Proving Ultimate Limitations on Computers

Stephen Cook

Presented at Classroom Adventures in Mathematics: Summer Institute

August 17, 2011

Computational Complexity

This is a branch of mathematics that studies how much time and memory are required for computers to solve various computational problems.

In particular we try to prove that certain problems can never be solved because they will take too much time, even on the fastest conceivable computers.

This is important, because present methods for computer security depend on unproven assumptions that computers cannot solve certain problems.

Search Problems

These are problems that have a very large (exponentially large) number of potential solutions. It may be hard to find any solution, but given a solution it is easy to verify that it is a solution.

Easy example: finding the square root of a large integer.

e.g. $\sqrt{9345249} = 3057$

It's easy to check that $3057 \times 3057 = 9345249$.

Multiplication is easy for computers using the elementary school algorithm, even if the numbers have thousands of digits.

But what about finding square roots?

Can a computer find the square root of a 60 digit number (to the nearest integer) using ‘blind search’?

Supose n has 60 digits, so

$$10^{59} \leq n < 10^{60}$$

and

$$10^{29} < \sqrt{n} < 10^{30}$$

so there are about 10^{30} possibilities for $\lfloor \sqrt{n} \rfloor$

How long would it take a computer to square each of these 10^{30} possibilities?

Assume that the computer can square a trillion $= 10^{12}$ numbers in a second.

There are

$$365 \times 24 \times 60 \times 60 = 31,536,000 < 10^8$$

seconds in a year, so the computer can square fewer than

$$10^8 \times 10^{12} = 10^{20}$$

numbers in a year. So the computer would take more than

$$10^{30}/10^{20} = 10^{10}$$

years (i.e. more than 10 billion years).

So computers cannot use blind search to find square roots of 60 digit numbers. But there are faster ways:

binary search
Home in on $\sqrt{n}$ by successive approximations.

Suppose $x_1^2 \le n < x_2^2$. Then

$x_1 \le \sqrt{n} < x_2$.

Let $a = (x_1 + x_2)/2$.

If $a^2 < n$ choose $y_1 = x_1$ and $y_2 = a$
Otherwise choose $y_1 = a$ and $y_2 = x_2$.

In either case $y_1 \le \sqrt{n} < y_2$

and $(x_2 - x_1) = \frac{1}{2}(y_2 - y_1)$

The error is cut in half in one step.

In k steps the error is divided by 2^k.

If n has 60 digits, how many steps does binary search take to find $\sqrt{n}$?

Recall that k steps of binary search reduces the error by a factor of 2^k.

$n < 10^{60}$, so $\sqrt{n} < 10^{30}$

Let $x_1 = 1$ and $x_2 = 10^{30}$, so $(x_2 - x_1) < 10^{30}$.

After 100 steps of binary search we can reduce the error from 10^{30} to 1.

This is because

$2^{100} = (2^{10})^{10} = (1024)^{10} > (10^3)^{10} = 10^{30}$

Thus $2^{100} > 10^{30}$.

100 steps takes a laptop a fraction of a second.

But Newton's Method is much faster even than binary search.

Compute successive approximations

$$x_1, x_2, x_3, \ldots$$

to $\sqrt{n}$.

$$x_{i+1} = \frac{1}{2}(x_i + \frac{n}{x_i})$$

Let x_1 be the first one or two digits in $\sqrt{n}$, obtained by observing the first two digits of n.

Then the number of significant digits of successive approximations $x_2, x_3, ...$ doubles with each iteration.

In the case n has 60 digits, the difference between x_6 and $\sqrt{n}$ is less than 1.

Summary so far

Using blind search to find the square root of a 60 digit number takes a computer 10 billion years, but Newton's method takes a fraction of a second.

