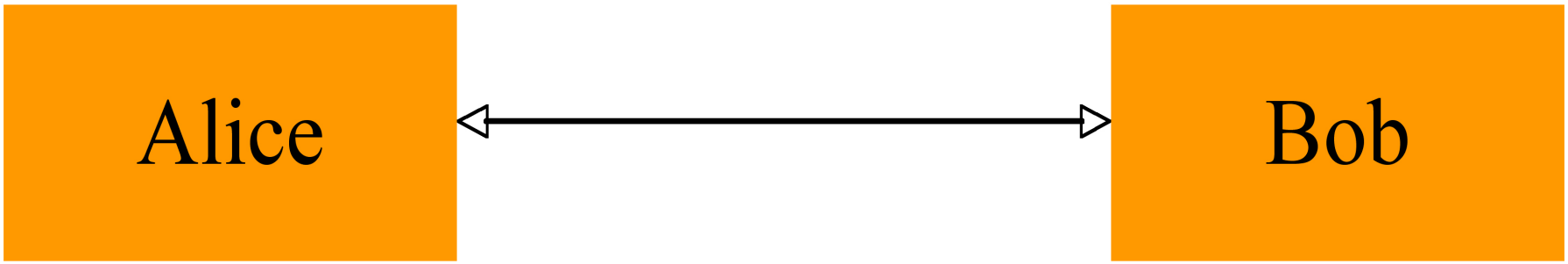
In 1637, the French mathematician Pierre de Fermat, wrote some comments in the margin of a copy of Diophantus' book, Arithmetica. Fermat claimed he had a wonderful proof that no such solution exists, but the proof is too large to fit in the margin.

