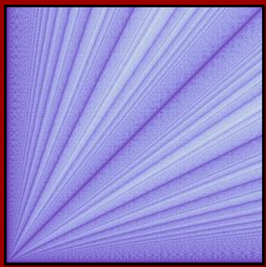
15-251: Great Theoretical Ideas in Computer Science
Lecture 20

Computational Arithmetic



Let B be a natural number.   Say, a Big one.

B = 36185027886661311069865932815214971104557430211692603585367759320207626861017237846234873269807102970128874356021481964232857782295671675021393065473695394365322208211694158783076964982631058971773918152503322026635065098926803831948392738815054324220771791218388882819961484080523021968898666372006062526501310964926475205090003984176122058711164567946559044971683604424076996342718304654479802116829701349077414009047634829067182274396120369814230709966434551334146376168244238601078897410581312713062262142086360082246515109610189789006815067664901594246966730927620844732714004599013904409378141724958467722895014360827736997469288319568431436186292967922716752485131607758720764878450583672316031730798174714175190513570296719911529635804128381848417337 82

Let B be a natural number.   Say, a Big one.

B = 56930300205239999934796429046219117250985670205562581027662514872340310944 29

B ≈ 5.7×10^{75} (5.7 “quattorvigintillion”)

B’s **magnitude** is enormous.

Roughly the *number of atoms in the universe*, or the *age of the universe in Planck time units*.

Definition (for today’s lecture):

len(B) = # bits to write B
≈ $\log_2(B)$

For our 5.7 quattorvigintillion B,
len(B) = 251 bits ≈ 32 bytes.

For cryptography purposes, this B is so very small!
SSH / RSA keys usually 10 times longer, 2048 bits.

Arithmetical algorithms on big numbers

Remember:
- Arithmetic is not free.
- Numbers are written in binary.
- If input is B, input length is len(B).
- Think of algorithms as performing **string**-manipulation.
- Think about solving these problems **by hand** on 20-digit numbers.

Addition

```
  36185027886661311069865932815214971104   A
+ 65743021169260358536775932020762686101   B
= 101928049055921669606641864835977657205  C
```

Easy to check “grade school algorithm” is **linear time**.

I.e., assuming len(A), len(B) ≤ n, running time to produce C is O(n).

Multiplication

$$\begin{array}{rl} 36185027886661311069865932815214971104 & A \\ \times \quad 5932020762686101 & B \\ \hline 2146503367220504639466513582026984044526098681374255 04 & C \end{array}$$

len(B) rows

len(C) = log(A×B) = log(A)+log(B) = len(A)+len(B) ≤ 2len(A)

Running time = O(len(A) len(B)).
If len(A), len(B) ≤ n, this is O(n^2).
Even faster algs exist, used in practice.
Running time slightly worse than O(n log n).

Division

$$\begin{array}{rl} 36185027886661311069865932815214971104 & A \\ \div \quad 5932020762686101 & B \end{array}$$

Division

Q

6099949635084593037586

B 5932020762686101 | 36185027886661311069865932815214971104 A

3960087002178918

R

A = Q·B+R

$Q = \left\lfloor \frac{A}{B} \right\rfloor$

R = A mod B

This alg also runs in O(len(A) len(B)) time.

Prime factorization

A = 5693030020523999993479642904621911725098567020556258102766251487234031094429

Say we even just want to find **some** divisor of A.

for B = 2, 3, 4, 5, …
  test if A mod B = 0

Prime factorization

A = 5693030020523999993479642904621911725098567020556258102766251487234031094429

Say we even just want to find **some** divisor of A.

for B = 2, 3, 4, 5, …
  test if A mod B = 0

The solution is:

68452332409801603635385895997250919383 × 83167801886452917478124266362673045163

Each factor is ≈ *age of universe in Planck time*.

Prime factorization

A = 5693030020523999993479642904621911725098567020556258102766251487234031094429

Say we even just want to find **some** divisor of A.

for B = 2, 3, 4, 5, …
  test if A mod B = 0

Can check B up to $\sqrt{A}$ but $\sqrt{A} = \sqrt{2^{\mathrm{len}(A)}} = 2^{\frac{1}{2}\mathrm{len}(A)}$.

