Introduction to logarithms

Dennis Yurichev <dennis(a)yurichev.com>

12-August-2015

Contents

1 Introduction

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 $log_{10}(googol) = 100$.

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

Listing 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 2: Wolfram Mathematica

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

So it has 39 digits.

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

Listing 3: Wolfram Mathematica

```
In[]:= 2^1024
Out[]= 179769313486231590772930519078902473361797697894230657273430081\
157732675805500963132708477322407536021120113879871393357658789768814\
416622492847430639474124377767893424865485276302219601246094119453082 9\
520850057688381506823424628814739131105408272371633505106845862982399 4\
724593847971630483535632962422413721 6

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

309 decimal digits.

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:

$$(sign)significand \times 10^{exponent} \tag{1}$$

For example:

$$-1.987126381 \times 10^{41} \tag{2}$$

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).

2 Logarithmic scale

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:

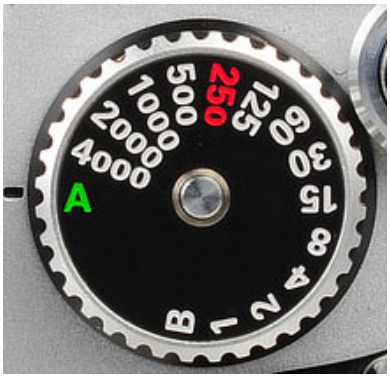


Figure 1: Shutter speed ($\frac{1}{x}$ of second) knob on photo camera

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.

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 in there: http://www.3dnews.ru/270838/page-3.html.

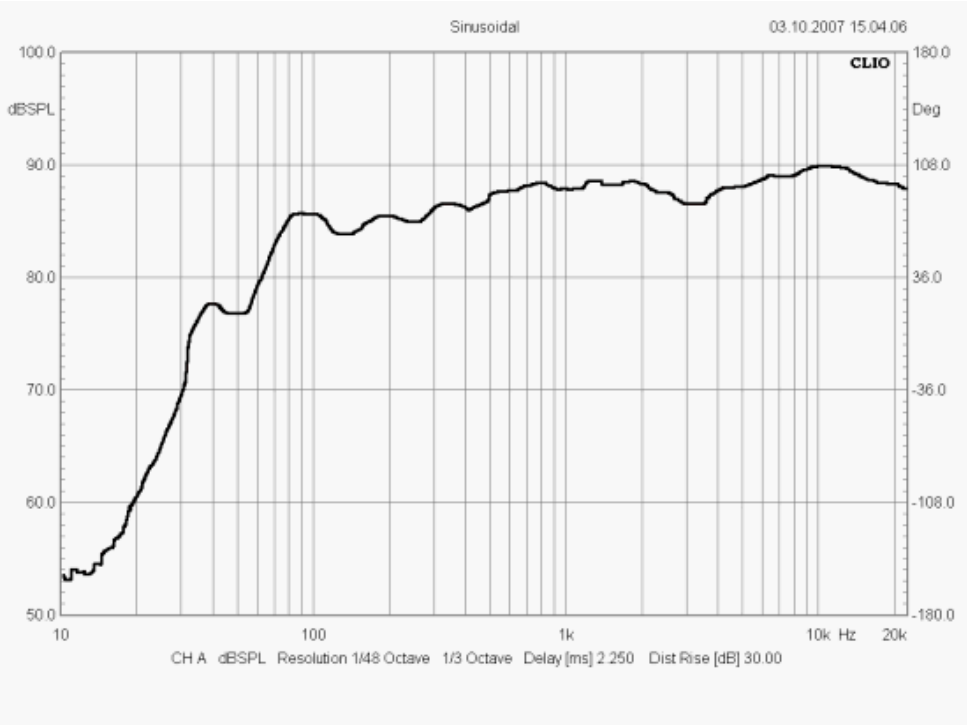


Figure 2: Frequency response (also known as *Bode plot*) of some loudspeaker

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

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

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.

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].