Prime Factorization

Every integer greater than one can be written uniquely as a product of primes.
[Euclid]

$$\begin{aligned} 60 &= 2 \times 2 \times 3 \times 5 \\ 221 &= 13 \times 17 \\ 228947 &= 283 \times 809 \end{aligned}$$

Prime Factorization

Every integer greater than one can be written uniquely as a product of primes.
[Euclid]

$$\begin{aligned} 60 &= 2 \times 2 \times 3 \times 5 \\ 221 &= 13 \times 17 \\ 228947 &= 283 \times 809 \end{aligned}$$

Paradox: There are practical methods that allow computers to determine whether a number of several hundred digits is prime (brute force does NOT work).

But there is no known such practical method for finding the factors of such a number, even though it has been determined not to be prime.

In fact the security of the RSA encryption scheme depends on the assumption that the following problem cannot be solved by computers:

Given a number $N = P \times Q$ where P and Q are large random prime numbers (say 400 digits each) which are secretly generated:

Find P and Q.

How hard is it to factor the product of two large random primes?

Note that this is a Search Problem (because given P it is easy to find $Q = N/P$ and check that $N = P \times Q$).

Current (2010) World Record (Wikipedia):

A 232 digit number known as RSA-768 was factored in Dec, 2009 using hundreds of machines over 2 years.

$$\underbrace{1230\cdots3413}_{\text{232 digits}} = \underbrace{3347\cdots9489}_{\text{116 digits}} \times \underbrace{3673\cdots8917}_{\text{116 digits}}$$

Current (2010) World Record (Wikipedia):

A 232 digit number known as RSA-768 was factored in Dec, 2009 using hundreds of machines over 2 years.

$$\underbrace{1230\cdots3413}_{232\text{ digits}} = \underbrace{3347\cdots9489}_{116\text{ digits}} \times \underbrace{3673\cdots8917}_{116\text{ digits}}$$

Summary so far:

Blind search does not work for either finding square roots or factoring large numbers.

However there are fast methods for finding square roots, but no known fast methods for factoring.

Unfortunately we have been unable to prove that there are no fast methods for factoring.

Public Key Encryption

Diffie and Hellman 1976

RSA (Rivest, Shamir, Adleman) 1978

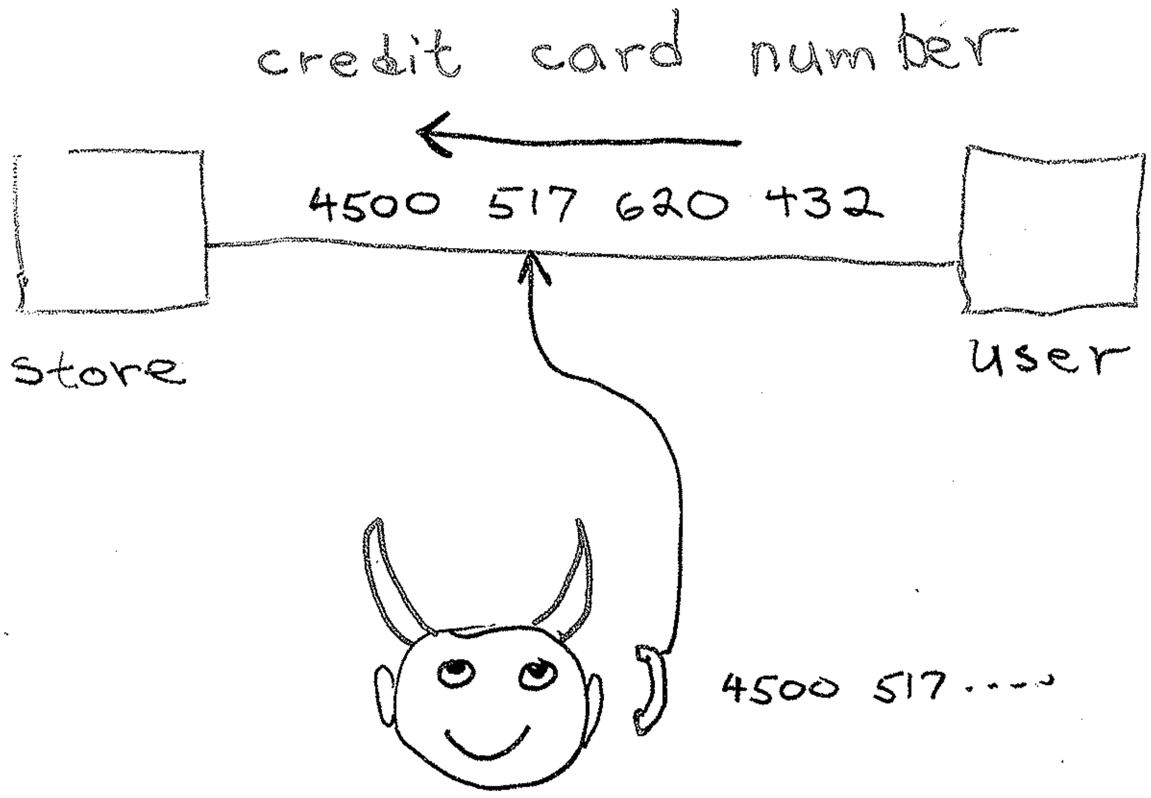
It is possible to securely send an encrypted message on a public channel without the parties agreeing ahead of time on a secret key.