Running time is **exponential** in input length.

Prime factorization

There are significantly faster known algorithms, but they're all still exponential time.

Indeed, much of **cryptography** relies on the assumption that there is no efficient factoring algorithm!

Primality testing

A = 5693030020523999993479642904621911725098567020556258102766251487234031094429

Try the following in Maple (or Wolfram Alpha):

ifactor(5693030020523999993479642904621911725098567020556258102766251487234031094429)

It will calculate forever, until you stop it.

Then try:

isprime(5693030020523999993479642904621911725098567020556258102766251487234031094429)

It will output *false* instantly.

How does it do that?!

Primality testing

An **amazing** fact:
There's a poly-time algorithm for primality testing.

Agrawal, Kayal, Saxena, 2002

undergraduates, at the time

Primality testing

Although, that's not what Maple / WA use.

The best version of AKS Algorithm is ~$O(n^{6})$ time.

Not feasible when run on an $n=2048$ bit number.

Everyone uses the **Miller–Rabin** algorithm (1975).

CMU professor

Primality testing

Although, that's not what Maple / WA use.

The best version of AKS Algorithm is ~$O(n^{6})$ time.

Not feasible when run on an $n=2048$ bit number.

Everyone uses the **Miller–Rabin** algorithm (1975).

Its running time is ~$O(n^{2})$.

What's the catch?

It's a **randomized algorithm**.
It errs with some tiny probability (say, 2^{-100}).

Generating a prime

Say we need an n-bit long prime number.

loop:	let B be a random n-bit number test if B is prime

"Prime Number Theorem":
About $1/n$ fraction of n-bit numbers are prime.
⇒ Expected running time of above is ~$O(n^3)$.

This is basically the only known algorithm!
No poly-time **deterministic** alg. is known!

Primality testing again

By the end of class, you'll be able to prove:

Wilson's Theorem:
B is prime ⇔ (B−1)! + 1 divisible by B.

Q: Why not use this as a primality test?

Midterm 2: **Cannot** compute (B−1)! in poly time.

Cannot even compute 2^B in poly time. Why?

If B is 5.7 quattorvigintillion (len(B)=251)
answer length exceeds *# of particles in universe*!

Modular Exponentiation

However, you **can** compute 2^B mod C
in polynomial time.

In general, assuming len(A), len(B), len(C) ≤ n,

can compute $\boxed{A^B \bmod C}$ in poly(n) time.

Let's prove this.

Modular Exponentiation

Example: Compute 2337^{32} mod 100.
By hand.

Bad idea: 2337 × 2337 = 5461569
2337 × 5461569 = 12763686753
2337 × 12763686753 = · · ·

(30 more multiplications later...)

= 6267275651521555116531888866686685883134758242366656073967550089057701462366355372282166960309706128289228**81**

Modular Exponentiation

Example: Compute 2337^{32} mod 100.
By hand.

Smart idea 1:

Reduce mod 100 after every step.

Smart idea 2:

Don't multiply 32 times; square 5 times.

$2337^1 \to 2337^2 \to 2337^4 \to 2337^8 \to 2337^{16} \to 2337^{32}$

Modular Exponentiation

Smart idea 2:

Don't multiply 32 times; square 5 times.

$2337^1 \to 2337^2 \to 2337^4 \to 2337^8 \to 2337^{16} \to 2337^{32}$

Lucky (?) that exponent was a power of 2.

Q: What if we had wanted 2337^{34} ?

A: Multiply together 2337^{32} and 2337^2.

Modular Exponentiation

Smart idea 2:

Don't multiply 32 times; square 5 times.

$2337^1 \to 2337^2 \to 2337^4 \to 2337^8 \to 2337^{16} \to 2337^{32}$

Lucky (?) that exponent was a power of 2.

Q: What if we had wanted 2337^{53} ?

