

Mathematics for programmers (early draft)

Dennis Yurichev <dennis(@)yurichev.com>

November 24, 2019

Contents

1	Prime numbers	1
1.1	Integer factorization	2
1.1.1	Using composite number as a container	2
1.1.2	Using composite number as a container (another example)	4
1.2	Coprime numbers	5
1.3	Semiprime numbers	6
1.4	How RSA ¹ works	6
1.4.1	Fermat little theorem	7
1.4.2	Euler's totient function	7
1.4.3	Euler's theorem	7
1.4.4	RSA example	7
1.4.5	So how it works?	9
1.4.6	Breaking RSA	10
1.4.7	The difference between my simplified example and a real RSA algorithm	10
1.4.8	The RSA signature	11
1.4.9	Hybrid cryptosystem	11
2	Modulo arithmetics	13
2.1	Quick introduction into modular arithmetic	13
2.1.1	Modular arithmetic on CPUs	14
2.1.2	Remainder of division by modulo 2^n	15
2.1.3	Getting random numbers	16
2.2	Modulo inverse, part I	17
2.2.1	No remainder?	18
2.3	Modulo inverse, part II	19
2.4	Reversible linear congruential generator	20
2.5	Getting magic number using extended Euclidean algorithm	22
3	Probability	25
3.1	Text strings right in the middle of compressed data	25
3.2	Autocomplete using Markov chains	27
3.2.1	Dissociated press	27
3.2.2	Autocomplete	32
3.2.3	Further work	40
3.2.4	The files	41
3.2.5	Read more	41
3.3	random.choices() in Python 3	41

¹Rivest-Shamir-Adleman cryptosystem

4	Combinatorics	45
4.1	Soldering a headphones cable	45
4.2	Vehicle license plate	46
4.3	Forgotten password	47
4.4	Executable file watermarking/steganography using Lehmer code and factorial number system	53
4.5	De Bruijn sequences; leading/trailing zero bits counting	58
4.5.1	Introduction	58
4.5.2	Trailing zero bits counting	59
4.5.3	Leading zero bits counting	63
4.5.4	Performance	64
4.5.5	Applications	64
4.5.6	Generation of De Bruijn sequences	64
4.5.7	Other articles	64
5	Galois Fields, GF(2) and yet another explanation of CRC²	65
5.1	What is wrong with checksum?	65
5.2	Division by prime	65
5.3	(Binary) long division	66
5.4	(Binary) long division, version 2	67
5.5	Shortest possible introduction into GF(2)	69
5.6	CRC32	70
5.7	Rationale	71
5.8	Further reading	72
6	Logarithms	73
6.1	Introduction	73
6.1.1	Children's approach	73
6.1.2	Scientists' and engineers' approach	74
6.2	Logarithmic scale	74
6.2.1	In human perception	74
6.2.2	In electronics engineering	75
6.2.3	In IT	76
6.2.4	Web 2.0	77
6.3	Multiplication and division using addition and subtraction	77
6.3.1	Logarithmic slide rule	77
6.3.2	Logarithmic tables	79
6.3.3	Working with very small and very large numbers	80
6.3.4	IEEE 754: adding and subtracting exponents	82
6.4	Exponentiation	83
6.5	Square root	83
6.6	Base conversion	84
6.7	Binary logarithm	85
6.7.1	Denoting a number of bits for some value	85
6.7.2	Calculating binary logarithm	87
6.7.3	$O(\log n)$ time complexity	88
6.8	Common (base 10) logarithms	89
6.9	Natural logarithm	90
6.9.1	Savings account in your bank	90
6.9.2	Exponential decay	92

²Cyclic redundancy check

7	Symbolic computation	97
7.1	Rational data type	98
8	Graph theory	101
8.1	Clique in graph theory	101
8.1.1	Social graph: simple example	101
8.1.2	Social graph: IRC network	102
8.1.3	Attempt to find communities in IRC social graph	103
8.1.4	Social graph: social networks	107
8.1.5	Links graph: Wikipedia	107
8.1.6	Social graph: LiveJournal spammers	108
8.1.7	Links graph: link farms	108
8.1.8	Further reading	108
9	GCD and LCM	109
9.1	GCD	109
9.2	Oculus VR Flicks and GCD	111
9.3	LCM	112
9.3.1	File copying routine	113
10	Linear algebra	115
10.1	Gaussian elimination: Kirchhoff’s circuit laws	115
10.1.1	Gaussian elimination	116
11	Acronyms used	121

Thanks

Thanks to those who helped: Slava “Avid” Kazakov, Serhiy Matviychuk.

Chapter 1

Prime numbers

Prime numbers are the numbers which has no divisors except itself and 1. This can be represented graphically.

Let's take 9 balls or some other objects. 9 balls can be arranged into rectangle:

```
ooo
ooo
ooo
```

So are 12 balls:

```
oooo
oooo
oooo
```

Or:

```
ooo
ooo
ooo
ooo
```

So 9 and 12 are not prime numbers. 7 is prime number:

```
ooooooo
```

Or:

```
o
o
o
```

- o
- o
- o
- o

It's not possible to form a rectangle using 7 balls, or 11 balls or any other prime number.

The fact that balls can be arranged into rectangle shows that the number can be divided by the number which is represented by height and width of rectangle. Balls of prime number can be arranged vertically or horizontally, meaning, there are only two divisors: 1 and the prime number itself.

1.1 Integer factorization

Natural number can be either prime or composite number. Composite number is a number which can be broken up by product of prime numbers. Let's take 100. It's not a prime number.

By the *fundamental theorem of arithmetic*, any number can be represented as product of prime numbers, in only one single way.

So the *composite number* phrase means that the number is *composed* of prime numbers.

Let's factor 100 in Wolfram Mathematica:

Listing 1.1: Wolfram Mathematica

```
In [] := FactorInteger[100]
Out [] = {{2, 2}, {5, 2}}
```

This mean that 100 can be constructed using 2 and 5 prime numbers ($2^2 \cdot 5^2$):

Listing 1.2: Wolfram Mathematica

```
In [] := 2^2*5^2
Out [] = 100
```

1.1.1 Using composite number as a container

Even more than that, it's possible to encode some information in prime numbers using factoring. Let's say, we would encode the "Hello" text string. First, let's find ASCII codes of each character in the string:

Listing 1.3: Wolfram Mathematica

```
In [] := ToCharacterCode["Hello"]
Out [] = {72, 101, 108, 108, 111}
```

Let's find a first 5 prime numbers, each number for each character:

Listing 1.4: Wolfram Mathematica

```
In []:= Map[Prime[#] &, Range[5]]
Out []= {2, 3, 5, 7, 11}
```

Build a huge number using prime numbers as bases and ASCII codes as exponents, then get a product of all them ($2^{72} \cdot 3^{101} \cdot 5^{108} \cdot 7^{108} \cdot 11^{111}$):

Listing 1.5: Wolfram Mathematica

```
In []:= tmp = 2^72*3^101*5^108*7^108*11^111
Out []= \
1649465578065933994718255257642275679479006861206428830641826551739434\
9344066214616222018844835866267141943107823334187149334898562231349428\
5708281252457614466981636618940129599457183300076472809826225406689893\
5837524252859632074600687844523389231265776082000229507684707641601562\
50000000000000000000000000000000000000000000000000000000000000000000\
000
```

It's a big number, but Wolfram Mathematica is able to factor it back:

Listing 1.6: Wolfram Mathematica

```
In []:= FactorInteger[tmp]
Out []= {{2, 72}, {3, 101}, {5, 108}, {7, 108}, {11, 111}}
```

A first number in each pair is prime number and the second is exponent. Get the text string back:

Listing 1.7: Wolfram Mathematica

```
In []:= FromCharacterCode[Map#[[2]] &, tmp]]
Out []= "Hello"
```

That allows to have some fun. Let's add exclamation point to the end of string by manipulating only the *big number*. ASCII code of exclamation point is 33. The next prime number after 11 is 13. So add it (by multiplying by 13^{33}):

Listing 1.8: Wolfram Mathematica

```
In []:= tmp = tmp*13^33
Out []= \
9494539005656577744061615691556750598033024729435332190254469113536733\
9032823543118405499931761589928052797992206631285822671397023217541663\
5920521812548793623881568510051214975599793760307837993570818136014139\
9497958680836430182400405525832564875875193876694267121604212637095253\
072514545272861141711473464965820312500000000000000000000000000000\
00000000000000000000000000000000000000000000000000000000000000000000
```

So we got new number. Let's factor it back and decode:

Listing 1.9: Wolfram Mathematica

```
In []:= factored = FactorInteger[tmp]
Out []= {{2, 72}, {3, 101}, {5, 108}, {7, 108}, {11, 111}, {13, 33}}

In []:= FromCharacterCode[Map#[[2]] &, factored]]
Out []= "Hello!"
```

Wow, that works. Will it be possible to remove one 'l' character from the string at the third position? The 'l' has the ASCII code of 108 and it is exponent for two prime numbers in our expression: 5 (first 'l') and 7 (second 'l').

To knock out the character, we divide the *big number* by the corresponding prime number with the exponent of 108 (divide by 5^{108}):

Listing 1.10: Wolfram Mathematica

```
In []:= tmp = tmp/5^108
Out []= \
3081154065769189664244341216329094565621009415122099836376732969546063\
1079164051611808432546107410277501678916823138724630810880390384343750\
1196528030610615786507542545262118293483878711112407171889948257893463\
8494741216231004109210436295299274515484540190050751059821909485854359\
9630924207126074604240892753608704

In []:= factored = FactorInteger[tmp]
Out []= {{2, 72}, {3, 101}, {7, 108}, {11, 111}, {13, 33}}

In []:= FromCharacterCode[Map#[[2]] &, factored]]
Out []= "Helo!"
```

1.1.2 Using composite number as a container (another example)

Let's say, the initial *container* number is 1. Let's increment the number at the second position within it by multiplying by the first prime number (3):

Listing 1.11: Wolfram Mathematica

```
In []:= tmp = 1*3
Out []= 3
```

Then let's set the number at fourth position to 123. The fourth prime number is 7 (the percent sign in Mathematica denotes the last result):

Listing 1.12: Wolfram Mathematica

```
In []:= tmp = tmp*7^123
Out []= 26557071110804040505330743411815438275701018334410643480070773\
5780279761186999642944265644421128096489029
```

Then let's set the number at fifth position to 456. The fifth prime number is 11:

Listing 1.13: Wolfram Mathematica

```
In []:= tmp = tmp*11^456
Out []= 19917948660639605307938425372554395433764512138284060223646519\
1257621966825293455339751080188144910510813322192288287162176499976800\
9147068591160798243308591883294649069355015558472564457422829073938118\
4396204999051856879940101934339913600942451006747982291524910653185084\
4057972896537894301213735252418639782974592077393028390193060138936503\
0125465578958567377627063815620261557939484036628536230966158222960762\
8509899641574477547658142457598168497006309608599830554758951672484533\
9216105863754463712957143732563375990198901073698626584903164027850451\
8825659837114080589573087269
```

Then let's decrement the number at fourth position, the fourth prime number is 7:

Listing 1.14: Wolfram Mathematica

```
In []:= tmp = tmp/7
Out []= 28454212372342293297054893389363422048235017340405800319495027\
3225174238321847793342501543125921300729733317417554695945966428538287\
0210097987372568919012274118992355813364307940675092082032612962768740\
6280292855788366971343002763342733715632072866782831845035586647407263\
4368532709339849001733907503455199689963702967704326271704371627052147\
1607807969940810539467234022314659368484977195183623187094511747086804\
0728428059392110782368774939425954995723299440856900792512788103549334\
1737294091077805304224491046519108557427001533855180835575948611214931\
260808548159154369939012467
```

Let's factor the composite number and get all the numbers we set inside *container* (1, 122, 456):

Listing 1.15: Wolfram Mathematica

```
In []:= FactorInteger[tmp]
Out []= {{3, 1}, {7, 122}, {11, 456}}
```

So the resulting number has $3 \cdot 7^{122} \cdot 11^{456}$ form at the end.

This is somewhat wasteful way to store the numbers, but out of curiosity: since there are infinite number of prime numbers and so any number of infinitely big numbers can be *stored* within one (although huge) composite number.

1.2 Coprime numbers

Coprime numbers are the 2 or more numbers which don't have any common divisors. In mathematical lingo, the [GCD](#)¹ of all coprime numbers is 1.

¹Greatest Common Divisor

3 and 5 are coprimes. So are 7 and 10. So are 4, 5 and 9.

Coprime numbers are the numerator and denominator in fraction which cannot be reduced further (irreducible fraction). For example, $\frac{130}{14}$ is $\frac{65}{7}$ after reduction (or simplification), 65 and 7 are coprime to each other, but 130 and 14 are not (they have 2 as common divisor).

One application of coprime numbers in engineering is to make number of cogs on cogwheel and number of chain elements on chain to be coprimes. Let's imagine bike cogwheels and chain:

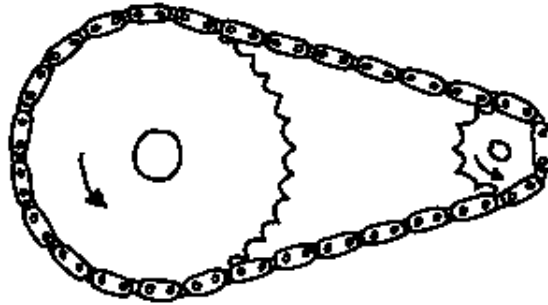


Figure 1.1: The picture was taken from www.mechanical-toys.com

If you choose 5 as number of cogs on one of cogwheel and you have 10 or 15 or 20 chain elements, each cog on cogwheel will meet some set of the same chain elements. For example, if there are 5 cogs on cogwheel and 20 chain elements, each cog will meet only 4 chain elements and vice versa: each chain element has its *own* cog on cogwheel. This is bad because both cogwheel and chain will run out slightly faster than if each cog would interlock every chain elements at some point. To reach this, number of cogs and chain elements could be coprime numbers, like 5 and 11, or 5 and 23. That will make each chain element interlock each cog evenly, which is better.

1.3 Semiprime numbers

Semiprime is a product of two prime numbers.

An interesting property of semiprime:

In 1974 the Arecibo message was sent with a radio signal aimed at a star cluster . It consisted of 1679 binary digits intended to be interpreted as a 23×73 bitmap image. The number 1679 = 23×73 was chosen because it is a semiprime and therefore can only be broken down into 23 rows and 73 columns, or 73 rows and 23 columns.

(<https://en.wikipedia.org/wiki/Semiprime>)

1.4 How RSA works

RSA public-key cryptosystem (named after its inventors: Ron Rivest, Adi Shamir and Leonard Adleman) is the most used asymmetric public-private key cryptosystem.

1.4.1 Fermat little theorem

Fermat little theorem states that if p is prime, this congruence is valid for any a in the environment of modulo arithmetic of base p :

$$a^{p-1} \equiv 1 \pmod{p}.$$

There are proofs, which are, perhaps, too tricky for this article which is intended for beginners. So far, you can just take it as granted.

This theorem may be used to sieve prime numbers. So you take, for example, 10 and test it. Let's take some random a value (123) (Wolfram Mathematica):

Listing 1.16: Wolfram Mathematica

```
In [] := Mod [123^(10 - 1), 10]
Out [] = 3
```

We've got 3, which is not 1, indicating the 10 is not prime. On the other hand, 11 is prime:

Listing 1.17: Wolfram Mathematica

```
In [] := Mod [123^(11 - 1), 11]
Out [] = 1
```

This method is not perfect, some composite p numbers can lead to 1, for example $p=1105$, but can be used as a method to sieve vast amount of prime numbers candidates.

1.4.2 Euler's totient function

It is a number of coprime numbers under some n . Denoted as $\phi(n)$, pronounced as *phi*. For the sake of simplification, you may just keep in mind that if $n = pq$ (i.e., product of two prime numbers), $\varphi(pq) = (p - 1)(q - 1)$. This is true for RSA environment.

1.4.3 Euler's theorem

Euler's theorem is a generalization of Fermat little theorem. It states:

$$a^{\varphi(n)} \equiv 1 \pmod{n}$$

But again, for the sake of simplification, we may keep in mind that Euler's theorem in the RSA environment is this:

$$a^{(p-1)(q-1)} \equiv 1 \pmod{n}$$

... where $n = pq$ and both p and q are prime numbers.

This theorem is central to RSA algorithm.

1.4.4 RSA example

There are *The Sender* and *The Receiver*. *The Receiver* generates two big prime numbers (p and q) and publishes its product ($n = pq$). Both p and q are kept secret.

For the illustration, let's randomly pick p and q among the first 50 prime numbers in Wolfram Mathematica:

Listing 1.18: Wolfram Mathematica

```
In [] := p = Prime[RandomInteger[50]]
Out [] = 89

In [] := q = Prime[RandomInteger[50]]
Out [] = 43

In [] := n = p*q
Out [] = 3827
```

3827 is published as public key, named *public key modulus* or *modulo*. It is semiprime. There is also public key exponent e , which is not secret, is often 65537, but we will use 17 to keep all results tiny.

Now *The Sender* wants to send a message (123 number) to *The Receiver* and he/she uses one-way function:

Listing 1.19: Wolfram Mathematica

```
In [] := e = 17
Out [] = 17

In [] := encrypted = Mod[123^e, n]
Out [] = 3060
```

3060 is encrypted message, which can be decrypted only using p and q values separately. This is one-way function, because only part of exponentiation result is left. One and important consequence is that even *The Sender* can't decrypt it. This is why you can encrypt a piece of text in PGP/GnuPG to someone using his/her public key, but can't decrypt it. Perhaps, that's how CryptoLockers works, making impossible to decrypt the files.

To recover message (123), p and q values must be known.

First, we get the result of Euler's totient function $(p - 1)(q - 1)$ (this is the point where p and q values are needed):

Listing 1.20: Wolfram Mathematica

```
In [] := totient = (p - 1)*(q - 1)
Out [] = 3696
```

Now we calculating decrypting exponent using multiplicative modulo inverse (multiplicative inverse was also described in here (2) ($e^{-1} \pmod{totient = (p - q)(q - 1)}$)):

Listing 1.21: Wolfram Mathematica

```
In [] := d = PowerMod[e, -1, totient]
Out [] = 2609
```

Now decrypt the message:

Listing 1.22: Wolfram Mathematica

```
In [18] := Mod[encrypted^d, n]
Out [18]= 123
```

So the d exponent forms another one-way function, restoring the work of what was done during encryption.

1.4.5 So how it works?

It works, because e and d exponents are reciprocal to each other by modulo $totient = (p - 1)(q - 1)$:

Listing 1.23: Wolfram Mathematica

```
In [] := Mod[e*d, totient] (* check *)
Out [] = 1
```

This allows...

$$m^{ed} = m \pmod{n} \tag{1.1}$$

Or in Mathematica:

Listing 1.24: Wolfram Mathematica

```
In [] := Mod[123^(e*d), n]
Out [] = 123
```

So the encryption process is $m^e \pmod{n}$, decryption is $(m^e)^d = m \pmod{n}$.

To prove congruence 1.1, we first should prove the following congruence:

$$ed \equiv 1 \pmod{((p - 1)(q - 1))}$$

... using modular arithmetic rules, it can be rewritten as:

$$ed = 1 + h(p - 1)(q - 1)$$

h is some unknown number which is present here because it's not known how many times the final result was rounded while exponentiation (this is modulo arithmetic after all).

So $m^{ed} = m \pmod{n}$ can be rewritten as:

$$m^{1+h((p-1)(q-1))} \equiv m \pmod{n}$$

...and then to:

$$m(m^{(p-1)(q-1)})^h \equiv m \pmod{n}$$

The last expression can be simplified using Euler's theorem (stating that $a^{(p-1)(q-1)} \equiv 1 \pmod{n}$). The result is:

$$m(1)^h \equiv m \pmod{n}$$

... or just:

$$m \equiv m \pmod{n}$$

1.4.6 Breaking RSA

We can try to factor n semiprime (or RSA modulus) in Mathematica:

Listing 1.25: Wolfram Mathematica

```
In [] := FactorInteger[n]
Out [] = {{43, 1}, {89, 1}}
```

And we getting correct p and q , but this is possible only for small values. When you use some big ones, factorizing is extremely slow, making RSA unbreakable, if implemented correctly.

The bigger p , q and n numbers, the harder to factorize n , so the bigger keys in bits are, the harder it to break.

1.4.7 The difference between my simplified example and a real RSA algorithm

In my example, public key is $n = pq$ (product) and secret key are p and q values stored separately. This is not very efficient, to calculate totient and decrypting exponent each time. So in practice, a public key is n and e , and a secret key is at least n and d , and d is stored in secret key precomputed.

For example, here is my PGP public key²:

```
dennis@...:~$ gpg --export-options export-reset-subkey-passwd --export-secret-
  subkeys 0x3B262349\! | pgpdump
Old: Secret Key Packet(tag 5)(533 bytes)
  Ver 4 - new
  Public key creation time - Tue Jun 30 02:08:38 EEST 2015
  Pub alg - RSA Encrypt or Sign(pub 1)
  RSA n(4096 bits) - ...
  RSA e(17 bits) - ...
...
```

... so there are available openly big (4096 bits) n and e (17 bits).

And here is my PGP secret key:

```
dennis@...:~$ gpg --export-options export-reset-subkey-passwd --export-secret-
  subkeys 0x55B5C64F\! | pgpdump
gpg: about to export an unprotected subkey

You need a passphrase to unlock the secret key for
user: "Dennis Yurichev <dennis@yurichev.com>"
4096-bit RSA key, ID 55B5C64F, created 2015-06-29

gpg: gpg-agent is not available in this session
Old: Secret Key Packet(tag 5)(533 bytes)
  Ver 4 - new
  Public key creation time - Tue Jun 30 02:08:38 EEST 2015
```

²<https://yurichev.com/pgp.html>


```

    Pub alg - RSA Encrypt or Sign(pub 1)
    RSA n(4096 bits) - ...
    RSA e(17 bits) - ...
...
Old: Secret Subkey Packet(tag 7)(1816 bytes)
    Ver 4 - new
    Public key creation time - Tue Jun 30 02:08:38 EEST 2015
    Pub alg - RSA Encrypt or Sign(pub 1)
    RSA n(4096 bits) - ...
    RSA e(17 bits) - ...
    RSA d(4093 bits) - ...
    RSA p(2048 bits) - ...
    RSA q(2048 bits) - ...
    RSA u(2048 bits) - ...
    Checksum - 94 53
...

```

... it has all variables stored in the file, including d , p and q .

1.4.8 The RSA signature

It's possible to sign a message to publish it, so everyone can check the signature. For example, *The Publisher* wants to sign the message (let's say, 456). Then he/she uses d exponent to compute signature:

Listing 1.26: Wolfram Mathematica

```

In [] := signed = Mod[456^d, n]
Out [] = 2282

```

Now he publishes $n = pq$ (3827), e (17 in our example), the message (456) and the signature (2282). Some other *Consumers* can verify its signature using e exponent and n :

Listing 1.27: Wolfram Mathematica

```

In [] := Mod[2282^e, n]
Out [] = 456

```

... this is another illustration that e and d exponents are complement each other, by modulo $totient = (p - 1)(q - 1)$.

The signature can only be generated with the access to p and q values, but it can be verified using product ($n = pq$) value.

1.4.9 Hybrid cryptosystem

RSA is slow, because exponentiation is slow and exponentiation by modulo is also slow. Perhaps, this is the reason why it was considered impractical by GCHQ when Clifford Cocks first came with this idea in 1970s. So in practice, if *The Sender* wants to encrypt some big piece of data to *The Receiver*, a random number is generated, which is used as a key for symmetrical cryptosystem like DES or AES. The piece of data is encrypted by the random key. The key is then encrypted by **RSA** to *The Receiver* and destroyed. *The Receiver* recovers the random key using **RSA** and decrypts all the data using DES or AES.

Chapter 2

Modulo arithmetics

2.1 Quick introduction into modular arithmetic

Modular arithmetic is an environment where all values are limited by some number (modulo). Many textbooks has clock as example. Let's imagine old mechanical analog clock. There hour hand points to one of number in bounds of 0..11 (zero is usually shown as 12). What hour will be if to sum up 10 hours (no matter, AM or PM) and 4 hours? $10+4$ is 14 or 2 by modulo 12. Naively you can just sum up numbers and subtract modulo base (12) as long as it's possible.

Modern digital watch shows time in 24 hours format, so hour there is a variable in modulo base 24. But minutes and seconds are in modulo 60 (let's forget about leap seconds for now).

Another example is US imperial system of measurement: human height is measured in feets and inches. There are 12 inches in feet, so when you sum up some lengths, you increase feet variable each time you've got more than 12 inches.

Another example I would recall is password cracking utilities. Often, characters set is defined in such utilities. And when you set all Latin characters plus numbers, you've got $26+10=36$ characters in total. If you brute-forcing a 6-characters password, you've got 6 variables, each is limited by 36. And when you increase last variable, it happens in modular arithmetic rules: if you got 36, set last variable to 0 and increase penultimate one. If it's also 36, do the same. If the very first variable is 36, then stop. Modular arithmetic may be very helpful when you write multi-threading (or distributed) password cracking utility and you need to slice all passwords space by even parts.

This is yet another application of modulo arithmetic, which many of us encountered in childhood.

Given a counting rhyme, like:

```
Eeny, meeny, miny, moe,  
Catch a tiger by the toe.  
If he hollers, let him go,  
Eeny, meeny, miny, moe.
```

(https://en.wikipedia.org/wiki/Eeny,_meeny,_miny,_moe)

... predict, who will be choosen.

If I'm correct, that rhyme has 16 items, and if a group of kids consist of, say, 5 kids, who will be choosen? $16 \bmod 5 = 1$, meaning, the next kid after the one at whom counting had begun.

Or 7 kids, $16 \bmod 7 = 2$. Count two kids after the first one.

If you can calculate this quickly, you can probably get an advantage by choosing a better place within a circle...

Now let's recall old mechanical counters which were widespread in pre-digital era:



Figure 2.1: The picture was stolen from <http://www.featurepics.com/> — sorry for that!

This counter has 6 wheels, so it can count from 0 to $10^6 - 1$ or 999999. When you have 999999 and you increase the counter, it will resetting to 000000— this situation is usually understood by engineers and computer programmers as overflow. And if you have 000000 and you decrease it, the counter will show you 999999. This situation is often called “wrap around”. See also: http://en.wikipedia.org/wiki/Integer_overflow.

2.1.1 Modular arithmetic on CPUs

The reason I talk about mechanical counter is that CPU registers acting in the very same way, because this is, perhaps, simplest possible and efficient way to compute using integer numbers.

This implies that almost all operations on integer values on your CPU is happens by modulo 2^{32} or 2^{64} depending on your CPU. For example, you can sum up 0x87654321 and 0xDEADBABA, which resulting in 0x16612FDDB. This value is too big for 32-bit register, so only 0x6612FDDB is stored, and leading 1 is dropped. If you will multiply these two numbers, the actual result it 0x75C5B266EDA5BFFA, which is also too big, so only low 32-bit part is stored into destination register: 0xEDA5BFFA. This is what happens when you multiply numbers in plain C/C++ language, but some readers may argue: when sum is too big for register, CF (carry flag) is set, and it can be used after. And there is x86 MUL instruction which in fact produces 64-bit result in 32-bit environment (in EDX:EAX registers pair). That's true, but observing just 32-bit registers, this is exactly environment of modulo with base 2^{32} .

Now that leads to a surprising consequence: almost every result of arithmetic operation stored in general purpose register of 32-bit CPU is in fact remainder of division operation: result is always divided by 2^{32} and remainder is left in register. For example, 0x16612FDDB is too large for storage, and it's divided by 2^{32} (or 0x100000000). The result of division (quotient) is 1 (which is dropped) and remainder is 0x6612FDDB (which is stored as a result). 0x75C5B266EDA5BFFA divided by 2^{32} (0x100000000) produces 0x75C5B266 as a result of division (quotient) and 0xEDA5BFFA as a remainder, the latter is stored.

And if your code is 32-bit one in 64-bit environment, CPU registers are bigger, so the whole result can be stored there, but high half is hidden behind the scenes – because no 32-bit code can access it.