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

And Now For Something Completely Different: Encryption

Encryption: Basic Model

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

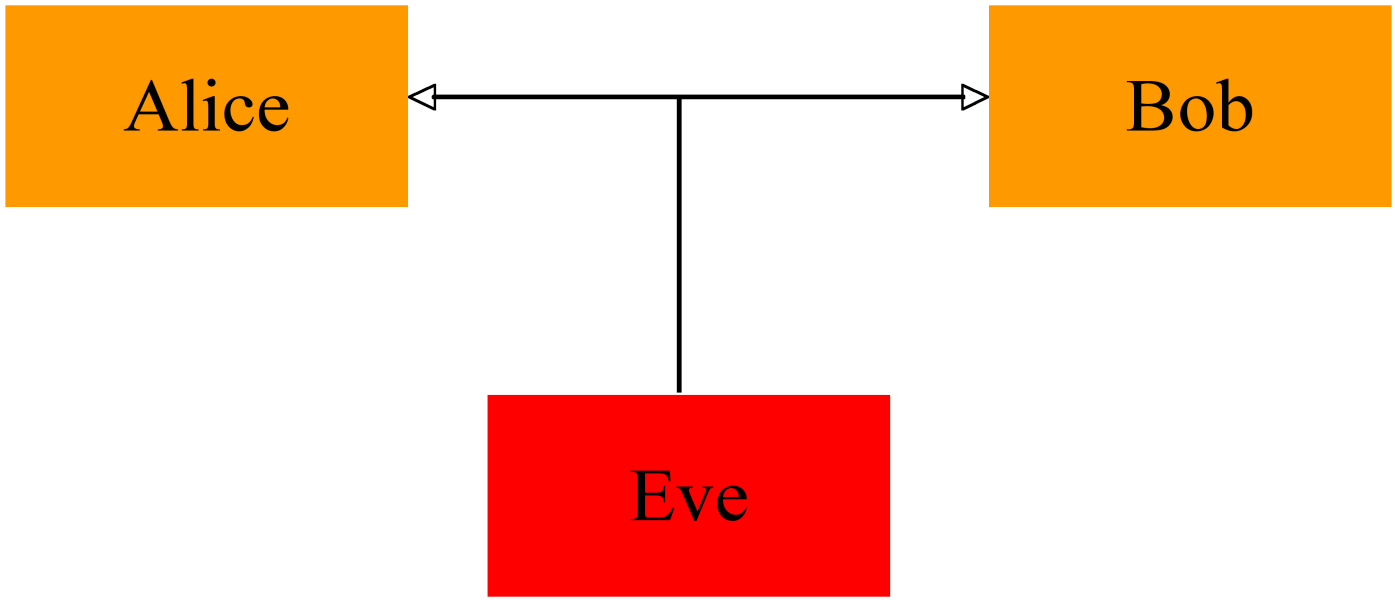


1. Two parties – Alice and Bob.
2. Reliable communication line.
3. Encryption algorithm, E.
4. Decryption algorithm, D.
5. Shared, secret key: $k_{A,B}$ (used both for encryption and decryption).

6. Goal: send a message M confidentially.

Adversarial Model: Passive Eavesdropper

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



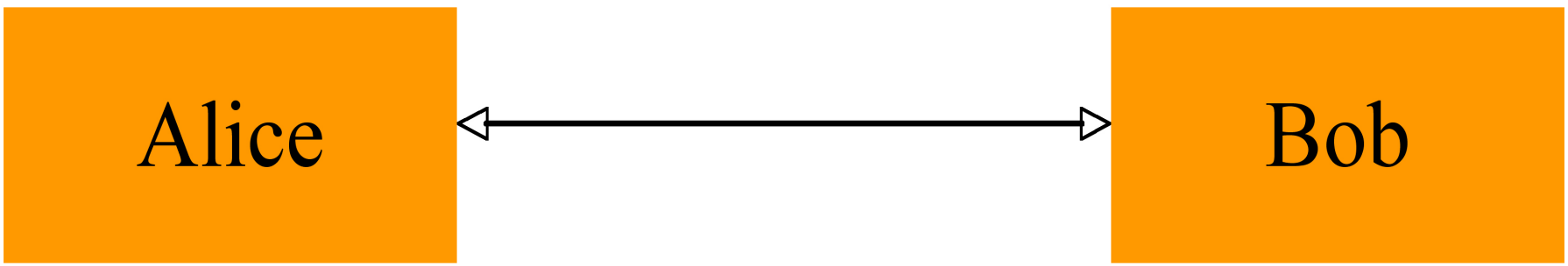
- Eve attempts to discover information about M
- Eve knows the algorithms E, D
- Eve knows the message space
- Eve has intercepted $E_{k_{A,B}}(M)$
- Eve does not know $k_{A,B}$

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 secret Keys

Diffie and Hellman also proposed public exchange of secret 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).

Discrete Log modulo p: a One Way Function

- 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

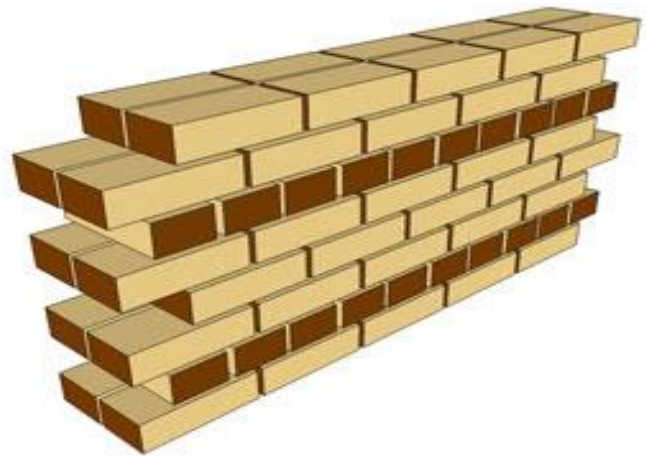
- Public parameters: A large prime p (1024 bit long, say) and a random element g in in the range $1 < g < p - 1$.
- Alice chooses at random an integer a from the interval $[2..p-2]$. She sends $x = g^a \pmod p$ to Bob (over the insecure channel).
- Bob chooses at random an integer b from the interval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

```
def DH_exchange():
    """ generates a shared DH key """
    n = int(input("How many bits for the prime number? "))
    p = find_prime(n)
    print("p =",p, "a large prime")
    g = random.randint(2,p-1)
    print("g =",g, "random 1<g<p")
    a = random.randint(2,p-1)# Alice's  secret
    print("a = ? random secret of Alice")
    b = random.randint(2,p-1)# Bob's  secret
    print("b = ? random secret of Bob")
    x = pow(g,a,p) #Alice's transmission
    print("x =",x, "Alice sends to Bob x = g**a%p")
    y = pow(g,b,p) #Bob's transmission
    print("y =",y, "Bob sends to Alice y = g**b%p")
    key_A = pow(y,a,p) #shared key on Alice's side
    print("key_A =", key_A, "shared key on Alice's side y**a%p")
    key_B = pow(x,b,p) #shared key on Bob's side
    print("key_B =", key_B, "shared key on Bob's side x**b%p")
    if key_A != key_B:
        print("This can't happen!", key_A, "!=", key_B)
```