A: Multiply powers: 32 + 16 + 4 + 1.

Here I used that binary rep. of 53 is 110101.

Modular Exponentiation

In general, to compute $A^B \bmod C$,
where A, B, C are ≤ n bits long:

1. Repeatedly square A, always mod C.
   Do this n times.
2. Multiply together the powers of A corresponding to binary digits of B (again, always mod C).

Running time is a little more than $O(n^2 \log n)$.

Greatest Common Divisor (GCD)

A = 65743021169260358536775932020762686101

B = 36185027886661311069865932815214971104

What is G = gcd(A,B)?

Grade school algorithm:

1. Factor A and B.
2. ☹

Euclid's Algorithm finds GCD in poly-time!
It's arguably the **first ever algorithm**.

(PS: It was not invented by Euclid.
It was invented some time in 500—375 B.C.E.)

Greatest Common Divisor (GCD)

What is G = GCD(A,B)?

Observation 1: Suppose g is a divisor of A & B.
Then g is also a divisor of A−B.

E.g., 6 is a divisor of 600 & 12,
so 6 is also a divisor of 588.

But is **GCD**(A,B) = **GCD**(A−B,B)? Yes!
If so, we "make progress"
by reducing the size of our two numbers.

Greatest Common Divisor (GCD)

What is G = GCD(A,B)?

Observation 1: Suppose g is a divisor of A & B.
Then g is also a divisor of A−B.

Conversely: If g is a divisor of A−B & B,
then g is also a divisor of A.

But is **GCD**(A,B) = **GCD**(A−B,B)? Yes!

Greatest Common Divisor (GCD)

What is G = GCD(A,B)?

Observation 1: Suppose g is a divisor of A & B.
Then g is also a divisor of A−B.

Conversely: If g is a divisor of A−B & B,
then g is also a divisor of A.

Therefore: The common divisors of A & B
are exactly the same as
the common divisors of A−B & B.

So indeed GCD(A,B) = GCD(B,A−B).

Warmup to Euclid's GCD Algorithm

GCD(42,30) = GCD(30,12) (using 42−30=12)
= GCD(18,12) (using 30−12=18)
= GCD(12,6) (using 18−12=6)
= GCD(6,6) (using 12−6=6)
= GCD(6,0) (using 6−6=0)

Stop when you get to 0, as GCD(A,0) = A.

Answer: GCD(42,30) = **6**.

Warmup to Euclid's GCD Algorithm

Cool, let's do another one!

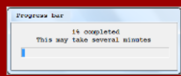
GCD(6004,6) = GCD(5998,6)

= GCD(5992,6)

= GCD(5986,6)

• • •

= GCD(4 ,6)



In general:

GCD(A,B) eventually gets to GCD(A mod B,B).

Euclid's GCD Algorithm:

GCD(A,B):
if B = 0, return A
return **GCD**(B, A mod B)

Example: GCD(100,18)

= GCD(18,10) (using 100 mod 18 = 10)

= GCD(10,8) (using 18 mod 10 = 8)

= GCD(8,2) (using 10 mod 8 = 2)

= GCD(2,0) (using 8 mod 2 = 0)

= **2**

Euclid's GCD Algorithm:

GCD(A,B):
if B = 0, return A
return **GCD**(B, A mod B)

Summary:

100

18

10

8

2

Euclid's GCD Algorithm:

GCD(A,B):
if B = 0, return A
return **GCD**(B, A mod B)

Run-time? Each step computes a "mod", which is polynomial time.

So suffices to show only poly many steps.

Euclid's GCD Algorithm:

GCD(A,B):
if B = 0, return A
return **GCD**(B, A mod B)

Run-time?

A

B

A mod B

• • •

Claim:
Decreases by factor ½ or more

Proof: If A ≥ 2B then it's true, ∵ (A mod B) < B.

If A < 2B then it's true,
∵ we subtracted off B, which is ≥ ½ A.

Euclid's GCD Algorithm:

GCD(A,B):
if B = 0, return A
return **GCD**(B, A mod B)

Run-time?