By the way, this is the reason why remainder calculation is often called "division by modulo". C/C++ has a percent sign ("%") for this operation, but some other PLs like Pascal and Haskell has "mod" operator.

Usually, almost all sane computer programmers works with variables as they never wrapping around and value here is always in some limits which are defined preliminarily. However, this implicit division operation or "wrapping around" can be exploited usefully.

2.1.2 Remainder of division by modulo 2^n

... can be easily computed with AND operation. If you need a random number in range of 0..16, here you go: `rand()&0xF`. That helps sometimes.

For example, you need a some kind of wrapping counter variable which always should be in 0..16 range. What you do? Programmers often write this:

```
int counter=0;
...
counter++;
if (counter==16)
    counter=0;
```

But here is a version without conditional branching:

```
int counter=0;
...
counter++;
counter=counter&0xF;
```

As an example, this I found in the git source code:

```
char *sha1_to_hex(const unsigned char *sha1)
{
    static int bufno;
    static char hexbuffer[4][GIT_SHA1_HEXSZ + 1];
    static const char hex[] = "0123456789abcdef";
    char *buffer = hexbuffer[3 & ++bufno], *buf = buffer;
    int i;

    for (i = 0; i < GIT_SHA1_RAWSZ; i++) {
        unsigned int val = *sha1++;
        *buf++ = hex[val >> 4];
        *buf++ = hex[val & 0xf];
    }
    *buf = '\0';
}
```

```
    return buffer;
}
```

(<https://github.com/git/git/blob/aa1c6fdf478c023180e5ca5f1658b00a72592dc6/hex.c>)

This function returns a pointer to the string containing hexadecimal representation of SHA1 digest (like "4e1243bd22c66e76c2ba9eddc1f91394e57f9f83"). But this is plain C and you can calculate SHA1 for some block, get pointer to the string, then calculate SHA1 for another block, get pointer to the string, and both pointers are still points to the same string buffer containing the result of the second calculation. As a solution, it's possible to allocate/deallocate string buffer each time, but more hackish way is to have several buffers (4 are here) and fill the next each time. The *bufno* variable here is a buffer counter in 0..3 range. Its value increments each time, and its value is also always kept in limits by AND operation ($3 \& ++bufno$).

The author of this piece of code (seemingly Linus Torvalds himself) went even further and forgot (?) to initialize *bufno* counter variable, which will have random garbage at the function start. Indeed: no matter, which buffer we are starting each time! This can be mistake which isn't affect correctness of the code, or maybe this is left so intentionally – I don't know.

2.1.3 Getting random numbers

When you write some kind of videogame, you need random numbers, and the standard C/C++ `rand()` function gives you them in 0..0x7FFF range (MSVC) or in 0..0x7FFFFFFF range (GCC). And when you need a random number in 0..10 range, the common way to do it is:

```
X_coord_of_something_spawned_somewhere=rand() % 10;
Y_coord_of_something_spawned_somewhere=rand() % 10;
```

No matter what compiler do you use, you can think about it as 10 is subtraced from `rand()` result, as long as there is still a number bigger than 10. Hence, result is remainder of division of `rand()` result by 10.

One nasty consequence is that neither 0x8000 nor 0x80000000 cannot be divided by 10 evenly, so you'll get some numbers slightly more often than others.

I tried to calculate in Mathematica. Here is what you get if you write `<i>rand()`

```
In [] := Counts[Map[Mod[#, 3] &, Range[0, 168 - 1]]]
Out [] = <|0 -> 10923, 1 -> 10923, 2 -> 10922|>
```

So a number 2 appers slightly seldom than others.

Here is a result for `rand() % 10`:

```
In [] := Counts[Map[Mod[#, 10] &, Range[0, 168 - 1]]]
Out [] = <|0 -> 3277, 1 -> 3277, 2 -> 3277, 3 -> 3277, 4 -> 3277,
5 -> 3277, 6 -> 3277, 7 -> 3277, 8 -> 3276, 9 -> 3276|>
```

Numbers 8 and 9 appears slightly seldom (3276 against 3277).

Here is a result for `rand() % 100`:

```

In []:= Counts[Map[Mod[#, 100] &, Range[0, 16^^8000 - 1]]]
Out []= <|0 -> 328, 1 -> 328, 2 -> 328, 3 -> 328, 4 -> 328, 5 -> 328,
  6 -> 328, 7 -> 328, 8 -> 328, 9 -> 328, 10 -> 328, 11 -> 328,
  12 -> 328, 13 -> 328, 14 -> 328, 15 -> 328, 16 -> 328, 17 -> 328,
  18 -> 328, 19 -> 328, 20 -> 328, 21 -> 328, 22 -> 328, 23 -> 328,
  24 -> 328, 25 -> 328, 26 -> 328, 27 -> 328, 28 -> 328, 29 -> 328,
  30 -> 328, 31 -> 328, 32 -> 328, 33 -> 328, 34 -> 328, 35 -> 328,
  36 -> 328, 37 -> 328, 38 -> 328, 39 -> 328, 40 -> 328, 41 -> 328,
  42 -> 328, 43 -> 328, 44 -> 328, 45 -> 328, 46 -> 328, 47 -> 328,
  48 -> 328, 49 -> 328, 50 -> 328, 51 -> 328, 52 -> 328, 53 -> 328,
  54 -> 328, 55 -> 328, 56 -> 328, 57 -> 328, 58 -> 328, 59 -> 328,
  60 -> 328, 61 -> 328, 62 -> 328, 63 -> 328, 64 -> 328, 65 -> 328,
  66 -> 328, 67 -> 328, 68 -> 327, 69 -> 327, 70 -> 327, 71 -> 327,
  72 -> 327, 73 -> 327, 74 -> 327, 75 -> 327, 76 -> 327, 77 -> 327,
  78 -> 327, 79 -> 327, 80 -> 327, 81 -> 327, 82 -> 327, 83 -> 327,
  84 -> 327, 85 -> 327, 86 -> 327, 87 -> 327, 88 -> 327, 89 -> 327,
  90 -> 327, 91 -> 327, 92 -> 327, 93 -> 327, 94 -> 327, 95 -> 327,
  96 -> 327, 97 -> 327, 98 -> 327, 99 -> 327|>

```

...now larger part of numbers happens slightly seldom, these are 68...99.

This is sometimes called *modulo bias*. It's perhaps acceptable for videogames, but may be critical for scientific simulations, including Monte Carlo method.

Constructing a PRNG¹ with uniform distribution may be tricky, there are couple of methods:

http://www.reddit.com/r/algorithms/comments/39tire/using_a_01_generator_generate_a_random_number/,
<http://www.prismodelchecker.org/casestudies/dice.php>.

2.2 Modulo inverse, part I

Example: this piece of code divides by 17:

```

#include <stdio.h>
#include <stdint.h>

uint32_t div17 (uint32_t a)
{
    return a*0xf0f0f0f1;
};

int main()
{
    printf ("%d\n", div17(1700)); // result=100
    printf ("%d\n", div17(34)); // result=2
    printf ("%d\n", div17(2091)); // result=123
};

```

¹Pseudorandom number generator

How it works?

Let's imagine, we work on 4-bit CPU, it has 4-bit registers, each can hold a value in 0..15 range.

Now we want to divide by 3 using multiplication. Let's find modulo inverse of 3 using Wolfram Mathematica:

```
In [] := PowerMod[3, -1, 16]
Out [] = 11
```

This is in fact solution of a $3m = 16k + 1$ equation ($16 = 2^4$):

```
In [] := FindInstance[3 m == 16 k + 1, {m, k}, Integers]
Out [] = {{m -> 11, k -> 2}}
```

The "magic number" for division by 3 is 11. Multiply by 11 instead of dividing by 3 and you'll get a result (quotient).

This works, let's divide 6 by 3. We can now do this by multiplying 6 by 11, this is $66=0x42$, but on 4-bit register, only $0x2$ will be left in register ($0x42 \equiv 2 \pmod{2^4}$). Yes, 2 is correct answer, $6/3=2$.

Let's divide 3, 6 and 9 by 3, by multiplying by 11 (m).

	123456789 abcdef0	12 ... f0	123456789 abcdef0	12 ... f0	123456789 abcdef0	12 ... f0	123456789 abcdef0
m=11	*****
3/3 3m=33	*****	** ... **	*
6/3 6m=66	*****	** ... **	*****	** ... **	**
9/3 9m=99	*****	** ... **	*****	** ... **	*****	** ... **	***

A "protruding" asterisk(s) ("*") in the last non-empty chunk is what will be left in 4-bit register. This is 1 in case of 33, 2 if 66, 3 if 99.

In fact, this "protrusion" is defined by 1 in the equation we've solved. Let's replace 1 with 2:

```
In [] := FindInstance[3 m == 16 k + 2, {m, k}, Integers]
Out [] = {{m -> 6, k -> 1}}
```

Now the new "magic number" is 6. Let's divide 3 by 3. $3*6=18=0x12$, 2 will be left in 4-bit register. This is incorrect, we have 2 instead of 1. 2 asterisks are "protruding". Let's divide 6 by 3. $6*6=36=0x24$, 4 will be left in the register. This is also incorrect, we now have 4 "protruding" asterisks instead of correct 2.

Replace 1 in the equation by 0, and nothing will "protrude".

2.2.1 No remainder?

Now the problem: this only works for dividends in $3x$ form, i.e., which can be divided by 3 with no remainder. Try to divide 4 by 3, $4*11=44=0x2c$, 12 will be left in register, this is incorrect. The correct quotient is 1.

We can also notice that the 4-bit register is "overflowed" during multiplication twice as much as in "incorrect" result in low 4 bits.

Here is what we can do: use only high 4 bits and drop low 4 bits. $4*11=0x2c$ and 2 is high 4 bits. Divide 2 by 2, this is 1.

Let's "divide" 8 by 3. $8 * 11 = 88 = 0x58$. $5/2 = 2$, this is correct answer again.

Now this is the formula we can use on our 4-bit CPU to divide numbers by 3: " $x * 3 \gg 4 / 2$ " or " $x * 3 \gg 5$ ". This is the same as almost all modern compilers do instead of integer division, but they do this for 32-bit and 64-bit registers.

2.3 Modulo inverse, part II

A very simple function:

```
int f(int a)
{
    return a/9;
};
```

If compiled by non-optimizing GCC 4.4.1...

Listing 2.1: Non-optimizing GCC 4.4.1

```
public f
proc near
arg_0 = dword ptr 8

push    ebp
mov     ebp, esp
mov     ecx, [ebp+arg_0]
mov     edx, 954437177 ; 38E38E39h
mov     eax, ecx
imul   edx
sar     edx, 1
mov     eax, ecx
sar     eax, 1Fh
mov     ecx, edx
sub     ecx, eax
mov     eax, ecx
pop     ebp
retn
f      endp
```

And it can be rewritten back to pure C like that:

```
#include <stdint.h>

uint32_t divide_by_9 (uint32_t a)
{
    return ((uint64_t)a * (uint64_t)954437177) >> 33; // 954437177 = 0x38e38e39
};
```

This function still works, without division operation. How?

From school-level mathematics, we can remember that division by 9 can be replaced by multiplication by $\frac{1}{9}$. In fact, sometimes compilers do so for floating-point arithmetics, for example, FDIV instruction in x86 code can be replaced by FMUL. At least MSVC 6.0 will replace division by 9 by multiplication by 0.111111... and sometimes it's hard to be sure, what operation was in the original source code.

But when we operate over integer values and integer CPU registers, we can't use fractions. However, we can rework fraction like that:

$$result = \frac{x}{9} = x \cdot \frac{1}{9} = x \cdot \frac{1 \cdot MagicNumber}{9 \cdot MagicNumber}$$

Given the fact that division by 2^n is very fast (using shifts), we now should find that *MagicNumber*, for which the following equation will be true: $2^n = 9 \cdot MagicNumber$.

Division by 2^{32} is somewhat hidden: lower 32-bit of product in EAX is not used (dropped), only higher 32-bit of product (in EDX) is used and then shifted by additional 1 bit.

In other words, the assembly code we have just seen multiplies by $\frac{954437177}{2^{32+1}}$, or divides by $\frac{2^{32+1}}{954437177}$. To find a divisor we just have to divide numerator by denominator. Using Wolfram Alpha, we can get 8.99999999... as result (which is close to 9).

Many people miss "hidden" division by 2^{32} or 2^{64} , when lower 32-bit part (or 64-bit part) of product is not used. This is why division by multiplication is difficult to understand at the beginning.

2.4 Reversible linear congruential generator

LCG² PRNG is very simple: just multiply seed by some value, add another one and here is a new random number. Here is how it is implemented in MSVC (the source code is not original one and is reconstructed by me):

```
uint32_t state;

uint32_t rand()
{
    state=state*214013+2531011;
    return (state>>16)&0x7FFF;
};
```

The last bit shift is attempt to compensate LCG weakness and we may ignore it so far. Will it be possible to make an inverse function to rand(), which can reverse state back? First, let's try to think, what would make this possible? Well, if state internal variable would be some kind of BigInt or BigNum container which can store infinitely big numbers, then, although state is increasing rapidly, it would be possible to reverse the process. But *state* isn't BigInt/BigNum, it's 32-bit variable, and summing operation is easily reversible on it (just subtract 2531011 at each step). As we may know now, multiplication is also reversible: just multiply the state by modular multiplicative inverse of 214013!

```
#include <stdio.h>
#include <stdint.h>

uint32_t state;
```

²Linear congruential generator

```

void next_state()
{
    state=state*214013+2531011;
};

void prev_state()
{
    state=state-2531011; // reverse summing operation
    state=state*3115528533; // reverse multiply operation. 3115528533 is modular
                           inverse of 214013 in 232.
};

int main() [style=customc]
{
    state=12345;

    printf ("state=%d\n", state);
    next_state();
    printf ("state=%d\n", state);
    next_state();
    printf ("state=%d\n", state);
    next_state();
    printf ("state=%d\n", state);

    prev_state();
    printf ("state=%d\n", state);
    prev_state();
    printf ("state=%d\n", state);
    prev_state();
    printf ("state=%d\n", state);
};

```

Wow, that works!

```

state=12345
state=-1650445800
state=1255958651
state=-456978094
state=1255958651
state=-1650445800
state=12345

```

It's hard to find a real-world application of reversible LCG, but this could be the one: a media player with forward/backward buttons. Once shuffle is clicked, random number is generated (number of item to be played). User clicks forward: get a new random number by calculating the next state. User clicks backward: get it by calculating the previous state. Thus, a user could navigate through some "virtual" (but consistent) playlist, which is even not present in media player's memory!

2.5 Getting magic number using extended Euclidean algorithm

Extended Euclidean algorithm can find x/y for given a/b in the diophantine equation: $ax + by = gcd(a, b)$.

x/y are also known as “Bézout coefficients”.

However, since a/b are coprime to each other, $gcd(a, b) = 1$, and the algorithm will find x/y for the $ax + by = 1$ equation.

Let’s plug $a = 3$ and $b = 2^{16}$ (like if we finding magic constant for 16-bit CPU):

```
#include <assert.h>
#include <stdio.h>
#include <stdint.h>

// copyasted and reworked from https://en.wikipedia.org/wiki/Extended\_Euclidean\_algorithm
void EGCD (int a, int b)
{
    int s=0; int old_s=1;
    int t=1; int old_t=0;
    int r=b; int old_r=a;
    int tmp;

    while (r!=0)
    {
        int quotient=old_r/r;
        tmp=r; r=old_r - quotient*r; old_r=tmp;
        tmp=s; s=old_s - quotient*s; old_s=tmp;
        tmp=t; t=old_t - quotient*t; old_t=tmp;
    };
    printf ("GCD: %d\n", old_r);
    if (old_r!=1)
    {
        printf("%d and %d are not coprimes!\n", a, b);
        return;
    };
    printf ("Bézout coefficients: %d %d\n", old_s, old_t);
    printf ("quotients by the GCD (s,t): %d %d\n", s, t);

    // see also: https://math.stackexchange.com/q/1310415
    if (old_s>=0 && old_t<=0)
        printf ("corrected coefficients: %d(0x%x) %d(0x%x)\n", old_s,
            old_s, -old_t, -old_t);
    else if (old_s<=0 && old_t >=0)
        printf ("corrected coefficients: %d(0x%x) %d(0x%x)\n", old_s+b,
            old_s+b, -old_t+a, -old_t+a);
    else
        assert(0);
};

void main()
{
```

```
EGCD(3, 0x10000);  
};
```

```
GCD: 1  
Bézout coefficients: -21845 1  
quotients by the GCD (s,t): 65536 -3  
corrected coefficients: 43691(0xaaab) 2(0x2)
```

That means, $x=-21845, y=1$. Is it correct? $-21845 * 3 + 65536 * 1 = 1$

$65536-21845=0xaaab$, and this is the magic number for division by 3 on 16-bit CPU.

$GCD(\text{any_odd_number}, 2^n) = 1$ for any values. Hence, we can find a magic number for any even number. But, this is not true for even numbers. We can't find magic coefficient for even number like 10. But you can find it for 5, and then add additional division by 2 using shift right.

If you try to compile $\frac{x}{10}$ code, GCC can do it like:

```
push    %ebp  
mov     %esp,%ebp  
mov     0x8(%ebp),%eax  
mov     $0xcccccccd,%edx  
mul     %edx  
mov     %edx,%eax  
shr     $0x3,%eax  
pop     %ebp  
ret
```

This is in fact the magic number for division by 5. And there is 3 instead of 2 in the SHR instruction, so the result is divided by 2.

Extended Euclidean algorithm is probably an efficient way of finding magic number, but needless to say, this equation can be solved using other ways. In "SAT/SMT by Example"³ you can find a method of finding "magic number" using SMT-solver.

³https://yurichev.com/writings/SAT_SMT_by_example.pdf

Chapter 3

Probability

3.1 Text strings right in the middle of compressed data

You can download Linux kernels and find English words right in the middle of compressed data:

```
% wget https://www.kernel.org/pub/linux/kernel/v4.x/linux-4.10.2.tar.gz
% xxd -g 1 -seek 0x515c550 -l 0x30 linux-4.10.2.tar.gz

0515c550: c5 59 43 cf 41 27 85 54 35 4a 57 90 73 89 b7 6a  .YC.A'.T5JW.s..j
0515c560: 15 af 03 db 20 df 6a 51 f9 56 49 52 55 53 3d da  .... .jq.VIRUS=.
0515c570: 0e b9 29 24 cc 6a 38 e2 78 66 09 33 72 aa 88 df  ..)$.j8.xf.3r...
```

```
% wget https://cdn.kernel.org/pub/linux/kernel/v2.3/linux-2.3.3.tar.bz2
% xxd -g 1 -seek 0xa93086 -l 0x30 linux-2.3.3.tar.bz2

00a93086: 4d 45 54 41 4c cd 44 45 2d 2c 41 41 54 94 8b a1  METAL.DE-,AAT...
00a93096: 5d 2b d8 d0 bd d8 06 91 74 ab 41 a0 0a 8a 94 68  ]+.....t.A....h
00a930a6: 66 56 86 81 68 0d 0e 25 6b b6 80 a4 28 1a 00 a4  fV..h..%k...(...
```

One of Linux kernel patches in compressed form has the “Linux” word itself:

```
% wget https://cdn.kernel.org/pub/linux/kernel/v4.x/testing/patch-4.6-rc4.gz
% xxd -g 1 -seek 0x4d03f -l 0x30 patch-4.6-rc4.gz

0004d03f: c7 40 24 bd ae ef ee 03 2c 95 dc 65 eb 31 d3 f1  .@$.....,..e.1..
0004d04f: 4c 69 6e 75 78 f2 f3 70 3c 3a bd 3e bd f8 59 7e  Linux..p<:.>..Y~
0004d05f: cd 76 55 74 2b cb d5 af 7a 35 56 d7 5e 07 5a 67  .vUt+...z5V.^.Zg
```

Other English words I’ve found in other compressed Linux kernel trees:

```
linux-4.6.2.tar.gz: [maybe] at 0x68e78ec
linux-4.10.14.tar.xz: [OCEAN] at 0x6bf0a8
linux-4.7.8.tar.gz: [FUNNY] at 0x29e6e20
linux-4.6.4.tar.gz: [DRINK] at 0x68dc314
linux-2.6.11.8.tar.bz2: [LUCKY] at 0x1ab5be7
linux-3.0.68.tar.gz: [BOOST] at 0x11238c7
linux-3.0.16.tar.bz2: [APPLE] at 0x34c091
linux-3.0.26.tar.xz: [magic] at 0x296f7d9
linux-3.11.8.tar.bz2: [TRUTH] at 0xf635ba
linux-3.10.11.tar.bz2: [logic] at 0x4a7f794
```

There is a nice illustration of apophenia and pareidolia (human's mind ability to see faces in clouds, etc) in Lurkmore, Russian counterpart of Encyclopedia Dramatica. As they wrote in the article about electronic voice phenomenon¹, you can open any long enough compressed file in hex editor and find well-known 3-letter Russian obscene word, and you'll find it a lot: but that means nothing, just a mere coincidence.

And I was interested in calculation, how big compressed file must be to contain all possible 3-letter, 4-letter, etc, words? In my naive calculations, I've got this: probability of the first specific byte in the middle of compressed data stream with maximal entropy is $\frac{1}{256}$, probability of the 2nd is also $\frac{1}{256}$, and probability of specific byte pair is $\frac{1}{256 \cdot 256} = \frac{1}{256^2}$. Probability of specific triple is $\frac{1}{256^3}$. If the file has maximal entropy (which is almost unachievable, but ...) and we live in an ideal world, you've got to have a file of size just $256^3 = 16777216$, which is 16-17MB. You can check: get any compressed file, and use *rafind2* to search for any 3-letter word (not just that Russian obscene one).

It took ≈ 8 -9 GB of my downloaded movies/TV series files to find the word "beer" in them (case sensitive). Perhaps, these movies wasn't compressed good enough? This is also true for a well-known 4-letter English obscene word.

My approach is naive, so I googled for mathematically grounded one, and have find this question: "Time until a consecutive sequence of ones in a random bit sequence"². The answer is: $(p^{-n}-1)/(1-p)$, where p is probability of each event and n is number of consecutive events. Plug $\frac{1}{256}$ and 3 and you'll get almost the same as my naive calculations.

So any 3-letter word can be found in the compressed file (with ideal entropy) of length $256^3 \approx 17MB$, any 4-letter word — $256^4 = 4.7GB$ (size of DVD). Any 5-letter word — $256^5 \approx 1TB$.

For the piece of text you are reading now, I mirrored the whole kernel.org website (hopefully, sysadmins can forgive me), and it has $\approx 430GB$ of compressed Linux Kernel source trees. It has enough compressed data to contain these words, however, I cheated a bit: I searched for both lowercase and uppercase strings, thus compressed data set I need is almost halved.

This is quite interesting thing to think about: 1TB of compressed data with maximal entropy has all possible 5-byte chains, but the data is encoded not in chains itself, but in the order of chains (no matter of compression algorithm, etc).

Now the information for gamblers: one should throw a dice ≈ 42 times to get a pair of six, but no one will tell you, when exactly this will happen. I don't remember, how many times coin was tossed in the "Rosencrantz & Guildenstern Are Dead" movie, but one should toss it ≈ 2048 times and at some point, you'll get 10 heads in a row, and at some other point, 10 tails in a row. Again, no one will tell you, when exactly this will happen.

Compressed data can also be treated as a stream of random data, so we can use the same mathematics to determine probabilities, etc.

¹<http://archive.is/gYnFL>

²<http://math.stackexchange.com/questions/27989/time-until-a-consecutive-sequence-of-ones-in-a-random-bit-sequence/27991#27991>

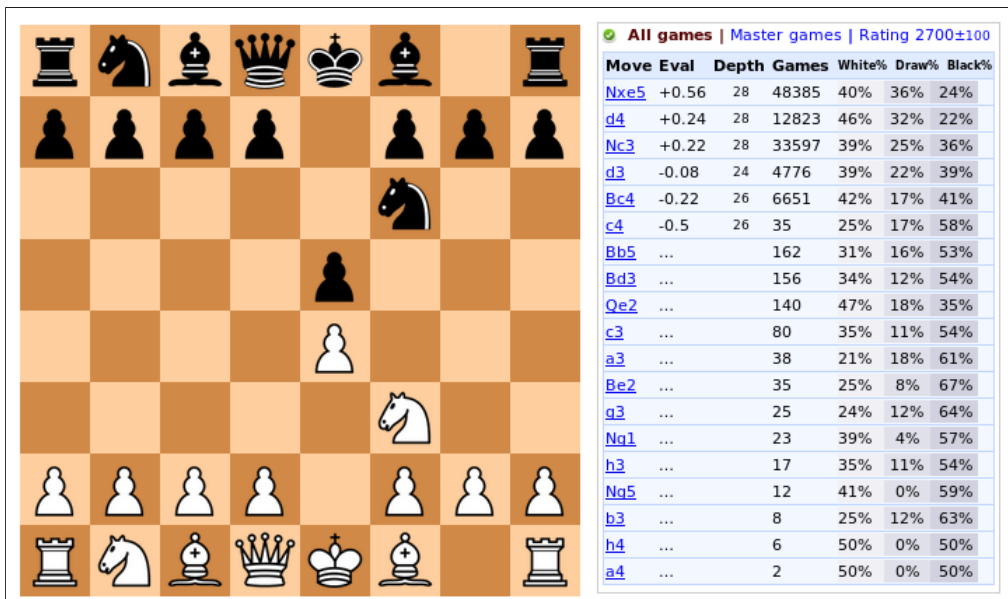
If you can live with strings of mixed case, like “bEeR”, probabilities and compressed data sets are much lower: $128^3 = 2MB$ for all 3-letter words of mixed case, $128^4 = 268MB$ for all 4-letter words, $128^5 = 34GB$ for all 5-letter words, etc.

Moral of the story: whenever you search for some patterns, you can find it in the middle of compressed blob, but that means nothing else then coincidence. In philosophical sense, this is a case of selection/confirmation bias: you find what you search for in “The Library of Babel”³.

3.2 Autocomplete using Markov chains

TL;DR: collect statistics, for a given natural language, what words come most often after a word/pair of words/triplet of words.

What are most chess moves played after 1.e4 e5 2.Nf3 Nf6? A big database of chess games can be queried, showing statistics:



(from <https://chess-db.com/>)

Statistics shown is just number of games, where a corresponding 3rd move was played.

The same database can be made for natural language words.

3.2.1 Dissociated press

This is a well-known joke: https://en.wikipedia.org/wiki/Dissociated_press, <https://en.wikipedia.org/wiki/SCiGen>, https://en.wikipedia.org/wiki/Mark_V._Shaney.

I wrote a Python script, took Sherlock Holmes stories from Gutenberg library...

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
```

³A short story by Jorge Luis Borges

```

# python 3! due to random.choices()

import operator, random
from collections import defaultdict

with open ("all.txt", "r") as myfile:
    data=myfile.read()

sentences=data.lower().replace('\r',' ').replace('\n',' ').replace('?','').
    replace('!','.').replace('"','').replace("'",'.').replace("\",".").replace(
    ',' , ' ').replace('-', ' ').replace(' ',' ').replace('\',' ').split(".")

def remove_empty_words(l):
    return list(filter(lambda a: a != '', l))

# key=list of words, as a string, delimited by space
# (I would use list of strings here as key, but list is not hashable)
# val=dict, k: next word; v: occurrences
first={}
second={}

def update_occ(d, seq, w):
    if seq not in d:
        d[seq]=defaultdict(int)

    d[seq][w]=d[seq][w]+1

for s in sentences:
    words=s.replace(',',' ').split(" ")
    words=remove_empty_words(words)
    if len(words)==0:
        continue
    for i in range(len(words)):
        if i>=1:
            update_occ(first, words[i-1], words[i])
        if i>=2:
            update_occ(second, words[i-2]+" "+words[i-1], words[i])

"""
print ("first table:")
for k in first:
    print (k)
    # https://stackoverflow.com/questions/613183/how-do-i-sort-a-dictionary-by-
    value
    s=sorted(first[k].items(), key=operator.itemgetter(1), reverse=True)
    print (s[:20])
    print ("")
"""
"""
print ("second table:")

```

```

for k in second:
    print (k)
    # https://stackoverflow.com/questions/613183/how-do-i-sort-a-dictionary-by-
    # value
    s=sorted(second[k].items(), key=operator.itemgetter(1), reverse=True)
    print (s[:20])
    print ("")
"""

text=["it", "is"]

# https://docs.python.org/3/library/random.html#random.choice
def gen_random_from_tbl(t):
    return random.choices(list(t.keys()), weights=list(t.values()))[0]

text_len=len(text)

# generate at most 100 words:
for i in range(200):

    last_idx=text_len-1

    tmp=text[last_idx-1]+" "+text[last_idx]
    if tmp in second:
        new_word=gen_random_from_tbl(second[tmp])
    else:
        # fall-back to 1st order
        tmp2=text[last_idx]
        if tmp2 not in first:
            # dead-end
            break
        new_word=gen_random_from_tbl(first[tmp2])

    text.append(new_word)
    text_len=text_len+1

print (" ".join(text))

```

And here are some 1st-order Markov chains. First part is a first word. Second is a list of words + probabilities of appearance of each one, in the Sherlock Holmes stories. However, probabilities are in form of words' numbers.

In other word, how often each word appears after a word?

