# Extended Introduction to Computer Science CS1001.py 

# Chapter F Topics in Number Theory: <br> Lecture 13 <br> Factoring and Primality Testing 

Michal Kleinbort, Amir Rubinstein

School of Computer Science
Tel-Aviv University
Fall Semester 2023-24
http://tau-cs1001-py.wikidot.com

* Slides based on a course designed by Prof. Benny Chor


## Topics in Number Theory: Plan

1. Exponentiation of integers - last lecture
2. Primality testing (using Fermat's "little theorem")
3. Diffie-Helman secret key exchange
4. Euclid's GCD (greatest common divisor)

## Prime Numbers

- A prime number is a positive integer, divisible only by 1 and by itself. Other numbers are called composite.
- So $10,001=73 \cdot 137$ is not a prime (it is a composite number), but 10,007 is prime.


## Large Prime Numbers

- There are some fairly large primes out there.

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


- The largest known prime number (as of December 2023, found in 2018 by Patrick Laroche) is $2^{82,589,933}-1$. It has more than 24 million digits when written in base 10.
- https://en.wikipedia.org/wiki/Largest known prime number


## Infinitely Many Primes

- 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 $t=p_{1} \cdot p_{2} \cdot \ldots \cdot p_{k}+1$ cannot be divisible by any of the $p_{i}$ 's, so none of them are its prime factors. So, either $t$ is prime or has other prime factors.
(Note that $t$ need not be a prime itself,

$$
\text { e.g. } 2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdot 13+1=30,031=59 \cdot 509)
$$

## Density of Prime Numbers

- The Prime Number Theorem (19 ${ }^{\text {th }}$ century):

A random $n$ bit number is a prime with probability roughly $O\left(\frac{1}{n}\right)$.

- So, the probability to be prime decreases with $n$, but the number of $n$ bit primes increases with $n$ (recall that an $n$-bit number is between $2^{n-1}$ and $2^{n}-1$, so there are $2^{n-1}$ numbers in the range, and therefore $O\left(\frac{1}{n} \cdot 2^{n-1}\right)$ prime numbers with $n$ bits.
- Informally, this means there are plenty of primes of any size, and it is quite easy to hit one by just picking at random.


## Use of Prime Numbers

- Cryptosystems: Primes (always huge primes) are used to improve confidentiality (encryption) and authenticity (digital signatures) of communication.
- Error correction codes: Primes (typically small primes) are used to improve reliability of communication and data storage.
- Data structures and algorithms: Primes are sometimes used to improve complexity, e.g. by decreasing the probability of the worst case.
- Many more...


## Two Common Problems

- Suppose we are given a number $N$.
- Factorization problem: find $N$ 's prime factors This is an example for a search problem
- Primality testing problem: decide if $N$ is prime or not. This is an example for a decision (yes/no) problem
- Which of these two problems sounds presumably easier?
- We will deal here with the latter and get back to factorization in the end of the class.


## Primality Testing by Trial Division

- If $N$ is composite, then we can write $N=K \cdot L$, 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 testing if $N$ is prime or not:


## Trial Division Algorithm

- Go over all $m$ in the range $2 \leq D \leq \sqrt{N}$.
- For each such $m$, 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 in Python

```
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 # \mathbb{N}\mathrm{ is composite}
    # we get here if no divisor was found
    print(N, "is prime")
    return True
```

- Executions in class.
- What are worst and best cases in terms of runtime?
- Seems very good, right?
- Think again.


## Trial Division: Time Complexity

- This algorithm takes up to $\sqrt{N}$ modulo operations in the worst case (actually, it may take more basic operations, as modulo on long integers takes more than a single step).
- Should we consider it efficient or inefficient?
- Try trial_division (2**321 + 17)
- $\sqrt{N}$ seems reasonable, but it is not
- $N$ is an $n$ bit number.
- So, $2^{n-1} \leq N<2^{n}$
- So, $2^{(n-1) / 2} \leq \sqrt{N}<2^{n / 2}$
- In other words, $\sqrt{2}^{n-1} \leq \sqrt{N}<\sqrt{2}^{n}$
- So $\sqrt{N}=\Theta\left(\sqrt{2}^{n}\right)$
- Exponential in the input size


## Comic Relief*



## Primality Testing via Witnesses

- Basic Idea [Solovay-Strassen, 1977]: To show that $N$ is composite, enough to find evidence that $N$ does not behave like a prime.
- We will look for witnesses that will provide such evidence to the compositeness of $N$.
- Note: a prime factor of $N$ is such a witness, but there are other witnesses, as we will now see. These other witnesses for $N$ 's compositeness may not tell us anything about $N$ 's factorization.