A

B

A mod B

So this product…

…is ≤ ½ of this product.

A·B goes down by factor of ½ or better at each step.

Euclid's GCD Algorithm:

GCD(A,B):
    if B = 0, return A
    return **GCD**(B, A mod B)

Run-time? ∴ total # of steps is:

≤ log(A·B) = log(A) + log(B) = len(A) + len(B)

O(n) steps if len(A), len(B) ≤ n.

A·B goes down by factor of ½ or better at each step.

Euclid's GCD Algorithm:

GCD(A,B):
    if B = 0, return A
    return **GCD**(B, A mod B)

Run-time? ∴ total # of steps is:

≤ log(A·B) = log(A) + log(B) = len(A) + len(B)

O(n) steps if len(A), len(B) ≤ n.

∴ total run-time is poly(n).

(In fact, roughly $O(n^2)$.)

The intrinsic complexity of GCD

Euclid's Algorithm computes GCD in ~$O(n^2)$ time.

Not so great in practice. Say n = 100,000?

There are faster algorithms! ~O(n log n), in fact.

Major open problem in computer science:
Is GCD computation efficiently parallelizable?

I.e., is there a circuit family (C_n) with poly(n) gates and **polylog(n) depth** that computes the GCD of two n-bit numbers?

A bonus from Euclid's Algorithm…

Definition:

Say that C is a **miix** of A and B if it's an integer linear combination of them:
$C = k\cdot A + \ell\cdot B$ for some $k,\ell \in \mathbb{Z}$.

(Note: Not a real term. You are not allowed to use it. ☺)

Example: 2 is a miix of 14 and 10 because 2 = (−2)·14 + 3·10

(Hence **any** multiple of 2 is a miix of 14 and 10. To get 2m as a miix, multiply the equation by m.)

Definition:

Say that C is a **miix** of A and B if it's an integer linear combination of them:
$C = k\cdot A + \ell\cdot B$ for some $k,\ell \in \mathbb{Z}$.

(Note: Not a real term. You are not allowed to use it. ☺)

Non-example:

7 is **not** a miix of 55 and 40, because any miix would be divisible by 5

7 is **not** a miix of 55 and 40,
because any miix would be divisible by 5

If A and B are both divisible by some F
then any miix of A and B must be too.

So if C is a miix of A and B,
then C must be a multiple of GCD(A,B).

Conversely, is GCD(A,B) always a miix of A and B?

Yes! It's a bonus of Euclid's GCD Algorithm.

Euclid's GCD Algorithm:

GCD(A,B):
  if B = 0, return A
  return **GCD**(B, A mod B)

Example: GCD(100,18)
= GCD(18,10) (using 100 mod 18 = 10)
= GCD(10,8) (using 18 mod 10 = 8)
= GCD(8,2) (using 10 mod 8 = 2)
= GCD(2,0) (using 8 mod 2 = 0)
= **2**

Summary of Euclid getting GCD(100,18) = 2:

100 18 10 8 2

8 is a miix of 18 & 10

2 is a miix of 10 & 8

∴ 2 is a miix of 18 & 10

Fact #1: If A mod B = R then R is a miix of A and B.
Because by definition, R = A − qB for some q.

Fact #2: If R is a miix of A and B,
and B is a miix of A and C,
then R is a miix of A and C.

Summary of Euclid getting GCD(100,18) = 2:

100 18 10 8 2

10 is a miix of 100 & 18

∴ 2 is a miix of 18 & 10

Fact #1: If A mod B = R then R is a miix of A and B.
Because by definition, R = A − qB for some q.

Fact #2: If R is a miix of A and B,
and B is a miix of A and C,
then R is a miix of A and C.

Summary of Euclid getting GCD(100,18) = 2:

100 18 10 8 2

10 is a miix of 100 & 18

∴ 2 is a miix of 100 & 18

Fact #1: If A mod B = R then R is a miix of A and B.
Because by definition, R = A − qB for some q.