But security depends on the assumption that a suitable problem (e.g. factoring) is intractable for computers.

Previously the parties needed to agree on a secret key.

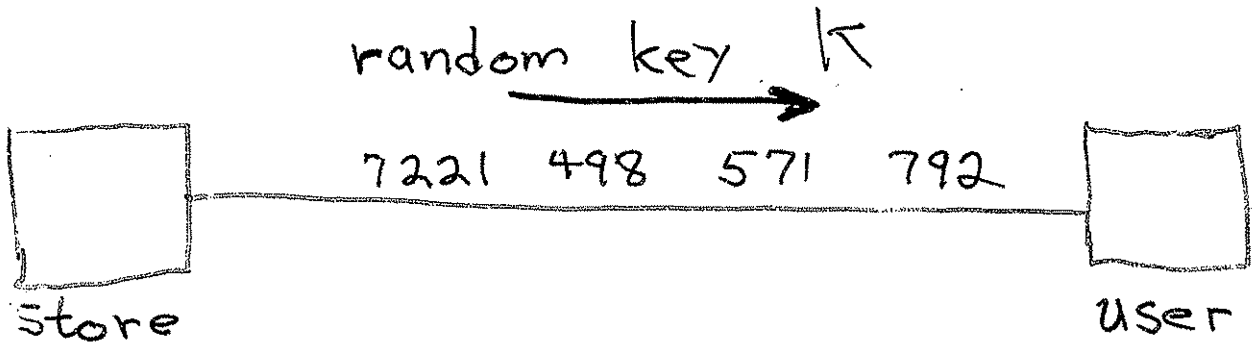
Secret Key Example:
During WWII the German Navy used Enigma Machines to code messages.
The Secret: The settings of the code disks needed to be agreed upon first.