```

return
[('to', 28), ('or', 12), ('of', 8), ('the', 8), ('for', 5), ('with', 4), ('by',
4), ('journey', 4), ('and', 3), ('when', 2), ('ticket', 2), ('from', 2), ('at
', 2), ('you', 2), ('i', 2), ('since', 1), ('on', 1), ('caused', 1), ('but',
1), ('it', 1)]

of
[('the', 3206), ('a', 680), ('his', 584), ('this', 338), ('it', 304), ('my',

```

```
295), ('course', 225), ('them', 167), ('that', 138), ('your', 137), ('our', 135), ('us', 122), ('her', 116), ('him', 111), ('an', 105), ('any', 102), ('these', 92), ('which', 89), ('all', 82), ('those', 75)]
```

```
by  
[('the', 532), ('a', 164), ('his', 67), ('my', 39), ('no', 34), ('this', 31), ('an', 30), ('which', 29), ('your', 21), ('that', 19), ('one', 17), ('all', 17), ('jove', 16), ('some', 16), ('sir', 13), ('its', 13), ('him', 13), ('their', 13), ('it', 11), ('her', 10)]
```

```
this  
[('is', 177), ('agreement', 96), ('morning', 81), ('was', 61), ('work', 60), ('man', 56), ('case', 43), ('time', 39), ('matter', 37), ('ebook', 36), ('way', 36), ('fellow', 28), ('room', 27), ('letter', 24), ('one', 24), ('i', 23), ('young', 21), ('very', 20), ('project', 18), ('electronic', 18)]
```

```
no  
[('doubt', 179), ('one', 134), ('sir', 61), ('more', 56), ('i', 55), ('no', 42), ('other', 36), ('means', 36), ('sign', 34), ('harm', 23), ('reason', 22), ('difficulty', 22), ('use', 21), ('idea', 20), ('longer', 20), ('signs', 18), ('good', 17), ('great', 16), ('trace', 15), ('man', 15)]
```

Now some snippets from 2nd-order Markov chains.

```
the return  
[('of', 8), ('track', 1), ('to', 1)]
```

```
return of  
[('sherlock', 6), ('lady', 1), ('the', 1)]
```

```
for the  
[('use', 22), ('first', 12), ('purpose', 9), ('sake', 8), ('rest', 7), ('best', 7), ('crime', 6), ('evening', 6), ('ebooks', 6), ('limited', 6), ('moment', 6), ('day', 5), ('future', 5), ('last', 5), ('world', 5), ('time', 5), ('loss', 4), ('second', 4), ('night', 4), ('remainder', 4)]
```

```
use of  
[('the', 15), ('anyone', 12), ('it', 6), ('project', 6), ('and', 6), ('having', 2), ('this', 1), ('my', 1), ('a', 1), ('horses', 1), ('some', 1), ('arguing', 1), ('troubling', 1), ('artificial', 1), ('invalids', 1), ('cocaine', 1), ('disguises', 1), ('an', 1), ('our', 1)]
```

```
you may  
[('have', 17), ('copy', 13), ('be', 13), ('do', 9), ('choose', 7), ('use', 6), ('obtain', 6), ('convert', 6), ('charge', 6), ('demand', 6), ('remember', 5), ('find', 5), ('take', 4), ('think', 3), ('possibly', 3), ('well', 3), ('know', 3), ('not', 3), ('say', 3), ('imagine', 3)]
```

```
the three  
[('of', 8), ('men', 4), ('students', 2), ('glasses', 2), ('which', 1), ('strips
```

, 1), ('is', 1), ('he', 1), ('gentlemen', 1), ('enterprising', 1), ('massive', 1), ('quarter', 1), ('randalls', 1), ('fugitives', 1), ('animals', 1), ('shrieked', 1), ('other', 1), ('murderers', 1), ('fir', 1), ('at', 1)]

it was

[('a', 179), ('the', 78), ('not', 68), ('only', 40), ('in', 30), ('evident', 28), ('that', 28), ('all', 25), ('to', 21), ('an', 18), ('my', 17), ('at', 17), ('impossible', 17), ('indeed', 15), ('no', 15), ('quite', 15), ('he', 14), ('of', 14), ('one', 12), ('on', 12)]

Now two words is a key in dictionary. And we see here an answer for the question "how often each words appears after a sequence of two words?"

Now let's generate some rubbish:

it is just to the face of granite still cutting the lower part of the shame which i would draw up so as i can t have any visitors if he is passionately fond of a dull wrack was drifting slowly in our skin before long vanished in front of him of this arch rebel lines the path which i had never set foot in the same with the heads of these have been supplemented or more waiting to tell me all that my assistant hardly knowing whether i had 4 pounds a month however is foreign to the other hand were most admirable but because i know why he broke into a curve by the crew had thought it might have been on a different type they were about to set like a plucked chicken s making the case which may help to play it at the door and called for the convenience of a boat sir maybe i could not be made to draw some just inference that a woman exactly corresponding to the question was how a miners camp had been dashed savagely against the rock in front of the will was drawn across the golden rays and it

it is the letter was composed of a tall stout official had come the other way that she had been made in this i expect that we had news that the office of the mud settling or the most minute exactness and astuteness represented as i come to see the advantages which london then and would i could not speak before the public domain ebooks in compliance with any particular paper edition as to those letters come so horribly to his feet upon the wet clayey soil but since your wife and of such damage or cannot be long before the rounds come again whenever she might be useful to him so now my fine fellow will trouble us again and again and making a little wizened man darted out of the baskervilles *** produced by roger squires updated editions will be the matter and let out forty three diamonds of the attack we carried him into the back and white feather were but a terrible fellow he is not for your share in an instant holmes clapped his hands and play with me anyhow i d ha known you under that name in a considerable treasure was hid for no other

it is the unofficial force the shutter open but so far as to what i say to me like that which is rather too far from at ease for i knew that we were driving; but soon it came just as he opened it myself i was not a usual signal between you and you thought it might be removed to a magistrate than upon the luminous screen of the one word would he have wanted there are business relations between him and we listened to his eyes to the spot as

soon as their master s affairs of life since she was but one day we hoped would screen me from under his arm chair of shining red leather chair his flask in his chair and gave me his name is sherlock holmes took each face of a barmaid in bristol and marry her but learned from dr

it is selden the criminal or lunatic who had left a match might mar my career had reached the tower of the crime was committed before twelve foolish tradesmen in a bad lot though the authorities are excellent at amassing facts though what its object might be a better nerve for the creature flapped and struggled and writhed his hands in her bosom lady hilda was down on the table and the curious will so suddenly upon the floor were now nearly five thousand pounds will be impossible but i struck him down and roared into the shadow of the thing seemed simplicity itself said i you seem most fortunate in having every characteristic of the place is deserted up to keep some clandestine appointment and found as i have no desire to settle this little matter of fact of absolute ignorance and he with a similar pearl without any light upon what he had found ourselves at the time and my heart sank for barrymore at the close of november and holmes fears came to think over the afghan campaign yet shown by this time at which he now and i have seen his death this morning he walked straight into

By first look, these pieces of text are visually OK, but it is senseless. Some people (including me) find it amusing.

Spammers also use this technique to make email message visually similar to a meaningful text, albeit it is not meaningful at all, rather absurdic and funny.

3.2.2 Autocomplete

It's surprising how easy this can be turned into something rather practically useful.

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import operator
from collections import defaultdict

with open ("all.txt", "r") as myfile:
    data=myfile.read()

sentences=data.lower().replace('\r',' ').replace('\n',' ').replace('?',' ').
    replace('!', ' ').replace('"', ' ').replace("'", ' ').replace("\", ".").replace(
    ' ',' ').replace('-', ' ').replace(' ',' ').replace('\',' ').split(".")

def remove_empty_words(l):
    return list(filter(lambda a: a != '', l))

# key=list of words, as a string, delimited by space
# (I would use list of strings here as key, but list is not hashable)
# val=dict, k: next word; v: occurrences
first={}
second={}
```

```

third={}

def update_occ(d, seq, w):
    if seq not in d:
        d[seq]=defaultdict(int)

    d[seq][w]=d[seq][w]+1

for s in sentences:
    words=s.replace(',',' ').split(" ")
    words=remove_empty_words(words)
    if len(words)==0:
        continue
    for i in range(len(words)):
        # only two words available:
        if i>=1:
            update_occ(first, words[i-1], words[i])
        # three words available:
        if i>=2:
            update_occ(second, words[i-2]+" "+words[i-1], words[i])
        # four words available:
        if i>=3:
            update_occ(third, words[i-3]+" "+words[i-2]+" "+words[i-1], words[i
                ])

"""
print ("third table:")
for k in third:
    print (k)
    # https://stackoverflow.com/questions/613183/how-do-i-sort-a-dictionary-by-
    value
    s=sorted(third[k].items(), key=operator.itemgetter(1), reverse=True)
    print (s[:20])
    print ("")
"""

test="i can tell"
#test="who did this"
#test="she was a"
#test="he was a"
#test="i did not"
#test="all that she"
#test="have not been"
#test="who wanted to"
#test="he wanted to"
#test="wanted to do"
#test="it is just"
#test="you will find"
#test="you shall"
#test="proved to be"

```

```

test_words=test.split(" ")

test_len=len(test_words)
last_idx=test_len-1

def print_stat(t):
    total=float(sum(t.values()))

    # https://stackoverflow.com/questions/613183/how-do-i-sort-a-dictionary-by-value
    s=sorted(t.items(), key=operator.itemgetter(1), reverse=True)
    # take 5 from each sorted table
    for pair in s[:5]:
        print ("%s %d%%" % (pair[0], (float(pair[1])/total)*100))

if test_len>=3:
    tmp=test_words[last_idx-2]+" "+test_words[last_idx-1]+" "+test_words[
        last_idx]
    if tmp in third:
        print ("* third order. for sequence:",tmp)
        print_stat(third[tmp])

if test_len>=2:
    tmp=test_words[last_idx-1]+" "+test_words[last_idx]
    if tmp in second:
        print ("* second order. for sequence:", tmp)
        print_stat(second[tmp])

if test_len>=1:
    tmp=test_words[last_idx]
    if tmp in first:
        print ("* first order. for word:", tmp)
        print_stat(first[tmp])
print ("")

```

First, let's also make 3rd-order Markov chains tables. There are some snippets from it:

```

the return of
[('sherlock', 6), ('lady', 1), ('the', 1)]

the use of
[('the', 13), ('anyone', 12), ('project', 6), ('having', 2), ('a', 1), ('horses',
1), ('some', 1), ('troubling', 1), ('artificial', 1), ('invalids', 1), ('
disguises', 1), ('it', 1)]

of the second
[('stain', 5), ('floor', 3), ('page', 1), ('there', 1), ('person', 1), ('party',
1), ('day', 1)]

it was in

```



```
[('the', 9), ('vain', 5), ('a', 4), ('his', 2), ('83', 1), ('one', 1), ('motion', 1), ('truth', 1), ('my', 1), ('this', 1), ('march', 1), ('january', 1), ('june', 1), ('me', 1)]
```

was in the

```
[('room', 3), ('habit', 3), ('house', 2), ('last', 2), ('loft', 2), ('spring', 1), ('train', 1), ('bar', 1), ('power', 1), ('immediate', 1), ('year', 1), ('midst', 1), ('forenoon', 1), ('centre', 1), ('papers', 1), ('best', 1), ('darkest', 1), ('prime', 1), ('hospital', 1), ('nursery', 1)]
```

murder of the

```
[('honourable', 1), ('discoverer', 1)]
```

particulars of the

```
[('crime', 1), ('inquest', 1), ('habits', 1), ('voyage', 1)]
```

the death of

```
[('the', 8), ('sir', 8), ('captain', 3), ('their', 2), ('this', 2), ('his', 2), ('her', 2), ('dr', 2), ('sherlock', 1), ('mrs', 1), ('a', 1), ('van', 1), ('that', 1), ('drebber', 1), ('mr', 1), ('stangerson', 1), ('two', 1), ('selden', 1), ('one', 1), ('wallenstein', 1)]
```

one of the

```
[('most', 23), ('best', 3), ('main', 3), ('oldest', 2), ('greatest', 2), ('numerous', 2), ('corners', 2), ('largest', 2), ('papers', 2), ('richest', 2), ('servants', 2), ('moor', 2), ('finest', 2), ('upper', 2), ('very', 2), ('broad', 2), ('side', 2), ('highest', 2), ('australian', 1), ('great', 1)]
```

so far as

```
[('i', 17), ('to', 8), ('that', 2), ('the', 2), ('it', 2), ('was', 1), ('we', 1), ('they', 1), ('he', 1), ('you', 1), ('a', 1), ('your', 1), ('this', 1), ('his', 1), ('she', 1)]
```

You see, they look as more precise, but tables are just smaller. You can't use them to generate rubbish. 1st-order tables big, but "less precise".

And here I test some 3-words queries, like as if they inputted by user:

```
"i can tell"

* third order. for sequence: i can tell
you 66%
my 16%
them 16%
* second order. for sequence: can tell
you 23%
us 17%
me 11%
your 11%
this 5%
* first order. for word: tell
```

you 35%
me 25%
us 6%
him 5%
the 4%

"she was a"

* third order. for sequence: she was a
blonde 12%
child 6%
passenger 6%
free 6%
very 6%

* second order. for sequence: was a
very 4%
man 3%
small 3%
long 2%
little 2%

* first order. for word: a
very 2%
man 2%
little 2%
few 2%
small 1%

"he was a"

* third order. for sequence: he was a
man 11%
very 7%
young 3%
tall 3%
middle 2%

* second order. for sequence: was a
very 4%
man 3%
small 3%
long 2%
little 2%

* first order. for word: a
very 2%
man 2%
little 2%
few 2%
small 1%

"i did not"

* third order. for sequence: i did not
know 22%
say 9%
even 3%
tell 3%
mind 3%

* second order. for sequence: did not
know 11%
take 3%
wish 3%
go 2%
say 2%

* first order. for word: not
a 4%
be 4%
have 3%
been 3%
to 2%

"all that she"

* third order. for sequence: all that she
said 100%

* second order. for sequence: that she
had 22%
was 19%
would 7%
is 5%
has 5%

* first order. for word: she
was 12%
had 10%
is 5%
said 3%
would 3%

"have not been"

* third order. for sequence: have not been
able 25%
here 25%
employed 12%
shadowed 12%
personally 12%

* second order. for sequence: not been
for 9%
in 6%
there 6%
a 4%
slept 4%

* first order. for word: been
a 5%
in 4%
the 2%
so 2%
taken 1%

"who wanted to"

* third order. for sequence: who wanted to
see 100%

* second order. for sequence: wanted to
see 15%

know 15%
speak 10%

ask 10%
hide 5%

* first order. for word: to

the 11%

be 6%

me 4%

his 2%

my 2%

"he wanted to"

* third order. for sequence: he wanted to
know 50%

do 50%

* second order. for sequence: wanted to

see 15%

know 15%

speak 10%

ask 10%

hide 5%

* first order. for word: to

the 11%

be 6%

me 4%

his 2%

my 2%

"wanted to do"

* third order. for sequence: wanted to do
the 100%

* second order. for sequence: to do

with 23%

so 14%

it 10%

the 4%
and 3%
* first order. for word: do
you 24%
not 20%
with 6%
so 4%
it 4%

"it is just"

* third order. for sequence: it is just
possible 42%
as 33%
killing 4%
such 4%
in 4%
* second order. for sequence: is just
possible 25%
as 22%
the 8%
a 8%
to 5%
* first order. for word: just
as 13%
now 5%
a 4%
to 3%
the 2%

"you will find"

* third order. for sequence: you will find
that 20%
it 20%
me 8%
the 8%
a 6%
* second order. for sequence: will find
it 19%
that 17%
the 9%
me 7%
a 5%
* first order. for word: find
that 13%
the 10%
it 9%
out 8%
a 6%

```

"you shall"

* second order. for sequence: you shall
have 15%
see 12%
know 12%
hear 9%
not 9%
* first order. for word: shall
be 20%
have 7%
not 4%
see 3%
i 3%

"proved to be"

* third order. for sequence: proved to be
a 42%
the 14%
too 4%
something 4%
of 4%
* second order. for sequence: to be
a 11%
the 3%
in 3%
able 2%
so 2%
* first order. for word: be
a 7%
the 4%
in 2%
of 2%
no 2%

```

Perhaps, results from all 3 tables can be combined, with the data from 3rd order table used in highest priority (or weight).

And this is it — this can be shown to user. Aside of Conan Doyle works, your software can collect user's input to adapt itself for user's lexicon, slang, memes, etc. Of course, user's "tables" should be used with highest priority.

I have no idea, what Apple/Android devices using for hints generation, when user input text, but this is what I would use as a first idea.

As a bonus, this can be used for language learners, to get the idea, how a word is used in a sentence.

3.2.3 Further work

Comma can be a separator as well, etc...

3.2.4 The files

... including Conan Doyle's stories (2.5M). <https://github.com/DennisYurichev/Math-for-programmers/tree/master/prob/markov>. Surely, any other texts can be used, in any language...

Another related post is about typos: https://yurichev.com/blog/fuzzy_string/.

3.2.5 Read more

[Meaningful Random Headlines by Markov Chain](#)

[A discussion at hacker news](#)

3.3 random.choices() in Python 3

This is a very useful function⁴: weights (or probabilities) can be added.

(I used it in my Markov chains example (3.2).)

For example:

```
#!/usr/bin/env python3
import random

for i in range(1000):
    print (random.choices("123", [25, 60, 15]))
```

Let's generate 1000 random numbers in 1..3 range:

```
$ python3 tst.py | sort | uniq -c
  234 ['1']
  613 ['2']
  153 ['3']
```

"1" is generated in 25% of cases, "2" in 60% and "3" in 15%. Well, almost.

Here is another use of it.

You know, when you send an email, the final destination is a server somewhere. But it may be irresponsible. So network engineers add additional servers, "relays", which can hold your email for some time.

For example, what is about gmail.com?

```
% dig gmail.com MX
...
;; ANSWER SECTION:
gmail.com.          3600    IN      MX      5 gmail-smtp-in.l.google.com.
```

⁴<https://docs.python.org/3/library/random.html#random.choices>

gmail.com.	3600	IN	MX	10	alt1.gmail-smtp-in.l.google.com.
gmail.com.	3600	IN	MX	20	alt2.gmail-smtp-in.l.google.com.
gmail.com.	3600	IN	MX	30	alt3.gmail-smtp-in.l.google.com.
gmail.com.	3600	IN	MX	40	alt4.gmail-smtp-in.l.google.com.
...					

The first server is primary (marked with 5). Other 4 (alt...) are relays. They can hold emails for user@gmail.com if the main server is down. Of course, relays also can be down. So an MTA (Message transfer agent) tries to send an email via the first server in list, then via the second, etc. If all are down, MTA is waiting for some time (not infinitely).

See also: https://en.wikipedia.org/wiki/MX_record.

A number (5/10/20/30/40) is priority:

MX records contain a preference indication that MUST be used in sorting if more than one such record appears (see below). Lower numbers are more preferred than higher ones. If there are multiple destinations with the same preference and there is no clear reason to favor one (e.g., by recognition of an easily reached address), then the sender-SMTP MUST randomize them to spread the load across multiple mail exchangers for a specific organization.

(<https://tools.ietf.org/html/rfc5321>)

Now if you want your MTA be polite, you can make it poke relays with some probability, unloading the main mail server. In any case, the internal network withing Google is way better than a link between you and any of these mail servers. And it would be OK to drop an email to any of these mail servers listed in MX records.

This is how a destination server can be chosen:

```
random.choices(range(4), weights=[1/5, 1/10, 1/20, 1/40])
```

I'm using reciprocal weights (1/x) because the lower priority, the higher probability it is to be chosen.

What if I want to send 100 emails to someone@gmail.com?

```
>>> [random.choices(range(4), weights=[1/5, 1/10, 1/20, 1/40])[0] for x in range(100)]
[1, 1, 2, 1, 0, 2, 1, 0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1, 1, 0, 1, 0, 2, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 3, 0, 0, 2, 1, 0, 0, 0, 0, 1, 2, 2, 1, 0, 0, 0, 1, 2, 0, 0, 0, 0, 0, 0, 0, 3, 0, 0, 0, 1, 0, 2, 1, 0, 0, 2, 0, 0, 0, 3, 2, 0, 1, 2, 0, 1, 1, 3, 1, 1, 1, 1]
```


1000? (I'm using the `collections.Counter`⁵ here for gathering statistics).

```
>>> Counter([random.choices(range(4), weights=[1/5, 1/10, 1/20, 1/40])[0] for x
in range(1000)])
Counter({0: 535, 1: 268, 2: 129, 3: 68})
```

535 emails will be sent via the first (primary) mail server, 268/129/68 – via corresponding relays.

This is probably not how MTAs usually operates, but this is how it could be done.

⁵<https://docs.python.org/3/library/collections.html#collections.Counter>

Chapter 4

Combinatorics

4.1 Soldering a headphones cable

This is a real story: I tried to repair my headphone's cable, soldering 3 wires with minijack together. But which is left? Right? I couldn't even find ground wire. I asked myself, how many ways there are to solder 3 wires to 3-pin minijack? I could try them all and pick the combination that sounds best.

With Python's `itertools` module this is just:

```
import itertools

wires=["red", "green", "blue"]

for i in itertools.permutations(wires):
    print i
```

```
('red', 'green', 'blue')
('red', 'blue', 'green')
('green', 'red', 'blue')
('green', 'blue', 'red')
('blue', 'red', 'green')
('blue', 'green', 'red')
```

(Just 6 ways.)

What if there are 4 wires?

```
import itertools

wires=["red", "green", "blue", "yellow"]

for i in itertools.permutations(wires):
    print i
```

```
('red', 'green', 'blue', 'yellow')
('red', 'green', 'yellow', 'blue')
('red', 'blue', 'green', 'yellow')
('red', 'blue', 'yellow', 'green')
```

```
('red', 'yellow', 'green', 'blue')
('red', 'yellow', 'blue', 'green')
('green', 'red', 'blue', 'yellow')
('green', 'red', 'yellow', 'blue')
('green', 'blue', 'red', 'yellow')
('green', 'blue', 'yellow', 'red')
('green', 'yellow', 'red', 'blue')
('green', 'yellow', 'blue', 'red')
('blue', 'red', 'green', 'yellow')
('blue', 'red', 'yellow', 'green')
('blue', 'green', 'red', 'yellow')
('blue', 'green', 'yellow', 'red')
('blue', 'yellow', 'red', 'green')
('blue', 'yellow', 'green', 'red')
('yellow', 'red', 'green', 'blue')
('yellow', 'red', 'blue', 'green')
('yellow', 'green', 'red', 'blue')
('yellow', 'green', 'blue', 'red')
('yellow', 'blue', 'red', 'green')
('yellow', 'blue', 'green', 'red')
```

(24 ways.)

This is what is called *permutation* in combinatorics.

4.2 Vehicle license plate

There was a hit and run. And you're police officer. And this is what your only witness says about 4-digit license plate of the guilty vehicle:

```
There was 13: 1 and 3 together, I'm sure, but not sure where, 13 as first 2
  digits, last 2 digits or in the middle.
And there was also 6, or maybe 8, or maybe even 9, not sure which, but one of
  them.
```

Combinatorics textbooks are abound with exercises like this: can you enumerate all possible 4-digit numbers constrained in this way?

```
import itertools

part1_list=["13"]
part2_list=["6", "8", "9"]
part3_list=["0", "1", "2", "3", "4", "5", "6", "7", "8", "9"]

for i in itertools.product(part1_list, part2_list, part3_list):
    for j in itertools.permutations(i):
        print "".join(j)
```

```
1360
1306
6130
6013

...

9139
9913
9139
9913
```

(180 numbers)

This is something you can query registered vehicle database with...

4.3 Forgotten password

You forgot a password, but this is what you remember: there was a name of your parent, or wife, or one of children. Also, someone's year of birth. And one punctuation character, which are so recommended in passwords. Can you enumerate all possible passwords?

```
import itertools

part1_list=["jake", "melissa", "oliver", "emily"]
part2_list=["1987", "1954", "1963"]
part3_list=["!", "@", "#", "$", "%", "&", "*", "-", "=", "_", "+", ".", ","]

for part1 in part1_list:
    for part2 in part2_list:
        for part3 in part3_list:
            l=[part1, part2, part3]
            for i in list(itertools.permutations(l)):
                print "".join(i)
```

```
jake1987!
jake!1987
1987jake!
1987!jake
!jake1987

...

1963emily,
1963,emily
,emily1963
,1963emily
```

(936 of them in total)

But nested for's are not aesthetically pleasing. They can be replaced with "cartesian product" operation:

```
import itertools

part1_list=["jake", "melissa", "oliver", "emily"]
part2_list=["1987", "1954", "1963"]
part3_list=["!", "@", "#", "$", "%", "&", "*", "-", "=", "_", "+", ".", ","]

for l in itertools.product(part1_list, part2_list, part3_list):
    for i in list(itertools.permutations(l)):
        print "".join(i)
```

And this is a way to memorize it: the length of the final result equals to lengths of all input lists multiplied with each other (like "product").

```
import itertools

part1_list=["jake", "melissa", "oliver", "emily"] # 4 elements
part2_list=["1987", "1954", "1963"] # 3 elements
part3_list=["!", "@", "#", "$", "%", "&", "*", "-", "=", "_", "+", ".", ","] # 13 elements

for l in itertools.product(part1_list, part2_list, part3_list):
    print l
```

```
('jake', '1987', '!')
('jake', '1987', '@')
('jake', '1987', '#')
('jake', '1987', '$')
('jake', '1987', '%')
('jake', '1987', '&')
('jake', '1987', '*')

...

('emily', '1963', '*')
('emily', '1963', '-')
('emily', '1963', '=')
('emily', '1963', '_')
('emily', '1963', '+')
('emily', '1963', '.')
('emily', '1963', ',')
```

$4 \times 3 \times 13 = 156$, and this is a size of a list, to be permuted...

