Extended Introduction to Computer Science
CS1001.py

Lecture 13B: Number Theoretic Algorithms: Factoring Integers, primality testing

Guest Instructor: Benny Chor

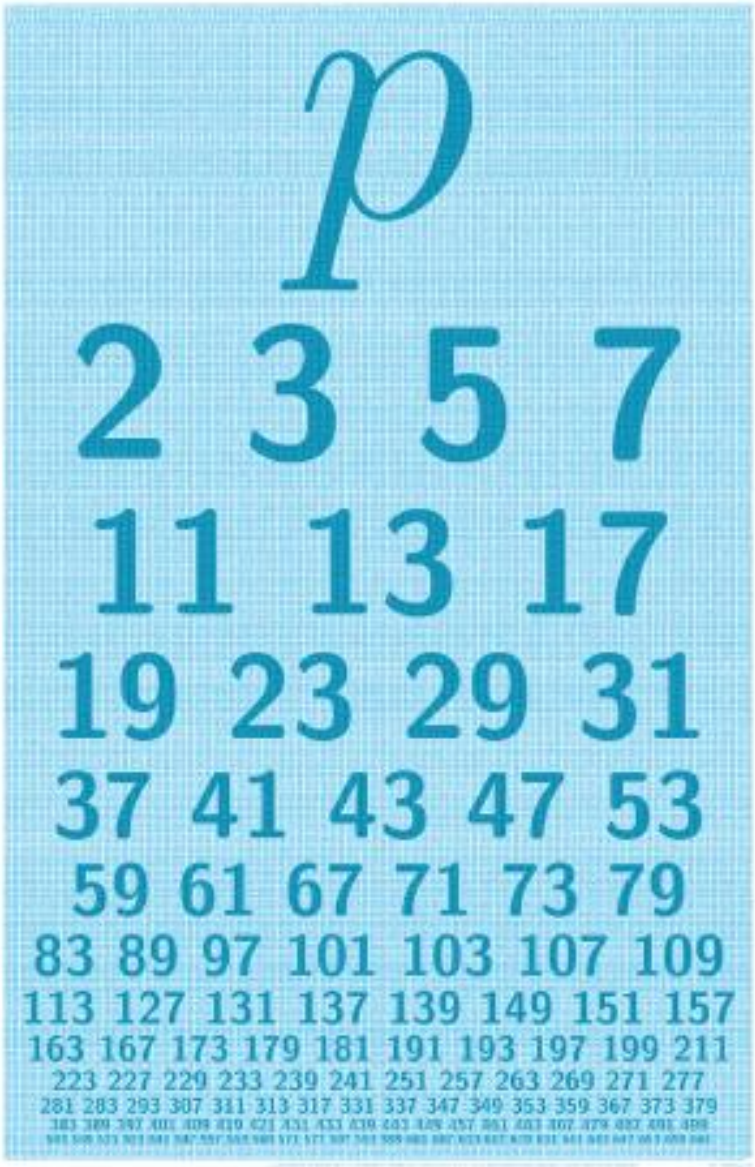
Lecture 13B: Plan

Number Theory Algorithms:

- Prime Numbers:
- Trial division
- Fermat's "little theorem"
- randomized primality testing.

**

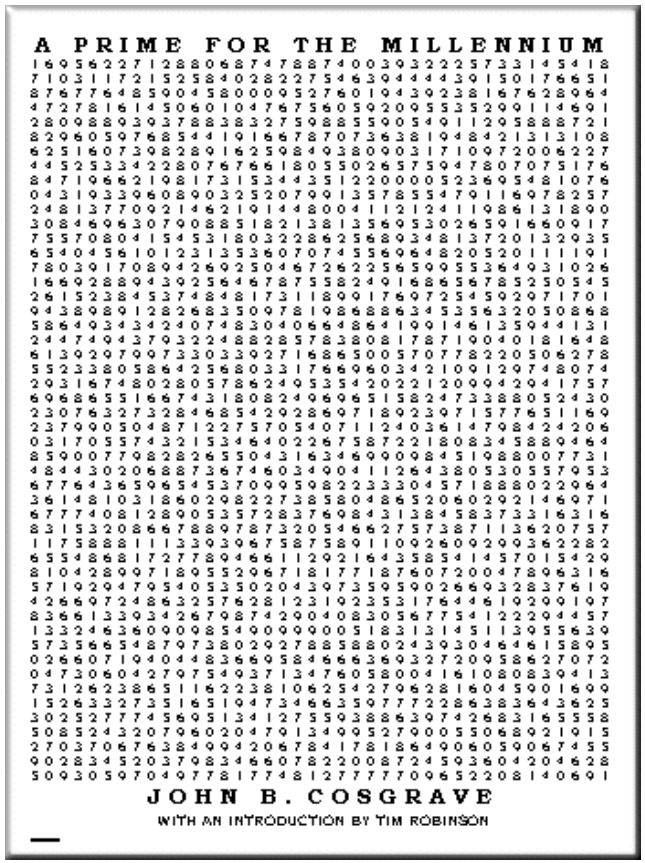
Prime Numbers and Randomized Primality Testing



(figure taken from the 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 חסר כאן וגם 2 בחזקת 57,885,161 פחות 1. מספרים ראשוניים

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

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

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.

(screenshot from scientific 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: Python Code

```
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!

Try, , *e.g.* the 61 bit number $2^{61} - 1$.

```
>>> trial_division(2**61-1)
2305843009213693951 is prime
True
```

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
0.00282299999999909
>>> elapsed("trial_division (2**40+15)")
1099511627791 is prime
0.1665870000000004
>>> 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
0.035039999999995075
>>> elapsed("trial_division (2**55+11)")
36028797018963979 is prime
29.706794
>>> elapsed("trial_division(2**61-1)")
2305843009213693951 is prime
290.931049    # almost 5 minutes
```

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.

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 ring Z_p^*. But not being familiar with (mathematical) groups, rings, 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 saw how to modify the exponentiation function 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 (reminder)

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 - coming up next.
- Diffie Hellman Key Exchange - next class.

- Rivest-Shamir-Adelman (RSA) public key cryptosystem (PKC) - in an elective crypto course.

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 that if N is composite, then at least 1/2 of all $a \in \{1, \ldots, N-1\}$ are witnesses.

Randomized Primality Testing (2)

It was shown that if N is composite, then at least $1/2$ 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 $1/2$, 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 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".
- Solovay-Strassen Miller-Rabin used the fact that if N is composite, then at least $1/2$ of all $a \in \{1, \ldots, N-1\}$ are witnesses.
- To err, all random choices of a's should yield non-witnesses. Therefore,

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

Properties of Fermat Primality Testing, cont.

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

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

- 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 (maybe not such a great example with the present US president), 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+1)
            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 ladies and 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, the number of bits is $n = 57,885,161$.

Trust, But Check!

We said (quoting S&S, R&M) that if if N is composite, then at least 1/2 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 "intro to modern 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).