Shopping by the Internet

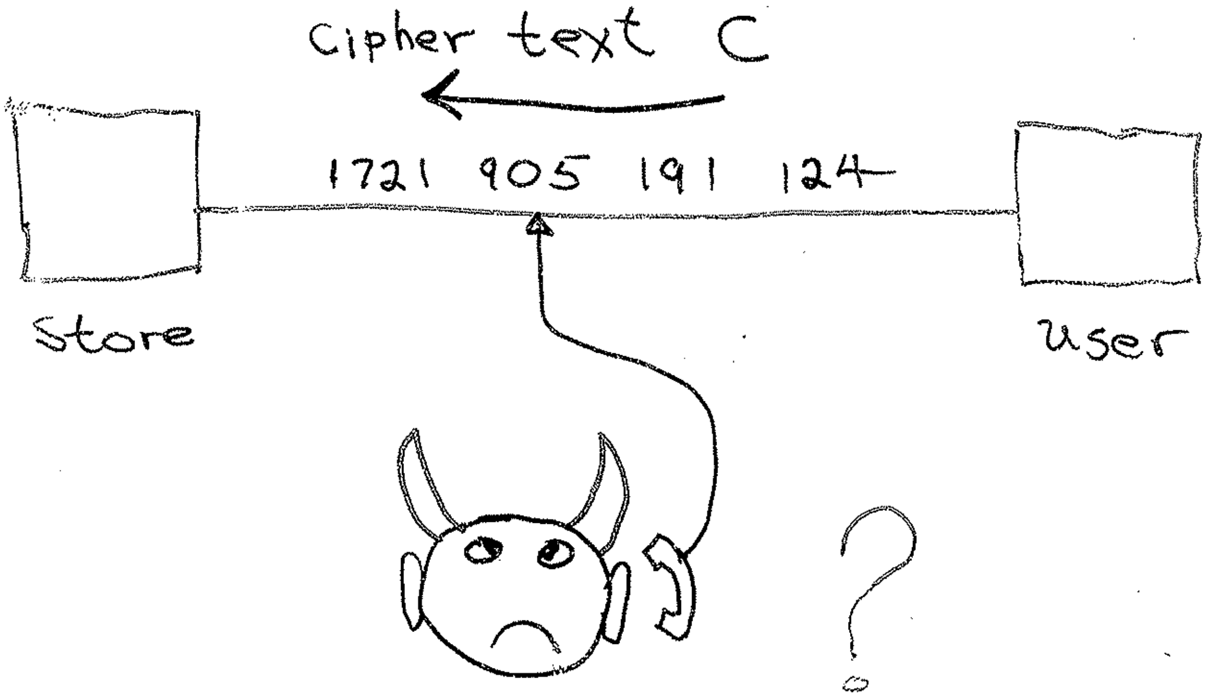


The eavesdropper is happy because she learns your credit card number.

Private Key Cryptosystem



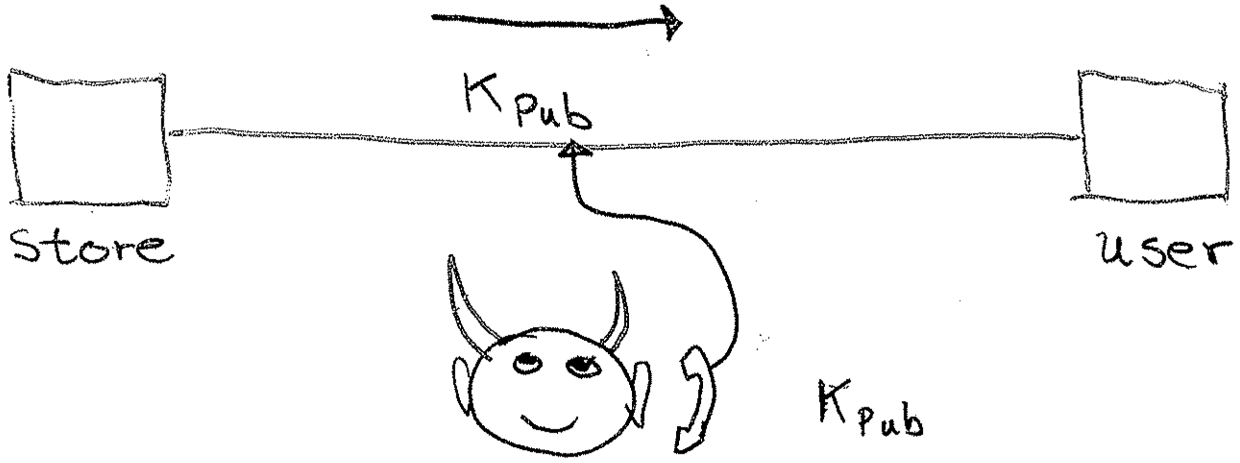
KEY:	K	= 7221 498 571 792
CHARGE #:	M	= 4500 517 620 432
CIPHER TEXT	C	= 1721 905 191 124