Fact #2: If R is a miix of A and B,
and B is a miix of A and C,
then R is a miix of A and C.

Summary of Euclid getting GCD(100,18) = 2:

100 18 10 8 2

10 is a miix of 100 & 18

∴ 2 is a miix of 100 & 18

Summary:
If G = GCD(A,B), then G is a miix of A and B.
And you can get the k and ℓ such that
G = k·A + ℓ·B
from Euclid's Alg. with a little bookkeeping.

Summary of arithmetical algs.

Poly time:
- Addition
- Multiplication
- Integer division & mod
- Primality testing
- GCD
- Modular exponentiation

Believed not poly time:
- Factoring

Not poly time:
- Factorial
- Non-modular exponentiation

Modular arithmetic refresher

Sometimes in arithmetic we "*work mod M*".
E.g., on a clock, the hours go mod 12.
In computer hardware, arithmetic is often mod 2^{64}.

"A and B are equivalent mod M",

" $A \equiv_M B$ ",

means A, B have same remainder mod M.

mod M, every integer is equivalent to exactly one of 0, 1, 2, 3, …, M−1.

Addition mod M

Addition, +, "plays nice" mod M:

$$A \equiv_M B$$
$$A' \equiv_M B'$$
$$\Rightarrow \quad A+A' \equiv_M B+B'$$

We may define a new number system
$\mathbb{Z}_M$
with elements 0, 1, 2, …, M−1,
and basic operation +.

Addition mod M

E.g.: $\mathbb{Z}_5 = \{0, 1, 2, 3, 4\}$, with this + table…

+	0	1	2	3	4
0	0	1	2	3	4
1	1	2	3	4	0
2	2	3	4	0	1
3	3	4	0	1	2
4	4	0	1	2	3

(0 has special property: 0+A = A+0 = A for all A)

Subtraction mod M

"*What about* subtraction *in* $\mathbb{Z}_M$*?*", you might say.

To define it, we first define "−B".
Then "A−B" just means "A + (−B)".

Given B, we define "−B" to be
"the number in $\mathbb{Z}_M$ such that B + (−B) = 0".

Negatives mod M

+	0	1	2	3	4
0	0	1	2	3	4
1	1	2	3	4	0
2	2	3	4	0	1
3	3	4	0	1	2
4	4	0	1	2	3

In $\mathbb{Z}_5$ …

−2 = 3

−4 = 1

−0 = 0

Note: −B exists & is unique because each row is a **permutation** of 0, 1, 2, …, M−1, so 0 appears exactly once.

Multiplication mod M

Multiplication, •, also "plays nice" mod M:

$$A \equiv_M B$$
$$A' \equiv_M B'$$
$$\Rightarrow \quad A \bullet A' \equiv_M B \bullet B'$$

Multiplication mod 5

•	0	1	2	3	4
0	0	0	0	0	0
1	0	1	2	3	4
2	0	2	4	1	3
3	0	3	1	4	2
4	0	4	3	2	1

(1 has special property: 1•A = A•1 = A for all A)

Division mod M

"*What about* division *in* $\mathbb{Z}_M$*?*", you might say.

Similar to subtraction, we'd like to define "B^{-1}".
Then "A÷B" could just mean "$A \bullet B^{-1}$".

So given B, can we define "B^{-1}" to be
"the number in $\mathbb{Z}_M$ such that $B \bullet B^{-1} = 1$"?

There are some problems…

Reciprocals mod 5

•	0	1	2	3	4
0	0	0	0	0	0
1	0	1	2	3	4
2	0	2	4	1	3
3	0	3	1	4	2
4	0	4	3	2	1

$0^{-1} =$ undefined
$1^{-1} = 1$
$2^{-1} = 3$
$3^{-1} = 2$
$4^{-1} = 4$

Well, that's all right.
We're used to not being able to divide by 0.

