“Two is a most odd prime
because
Two is the least even prime.”

-- Dr. Dan Jurca

“That’s a big prime!”

Image by Matthew Harvey © 2003

A Grand Coding Challenge! Finding a new Largest Known Prime

The Great indoor sport of hunting for world record-sized prime numbers

Landon Curt Noll
prime-tutorial-mail@asthe.com
www.isthe.com/chongo

v1.85 — 2022 Apr 25

http://www.isthe.com/chongo/tech/math/prime/prime-tutorial.pdf

Agenda - Part 1 - Mersenne Primes

- Part 1.A & 1.B - 75 minutes (09:00 - 10:15)

- 2^2-1: What is a Prime Number?
- 2^3-1: 423+ Years of Largest known primes
- 2^5-1: Factoring vs. Primality Testing
- 2^7-1: Lucas-Lehmer Test for Mersenne Numbers
- $2^{13}-1$: The Mersenne Exponential Wall
- $2^{17}-1$: Pre-screening Lucas-Lehmer Test Candidates
- $2^{19}-1$: How Fast Can You Square?

- Part 1 Exercise and Quiz - 10 minutes (10:15 - 10:25)
- Discuss Part 1 Questions - 5 minutes (10:25 - 10:30)

Image Credit:
Flickr user forkergirl
Creative Commons License

Agenda - Break

- Break - 30 minutes (10:30 - 11:00)

Image Credit:
Flickr user Rajiv Patel (Rajiv's View)
Creative Commons License

Agenda - Part 2 - Large Riesel Primes Faster

- Part 2 - 75 minutes (11:00 - 12:15)

- $2^{31}-1$: Riesel Test: Searching sideways
- $2^{61}-1$: Pre-screening Riesel test candidates
- $2^{89}-1$: Multiply+Add in Linear Time
- $2^{127}-1$: Final Words and Some Encouragement
- $2^{521}-1$: Resources

- Part 2 Exercise and Quiz - 10 minutes (12:15 - 12:25)
- Discuss Part 2 Questions - 5 minutes (12:25 - 12:30)

- Optional Discussion / General Q&A - As needed (12:30- TBD)

Image Credit:
Flickr user anarchosyn
Creative Commons License

Part 1.A - Mersenne Primes

- 2^2-1: What is a Prime Number?
- 2^3-1: 423+ Years of Largest known primes
- 2^5-1: Factoring vs. Primality Testing
- 2^7-1: Lucas-Lehmer Test for Mersenne Numbers

King Henry VIII's armor
Image Credit:
wallyg Flickr user
Creative Commons License

Some Notation

- Common assumption in many number theory papers:
  - A variable is an integer unless otherwise stated
- $M(p) = 2^p-1$
  - p is often prime :-)

Image credit:
Flickr user fatllama
Creative Commons License

- The symbol ≡ means “identical to”
  - Think =
    - Difference between = and ≡ is important to mathematicians
    - The difference is not important to understand how to perform the test
- mod (short for modulus)
  - Think “divide and leave the remainder”
  - 5 mod 2 ≡ 1   14 mod 4 ≡ 2   21 mod 7 ≡ 0

2^2-1: What is a Prime Number?

- A natural number (1,2,3, …) is prime if and ONLY IF:
  - it has only 2 **distinct** natural number divisors
    - 1 and itself

- The first 25 primes:
  - 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
    - There are 25 primes < 100

- 6 is not prime because: 2 * 3 = 6
  - 1, 2, 3, and 6 are factors of 6 (i.e., 6 has 4 distinct natural number divisors)

Image Credit:
Flickr user amandabhslater
Creative Commons License

Image by
Landon Curt Noll © 2011

Why is 1 not prime?

- Almost nobody on record defined 1 as prime until Stevin in 1585
- From the mid 18th century to the start of the 20th century
  - There were many who called 1 a prime
- Today we commonly use definitions where 1 is not prime
- Fundamental theorem of arithmetic in commonly use today does not assume that 1 is prime
  - Any natural number can be expressed as a unique (ignoring order) product of primes
  - 1400 = 2 * 2 * 2 * 5 * 5 * 7
    - No other product of primes = 1400
  - If 1 were prime:
    - 1400 = 2 * 2 * 2 * 5 * 5 * 7 * 1
    - 1400 = 2 * 2 * 2 * 5 * 5 * 7 * 1 * 1* …
- Q: What is a "mathematical **definition**"? A: The pragmatic answer:
  - .. something that the mathematical community agrees upon
- Q: What is a "mathematical **truth**"? A: The pragmatic answer:
  - .. something that the mathematical community has studied and has been demonstrated to be true

What is the Largest Known Prime: $2^{82589933}-1$

- 24 862 048 decimal digits
  - 4973 pages (100 lines, 50 digits per line)
  - https://lcn2.github.io/mersenne-english-name/m82589933/prime-c.html
    - 1,488,944,457,420,413,
    - 255,478,064,584,723,979,166,030,262,739,927,953,241,852,712,894,252,132,393,
      - … 436 173 lines skipped here …
    - 557,947,958,297,531,595,208,807,192,693,676,521,782,184,472,526,640,076,912,
    - 114,355,308,311,969,487,633,766,457,823,695,074,037,951,210,325,217,902,591

Image by Matthew Harvey
© 2003

- The English name for this prime is over 656 megabytes long:
  - Double sided printing, 100 lines per side, requires over 82 reams (500 sheet per ream) of paper!
  - https://lcn2.github.io/mersenne-english-name/m82589933/prime.html
    - one octomilliamilliaduocenseptenoctoginmilliatrecenoctoquadragintillion,
    - four hundred eighty eight octomilliamilliaduocenseptenoctoginmilliatrecenseptenquadragintillion,
    - nine hundred forty four octomilliamilliaduocenseptenoctoginmilliatrecensexquadragintillion,
      - … 8 280 068 lines skipped here …
    - two hundred seventeen million,
    - nine hundred two thousand,
    - five hundred ninety one

There is No Largest Prime -
The Largest Known Prime Record can always be Broken!

- Assume there are finitely many primes (and 1 is not a prime)
- Let A be the product of “all primes”
- Let p be a prime that divides A+1
- Since p divides A
  - Because A is the product of “all primes”
- And since p divides A+1
- Therefore p must divide 1
  - Which is impossible
- Which contradicts our original assumption

Image Credit:
Flickr user jurvetson
Creative Commons License

What is a Mersenne Prime?

- Mersenne number: 2^n-1
  - Examples: 2^3-1 $2^{11}-1$ $2^{67}-1$ $2^{23209}-1$
- A Mersenne prime is a mersenne number that is prime
  - Examples: 2^3-1 $2^{23209}-1$
- Why the name Mersenne?
  - Marin Mersenne: A 17th century french monk
  - Mathematician, Philosopher, Musical Theorist
  - Claimed when p = 2, 3, 5, 7, 13, 17, 19, 31, 67, 127, 257 then 2^p-1 was prime
    - $2^{61}-1$ proven prime in 1883 - was Mersenne's 67 was a typo of 61?
    - $2^{67}-1 = 761838257287 \times 193707721$ in 1903 - Still a typo?

      3 years of Saturdays for Cole to factor by hand:147573952589676412927
    - $2^{89}-1$ proven prime in 1911 - OK he missed one - 2nd strike
    - $2^{107}-1$ proven prime in 1914 - 3rd strike - Forget it!
  - After more than 300 years his name stuck

Image Credit: Wikipedia

2^3-1: 423+ Years of Largest Known Primes

- Earliest explicit study of primes: Greeks (around 300 BCE)

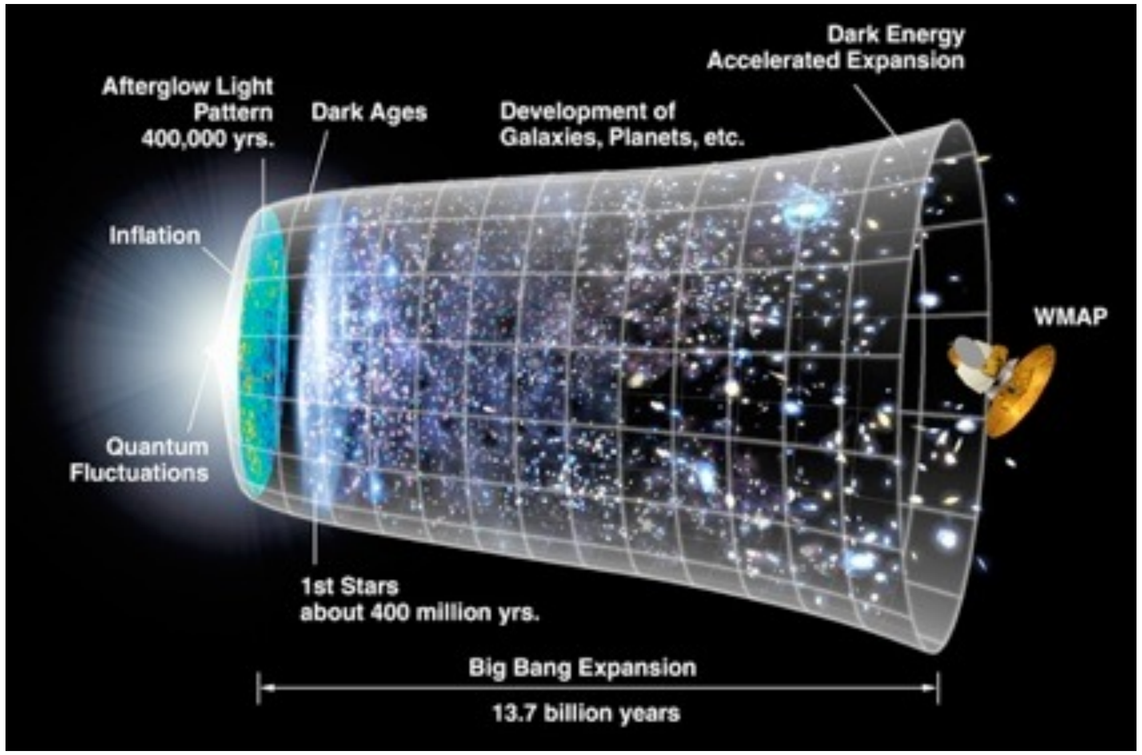


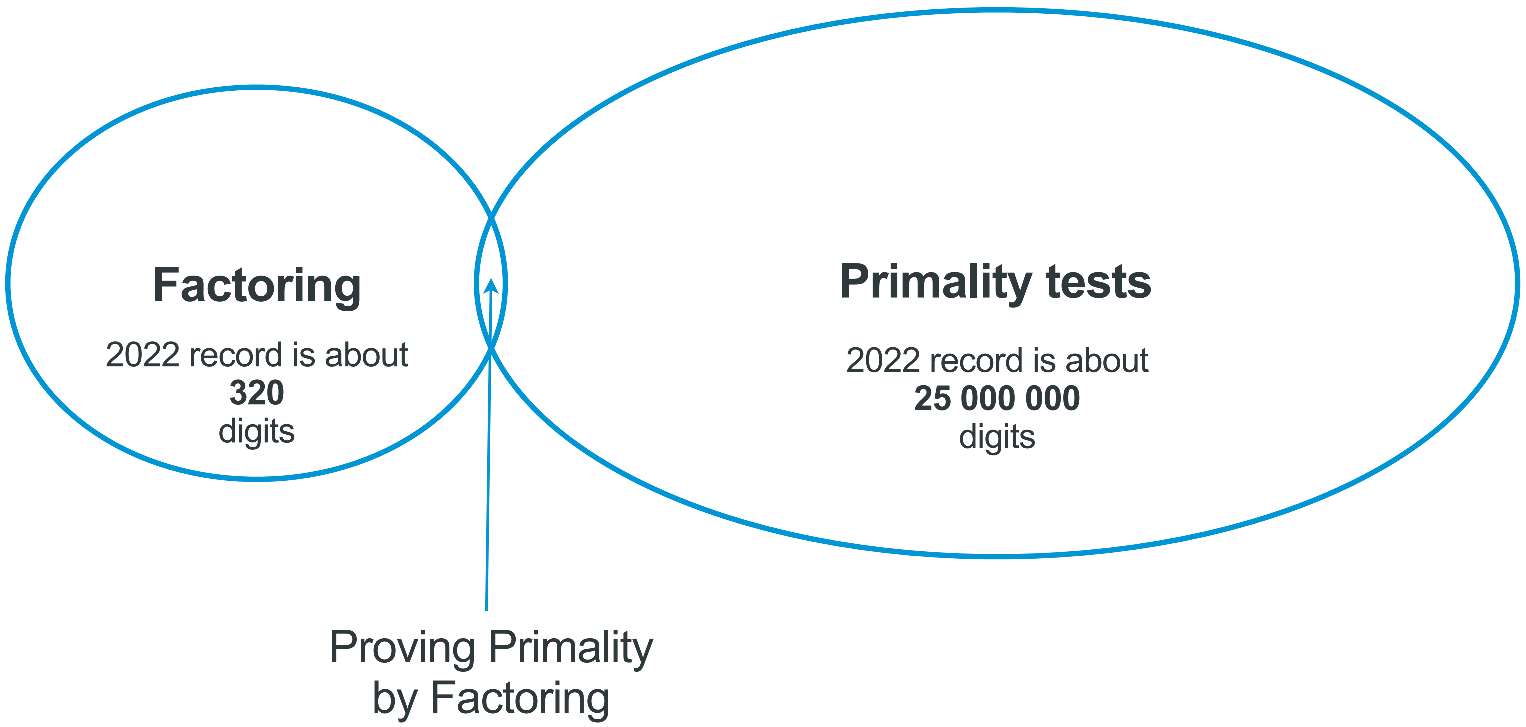
Image Credit:
Wikipedia
Creative Commons License

- 1588: First published largest known primes
  - Cataldi proved 131701 (2^{17}-1) & 524287 (2^{19}-1) were prime
  - Produced an complete table of primes up to 743
    - Made an exhaustive factor search of 2^{17}-1 & 2^{19}-1

      By hand, using roman numerals!
    - Held the record for more than 2 centuries!
- 1772: Euler proved 2^{31}-1 (2147483647) was prime
  - A clever proof to eliminate almost all potential factors, trial division for the rest
  - Euler said: "2^{31}-1 is probably the greatest (prime) that ever will be discovered … it is not likely that any person will attempt to find one beyond it."
- 1867: Landry completely factored 2^{59}-1 = 179951 * 3203431780337
  - 3203431780337 was the largest known prime by the fundamental theorem of arithmetic
    - By trial division after eliminating almost all potential factors

2^5-1: Factoring vs. Primality testing

- Factoring and Prime testing methods overlap only in the trivial case:



- Useful to test numbers with only a “handful of digits”

- 1951: Ferrier factored $2^{148}+1$ and proved that $(2^{148}+1)/17$ was prime
  - Using a desk calculator after eliminating most factor candidates
  - Largest record prime, 44 digits, discovered without the use of a digital computer

- Largest “general” number factored in 2012 had only 320 digits
  - Primes larger than 320 digits were discovered in 1952

1st Prime Records without Factoring, by Hand

- 1876: Édouard Lucas proved 2^{127}-1 was prime
  - 170141183460469231731687303715884105727
  - Édouard Lucas made significant contributions to our understanding of Fibonacci-like Lucas sequences
    - Lucas sequences are the heart of the Lucas-Lehmer test for Mersenne Primes

- Lucas proved that 2^{127}-1 had a property that only possible when 2^P-1 was prime

Pseudo-primality Tests

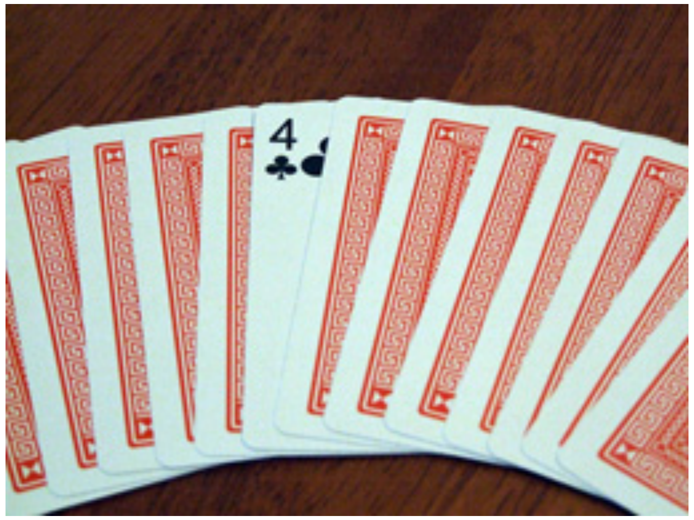


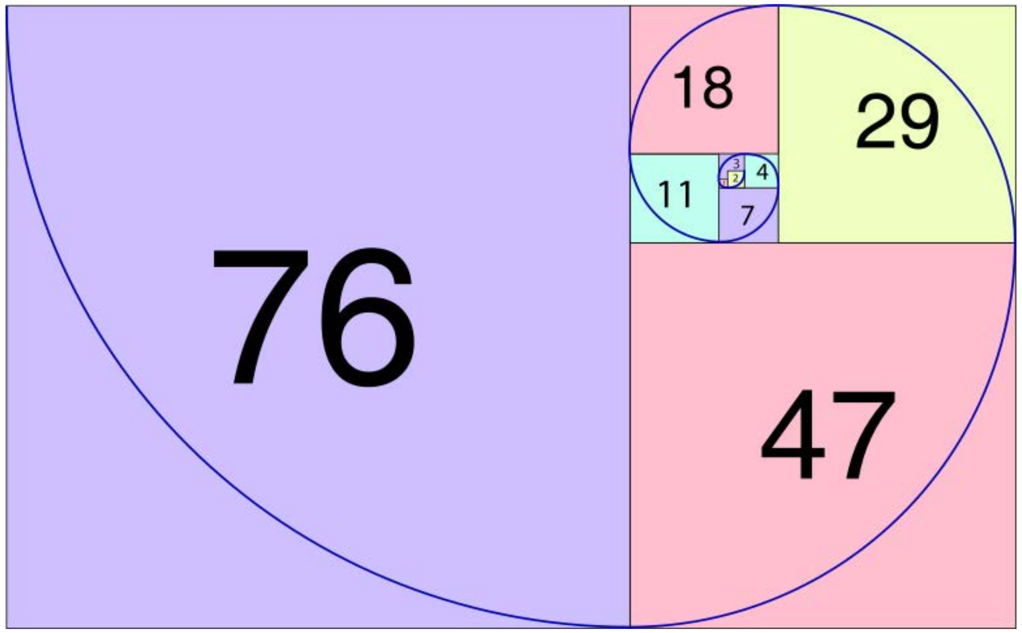
Image Credit:
Flickr user SpacePotato
Creative Commons License

- Some mathematical tests are true when a number is prime
- A pseudo primality test
  - A property that every prime number must pass … however some non-primes also pass
- Fermat pseudoprime test
  - If p is an odd prime, and a does not divide p, then $a^{(p-1)}-1$ is divisible by p
    - Let: p = 23 and a = 2 which is not a factor of 23, then $2^{22}-1$ = 4194303 and 23 * 182361 = 4194303
  - However 341 also passes the test
    - for a = 2: $2^{340}-1$ is divisible by 341 but 341 = 11 * 31
- Passing a Pseudoprime test does NOT PROVE that a number is prime!
  - Failing a Pseudoprime test only proves that a number is not prime
- There are an infinite number of Fermat pseudoprimes
  - There are an infinite number of Fermat pseudoprimes that pass for every allowed value of “a”
    - These are called Carmichael numbers