Now the new problem: some Latin characters may be uppercased, some are lowercased. I'll add another "cartesian product" operation to alter a final string in all possible ways:

```
import itertools, string
```

```

part1_list=["jake", "melissa", "oliver", "emily"]
part2_list=["1987", "1954", "1963"]
part3_list=["!", "@", "#", "$", "%", "&", "*", "-", "=", "_", "+", ".", ", "]

for l in itertools.product(part1_list, part2_list, part3_list):
    for i in list(itertools.permutations(l)):
        s="".join(i)
        t=[]
        for char in s:
            if char.isalpha():
                t.append([string.lower(char), string.upper(char)])
            else:
                t.append([char])
        for q in itertools.product(*t):
            print "".join(q)

```

```

JAke1987!
JAkE1987!
JAKe1987!
JAKE1987!
jake!1987
jakE!1987
jaKe!1987
jaKE!1987

...

,1963eMIly
,1963eMIly
,1963eMILy
,1963eMILY
,1963Emily
,1963EmilY
,1963EmiLy

```

(56160 passwords)

Now leetspeak¹ This is somewhat popular only among youngsters, but still, this is what people of all age groups do: replacing "o" with "0" in their passwords, "e" with "3", etc. Let's add this as well:

```

import itertools, string

part1_list=["jake", "melissa", "oliver", "emily"]
part2_list=["1987", "1954", "1963"]
part3_list=["!", "@", "#", "$", "%", "&", "*", "-", "=", "_", "+", ".", ", "]

for l in itertools.product(part1_list, part2_list, part3_list):
    for i in list(itertools.permutations(l)):

```

¹urbandictionary.com

```

s="".join(i)
t=[]
for char in s:
    if char.isalpha():
        to_be_appended=[string.lower(char), string.upper(char)]
        if char.lower()=='e':
            to_be_appended.append('3')
        elif char.lower()=='i':
            to_be_appended.append('1')
        elif char.lower()=='o':
            to_be_appended.append('0')
        t.append(to_be_appended)
    else:
        t.append([char])
for q in itertools.product(*t):
    print "".join(q)

```

```

jake1987!
jakE1987!
jak31987!
jaKe1987!
jaKE1987!
jaK31987!

...

,1963EM11Y
,1963EM1Ly
,1963EM1LY
,19633mily
,19633milY
,19633miLy

```

(140400 passwords)

Obviously, you can't try all 140400 passwords on Facebook, Twitter or any other well-protected internet service. But this is a piece of cake to brute-force them all on password protected RAR-archive or feed them all to John the Ripper, HashCat, etc.

All the files: <https://github.com/DennisYurichev/Math-for-programmers/tree/master/comb/password>.

Now let's use combinations from itertools Python package.

Let's say, you remember that your password has maybe your name, maybe name of your wife, your year of birth, or her, and maybe couple of symbols like !, \$, ^.

```
import itertools
```



```
parts=["den", "xenia", "1979", "1985", "secret", "!", "$", "^"]

for i in range(1, 6): # 1..5
    for combination in itertools.combinations(parts, i):
        print "".join(combination)
```

Here we enumerate all combinations of given strings, 1-string combinations, then 2-, up to 5-string combinations. No string can appear twice.

```
...
denxenia1979
denxenia1985
denxeniasecret
denxenia!
denxenia$
denxenia^
den19791985
den1979secret
den1979!
...
xenia1985secret$^
xenia1985!$^
xeniasecret!$^
19791985secret!$
19791985secret!^
...
```

(218 passwords)

Now let's permute all string in all possible ways:

```
import itertools

parts=["den", "xenia", "1979", "1985", "secret", "!", "$", "^"]

for i in range(1, 6): # 1..5
    for combination in itertools.combinations(parts, i):
        for permutation in itertools.permutations(list(combination)):
            print "".join(permutation)
```

```
...
^den
xenia1979
1979xenia
xenia1985
1985xenia
xeniasecret
```

```
secretxenia
xenia!
!xenia
...
^!$1985secret
^!$secret1985
^$1985secret!
^$1985!secret
^$secret1985!
^$secret!1985
^$!1985secret
^$!secret1985
```

(8800 passwords)

And finally, let's alter all Latin characters in lower/uppercase ways and add leetspeek, as I did before:

```
import itertools, string

parts=["den", "xenia", "1979", "1985", "!", "$", "^"]

for i in range(1, 6): # 1..5
    for combination in itertools.combinations(parts, i):
        for permutation in itertools.permutations(list(combination)):
            s="".join(permutation)
            t=[]
            for char in s:
                if char.isalpha():
                    to_be_appended=[string.lower(char), string.upper(char)]
                    if char.lower()=='e':
                        to_be_appended.append('3')
                    elif char.lower()=='i':
                        to_be_appended.append('1')
                    elif char.lower()=='o':
                        to_be_appended.append('0')
                    t.append(to_be_appended)
                else:
                    t.append([char])
            for q in itertools.product(*t):
                print "".join(q)
```

```
...
dEnxenia
dEnxeniA
dEnxenIa
...
D3nx3N1a
D3nx3N1A
```

```
D3nXenia
D3nXeniA
D3nXeniIa
...
^$1979!1985
^$19851979!
^$1985!1979
^$!19791985
^$!19851979
```

(1,348,657 passwords)

Again, you can't try to crack remote server with so many attempts, but this is really possible for password-protected archive, known hash, etc...

4.4 Executable file watermarking/steganography using Lehmer code and factorial number system

In short: how to hide information not in objects, but in *order* of objects.

Almost any binary executable file has text strings like (these are from CRT):

```
.rdata:0040D398 aR6002FloatingP:
.rdata:0040D398          text "UTF-16LE", 'R6002',0Dh,0Ah
.rdata:0040D398          text "UTF-16LE", '- floating point support not
    loaded',0Dh,0Ah,0
.rdata:0040D3F2          align 8
.rdata:0040D3F8 aR6008NotEnough:
.rdata:0040D3F8          text "UTF-16LE", 'R6008',0Dh,0Ah
.rdata:0040D3F8          text "UTF-16LE", '- not enough space for
    arguments',0Dh,0Ah,0
.rdata:0040D44C          align 10h
.rdata:0040D450 aR6009NotEnough:
.rdata:0040D450          text "UTF-16LE", 'R6009',0Dh,0Ah
.rdata:0040D450          text "UTF-16LE", '- not enough space for
    environment',0Dh,0Ah,0
.rdata:0040D4A8 aR6010AbortHasB:
.rdata:0040D4A8          text "UTF-16LE", 'R6010',0Dh,0Ah
.rdata:0040D4A8          text "UTF-16LE", '- abort() has been called',0Dh
    ,0Ah,0
.rdata:0040D4EE          align 10h
.rdata:0040D4F0 aR6016NotEnough:
.rdata:0040D4F0          text "UTF-16LE", 'R6016',0Dh,0Ah
.rdata:0040D4F0          text "UTF-16LE", '- not enough space for thread
    data',0Dh,0Ah,0
.rdata:0040D548 aR6017Unexpecte:
.rdata:0040D548          text "UTF-16LE", 'R6017',0Dh,0Ah
.rdata:0040D548          text "UTF-16LE", '- unexpected multithread lock
    error',0Dh,0Ah,0
```

```
.rdata:0040D5A2                align 8
.rdata:0040D5A8 aR6018Unexpecte:
.rdata:0040D5A8                text "UTF-16LE", 'R6018',0Dh,0Ah
.rdata:0040D5A8                text "UTF-16LE", '- unexpected heap error',0Dh,0
    Ah,0
.rdata:0040D5EA                align 10h
```

Can we hide some information there? Not in string themselves, but in *order* of strings? Given the fact, that compiler doesn't guarantee at all, in which order the strings will be stored in object/executable file.

Let's say, we've got 26 text strings, and we can swap them how we want, because their order isn't important at all. All possible permutations of 26 objects is $26! = 403291461126605635584000000$.

How much information can be stored here? $\log_2(26!) = 88$, i.e., 88 bits or 11 bytes!

11 bytes of your data can be converted to a (big) number and back, OK.

What is next? Naive way is: enumerate all permutations of 26 objects, number each, find permutation of the number we've got and permute 26 text strings, store to file and that's it. But we can't iterate over $403291461126605635584000000$ permutations.

This is where factorial number system² and Lehmer code³ can be used. In short, for all my non-mathematically inclined readers, this is a way to find a permutation of specific number without use of any significant resources. And back: given specific permutation, we can find its number.

This piece of code I've copy-pasted from <https://gist.github.com/lukmdo/7049748> and reworked slightly:

```
# python 3.x

import math

def iter_perm(base, *rargs):
    """
    :type base: list
    :param rargs: range args [start,] stop[, step]
    :rtype: generator
    """
    if not rargs:
        rargs = [math.factorial(len(base))]
    for i in range(*rargs):
        yield perm_from_int(base, i)

def int_from_code(code):
    """
    :type code: list
    :rtype: int
    """
    num = 0
    for i, v in enumerate(reversed(code), 1):
        num *= i
        num += v
```

²https://en.wikipedia.org/wiki/Factorial_number_system

³https://en.wikipedia.org/wiki/Lehmer_code

```

    return num

def code_from_int(size, num):
    """
    :type size: int
    :type num: int
    :rtype: list
    """
    code = []
    for i in range(size):
        num, j = divmod(num, size - i)
        code.append(j)

    return code

def perm_from_code(base, code):
    """
    :type base: list
    :type code: list
    :rtype: list
    """

    perm = base.copy()
    for i in range(len(base) - 1):
        j = code[i]
        if i != i+j:
            print ("swapping %d, %d" % (i, i+j))
            perm[i], perm[i+j] = perm[i+j], perm[i]

    return perm

def perm_from_int(base, num):
    """
    :type base: list
    :type num: int
    :rtype: list
    """

    code = code_from_int(len(base), num)
    print ("Lehmer code=", code)
    return perm_from_code(base, code)

def code_from_perm(base, perm):
    """
    :type base: list
    :type perm: list
    :rtype: list
    """

    p = base.copy()

```

```

n = len(base)
pos_map = {v: i for i, v in enumerate(base)}

w = []
for i in range(n):
    d = pos_map[perm[i]] - i
    w.append(d)

    if not d:
        continue
    t = pos_map[perm[i]]
    pos_map[p[i]], pos_map[p[t]] = pos_map[p[t]], pos_map[p[i]]
    p[i], p[t] = p[t], p[i]

return w

def int_from_perm(base, perm):
    """
    :type base: list
    :type perm: list
    :rtype: int
    """
    code = code_from_perm(base, perm)
    return int_from_code(code)

def bin_string_to_number(s):
    rt=0
    for i, c in enumerate(s):
        rt=rt<<8
        rt=rt+ord(c)
    return rt

def number_to_bin_string(n):
    rt=""
    while True:
        r=n & 0xff
        rt=rt+chr(r)
        n=n>>8
        if n==0:
            break
    return rt[::-1]

s="HelloWorld"
print ("s=", s)
num=bin_string_to_number (s)
print ("num=", num)
perm=perm_from_int(list(range(26)), num)
print ("permutation/order=", perm)

num2=int_from_perm(list(range(26)), [14, 17, 9, 19, 11, 16, 23, 0, 2, 13, 20,

```

```
18, 21, 24, 10, 1, 22, 4, 7, 6, 15, 12, 5, 3, 8, 25])
print ("recovered num=", num2)
s2=number_to_bin_string(num2)
print ("recovered s=", s2)
```

I'm encoding a "HelloWorld" binary string (in fact, any 11 bytes can be used) into a number. Number is then converted into Lehmer code. Then the `perm_from_code()` function permute initial *order* according to the input Lehmer code:

```
s= HelloWorld
num= 341881320659934023674980
Lehmer code= [14, 16, 7, 16, 7, 11, 17, 7, 1, 4, 10, 7, 9, 11, 6, 2, 6, 1, 2, 4,
0, 0, 0, 0, 0, 0]
swapping 0, 14
swapping 1, 17
swapping 2, 9
swapping 3, 19
swapping 4, 11
swapping 5, 16
swapping 6, 23
swapping 7, 14
swapping 8, 9
swapping 9, 13
swapping 10, 20
swapping 11, 18
swapping 12, 21
swapping 13, 24
swapping 14, 20
swapping 15, 17
swapping 16, 22
swapping 17, 18
swapping 18, 20
swapping 19, 23
permutation/order= [14, 17, 9, 19, 11, 16, 23, 0, 2, 13, 20, 18, 21, 24, 10, 1,
22, 4, 7, 6, 15, 12, 5, 3, 8, 25]
```

This is it: [14, 17, 9, 19, 11, 16, 23, 0, 2, 13, 20, 18, 21, 24, 10, 1, 22, 4, 7, 6, 15, 12, 5, 3, 8, 25]. First put 14th string, then 17s string, then 9th one, etc.

Now you've got a binary file from someone and want to read watermark from it. Get an order of strings from it and convert it back to binary string:

```
recovered num= 341881320659934023674980
recovered s= HelloWorld
```

If you have more text strings (not unusual for most executable files), you can encode more.

100 strings: $\log_2(100!) = 524$ bits = 65 bytes.

1000 strings: $\log_2(1000!) = 8529$ bits = 1066 bytes! You can store some text here!

How would you force a C/C++ compiler to make specific order of text strings? This is crude, but workable:

```
char blob []="hello1\0hello2\0";
char *msg1=blob;
char *msg2=blob+8;

printf ("%s\n", msg1);
printf ("%s\n", msg2);
```

They can be even aligned on 16-byte border.

... or they can be placed into `./asm` assembly file and compiled into `./obj` and then linked to your program.

... or you can swap text strings in already compiled executable and correct their addresses in corresponding instructions. If an executable file is not packed/obfuscated, this is possible.

Aside of order of text strings, you can try to hack a linker and reorder object files in the final executable. Of course, no one cares about its order. And go figure out, what is hidden there.

Surely, hidden data can be encrypted, checksum or MAC can be added, etc.

Other ideas to consider: reorder functions and fix all addresses, reorder basic blocks within a function, register allocator hacking, etc.

Links I find helpful in understanding factorial number system and Lehmer code, aside of Wikipedia:

- <https://gist.github.com/lukmdo/7049748>
- <https://github.com/scmorse/permutils/blob/master/src/permutils.js>
- <http://2ality.com/2013/03/permutations.html>
- <http://www.keithschwarz.com/interesting/code/factoradic-permutation/FactoradicPermutation>

4.5 De Bruijn sequences; leading/trailing zero bits counting

4.5.1 Introduction

Let's imagine there is a very simplified code lock accepting 2 digits, but it has no "enter" key, it just checks 2 last entered digits. Our task is to brute force each 2-digit combination. Naïve method is to try 00, 01, 02 ... 99. That require $2*100=200$ key pressings. Will it be possible to reduce number of key pressings during brute-force? It is indeed so, with the help of De Bruijn sequences. We can generate them for the code lock, using Wolfram Mathematica:

```
In []:= DeBruijnSequence[{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}, 2]
Out []= {6, 8, 6, 5, 4, 3, 2, 1, 7, 8, 7, 1, 1, 0, 9, 0, 8, 0, 6, 6, \
0, 5, 5, 0, 4, 4, 0, 3, 3, 0, 2, 7, 2, 2, 0, 7, 7, 9, 8, 8, 9, 9, 7, \
0, 0, 1, 9, 1, 8, 1, 6, 1, 5, 1, 4, 1, 3, 7, 3, 1, 2, 9, 2, 8, 2, 6, \
2, 5, 2, 4, 7, 4, 2, 3, 9, 3, 8, 3, 6, 3, 5, 7, 5, 3, 4, 9, 4, 8, 4, \
6, 7, 6, 4, 5, 9, 5, 8, 5, 6, 9}
```

The result has exactly 100 digits, which is 2 times less than our initial idea can offer. By scanning visually this 100-digits array, you'll find any number in 00..99 range. All numbers are overlapped with each other: second half of each number is also first half of the next number, etc.

Here is another. We need a sequence of binary bits with all 3-bit numbers in it:

```
In [] := DeBruijnSequence[{0, 1}, 3]
Out [] = {1, 0, 1, 0, 0, 0, 1, 1}
```

Sequence length is just 8 bits, but it has all binary numbers in 000..111 range. You may visually spot 000 in the middle of sequence. 111 is also present: two first bits of it at the end of sequence and the last bit is in the beginning. This is so because De Bruijn sequences are cyclic.

There is also visual demonstration: <http://demonstrations.wolfram.com/DeBruijnSequences/>.

4.5.2 Trailing zero bits counting

In the [Wikipedia article about De Bruijn sequences](#) we can find:

The symbols of a De Bruijn sequence written around a circular object (such as a wheel of a robot) can be used to identify its angle by examining the n consecutive symbols facing a fixed point.

Indeed: if we know De Bruijn sequence and we observe only part of it (any part), we can deduce exact position of this part within sequence.

Let's see, how this feature can be used.

Let's say, there is a need to detect position of input bit within 32-bit word. For 0x1, the algorithm should report 1. 2 for 0x2. 3 for 0x4. And 31 for 0x80000000.

The result is in 0..31 range, so the result can be stored in 5 bits.

We can construct binary De Bruijn sequence for all 5-bit numbers:

```
In [] := tmp = DeBruijnSequence[{0, 1}, 5]
Out [] = {1, 1, 1, 0, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 0, 0, 0, 1, 0, 1,
  0, 0, 1, 0, 0, 0, 0, 0}

In [] := BaseForm[FromDigits[tmp, 2], 16]
Out [] := e6bec520
```

Let's also recall that division some number by 2^n number is the same thing as shifting it by n bits. So if you divide 0xe6bec520 by 1, the result is not shifted, it is still the same. If if divide 0xe6bec520 by 4 (2^2), the result is shifted by 2 bits. We then take result and isolate lowest 5 bits. This result is unique number for each input. Let's shift 0xe6bec520 by all possible count number, and we'll get all possible last 5-bit values:

```
In [] := Table[BitAnd[BitShiftRight[FromDigits[tmp, 2], i], 31], {i, 0, 31}]
Out [] = {0, 16, 8, 4, 18, 9, 20, 10, 5, 2, 17, 24, 12, 22, 27, 29, \
  30, 31, 15, 23, 11, 21, 26, 13, 6, 19, 25, 28, 14, 7, 3, 1}
```

The table has no duplicates:

```
In []:= DuplicateFreeQ [%]
Out []= True
```

Using this table, it's easy to build a *magic* table. OK, now working C example:

```
#include <stdint.h>
#include <stdio.h>

int magic_tbl[32];

// returns single bit position counting from LSB
// not working for i==0
int bitpos (uint32_t i)
{
    return magic_tbl[(0xe6bec520/i) & 0x1F];
};

int main()
{
    // construct magic table
    // may be omitted in production code
    for (int i=0; i<32; i++)
        magic_tbl[(0xe6bec520/(1<<i)) & 0x1F]=i;

    // test
    for (int i=0; i<32; i++)
    {
        printf ("input=0x%x, result=%d\n", 1<<i, bitpos (1<<i));
    };
};
```

Here we feed our bitpos() function with numbers in 0..0x80000000 range and we got:

```
input=0x1, result=0
input=0x2, result=1
input=0x4, result=2
input=0x8, result=3
input=0x10, result=4
input=0x20, result=5
input=0x40, result=6
input=0x80, result=7
input=0x100, result=8
input=0x200, result=9
input=0x400, result=10
input=0x800, result=11
input=0x1000, result=12
input=0x2000, result=13
```

```
input=0x4000, result=14
input=0x8000, result=15
input=0x10000, result=16
input=0x20000, result=17
input=0x40000, result=18
input=0x80000, result=19
input=0x100000, result=20
input=0x200000, result=21
input=0x400000, result=22
input=0x800000, result=23
input=0x1000000, result=24
input=0x2000000, result=25
input=0x4000000, result=26
input=0x8000000, result=27
input=0x10000000, result=28
input=0x20000000, result=29
input=0x40000000, result=30
input=0x80000000, result=31
```

The `bitpos()` function actually counts trailing zero bits, but it works only for input values where only one bit is set. To make it more practical, we need to devise a method to drop all leading bits except of the last one. This method is very simple and well-known:

```
input & (-input)
```

This bit twiddling hack can solve the job. Feeding `0x11` to it, it will return `0x1`. Feeding `0xFFFF0000`, it will return `0x10000`. In other words, it leaves lowest significant bit of the value, dropping all others.

It works because negated value in two's complement environment is the value with all bits flipped but also 1 added (because there is a zero in the middle of ring). For example, let's take `0xF0`. `-0xF0` is `0x10` or `0xFFFFF10`. ANDing `0xF0` and `0xFFFFF10` will produce `0x10`.

Let's modify our algorithm to support true trailing zero bits count:

```
#include <stdint.h>
#include <stdio.h>

int magic_tbl[32];

// not working for i==0
int tzcnt (uint32_t i)
{
    uint32_t a=i & (-i);
    return magic_tbl[(0xe6bec520/a) & 0x1F];
};

int main()
{
    // construct magic table
```

```

// may be omitted in production code
for (int i=0; i<32; i++)
    magic_tbl[(0xe6bec520/(1<<i)) & 0x1F]=i;

// test:
printf ("%d\n", tzcnt (0xFFFF0000));
printf ("%d\n", tzcnt (0xFFFF0010));
};

```

It works!

```

16
4

```

But it has one drawback: it uses division, which is slow. Can we just multiply De Bruijn sequence by the value with the bit isolated instead of dividing sequence? Yes, indeed. Let's check in Mathematica:

```

In [] := BaseForm[16^e6bec520*16^80000000, 16]
Out [] := 0x735f629000000000

```

The result is just too big to fit in 32-bit register, but can be used. MUL/IMUL instruction 32-bit x86 CPUs stores 64-bit result into two 32-bit registers pair, yes. But let's suppose we would like to make portable code which will work on any 32-bit architecture. First, let's again take a look on De Bruijn sequence Mathematica first produced:

```

In [] := tmp = DeBruijnSequence[{0, 1}, 5]
Out [] = {1, 1, 1, 0, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 0, 0, \
0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0}

```

There is exactly 5 bits at the end which can be dropped. The "magic" constant will be much smaller:

```

In [] := BaseForm[BitShiftRight[FromDigits[tmp, 2], 5], 16]
Out [] := 0x735f629

```

The "magic" constant is now "divided by 32 (or 1»5)". This means that the result of multiplication of some value with one isolated bit by new magic number will also be smaller, so the bits we need will be stored at the high 5 bits of the result.

De Bruijn sequence is not broken after 5 lowest bits dropped, because these zero bits are "relocated" to the start of the sequence. Sequence is cyclic after all.

```

#include <stdint.h>
#include <stdio.h>

int magic_tbl[32];

```

```

// not working for i==0
int tzcnt (uint32_t i)
{
    uint32_t a=i & (-i);
    // 5 bits we need are stored in 31..27 bits of product, shift and isolate them
    // after multiplication:
    return magic_tbl[((0x735f629*a)>>27) & 0x1F];
};

int main()
{
    // construct magic table
    // may be omitted in production code
    for (int i=0; i<32; i++)
        magic_tbl[(0x735f629<<i >>27) & 0x1F]=i;

    // test:
    printf ("%d\n", tzcnt (0xFFFF0000));
    printf ("%d\n", tzcnt (0xFFFF0010));
};

```

4.5.3 Leading zero bits counting

This is almost the same task, but most significant bit must be isolated instead of lowest. This is typical algorithm for 32-bit integer values:

```

x |= x >> 1;
x |= x >> 2;
x |= x >> 4;
x |= x >> 8;
x |= x >> 16;

```

For example, 0x100 becomes 0x1ff, 0x1000 becomes 0x1fff, 0x20000 becomes 0x3ffff, 0x12340000 becomes 0x1ffffff. It works because all 1 bits are gradually propagated towards the lowest bit in 32-bit number, while zero bits at the left of most significant 1 bit are not touched.

It's possible to add 1 to resulting number, so it will becomes 0x2000 or 0x20000000, but in fact, since multiplication by magic number is used, these numbers are very close to each other, so there are no error.

This example I used in my reverse engineering exercise from 15-Aug-2015: <https://yurichev.com/blog/2015-aug-18/>.

```

int v[64]=
    { -1,31, 8,30, -1, 7,-1,-1, 29,-1,26, 6, -1,-1, 2,-1,
      -1,28,-1,-1, -1,19,25,-1, 5,-1,17,-1, 23,14, 1,-1,
        9,-1,-1,-1, 27,-1, 3,-1, -1,-1,20,-1, 18,24,15,10,
        -1,-1, 4,-1, 21,-1,16,11, -1,22,-1,12, 13,-1, 0,-1 };

int LZCNT(uint32_t x)
{

```

```
x |= x >> 1;
x |= x >> 2;
x |= x >> 4;
x |= x >> 8;
x |= x >> 16;
x *= 0x4badf0d;
return v[x >> 26];
}
```

This piece of code I took from [here](#). It is slightly different: the table is twice bigger, and the function returns -1 if input value is zero. The magic number I found using just brute-force, so the readers will not be able to google it, for the sake of exercise. (By the way, I've got 12,665,720 magic numbers which can serve this purpose. This is about 0.294

The code is tricky after all, and the moral of the exercise is that practicing reverse engineer sometimes may just observe input/outputs to understand code's behaviour instead of diving into it.

4.5.4 Performance

The algorithms considered are probably fastest known, they has no conditional jumps, which is very good for CPUs starting at RISCs. Newer CPUs has LZCNT and TZCNT instructions, even 80386 had BSF/BSR instructions which can be used for this: https://en.wikipedia.org/wiki/Find_first_set. Nevertheless, these algorithms can be still used on cheaper RISC CPUs without specialized instructions.

4.5.5 Applications

Number of leading zero bits is binary logarithm of value: 6.

These algorithms are also extensively used in chess engines programming, where each piece is represented as 64-bit bitmask (chess board has 64 squares): <http://chessprogramming.wikispaces.com/BitScan>.

There are more: https://en.wikipedia.org/wiki/Find_first_set#Applications.

4.5.6 Generation of De Bruijn sequences

De Bruijn graph is a graph where all values are represented as vertices (or nodes) and each edge (or link) connects two nodes which can be "overlapped". Then we need to visit each edge only once, this is called *eulerian path*. It is like the famous *task of seven bridges of Königsberg*: traveller must visit each bridge only once.

There are also simpler algorithms exist: https://en.wikipedia.org/wiki/De_Bruijn_sequence#Algorithm.

4.5.7 Other articles

At least these are worth reading: <http://supertech.csail.mit.edu/papers/debruijn.pdf>, <http://alexandria.tue.nl/repository/books/252901.pdf>, Wikipedia Article about De Bruijn sequences.

<https://chessprogramming.wikispaces.com/De+Bruijn+sequence>, <https://chessprogramming.wikispaces.com/De+Bruijn+Sequence+Generator>.

Chapter 5

Galois Fields, GF(2) and yet another explanation of CRC

5.1 What is wrong with checksum?

If you just sum up values of several bytes, two bit flips (increment one bit and decrement another bit) can give the same checksum. No good.

5.2 Division by prime

You can represent a file of buffer as a (big) number, then to divide it by prime. The remainder is then very sensitive to bit flips. For example, a prime $0x10015$ (65557).

Wolfram Mathematica:

```
In [] := divisor=16^^10015
Out [] = 65557

In [] := BaseForm[Mod[16^^abcdef1234567890, divisor], 16]
Out [] = d8c1

In [] := BaseForm[Mod[16^^abcdef0234567890, divisor], 16]
Out [] = bd31

In [] := BaseForm[Mod[16^^bbcdef1234567890, divisor], 16]
Out [] = 382b

In [] := BaseForm[Mod[16^^abcdee1234567890, divisor], 16]
Out [] = 1fd6

In [] := BaseForm[Mod[16^^abcdef0234567891, divisor], 16]
Out [] = bd32
```

This is what is called “avalanche effect” in cryptography: one bit flip of input can affect many bits of output. Go figure out which bits must be also flipped to preserve specific remainder.

You can build such a divisor in hardware, but it would require at least one adder or subtractor, you will have a carry-ripple problem in simple case, or you would have to create more complicated circuit.

5.3 (Binary) long division

Binary long division is in fact simpler than the paper-n-pencil algorithm taught in schools.

The algorithm is:

- 1) Allocate some “tmp” variable and copy dividend to it.
- 2) Pad divisor by zero bits at left so that **MSB**¹ of divisor is at the place of **MSB** of the value in tmp.
- 3) If the divisor is larger than tmp or equal, subtract divider from tmp and add 1 bit to the quotient. If the divisor is smaller than tmp, add 0 bit to the quotient.
- 4) Shift divisor right. If the divisor is 0, stop. Remainder is in tmp.
- 5) Goto 3

The following piece of code I've cypasted from somewhere:

```
unsigned int divide(unsigned int dividend, unsigned int divisor)
{
    unsigned int tmp = dividend;
    unsigned int denom = divisor;
    unsigned int current = 1;
    unsigned int answer = 0;

    if (denom > tmp)
        return 0;

    if (denom == tmp)
        return 1;

    // align divisor:
    while (denom <= tmp)
    {
        denom = denom << 1;
        current = current << 1;
    }

    denom = denom >> 1;
    current = current >> 1;

    while (current!=0)
    {
        printf ("current=%d, denom=%d\n", current, denom);
        if (tmp >= denom)
        {
```

¹Most Significant Bit


```

        tmp -= denom;
        answer |= current;
    }
    current = current >> 1;
    denom = denom >> 1;
}
printf ("tmp/remainder=%d\n", tmp); // remainder!
return answer;
}

```

(<https://github.com/DennisYurichev/Math-for-programmers/blob/master/GF2/div.c>)

Let's divide 1234567 by 813 and find remainder:

```

current=1024, denom=832512
current=512, denom=416256
current=256, denom=208128
current=128, denom=104064
current=64, denom=52032
current=32, denom=26016
current=16, denom=13008
current=8, denom=6504
current=4, denom=3252
current=2, denom=1626
current=1, denom=813
tmp/remainder=433
1518

```

5.4 (Binary) long division, version 2

Now let's say, you only need to compute a remainder, and throw away a quotient. Also, maybe you work on some kind BigInt values and you've got a function like `get_next_bit()` and that's it.