Reciprocals mod 6

•	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	1	2	3	4	5
2	0	2	4	0	2	4
3	0	3	0	3	0	3
4	0	4	2	0	4	2
5	0	5	4	3	2	1

$0^{-1} =$ undefined
$1^{-1} = 1$
$2^{-1} =$ undefined!
$3^{-1} =$ undefined!
$4^{-1} =$ undefined!
$5^{-1} = 5$

Huh. We only have two #'s with reciprocals.

Reciprocals mod 7

•	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6
2	0	2	4	6	1	3	5
3	0	3	6	2	5	1	4
4	0	4	1	5	2	6	3
5	0	5	3	1	6	4	2
6	0	6	5	4	3	2	1

Every number except 0 has
a multiplicative inverse.

Reciprocals mod 8

•	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7
2	0	2	4	6	0	2	4	6
3	0	3	6	1	4	7	2	5
4	0	4	0	4	0	4	0	4
5	0	5	2	7	4	1	6	3
6	0	6	4	2	0	6	4	2
7	0	7	6	5	4	3	2	1

{1, 3, 5, 7} have inverses; {0, 2, 4, 8} don't

When does B have a reciprocal mod M?

⇔ $\exists\, k$ such that $k \cdot B \equiv_M 1$

⇔ $\exists\, k, q$ such that $k \cdot B = q \cdot M + 1$

⇔ $\exists\, k, q$ such that $1 = k \cdot B + (-q) \cdot M$

⇔ 1 is a "miix" of B and M

⇔ $\boxed{\mathrm{GCD}(B,M) = 1}$

When does B have a reciprocal mod M?

⇔ $\boxed{\mathrm{GCD}(B,M) = 1}$

Check: mod 5: {1,2,3,4} had reciprocals

mod 6: {1,5} had reciprocals

mod 7: {1,2,3,4,5,6} had reciprocals

mod 8: {1,3,5,7} had reciprocals

Note: mod a prime, **all** nonzeros have reciprocal

Definition:

$\mathbb{Z}^*_M$ is the set of numbers B, mod M, which have GCD(B,M) = 1; i.e., have reciprocals.

Weird notation: $\varphi(M) = |\mathbb{Z}^*_M|$.

Important fact:

$\mathbb{Z}^*_M$ is "closed" under multiplication mod M.

I.e., $A, B \in \mathbb{Z}^*_M \Rightarrow A \cdot B \in \mathbb{Z}^*_M$

Proof: $A \cdot B$ has a reciprocal, namely $B^{-1} \cdot A^{-1}$.

$\mathbb{Z}^*_5$

•	0	1	2	3	4
0	0	0	0	0	0
1	0	1	2	3	4
2	0	2	4	1	3
3	0	3	1	4	2
4	0	4	3	2	1

$\varphi(5) = 4$

In general, if P is prime then $\varphi(P) = P-1$.

$\mathbb{Z}^*_8$

•	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7
2	0	2	4	6	0	2	4	6
3	0	3	6	1	4	7	2	5
4	0	4	0	4	0	4	0	4
5	0	5	2	7	4	1	6	3
6	0	6	4	2	0	6	4	2
7	0	7	6	5	4	3	2	1

$\mathbb{Z}_8^*$

•	1	3	5	7
1	1	3	5	7
3	3	1	7	5
5	5	7	1	3
7	7	5	3	1

$\varphi(8) = 4$

$\mathbb{Z}_{15}^*$

•	1	2	4	7	8	11	13	14
1	1	2	4	7	8	11	13	14
2	2	4	8	14	1	7	11	13
4	4	8	1	13	2	14	7	11
7	7	14	13	4	11	2	1	8
8	8	1	2	11	4	13	14	7
11	11	7	14	2	13	1	8	4
13	13	11	7	1	14	8	4	2
14	14	13	11	8	7	4	2	1

$\varphi(15) = 8$

Exercise:

If P,Q distinct primes, $\varphi(PQ) = (P-1)(Q-1)$.