Lucas Sequences

- For a given P & Q
  - $U_0 = 0 \quad U_1 = 1 \quad U_n = P*U_{n-1} - Q*U_{n-2}$ for n > 1
  - $V_0 = 2 \quad V_1 = P \quad V_n = P*V_{n-1} - Q*V_{n-2}$ for n > 1
- Fibonacci Sequence - Lucas Sequence special case
  - $P = 1 \quad Q = -1 \quad U_n = P*U_{n-1} - Q*U_{n-2}$
  - $U_0 = 1 \quad U_1 = 1 \quad U_n = U_{n-1} + U_{n-2}$
  - 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, …
- Lucas Numbers - Useful for primality testing
  - $P = 1 \quad Q = -1 \quad V_n = P*V_{n-1} - Q*V_{n-2}$
  - $V_0 = 2 \quad V_1 = 1 \quad V_n = V_{n-1} + V_{n-2}$
  - 2, 1, 3, 4, 7, 11, 18, 29, 47, 76, 123, 199,

Image Credit: Wikipedia



Lucas Pseudo-primes

- If n is prime, then V_n mod n = 1
  - 2, 1, 3, 4, 7, 11, 18, 29, 47, 76, 123, 199, …
- However, V_n mod n = 1 for some n that are not prime:
- V_{705} % 705 = 1
- V_{2465} % 2465 = 1
- V_{2737} % 2737 = 1
- V_{3745} % 3745 = 1
- V_{4181} % 4181 = 1
- V_{5777} % 5777 = 1
- V_{6721} % 6721 = 1

V(n)	n	V(n) % n
2	0	
1	1	
3	**2**	1
4	**3**	1
7	4	3
11	**5**	1
18	6	0
29	**7**	1
47	8	7
76	9	4
123	10	3
199	**11**	1
322	12	10
521	**13**	1
843	14	3
1364	15	14
2207	16	15
3571	**17**	1
5778	18	0
9349	**19**	1
15127	20	7
24476	21	11
39603	22	3
64079	**23**	1
103682	24	2
167761	25	11
271443	26	3
439204	27	22
710647	28	7
1149851	**29**	1
1860498	30	18
3010349	**31**	1

Jumping ahead in the Lucas Sequence

- $V_n = V_{n-1} + V_{n-2}$
- $V_{2n} = V_n^{\,2} - 2$
- V_{2n} grows to be huge!

n	V(2^n)	2^n
1	3	2
2	7	4
3	47	8
4	2207	16
5	4870847	32
6	23725150497407	64
7	562882766124611619513723647	128
8	316837008400094222150776738 483768236006420971486980607	256
9	100385689891921376688754239 992826256704879627683181901 515099398613465618884806971 304035121947368905594088447	512
10	100772867350770056609820080 610650730680744753004660124 446293884875747696521156517 635000261283676793017447903 659202787756017660002174559 979308098751086395045787668 536036255051626821777084330 23235042368022152858871807	1024

2^7-1: Lucas-Lehmer Test for Mersenne Numbers

- Some primality tests are definitive
- In 1930, Dr. D. H. Lehmer extended Lucas's work
  - This test was the subject of Dr. Lehmer's Thesis
- Known as a Lucas-Lehmer test
  - A definitive primality test
- The most efficient proof of primality known
  - Work to prove primality vs. size of the number tested
    - Theoretical argument suggests test may be the most efficient possible
- It was my honor and pleasure to study under Dr. Lehmer
  - One of the greatest computational mathematicians of our time
    - Like prime numbers, there will always be greater mathematicians :)
  - Was willing to teach math to a couple of high school kids like me

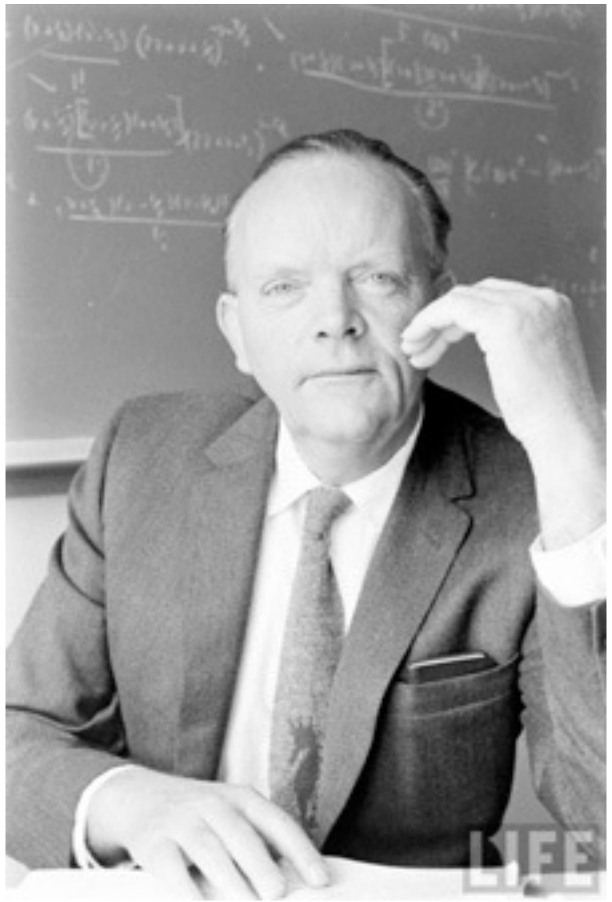


Image Credit: Time-Life Magazine

Lucas Sequence for 2^n-1

- $S_2 = 4$
- $S_{n+1} = S_n{}^2 - 2$
- If p is odd prime,

  then for $m = 2^p-1$,
  if and only if $S_m \bmod m = 0$,
  then m is prime!
- You don’t need the exact value of S_m only S_m mod m

n	U(2^n - 1)	2^n - 1	U(2^n - 1) % 2^n-1
1		1	
2	4	3	1
3	14	7	0
4	194	15	14
5	37634	31	0
6	1416317954	63	23
7	2005956546822746114	127	0
8	402386166774103602282563565610210099 4	255	0
9	16191462721115671781777559070120513664958590125499158514329308740975788034	511	0
10	262163465049278514526059369557563039213647877559524545911906005349555773831236935015956281848933426999307982418664943276943901608919396607297585154	1023	0

Lucas-Lehmer test *

- $M(p) = 2^p-1$ is prime IF AND ONLY IF p is odd prime and $U_p \equiv 0 \bmod (2^p-1)$
  - Where $U_2 = 4$
  - and $U_{x+1} \equiv (U_x^2 - 2) \bmod (2^p-1)$

Image Credit: Wikipedia
Creative Commons License

* This is Landon Noll's preferred version of the test:
others let $U_1=4$ and test for $U_{(p-1)} \equiv 0 \bmod 2^p-1$,
and still others let $U_2=4$ and test for $U_{(p-1)} \equiv 0 \bmod 2^p-1$

Lucas-Lehmer Test - Mersenne Prime Test

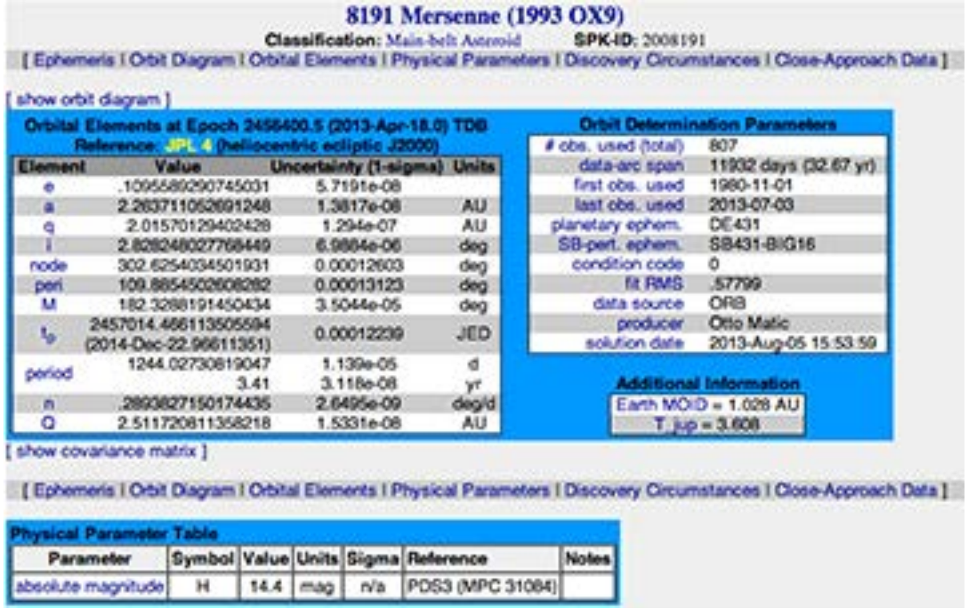
- Mersenne prime test for $M(p) = 2^p - 1$ where p is an odd prime

- Let $U_2 = 4$

- Repeat until U_p is calculated: $U_{i+1} \equiv (U_x^2 - 2) \bmod (2^p - 1)$
  - Square the previous U_i term
  - Subtract 2
  - mod (2^p-1) (divide by 2^p-1 and take the remainder)

- Does the final $U_p \equiv 0$???
  - Yes - $M(p) = 2^p-1$ is prime
  - No - $M(p) = 2^p-1$ is not prime

Minor Planet 8191
is named after Mersenne
$8191 = 2^{13}-1$



8191 Mersenne (1993 OX9)

Classification: Main-belt Asteroid SPK-ID: 2008191

[Ephemeris | Orbit Diagram | Orbital Elements | Physical Parameters | Discovery Circumstances | Close-Approach Data]

[show orbit diagram]

Orbital Elements at Epoch 2456400.5 (2013-Apr-18.0) TDB
Reference: JPL 4 (heliocentric ecliptic J2000)

Element	Value	Uncertainty (1-sigma)	Units
e	.1095580290745031	5.7191e-08	
a	2.263711052691248	1.3817e-08	AU
q	2.01570129402428	1.294e-07	AU
i	2.828248027768449	6.9884e-06	deg
node	302.6254034501931	0.00012603	deg
peri	109.8854502608282	0.00013123	deg
M	182.3288191450434	3.5044e-05	deg
t_p	2457014.466113505594 (2014-Dec-22.96611351)	0.00012239	JED
period	1244.02730819047	1.139e-05	d
	3.41	3.118e-08	yr
n	.2893827150174435	2.6495e-09	deg/d
Q	2.511720811358218	1.5331e-08	AU

Orbit Determination Parameters

# obs. used (total)	807
data-arc span	11932 days (32.67 yr)
first obs. used	1980-11-01
last obs. used	2013-07-03
planetary ephem.	DE431
SB-pert. ephem.	SB431-BIG16
condition code	0
fit RMS	.57799
data source	ORB
producer	Otto Matic
solution date	2013-Aug-05 15:53:59

Additional Information

Earth MOID = 1.028 AU
T_jup = 3.608

[show covariance matrix]

[Ephemeris | Orbit Diagram | Orbital Elements | Physical Parameters | Discovery Circumstances | Close-Approach Data]

Physical Parameter Table

Parameter	Symbol	Value	Units	Sigma	Reference	Notes
absolute magnitude	H	14.4	mag	n/a	PDS3 (MPC 31084)	

Lucas-Lehmer Test Example.0

Image Credit:

Flickr user duegnazio
Creative Commons License

- Is $M(5) = 2^5-1 = 31$ prime?
- 5 is odd prime so according to the Lucas-Lehmer test:
  - 2^5-1 prime if and only if $U_5 \equiv 0 \bmod 31$
  - where $U_2 = 4$ and $U_{x+1} \equiv U_x^2 - 2 \bmod 31$
- $U_2 =$ 4 (by definition)

Lucas-Lehmer Test Example.1

Image Credit:

Flickr user duegnazio
Creative Commons License

- Is $M(5) = 2^5-1 = 31$ prime?
- 5 is prime so according to the Lucas-Lehmer test:
  - 2^5-1 prime if and only if $U_5 \equiv 0 \bmod 31$
  - where $U_2 = 4$ and $U_{x+1} \equiv U_x^2 - 2 \bmod 31$
- $U_2 = 4$ (by definition)
- $U_3 = 4^2 - 2 =$

Lucas-Lehmer Test Example.2

- Is $M(5) = 2^5-1 = 31$ prime?

- 5 is prime so according to the Lucas-Lehmer test:
  - 2^5-1 prime if and only if $U_5 \equiv 0 \bmod 31$
  - where $U_2 = 4$ and $U_{x+1} \equiv U_x^2 - 2 \bmod 31$

Image Credit:

Flickr user duegnazio
Creative Commons License

- $U_2 = \quad 4 \qquad$ (by definition)
- $U_3 = \quad 4^2 - 2 = \quad 14 \bmod 31 \equiv$

Lucas-Lehmer Test Example.3

Image Credit:

Flickr user duegnazio
Creative Commons License

- Is $M(5) = 2^5-1 = 31$ prime?

- 5 is prime so according to the Lucas-Lehmer test:
  - 2^5-1 prime if and only if $U_5 \equiv 0 \bmod 31$
  - where $U_2 = 4$ and $U_{x+1} \equiv U_x^2 - 2 \bmod 31$

- $U_2 = 4$ (by definition)
- $U_3 = 4^2 - 2 = 14 \bmod 31 \equiv 14$
- $U_4 = 14^2 - 2 =$

Lucas-Lehmer Test Example.4

Image Credit:

Flickr user duegnazio
Creative Commons License

- Is $M(5) = 2^5-1 = 31$ prime?

- 5 is prime so according to the Lucas-Lehmer test:
  - 2^5-1 prime if and only if $U_5 \equiv 0 \bmod 31$
  - where $U_2 = 4$ and $U_{x+1} \equiv U_x^2 - 2 \bmod 31$

- $U_2 = \quad 4 \qquad$ (by definition)
- $U_3 = \quad 4^2 - 2 = \quad 14 \bmod 31 \equiv 14$
- $U_4 = 14^2 - 2 = 194 \bmod 31 \equiv$

Lucas-Lehmer Test Example.5

- Is $M(5) = 2^5-1 = 31$ prime?

- 5 is prime so according to the Lucas-Lehmer test:
  - 2^5-1 prime if and only if $U_5 \equiv 0 \bmod 31$
  - where $U_2 = 4$ and $U_{x+1} \equiv U_x^2 - 2 \bmod 31$

Image Credit:

Flickr user duegnazio
Creative Commons License

- $U_2 = 4$ (by definition)
- $U_3 = 4^2 - 2 = 14 \bmod 31 \equiv 14$
- $U_4 = 14^2 - 2 = 194 \bmod 31 \equiv 8$
- $U_5 = 8^2 - 2 =$

Lucas-Lehmer Test Example.6

- Is $M(5) = 2^5-1 = 31$ prime?

- 5 is prime so according to the Lucas-Lehmer test:
  - 2^5-1 prime if and only if $U_5 \equiv 0 \bmod 31$
  - where $U_2 = 4$ and $U_{x+1} \equiv U_x^2 - 2 \bmod 31$

Image Credit:

Flickr user duegnazio
Creative Commons License

- $U_2 = 4$ (by definition)
- $U_3 = 4^2 - 2 = 14 \bmod 31 \equiv 14$
- $U_4 = 14^2 - 2 = 194 \bmod 31 \equiv 8$
- $U_5 = 8^2 - 2 = 62 \bmod 31 \equiv$

Lucas-Lehmer Test Example.7

- Is $M(5) = 2^5-1 = 31$ prime?

- 5 is prime so according to the Lucas-Lehmer test:
  - 2^5-1 prime if and only if $U_5 \equiv 0 \bmod 31$
  - where $U_2 = 4$ and $U_{x+1} \equiv U_x^2 - 2 \bmod 31$

Image Credit:

Flickr user duegnazio
Creative Commons License

- $U_2 = 4$ (by definition)
- $U_3 = 4^2 - 2 = 14 \bmod 31 \equiv 14$
- $U_4 = 14^2 - 2 = 194 \bmod 31 \equiv 8$
- $U_5 = 8^2 - 2 = 62 \bmod 31 \equiv 0$

- Because $U_5 \equiv 0 \bmod 31$ we know that 31 is prime

Lucas-Lehmer Test Example II

Image Credit:

Flickr user duegnazio
Creative Commons License

- Is $M(11) = 2^{11}-1 = 2047$ prime?

- 11 is prime so according to the Lucas-Lehmer test:
  - $2^{11}-1$ prime if and only if $U_{11} \equiv 0 \bmod 2^{11}-1$

- Calculating U_{11}
  - $U_2 = 4$ (by definition)
  - $U_3 = 4^2 - 2 = 14 \bmod 2047 \equiv 14$
  - $U_4 = 14^2 - 2 = 194 \bmod 2047 \equiv 194$
  - $U_5 = 194^2 - 2 = 37634 \bmod 2047 \equiv 788$
  - $U_6 = 788^2 - 2 = 620942 \bmod 2047 \equiv 701$
  - $U_7 = 701^2 - 2 = 491399 \bmod 2047 \equiv 119$
  - $U_8 = 119^2 - 2 = 14159 \bmod 2047 \equiv 1877$
  - $U_9 = 1877^2 - 2 = 3523127 \bmod 2047 \equiv 240$
  - $U_{10} = 240^2 - 2 = 57598 \bmod 2047 \equiv 282$
  - $U_{11} = 282^2 - 2 = 79522 \bmod 2047 \equiv 1736$ <<== not 0 therefore 2047 is not prime (23 * 89 = 2047)

Primality Testing in the Age of Digital Computers

- 1951: Miller and Wheeler proved $180*(2^{127}-1)^2 + 1$ prime using EDSAC1
  - 5210644015679228794060694325390955853335898483908056458352183851018372555735221
  - A 79 digit prime
  - Using a specialized proof of primality

- 1952: Robison and Lehmer using the SWAC using the Lucas-Lehmer test
  - 1952 Jan 30 $2^{521}-1$ is prime
  - 1952 Jan 30 $2^{607}-1$ is prime
  - 1952 June 25 $2^{1279}-1$ is prime
  - 1952 Oct 7 $2^{2203}-1$ is prime
  - 1952 Oct 9 $2^{2281}-1$ is prime

Image Credit:
Flickr user skreuzer
Creative Commons License

- Robison coded the SWAC over the 1951 Christmas holiday
  - By hand writing down the machine code as digits using only the SWAC manual
  - Was Robison's first computer program he ever wrote
  - Ran successfully the very first time!