What we can do: tmp value will be shifted at each iteration, while divisor is not:

```

uint8_t *buf;
int buf_pos;
int buf_bit_pos;

int get_bit()
{
    if (buf_pos==-1)
        return -1; // end

    int rt=(buf[buf_pos] >> buf_bit_pos) & 1;
    if (buf_bit_pos==0)
    {
        buf_pos--;
    }
}

```

```

        buf_bit_pos=7;
    }
    else
        buf_bit_pos--;
    return rt;
};

uint32_t remainder_arith(uint32_t dividend, uint32_t divisor)
{
    buf=(uint8_t*)&dividend;
    buf_pos=3;
    buf_bit_pos=7;

    uint32_t tmp=0;

    for(;;)
    {
        int bit=get_bit();
        if (bit==-1)
        {
            printf ("exit. remainder=%d\n", tmp);
            return tmp;
        };

        tmp=tmp<<1;
        tmp=tmp|bit;

        if (tmp>=divisor)
        {
            printf ("%d greater or equal to %d\n", tmp, divisor);
            tmp=tmp-divisor;
            printf ("new tmp=%d\n", tmp);
        }
        else
            printf ("tmp=%d, can't subtract\n", tmp);
    };
}

```

(https://github.com/DennisYurichev/Math-for-programmers/blob/master/GF2/div_both.c)

Let's divide 1234567 by 813 and find remainder:

```

tmp=0, can't subtract
tmp=0, can't subtract
tmp=0, can't subtract
tmp=0, can't subtract
tmp=0, can't subtract
tmp=0, can't subtract
tmp=0, can't subtract
tmp=0, can't subtract
tmp=0, can't subtract

```

```

tmp=0, can't subtract
tmp=0, can't subtract
tmp=1, can't subtract
tmp=2, can't subtract
tmp=4, can't subtract
tmp=9, can't subtract
tmp=18, can't subtract
tmp=37, can't subtract
tmp=75, can't subtract
tmp=150, can't subtract
tmp=301, can't subtract
tmp=602, can't subtract
1205 greater or equal to 813
new tmp=392
tmp=785, can't subtract
1570 greater or equal to 813
new tmp=757
1515 greater or equal to 813
new tmp=702
1404 greater or equal to 813
new tmp=591
1182 greater or equal to 813
new tmp=369
tmp=738, can't subtract
1476 greater or equal to 813
new tmp=663
1327 greater or equal to 813
new tmp=514
1029 greater or equal to 813
new tmp=216
tmp=433, can't subtract
exit. remainder=433

```

5.5 Shortest possible introduction into GF(2)

There is a difference between digit and number. Digit is a symbol, number is a group of digits. 0 can be both digit and number.

Binary digits are 0 and 1, but a binary number can be any.

There are just two numbers in Galois Field (2): 0 and 1. No other numbers.

What practical would you do with just two numbers? Not much, but you can pack GF(2) numbers into some kind of structure or tuple or even array. Such structures are represented using polynomials. For example, CRC32 polynomial you can find in source code is 0x04C11DB7. Each bit represent a number in GF(2), not a digit. The 0x04C11DB7 polynomial is written as:

$$x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$$

Wherever x^n is present, that means, you have a bit at position n . Just x means, bit present at LSB. There is, however, bit at x^{32} , so the CRC32 polynomial has the size of 33 bits, but the **MSB** is always 1 and is omitted in all algorithms.

It's important to say that unlike in algebra, GF(2) polynomials are never evaluated here. x is symbol is present merely as a convention. People represent GF(2) "structures" as polynomials to emphasize the fact that "numbers" are isolated from each other.

Now, subtraction and addition are the same operations in GF(2) and actually works as XOR. This is present in many tutorials, so I'll omit this here.

Also, by convention, whenever you compare two numbers in GF(2), you only compare two most significant bits, and ignore the rest.

5.6 CRC32

Now we can take the binary division algorithm and change it a little:

```
uint32_t remainder_GF2(uint32_t dividend, uint32_t divisor)
{
    // necessary bit shuffling/negation to make it compatible with other CRC32
    // implementations.
    // N.B.: input data is not an array, but a 32-bit integer, hence we need to swap
    // endiannes.
    uint32_t dividend_negated_swapped = ~swap_endianness32(bitrev32(dividend
    ));
    buf=(uint8_t*)&dividend_negated_swapped;
    buf_pos=3;
    buf_bit_pos=7;

    uint32_t tmp=0;

    // process 32 bits from the input + 32 zero bits:
    for(int i=0; i<32+32; i++)
    {
        int bit=get_bit();
        int shifted_bit=tmp>>31;

        // fetch next bit:
        tmp=tmp<<1;
        if (bit==-1)
        {
            // no more bits, but continue, we fetch 32 more zero bits.
            // shift left operation set leftmost bit to zero.
        }
        else
        {
            // append next bit at right:
            tmp=tmp|bit;
        }

        // at this point, tmp variable/value has 33 bits: shifted_bit + tmp
        // now take the most significant bit (33th) and test it:
        // 33th bit of polynomial (not present in "divisor" variable is always 1
```

```

        // so we have to only check shifted_bit value
        if (shifted_bit)
        {
            // use only 32 bits of polynomial, ignore 33th bit, which is
            // always 1:
            tmp=tmp^divisor;
        };
    };
    // bit shuffling/negation for compatibility once again:
    return ~bitrev32(tmp);
}

```

(https://github.com/DennisYurichev/Math-for-programmers/blob/master/GF2/div_both.c)

And voila, this is the function which computes CRC32 for the input 32-bit value.

There are only 3 significant changes:

- XOR instead of minus.
- Only **MSB** is checked during comparison. But the **MSB** of all CRC polynomials is always 1, so we only need to check **MSB** (33th bit) of the tmp variable.
- There are 32+32=64 iterations instead of 32. As you can see, only **MSB** of tmp affects the whole behaviour of the algorithm. So when tmp variable is filled by 32 bits which never affected anything so far, we need to "blow out" all these bits through 33th bit of tmp variable to get correct remainder (or CRC32 sum).

All the rest algorithms you can find on the Internet are optimized version, which may be harder to understand. No algorithms used in practice "blows" anything "out" due to optimization. Many practical algorithms are either bitwise (process input stream by bytes, not by bits) or table-based.

My goal was to write two functions, as similar to each other as possible, to demonstrate the difference.

So the CRC value is in fact remainder of division of input data by CRC polynomial in GF(2) environment. As simple as that.

5.7 Rationale

Why use such an unusual mathematics? The answer is: many GF(2) operations can be done using bit shifts and XOR, which are very cheap operations.

Electronic circuit for CRC generator is extremely simple, it consists of only shift register and XOR gates. This one is for CRC16:

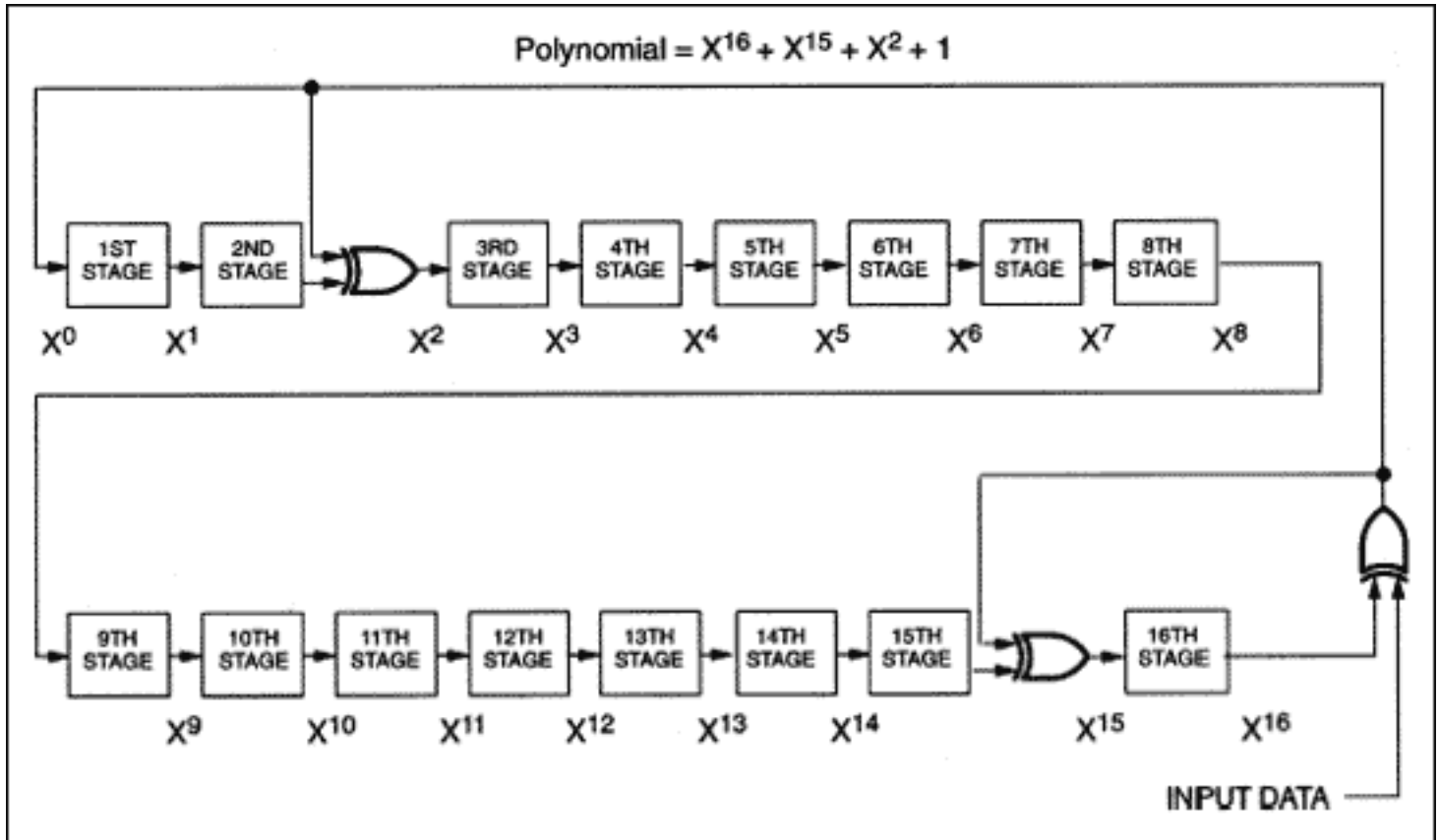


Figure 5.1:

(The source of image: <https://olimex.wordpress.com/2014/01/10/weekend-programming-challenge-week-39-crc->
)

Only 3 XOR gates are present aside of shift register.

The following page has animation: https://en.wikipedia.org/wiki/Computation_of_cyclic_redundancy_checks.

It can be implemented maybe even using vacuum tubes.

And the task is not to compute remainder according to rules of arithmetics, but rather to detect errors.

Compare this to a division circuit with at least one binary adder/subtractor, which will have carry-ripple problem. On the other hand, addition over GF(2) has no carries, hence, this problem absent.

5.8 Further reading

These documents I've found interesting/helpful:

- http://www.ross.net/crc/download/crc_v3.txt
- <https://www.kernel.org/doc/Documentation/crc32.txt>
- <http://web.archive.org/web/20161220015646/http://www.hackersdelight.org/crc.pdf>

Chapter 6

Logarithms

6.1 Introduction

6.1.1 Children's approach

When children argue about how big their favorite numbers are, they speaking about how many zeroes it has: “ x has n zeroes!” “No, my y is bigger, it has $m > n$ zeroes!”

This is exactly notion of common (base 10) logarithm.

Googol (10^{100}) has 100 zeroes, so $\log_{10}(\text{googol}) = 100$.

Let's take some big number, like 12th Mersenne prime:

Listing 6.1: Wolfram Mathematica

```
In [] := 2^127 - 1
Out [] = 170141183460469231731687303715884105727
```

Wow, it's so big. How can we measure it in childish terms? How many digits it has? We can count using common (base 10) logarithm:

Listing 6.2: Wolfram Mathematica

```
In [] := Log[10, 2^127 - 1] // N
Out [] = 38.2308
```

So it has 39 digits.

Another question, how many decimal digits 1024-bit RSA key has?

Listing 6.3: Wolfram Mathematica

```
In [] := 2^1024
Out [] = 17976931348623159077293051907890247336179769789423065727343008\
1157732675805500963132708477322407536021120113879871393357658789768814\
416622492847430639474124377678934248654852763022196012460941194530829\
```

```
5208500576883815068234246288147391311054082723716335051068458629823994\  
7245938479716304835356329624224137216
```

```
In [] := Log10 [2^1024] // N  
Out [] = 308.255
```

309 decimal digits.

6.1.2 Scientists' and engineers' approach

Interestingly enough, scientists' and engineers' approach is not very different from children's. They are not interesting in noting each digit of some big number, they are usually interesting in three properties of some number: 1) sign; 2) first n digits (significand or mantissa); 3) exponent (how many digits the number has).

The common way to represent a real number in handheld calculators and FPUs is:

$$(sign)significand \times 10^{exponent} \quad (6.1)$$

For example:

$$- 1.987126381 \times 10^{41} \quad (6.2)$$

It was common for scientific handheld calculators to use the first 10 digits of significand and ignore everything behind. Storing the whole number down to the last digit is 1) very expensive; 2) hardly useful.

The number in IEEE 754 format (most popular way of representing real numbers in computers) has these three parts, however, it has different base (2 instead of 10).

6.2 Logarithmic scale

6.2.1 In human perception

Logarithmic scale is very natural to human perceptions, including eyes. When you ask average human to judge on current lighting, he/she may use words like “dark”, “very dark”, “normal”, “twilight”, “bright”, “like on beach”. In human language, there are couple of steps between “dark” and “bright”, but luminous intensity may differ by several orders of magnitude. Old cheap “point-n-shoot” photo cameras also has scale expressed in natural human languages. But professional photo cameras also has logarithmic scales:

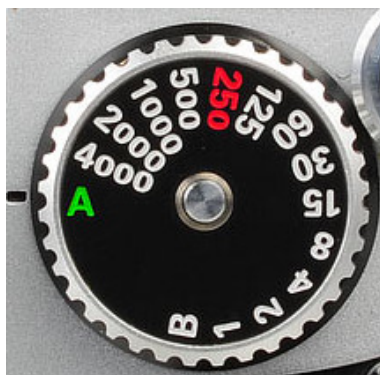


Figure 6.1: Shutter speed ($\frac{1}{x}$ of second) knob on photo camera

Another logarithmic scale familiar to anyone is decibel. Even on cheap mp3 players and smartphones, where the volume is measured in conventional percents, this scale is logarithmic, and the difference between 50% and 60% may be much larger in sound pressure terms.

Yet another familiar to anyone logarithmic scale is Richter magnitude scale ¹. The Richter scale is practical, because when people talk about earthquakes, they are not interesting in exact scientific values (in Joules or TNT equivalent), they are interesting in how bad damage is.

6.2.2 In electronics engineering

The loudspeakers are not perfect, so its output is non-linear in relation to input frequency. In other word, loudspeaker has different loudness at different frequency. It can be measured easily, and here is an example of plot of some speaker, I took it there: <http://www.3dnews.ru/270838/page-3.html>.

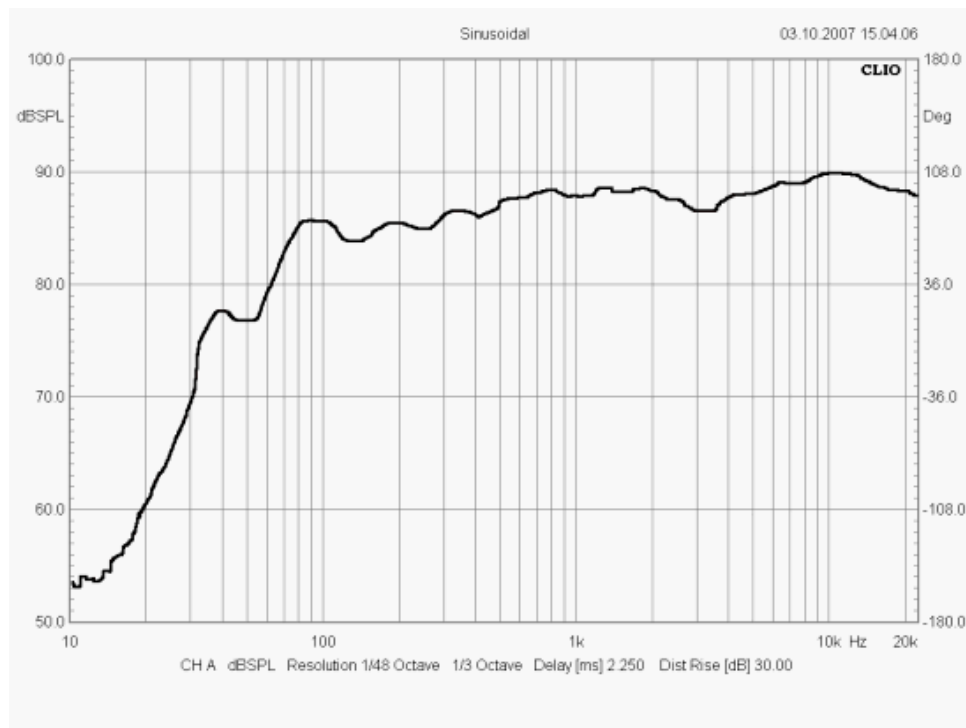


Figure 6.2: Frequency response (also known as *Bode plot*) of some loudspeaker

Both axis on this plot are logarithmic: y axis is loudness in decibel and x axis is frequency in Hertz. Needless to say, the typical loudspeaker has bass/medium speaker + tweeter (high frequency speaker). Some of more advanced loudspeaker has 3 speakers: bass, medium and tweeter. Or even more. And since the plot is logarithmic, each of these 2 or 3 speakers has their own part of plot, and these parts has comparable size. If the x axis would be linear instead of logarithmic, the main part of it would be occupied by frequency response of tweeter alone, because it has widest frequency range. While bass speaker has narrowest frequency range, it would have very thin part of the plot.

y axis (vertical) of the plot is also logarithmic (its value is shown in decibels). If this axis would be linear, the main part of it would be occupied by very loud levels of sound, while there would be thinnest line at the bottom reserved for normal and quiet level of sounds.

Both of that would make plot unusable and impractical. So both axis has logarithmic scale. In strict mathematics terms, the plot shown is called *log-log plot*, which means that both axis has logarithmic scale.

¹https://en.wikipedia.org/wiki/Richter_magnitude_scale

Summarizing, both electronics engineers and HiFi audio enthusiasts use these plots to compare quality of speakers. These plots are often used in loudspeakers reviews ².

6.2.3 In IT

git, like any other VCS, can show a graph, how many changes each file got in each commit, for example:

```
$ git log --stat
...
commit 2fb3437fa753d59ba37f3d11c7253583d4b87c99
Author: Dennis Yurichev <dennis@yurichev.com>
Date:   Wed Nov 19 14:14:07 2014 +0200

    reworking `64-bit in 32-bit environment' part

patterns/185_64bit_in_32_env/0.c          | 6 --
patterns/185_64bit_in_32_env/0_MIPS.s     | 5 -
patterns/185_64bit_in_32_env/0_MIPS_IDA.lst | 5 -
patterns/185_64bit_in_32_env/0_MSVC_2010_0x.asm | 5 -
patterns/185_64bit_in_32_env/1.c         | 20 ----
patterns/185_64bit_in_32_env/1_GCC.asm    | 27 -----
patterns/185_64bit_in_32_env/1_MSVC.asm   | 31 -----
patterns/185_64bit_in_32_env/2.c         | 16 ----
patterns/185_64bit_in_32_env/2_GCC.asm    | 41 -----
patterns/185_64bit_in_32_env/2_MSVC.asm   | 32 -----
patterns/185_64bit_in_32_env/3.c         | 6 --
patterns/185_64bit_in_32_env/3_GCC.asm    | 6 --
patterns/185_64bit_in_32_env/3_MSVC.asm   | 8 --
patterns/185_64bit_in_32_env/4.c         | 11 ---
patterns/185_64bit_in_32_env/4_GCC.asm    | 35 -----
patterns/185_64bit_in_32_env/4_MSVC.asm   | 30 -----
patterns/185_64bit_in_32_env/conversion/4.c | 6 ++
patterns/185_64bit_in_32_env/conversion/Keil_ARM_03.s | 4 +
patterns/185_64bit_in_32_env/conversion/MSVC2012_0x.asm | 6 ++
patterns/185_64bit_in_32_env/conversion/main.tex | 48 ++++++++
patterns/185_64bit_in_32_env/main.tex    | 127
+-----
...

```

This scale is not logarithmical (I had a look into git internals), but this is exact place where logarithmical scale can be used. When software developer got such report, he/she don't interesting in exact numbers of lines changed/added/removed. He/she wants to see an outlook: which files got most changes/additions/removals, and which got less.

There is also a constraint: the space on the terminal is limited, so it's not possible to draw a minus or plus sign for each changed line of code.

²Some of speakers of USSR era (like Latvian Radiotekhnika S-30 and S-90) had such plots right on the surface of speaker box, presumably, for marketing purposes.

Another example is Bitcoin client “signal reception strength”, apparently, modeled after mobile phone signal indicator:

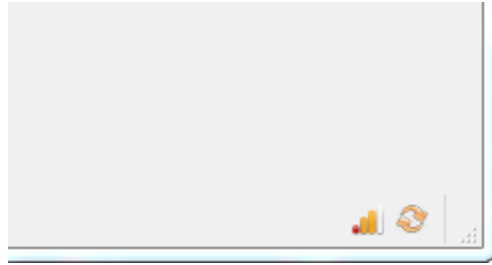


Figure 6.3: Bitcoin client

These bars indicating, how many connections client currently has. Let’s imagine, client can support up to 1000 connections, but user is never interesting in precise number, all he/she wants to know is how good its link with Bitcoin network is. I don’t know how Bitcoin calculates this, but I think one bar could indicate that client has only 1 connection, two bars — 2-5, three bars — up to 10-20, and four bars — anything bigger. This is also logarithmic scale. On contrary, if you divide 1000 by 4 even parts, and one bar will be fired if you’ve got 250 connections, two bars if you’ve got 500, etc, this would make the indicator useless, such indicators are no better than simple “on/off” lamp.

6.2.4 Web 2.0

Sites like GitHub, Reddit, Twitter sometimes shows how long some event was ago, instead of precise date (at least in 2015). For example, Reddit may show date as “3 years ago”, “11 months ago”, “3 weeks ago”, “1 day ago”, “10 hours ago”, etc, down to minutes and seconds. You wouldn’t see “3 years and 11 hours ago”. This is also logarithmic scale. When some event happens 10 months ago, users are typically not interested in precision down to days and hours. When something happens 2 years ago, users usually not interested in number of months and days in addition to these 2 years.

6.3 Multiplication and division using addition and subtraction

It is possible to use addition instead of multiplication, using the following rule:

$$\log_{base}(ab) = \log_{base}(a) + \log_{base}(b) \quad (6.3)$$

...while base can be any number.

It’s like summing number of zeroes of two numbers. Let’s say, you need to multiply 100 by 1000. Just sum number of their zeroes (2 and 3). The result is the number with 5 zeroes. It’s the same as $\log_{10}(100) + \log_{10}(1000) = \log_{10}(100000)$.

Division can be replaced with subtraction in the very same way.

6.3.1 Logarithmic slide rule

Here is very typical slide rule³. It has many scales, but take a look on C and D scales, they are the same:

³I took screenshots at <http://museum.syssrc.com/static/sliderule.html>

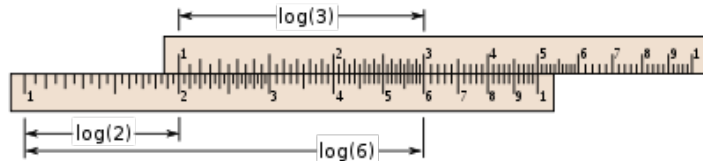


Figure 6.6: Example from Wikipedia

6.3.2 Logarithmic tables

As we can see, the precision of logarithmic slide rule is up to 1 or 2 decimal digits after point. Using precomputed logarithmic tables, it's possible to calculate product of two numbers with a precision up to ≈ 4 digits.

First, find common (base of 10) logarithms of each number using logarithmic table:

Таблица XIII. МАНТИССЫ ДЕСЯТИЧНЫХ ЛОГАРИФМОВ.

<i>N</i>	0	1	2	3	4	5	6	7	8	9	1	2	3	4	5	6	7	8	9
65	8129	8136	8142	8149	8156	8162	8169	8176	8182	8189	1	1	2	3	3	4	5	5	6
66	8195	8202	8209	8215	8222	8228	8235	8241	8248	8254	1	1	2	3	3	4	5	5	6
67	8261	8267	8274	8280	8287	8293	8299	8306	8312	8319	1	1	2	3	3	4	5	5	6
68	8325	8331	8338	8344	8351	8357	8363	8370	8376	8382	1	1	2	3	3	4	4	5	6
69	8388	8395	8401	8407	8414	8420	8426	8432	8439	8445	1	1	2	2	3	4	4	5	6

Figure 6.7: Logarithmic tables

Then add these numbers. Find the number you got in table of powers of 10 (10^x , also called “anti-log table”):

Таблица XIV. ЗНАЧЕНИЯ ФУНКЦИИ 10^x (ДЕСЯТИЧНЫЕ АНТИЛОГАРИФМЫ).

<i>m</i>	0	1	2	3	4	5	6	7	8	9	1	2	3	4	5	6	7	8	9
,00	1000	1002	1005	1007	1009	1012	1014	1016	1019	1021	0	0	1	1	1	1	2	2	2
,01	1023	1026	1028	1030	1033	1035	1038	1040	1042	1045	0	0	1	1	1	1	2	2	2
,02	1047	1050	1052	1054	1057	1059	1062	1064	1067	1069	0	0	1	1	1	1	2	2	2
,03	1072	1074	1076	1079	1081	1084	1086	1089	1091	1094	0	0	1	1	1	1	2	2	2
,04	1096	1099	1102	1104	1107	1109	1112	1114	1117	1119	0	1	1	1	1	2	2	2	2

Figure 6.8: Antilog tables

Resulting number is a product. The whole process may be faster than to multiply using long multiplication method using paper-n-pencil taught in schools.

Screenshots I took from the Bradis' book, once popular in USSR. Another well-known book in western world with logarithmic and other tables is Daniel Zwillinger - CRC Standard Mathematical Tables and Formulae (up to 30th edition, the logarithmic tables are dropped after).

6.3.3 Working with very small and very large numbers

It's hard to believe, but the rule used on logarithmic slide rule for multiplication is still used sometimes in software code. It's a problem to work with very small (denormalized) numbers⁴ encoded in IEEE 754 standard.

Here is my attempt to calculate $\frac{1.234 \times 10^{-300} \cdot 2.345678901234 \times 10^{-24}}{3.456789 \times 10^{-50}}$:

Listing 6.4: C code

```
#include <stdio.h>
#include <math.h>

int main()
{
    double a=1.234e-300;
    double b=2.345678901234e-24;
    double c=3.456789e-50;
    printf (".30e\n", a*b/c);
};
```

The output is $1.429261797122261460966983388190 \times 10^{-274}$, which is incorrect. When using debugger, we can see that the multiplication operation raises *inexact exception* and *underflow exception* in FPU. The division operation also raises *inexact exception*.