## Witnesses for Compositeness: type 1

FACT $_{N}$ is the set of divisors (factors) of $N$

This set includes every integer $a$ such that:

- $1<a<N$
- $N \% a=0($ also written as: $N=0(\bmod a))$

For example: $5 \in \mathrm{FACT}_{10}$

How to find such a witness? Use Trial Division
Worst case complexity $O\left(2^{n / 2}\right)$

## Witnesses for Compositeness: type 2

$\mathrm{GCD}_{N}$ is the set of non-coprimes of $N$
This set includes every integer $a$ such that:

- $1<a<N$
- $\operatorname{gcd}(N, a)>1$, where $\operatorname{gcd}(N, a)$ is the greatest common divisor of $N$ and $a$

For example: $8 \in \mathrm{GCD}_{10}$, since $\operatorname{gcd}(8,10)=2>1$, and 2 is a factor of 10

Note: $\mathrm{FACT}_{N} \subseteq \mathrm{GCD}_{N}$
How to find such a witness? Go over all $a \in[2, \ldots, N-1]$, and look for a value for which $\operatorname{gcd}(N, a)>1$. Efficiently compute the $\operatorname{gcd}$ of two numbers using Euclid's algorithm

Worst case complexity $O\left(N \cdot n^{3}\right)=O\left(2^{n} \cdot n^{3}\right)$

## Fermat's Little Theorem*

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

Then $a^{p-1}=1(\bmod p)$.

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 $a^{N-1} \neq 1(\bmod N)$.

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

## Fermat's Little Theorem: Example

Let us show that the following 164 digits integer, $N$, is composite. We will use Fermat test, employing the good old pow function.
>>> $\mathrm{N}=57586096570152913699974892898380567793532123114264532903689671329$
43152103259505773547621272182134183706006357515644099320875282421708540 9959745236008778839218983091
>>> a $=65$
>>> pow(a ,N-1, N)
28361384576084316965644957136741933367754516545598710311795971496746369 83813383438165679144073738154035607602371547067233363944692503612270610
9766372616458933005882 \# 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: $\left.N=\left(2^{271}+855\right)\left(2^{273}+5\right)\right)$.

## Witnesses for Compositeness: type 3

FERM $_{N}$ is the set of Fermat's witnesses of $N$
This set includes every integer $a$ such that:

- $1<a<N$
- $a^{N-1} \neq 1(\bmod N)$

For example: $3 \in \mathrm{FERM}_{10}$, since: $3^{10-1}(\bmod 10)=19683(\bmod 10)=3 \neq 1$

Note (requires proof, omitted): $\mathrm{FACT}_{N} \subseteq \mathrm{GCD}_{N} \subseteq \mathrm{FERM}_{N}$
How to find such a witness? Go over all $a \in[2, \ldots, N-1]$, and for each $a$ use Fermat's test (test whether $a^{N-1} \neq 1(\bmod N)$ ).
Worst case complexity $O\left(N \cdot n^{3}\right)=O\left(2^{n} \cdot n^{3}\right)$

## Fermat's Witnesses

- You may be wondering when this is all going to...
- All the witnesses we saw require in the worst case (when $N$ is prime) exponential time complexity
- But now randomization will come to our aid
- It was shown by Miller and Rabin (1980) that if $N$ is composite, then at least $1 / 2$ of $\{1,2, \ldots, N-1\}$ are Fermat witnesses for $N$ 's compositeness
- In other words, $\mid$ FERM $_{N} \mid \geq N / 2$
- Example: what are $\mid$ FERM $_{1000} \mid$ and $\mid$ FERM $_{1001} \mid$ ?


## Randomized Primality Testing

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


## Randomized Primality Testing (2)

If $N$ is prime, then by Fermat's little theorem, no $a \in\{2, \ldots, N-1\}$ is a witness.

Picking $a \in\{2, \ldots, N-1\}$ at random yields an algorithm that gives the right answer if $N$ is composite with probability at least $1 / 2$, 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 $1 / 2$.