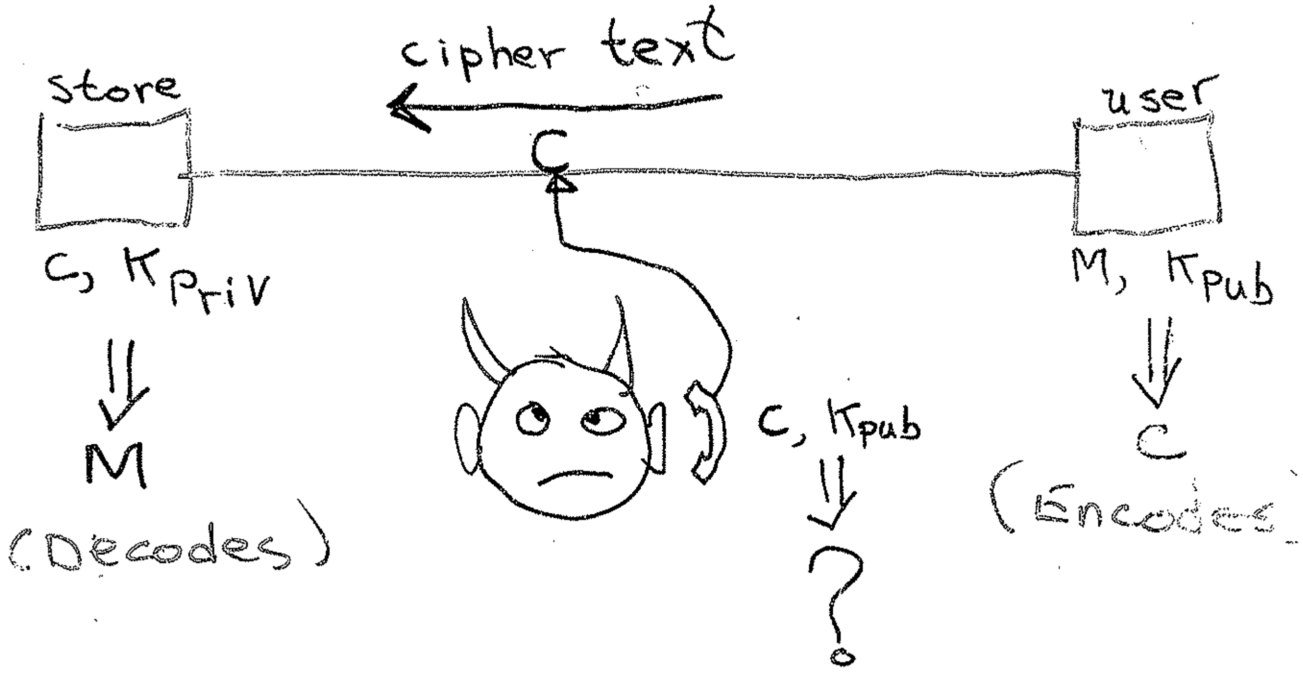


Public Key Cryptosystem

Requires: Public Key K_{Pub} (for encoding)

Private Key K_{Priv} (for decoding)





Prime Testing

Fermat's Little Theorem If p is a prime number and $1 < a < p$ then $a^{p-1} - 1$ is divisible by p.

Thus to show that p is NOT prime, it suffices to find a such that

$$1 < a < p \text{ and } a^{p-1} - 1 \text{ is NOT divisible by } p \quad (1)$$

For most numbers p, if p is not a prime, then at least half of the numbers a with $1 < a < p$ will detect that p is not prime.

Choose 100 random values of a. If even one of them satisfies (1) then you know p is not prime.