Let's check in Wolfram Mathematica:

Listing 6.5: Wolfram Mathematica

```
In [] := a = 1.234*10^(-300);
In [] := b = 2.345678901234*10^(-24);
In [] := c = 3.456789*10^(-50);
In [] := a*b/c
Out [] = 8.37357*10^-275
```

The underflow exception raised in my C program because result of multiplication is in fact $2.894567764122756 \times 10^{-324}$, which is even smaller than smallest denormalized number FPU can work with.

Let's rework our example to compute it all using natural logarithms ($\exp(x)$ is a C standard function, which computes e^x and $\log(x)$ here is $\log_e(x)$ (or $\ln(x)$):

⁴Denormalized numbers in double-precision floating point format are numbers between $\approx 10^{324}$ and $\approx 10^{308}$

Listing 6.6: C code

```
#include <stdio.h>
#include <math.h>

int main()
{
    double a=1.234e-300;
    double b=2.345678901234e-24;
    double c=3.456789e-50;
    printf ( "%.30e\n", exp(log(a)+log(b)-log(c)));
};
```

Now the output is $8.373573753338710216281125792150 \times 10^{-275}$, same as Mathematica reported.

The same problem with very large numbers.

Listing 6.7: C code

```
#include <stdio.h>
#include <math.h>

int main()
{
    double a=1.234e+300;
    double b=2.345678901234e+24;
    double c=3.456789e+50;
    printf ( "%.30e\n", a*b/c);
};
```

When this program running, its result is “inf”, meaning ∞ , i.e., overflow occurred. When using debugger, we can see than the multiplication operation raises *inexact exception* plus *overflow exception*. The correct value in Wolfram Mathematica is...

Listing 6.8: Wolfram Mathematica

```
In [] := a = 1.234*10^300;
In [] := b = 2.345678901234*10^24;
In [] := c = 3.456789*10^50;
In [] := a*b/c
Out [] = 8.37357*10^273
```

Let’s rewrite our C example:

Listing 6.9: C code

```

int main()
{
    double a=1.234e+300;
    double b=2.345678901234e+24;
    double c=3.456789e+50;
    printf ( "%.30e\n", exp(log(a)+log(b)-log(c)));
};

```

Now the program reports $8.373573753337712538419923350878 \times 10^{273}$, which is correct value.

The way of representing all numbers as their logarithms called “logarithmic number system”⁵. It allows to work with numbers orders of magnitude lower than FPU can handle.

So why all computations are not performed using logarithms, if it’s so good? It’s better only for very small or very large numbers. Working with small and medium numbers, precision of its logarithmic versions will be much more important and harder to control.

Also, finding logarithm of a number with the following exponentiation are operations may be slower than multiplication itself.

6.3.4 IEEE 754: adding and subtracting exponents

IEEE 754 floating point number consists of sign, significand and exponent. Internally, its simplified representation is:

$$(-1) \cdot sign \cdot significand \times 2^{exponent} \quad (6.4)$$

Given that, the FPU may process significands and exponents separately during multiplication, but when it processes exponents of two numbers, they are just summed up. For example:

$$significand_1 \times 2^{10} \cdot significand_2 \times 2^{50} = significand_3 \times 2^{60} \quad (6.5)$$

...precise values of significands are omitted, but we can be sure, if the first number has exponent of 10, the second has 50, the exponent of the resulting number will be ≈ 60 .

Conversely, during division, exponent of divisor is subtracted from the exponent of the dividend.

$$\frac{significand_1 \times 2^{10}}{significand_2 \times 2^{50}} = significand_3 \times 2^{-40} \quad (6.6)$$

I don’t have access to Intel or AMD FPU internals, but I can peek into OpenWatcom FPU emulator libraries⁶.

Here is summing of exponents during multiplication:

<https://github.com/open-watcom/open-watcom-v2/blob/86dbaf24bf7f6a5c270f5a6a50925f468d8d292b/bld/fpuemu/386/asm/fldm386.asm#L212>.

And here is subtracting of exponents during division:

<https://github.com/open-watcom/open-watcom-v2/blob/e649f6ed488eeebbc7ba9aeed8193d893288d398/bld/fpuemu/386/asm/fldd386.asm#L237>.

⁵https://en.wikipedia.org/wiki/Logarithmic_number_system

⁶It was a time in 1980s and 1990s, when FPU was expensive and it could be bought separately in form of additional chip and added to x86 computer. And if you had run a program which uses FPU on the computer where it’s missing, FPU emulating library might be an option. Much slower, but better than nothing.

Here is also multiplication function from FPU emulator in Linux kernel: https://github.com/torvalds/linux/blob/da957e111bb0c189a4a3bf8a00caaecb59ed94ca/arch/x86/math-emu/reg_u_mul.S#L93.

6.4 Exponentiation

Using equation 6.3 we may quickly notice that

$$b^n = \underbrace{b \times \dots \times b}_n = base^{(\log_{base}(b))^*n} \quad (6.7)$$

That works with any logarithmic base. In fact, this is the way how exponentiation is computed on computer. x86 CPU and x87 FPU has no special instruction for it.

This is the way how pow() function works in Glibc: https://github.com/lattera/glibc/blob/master/sysdeps/x86_64/fpu/e_pow1.S#L189:

Listing 6.10: Glibc source code, fragment of the pow() function

```
...
7:      fyl2x          // log2(x) : y
8:      fmul    %st(1) // y*log2(x) : y
      fst      %st(1) // y*log2(x) : y*log2(x)
      frndint          // int(y*log2(x)) : y*log2(x)
      fsubr    %st, %st(1) // int(y*log2(x)) : fract(y*log2(x))
      fxch          // fract(y*log2(x)) : int(y*log2(x))
      f2xm1          // 2^fract(y*log2(x))-1 : int(y*log2(x))
      faddl    M0(one) // 2^fract(y*log2(x)) : int(y*log2(x))
      fscale          // 2^fract(y*log2(x))*2^int(y*log2(x)) : int(y*
      log2(x))
      fstp     %st(1) // 2^fract(y*log2(x))*2^int(y*log2(x))
...
```

x87 FPU has the following instructions used Glibc's version of pow() function: FYL2X (compute $y \cdot \log_2 x$), F2XM1 (compute $2^x - 1$). Even more than that, FYL2X instruction doesn't compute binary logarithm alone, it also performs multiplication operation, to provide more easiness in exponentiation computation.

It works because calculating 2^x (exponentiation with base 2) is faster than exponentiation of arbitrary number.

Using hacker's tricks, it's also possible to take advantage of the IEEE 754 format and SSE instructions set:

<http://stackoverflow.com/a/6486630/4540328>.

6.5 Square root

Likewise, square root can be computed in the following way:

$$\sqrt[2]{x} = 2^{\frac{\log_2 x}{2}} \quad (6.8)$$

This leads to an interesting consequence: if you have a value stored in logarithmical form and you need to take square root of it and leave it in logarithmical form, all you need is just to divide it by 2.

And since floating point numbers encoded in IEEE 754 has exponent encoded in logarithmical form, you need just to shift it right by 1 bit to get square root: https://en.wikipedia.org/wiki/Methods_of_computing_square_roots#Approximations_that_depend_on_the_floating_point_representation.

Likewise, cube root and nth root can be calculated using logarithm of corresponding base:

$$\sqrt[b]{x} = b^{\frac{\log_b x}{b}} \quad (6.9)$$

6.6 Base conversion

FYL2X and F2XM1 instructions are the only logarithm-related x87 FPU has. Nevertheless, it's possible to compute logarithm with any other base, using these. The very important property of logarithms is:

$$\log_y(x) = \frac{\log_a(x)}{\log_a(y)} \quad (6.10)$$

So, to compute common (base 10) logarithm using available x87 FPU instructions, we may use this equation:

$$\log_{10}(x) = \frac{\log_2(x)}{\log_2(10)} \quad (6.11)$$

...while $\log_2(10)$ can be precomputed ahead of time.

Perhaps, this is the very reason, why x87 FPU has the following instructions: FLDL2T (load $\log_2(10) = 3.32193...$ constant) and FLDL2E (load $\log_2(e) = 1.4427...$ constant).

Even more than that. Another important property of logarithms is:

$$\log_y(x) = \frac{1}{\log_x(y)} \quad (6.12)$$

Knowing that, and the fact that x87 FPU has FYL2X instruction (compute $y \cdot \log_2 x$), logarithm base conversion can be done using multiplication:

$$\log_y(x) = \log_a(x) \cdot \log_y(a) \quad (6.13)$$

So, computing common (base 10) logarithm on x87 FPU is:

$$\log_{10}(x) = \log_2(x) \cdot \log_{10}(2) \quad (6.14)$$

Apparently, that is why x87 FPU has another pair of instructions:

FLDLG2 (load $\log_{10}(2) = 0.30103...$ constant) and FLDLN2 (load $\log_e(2) = 0.693147...$ constant).

Now the task of computing common logarithm can be solved using just two FPU instructions: FYL2X and FLDLG2.

This piece of code I found inside of Windows NT4 (src/OS/nt4/private/fp32/tran/i386/87tran.asm), this function is capable of computing both common and natural logarithms:

Listing 6.11: Assembly language code

```
lab fFLOGm
    fldlg2                ; main LOG10 entry point
    jmp     short fFYL2Xm

lab fFLNm                ; main LN entry point
    fldln2

lab fFYL2Xm
    fxch
    or     cl, cl         ; if arg is negative
    JSNZ  Y12XArgNegative ; return a NAN
    fyl2x                ; compute y*log2(x)
    ret
```

6.7 Binary logarithm

Sometimes denoted as $\text{lb}()$, binary logarithms are prominent in computer science, because numbers are usually stored and processed in computer in binary form.

6.7.1 Denoting a number of bits for some value

How many bits we need to allocate to store googol number (10^{100})?

Listing 6.12: Wolfram Mathematica

```
In [] := Log2[10^100] // N
Out [] = 332.193
```

Binary logarithm of some number is the number of how many bits needs to be allocated.

If you have a variable which always has 2^x form, it's a good idea to store a binary logarithmic representation ($\log_2(x)$) instead of it. There are at least two reasons: 1) the programmer shows to everyone that the number has always 2^x form; 2) it's error-prone, it's not possible to accidentally store a number in some other form to this variable, so this is some kind of protection; 3) logarithmic representation is more compact. There is, however, performance issue: the number must be converted back, but this is just one shifting operation ($1 \ll \log_n$).

Here is an example from NetBSD NTP client (netbsd-5.1.2/usr/src/dist/ntp/include/ntp.h):

Listing 6.13: C code

```
/*
 * Poll interval parameters
 */
...
```

```

#define NTP_MINPOLL      4      /* log2 min poll interval (16 s) */
#define NTP_MINDPOLL     6      /* log2 default min poll (64 s) */
#define NTP_MAXDPOLL    10     /* log2 default max poll ( 17 m) */
#define NTP_MAXPOLL     17     /* log2 max poll interval ( 36 h) */

```

Couple examples from zlib (deflate.h):

Listing 6.14: C code

```

uInt  w_size;      /* LZ77 window size (32K by default) */
uInt  w_bits;      /* log2(w_size) (8..16) */
uInt  w_mask;      /* w_size - 1 */

```

Another piece from zlib (contrib/blast/blast.c):

Listing 6.15: C code

```

int dict;          /* log2(dictionary size) - 6 */

```

If you need to generate bitmasks in range 1, 2, 4, 8...0x80000000, it is good idea to assign self-documenting name to iterator variable:

Listing 6.16: C code

```

for (log2_n=1; log2_n<32; log2_n++)
    1<<log2_n;

```

Now about compactness, here is the fragment I found in OpenBSD, related to SGI IP22 architecture ⁷
(OS/OpenBSD/sys/arch/sgi/sgi/ip22_machdep.c):

Listing 6.17: C code

```

/*
 * Secondary cache information is encoded as WLLSSSS, where
 * WW is the number of ways
 *   (should be 01)
 * LL is Log2(line size)
 *   (should be 04 or 05 for IP20/IP22/IP24, 07 for IP26)
 * SS is Log2(cache size in 4KB units)
 *   (should be between 0007 and 0009)
 */

```

Here is another example of using binary logarithm in Mozilla JavaScript engine (JIT compiler) ⁸. If some number is multiplied by 2^x , the whole operation can be replaced by bit shift left.

The following code (js/src/jit/mips/CodeGenerator-mips.cpp), when translating multiplication operation into

⁷<http://www.linux-mips.org/wiki/IP22>

⁸<http://fossies.org/linux/seamonkey/mozilla/js/src/jit/mips/CodeGenerator-mips.cpp>

MIPS machine code, first, get assured if the number is really has 2^x form, then it takes binary logarithm of it and generates MIPS SLL instruction, which states for “Shift Left Logical”.

Listing 6.18: Mozilla JavaScript JIT compiler (translating multiplication operation into MIPS bit shift instruction)

```
bool
CodeGeneratorMIPS::visitMulI(LMulI *ins)
{
    default:
        uint32_t shift = FloorLog2(constant);

        if (!mul->canOverflow() && (constant > 0)) {
            // If it cannot overflow, we can do lots of optimizations.
            uint32_t rest = constant - (1 << shift);

            // See if the constant has one bit set, meaning it can be
            // encoded as a bitshift.
            if ((1 << shift) == constant) {
                masm.ma_sll(dest, src, Imm32(shift));
                return true;
            }
        }

        ...
}
```

Thus, for example, $x = y \cdot 1024$ (which is the same as $x = y \cdot 2^{10}$) translates into $x = y \ll 10$.

6.7.2 Calculating binary logarithm

If all you need is integer result of binary logarithm ($abs(\log_2(x))$ or $\lfloor \log_2(x) \rfloor$), calculating is just counting all binary digits in the number minus 1. In practice, this is the task of calculating leading zeroes.

Here is example from Mozilla libraries (`mfbt/MathAlgorithms.h`⁹):

Listing 6.19: Mozilla libraries

```
class FloorLog2<T, 4>
{
public:
    static uint_fast8_t compute(const T aValue)
    {
        return 31u - CountLeadingZeroes32(aValue | 1);
    }
};

inline uint_fast8_t
CountLeadingZeroes32(uint32_t aValue)
{
    return __builtin_clz(aValue);
}
```

⁹<http://fossies.org/linux/seamonkey/mozilla/mfbt/MathAlgorithms.h>

```
}
```

Latest x86 CPUs has LZCNT (Leading Zeroes CouNT) instruction for that¹⁰, but there is also BSR (Bit Scan Reverse) instruction appeared in 80386, which can be used for the same purpose. More information about this instruction on various architectures: https://en.wikipedia.org/wiki/Find_first_set.

There are also quite esoteric methods to count leading zeroes without this specialized instruction: http://yurichev.com/blog/de_bruijn/.

6.7.3 $O(\log n)$ time complexity

Time complexity¹¹ is a measure of speed of a specific algorithm in relation to the size of input data.

$O(1)$ – time is always constant, to matter what size of input data. Simplest example is object getter – it just returns some value.

$O(n)$ – time is linear, growing according to the size of input data. Simplest example is search for some value in the input array. The larger array, the slowest search.

$O(\log n)$ – time is logarithmic to the input data. Let's see how this can be.

Let's recall child's number guessting game¹². One player think about some number, the other should guess it, offering various versions. First player answers, is guessed number is larger or less. A typical dialogue:

```
-- I think of a number in 1..100 range.
-- Is it 50?
-- My number is larger.
-- 75?
-- It is lesser.
-- 63?
-- Larger.
-- 69?
-- Larger.
-- 72?
-- Lesser.
-- 71?
-- Correct.
```

Best possible strategy is to divide the range in halves. The range is shorten at each step by half. At the very end, the range has length of 1, and this is correct answer. Maximal number of steps using the strategy described here are $\log_2(\text{initial_range})$. In our example, initial range is 100, so the maximum number of steps is 6.64... or just 7. If the initial range is 200, maximum number of steps are $\log_2(200) = 7.6..$ or just 8. The number of steps increasing by 1 when the range is doubled. Indeed, doubled range indicates that the guesser needs just one more step at the start, not more. If the initial range is 1000, numbers of steps are $\log_2(1000) = 9.96...$ or just 10.

This is exactly $O(\log n)$ time complexity.

Now let's consider couple of practical real-world algorithms. One interesting thing is that if the input array is sorted,

¹⁰GNU `__builtin_clz()` function on x86 architecture can be think for LZCNT

¹¹https://en.wikipedia.org/wiki/Time_complexity

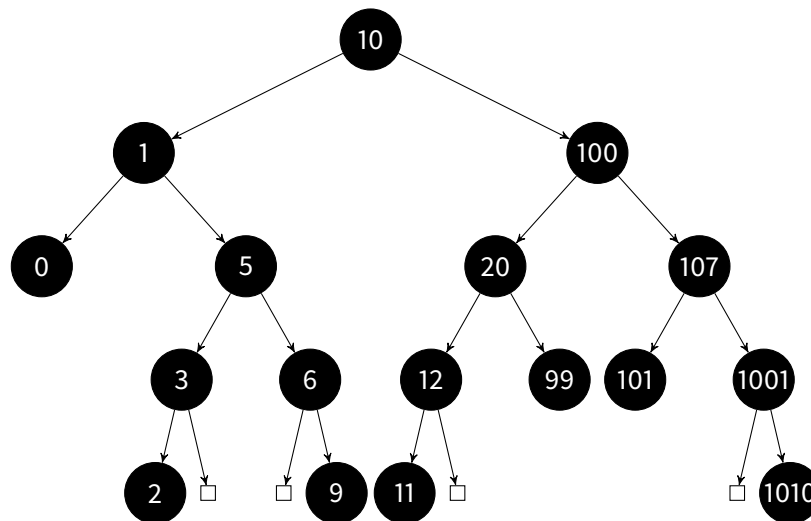
¹²http://rosettacode.org/wiki/Guess_the_number

and its size is known, and we need to find some value in it, the algorithm works exactly in the same way as child's number guessing game! The algorithm starts in the middle of array and compare the value there with the value sought-after. Depending on the result (larger or lesser), the *cursor* is moved left or right and operating range is decreasing by half. This is called binary search¹³, and there is the `bsearch()` function in standard C/C++ library¹⁴.

Here is how binary search is used in git: <https://www.kernel.org/pub/software/scm/git/docs/git-bisect.html>.

Another prominent example in CS is binary trees. They are heavily used internally in almost any programming language, when you use set, map, dictionary, etc.

Here is a simple example with the following numbers (or keys) inserted into binary tree: 0, 1, 2, 3, 5, 6, 9, 10, 11, 12, 20, 99, 100, 101, 107, 1001, 1010.



And here is how binary tree search works: put *cursor* at the root. Now compare the value under it with the value sought-after. If the value we are seeking for is lesser than the current, take a move into left node. If it's bigger, move to the right node. Hence, each left descendant node has value lesser than in ascendant node. Each right node has value which is bigger. The tree must be rebalanced after each modification (I gave examples of it in my book about reverse engineering (<http://beginners.re/>, 51.4.4)). Nevertheless, lookup function is very simple, and maximal number of steps is $\log_n(\text{number_of_nodes})$. We've got 17 elements in the tree at the picture, $\log_2(17) = 4.08\dots$, indeed, there are 5 tiers in the tree.

6.8 Common (base 10) logarithms

Also known as “decimal logarithms”. Denoted as `lg` on handheld calculators.

10 is a number inherently linked with human's culture, since almost all humans has 10 digits. Decimal system is a result of it. Nevertheless, 10 has no special meaning in mathematics and science in general. So are common logarithms.

One notable use is a decibel logarithmic scale, which is based of common logarithm.

Common logarithms are sometimes used to calculate space for decimal number in the string or on the screen. How many characters you should allocate for 64-bit number? 20, because $\log_{10}(2^{64}) = 19.2\dots$

¹³https://en.wikipedia.org/wiki/Binary_search_algorithm

¹⁴<http://en.cppreference.com/w/cpp/algorithm/bsearch>

Functions like `itoa()`¹⁵ (which converts input number to a string) can calculate output buffer size precisely, calculating common logarithm of the input number.

6.9 Natural logarithm

Natural logarithm (denoted as `ln` on handheld calculators, and sometimes denoted just as `log`) is logarithm of base $e = 2.718281828\dots$. Where this constant came from?

6.9.1 Savings account in your bank

Let's say you make a deposit into bank, say, 100 dollars (or any other currency). They offer 2.5% per year (annual percentage yield). This mean, you'll can get doubled amount of money (200 dollars) after 40 years. So far so good. But some banks offers compound interest. Also called "complex percent" in Russian language, where "complex" in this phrase is closer to the word "folded". This mean, after each year, they pretend you withdraw your money with interest, then redeposit them instantly. Banks also say that the interest is recapitalized once a year. Let's calculate final amount of money after 40 years:

Listing 6.20: Python code

```
#!/usr/bin/env python

initial=100 # 100 dollars, or any other currency
APY=0.025   # Annual percentage yield = 2.5%

current=initial

# 40 years
for year in range(40):
    # what you get at the end of each year?
    current=current+current*APY
    print "year=", year, "amount at the end", current
```

```
year= 0 amount at the end 102.5
year= 1 amount at the end 105.0625
year= 2 amount at the end 107.6890625
year= 3 amount at the end 110.381289063
...
year= 36 amount at the end 249.334869861
year= 37 amount at the end 255.568241608
year= 38 amount at the end 261.957447648
year= 39 amount at the end 268.506383839
```

The thing is that the final amount (268.50...) is aimed toward e constant.

Now there is another bank, which offers to recapitalize your deposit each month. We'll rewrite our script slightly:

Listing 6.21: Python code

```
#!/usr/bin/env python
```

¹⁵<http://www.cplusplus.com/reference/cstdlib/itoa/>


```

initial=100 # $100
APY=0.025 # Annual percentage yield = 2.5%

current=initial

# 40 years
for year in range(40):
    for month in range(12):
        # what you get at the end of each month?
        current=current+current*(APY/12)
        print "year=", year, "month=", month, "amount", current

```

```

year= 0 month= 0 amount 100.208333333
year= 0 month= 1 amount 100.417100694
year= 0 month= 2 amount 100.626302988
year= 0 month= 3 amount 100.835941119
...
year= 39 month= 8 amount 269.855455383
year= 39 month= 9 amount 270.417654248
year= 39 month= 10 amount 270.981024361
year= 39 month= 11 amount 271.545568162

```

The final result is even closer to e constant.

Let's imagine there is a bank which allows to recapitalize each day:

Listing 6.22: Python code

```

#!/usr/bin/env python

initial=100 # $100
APY=0.025 # Annual percentage yield = 2.5%

current=initial

# 40 years
for year in range(40):
    for month in range(12):
        for day in range(30):
            # what you get at the end of each day?
            current=current+current*(APY/12/30)
            print "year=", year, "month=", month, "day=", day, "amount", current

```

```

year= 0 month= 0 day= 0 amount 100.006944444
year= 0 month= 0 day= 1 amount 100.013889371
year= 0 month= 0 day= 2 amount 100.02083478
year= 0 month= 0 day= 3 amount 100.027780671

```

```

...
year= 39 month= 11 day= 26 amount 271.762123927
year= 39 month= 11 day= 27 amount 271.780996297
year= 39 month= 11 day= 28 amount 271.799869977
year= 39 month= 11 day= 29 amount 271.818744968

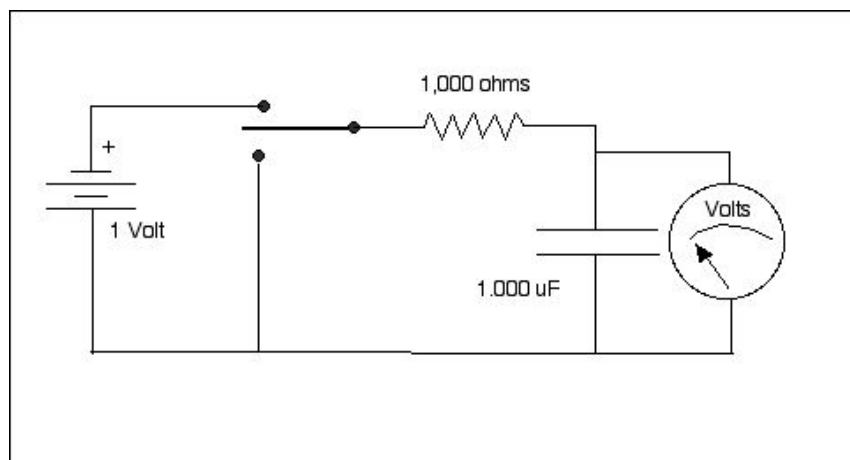
```

The final amount of money is more closer to e constant.

If to imagine some really crazy bank client who redeposit his deposit infinite number of times per each day, the final value after 40 years would be $100 \cdot e$. It's not possible in the real world, so the final amount is approaches this value, but is never equal to it. Mathematically speaking, its limit is $100 \cdot e$.

6.9.2 Exponential decay

Capacitor discharge



From electronics engineering course we may know that the capacitor discharging by half after $RC \ln(2)$ seconds, where C is capacity of capacitor in farads and R resistance of resistor in ohms. Given $1k\Omega$ resistor and $1000\mu F$ capacitor, what its voltage after 1 seconds will be? after 2 seconds? It's discharge can be calculated using this equation:

$$V = V_0 \cdot e^{\frac{-t}{RC}}$$

...where V_0 is initial charge in volts, t is time in seconds and e is base of natural logarithm.

Let's see it in Wolfram Mathematica:

Listing 6.23: Wolfram Mathematica

```

r = 1000; (* resistance in ohms *)
c = 0.001; (* capacity in farads *)
v = 1; (* initial voltage *)
Plot[v*E^((-t)/(r*c)), {t, 0, 5},

```

```

GridLines -> {{Log[2], Log[2]*2, Log[2]*3}, {0.5, 0.25, 0.125}},
Epilog -> {Text["ln(2)", {Log[2], 0.05}],
  Text["ln(2)*2", {Log[2]*2, 0.05}],
  Text["ln(2)*3", {Log[2]*3, 0.05}],
  Text["1/2", {0.1, 0.5}], Text["1/4", {0.1, 0.25}],
  Text["1/8", {0.1, 0.125}]}, AxesLabel -> {seconds, voltage}

```

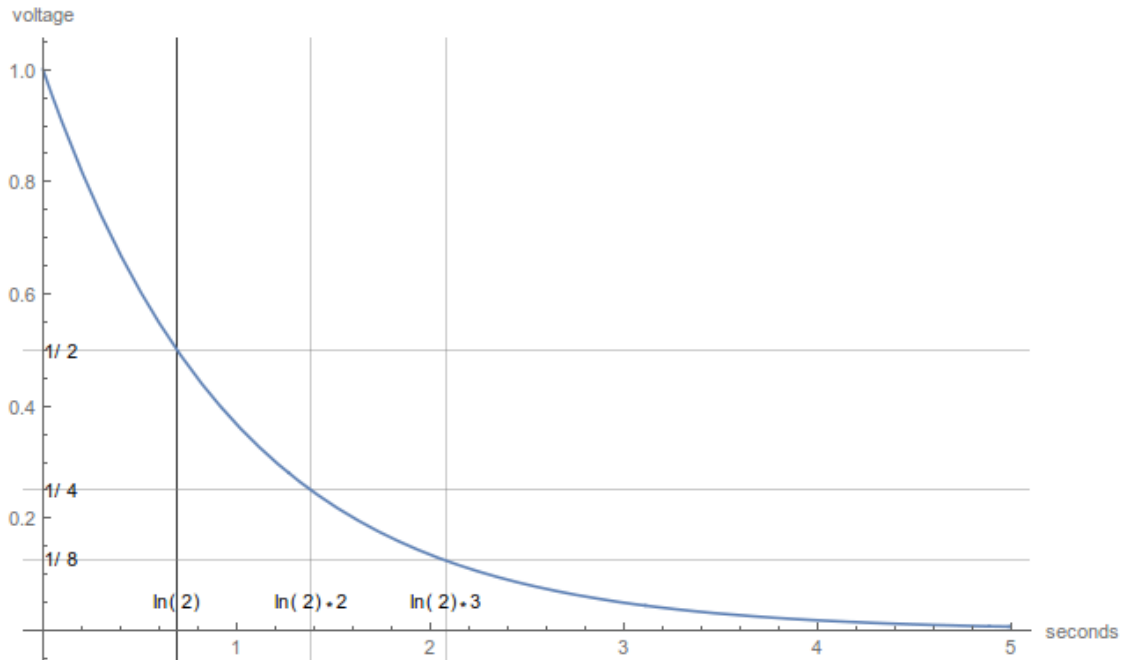


Figure 6.9: Capacitor voltage during discharge

As we can see, $\frac{1}{2}$ of initial charge is left after $\ln(2)$ seconds (0.69...), and $\frac{1}{4}$ of charge is left after $\ln(4)$ seconds (1.38...).