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_Ox.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_O3.s     |   4 +
 patterns/185_64bit_in_32_env/conversion/MSVC2012_Ox.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.

[2] Some of speakers of USSR era (like Latvian Radiotehnika 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:

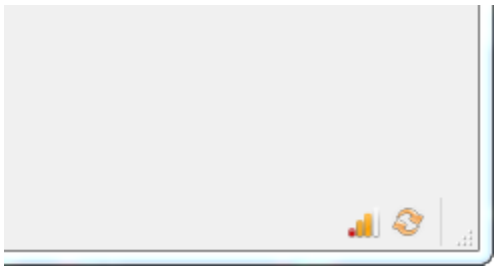


Figure 3: Bitcoin client

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 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.

2.4 Web 2.0

Sites like GitHub, Reddit, Twitter sometimes shows how long some event was ago, instead of precise date. 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 interesting in precision down to days and hours. When something happens 2 years ago, users usually not interesting in number of months and days in addition to these 2 years.

3 Multiplication and division using addition and subtraction

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

$$\log_{base}(ab) = \log_{base}(a) + \log_{base}(b) \tag{3}$$

…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 if 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.

3.1 Logarithmic slide rule

Here is very typical slide rule[3]. It has many scales, but take a look on C and D scales, they are the same:

[3] I took screenshots at http://museum.syssrc.com/static/sliderule.html

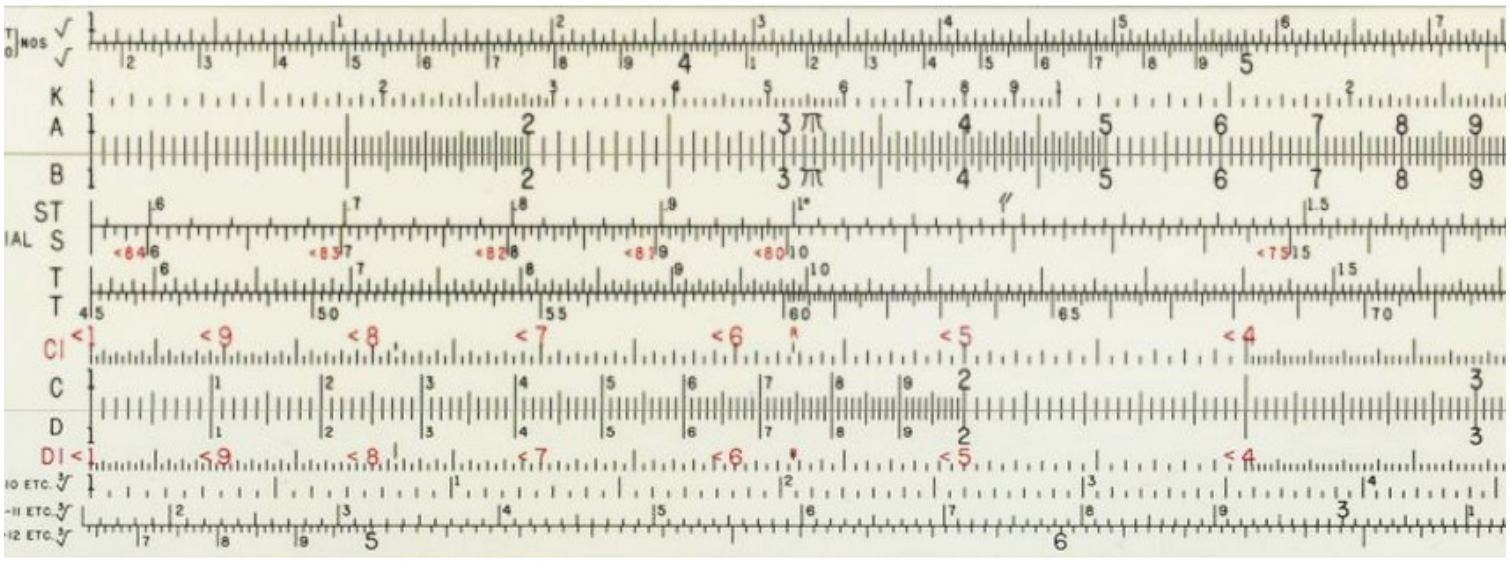


Figure 4: Initial state of slide rule

Now shift the core of rule so C scale at 1 will point to 1.2 at D scale:

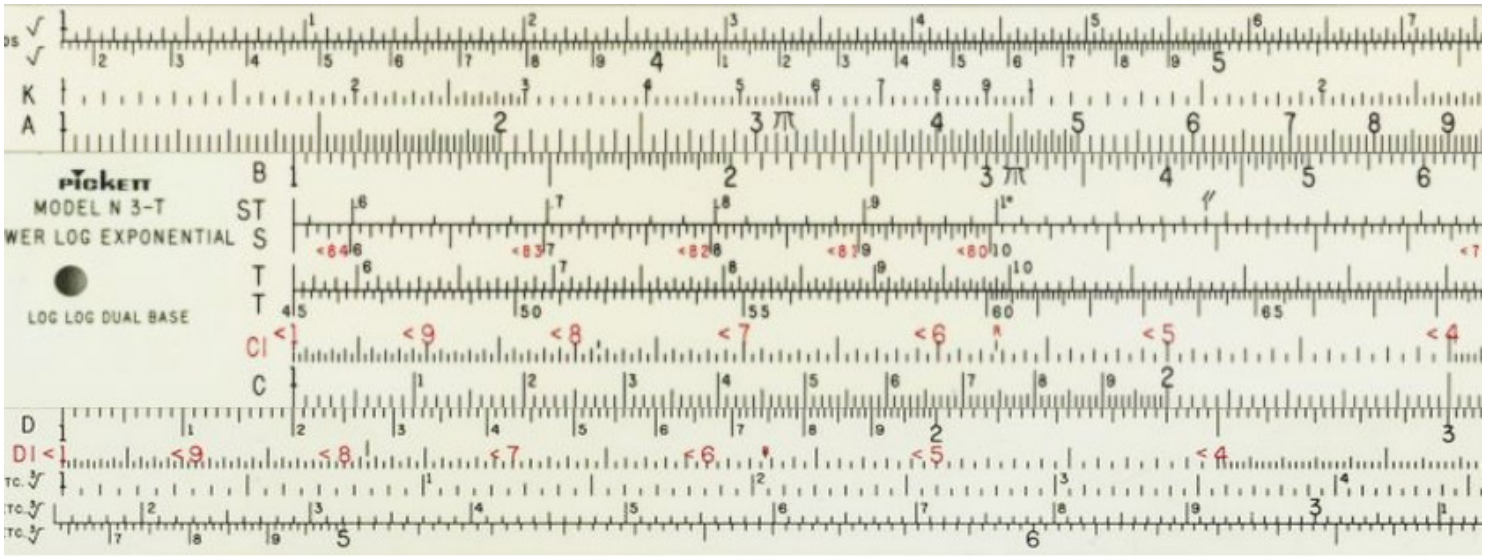


Figure 5: C scale shifted

Find 2 at C scale and find corresponding value at D scale (which is 2.4). Indeed, $1.2 \cdot 2 = 2.4$. It works because by sliding scales we actually add distance between 1 and 1.2 (at any scale) to the distance between 1 and 2 (at any scale). But since these scales logarithmic, addition of logarithmic values is the same as multiplication.