There are numbers called *Carmichal numbers* which are not prime but which cannot be detected by (1), but Carmichal numbers can be recognized.

Computational Complexity Theory

Classifies problems according to their computational difficulty.

The class P(Polynomial Time)
[Cobham, Edmonds, 1965]

P consists of all problems that have an efficient (e.g. n, n^2...) algorithm.
(n is the input length)

Examples in P:
• Addition, Multiplication, Square Roots
• Shortest Path (MapQuest)
• Network flows (Internet Routing)
• Pattern matching (Spell Checking, Text Processing)
• Fast Fourier Transform (Audio and Image processing, Oil exploration)
• Recognizing Prime Numbers [AgrawalKayal-Saxena 2002

.

.

.

The class NP(Nondeterministic Polynomial Time)

NP consists of all *search* problems whose solutions can be efficiently (i.e. in polynomial time) *verified*.

Examples in NP (besides everything in **P**)

- Prime Factorization
- Cracking cryptographic protocols.
- Scheduling delivery trucks, airlines,

hockey matches, exams, ... (‘Verify a solution’ means checking that it satisfies various constraints)

P versus **NP**

P: Problems for which solutions can be efficiently *found*

NP: Problems for which solutions can be efficiently *verified*

Conjecture: $\mathbf{P} \neq \mathbf{NP}$

Most computer scientists believe this conjecture.

But is seems to be incredibly hard to prove.

Why is proving P $\neq$ NP difficult?

One reason is that some search problems in **NP** (such as finding a square root) turn out to easy.
Here is another easy example.

Matching: Given a large group of people, we want to pair them up to work on projects. We know which pairs of people are compatible, and (if possible) we want to put them all in compatible pairs.

If there are 50 or more people, a brute force approach of trying all possible pairings would take billions of years.

However in 1965 Jack Edmonds found an ingenious efficient algorithm.
So this problem is in **P**.

How can we identify the hard **NP** problems?

NP-Complete Problems

These are the hardest **NP** problems.

A problem A is p-*reducible* to a problem B if an "oracle" for B can be used to efficiently solve A.

If A is p-*reducible* to B then any efficient procedure for solving B can be turned into an efficient procedure for A.

If A is p-*reducible* to B and B is in **P** then A is in **P**.

Definition: A problem B is **NP**-*complete* if B is in **NP** and every problem A in **NP** is p-reducible to B.

Theorem: If A is **NP**-complete and A is in **P** then **P** = **NP**.

To show **P** = **NP** you just need to find a fast (polynomial-time) algorithm for one **NP**-complete problem!!.

A great many (thousands) of problems have been shown to be **NP**-complete.

Most scheduling problems (delivery trucks, exams etc) are **NP**-complete.

Example: The following simple exam scheduling problem is **NP**-complete:

We need to schedule N examinations, and only three time slots are available. We are given a list of exam conflicts:
A conflict is a pair of exams that cannot be offered at the same time, because some student needs to take both of them.

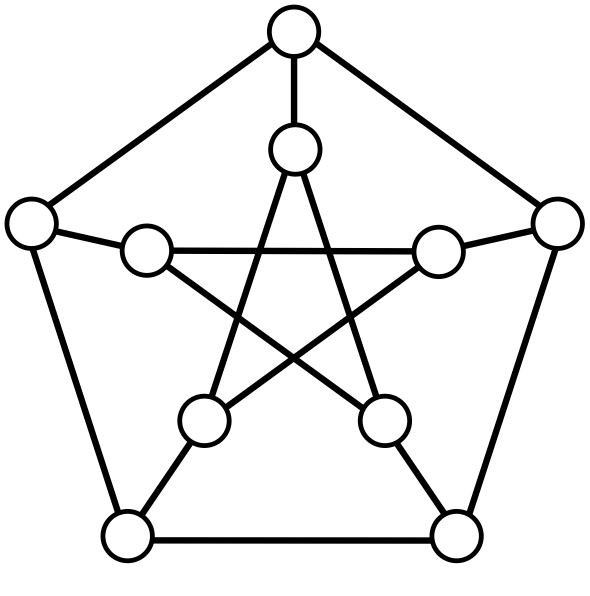
Problem: Is there a way of assigning each exam to one of the time slots T1, T2, T3, so that no two conflicting exams are assigned to the same time slot.