Diffie and Hellman Key Exchange: Execusions

```
>>> DH_exchange()
How many bits for the prime number? 3
p = 7 a large prime
g = 5 random 1<g<p
a = ? random secret of Alice
b = ? random secret of Bob
x = 3 Alice sends to Bob x = g**a%p
y = 3 Bob sends to Alice y = g**b%p
key_A = 5 shared key on Alice side y**a%p
key_B = 5 shared key on Bob side x**b%p
```

Diffie and Hellman Key Exchange: Execusions

```
>>> DH_exchange()
How many bits for the prime number? 512
p = 13190499215984842627469418768459072964940489777650242166499136
67572942585710004529705428629627096002415235510640860010092771785
63855377781646396343073541017 a large prime
g = 72727808700893374290571123633722616171303168201540293030506830
53774078939715449762130396601304583206497202013765507327982133966
88684688812246197975068070 4 random 1<g<p
a = ? random secret of Alice
b = ? random secret of Bob
x = 77859685459587905325192279504421785829517027605729957580187515
57023021558388381459845872273783896810637205663727718839962373297
76833427763130870893457111 7 Alice sends to Bob x = g**a%p
y = 81592503968207908209903492437727139350000062871146635960125868
91311949910003854853265340938404654720688124342952020483636286841
18577116264783369719255121 4 Bob sends to Alice y = g**b%p
key_A = 37885996270995313862289346616118085884932181277516223437861
91138720499800909030723432904837832705963381108701807210838553516
82882532754540985222390834162 5 shared key on Alice side y**a%p
key_B = 37885996270995313862289346616118085884932181277516223437861
91138720499800909030723432904837832705963381108701807210838553516
82882532754540985222390834162 5 shared key on Bob side x**b%p
```

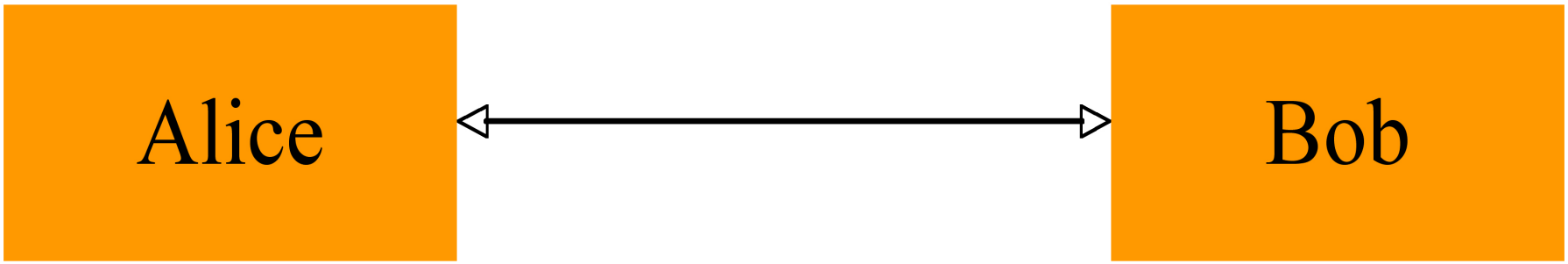
Diffie and Hellman – Final Remarks

- Recall that the length of the prime p in bits is $n = \lfloor \log_2 p \rfloor + 1$.
- Computation time for exchanging the key is $O(n^3) = O(\log_2^3 p)$ 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).