Mersenne Prime Exponents must be Prime

- If $M(p) = 2^p-1$ is prime, then p must be prime
- If x is not prime, then $M(x) = 2^x-1$ is **not** prime
- Look at $M(9)$ 2^9-1 in binary
  - 111111111
- We can rewrite M(9) as this product:
  - 1001001
    x 111

    111111111

Landon Curt Noll and the
Palomar 200-inch telescope

- If x = y * z
  - then M(x) has M(y) and M(z) as factors AND therefore M(x) cannot be prime

Record Primes 1957 - 1961

- 1957: M(3217) 969 digits Riesel using BESK
- 1961: M(4423) 1332 digits Hurwitz & Selfridge using IBM 7090
  - The M(4423) was proven the prime same evening M(4253) was proven prime
  - Hurwitz noticed M(4423) before M(4253) because the way the output was stacked
  - Selfridge asked:
    - “Does a machine result need to be observed by a human before it can be said to be discovered?”
  - Hurwitz responded:
    - “… what if the computer operator who piled up the output looked?”
  - Landon believes the answer to Selfridge’s question is yes
  - Landon speculates that even if the computer operator looked, they very likely did not understand the meaning of the output:
    - Therefore Landon (and many others) believe M(4253) was never the largest known prime

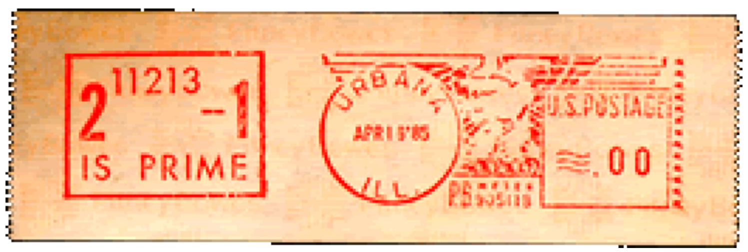
John Selfridge (1927 - 2010)

Image Credit:
Department of Computer Science
UIUC

Record Primes at UIUC: 1963

- 1963: M(9668)   2917 digits   Donald B. Gillies using the ILLIAC 2
- 1963: M(9941)   2993 digits   Donald B. Gillies using the ILLIAC 2
- 1963: M(11213)   3376 digits   Donald B. Gillies using the ILLIAC 2

Image Credit:
Chris K. Caldwell



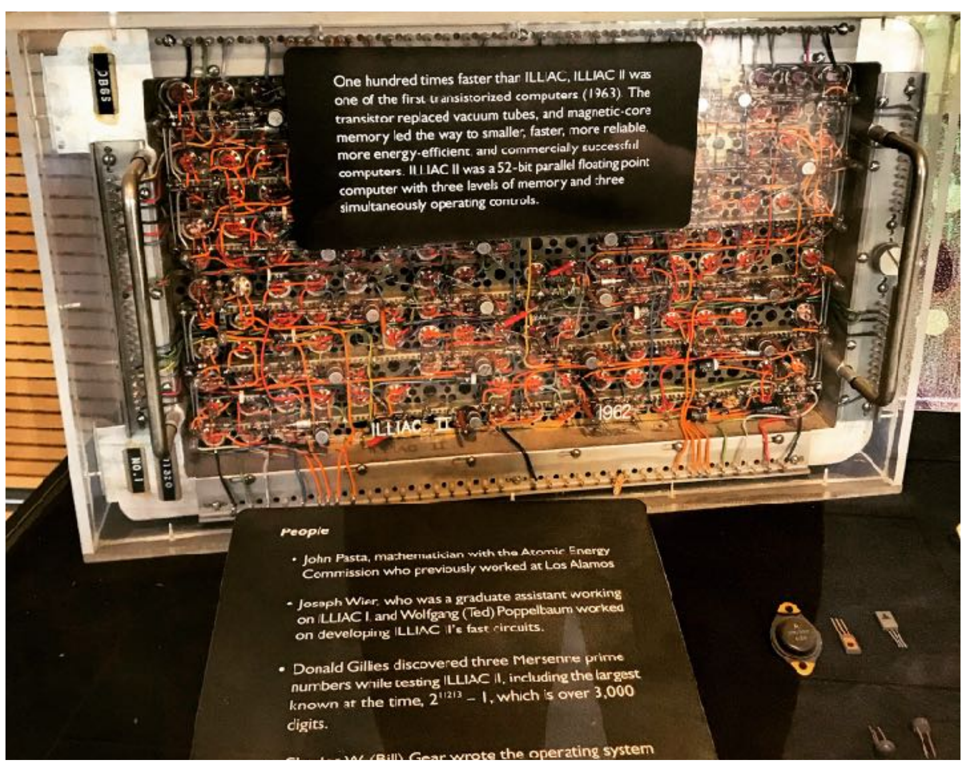


Image Credit:
Landon Noll

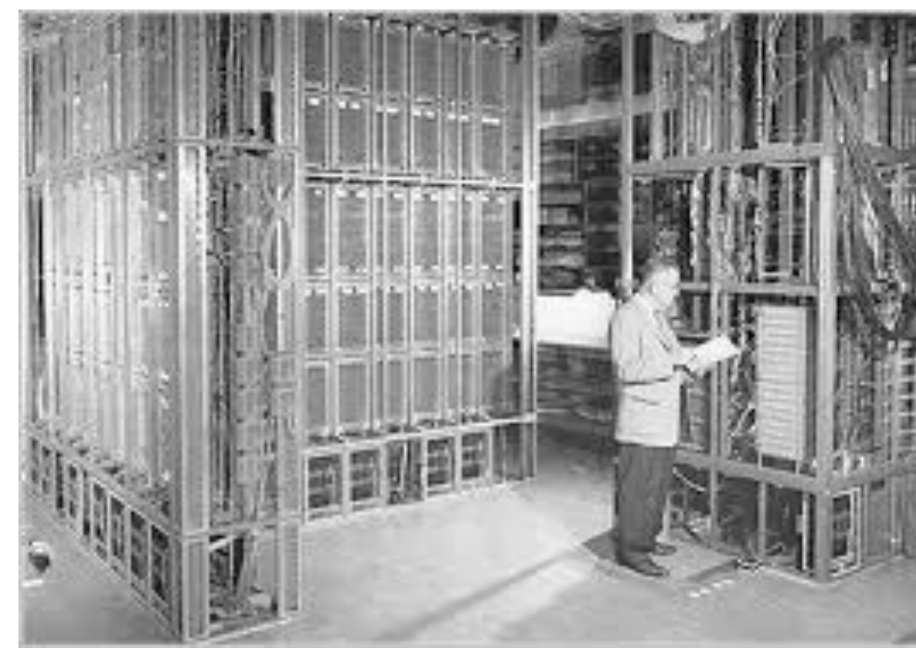


Image Credits:
Department of Computer Science
UIUC

- Largest known prime until:
- 1971: M(19937)   6002 digits   Tuckerman using the IBM 360/91

Landon’s Record Primes: 1978 - 1979

- 1978: M(21701) 6533 digits Noll & Nickel using the CDC Cyber 174
- 1979: M(23209) 6987 digits Noll using the CDC Cyber 174

Image Credit:
Paul Noll

Landon 367 days before
discovering M(21701)
Green Cake Reads:
“CHONGO 2^{19937}-1 is prime”

Image Credit:
Landon Curt Noll

- 1st working version of the code took 500+ hours to test M(21001) on 1 April 1977
- The 1 Oct 1978 version took 7 hours, 40 minutes and 20 seconds to test M(21701)
  - Proven prime on 1978 Oct 30
- Searched M(21001) thru M(24499) using 6000+ CPU Hours on Cyber 174
  - Used the facility account and much encouragement from Dr. Dan Jurca

Cray Record Primes

- 1979: M(44497) 13 395 digits Nelson & Slowinski using the Cray 1
- 1982: M(86243) 25 962 digits Slowinski using the Cray 1
- 1983: M(132049) 39 751 digits Slowinski using the Cray X-MP
- 1985: M(216091) 65 050 digits Slowinski using the Cray X-MP/24

Nelson & Slowinski
Discovered M(44497)

Image Credit:
Chris Caldwell

Image credit: Wikipedia
Creative Commons License

Part 1.B - Mersenne Prime Search

- $2^{13}-1$: The Mersenne Exponential Wall
- $2^{17}-1$: Pre-screening Lucas-Lehmer Test Candidates
- $2^{19}-1$: How Fast Can You Square?

Image Credit:
Daniel Gasienica
Flickr user
Creative Commons License

$2^{13}-1$: The Mersenne Exponential Wall

- The Lucas-Lehmer Test for M(p) requires computing p-1 terms of U_i :
  - $U_{i+1} \equiv U_i^2 - 2 \bmod 2^p-1$
- That is p-1 times performing ...
  - Sub-step 1: square a number
  - Sub-step 2: subtract 2
  - Sub-step 3: mod 2^p-1
- … on numbers between 0 and 2^p-2
  - On average numbers that are p bits long
    - or 2p bits when dealing with the result of the square

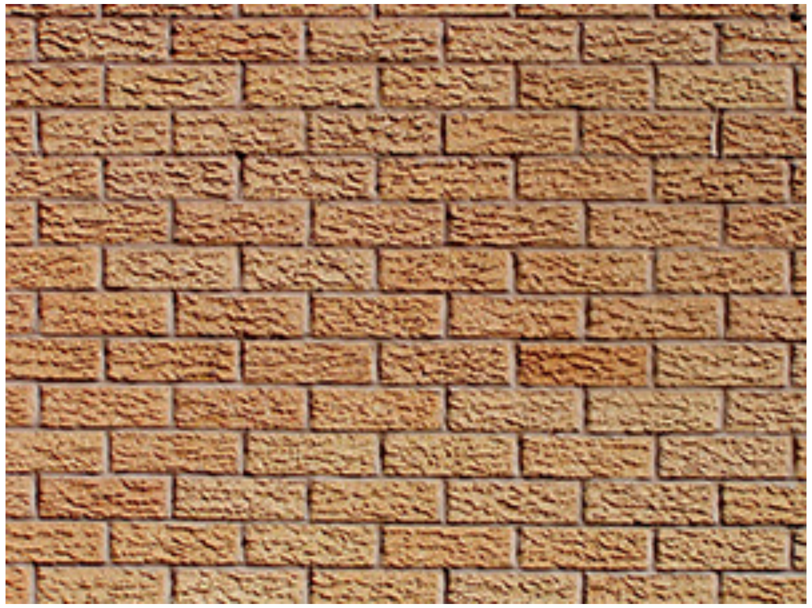


Image Credit:
Flickr user pigpogm
Creative Commons License

Sub-step 1: Square

- Consider this classical multiply:

			1	2	3	← 3 x 3 digit multiply
x			4	5	6	
			6	12	18	
		5	10	15		← 9 products
+	4	8	12		.	
	4	13	28	27	18	← 5 adds
carry	1	2	2	1		
	5	5	10	8	8	← 5 carry adds
carry		1				
	5	6	0	8	8	

- On the average a d x d digit multiply requires $O(d^2)$ operations:
  - Products: d^2
  - Adds: d^2

Sub-step 2: subtract 2

- This step is trivial

- On average requires 1 subtraction
  - O(1) steps

Child's coffee cup © Yogi
Permission to use with attribution
http://www.flickr.com/photos/yogi/163796078/

Sub-step 3: mod 2^p-1 by Shift and Add

- It turns out that this is easy too!
  - Just a shift and add!

- Split U_i^2-2 into two chunks and make low order chunk p bits long:

$$U_i^2-2 = \boxed{\quad J \quad}\boxed{\quad K \quad}$$

← p bits long →

- Then U_i^2-2 mod 2^p-1 ≡ J + K

- If J + K > 2^p-1 then split again
  - In this case the upper chunk will be 1, so just add 1 to the lower chunk

- So mod 2^p-1 can be done in O(d) steps

Sub-step 3: mod 2^p-1 - An Example

- Split L into two chunks and make low order chunk p bits long:

U_x^2-2 =

J	K
	← p bits long →

- For p=31, U_{22} = 1992425718

- U_{22}^2-2 = 3969760241747815522 =
  - 11011100010111011010111110100000111100110110001110110001100010

    ← 31 bits long →

- J = 1101110001011101101011111010000

  K = 0111100110110001110110001100010

  J+K = 10101011000001111100010000110010

- Now J + K > 2^{31}-1 so peel off the upper 1 bit and add it into the bottom

- 0101011000001111100010000110010

  1

  0101011000001111100010000110011 = 721929267

- U_{23} = U_{22}^2-2 mod 2^{31}-1 = 721929267

So Computing U(x)

- Sub-step 1: square requires $O(p^2)$ operations
- Sub-step 2: subtract requires O(1) operation
- Sub-step 3: mod 2^p-1 requires O(p) operations
- The time to square dominates over the time subtract and mod
- Computing U_i requires $O(p^2)$ operations
- We have to compute p-1 terms of U_i to test 2^p-1
- The prime test is $O(p^3)$ operations

Image Credit: Wikipedia
Creative Commons License

$O(p^3)$ doesn't Scale Nicely as P Grows

- If it takes a computer 1 day to test M(p)
- 8 days to test M(2*p)
- 4 months to test M(5*p)
- 2.7 years to test M(10*p)
- etc. !!!

Image Credit:
Flickr user sylvia@intrigue
Creative Commons License
Note that weight is in Kg

2^{17}-1: Pre-screening Lucas-Lehmer Test Candidates

- Performing the Lucas-Lehmer test on M(p) is time consuming
  - Even if it is very a very efficient definitive test given the size of the number testing

- Try to pre-screen potential candidates by looking for tiny factors
  - If you find a small factor of M(p) then there is no need to test

- It can be proven that a factor q of M(p) must be of this form:
  - $q \equiv 1 \bmod 8$ or $q \equiv 7 \bmod 8$
  - $q = 2*k*p+1$ for some integer $k > 1$

- Factor candidates of M(p) are either 4*p or 2*p apart
  - When p is large, you can skip over a lot of potential factors of M(p)

Image Credit:
Flickr user {platinum}
Creative Commons License

Pre-screen Factoring Rule of Thumb

- For a given set of Mersenne candidates: M(a), M(b), … M(z)
  - Where z is not much bigger than a (say a < z < a*1.1)
  - Start factoring candidates until the rate of finding factors is slower than the Lucas-Lehmer test for the M(z)

- Typically this rule of thumb will eliminate 50% of the candidates

Image Credit:
Flickr user raindog
Creative Commons License

2^{19}-1: How Fast Can You Square?

- The time to square dominates the subtract and mod
  - So Mersenne Prime testing comes down to how fast can you square

Image Credit:
Laurie Sefton
Used by permission

Classical Square Slightly Faster Than Multiply

- Because the digits are the same on both, we can cut multiplies in half:

			3	4	5	6
x			3	4	5	6
			18	24	30	36
		15	20	25	30	
	12	16	20	24		
9	12	15	18			

4 x 4 digit multiply

10 products
6 of which are doubled by shifting

- On the average a d x d digit square requires $O(d^2)$ operations:
  - Products: $d^2/2$
  - Shifts: $d^2/2$ (shifts are faster than products)
  - Adds: d^2

Reduce Digits by Increasing Base

- No need to multiply base 10

- If a computer can ...
  - Multiply two B bit words produce a 2*B product
  - Divide 2*B bit double word by B bit divisor and produce B bit dividend & remainder
  - Add or Subtract B bit words and produce a B bit sum or difference

- … then represent your digits in base 2^B
  - Each B bit word will be a digit in base 2^B

- Test M(p) requires p bit squares or p/B word squares

- Classical square requires $O((p/B)^2)$ operations
  - The work still grows by the square of the digits $O(d^2)$

Image Credit:
Laurie Sefton
Used by permission

Squaring by Transforms

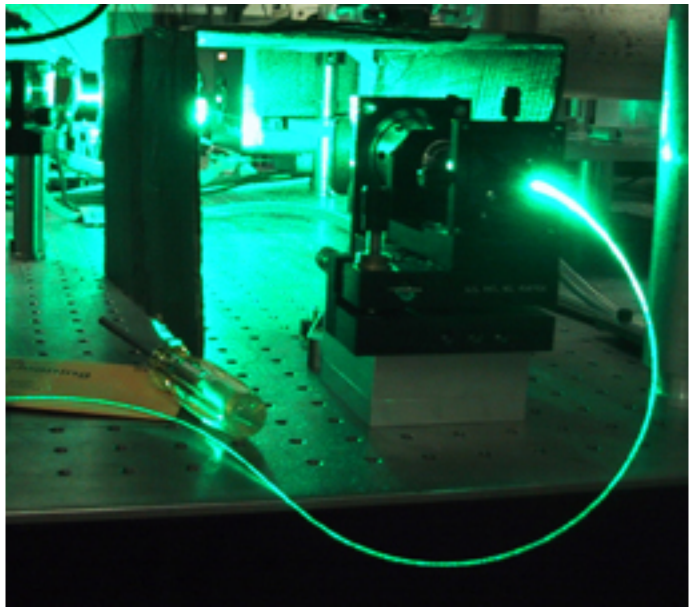
- Convolution Theorem states:
  - The Transform of the ordinary product equals dot product of the Transforms
  - T(x*y) = T(x) • T(y)
    - T(foo) is the transform of foo

- While ordinary product is $O(p^2)$ the dot product is O(p) !!!
  - Dot product: a[0]*b[0] + a[1]*b[1] + a[2]*b[2] + …. + a[max]*b[max]

- Multiplication by transform:
  - x*y = TINV(T(x) • T(y))
  - TINV(foo) is the inverse transform of foo

- A Square by Transform can approach O(d ln d)
  - ln d is natural log of d
  - Scales much much better than $O(d^2)$

Image Credit:
Flickr user fatllama
Creative Commons License



Squaring by Transform II

- Fast Fourier Transform (FFT)
  - An example of a Transform where the Convolution Theorem holds
  - There are more efficient Transforms for digital computers

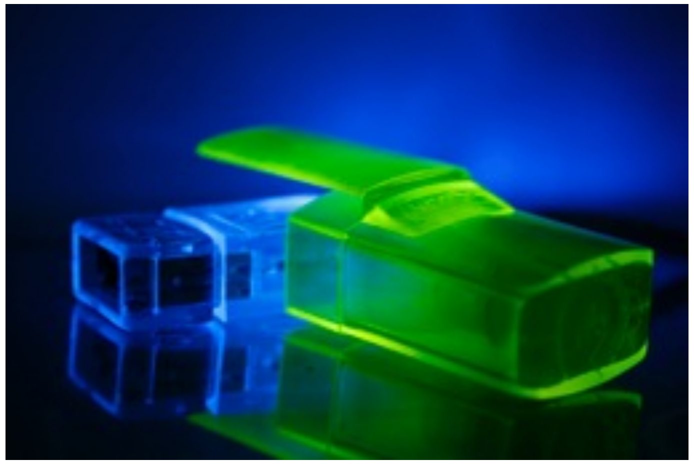


Image Credit:
Flickr user jepoirrier
Creative Commons License

- To compute $A = X^2$
  - Step 1: Transform X: $Y = T(X)$
  - Step 2: Compute dot product: $Z = Y \cdot Y$
  - Step 3: Inverse transform $A = TINV(Z)$
- The prime test is $O(p^2 \ln p)$ operations with Transform Squaring
  - ln p is natural log of p
  - If it takes a computer 1 day to test M(p)
  - 2.7 days to test M(2*p) (instead of 8 days)
  - 40 days to test M(5*p) (instead of 4 months)
  - 7.6 months to test M(10*p) (instead of 2.7 years)

Transform of an Integer?

- Treat the integer as a wave:
  - with bit value amplitude
  - with time starting from low order bit to high order bit
  - 0 1 1 0 0 1 0 1

- Assume that wave form is infinitely repeating:
  - 0 1 1 0 0 1 0 1 0 1 1 0 0 1 0 1 0 1 1 0 0 1 0 1 0 1 1 0 0 1 0 1 0 1 1 0 0 1 0 1 …

- Convert that wave from time domain into frequency domain:
  - Take the spectrum of the infinitely repeating waveform:

I faked this graph :-)

Digital Transforms are Approximations

- The effort to perform a perfect transform requires:
  - Computing infinite sums with infinite precision
    - Infinite operations are "Well beyond" the ability for finite computers to perform :-)
- Inverse Transform converts frequency domain ...