Indeed, if we are interested in precise time in seconds, when charge will be $\frac{1}{x}$, just calculate $\ln(x)$.

Now here is the same plot, but I added two more labels, $\frac{1}{3}$ and $\frac{1}{7}$:

Listing 6.24: Wolfram Mathematica

```

Plot[v*E^((-t)/(r*c)), {t, 0, 5},
GridLines -> {{Log[3], Log[7]}, {1/3, 1/7}},
Epilog -> {Text["ln(3)", {Log[3], 0.05}],
  Text["ln(7)", {Log[7], 0.05}],
  Text["1/3", {0.1, 1/3}], Text["1/7", {0.1, 1/7}]},
AxesLabel -> {seconds, voltage}

```

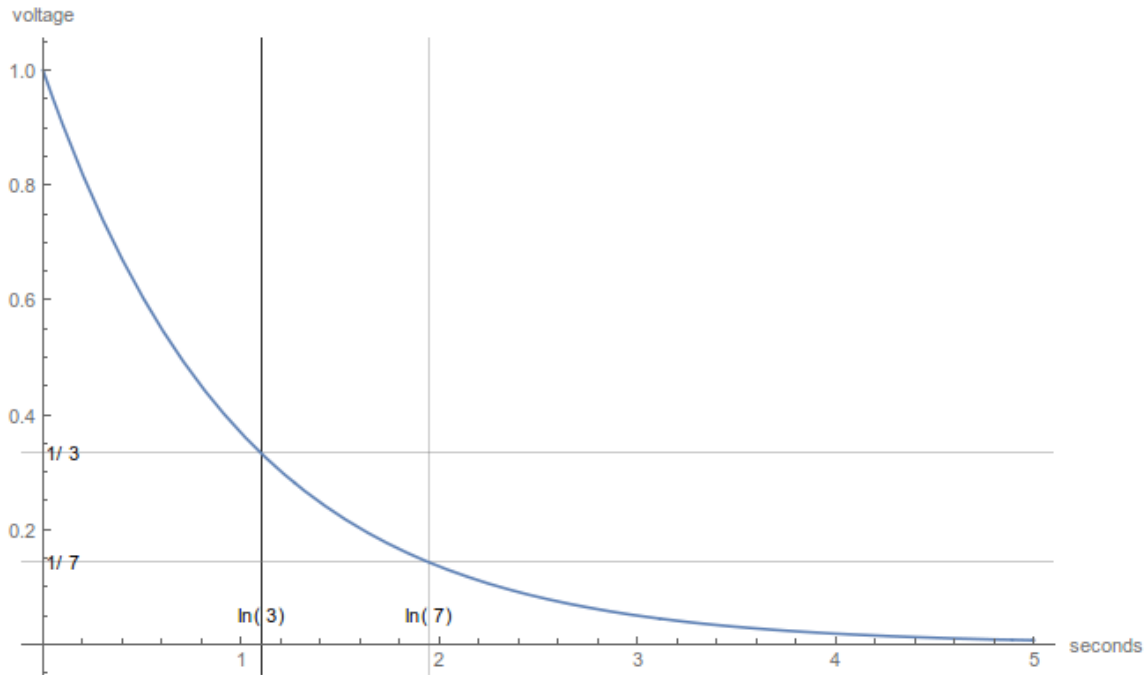


Figure 6.10: Capacitor voltage during discharge

...and we see that these points corresponds to $\ln(3)$ and $\ln(7)$. That means, $\frac{1}{3}$ of charge is left after $\ln(3) \approx 1.098...$ seconds and $\frac{1}{7}$ of charge after $\ln(7) \approx 1.945...$ seconds.

Radioactive decay

Radioactive decay is also exponential decay. Let's take Polonium 210 as an example¹⁶. It's half-life (calculated) is ≈ 138.376 days. That means that if you've got 1kg of Polonium 210, after ≈ 138 days, half of it (0.5 kg) left as ^{210}Po and another half is transformed into ^{206}Pb (isotope of lead¹⁷). After another ≈ 138 days, you'll get $\frac{3}{4}$ of isotope of lead and $\frac{1}{4}$ will left as ^{210}Po . After another ≈ 138 days, amount of Polonium will be halved yet another time, etc.

The equation of radioactive decay is:

$$N = N_0 e^{-\lambda t}$$

...where N is number of atoms at some point of time, N_0 is initial number of atoms, t is time, λ is decay constant. Decay of Polonium is exponential, but decay constant is the constant, defining how fast (or slow) it will fall.

Here we go in Mathematica, let's get a plot for 1000 days:

Listing 6.25: Wolfram Mathematica

```
l = 0.005009157516910051; (* decay constant of Polonium 210 *)

h1 = Log[2]/l
138.376
```

¹⁶<https://en.wikipedia.org/wiki/Polonium>

¹⁷https://en.wikipedia.org/wiki/Isotopes_of_lead#Lead-206

```

Plot[E^(-1*t), {t, 0, 1000},
  GridLines -> {{h1, h1*2, h1*3}, {0.5, 0.25, 0.125}},
  Epilog -> {Text["h1", {h1, 0.05}], Text["h1*2", {h1*2, 0.05}],
    Text["h1*3", {h1*3, 0.05}], Text["1/2", {30, 0.5}],
    Text["1/4", {30, 0.25}], Text["1/8", {30, 0.128}]},
  AxesLabel -> {days, atoms}

```

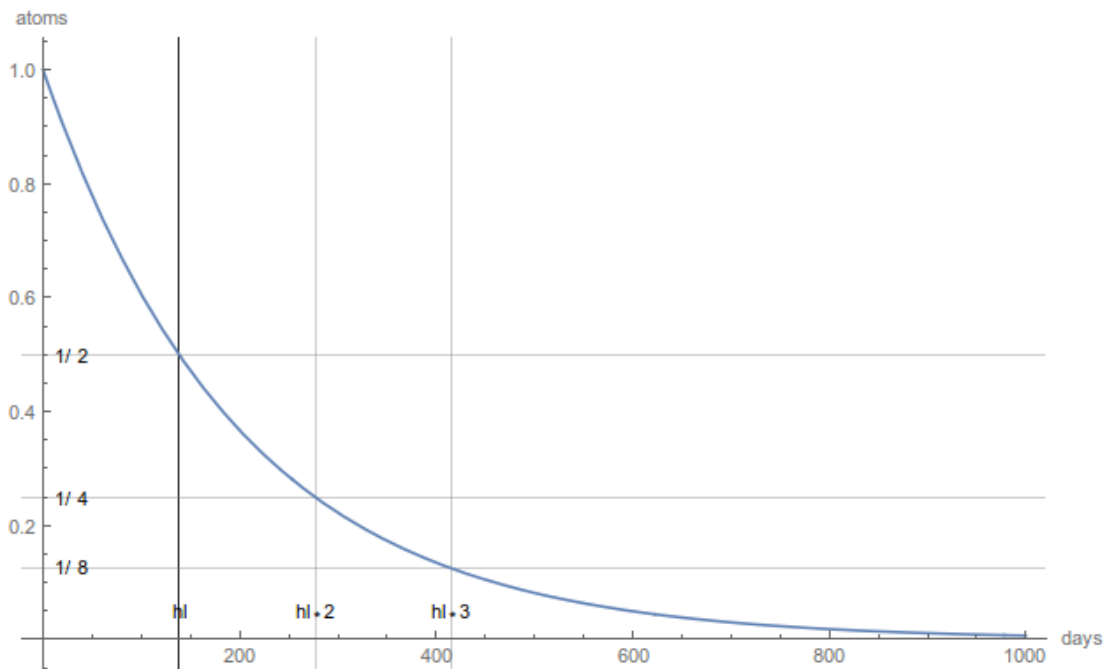


Figure 6.11: Exponential decay of Polonium 210

Beer froth

There is even the paper (got Ig Nobel prize in 2002), author's of which demonstrates that beer froth is also decays exponentially: <http://iopscience.iop.org/0143-0807/23/1/304/>, <https://classes.soe.ucsc.edu/math011a/Winter07/lecturenotes/beerdecay.pdf>.

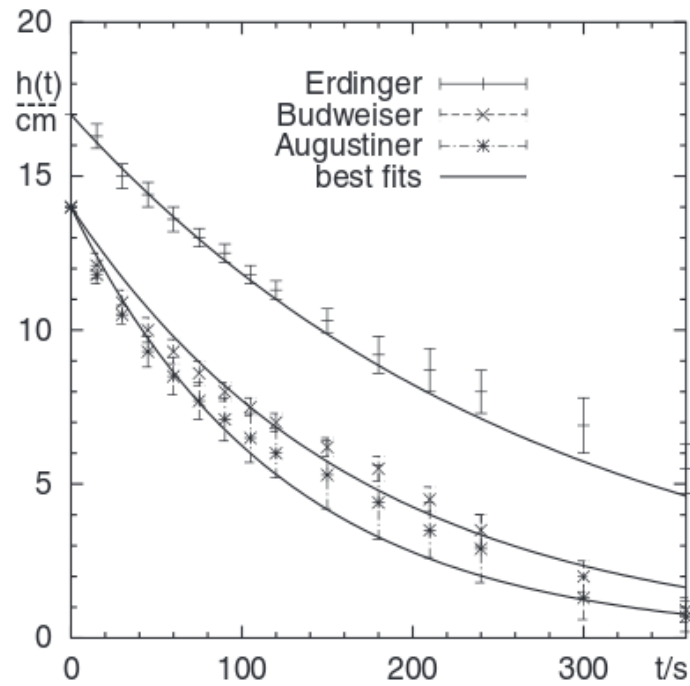


Figure 6.12: Results from the paper

The paper can be taken as a joke, nevertheless, it's a good demonstration of exponential decay.

Conclusion

Capacitor discharge and radioactive decay obeys the same law of halving some amount after equal gaps of time:

$$amount = amount_0 \cdot e^{-decay_constant \cdot time}$$

Decay constant in case of capacitor discharge defined by product of resistance and capacity. The bigger one of them, the slower decay.

Natural logarithm is used to calculate gap of time (half-life or half-time) judging by decay constant.

Chapter 7

Symbolic computation

Some numbers can only be represented in binary system approximately, like $\frac{1}{3}$ and π . If we calculate $\frac{1}{3} \cdot 3$ step-by-step, we may have loss of significance. We also know that $\sin(\frac{\pi}{2}) = 1$, but calculating this expression in usual way, we can also have some noise in result. Arbitrary-precision arithmetic¹ is not a solution, because these numbers cannot be stored in memory as a binary number of finite length.

How we could tackle this problem? Humans reduce such expressions using paper and pencil without any calculations. We can mimic human behaviour programmatically if we will store expression as tree and symbols like π will be converted into number at the very last step(s).

This is what Wolfram Mathematica² does. Let's start it and try this:

```
In [] := x + 2*8
Out [] = 16 + x
```

Since Mathematica has no clue what x is, it's left *as is*, but $2 \cdot 8$ can be reduced easily, both by Mathematica and by humans, so that is what has done. In some point of time in future, Mathematica's user may assign some number to x and then, Mathematica will reduce the expression even further.

Mathematica does this because it parses the expression and finds some known patterns. This is also called *term rewriting*³. In plain English language it may sounds like this: "if there is a $+$ operator between two known numbers, replace this subexpression by a computed number which is sum of these two numbers, if possible". Just like humans do.

Mathematica also has rules like "replace $\sin(\pi)$ by 0" and "replace $\sin(\frac{\pi}{2})$ by 1", but as you can see, π must be preserved as some kind of symbol instead of a number.

So Mathematica left x as unknown value. This is, in fact, common mistake by Mathematica's users: a small typo in an input expression may lead to a huge irreducible expression with the typo left.

Another example: Mathematica left this deliberately while computing binary logarithm:

```
In [] := Log[2, 36]
Out [] = Log[36]/Log[2]
```

¹https://en.wikipedia.org/wiki/Arbitrary-precision_arithmetic

²Another well-known symbolic computation system are [Maxima](#) and [SymPy](#)

³<https://en.wikipedia.org/wiki/Rewriting>

Because it has a hope that at some point in future, this expression will become a subexpression in another expression and it will be reduced nicely at the very end. But if we really need a numerical answer, we can force Mathematica to calculate it:

```
In [] := Log[2, 36] // N
Out [] = 5.16993
```

Sometimes unresolved values are desirable:

```
In [] := Union[{a, b, a, c}, {d, a, e, b}, {c, a}]
Out [] = {a, b, c, d, e}
```

Characters in the expression are just unresolved symbols⁴ with no connections to numbers or other expressions, so Mathematica left them *as is*.

Another real world example is symbolic integration⁵, i.e., finding formula for integral by rewriting initial expression using some predefined rules. Mathematica also does it:

```
In [] := Integrate[1/(x^5), x]
Out [] = -(1/(4 x^4))
```

Benefits of symbolic computation are obvious: it is not prone to loss of significance⁶ and round-off errors⁷, but drawbacks are also obvious: you need to store expression in (possible huge) tree and process it many times. Term rewriting is also slow. All these things are extremely clumsy in comparison to a fast FPU⁸.

“Symbolic computation” is opposed to “numerical computation”, the last one is just processing numbers step-by-step, using calculator, CPU⁹ or FPU.

Some task can be solved better by the first method, some others – by the second one.

7.1 Rational data type

Some LISP implementations can store a number as a ratio/fraction¹⁰, i.e., placing two numbers in a cell (which, in this case, is called *atom* in LISP lingo). For example, you divide 1 by 3, and the interpreter, by understanding that $\frac{1}{3}$ is an irreducible fraction¹¹, creates a cell with 1 and 3 numbers. Some time after, you may multiply this cell by 6, and the multiplication function inside LISP interpreter may return much better result (2 without *noise*).

Printing function in interpreter can also print something like $1 / 3$ instead of floating point number.

This is sometimes called “fractional arithmetic” [see TAOCP¹², 3rd ed., (1997), 4.5.1, page 330].

⁴Symbol like in LISP

⁵https://en.wikipedia.org/wiki/Symbolic_integration

⁶https://en.wikipedia.org/wiki/Loss_of_significance

⁷https://en.wikipedia.org/wiki/Round-off_error

⁸Floating-point unit

⁹Central processing unit

¹⁰https://en.wikipedia.org/wiki/Rational_data_type

¹¹https://en.wikipedia.org/wiki/Irreducible_fraction

¹²The Art Of Computer Programming (Donald Knuth's book)

This is not symbolic computation in any way, but this is slightly better than storing ratios/fractions as just floating point numbers.

Drawbacks are clearly visible: you need more memory to store ratio instead of a number; and all arithmetic functions are more complex and slower, because they must handle both numbers and ratios.

Perhaps, because of drawbacks, some programming languages offers separate (*rational*) data type, as language feature, or supported by a library¹³: Haskell, OCaml, Perl, Ruby, Python (*fractions*), Smalltalk, Java, Clojure, C/C++¹⁴.

¹³More detailed list: https://en.wikipedia.org/wiki/Rational_data_type

¹⁴By GNU Multiple Precision Arithmetic Library

Chapter 8

Graph theory

Graph is a group of nodes, some of them may be connected with each other, some are not. One of the popular examples is the map of country: there are cities and roads. "Cities" are called "nodes" or "vertices" in mathematics lingo, while "roads" are called "edges". Another popular example of graph is computer network, including Internet. Computer network is graph indeed, but it's closer to "sparse graph", because for the most part, computer networks are trees.

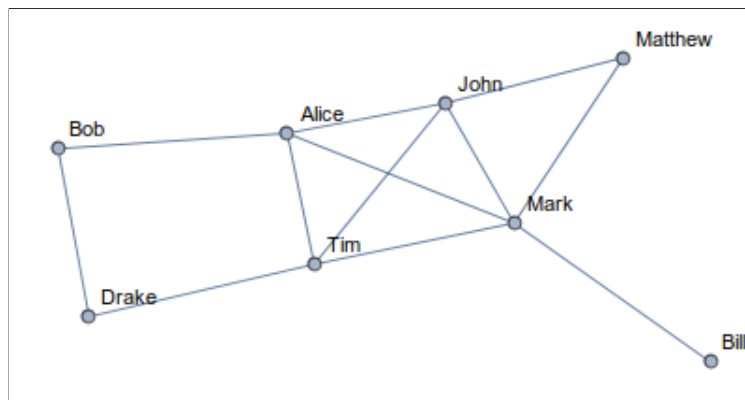
8.1 Clique in graph theory

"Clique" in everyday speech (especially in political news) denotes a tight-knit group of people inside of some community. In graph theory, "clique" is a subgraph (part of graph) each vertices ("nodes" or "members") of which are connected with each other.

8.1.1 Social graph: simple example

"Social graph" is a graph representing social links. Here is example I made in Wolfram Mathematica:

```
community =  
Graph[{John <-> Mark, John <-> Alice, Mark <-> Alice, Tim <-> Alice,  
  Matthew <-> John, Matthew <-> Mark, Tim <-> John, Drake <-> Tim,  
  Bob <-> Drake, Bill <-> Mark, Bob <-> Alice, Tim <-> Mark},  
VertexLabels -> "Name"]
```

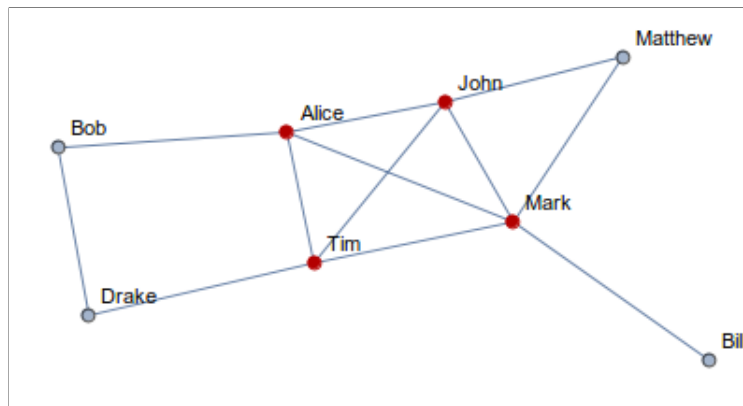


Let's try to find largest clique:

```
In []:= clique = FindClique[community]
Out []= {{John, Mark, Alice, Tim}}
```

Indeed, each of these four persons is connected to each among other 3. Wolfram Mathematica can highlight subgraph in graph:

```
HighlightGraph[community, clique]
```



8.1.2 Social graph: IRC network

Internet Relay Chat (IRC) is popular among open-source developers. One of the most popular IRC networks is Freenode. And one of the most crowded IRC channel there is #ubuntu, devoted to Ubuntu Linux. I used data from it, because all logs are available (starting at 2004), for example: <http://irclogs.ubuntu.com/2015/01/01/%23ubuntu.txt>.

When someone asks, and someone another going to answer the question, IRC users are address each other in this way:

```
[00:11] <synire> How would one find the path of an application installed using
terminal?
[00:11] <zykotick9> synire: "whereis foo"
[00:11] <synire> zykotick9: thanks!
```

It's not a rule, but well-established practice, so we can recover the information, which users talks to which users most often. Let's say, we would build a link between two IRC users if 1) they talk to each other at least 10-11 days (not necessary consequent); 2) do this at least 6 months (not necessary consequent).

The largest cliques of #ubuntu IRC channel in 10-11 years period are these:

```
* clique size 11
['ubottu', 'ActionParsnip', 'ikonian', 'Ben64', 'zykotick9', 'theadmin', '
dr_willis', 'MonkeyDust', 'usr13', 'bekks', 'iceroot']
* clique size 10
```

```

['ubottu', 'ActionParsnip', 'ikonia', 'jrib', 'bazhang', 'Pici', 'iceroot', 'theadmin', 'IdleOne', 'erUSUL']
* clique size 10
['ubottu', 'ActionParsnip', 'ikonia', 'jrib', 'bazhang', 'Pici', 'iceroot', 'theadmin', 'zykotick9', 'usr13']
* clique size 10
['ubottu', 'ActionParsnip', 'ikonia', 'jrib', 'bazhang', 'Pici', 'iceroot', 'sebsebseb', 'IdleOne', 'erUSUL']
* clique size 10
['ubottu', 'ActionParsnip', 'ikonia', 'jrib', 'Dr_Willis', 'Pici', 'edbian', 'IdleOne', 'Jordan_U', 'theadmin']
* clique size 10
['ubottu', 'ActionParsnip', 'ikonia', 'jrib', 'Dr_Willis', 'Pici', 'edbian', 'IdleOne', 'Jordan_U', 'sebsebseb']
* clique size 10
['ubottu', 'ActionParsnip', 'ikonia', 'jrib', 'Dr_Willis', 'Pici', 'erUSUL', 'iceroot', 'IdleOne', 'theadmin']
* clique size 10
['ubottu', 'ActionParsnip', 'ikonia', 'jrib', 'Dr_Willis', 'Pici', 'erUSUL', 'iceroot', 'IdleOne', 'sebsebseb']
* clique size 10
['ubottu', 'ActionParsnip', 'ikonia', 'jrib', 'Dr_Willis', 'Pici', 'erUSUL', 'iceroot', 'ubuntu', 'sebsebseb']
* clique size 10
['ubottu', 'ActionParsnip', 'ikonia', 'Ben64', 'histo', 'bekks', 'MonkeyDust', 'dr_willis', 'iceroot', 'usr13']
...

```

Perhaps, these users are frequenters of the channel. List of all cliques are here: <https://github.com/DennisYurichev/Math-for-programmers/blob/master/graph/cliقة/files/IRC/results.txt>. The output is not terse, because all listed cliques are cliques indeed, and single user or users group can be member of several cliques, that's correct. Cliques can be overlapped and be members of bigger cliques. It's possible to produce more human-like results using [more complex algorithms for finding communities](#).

The source code of my scripts here: <https://github.com/DennisYurichev/Math-for-programmers/tree/master/graph/cliقة/files/IRC>. I used the excellent [networkx](#) graph library.

8.1.3 Attempt to find communities in IRC social graph

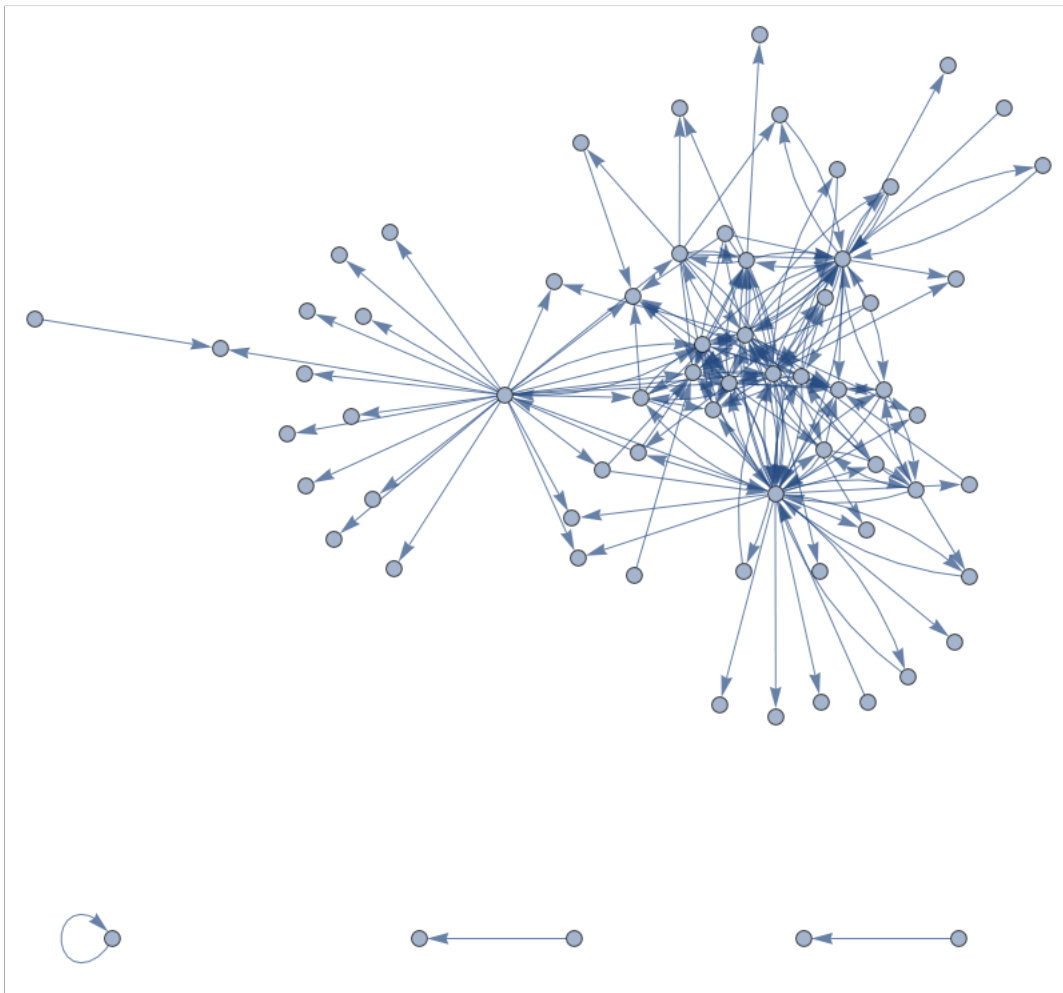
Wolfram Mathematica can try to find communities within social graph. Here I will import information about all IRC interactions from the start of 2013 till the summer of 2015. User nicknames are coded by numbers for simplicity.

```

In[]:= g2 =
Graph[{91708 -> 93574, 93414 -> 91525, 93414 -> 89579,
  90407 -> 93896, 93414 -> 93598, 93809 -> 5909, 93698 -> 93801,
  93163 -> 83317, 84930 -> 93896, 93414 -> 92947, 93414 -> 91708,
  93792 -> 92887, 84930 -> 91708, 91708 -> 84930, 88400 -> 93698,
  ...
  93809 -> 93475, 93698 -> 92887, 93801 -> 93670, 92887 -> 93598}]

```

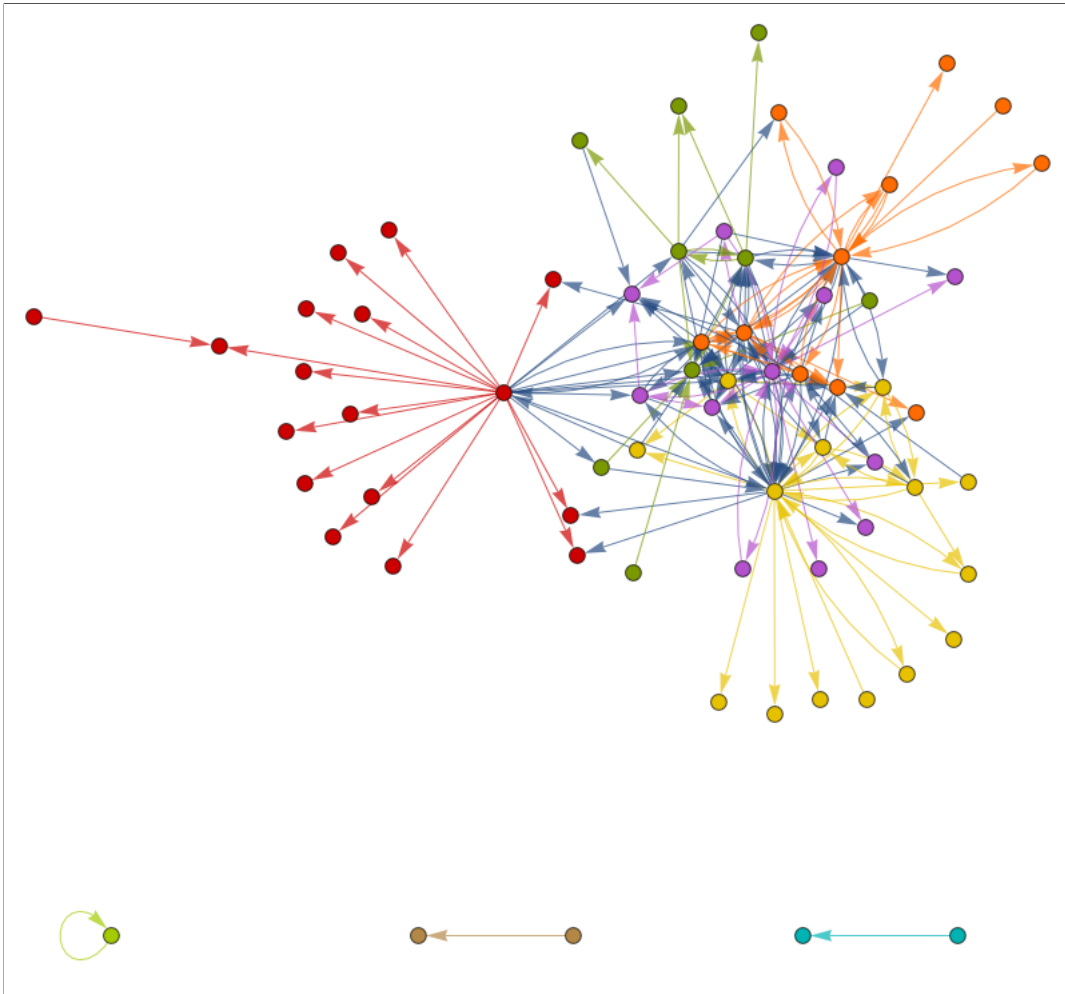
The resulting graph is:



There some artifacts (at the bottom) which can be ignored so far, I think. There is prominent centers: one huge and two others are smaller. I'm not sure, but I can suggest these parts of graph are just users who has different sleep patterns, or, more likely, from different time zones, so each important time zone (like Americas, Europe, Asia/Oceania) may have their own social communities. But again, I'm not sure, this should be investigated first.

