Computer Science 1001.py, Lecture 11b:

Number Theoretic Algorithms:
Factoring Integers by Trial Division
Randomized Primality Testing

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, 2015
http://tau-cs1001-py.wikidot.com

Lecture 11b: Plan

- Trial division and its computational complexity.
- Modular exponentiation (reminder).
- Fermat's little theorem.
- Randomized primality testing.

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):
    """ trial division for integer N """
    upper = round(N ** 0.5 + 0.5) # ceiling function of sqrt(N)
    for m in range(2,upper+1):
        if N % m == 0: # m divides N
            print(m, "is the smallest divisor of", N)
            return None
    # we get here if no divisor was found
    print(N, "is prime")
    return None
```

Trial Division: A Few Executions

Let us now run this on a few cases:

```
>>> 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.

Trial Division Performance: Actual Measurements

Let us now measure actual performance on a few cases. We first run the `clock` module (written by us), where the `elapsed` function is defined. Then we `import` the `trial_division` function.

```
>>> from division import trial_division
>>> 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
0.02222199999999964
>>> elapsed("trial_division (2**50+55)")
1125899906842679 is prime
5.829111
>>> elapsed("trial_division (2**55+9)")
5737 is the smallest divisor of 36028797018963977
0.003503999999995075
>>> 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?

Beyond Trial Division

Two possible directions:

- Find an efficient integer factoring algorithm.
- 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 ℓ 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 $\left\{a^1 \bmod c, a^2 \bmod c, a^4 \bmod c, a^8 \bmod c, \ldots, a^{2^{\ell-1}} \bmod c\right\}$.

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.

Efficient Modular Exponentiation: Complexity Analysis

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

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

Each product is smaller than c^2, which has at most 2ℓ bits, and computing the remainder of such product modulo c takes another $O(\ell^2)$ bit operations (using long division, also studied in elementary school).

All by all, computing $a^b \bmod c$, where $a, b, c \geq 2$ are all ℓ bit integers, takes $O(\ell^3)$ 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:       # 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):

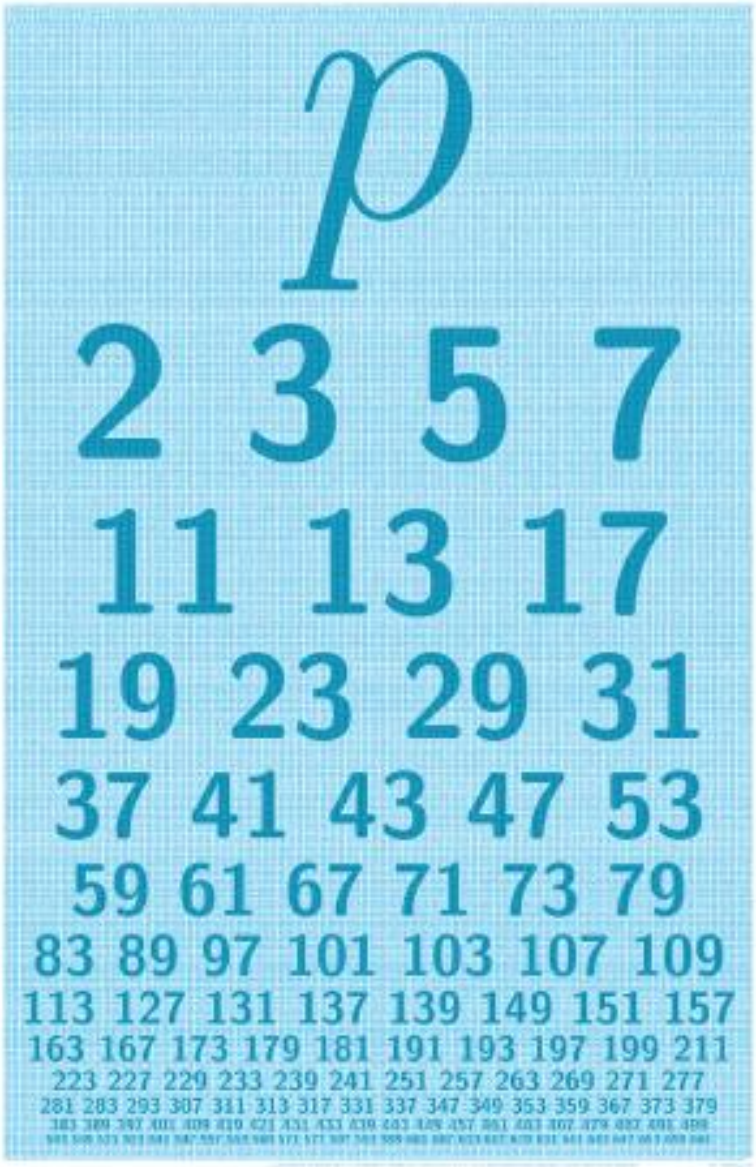