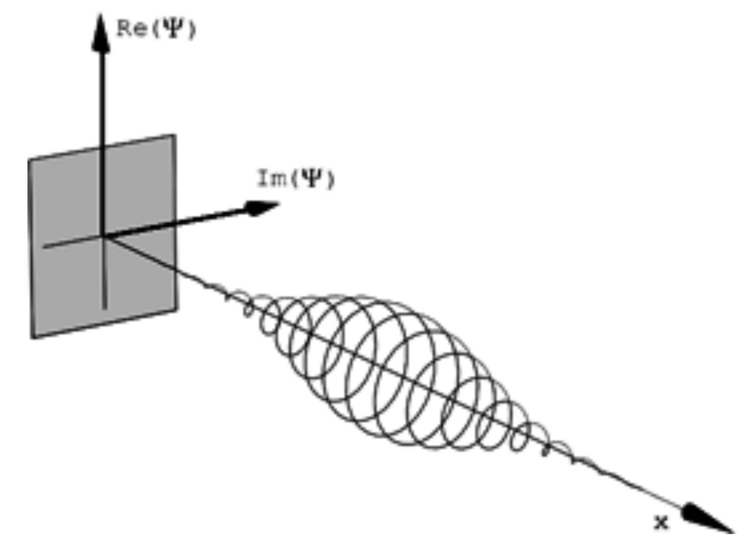
- ... back to time domain:
  - 0.17 0.97 1.04 -0.21 -0.06 0.95 -0.18 0.89
  - Because of "rounding" approximation errors the result is not pure binary
  - So we round to the nearest integer:
  - 0 1 1 0 0 1 0 1
- These examples assumed a 8-point 1D transform

Pad with Zeros to Hold the Final Product

- We need 2n bits to hold the product of two n-bit values
  - The Transform needs twice the points to hold the product
- We add n leading 0's to our values before we multiply:
  - 0 0 0 0 0 0 0 0 0 1 1 0 0 1 0 1

General Square Transform Algorithm

- To square p-bit value:
  - Pad the value with p leading 0 bits
    - Forms a 2*p-bit value: upper half 0's and lower half the value we wish to square]
  - The transform may require a certain number of points
    - Such as a power of two number of points
    - If needed, pad additional 0's until the required number of points is achieved
  - Perform the Transform on the padded value
  - Convolve the signal in the transform space
    - Dot product: Just 1 square for each transform point (not an n^2 operation)
  - Perform the Inverse Transform
  - Divide the real part of each digit by the number of points and round to the nearest integer
  - Propagate carries



FFT Square Example Output

- input: 0 0 3 2
- freq: (-1.251,3.001i) (0.248,0.003i) (-1.250,-3.005i) (6.257,0.007i)
  - After transform - FFT errors exaggerated for dramatic effect
- fft output: (0.091,-0.041i) (35.896,0.055i) (47.916,-0.127i) (16.183,0.127i)
  - after square and inverse transform - FFT errors exaggerated for dramatic effect
- round to integers: (0,0i) (36,0i) (48,0i) (16,0i)
- extract reals: 0 36 48 16
- scale output: 0 9 12 4
  - Divide each cell by the initial number of cells
- After carries propagated: 1 0 2 4

FFT Square Example Makefile

- Try the FFTW library:
  - http://www.fftw.org/

- Makefile:

```
# FFT square example using fftw
#
# See: http://www.fftw.org
#
# chongo (Landon Curt Noll) /\oo/\  -- Share and Enjoy!  :-)

fftsq: fftsq.c
	cc fftsq.c -lfftw3 -lm -Wall -o fftsq
```

FFT Square Example C Source.0

```
/*
 * FFT square example using fftw
 * See: http://www.fftw.org
 * chongo (Landon Curt Noll) /\oo/\  -- Share and Enjoy!  :-)
 */

#define N 4      /* points in FFT */

/* digit arrays - least significant digit first */
long input[N] = { 2, 3, 0, 0 }; /* input integer, upper half 0 padded */
long output[N];                 /* squared input */

#include <stdlib.h>
#include <math.h>
#include <fftw3.h>
#include <complex.h>

int
main(int argc, char *argv[])
{
    complex *in;                /* input as complex values */
    complex *freq;              /* transformed integer as complex values */
    complex *sq;                /* squared input */
    fftw_plan trans;            /* FFT plan for forward transform */
    fftw_plan invtrans;         /* FFT plan for inverse transform */
    int i;

    /* allocate for fftw */
    in = (complex *) fftw_malloc(sizeof(fftw_complex) * N);
    freq = (complex *) fftw_malloc(sizeof(fftw_complex) * N);
    sq = (complex *) fftw_malloc(sizeof(fftw_complex) * N);
```

FFT Square Example C Source.1

```
/*
 * load long integers into FFT input array
 */
for (i=0; i < N; ++i) {
    in[i] = (complex)input[i];  /* long integer to complex conversion */
}


/* debugging */
printf("input:  ");
for (i=N-1; i >= 0; --i) {
    printf(" %ld  ", input[i]);
}
putchar('\n');


/*
 * forward transform
 */
trans = fftw_plan_dft_1d(N, (fftw_complex*)in, (fftw_complex*)freq,
                         FFTW_FORWARD, FFTW_ESTIMATE);
fftw_execute(trans);
```

FFT Square Example C Source.2

```
/*
 * square the elements
 */
for (i=0; i < N; ++i) {
    freq[i] = freq[i] * freq[i];    /* square the complex value */
}


/* debugging */
printf("freq: ");
for (i=N-1; i >= 0; --i) {
    printf("(%f,%fi) ", creal(freq[i])/N, cimag(freq[i])/N);
}
putchar('\n');


/*
 * inverse transform
 */
invtrans = fftw_plan_dft_1d(N, (fftw_complex*)freq, (fftw_complex*)sq,
                            FFTW_BACKWARD, FFTW_ESTIMATE);
fftw_execute(invtrans);


/*
 * convert complex to rounded long integer
 */
for (i=0; i < N; ++i) {
    output[i] = (long)(creal(sq[i]) / (double)N); /* complex to scaled long integer */
}
```

FFT Square Example C Source.3

```
    /*
     * output the result
     */
    printf("fft output: ");
    for (i=N-1; i >= 0; --i) {
        printf("(%f,%fi) ", creal(sq[i]), cimag(sq[i]));
    }
    putchar('\n');
    /* NOTE: Carries are not propagated in this code */
    printf("scaled output: ");
    for (i=N-1; i >= 0; --i) {
        printf(" %ld  ", output[i]);
    }
    putchar('\n');


    /*
     * cleanup
     */
    fftw_destroy_plan(trans);
    fftw_destroy_plan(invtrans);
    fftw_free(in);
    fftw_free(freq);
    fftw_free(sq);
    exit(0);
}
```

FFT Square Example C Source - Just the Facts

```
/* load long integers into FFT input array */
for (i=0; i < N; ++i) {
    in[i] = (complex)input[i];      /* long integer to complex conversion */
}


/* forward transform */
trans = fftw_plan_dft_1d(N, in, freq, FFTW_FORWARD, FFTW_ESTIMATE);
fftw_execute(trans);


/* square the elements */
for (i=0; i < N; ++i) {
    freq[i] = freq[i] * freq[i];    /* square the complex value */
}


/* inverse transform */
invtrans = fftw_plan_dft_1d(N, freq, sq, FFTW_BACKWARD, FFTW_ESTIMATE);
fftw_execute(invtrans);


/* convert complex to rounded long integer */
for (i=0; i < N; ++i) {
    output[i] = (long)(creal(sq[i]) / (double)N) /* complex to scaled long integer */
}
/* NOTE: TODO: propagate carries */
```

The Details are in the Rounding!

- Just like in classical multiplication / squaring
  - Using a larger base helps
  - We do not need to put 1 digit per cell like in the previous "examples"

Image credit:
Flickr user veruus
Creative Commons License

- What base can we use?
  - Too small of a base: Slows down the test!
  - Too large of a base: The final rounding rounds to the wrong value

- Expect to use a base of "about 1/4" of the CPU's numeric precision
  - The Amdahl 1200 had a floating point 96 bit mantissa: 18900 point transform used a base of 2^{23}

- Analyze the digital rounding errors
  - Estimate the maximum precision you can use
  - Test your estimate
  - Test worst case energy spike patterns
  - Add check code to your multiply / square routine to catch any other mistakes
    - Verify that $U_x^2 \bmod 2^{64}-3 = (U_x \bmod 2^{64}-3)^2 \bmod 2^{64}-3$
    - Verify that complex part of point output rounds to 0

Try non-Fourier Transforms

- Some of the integer transforms perform well on some CPUs
  - Especially where integer CPU ops are fast vs. floating point
- PFA Fast Fourier Transform and on Winograd's radix FFTs
  - Used by Amdahl 6 to find a largest known prime
- Dr. Crandall's transform
  - See https://www.ams.org/journals/mcom/1994-62-205/S0025-5718-1994-1185244-1/S0025-5718-1994-1185244-1.pdf
  - GIMPS used Dr. Crandall's transform to find many largest known primes
  - See also https://www.daemonology.net/papers/fft.pdf
- Schönhage–Strassen Transform
  - Used by the GNU Multiple Precision Arithmetic Library
  - Used by FLINT
- Roll your own efficient Transform
  - Ask a friendly computational mathematician for advice

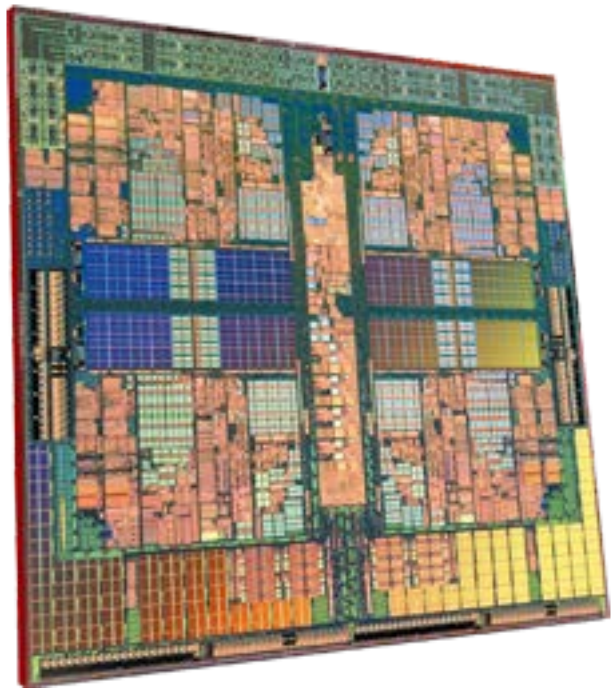


Image Credit: WikiPedia
Creative Commons License

Even Better: Number Theoretic Transforms

- Avoids complex arithmetic
  - Uses powers of integers modulo some prime instead of complex numbers

- Examples:
  - Schönhage–Strassen algorithm
    - https://tonjanee.home.xs4all.nl/SSAdescription.pdf
    - GNU Multiple Precision Arithmetic Library, See: https://gmplib.org
    - FLINT: Fast Library for Number Theory: http://www.flintlib.org
  - Crandall's Transform
    - https://www.ams.org/journals/mcom/1994-62-205/S0025-5718-1994-1185244-1/S0025-5718-1994-1185244-1.pdf
    - https://www.daemonology.net/papers/fft.pdf
  - Fürer's algorithm
    - Anindya De, Chandan Saha, Piyush Kurur and Ramprasad Saptharishi gave a similar algorithm that relies on modular arithmetic
    - Symposium on Theory of Computation (STOC) 2008, see https://arxiv.org/abs/0801.1416

- A good primer on Number Theoretic Transform Multiplication:
  - https://tonjanee.home.xs4all.nl/SSAdescription.pdf

Number Theoretic Transform Multiply Example

- Number-theoretic transforms in the integers modulo 337 are used, selecting 85 as an 8th root of unity
- Base 10 is used in place of base 2^w for illustrative purposes

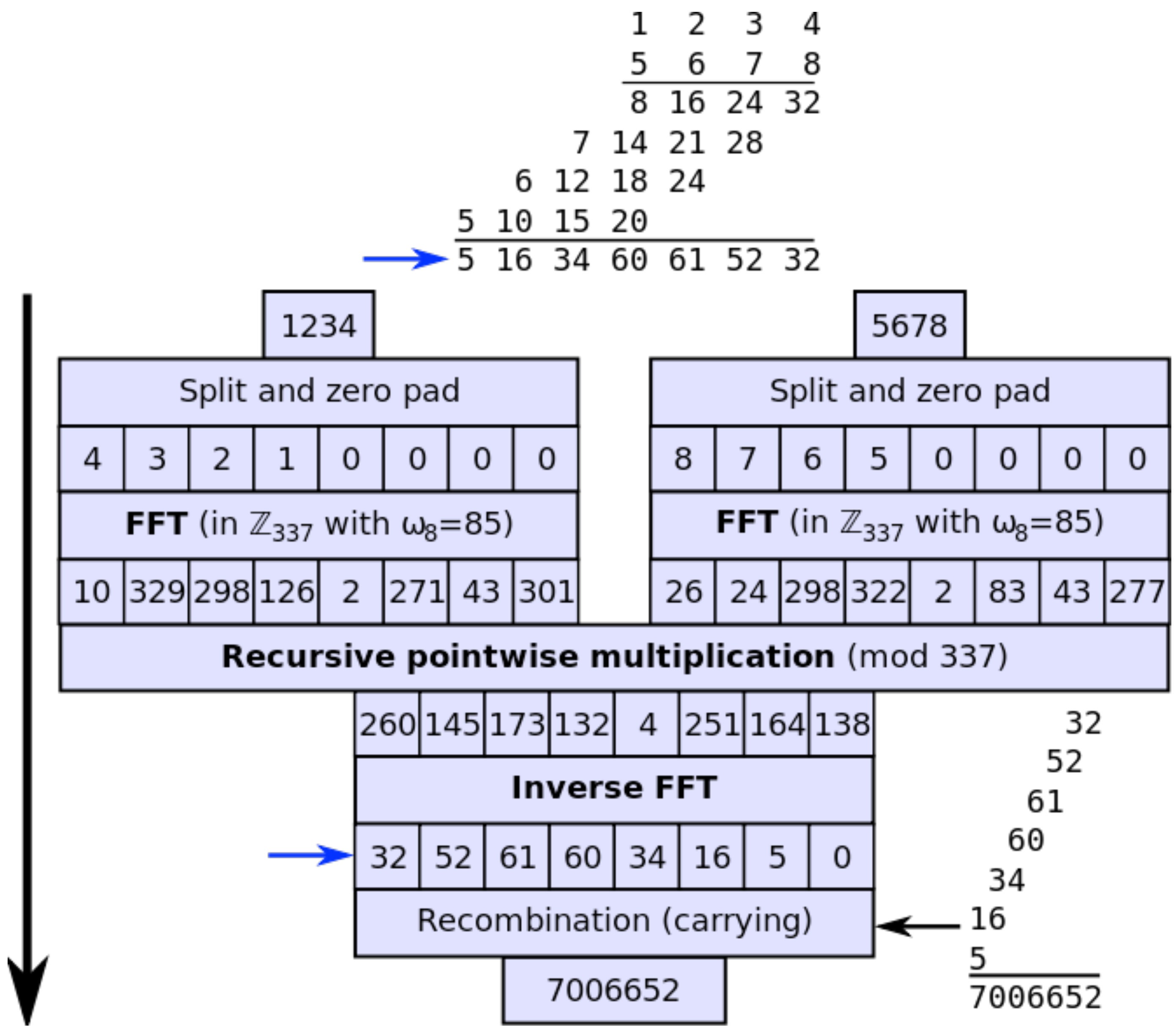


Image Credit: Wikipedia
Creative Commons License

Mersenne Test Revisited

- Start with a table of M(p) candidates (where p is prime)

- Look for small factors, tossing out those with factors that are not prime
  - Until the rate of tossing out candidates is slower than Lucas-Lehmer test rate

- For each M(p) remaining, perform the Lucas-Lehmer test
  - $U_2 = 4$ and $U_{i+1} \equiv U_i^2 - 2 \bmod M(p)$ until U_p is computed
    - Pad U_x with leading 0's (at least p bits, more if required by Transform size)
    - Transform
    - Square each point
    - Inverse Transform
    - Divide real parts of points by point count and round to integers
    - Propagate carries
    - Subtract 2
    - Mod M(p) using "shift and add" method
  - If $U_p \equiv 0$ then M(p) is prime, otherwise it is not prime

EFF Cooperative Computing Awards

- $50 000 - prime number with at least 1 000 000 decimal digits
  - Awarded 2000 April 2
- $100 000 - prime number with at least 10 000 000 decimal digits
  - Awarded 2009 October 22
- $150 000 - prime number with at least 100 000 000 decimal digits
  - Unclaimed as of 2022 Apr 25
- $250 000 - prime number with at least 1 000 000 000 decimal digits
  - Unclaimed as of 2022 Apr 25
  - BTW: Landon is on the EFF Cooperative Computing Award Advisory Board
    - And therefore Landon is **NOT** eligible for an award
    - Because Landon is an advisor, he will **NOT** give **private** advice to individuals seeking large primes
    - Landon does give public classes / lectures where the content + Q&A are open to anyone attending

EFF Cooperative Computing Awards II

- Funds donated by an anonymous donor to EFF

- Official Rules:
  - https://www.eff.org/awards/coop/rules
    - See also: https://www.eff.org/awards/coop/faq
  - Rules designed by Landon Curt Noll
  - See https://www.eff.org/awards/coop/primeclaim-43112609 for a valid claim

ELECTRONIC FRONTIER FOUNDATION

- Rule 4F: You must publish your proof in a refereed academic journal!
  - Your claim must include a citation and abstract of a published paper that announces the discovery and outlines the proof of primality. The cited paper must be published in a refereed academic journal with a peer review process that is approved by EFF.

- EFF Cooperative Computing Award Advisory Board
  - Landon Curt Noll (Chair), Simon Cooper, Chris K. Caldwell
  - Advisory Board members are not eligible to win an award

www.isthe.com/chongo/tech/math/prime/prime-tutorial.pdf
Questions for Part 1

Image Credit:
Flickr user bitzcelt
Creative Commons License

- 1) Was M(4253) ever the largest known prime?
  - Hint: See slide 30
- 2) How do we know that $2^{1000000000}-1$ is not prime?
  - Hint: See slide 29
- 3) Should one try to factor M(p) before running the Lucas-Lehmer test?
  - Hint: think about when p is a large prime AND see slide 41
- 4) If a Lucas-Lehmer test of M(p) using Classical Squaring takes 1 hour, how long would it take to test M(x) where x is about 100*p?
  - Hint: See slides 40 & 41
- 5) If it took GIMPS 12 days to prove M(82589933) is prime, how long should it take them to test a Mersenne prime just large enough to claim the $150000 award?
  - Hint: M(332192831) has 100 000 007 digits
  - Hint: See slides 49, 65, 66 [[NOTE: M(332192831) is likely not prime]] [[NOTE: They used Transforms to Square]]
- 6) Prove that $M(7) = 2^7-1 = 127$ is prime using the Lucas-Lehmer test
  - Hint: See slides 18, 19, 27, 28

Part 2 - Large Riesel Primes Faster

- $2^{31}-1$: Riesel Test: Searching sideways
- $2^{61}-1$: Pre-screening Riesel test candidates
- $2^{89}-1$: Multiply+Add in Linear Time
- $2^{127}-1$: Final Words and Some Encouragement
- $2^{521}-1$: Resources

Image Credit:
Flickr user NguyenDai
Creative Commons License

2^{31}-1: Riesel Test: Searching sideways

- While the Lucas-Lehmer test is the most efficient proof of primality known ...
- … It is not the most efficient method to find a new largest known prime!