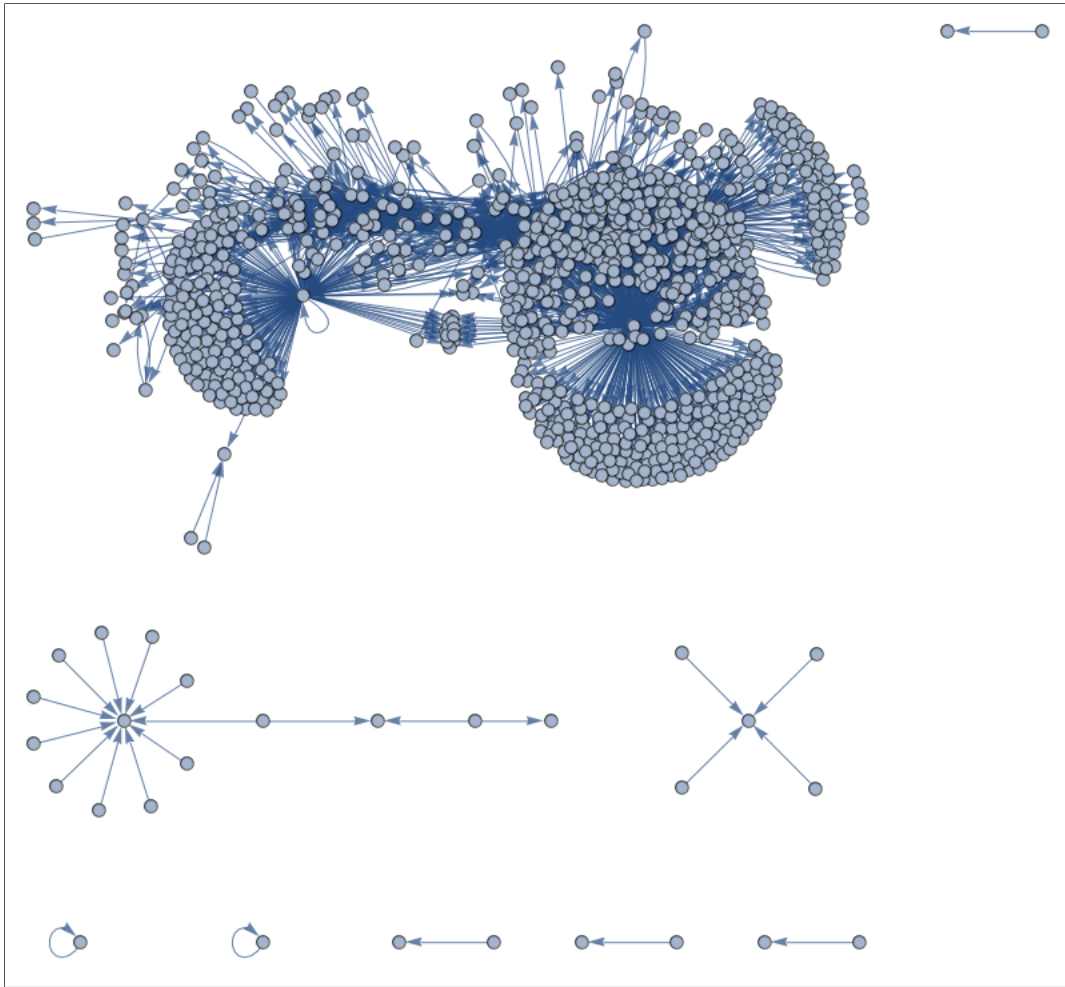
Let's try to find communities and highlight them within the graph:

```
c2 = FindGraphCommunities[g2];  
HighlightGraph[g2, Map[Subgraph[g2, #] &, c2]]
```



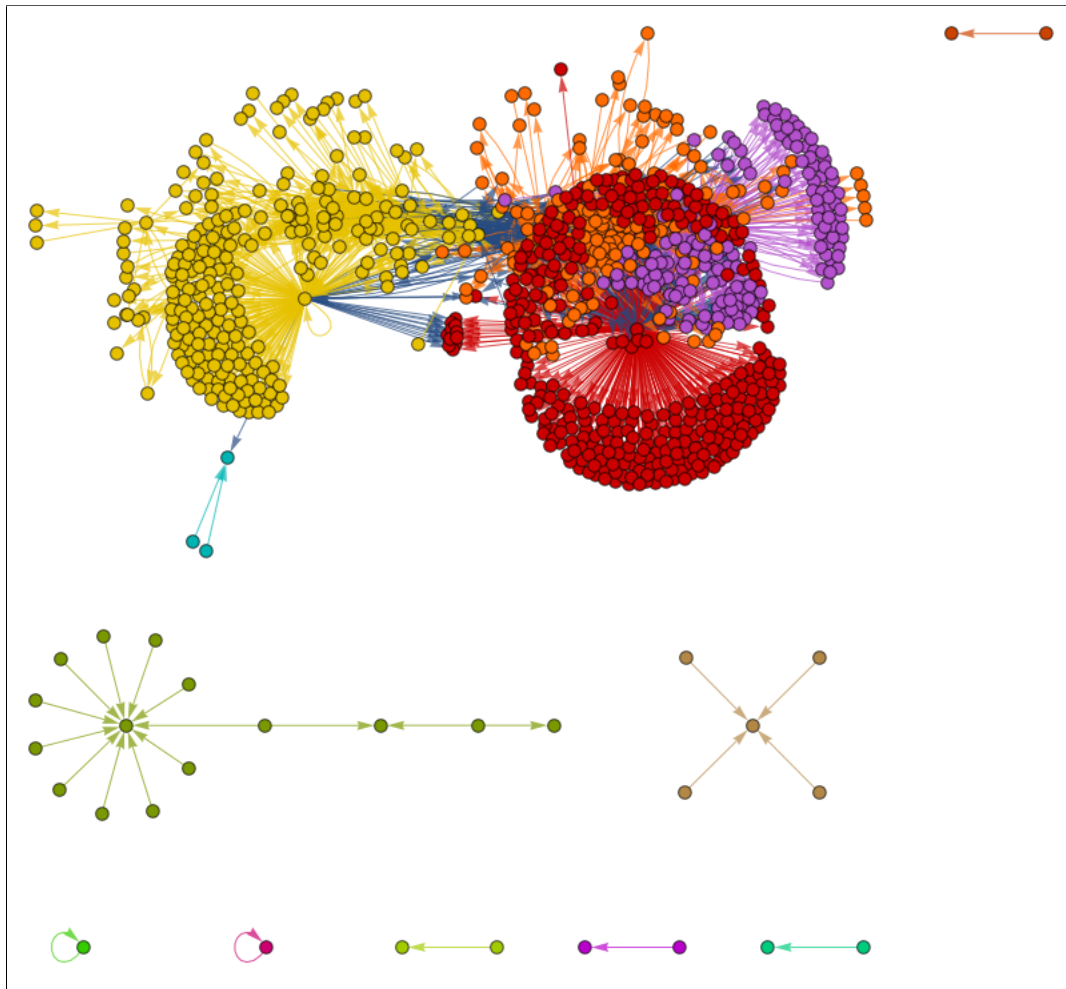
Hard to say if Mathematica right, but this is what it did.

Now let's take the whole graph of all IRC interactions starting at year 2004 till the summer of 2015. The graph is much bigger:



There are more artifacts.

Let's apply Mathematica's method to find communities:



Is it right? Maybe. Needless to say, since timespan is so long (at least 10 years), we can believe that some communities which may exist in 2004-2006 may be extinct in 2014-2015 (people got older, lost their interest in Ubuntu Linux, etc), but they all are visible on this graph.

Summary: perhaps, on our next experiment we should filter out IRC data by years and time zones.

8.1.4 Social graph: social networks

Perhaps, social networking websites like Facebook and Twitter in the "people you may know" tab shows you users of most populous (by your current friends) cliques. It may be much more complex in reality, but nevertheless, this is simplest possible way to offer you new social contacts.

8.1.5 Links graph: Wikipedia

Wikipedia has a lot of internal links, 463,000,000 in English Wikipedia as of summer 2015, if not to count user/talk/-media pages, etc. It's possible to build a graph where Wikipedia article is a vertex (or node) and a link from one article to another is edge. By link between articles we would call the case when the first article has the link to the second article, but also the second has the link to the first one.

Here are some examples of cliques I found this way. Number in parenthesis is clique size.

- Chess-related articles (9): Reuben Fine, Mikhail Botvinnik, Samuel Reshevsky, Max Euwe, FIDE, Alexander Alekhine, World Chess Championship, José Raúl Capablanca, AVRO 1938 chess tournament.

- Utah-related articles (9): Red Line (TRAX), Utah Transit Authority, Blue Line (TRAX), TRAX (light rail), Salt Lake City, Green Line (TRAX), FrontRunner, University of Utah, Utah.
- Articles related to Doctor Who (9): Doctor Who (film), Doctor Who, The Doctor (Doctor Who), Eighth Doctor, The Master (Doctor Who), Gallifrey, TARDIS, Doctor Who Magazine, Seventh Doctor.
- Space (9): New Horizons, Pioneer 11, Voyager 1, Europa (moon), Callisto (moon), Ganymede (moon), Jupiter, Io (moon), Pioneer 10.
- Hip hop music (9): G-funk, Dr. Dre, Death Row Records, Snoop Dogg, The Chronic, Gangsta rap, West Coast hip hop, N.W.A, Hip hop music.
- Metal music (9): Master of Puppets, Thrash metal, Cliff Burton, James Hetfield, Kirk Hammett, Metallica, Kill 'Em All, Ride the Lightning, Dave Mustaine.
- The Beatles (8): Break-up of the Beatles, The Beatles, George Harrison, Let It Be, John Lennon, Paul McCartney, Ringo Starr, Abbey Road.

Each Wikipedia article within any of these cliques has links to each article in clique.

Full lists of first 1000 largest cliques in English, Russian and Ukrainian Wikipedias plus source code of my scripts is here: <https://github.com/DennisYurichev/Math-for-programmers/tree/master/graph/clique/files/wikipedia>.

8.1.6 Social graph: LiveJournal spammers

LiveJournal is popular blogging platform in Russian-speaking Internet, which, as any other platform, flooded by spammers. I once tried, for experiment, to find a way to make distinction between them and human users. (I did this in 2010-2011, so this information may be not relevant these days).

Aside of false texts spammers posted to their blogs, spammers also mutually friended may spam accounts, so it was not unusual to register, let's say, 1000 fake accounts and friend each other.

If to build a graph of all links between LiveJournal users, and find largest cliques, there will be prominent unusually large cliques of LiveJournal users, up to 1000. In real world, you would not easily find a social group of 1000 persons who keeps mutual links with each other (there is interesting reading about it: [Dunbar's number](#)).

Well, spammers could lower this number, so each fake user would have 100-200 mutual friends instead of 1000 (which is less suspicious), but still, cliques were too perfect: each node connected to each other with very low amount of "external" links, leading to other spammer's cliques and human users.

8.1.7 Links graph: link farms

Talking about spammers, there was (or maybe still used today?) also a Black Hat SEO method to build "link farms": this is a collection of many websites which has links to each other. Interestingly, if you analyze link graph and find cliques, such farms are clearly visible.

8.1.8 Further reading

"SAT/SMT by Example"¹ has a short example, on how to find max cliques using MaxSAT.

¹https://yurichev.com/writings/SAT_SMT_by_example.pdf

Chapter 9

GCD and LCM

9.1 GCD

What is Greatest common divisor (GCD)?

Let's suppose, you want to cut a rectangle by squares. What is maximal square could be?

For a 14*8 rectangle, this is 2*2 square:

```
*****
*****
*****
*****
*****
*****
*****
*****
```

->

```
** ** ** ** ** ** ** ** ** 
** ** ** ** ** 
** ** ** ** 
** ** ** ** 
** ** ** ** 
** ** ** ** 
** ** ** ** 
** ** ** ** 
** ** ** ** 
** ** ** ** 
** ** ** ** 
** ** ** ** 
** ** **
```

What for 14*7 rectangle? It's 7*7 square:

```
*****
*****
*****
*****
*****
*****
*****
```

->

```
***** *****
***** *****
***** *****
***** *****
***** *****
***** *****
***** *****
```

14*9 rectangle? 1, i.e., smallest possible.

```
*****
*****
*****
*****
*****
*****
*****
*****
*****
*****
```

GCD of [coprimes](#) is 1.

GCD is also a common set of factors of several numbers. This we can see in Mathematica:

```
In [] := FactorInteger[300]
Out [] = {{2, 2}, {3, 1}, {5, 2}}

In [] := FactorInteger[333]
Out [] = {{3, 2}, {37, 1}}

In [] := GCD[300, 333]
Out [] = 3
```

i.e., $300 = 2^2 \cdot 3 \cdot 5^2$ and $333 = 3^2 \cdot 37$ and $GCD = 3$, which is smallest factor.

Or:

```
In []:= FactorInteger[11*13*17]
Out []= {{11, 1}, {13, 1}, {17, 1}}

In []:= FactorInteger[7*11*13*17]
Out []= {{7, 1}, {11, 1}, {13, 1}, {17, 1}}

In []:= GCD[11*13*17, 7*11*13*17]
Out []= 2431

In []:= 11*13*17
Out []= 2431
```

(Common factors are 11, 13 and 17, so $GCD = 11 \cdot 13 \cdot 17 = 2431$.)

9.2 Oculus VR Flicks and GCD

I've found this:

A flick (frame-tick) is a very small unit of time. It is $1/705600000$ of a second, exactly.

1 flick = $1/705600000$ second

This unit of time is the smallest time unit which is LARGER than a nanosecond, and can in integer quantities exactly represent a single frame duration for 24hz, 25hz, 30hz, 48hz, 50hz, 60hz, 90hz, 100hz, 120hz, and also $1/1000$ divisions of each.

This makes it suitable for use via `std::chrono::duration` and `std::ratio` for doing timing work against the system high resolution clock, which is in nanoseconds, but doesn't get slightly out of sync when doing common frame rates.

In order to accomodate media playback, we also support some common audio sample rates as well.

This list is not exhaustive, but covers the majority of digital audio formats. They are 8kHz, 16kHz, 22.05kHz, 24kHz, 32kHz, 44.1kHz, 48kHz, 88.2kHz, 96kHz, and 192kHz.

While humans can't hear higher than 48kHz, the higher sample rates are used for working audio files which might later be resampled or retimed.

The NTSC variations (~ 29.97 , etc) are actually defined as $24 * 1000/1001$ and $30 * 1000/1001$, which are impossible to represent exactly in a way where 1 second is exact, so we don't bother - they'll be inexact in any circumstance.

Details

```
1/24 fps frame: 2940000 flicks
1/25 fps frame: 2822400 flicks
1/30 fps frame: 2352000 flicks
1/48 fps frame: 1470000 flicks
1/50 fps frame: 1411200 flicks
1/60 fps frame: 1176000 flicks
1/90 fps frame: 784000 flicks
1/100 fps frame: 705600 flicks
1/120 fps frame: 588000 flicks
1/8000 fps frame: 88200 flicks
1/16000 fps frame: 44100 flicks
1/22050 fps frame: 32000 flicks
1/24000 fps frame: 29400 flicks
1/32000 fps frame: 22050 flicks
1/44100 fps frame: 16000 flicks
1/48000 fps frame: 14700 flicks
1/88200 fps frame: 8000 flicks
1/96000 fps frame: 7350 flicks
1/192000 fps frame: 3675 flicks
```

(<https://github.com/OculusVR/Flicks>)

Where the number came from? Let's enumerate all possible time intervals they want to use and find GCD using Mathematica:

```
In []:= GCD[1/24, 1/24000, 1/25, 1/25000, 1/30, 1/30000, 1/48, 1/50,
1/50000, 1/60, 1/60000, 1/90, 1/90000, 1/100, 1/100000, 1/120,
1/120000, 1/8000, 1/16000, 1/22050, 1/24000, 1/32000, 1/44100,
1/48000, 1/88200, 1/96000, 1/192000]
```

```
Out []= 1/705600000
```

Rationale: you may want to play a video with $\frac{1}{50}$ fps and, simultaneously, play audio with 96kHz sampling rate. Given that, you can change video frame after each 14112000 flicks and change one audio sample after each 7350 flicks. Use any other video fps and any audio sampling rate and you will have all time periods as integer numbers. No ratios any more.

On contrary, one nanosecond wouldn't fit: try to represent 1/30 second in nanoseconds, this is (still) ratio: 33333.33333... nanoseconds.

9.3 LCM

Many people use **LCM**¹ in school. Sum up $\frac{1}{4}$ and $\frac{1}{6}$. To find an answer mentally, you ought to find Lowest Common Denominator, which can be $4 \cdot 6 = 24$. Now you can sum up $\frac{6}{24} + \frac{4}{24} = \frac{10}{24}$.

But the lowest denominator is also a LCM. LCM of 4 and 6 is 12: $\frac{3}{12} + \frac{2}{12} = \frac{5}{12}$.

¹Least Common Multiple

9.3.1 File copying routine

Buffer: A storage device used to compensate for a difference in data rate of data flow or time of occurrence of events, when transmitting data from one device to another.

Clarence T. Jones, S. Percy Jones –
Patrick-Turner's Industrial
Automation Dictionary

In GNU coreutils, we can find that LCM is used to find optimal buffer size, if buffer sizes in input and output files are different. For example, input file has buffer of 4096 bytes, and output is 6144. Well, these sizes are somewhat suspicious. I made up this example. Nevertheless, LCM of 4096 and 6144 is 12288. This is a buffer size you can allocate, so that you will minimize number of read/write operations during copying.

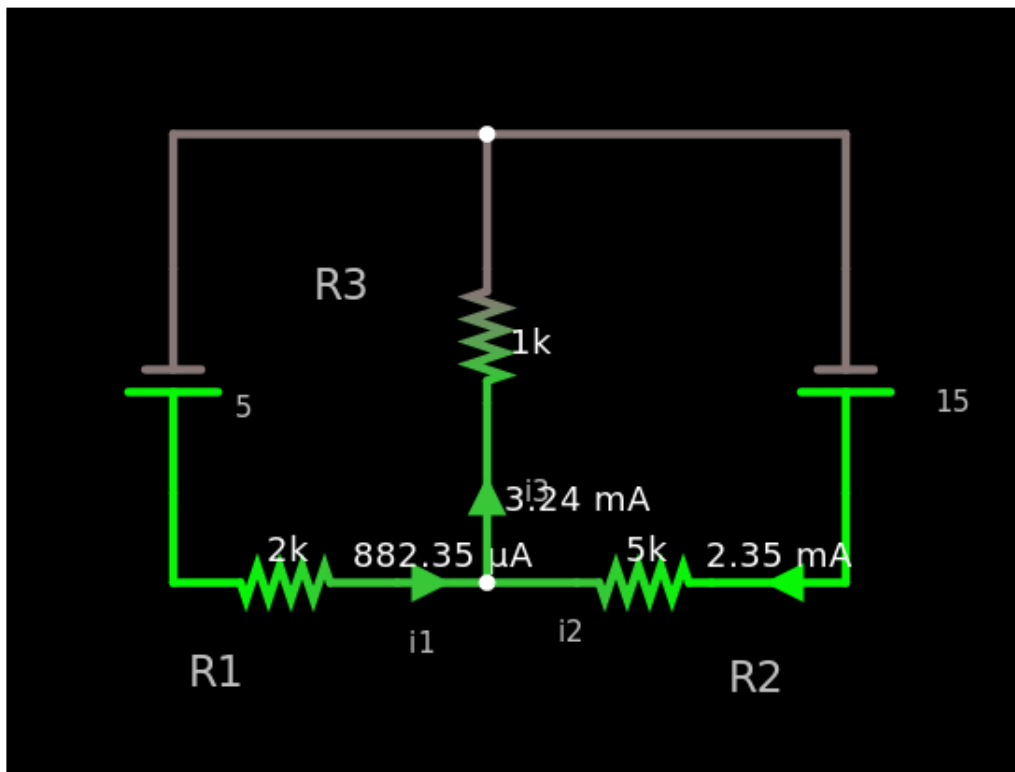
<https://github.com/coreutils/coreutils/blob/4cb3f4faa435820dc99c36b30ce93c7d01501f65/src/copy.c#L1246>. <https://github.com/coreutils/coreutils/blob/master/gl/lib/buffer-lcm.c>.

Chapter 10

Linear algebra

10.1 Gaussian elimination: Kirchhoff's circuit laws

The circuit I've created on falstad.com¹:



Click here to open it on their website and run: <http://tinyurl.com/y8raoud3>.

The problem: find all 3 current values in 2 loops. This is usually solved by solving a system of linear equations.

Overkill, but Z3 SMT-solver can be used here as well, since it can solve linear equations as well, over real numbers:

```
from z3 import *  
  
i1, i2, i3 = Reals ("i1 i2 i3")
```

¹<http://falstad.com/circuit/>

```

R1=2000
R2=5000
R3=1000

V1=5 # left
V2=15 # right

s=Solver()

s.add(i3 == i1+i2)

s.add(V1 == R1*i1 + R3*i3)
s.add(V2 == R2*i2 + R3*i3)

print s.check()
m=s.model()
print m
print m[i1].as_decimal(6)
print m[i2].as_decimal(6)
print m[i3].as_decimal(6)

```

And the result:

```

sat
[i3 = 11/3400, i1 = 3/3400, i2 = 1/425]
0.000882?
0.002352?
0.003235?

```

Same as on falstad.com online simulator.

Z3 represents real numbers as fractions, then we convert them to numerical form...

Further work: take a circuit as a graph and build a system of equations.

10.1.1 Gaussian elimination

SMT-solver is overkill, these linear equations can be solved using simple and well-known Gaussian elimination.

First, we rewrite the system of equation:

$$\begin{array}{rcl}
 i1 + & i2 - & i3 == 0 \\
 R1*i1 + & & R3*i3 == V1 \\
 & R2*i2 + & R3*i3 == V2
 \end{array}$$

Or in matrix form:

```
[ 1, 1, -1 | 0 ]
[ 2000, 0, 1000 | 5 ]
[ 0, 5000, 1000 | 15 ]
```

I can solve it using Wolfram Mathematica, using RowReduce².

```
In[1]:= RowReduce[{{1, 1, -1, 0}, {2000, 0, 1000, 5}, {0, 5000, 1000, 15}}]
Out[1]= {{1,0,0,3/3400},{0,1,0,1/425},{0,0,1,11/3400}}

In[2]:= 3/3400//N
Out[2]= 0.000882353

In[3]:= 1/425//N
Out[3]= 0.00235294

In[4]:= 11/3400//N
Out[4]= 0.00323529
```

This is the same result: i1, i2 and i3 in numerical form.

ReduceRow's output is:

```
[ 1,0,0 | 3/3400 ]
[ 0,1,0 | 1/425 ]
[ 0,0,1 | 11/3400 ]
```

... back to expressions, this is:

```
1*i1 + 0*i2 + 0*i3 = 3/3400
0*i1 + 1*i2 + 0*i3 = 1/425
0*i1 + 0*i2 + 1*i3 = 11/3400
```

In other words, this is just what i1/i2/i3 are.

Now something down-to-earth, C example I've copied from Rosetta Code³, working with no additional libraries, etc:

```
// copied from https://rosettacode.org/wiki/Gaussian\_elimination#C

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define mat_elem(a, y, x, n) (a + ((y) * (n) + (x)))
```

²<http://reference.wolfram.com/language/ref/RowReduce.html>

³https://rosettacode.org/wiki/Gaussian_elimination#C

```

void swap_row(double *a, double *b, int r1, int r2, int n)
{
    double tmp, *p1, *p2;
    int i;

    if (r1 == r2) return;
    for (i = 0; i < n; i++) {
        p1 = mat_elem(a, r1, i, n);
        p2 = mat_elem(a, r2, i, n);
        tmp = *p1, *p1 = *p2, *p2 = tmp;
    }
    tmp = b[r1], b[r1] = b[r2], b[r2] = tmp;
}

void gauss_eliminate(double *a, double *b, double *x, int n)
{
#define A(y, x) (*mat_elem(a, y, x, n))
    int i, j, col, row, max_row, dia;
    double max, tmp;

    for (dia = 0; dia < n; dia++) {
        max_row = dia, max = A(dia, dia);

        for (row = dia + 1; row < n; row++)
            if ((tmp = fabs(A(row, dia))) > max)
                max_row = row, max = tmp;

        swap_row(a, b, dia, max_row, n);

        for (row = dia + 1; row < n; row++) {
            tmp = A(row, dia) / A(dia, dia);
            for (col = dia+1; col < n; col++)
                A(row, col) -= tmp * A(dia, col);
            A(row, dia) = 0;
            b[row] -= tmp * b[dia];
        }
    }
    for (row = n - 1; row >= 0; row--) {
        tmp = b[row];
        for (j = n - 1; j > row; j--)
            tmp -= x[j] * A(row, j);
        x[row] = tmp / A(row, row);
    }
#undef A
}

int main(void)
{
    double a[] = {
        1, 1, -1,

```

```
        2000, 0, 1000,  
        0, 5000, 1000  
};  
double b[] = { 0, 5, 15 };  
double x[3];  
int i;  
  
gauss_eliminate(a, b, x, 3);  
  
for (i = 0; i < 3; i++)  
    printf("%g\n", x[i]);  
  
return 0;  
}
```

I run it:

```
0.000882353  
0.00235294  
0.00323529
```

See also: https://en.wikipedia.org/wiki/Gaussian_elimination,
<http://mathworld.wolfram.com/GaussianElimination.html>.

But a fun with SMT solver is that we can solve these equations without any knowledge of linear algebra, matrices, Gaussian elimination and whatnot.

According to the source code of Z3, it can perform Gaussian Elimination, perhaps, whenever it can do so.

Some people try to use Z3 to solve problems with operational amplifiers: [1](#), [2](#).

Chapter 11

Acronyms used

GCD Greatest Common Divisor	5
LCM Least Common Multiple	112
LCG Linear congruential generator	20
CPU Central processing unit	98
FPU Floating-point unit	98
PRNG Pseudorandom number generator	17
CRC Cyclic redundancy check	4
RSA Rivest–Shamir–Adleman cryptosystem	3
TAOCP The Art Of Computer Programming (Donald Knuth’s book)	98
MSB Most Significant Bit	66