This problem is also known as graph 3-colourability.

Graph 3-colourability

A **graph** is a collection of nodes, with certain pairs of nodes connected by an edge.

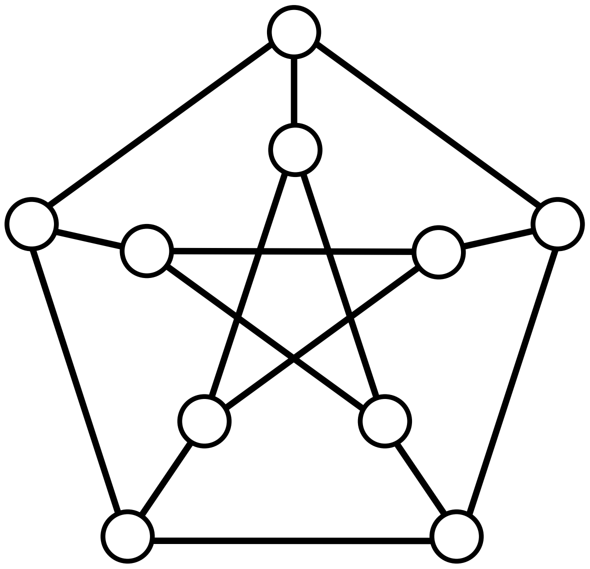
Problem: Given a graph, determine whether each node can be coloured red, blue, or green, so that the endpoints of each edge have different colours.

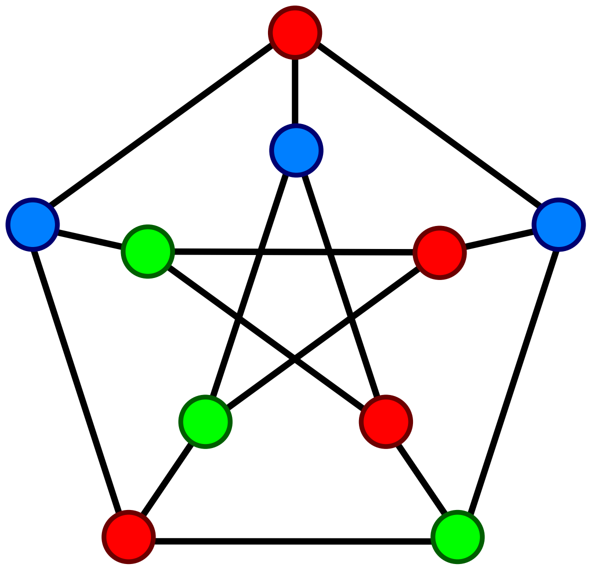


Graph 3-colourability

A **graph** is a collection of nodes, with certain pairs of nodes connected by an edge.

Problem: Given a graph, determine whether each node can be coloured red, blue, or green, so that the endpoints of each edge have different colours.
This problem is NP-complete.





Lance Fortnow has an article on **P** and **NP** in the September 2009 Communications of the ACM, in which he says

"The P versus NP problem has gone from an interesting problem related to logic to perhaps the most fundamental and important mathematical question of our time, whose importance only grows as computers become more powerful and widespread."

Clay Math Institute Millenium Problems: $ 1,000,000 each

- Birch and Swinnerton-Dyer Conjecture
- Hodge Conjecture
- Navier-Stokes Equations
- **P = NP?**
- Poincaré Conjecture (**Solved**)*
- Riemann Hypothesis
- Yang-Mills Theory

* Solved by Grigori Perelman 2003: Prize unclaimed