- Why? Well …

- Mersenne Primes are rare
  - Only 47 out of 43112609 Mersenne Numbers are prime
    - And even these odds are skewed (too good to be true), because of the pile of small Mersenne Primes
    - Only 7 of the 29260728 Mersenne numbers that are between 1 million to 10 million decimal digits in size, are prime
  - As p grows, Mersenne Prime M(p) get even more rare

- As p gets larger, the Lucas-Lehmer test with the best multiply worse than:
  - $O(p^2 \ln p)$
  - Worse still, numbers may grow large with respect to memory cache
    - Busting the cache slows down the code
  - The length of time to test will likely exceed the MTBF and MTBE
    - Mean Time Before Failure and Mean Time Before Error
  - You must verify (recheck your test) and have someone else independently verify (3rd test)
    - So plan on the time to test the number at least 3 times!
  - The GIMPS test for the 2018 largest known prime took 12 days

Advantages of Searching for $h*2^n-1$ Primes

- Riesel test for $h*2^n-1$ is almost as efficient as Lucas-Lehmer test for 2^p-1
  - Riesel test is about 10% slower than Lucas-Lehmer
    - When h is small enough … but not too small
  - Test is very similar to Lucas-Lehmer so many of the performance tricks apply

- Testing $h*2^n-1$ grows as n grows - Avoid the exponential wall (go sideways)
  - Solution: pick a fixed value n and change only the value of h
    - Use odd values of $h < 2^n$ (if h in even, divide by 2 and increase n until h is odd)
    - A practical bound for h is: $2*n < h < 16*n$
    - Better still keep $2*n < h <$ single precision unsigned integer (on a 64-bit machine, this might be 2^{32} or 2^{64})
  - N may be selected to optimize the algorithm used to square large integers

- Pre-screening can eliminate >98.5% of candidates

- When $2*n < h < 2^n$ primes of the form $h*2^n-1$ are not rare like Mersenne Primes
  - They tend appear about as often as your average prime that is about the same size
  - Odds that $h*2^n-1$ is prime when $2*n < h < 2^n$ is about 1 in $2*\ln(h*2^n-1)$
    - You can “guesstimate” the amount of time it will take to find a large prime

Mersenne Primes Dethroned

- 1989: $391581 * 2^{216193}-1$ 65087 digits   Amdahl 6 using the Amdahl 1200
  - Only 37 digits larger than M(216091) that was found in 1985
    - "Just a fart larger " - Dr. Shanks
  - BTW: The number we tested was really $783162 * 2^{216192}-1$
- Amdahl 6 team:
  - Landon Curt Noll, Gene Smith, Sergio Zarantonello, John Brown, Bodo Parady, Joel Smith
- Did not use the Lucas-Lehmer Test
- Squared numbers using Transforms
  - First use for testing non-Mersenne primes
  - First efficient use for small 1000 digit tests

Image Credit:
Mrs. Zarantonello

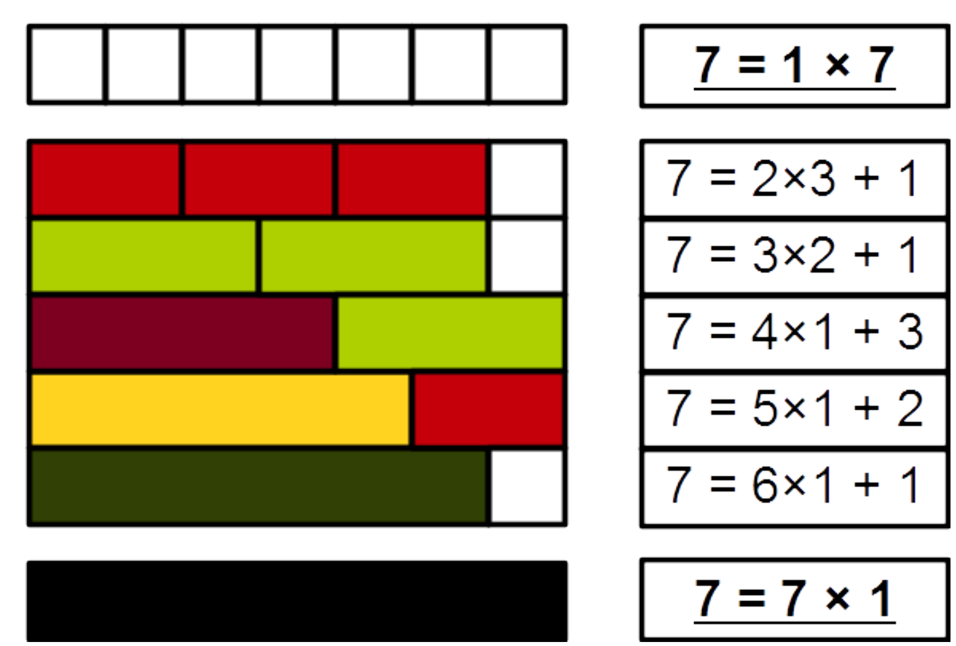
Riesel Test for $h*2^n-1$ is Lucas-Lehmer like

- $h*2^n-1$ is prime if and only if odd $h < 2^n$,

  $h*2^n-1$ not divisible by 3, and

  $U_n \equiv 0 \bmod h*2^n-1$
  - If h in even, divide by 2 and increase n until h is odd
  - $U_2 = V(h)$
    - We will talk about how to calculate V(h) in the slides that follow
  - $U_{x+1} \equiv U_x^2 - 2 \bmod h*2^n-1$

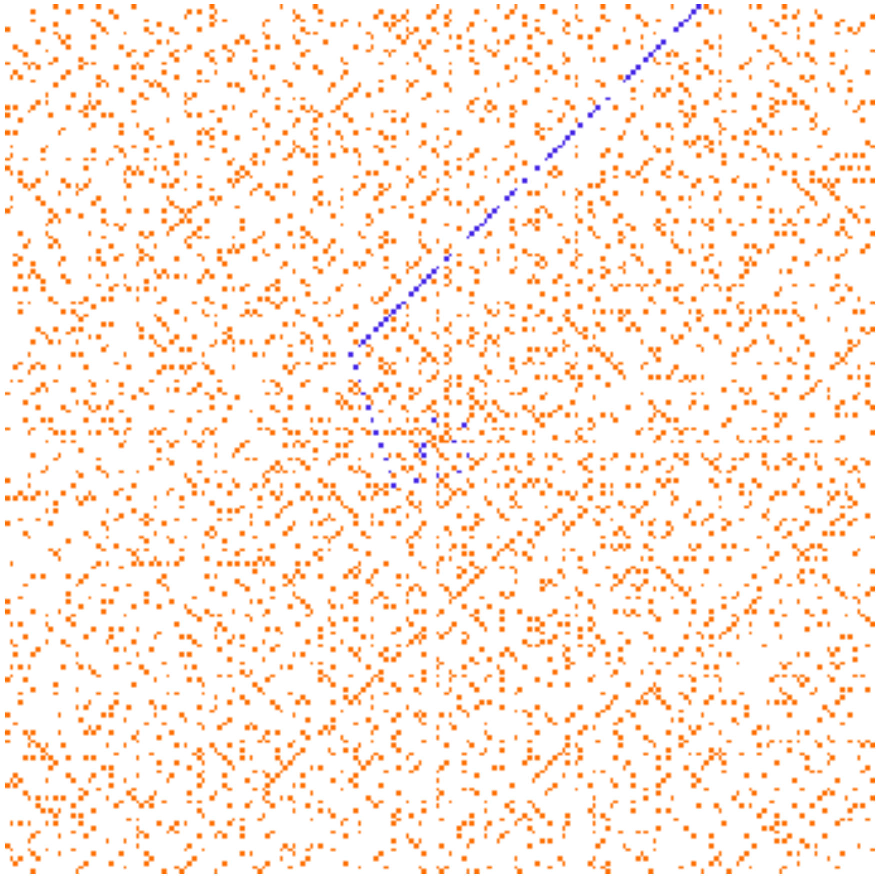
- Differences from the Lucas-Lehmer test
  - Need to verify $h*2^n-1$ is not a multiple of 3
  - The power of 2 does not have to be prime
  - We calculate mod $h*2^n-1$ not mod 2^n-1
  - U_2 depends on V(h) and is not always 4



7 is prime
Image Credit: Wikipedia

Example code for Riesel Test

- Example code for Riesel Test:
  - http://www.isthe.com/chongo/src/calc/lucas-calc
    - Source code contains lots and lots of comments with lots of references to papers - worth reading!
    - NOTE: Only use this code as a guide, calc by itself is not intended to find a new largest known prime
    - Written in Calc - A C-like multi-precision calculator: http://www.isthe.com/chongo/tech/comp/calc/
  - https://github.com/arcetri/gmprime
    - Written in C
    - Implements the algorithm of http://www.isthe.com/chongo/src/calc/lucas-calc
    - A potential code base from which to start optimization
    - Uses GMU MP
    - Extensive test code
    - Had debugging options
  - https://github.com/arcetri/goprime
    - A potential code base from which to start optimization
    - Once version written in go benchmarks several square methods
    - One version written in C that uses flint: http://www.flintlib.org
  - http://jpenne.free.fr/index2.html
    - LLR code implements Riesel test



The Ulan spiral
Image Credit:
Wikipedia

Prior to finding U(2) - Riesel test setup

- Pretest: Verify $h^* 2^n-1$ is not a multiple of 3
  - Do not test if ($h \equiv 1 \bmod 3$ AND n is even) NOR
    if ($h \equiv 2 \bmod 3$ AND n is odd)
    - This pretest is mandatory when h is not a multiple of 3
    - No need to test $h*2^n-1$ because in this case 3 is a factor!

- Test only odd h
  - Only test odd h, ignore even h
    - One can always divide h by 2 and add one to 1 until h becomes odd

- Riesel test requires $h < 2^n$
  - We recommend using odd h in this range: $2*n < h < 16*n$

Calculating U(2) when h is not a multiple of 3

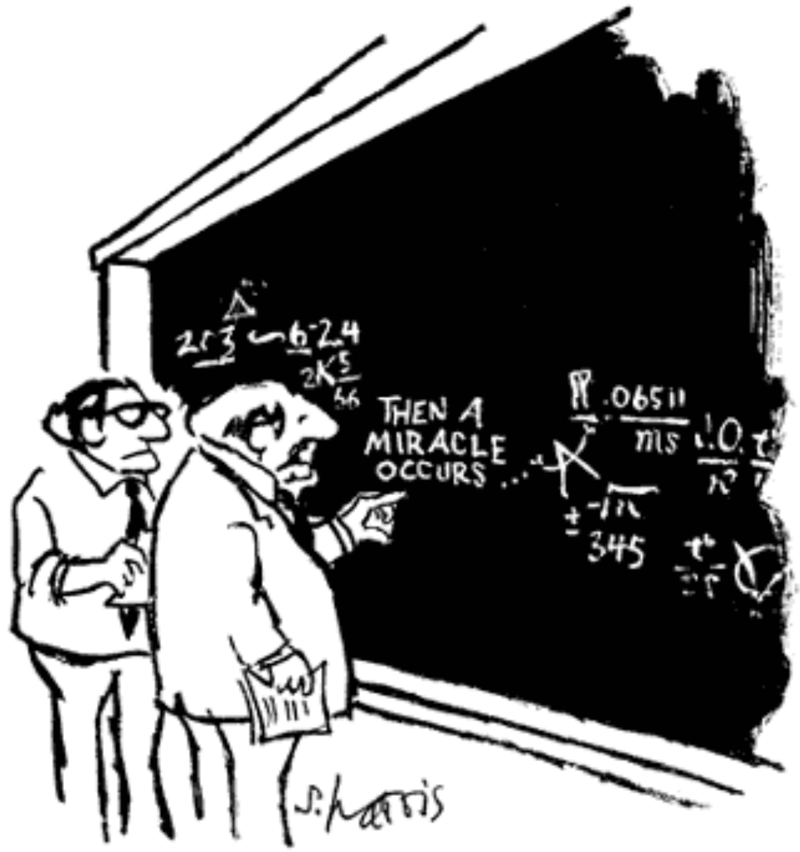
- Pretest: Verify that $h*2^n-1$ is not a multiple of 3
  - Do not test if ($h \equiv 1 \bmod 3$ AND n is even) NOR
    if ($h \equiv 2 \bmod 3$ AND n is odd)
- Note that we are considering only the case when h is odd
  - For even h, divide h by 2 and add one to 1 until h becomes odd
- Start with:
  - $V(0) = 2$
  - $V(1) = 4$ (NOTE: V(1) = 4 always works when h is not multiple of 3)
- Compute V(h) using these recursion formulas:
  - $V(i+1) = [V(1)*V(i) - V(i-1)] \bmod h*2^n-1$
  - $V(2*i) = [V(i)^2 - 2] \bmod h*2^n-1$
  - $V(2*i+1) = [V(i)*V(i+1) - V(1)] \bmod h*2^n-1$
- $U(2) = V(h)$

Calculating U(2) when h is a multiple of 3

- Pretest: Verify that $h*2^n-1$ is not a multiple of 3
  - Do not test if (h ≡ 1 mod 3 AND n is even) NOR
    if (h ≡ 2 mod 3 AND n is odd)
- Note that we are considering only the case when h is odd
  - For even h, divide h by 2 and add one to 1 until h becomes odd
- Start with:
  - V(0) = 2
  - V(1) = X > 2 where Jacobi(X-2, $h*2^n-1$) = 1
    and where Jacobi(X+2, $h*2^n-1$) = -1
    - Jacobi(a,b) is the Jacobi Symbol
    - See "A note on primality tests for N = $h*2^n-1$"

      An excellent 5 page paper by Öystein J. Rödseth, Department of Mathematics, University of Bergen, BIT Numerical Mathematics. 34 (3): 451–454.

      https://link.springer.com/article/10.1007/BF01935653
- Compute V(h) using these recursion formulas:
  - V(i+1) = [V(1)*V(i) - V(i-1)] mod $h*2^n-1$
  - V(2*i) = $[V(i)^2 - 2]$ mod $h*2^n-1$
  - V(2*i+1) = [V(i)*V(i+1) - V(1)] mod $h*2^n-1$
- U(2) = V(h)

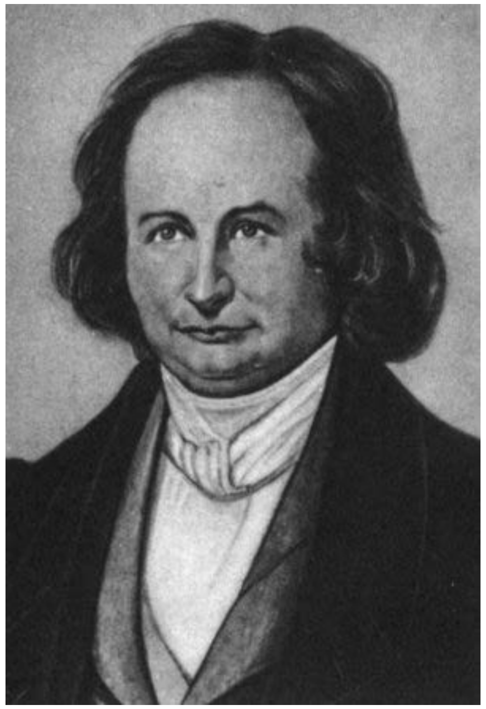


"I think you should be more explicit here in step two."