How can we guarantee a smaller error?

## Randomized Primality Testing (3)

- The input is an integer $N$
- Repeat 100 times
- Pick $a$ in the range $1<a \leq N-1$ at random and independently.
- Check if $a$ is a witness $\left(a^{N-1} \neq 1 \bmod N\right)$ (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".

To err, all random choices of $a$ 's should yield non-witnesses.
Therefore,

$$
\text { Probability of error }<\left(\frac{1}{2}\right)^{100} \lll 1
$$

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


## Randomized Primality Testing in Python

- The default parameter show_witness allows one to see the Fermat witness that was found for $N$.

```
def is_prime(N, show_witness=False):
    """ probabilistic test for N's compositeness """
    for i in range(0, 100):
    a = random.randint (2, N-1) # random integer in [2..N-1]
    if pow(a, N-1, N) != 1: # Fermat's test
                if show witness: # caller wishes to see a witness
                        print(N,"is composite")
                                print(a,"is a witness, found on iteration" ,i+1)
                        return False
    return True
```

```
>>> is_prime(11)
True
>>> is_prime(10)
False
>>> is_prime(10, True)
10 is composite
3 is a witness, found on iteration 1
```


## Time Complexity

- Let $n$ be the number of bits in $N$
- Number of iterations is $100=O(1)$
- Each iteration takes $O\left(n^{3}\right)$, since:
- Randomly selecting $a$ can be done in $O(n)$ time
- Modular exponentiation takes $O\left(n^{3}\right)$
- Overall time complexity is $O\left(n^{3}\right)$


## Pushing Time Complexity 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.
$\ggg N=2 * * 82589933-1$
>>> pow (56, N-1, N) == 1
\# patience, young lads !
\# and even more patience !!
- Here, $n=8258993$, so even a polynomial time $O\left(n^{3}\right)$ algorithm requires quite some time.


## Randomized Primality Testing Summary

Randomized: uses coin flips to pick the $a$ 's.
Run time is polynomial in $n$, the length of $N$
If $N$ is prime, the algorithm always outputs " $N$ is prime".

If $N$ is composite, the algorithm may err and outputs " $N$ is prime".

$$
\text { Probability of error }<\left(\frac{1}{2}\right)^{100} \lll 1 .
$$

## Finding Primes

- Suppose you want to find a prime number with $n$ bits.
- You can sample numbers (in the range $2^{n-1}$ to $2^{n}-1$ ) and check if the sampled number is prime, until you hit one.
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??
- Does this function always halt?
- How many samples are needed on average?


## Comic Relief*



## Final Remarks (for reference only) Carmichael numbers

- We said if $N$ is composite, then $\left|\operatorname{FERM}_{N}\right| \geq N / 2$.
- This is almost true.
- There are some annoying numbers, known as Carmichael numbers, where this does not happen.
- In fact, Carmichael numbers are exactly the composite numbers $N$ where $\mathrm{GCD}_{N}=\mathrm{FERM}_{N}$.
- However:
- These numbers are very rare and it is highly unlikely you'll run into one, unless you really try hard.
- Miller and Rabin devised a similar algorithm, but slightly more sophisticated, which 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.


## Final Remarks (for reference only)

## Deterministic Polynomial-time solution?

- 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.


## Final Remarks (for reference only)

## Deterministic Polynomial-time solution!

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 $O\left(n^{12}\right)$. In 2005, Carl Pomerance and H. W. Lenstra, Jr. improved this to running in time $O\left(n^{6}\right)$.


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

## Final Remarks (for reference only) Back to Integer Factorization

- Trial division can be used to find not only one divisor of a number, but all its prime factors. But as we saw this takes $O\left(2^{n / 2}\right)$ time for an $n$ bit number.
- The best integer factorization algorithm to date, called the general number field sieve algorithm, does so in $O\left(e^{8 n^{1 / 3} \cdot \log ^{2 / 3} n}\right)$.
- Factoring integers is believed to be a hard computational problem, for which we believe there is no polynomial time solution: until now, no Polynomial time algorithm was found. However, it has not been proven that such an algorithm does not exist.
- As you already know, this is not the case with the "opposite direction" - the problem of integer multiplication.
- The presumed one-way computational hardness is important for the algorithms used in cryptography such as RSA public-key encryption and the RSA digital signature.