•	1	2	4	7	8	11	13	14
1	1	2	4	7	8	11	13	14
2	2	4	8	14	1	7	11	13
4	4	8	1	13	2	14	7	11
7	7	14	13	4	11	2	1	8
8	8	1	2	11	4	13	14	7
11	11	7	14	2	13	1	8	4
13	13	11	7	1	14	8	4	2
14	14	13	11	8	7	4	2	1

Observation:
Each row of $\mathbb{Z}_M^*$ times table is a permutation of $\mathbb{Z}_M^*$.

(All entries in a row distinct: if A·B=A·B' then multiply by A^{-1} to deduce B=B'.)

Suppose we multiply all entries in row A

By definition: (A·1)(A·2)(A·4)(A·7)(A·8)(A·11)(A·13)(A·14)

But by permutation ppty: = (1)(2)(4)(7)(8)(11)(13)(14)

Dividing thru by common factor: $A^8 = 1$

•	1	2	4	7	8	11	13	14
1	1	2	4	7	8	11	13	14
2	2	4	8	14	1	7	11	13
4	4	8	1	13	2	14	7	11
7	7	14	13	4	11	2	1	8
8	8	1	2	11	4	13	14	7
11	11	7	14	2	13	1	8	4
13	13	11	7	1	14	8	4	2
14	14	13	11	8	7	4	2	1

Observation:
Each row of $\mathbb{Z}_M^*$ times table is a permutation of $\mathbb{Z}_M^*$.

(All entries in a row distinct: if A·B=A·B' then multiply by A^{-1} to deduce B=B'.)

This works in any $\mathbb{Z}_M^*$ and you get $A^{\varphi(M)} = 1$.

Dividing thru by common factor: $A^8 = 1$

Euler's Theorem:

For any M and any A with GCD(A,M) = 1,

$A^{\varphi(M)} \equiv_M 1$

Fermat's Little Theorem:

(corollary when M is prime)

If P is prime and A is not divisible by P,

$A^{P-1} \equiv_P 1$

Fermat's Little Theorem:

If P is prime and A is not divisible by P,

$A^{P-1} \equiv_P 1$

This suggests a potential Primality test…

Given M:

Pick a few random A's between 1 and M−1.
For each, compute A^{M-1} mod M. (Modular exponentiation.)
If you ever get ≠ 1, output "**M is composite**".
Otherwise, output, "**M is probably prime**".

Given M:

Pick a few random A's between 1 and M−1.

For each, compute A^{M-1} mod M. (Modular exponentiation.)

If you ever get ≠ 1, output "**M is composite**".

Otherwise, output, "**M is probably prime**".

This test does not work! ☹

There are a few, extremely rare, numbers M called **Carmichael Numbers** for which A^{M-1} mod M = 1 for all A, even though M is composite.

Given M:

Pick a few random A's between 1 and M−1.

For each, compute A^{M-1} mod M. (Modular exponentiation.)

If you ever get ≠ 1, output "**M is composite**".

Otherwise, output, "**M is probably prime**".

However, this **is** the basis of the efficient Miller−Rabin primality algorithm.

It just adds a few more number-theoretic tweaks.

Given M:

Pick a few random A's between 1 and M−1.

For each, compute A^{M-1} mod M. (Modular exponentiation.)

If you ever get ≠ 1, output "**M is composite**".

Otherwise, output, "**M is probably prime**".

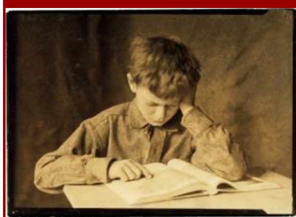
Finally:

Suppose you're trying to pick a **random** prime.

As Carmichael numbers are so rare, the above test works with **very high prob.** for random M.

In fact, just testing A = 2, 3 is (pretty much) good enough!

Study Guide



Arithmetic:
- +, ×, ÷, mod, GCD, modular exponent., primality, rand prime, **all efficient**

Algorithms to study:
- modular exponent., Euclid's Algorithm, miix-finding extension

Modular arithmetic:
- $(\mathbb{Z}_M, +)$, $(\mathbb{Z}^*_M, \cdot)$, $\varphi(M)$
- Euler's Theorem
- Fermat's Little Theorem