Image Credit:
Copyright © Sidney Harris
www.sciencecartoonsplus.com

Calculating the Jacobi symbol is easy

- Pre-condition: b must be an odd (i.e., b ≡ 1 mod 2) and $0 < a < b$
  - ```
    Jacobi(a,b) {
        j := 1
        while (a is not 0) {
            while (a is even) {
                a := a / 2
                if ((b ≡ 3 mod 8) or (b ≡ 5 mod 8))
                    j := - j
            }
            temp := a;  a := b;  b := temp        // exchange a and b
            if ((a ≡ 3 mod 4) and (b ≡ 3 mod 4))
                j := - j
            a := a mod b
        }
        if (b is 1)
            return j
        else
            return 0
    }
    ```



Carl Gustav Jacob Jacobi
Image Credit: Wikipedia
    ```

How to find V(1) when h is a multiple of 3

- Try these values of X in the following order:
  - 3, 5, 9, 11, 15, 17, 21, 27, 29, 35, 39, 41, 45, 51, 57, 59, 65, 69, 81
    - Search the list for X where Jacobi(X-2, $h*2^n-1$) = 1 and Jacobi(X+2, $h*2^n-1$) = -1

      Set V(1) to the first value of X that satisfies those 2 Jacobi equations
  - Fewer than 1 out of 1000000 cases, when h is an odd multiple of 3, are not satisfied by the above list

- If none of those values work for V(1), test odd values of X starting at 83
  - Find first odd X ≥ 83 where Jacobi(X-2, $h*2^n-1$) = 1 and Jacobi(X+2, $h*2^n-1$) = -1

- An implementation of this method using C & GNU MP:
  - https://github.com/arcetri/gmprime

Image Credit:
Flickr user amandabhslater
Creative Commons License

How to find V(1) when h is NOT a multiple of 3

- To speed up generating U(2) = V(h), we need to find a small V(1)
- If h is odd and not a multiple of 3, and
  if Jacobi(1, $h*2^n-1$) = 1 and Jacobi(5, $h*2^n-1$) = -1 then
  - V(1) = 3
- else
  - V(1) = 4
- 40% of $h*2^n-1$ values can use a V(1) value of 3
  - 4 always works for $h*2^n-1$ when h is not a multiple of 3
- An implementation of this method using C & GNU MP:
  - https://github.com/arcetri/gmprime

Image Credit:
Landon Curt Noll

Riesel Test example: $7*2^5-1 = 223$

- $7*2^5-1$ is prime if and only if $7 < 2^5$ and $U_5 \equiv 0 \bmod 7*2^5-1$
  - $V(0) = 2$
  - $V(1) = 3$ (because Jacobi(1,223) == 1 and Jacobi(5,223) == -1, we could also use 4 because h==7 is not a multiple of 3)
  - $V(i+1) = [V(1)*V(i) - V(i-1)] \bmod h*2^n-1$
  - $V(2*i) = [V(i)^2 - 2] \bmod h*2^n-1$
  - $V(2*i+1) = [V(i)*V(i+1) - V(1)] \bmod h*2^n-1$
- Calculating V(7) from V(0) and V(1)
  - $V(0) = 2$
  - $V(1) = 3$ (because Jacobi(1,223) == 1 and Jacobi(5,223) == -1, see the previous slide)
  - $V(2) = [V[1]^2 - 2] \bmod 223 = 7$
  - $V(3) = [V[1]*V[2] - V[1]] \bmod 223 = 18$
  - $V(4) = [V[2]^2 - 2] \bmod 223 = 47$
  - $V(5) = [V(1)*V(4) - V(3)] \bmod 223 = 123$
  - $V(6) = [V(1)*V(5) - V(4)] \bmod 223 = 99$
  - $V(7) = [V(1)*V(6) - V(5)] \bmod 223 = 174$

Image Credit:
Landon Curt Noll

Riesel Test example: $7*2^5-1 = 223$

- $7*2^5-1$ is prime if and only if $7 < 2^5$ and $U_5 \equiv 0 \bmod 7*2^5-1$
  - $U_2 = V(h)$
  - $U_{x+1} \equiv U_x^2 - 2 \bmod h*2^n-1$

- Riesel test: $7*2^5-1 = 223$
- $U_2 = V(7) = 174$
- $U_3 = 174^2 - 2 = 30274 \bmod 223 \equiv 169$
- $U_4 = 169^2 - 2 = 28559 \bmod 223 \equiv 15$
- $U_5 = 15^2 - 2 = 223 \bmod 223 \equiv 0$

Image Credit:
Landon Curt Noll

- Because $U_5 \equiv 0 \bmod 223$ we know that $7*2^5-1 = 223$ is prime

Calculating mod $h*2^n-1$

- Very similar to the "shift and add" method for mod 2^n-1

- Split the value into two chunks:

$U_x^2-2 =$

J	K
	← n bits long →

- Then $U_x^2-2 \bmod h*2^n-1 \equiv \mathrm{int}(J/h) + (J \bmod h)*2^n + K$

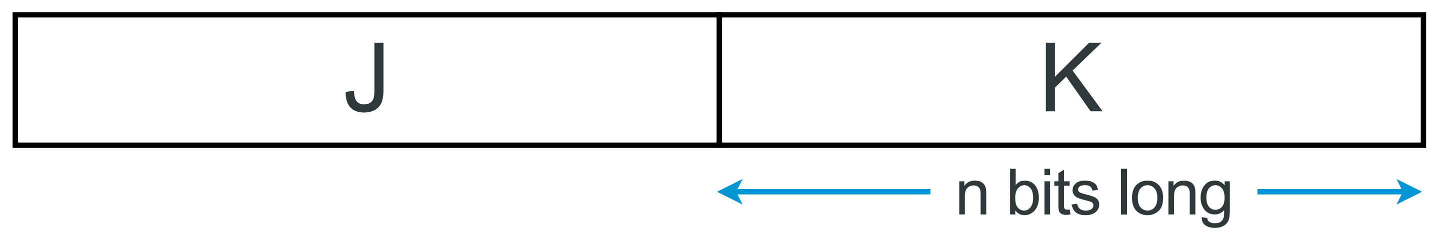
- If $\mathrm{int}(J/h) + (J \bmod h)*2^n + K > h*2^n-1$ then repeat the above

- Mod $h*2^n-1$ can be done in $O(d)$ steps

Keep h single precision, but not too single!

- Calculating mod $h*2^n-1$ requires computing: $int(J/h) + (J \bmod h)*2^n + K$
  - K is the first n bits, J is everything beyond the first n bits:

$U_x^2-2 =$



- Calculating int(J/h) and (J mod h) takes more time for double precision h
  - keep $h < 2^{63}$ (when testing on a 64-bit machine)

- Do NOT make h too small!
  - primes of the form $h*2^n-1$ tend to be rare when h is tiny
    - Keep $2*n < h$
  - But not too much greater than $2*n$ to avoid double precision mod speed issues
    - For example, keep: $2*n < h < 16*n$

2^{61}-1: Pre-screening Riesel Test Candidates

- Eliminate $h*2^n-1$ values that are a multiple of small primes
  - Avoid testing large values are "obviously" not prime
- We will use **sieving techniques** to quickly find multiples of small primes
- In order to understand these **sieving techniques** ...
  - Let first look in detail, of how to use the "Sieve of Eratosthenes" to find tiny primes
  - Then we will apply these ideas to quickly eliminate Riesel candidates that are multiple or small primes

The Sieve of Eratosthenes

- Sieve the integers
  - Given the integers:
    - 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 ...
  - Ignore 1 (we define it as not prime)
    - *1* 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 ...
  - The next unmarked number is prime .. **2**
    - *1* **2** 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 ...
  - .. cancel every 2nd value after that
    - *1* **2** 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ 9 ~~10~~ 11 ~~12~~ 13 ~~14~~ 15 ~~16~~ 17 ~~18~~ 19 ~~20~~ 21 ~~22~~ 23 ~~24~~ 25 ~~26~~ 27 ~~28~~ 29 ~~30~~ 31 ~~32~~ ...
  - The next value remaining, **3**, is prime so mark it and cancel every 3rd value after that
    - *1* **2** **3** *4* 5 ~~6~~ 7 *8* ~~9~~ *10* 11 ~~12~~ 13 *14* ~~15~~ *16* 17 ~~18~~ 19 *20* ~~21~~ *22* 23 ~~24~~ 25 *26* ~~27~~ *28* 29 ~~30~~ 31 *32* ...
  - And the same for **5**
    - *1* **2** **3** *4* **5** *6* 7 *8* *9* ~~10~~ 11 *12* 13 *14* ~~15~~ *16* 17 *18* 19 ~~20~~ *21* *22* 23 *24* ~~25~~ *26* *27* *28* 29 ~~30~~ 31 *32* ...
  - And **7** NOTE: Our list ends before $7^2 = 49$, so the mark remaining values as prime
    - *1* **2** **3** *4* **5** *6* **7** *8* *9* *10* **11** *12* **13** *14* *15* *16* **17** *18* **19** *20* *21* *22* **23** *24* *25* *26* *27* *28* **29** *30* **31** *32* ...

When the List does NOT Start with 1

- We can sieve over a segment of that integers that does not start with 1
  - Consider this list:
    - **100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121** ...
  - Start with 1st prime: **2**, find the first multiple of 2, cancel it & every 2nd value
    - ~~100~~ **101** ~~102~~ **103** ~~104~~ **105** ~~106~~ **107** ~~108~~ **109** ~~110~~ **111** ~~112~~ **113** ~~114~~ **115** ~~116~~ **117** ~~118~~ **119** ~~120~~ **121** ...
  - 2nd prime: **3**, find the first multiple of 3, cancel it & every 3rd value
    - *100* **101** ~~102~~ **103** *104* ~~105~~ *106* **107** ~~108~~ **109** *110* ~~111~~ *112* **113** ~~114~~ **115** *116* ~~117~~ *118* **119** ~~120~~ **121** ...
  - 3rd prime: **5**, find the first multiple of 5, cancel it & every 5th value
    - ~~100~~ **101** *102* **103** *104* ~~105~~ *106* **107** *108* **109** ~~110~~ *111* *112* **113** *114* ~~115~~ *116* *117* *118* **119** ~~120~~ **121** ...
  - 4th prime: **7**, find the first multiple of 7, cancel it & every 7th value
    - *100* **101** *102* **103** *104* ~~105~~ *106* **107** *108* **109** *110* *111* ~~112~~ **113** *114* *115* *116* *117* *118* ~~119~~ *120* **121** ...
  - 5th prime **11**, find the first multiple of 11, cancel it & every 11th value
    - *100* **101** *102* **103** *104* *105* *106* **107** *108* **109** ~~110~~ *111* *112* **113** *114* *115* *116* *117* *118* *119* *120* ~~121~~ ...
  - Because our list ends before $13^2 = 169$, the rest are prime
    - *100* **101** *102* **103** *104* *105* *106* **107** *108* **109** *110* *111* *112* **113** *114* *115* *116* *117* *118* *119* *120* *121* ...

Skipping the Even Numbers While Sieving

- When not starting at 1, we can ignore the even numbers and it still works
  - Consider this list:
    - **101 103 105 107 109 111 113 115 117 119 121 123 125 127 129 131 133 135 137 139 141** ...
  - No need to eliminate 2's since the values are all odd
  - Start with **3**, find the first multiple of 3, cancel it & every 3rd
    - **101 103** ~~105~~ **107 109** ~~111~~ **113 115** ~~117~~ **119 121** ~~123~~ **125 127** ~~129~~ **131 133** ~~135~~ **137 139** ~~141~~ ...
  - **5**: find the first multiple of 5, cancel it & every 5th value
    - **101 103** ~~105~~ **107 109** *111* **113** ~~115~~ *117* **119 121** *123* ~~125~~ **127** *129* **131 133** ~~135~~ **137 139** *141* ...
  - **7**: find the first multiple of 7, cancel it & every 7th value
    - **101 103** ~~105~~ **107 109** *111* **113** *115 117* ~~119~~ **121** *123 125* **127** *129* **131** ~~133~~ *135* **137 139** *141* ...
  - **11**: find the first multiple of 11, cancel it & every 11th value
    - **101 103** *105* **107 109** *111* **113** *115 117 119* ~~121~~ *123 125* **127** *129* **131** *133 135* **137 139** *141* ...
  - Because our list ends before $13^2 = 169$, the rest are prime
    - **101 103** *105* **107 109** *111* **113** *115 117 119 121 123 125* **127** *129* **131** *133 135* **137 139** *141* ...

Sieving Over an Arithmetic Sequence

- Consider the following arithmetic sequence
  - We will use the sequence 10*x + 1
    - **101 111 121 131 141 151 161 171 181 191 201 211 221 231 241 251 261 271 281 291 301** ...
  - None of the values are multiples of 2, so **3**: find the first multiple of 3, cancel every 3rd
    - **101** ~~111~~ **121 131** ~~141~~ **151 161** ~~171~~ **181 191** ~~201~~ **211 221** ~~231~~ **241 251** ~~261~~ **271 281** ~~291~~ **301** ...
  - None of the values are multiples of 5, so **7**: find the first multiple of 7, cancel every 7th
    - **101** *111* **121 131** *141* **151** ~~161~~ *171* **181 191** *201* **211 221** ~~231~~ **241 251** *261* **271 281** *291* ~~301~~ ...
  - **11**: find the first multiple of 11, cancel it & every 11th value
    - **101** *111* ~~121~~ **131** *141* **151** *161 171* **181 191** *201* **211 221** ~~231~~ **241 251** *261* **271 281** *291 301* ...
  - **13**: find the first multiple of 13, cancel it & every 13th value
    - **101** *111 121* **131** *141* **151** *161 171* **181 191** *201* **211** ~~221~~ *231* **241 251** *261* **271 281** *291 301* ...
  - **17**: find the first multiple of 17, cancel it & every 17th value
    - **101** *111 121* **131** *141* **151** *161 171* **181 191** *201* **211** ~~221~~ *231* **241 251** *261* **271 281** *291 301* ...
  - Because our list ends before $19^2 = 361$, the rest are prime
    - **101** *111 121* **131** *141* **151** *161 171* **181 191** *201* **211** *221 231* **241 251** *261* **271 281** *291 301* ...

Sieving Over a Sequence of Riesel Sequence

- For a given n, as h increases, $h*2^n-1$ is an arithmetic sequence
  - Consider $h*2^5-1$ for increasing h, all of which are odd so we need not sieve for **2**
    - $1*2^5-1=$**31** $2*2^5-1=$**63** $3*2^5-1=$**95** $4*2^5-1=$**127** $5*2^5-1=$**159** $6*2^5-1=$**191** $7*2^5-1=$**223** $8*2^5-1=$**255** $9*2^5-1=$**287**
  - **3**: find the first multiple of 3, and then cancel every 3rd
    - $1*2^5-1=$**31** $2*2^5-1=$~~63~~ $3*2^5-1=$**95** $4*2^5-1=$**127** $5*2^5-1=$~~159~~ $6*2^5-1=$**191** $7*2^5-1=$**223** $8*2^5-1=$~~255~~ $9*2^5-1=$**287**
  - **5**: find the first multiple of 5, cancel it, and then cancel every 5th value
    - $1*2^5-1=$**31** $2*2^5-1=$*63* $3*2^5-1=$~~95~~ $4*2^5-1=$**127** $5*2^5-1=$*159* $6*2^5-1=$**191** $7*2^5-1=$**223** $8*2^5-1=$~~255~~ $9*2^5-1=$**287**
  - **7**: find the first multiple of 7, cancel it, and then cancel every 7th value
    - $1*2^5-1=$**31** $2*2^5-1=$~~63~~ $3*2^5-1=$*95* $4*2^5-1=$**127** $5*2^5-1=$*159* $6*2^5-1=$**191** $7*2^5-1=$**223** $8*2^5-1=$*255* $9*2^5-1=$~~287~~
  - **11**: find the first multiple of 11 .. there is none in this list, so skip it
  - **13**: find the first multiple of 13 .. there is none in this list, so skip it
  - Because our list ends before $17^2 = 289$, the rest are prime
- Sieving a Riesel Sequence is not useful for finding a large prime
  - It helps quickly identify Riesel numbers that are NOT prime so we won't waste time on them
- Now let return to the quickly eliminating multiples of small primes ...

Pre-screening Riesel Candidates by Sieving

- Given an arithmetic sequence of Riesel numbers: $h*2^n-1$
  - for $2*n < h < 16*n$

- Our list (an arithmetic sequence) to candidates becomes:
  - $(2n+1)*2^n-1$ $(2n+2)*2^n-1$ $(2n+3)*2^n-1$ $(2n+4)*2^n-1$ … $(16n-1)*2^n-1$

- Build an array of bytes: c[0] c[1] .. c[2*n] c[2*n+1] .. c[16*n-1]
  - Where c[h] represents the candidate: $h*2^n-1$
  - Initially set c[0] .. c[2*n] = 0 as these values have too small of an h to be useful
    - c[0] == $0*2^n-1$ == 0 does not need to be primality tested
    - c[1] == $1*2^n-1$ == a mersenne number, might need to be primality tested, but is unlikely to be prime and isn't when n is not prime
  - Set c[2*n+1] .. c[16*n-1] = 1
    - These Riesel candidates have a $2*n < h < 16*n$

- For each test factor Q, **find the first element**, c[X], that is a multiple of Q
  - See the next slide for how we find the first element, $X*2^n-1$, that is a multiple of Q

- Clear c[X] and clear every Q-th element just like we did those sieve examples
  - for (y=X; y < 16*n; y += Q) { c[y] = 0; } /* these values are multiples of Q and therefore not prime */

How to Find the First Element that is Multiple of Q

- How to find the first X where $X*2^n-1$ is a multiple Q
  - We assume that Q is odd
    - Since $X*2^n-1$ is never even, one never needs to consider even values of Q

- Let $R = 2^n \bmod Q$
  - See the next 3 slides for how to compute R

- Let S = Modular multiplicative inverse of R mod Q
  - https://en.wikipedia.org/wiki/Modular_multiplicative_inverse
  - https://rosettacode.org/wiki/Modular_inverse#C
  - See 4 slides down for how we compute the modular multiplicative inverse

- Then the first h where $h*2^n-1$ is a multiple Q is: $S*2^n-1$
  - Sieve out c[S], c[S+Q], c[S+(2*Q)], c[S+(3*Q)], c[S+(4*Q)], c[S+(5*Q)], …
    - These are all multiples of Q and therefore cannot be prime

How to Quickly Compute $R = 2^n \bmod Q$

- One can quickly compute $R = 2^n \bmod Q$ by modular exponentiation

- Observe that:
  - If $y = 2^x \bmod Q$
  - then $2^{(2x)} \bmod Q = y^2 \bmod Q$ (the 0-bit case)
  - and $2^{(2x+1)} \bmod Q = 2*y^2 \bmod Q$ (the 1-bit case)

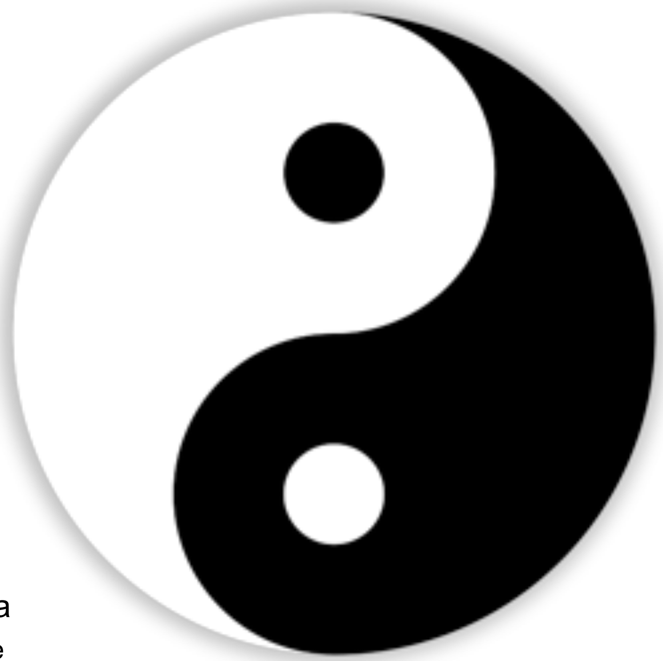


Image Credit: Wikipedia
Creative Commons License

Minimize the 1-bits in n for Speed's Sake!

- Note that computing $R = 2^n \bmod Q$ is faster when n, in binary, has fewer 1 bits

- For each 0-bit in n:
  - square and mod

- For each 1-bit in n:
  - square, multiply by 2, then mod

- It is best to minimize the number of 1-bits in n
  - Choose an n that is a small multiple of a power of 2
    - Such values of n have lots of 0-bits at the bottom

Image Credit:
Flickr user AceFrenzy
Creative Commons License

The Modular Exponent Trick - Small Example

- Compute R = 2^{117} mod 3391
  - In the example, we are pre-screening candidates of the form $h*2^n-1$, where n = 117
  - We show how to compute R = 2^{117} mod Q, where Q = 3391 is an example test factor
- The exponent of 2, in binary, is 117: **111**0101, we start with some leading bits
  - We start with on the leading **3** bits just for purposes of illustration
  - On CPUs with **w**-bit words, you should start with the **w** leading bits
- 2^7: Start with the leading bits where we can raise 2 to that power
  - Raise 2 to the leading **3** bits and mod: 2^7 mod 3391 ≡ **128**
- 2^{14}: Next bit in the exponent, 111**0**101 is **0**:
  - **0**-bit: square and mod: **128**^{2} mod 3391 ≡ **2820**
- 2^{29}: Next bit in the exponent, 1110**1**01 is **1**:
  - **1**-bit: square, multiply by 2, then mod: 2***2820**^{2} mod 3391 ≡ **1010**
- 2^{58}: Next bit in the exponent, 11101**0**1 is **0**:
  - **0**-bit: square and mod: **1010**^{2} mod 3391 ≡ **2800**
- 2^{117}: Next bit in the exponent, 111010**1** is **1**:
  - **1**-bit: square, multiply by 2, then mod: 2***2800**^{2} mod 3391 ≡ **16**
- Thus R = 2^{117} mod 3391 ≡ **16**
- While computing R = 2^n mod Q, the largest value encountered is < $2*Q^2$

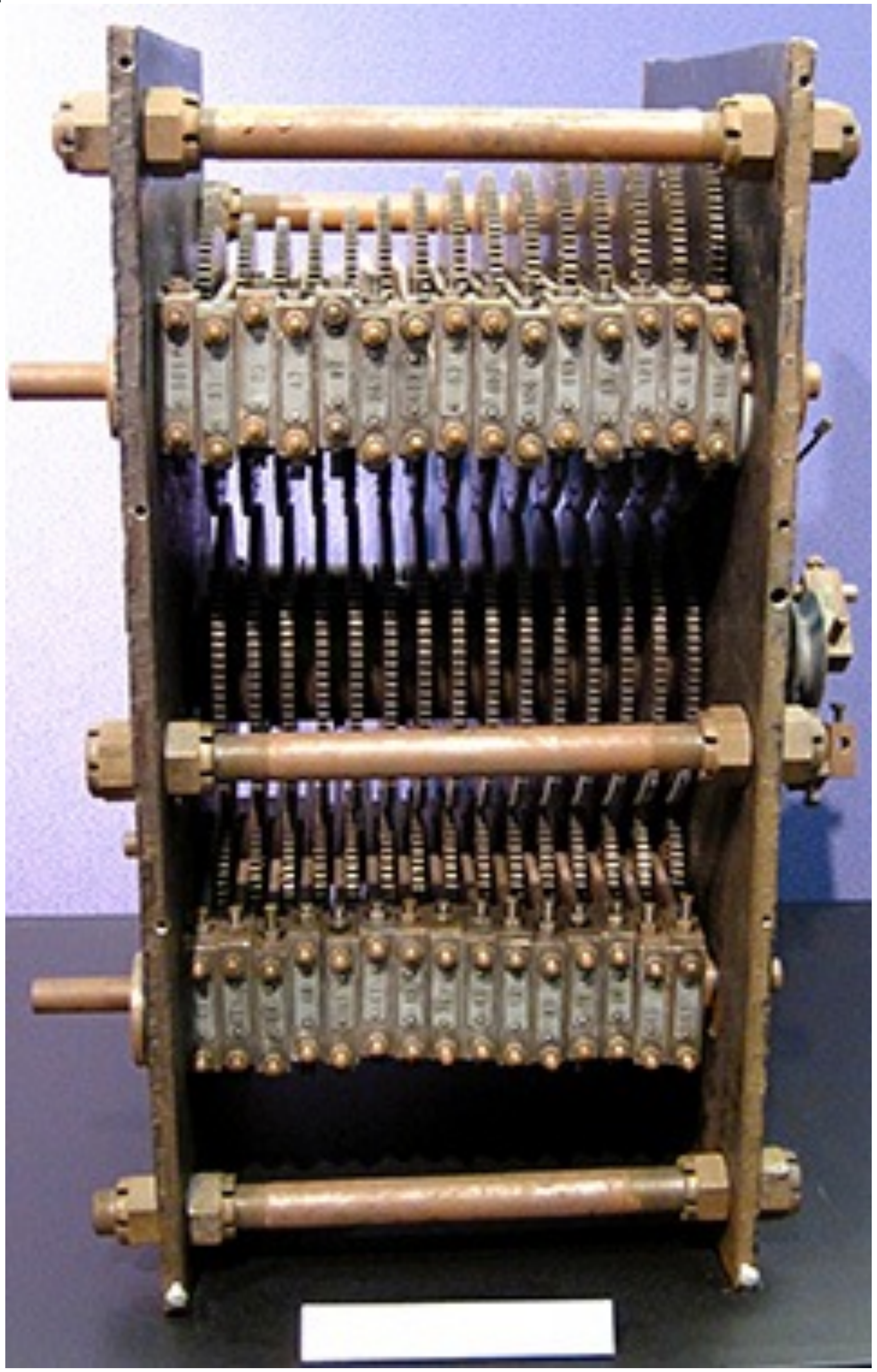


Image Credit:
Flickr user anton.kovalyov
Creative Commons License

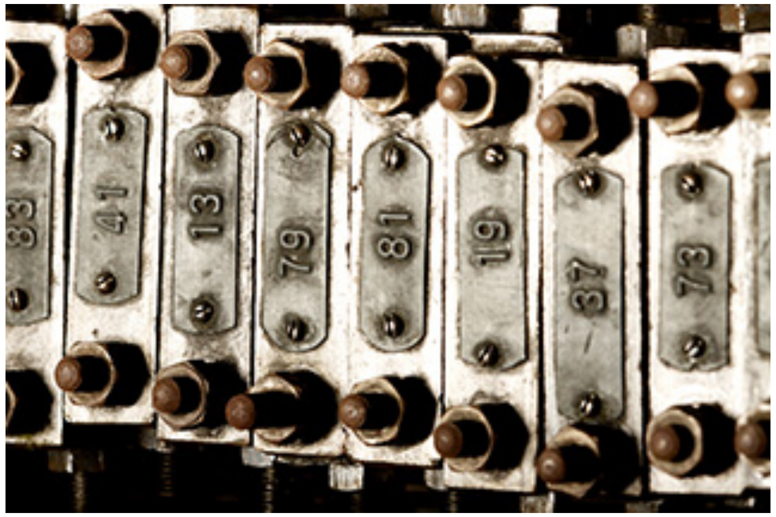
How to find the Modular Multiplicative Inverse of R mod Q