- Randomized primality testing.
- Diffie Hellman Key Exchange
- Rivest-Shamir-Adelman (RSA) public key cryptosystem (PKC)

We will discuss the first topic today, and the second topic next time.

We leave RSA PKC to the (elective) crypto course.

Intentionally Left Blank

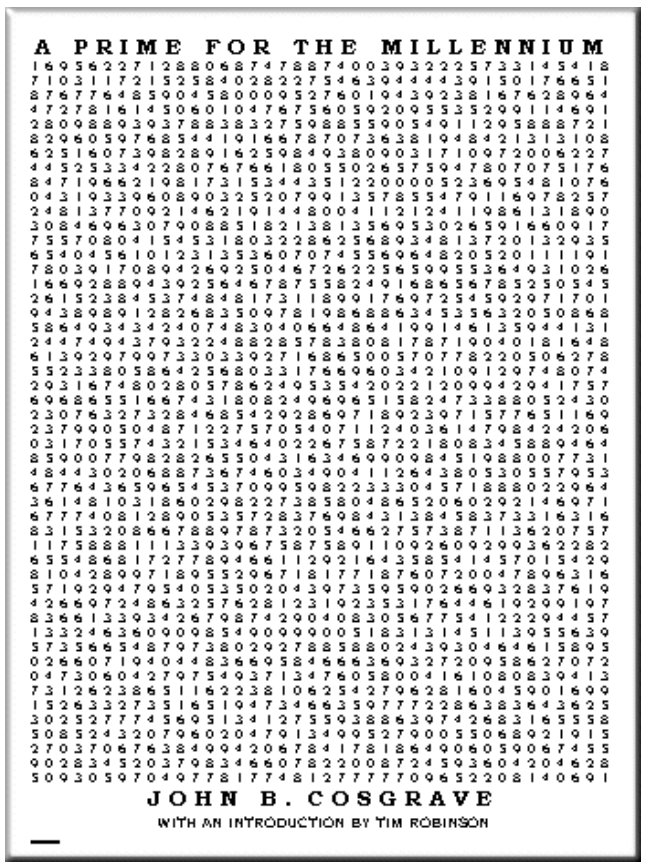
Prime Numbers and Randomized Primality Testing



(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.

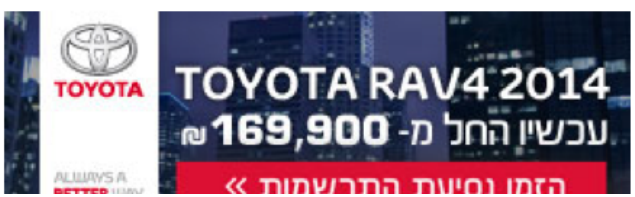
הבונוס על החשיפה: 3,000 דולר

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 an integer x.

- The prime number theorem: A random n bit number is a prime with probability roughly $1/n$.
- 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).

Randomized Testing of Primality/Compositeness

- Now that we know there are heaps of primes, we would like to efficiently test if a given integer is prime.
- Given an n bits integer m, $2^{n-1} \leq m < 2^n$, we want to determine if m is composite.
- The search problem, "given m , find all its factors" is believed to be intractable.
- 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.)

- Does this imply that determining if m is prime is also (believed to be) intractable?

Randomized Primality (Actually Compositeness) Testing

Question: Is there a better way to solve the decision problem (test if m is composite) than by solving the search problem (factor m)?

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

Fermat Test: Example

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

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

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

Randomized Primality Testing

- The input is an integer m with n bits ($2^{n-1} < m < 2^n$)
- 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 a is a witness, output "m is composite".
- If no witness found, output "m is prime".

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

Randomized Primality Testing (2)

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

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

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

However, this means that if m 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 m with n bits ($2^{n-1} < m < 2^n$)
- 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 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".
- 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,

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

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