## Final Remarks (for reference only) Fermat's Last Theorem

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

$$
\begin{gathered}
a^{2}+b^{2}=c^{2} \\
\text { e.g. } a=3, b=4, c=5 \text {, or } a=20, b=99, c=101 \text {, etc. }
\end{gathered}
$$

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).

## Appendix

## (for reference only)

Euclid's gcd: Proof of Correctness Using an Invariant, Multiplicative inverses,

Extended GCD

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

- Suppose $0<1<k$.
- We first show that $\operatorname{gcd}(\mathrm{k}, \mathrm{I})=\operatorname{gcd}(\mathrm{I}, \mathrm{k}-\mathrm{I})$.
- Denote $\mathrm{g}=\operatorname{gcd}(\mathrm{k}, \mathrm{l}), \mathrm{h}=\operatorname{gcd}(\mathrm{I}, \mathrm{k}-\mathrm{I})$
- Since g divides both k and I , it also divides k - I .
- Thus it divides both 1 and k - .
- Since $h$ is the greatest common divisor of $I$ and $k-I$, every divisor of I and k - I 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 $\mid$ and $k-\mid$ is also a divisor of $k$.
- Thus $h$ divides the gcd of $k$ and $I$, which is $g$.
- So $g$ divides $h$ and $h$ divides $g$.
- They are both positive, therefore g equals $h$.
- QED
- Note: 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 $0<\mathrm{I} \leq \mathrm{k}$. We just showed that $\operatorname{gcd}(\mathrm{k}, \mathrm{I})=\operatorname{gcd}(\mathrm{k}-\mathrm{I}, \mathrm{I})$.
- If $\mathrm{k}-\mathrm{I}<\mathrm{I}$, then $\mathrm{k}(\bmod \mathrm{I})=\mathrm{k}-\mathrm{I}$, and we are done.
- Otherwise, $k-\mid \geq \operatorname{gcd}(k, I)=\operatorname{gcd}(k-I, I)$.
- Repeating the previous argument, $\operatorname{gcd}(\mathrm{k}-\mathrm{I}, \mathrm{I})=\operatorname{gcd}(\mathrm{k}-2 \cdot \mathrm{I}, \mathrm{I})$.
- There is a unique $\mathrm{m} \geq 1$ such that $\mathrm{k}(\bmod \mathrm{I})=\mathrm{k}-\mathrm{ml}$.
- By the argument above,

$$
\begin{aligned}
& \operatorname{gcd}(\mathrm{k}, \mathrm{I})=\operatorname{gcd}(\mathrm{I}, \mathrm{k}-\mathrm{I})=\operatorname{gcd}(\mathrm{I}, \mathrm{k}-2 \cdot I)=\operatorname{gcd}(I, \mathrm{k}-3 \cdot I)= \\
& \cdots \cdot=\operatorname{gcd}(\mathrm{I}, \mathrm{k}-\mathrm{m} \cdot I)=\operatorname{gcd}(\mathrm{I}, \mathrm{k} \bmod I)
\end{aligned}
$$

## Proof of Correctness Using an Invariant:

## Conclusion

- In every iteration of Euclid's algorithm, we replace ( $k, I$ ) by (I, k mod I), 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)

QED

## Relative Primality and Multiplicative Inverses

$\operatorname{gcd}(28,31)=1$
$\operatorname{gcd}(12,35)=1$
$\operatorname{gcd}(527,621)=1$
$\operatorname{gcd}(1002,973)=1$

If $\operatorname{gcd}(k, m)=1$, we say that $k, m$ are relatively prime.
Suppose $\mathrm{k}, \mathrm{m}$ are relatively prime, and $\mathrm{k}<\mathrm{m}$.
Then there is a positive integer, $\mathrm{a}, \mathrm{a}<\mathrm{m}$, such that

$$
a \cdot k=1(\bmod m)
$$

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

$$
a \cdot k=1(\bmod m)
$$

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

$$
\begin{gathered}
10 \cdot 28=1(\bmod 31) \\
3 \cdot 12=1(\bmod 35) \\
218 \cdot 527=1(\bmod 621) \\
817 \cdot 937=1(\bmod 1002)
\end{gathered}
$$