```
/*
 * mul_inv - Modular Multiplicative Inverse
 *
 * given:
 *       R    an integer
 *       Q    an integer > 0 and where gcd(R,Q) = 1
 *            (i.e., R and Q have no common prime factors)
 *
 * returns:
 *       S = Modular Multiplicative Inverse of R mod Q
 */
int
mul_inv(int R, int Q)
{
    int Q0 = Q, t, q;
    int x0 = 0, S = 1;
    if (Q == 1) return 1;
    while (R > 1) {
        q = R / Q;
        t = Q; Q = R % Q; R = t;
        t = x0; x0 = S - q * x0; S = t;
    }
    if (S < 0) S += Q0;
    return S;
}
```

How Deep Should we Sieve? A Practical Answer

- Sieve Riesel candidates until the time between sieve eliminations becomes longer than the time it takes to run a Riesel Test
  - When it takes longer for the sieve to turn a c[y] from 1 to 0, just do Riesel tests
- From experience: Sieve screening can eliminate >98.5% of candidates
- NOTE: If you happen to sieve for a small non-prime, you just waste time
  - You simply just won't eliminate c[y] values that haven't already been eliminated
- However the work to determine of Q is prime may waste too much time! So how much work is OK?
  - Start sieving array of odd Q values while simultaneously sieving Riesel candidates with Q's that remain standing
  - When the time it takes to eliminate an odd Q is longer than the time to do a single sieve of Riesel candidates, stop sieving Q values and just Sieve Riesel candidates

Image Credit:
Flickr user Marcin Wichary
Creative Commons License



Riesel Test Revisited

- Pick large n and start with a table of $h*2^n-1$ where $2*n < h <$ limit
  - Where limit is less than the word size (say $h < 2^{32}$ or $h < 2^{64}$)
  - Start with some practical range for h, say: $2*n < h < 16*n$

- Look for small factors by sieving, tossing out those with factors of small primes

- For each $h*2^n-1$ remaining, perform the Riesel test (almost as fast as the Lucas-Lehmer)
  - $U_2 = V(h)$ and $U_{x+1} \equiv U_x^2 - 2 \bmod h*2^n-1$ until U_n is computed
    - Pad U_x with leading 0's (at least p bits, more if required by Transform size)
    - Transform
    - Square each point
    - Inverse Transform
    - Round to integers and/or normalize as needed
    - Propagate carries
    - Subtract 2
    - Mod $h*2^n-1$ using a slightly more involved "shift and add" method
  - If $U_p \equiv 0$ then $h*2^n-1$ is prime, otherwise it is not prime

Cray Records Return - Amdahl 6 lesson ignored

- 1992: M(756839) 227 832 digits Slowinski & Gage using the Cray 2
- 1994: M(859433) 258 716 digits Slowinski & Gage using the Cray C90
- 1995: M(1257787) 378 632 digits Slowinski & Gage using the Cray T94

Slowinski, Cray T94, Gage

Image Credit:
Chris Caldwell

GIMPS Record Era - Just testing 2^n-1

- Great Internet Mersenne Prime Search - Testing only Mersenne numbers (test 2^n -1 only, not $h*2^n$ -1)
  - https://www.mersenne.org
  - Woltman, Kurowski, et al. using Crandall's Transform Square Algorithm

- 1996: M(1398269) 420 921 digits GIMPS + Armengaud
- 1997: M(2976221) 895 932 digits GIMPS + Spence
- 1998: M(3021377) 909 526 digits GIMPS + Clarkson
- 1999: M(6972593) 2 098 960 digits GIMPS + Hajratwala
  - $50 000 Cooperative Computing Award winner - 1st known million digit prime
- 2001: M(13466917) 4 053 946 digits GIMPS + Cameron
- 2003: M(20996011) 6 320 430 digits GIMPS + Shafer
- 2004: M(24036583) 7 235 733 digits GIMPS + Findley
- 2005: M(25964951) 7 816 230 digits GIMPS + Nowak
- 2005: M(30402457) 9 152 052 digits GIMPS + Cooper *
- 2006: M(32582657) 9 808 358 digits GIMPS + Cooper *
- 2008: M(43112609) 12 978 189 digits GIMPS + Smith
  - $100 000 Cooperative Computing Award winner - 1st known 10 million digit prime
- 2013: M(57885161) 17 425 170 digits GIMPS + Cooper *
- 2016: M(74207281) 22 338 618 digits GIMPS + Cooper *
- 2017: M(77232917) 23 249 425 digits GIMPS + Pace
- 2018: M(82589933) 24 862 048 digits GIMPS + Laroche

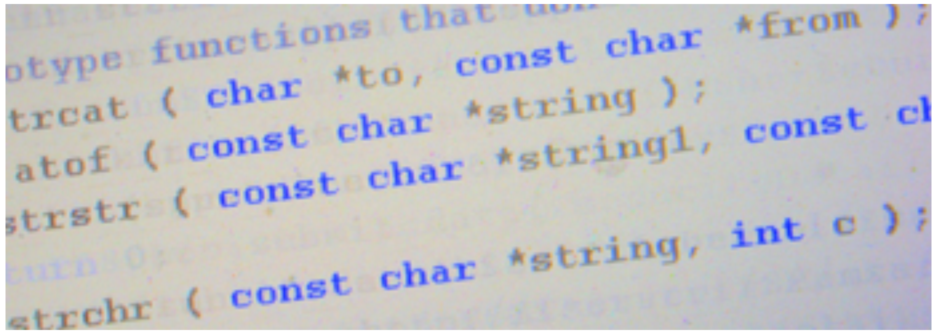


Image Credit:
Flickr user QualityFrog
Creative Commons License

* no relation to Simon :)

To be Fair to GIMPS

- GIMPS stands for Great Internet Mersenne Prime Search
- GIMPS is about searching for Mersenne Primes Only
- While testing Riesel numbers $h*2^n-1$ may be faster ...
  - Riesel testing is outside of their “charter” / purpose

Image Credit:
www.mersenne.org

2^{89}-1: Multiply+Add in Linear Time

- You can perform a n-bit multiply AND an n-bit add in 2*n clock cycles
  - If you have ⌈n/3⌉ simple 11-bit state machines
    - ⌈n/3⌉ mean n/3 rounded up to the next integer
  - See Knuth: Art of Computer Programming, Vol. 2, Section 4.3.3 E

Image Credit:
Dr. George Porter, UCSD

- Calculates u*v + q = a
  - The machine does a multiply and an add at the same time
- Can calculate U_n^2 - 2 in 2*n clock cycles
  - using ⌈n/3⌉ simple 11-bit state machines
- Hardware can do the slightly more involved “shift and add” in parallel
  - With the machine that is computing U_x^2 - 2
- Hardware can compute U_{n+1} in linear time!

11 bits of State in Each Machine

- Each state machine as 11 bits of state:
  - c, x0, y0, x1, y1, x, y, z0, z1, z2
    - All binary bits except for c which is a 2-bit binary value

c	x	y
x0		y0
x1		y1
z0	z1	z2

- 0th state machine is special:
  - 3, 0, 0, 0, 0, u(t), v(t), 0, 0, q(t)
  - The input bits are feed into x—>u(t),
    y—>v(t),
    z2—>q(t)
  - c is always 3, the other bits are always 0

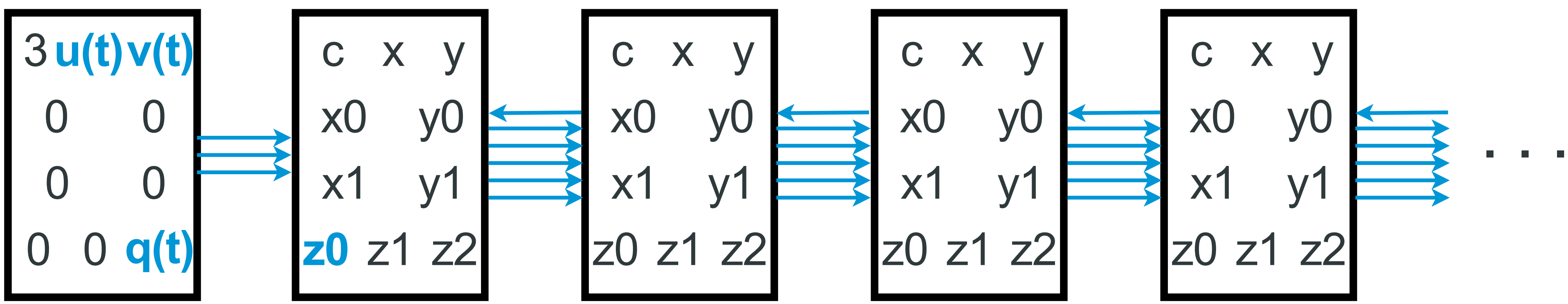
3	**u(t)**	**v(t)**
0		0
0		0
0	0	**q(t)**

- 1st state machine's z0 holds the answers at time t ≥ 1:
  - That z0 bit, at time t+1 holds bit t of the answer
    - answer bit of: a = u * v + q

c	x	y
x0		y0
x1		y1
z0	z1	z2

Build an Array of State Machines

- Assume a linear array of state machines S[0], S[1], S[2], …
  - If u, v, q are n-bits you need S[0] thru S[int(n/3)+1]
  - Initialize all state machine bits except S[0] are set to 0
- On each clock all state machines except the 0th:
  - Receive 1 bit from the right, 3 bits from the left, and copy over 2 bits from the left



- At clock t, feed in bit t of the input (u, v, q) into the 0th state machine's x, y, z2
  - When after the last input bit is feed, feed 0 bits
- Bit t of the answer is found in z0 of the 1st state machine at clock t+1

Simple State Machine Rules
These apply to all except left most machine

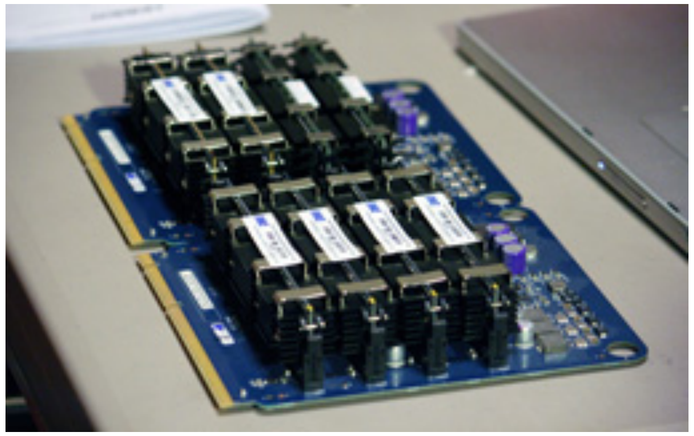


Image Credit:
Flickr user 37prime
Creative Commons License

- On each clock, state machines compute (z2, z1, z0):
  - Obtain z0 from right neighbor (call it z0Rr)
  - Obtain x, y, z2 from left neighbor (call them xL, yL, z2L)
  - If c == 0, (z2,z1,z0) = z0R + z1 + z2L + (xL & yL)
  - If c == 1, (z2,z1,z0) = z0R + z1 + z2L + (x0 & yL) + (xL & y0)
  - If c == 2, (z2,z1,z0) = z0R + z1 + z2L + (x0 & yL) + (xL & y0) + (x1 & y1)
  - If c == 3, (z2,z1,z0) = z0R + z1 + z2L + (x0 & yL) + (xL & y0) + (x1 & y) + (x & y1)
    - & means logical AND and + means add bits together into the 3 bit value (z2, z1, z0)
- On each clock, state machines copy from the left depending on c:
  - If c == 0, then copy x0,y0 from left neighbor into x0,y0
  - If c == 1, then copy x1,y1 from left neighbor into x1,y1
  - If c > 1, then copy x, y from left neighbor into x ,y
- On each clock, state machine increment c until it reaches 3:
  - c = minimum of (c+1, 3)
    - c is a 2-bit value

27.6 Million State Machine Array @ 100 GHz

- Multiply two 82.8 million bit numbers & add a 82.8 million bit digit number
  - In 0.00166 seconds!
- For Lucas-Lehmer or Riesel test:
  - Compute u*u + (-2)
    - Make u(t) = v(t) for all clocks
    - Add in the 2's compliment of -2
  - A simple front-end circuit can perform the "shift & add" for the mod
- Current record (as of 2019 Apr 16) is a 82 589 933 digit prime took 12 days
  - Used GIMPS code from http://www.mersenne.org
  - PC with an Intel i5-6600 CPU
- At 100 GHz, this machine could Riesel test a record sized prime in 37.9 hours!
  - More than 7.6 times faster per test!
  - It is certainly possible to build an ASIC with an even faster internal clock
  - Method increases linearly O(n) as the exponent grows
    - O(n) is MUCH better than O(n ln n), so for larger tests, this method will eventually become even faster than FFTs in software!
- Of course, you would need multiple units to be competitive with GIMPS

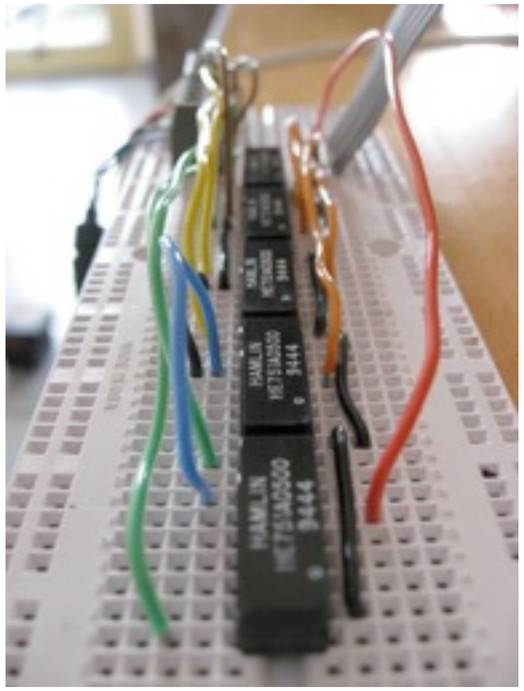


Image Credit:
Flickr user Quasimondo
Creative Commons License

2^{127}-1: Final Words and Some Encouragement

- Results (and records) goes to the first to calculate CORRECTLY ...
  - … not necessarily to the fastest tester

- A slow correct answer in infinitely better than a fast wrong answer!

- Compute smarter
  - You do NOT need to have the fastest machine to be the first to prove primality