Values on scales can be interpreted as values of other order of magnitude. We can say that 1 at C scale is actually point to 12 at D scale. Find 1.8 at D scale (which is 18 now), it points somewhere between 21 and 22. It's close: $12 \cdot 18 = 216$.

It works because of equation 3.

Here is another example from Wikipedia:

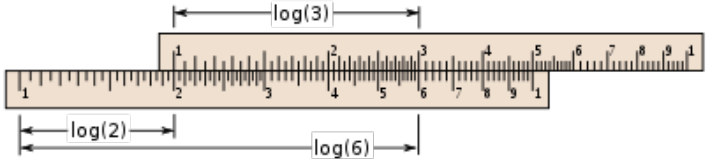


Figure 6: Example from Wikipedia

3.2 Logarithmic tables

As we can see, the precision of logarithmic slie 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 maybe 4 digits.

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

Таблица XIII. МАНТИССЫ ДЕСЯТИЧНЫХ ЛОГАРИФМОВ.

N	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 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).

m	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 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).

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) numbers [4] encoded in IEEE 754 standard.

Here is my attempt to calculate $\frac{1.234\times10^{-300}\cdot2.345678901234\times10^{-24}}{3.456789\times10^{-50}}$:

Listing 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 $1.42926179712226146096698338819 0 \times 10^{-274}$, 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 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 $2.894567764122756*10^{-324}$, 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 $ln(x)$)):

Listing 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 $8.37357375333871021628112579215 0 \times 10^{-275}$, same as Mathematica reported.

The same problem with very large numbers.

Listing 7: C code

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

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