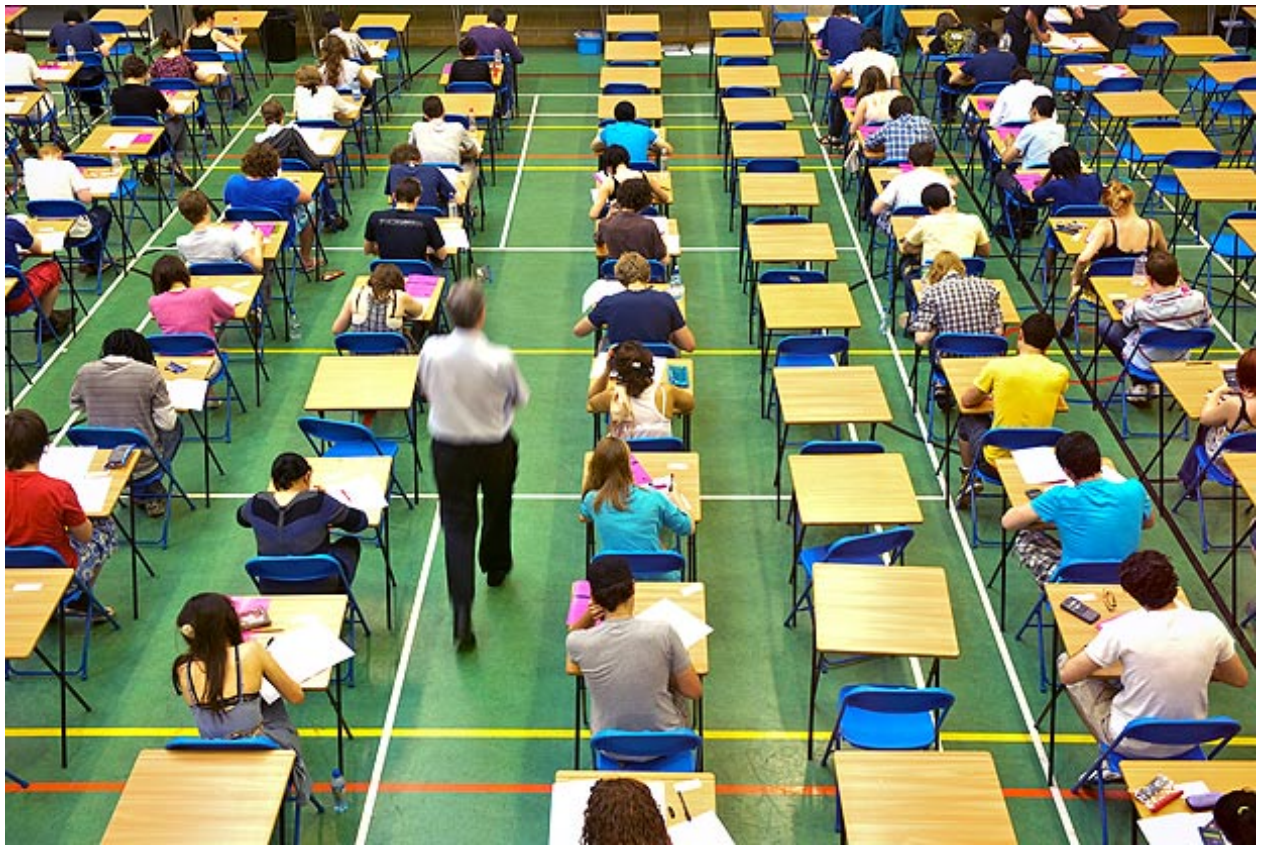
6. Comment: To learn how $k_{A,B}$ is employed in a classical, secret key crypto system, we refer you to the elective crypto course.
7. We did not explain or exemplify how classical crypto works.

Diffie and Hellman – Color Mixing analogy

https://www.youtube.com/watch?v=YEBfamv-_do (start at 2:25)

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 $a^{p-1} = 1 \mod p$. ♠