    - My 8 world records related to prime numbers did NOT use the fastest machine

- Pre-mature optimization is the bane of a correctly running program
  - Write your comments first
  - Code something that works, updating comments as needed
    - Start that code running
  - Then incrementally improve the comments, improve the code & retest
    - Update the running code when you are confident it works

Image Credit:
Flickr user Kaeru
Creative Commons License

Test, test and TEST!

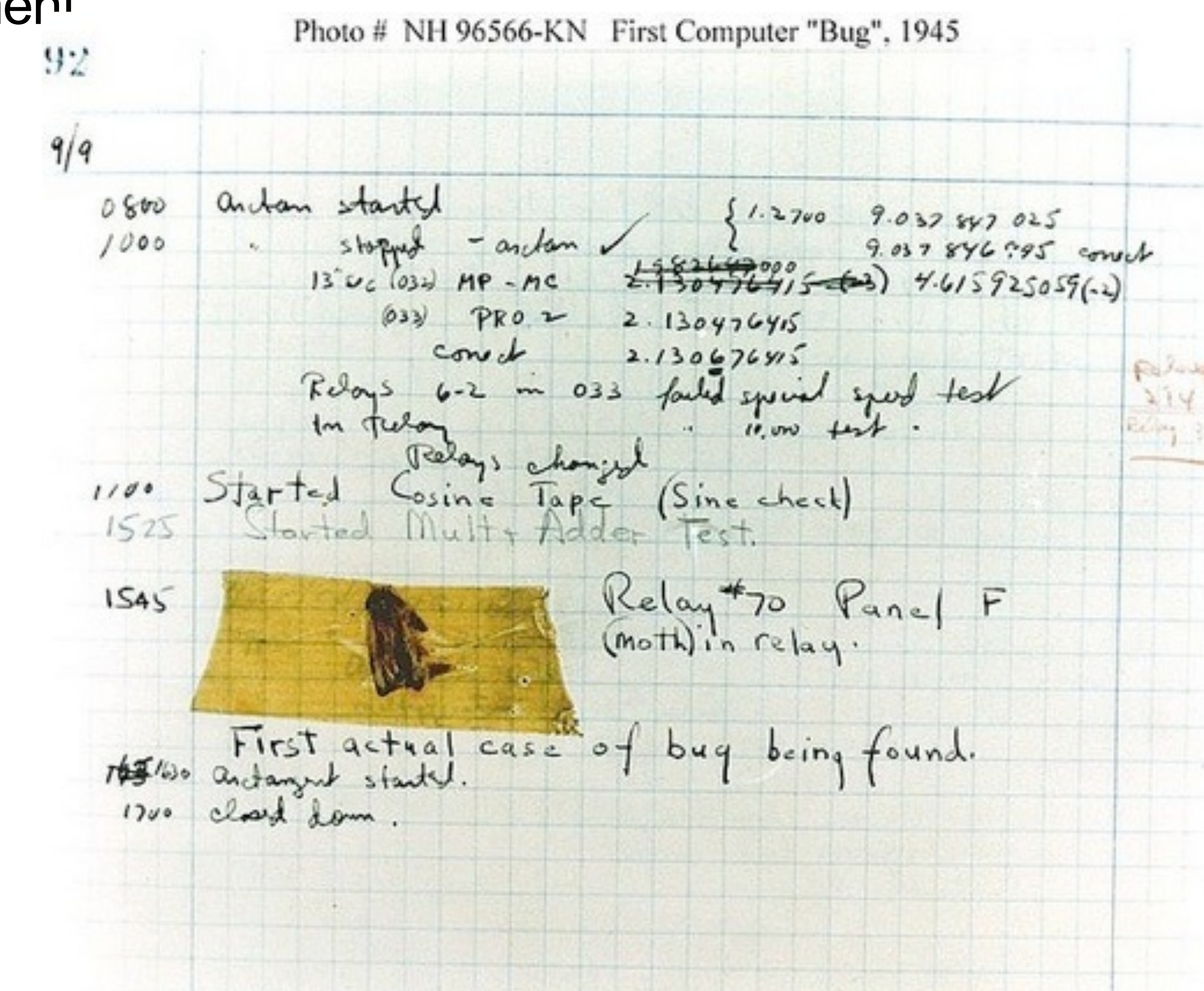
- Don’t trust the CPU / ALU
  - Put in checksums to sanity check square
  - Put in checksums to sanity check mod
  - 2001 Intel Celeron CPU had a Mean Time Between Errors (MTBE) of only 37 weeks!

- Don’t trust the Memory or Memory management
  - Uniquely mark pages in memory
    - Check for bad page fetches

- Don’t trust the system
  - Checkpoint in the middle of calculations
    - Restart program at last checkpoint
  - Backup! Test your backups!
  - Checksum code and data tables!

- Confirm all primality tests
  - After a number is tested, recheck the result!
    - Compare final U_x values
    - Test on different hardware
    - Better still, use different code to confirm test results

Image Credit:
Flickr user: flanker27
Creative Commons License



Most CPU cycles are NOT spent primality testing

- Expect to spend 1/3 or more of CPU time eliminating test candidates
- Expect to primality test each remaining test candidate at least twice
- Expect to spend 1/4 or more of CPU time in error checking
- Typically only 25% of CPU cycles will test a new prime candidate
  - ((100% - 1/3) / 2) * (1 - 1/4) = 25%
- You must verify (recheck your test) and have someone else independently verify (3rd test)
  - So plan on the time to test the number at least 3 times!
- While nothing is 100% error free:
  - Q: What is "mathematical truth"? A: The pragmatic answer:
    - Mathematical truth is something that the mathematical community has studied (peer reviewed) and has been shown to be true

Image Credit:
Flickr user benben
Creative Commons License

Find a new largest known prime (> $2^{82589933}$-1)

- Pick some n a bit larger than 82589933, say n = 82837504
  - If n as mostly 0 bits, the sieve (to eliminate potential candidates) goes faster
    - n = 100111100000000000000000000 in binary
  - Start with some practical range for h, say 165675008 < h < 1325400064
    - 2*82837504 < h < 16*82837504
- Look for small factors by sieving, tossing out those with factors that are not prime
  - Eliminate more than 98.5% of the candidates
    - before the sieve starts to take more time to eliminate a candidate than a prime test takes to run
- For each $h*2^{82837504}-1$ remaining, perform the Riesel test
  - $U_2 = V(h)$ and $U_{x+1} \equiv U_x^2 - 2 \bmod h*2^{82837504}-1$ until $U_{82837504}$ is computed
    - Pad U_x with leading 0's (at least p bits, more if required by Transform size)
    - Transform
    - Square each point
    - Inverse Transform
    - Round to integers
    - Propagate carries
    - Subtract 2
    - Mod $h*2^n-1$ using a slightly more involved "shift and add" method
  - If $U_p \equiv 0$ then $h*2^{82837504}-1$ is prime, otherwise it is not prime

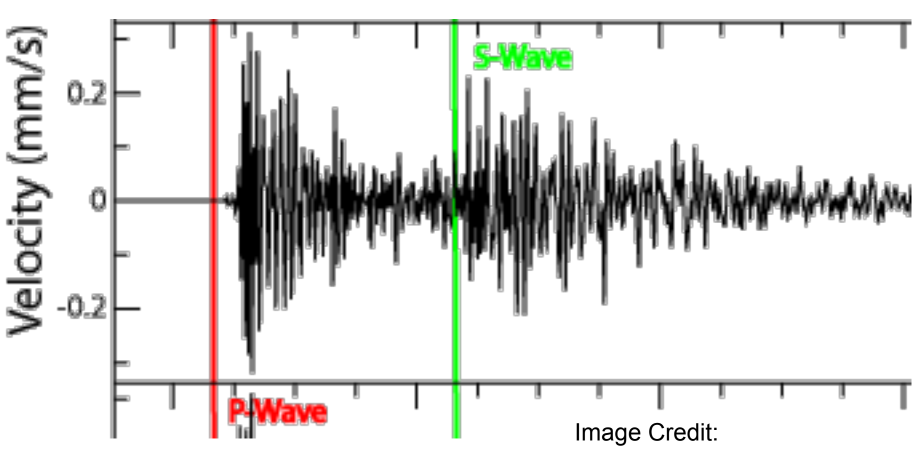


Image Credit:
Wikipedia
Creative Commons License

Riesel tests to find a new largest known prime

- Odds of $h*2^{82837504}-1$ prime ...
  - where 165675008 < h < 1325400064
    - where 2*82837504 < h < 16*82837504
- is about 1 in $2*\ln(h*2^{82837504}-1)$
  - About 1 in 2*(ln(h)+(82837504*ln(2)))
    - 1 in 107 569 027 for h near 114837203
    - 1 in 107 569 032 for h near 114837207

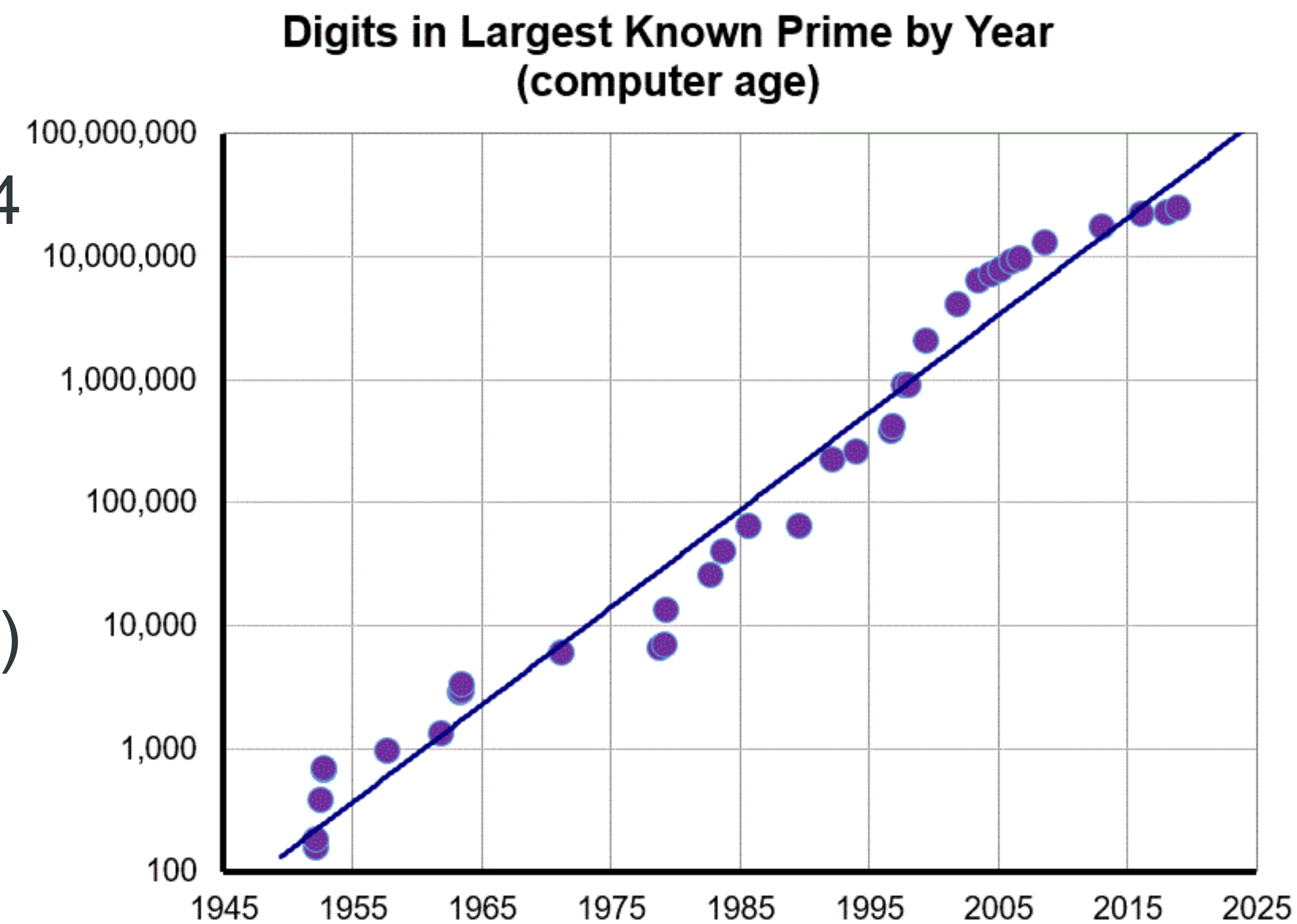


Image Credit:
Chris Caldwell

- Assume sieving eliminates >98.5% of the candidates
- Expect to perform about 1 613 535 Riesel tests of $h*2^{82837504}-1$

Finding a new largest known prime

- Could one of us, or a team among us find a new largest known prime?
  - Yes!

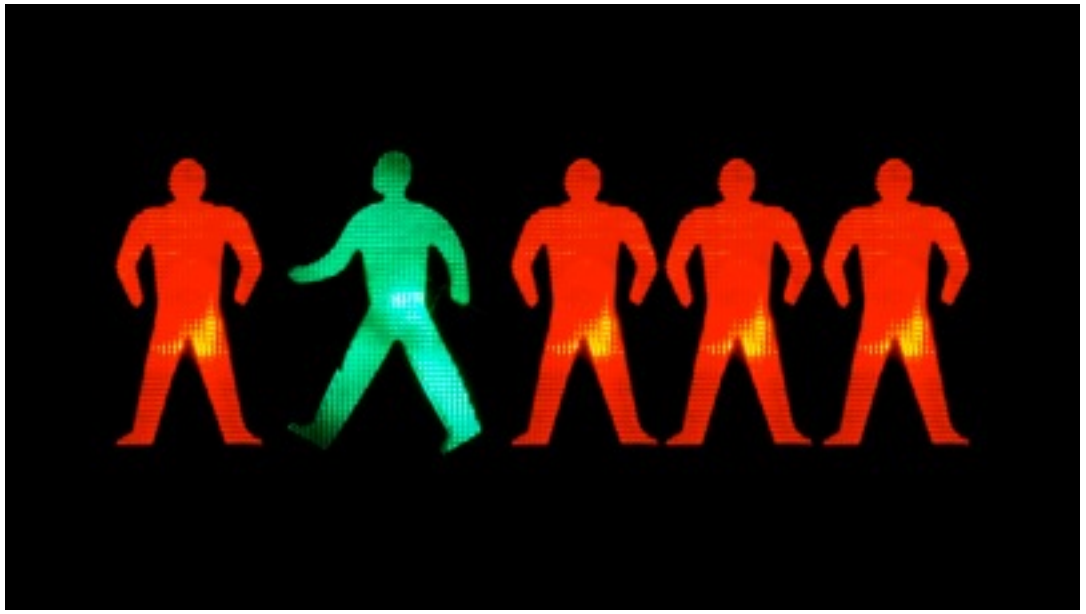


Image Credit:
Flickr user: My Buffo
Creative Commons License

- Focus on correctness of coding
  - Write code that runs correctly the first time
    - You don't have time to rerun!
- Focus on error correction and detection
  - Don't blindly trust hardware, firmware, operating system, system, drivers, compilers, etc.
  - Consider developing a tool to test newly manufactured hardware
  - Consider developing a tool that uses otherwise idle cycles
- Compute smarter
  - Hardware people: Consider building a fast multiply/add circuit
  - You do **NOT** need to use the fastest computer to gain a new world record!
  - Efficient networking between compute nodes will be key!

Don’t Become Discouraged

- As Dr. Lehmer was fond of saying:

“Happiness is just around the corner”

- Don’t get discouraged
  - You are searching on a many-sided polygon - you just have to find the right corner
- Work in a small team
  - Make use of complimentary strengths
- Write your own code where reasonable
  - Have different team members check each other’s code
  - When you use outside code
    - Always start with the source
    - Study their code, check for correctness, learn that code so well that you could write it yourself
      - You might end up re-writing it once you really understand what their code does

Image Credit:
Flickr user b3ni
Creative Commons License

And Above All ...

- Could someone in this room find a new largest known prime?
  - Yes!

- You **CAN** find a new largest known prime!
  - Never let someone tell you, you can't!

Edson Smith (Discoverer), Simon Cooper, Landon Curt Noll

Image Credit:
Flickr user quinnums
Creative Commons License

2^{521}-1: Resources

- The Prime Pages:
  - https://primes.utm.edu/
  - https://primes.utm.edu/notes/by_year.html#3
  - https://primes.utm.edu/prove/index.html
- Amdahl 6 method for implementing the Riesel test:
  - http://www.isthe.com/chongo/src/calc/lucas-calc
  - http://www.isthe.com/chongo/tech/comp/calc/index.html
- Transform resources and multiplication:
  - https://tonjanee.home.xs4all.nl/SSAdescription.pdf
  - http://www.flintlib.org
  - http://www.fftw.org/
  - https://en.wikipedia.org/wiki/Discrete_Fourier_transform#Polynomial_multiplication
  - http://www.apfloat.org/ntt.html
  - https://gmplib.org
  - https://arxiv.org/abs/0801.1416
  - https://cr.yp.to/f2mult/mateer-thesis.pdf
  - https://www.ams.org/journals/mcom/1994-62-205/S0025-5718-1994-1185244-1/S0025-5718-1994-1185244-1.pdf
  - https://www.daemonology.net/papers/fft.pdf

Image Credit:
Flickr user mr.beaver
Creative Commons License

$2^{521}-1$: Resources II

- Riesel primality test code:
  - https://github.com/arcetri/gmprime
  - https://github.com/arcetri/goprime
  - http://jpenne.free.fr/index2.html
- Verified primes of the form $h*2^n-1$
  - https://github.com/arcetri/verified-prime
- GIMPS:
  - https://www.mersenne.org
  - https://www.mersenne.org/download/
- On English names of large numbers:
  - http://www.isthe.com/chongo/tech/math/number/number.html
  - http://www.isthe.com/chongo/tech/math/number/howhigh.html
- Mersenne primes and the largest known Mersenne prime:
  - http://www.isthe.com/chongo/tech/math/prime/mersenne.html
  - http://www.isthe.com/chongo/tech/math/prime/mersenne.html#largest
- Cooperative Computing Award:
  - https://www.eff.org/awards/coop
  - https://www.eff.org/awards/coop/rules
- Obtain a recent edition of Knuth's:
  - The Art of Computer Programming, Volume 2, Semi-Numerical Algorithms: Especially Sections 4.3.1, 4.3.2, 4.3.3

Image Credit:
Flickr user: Lee Jordan
Creative Commons License

www.isthe.com/chongo/tech/math/prime/prime-tutorial.pdf
Questions for Part 2

Image Credit:
Flickr user bitzcelt
Creative Commons License

- 1) Why is it faster to search for a large prime of the form $h*2^n-1$ than 2^p-1?
  - Hint: See 69, 70
- 2) Assume M(92798969) is proven prime, what would a good choice of n (exponent of 2) to use when searching for a new largest known prime?
  - Hint: 92798969 in binary is: 101100001111111111111111001
  - Hint: See slides 92, 93, 94
- 3) How many state machines would it take to test $215802117*2^{77594624}-1$?
  - Hint: See slides 101, 105
- 4) What types of error checking could help correctly find a new largest known prime?
  - Hint: See slides 106, 107
- 5) Prove that $19*2^5-1 = 607$ is prime using the Riesel Test
  - Hint: U(2) = V(19) = 52
    - V(1) = 3 (although V(1) = 4 also works)
  - Hint: See slides 74, 75, 76, 80, 81

Bottom of talk.

Any Questions?

Thank you.

Landon Noll Touching the South Geographic Pole ± 1cm
Antarctica Expedition 2013

Landon Curt Noll
prime-tutorial-mail@asthe.com

http://www.isthe.com/chongo/tech/math/prime/prime-tutorial.pdf