```
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 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 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 slower than multiplication itself.

3.4 IEEE 754: adding and subtracting exponents

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

$$(-1)^{sign} \cdot significand \times 2^{exponent} \tag{4}$$

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:

$$significand_1 \times 2^{10} \cdot significand_2 \times 2^{50} = significand_3 \times 2^{60} \tag{5}$$

…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.

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

$$\frac{significand_1 \times 2^{10}}{significand_2 \times 2^{50}} = significand_3 \times 2^{-40} \tag{6}$$

I don't have access to Intel or AMD FPU internals, but I can peek into OpenWatcom FPU emulator libraries [6].

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.

4 Exponentiation

Using equation 3 we may quickly notice that

$$b^n = \underbrace{b \times \cdots \times b}_{n} = base^{(\log_{base}(b)) * n} \tag{7}$$

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_powl.S#L189:

Listing 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   MO(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.

5 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:

[6] It 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.

$$\log_y(x) = \frac{\log_a(x)}{\log_a(y)} \tag{8}$$

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

$$\log_{10}(x) = \frac{\log_2(x)}{\log_2(10)} \tag{9}$$

…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:

$$\log_y(x) = \frac{1}{\log_x(y)} \tag{10}$$

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:

$$\log_y(x) = \log_a(x) \cdot \log_y(a) \tag{11}$$

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

$$\log_{10}(x) = \log_2(x) \cdot \log_{10}(2) \tag{12}$$

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 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    Yl2XArgNegative         ;    return a NAN
        fyl2x                           ; compute y*log2(x)
        ret
```

6 Binary logarithm

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

6.1 Denoting a number of bits for some value

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

Listing 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; 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<<log_n`).

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

Listing 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 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 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 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 17: C code

```
                /*
                 * Secondary cache information is encoded as WWLLSSSS, 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 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".

[7] http://www.linux-mips.org/wiki/IP22

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

Listing 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 $x = y << 10$.

6.2 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 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);
 }
```

Latest x86 CPUs has LZCNT (Leading Zeroes CouNT) instruction for that [10], 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://chessprogramming.wikispaces.com/BitScan.

6.3 O(log n) time complexity

Time complexity[11] is a measure of speed of a specific algorithm in relation to the size of input data.

[9] http://fossies.org/linux/seamonkey/mozilla/mfbt/MathAlgorithms.h

[10] GNU `__builtin_clz()` function on x86 architecture can be thunk for LZCNT

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

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 game. One player think about some number, the other should guess it, offering various versions. First player answers, is guessed number is larger or less. Typical dialogue can be as the follows:

```
-- 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 lenght of 1, and this is correct answer. Maximal number of steps using the strategy described here are $\log_2(initial\_range)$. 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.

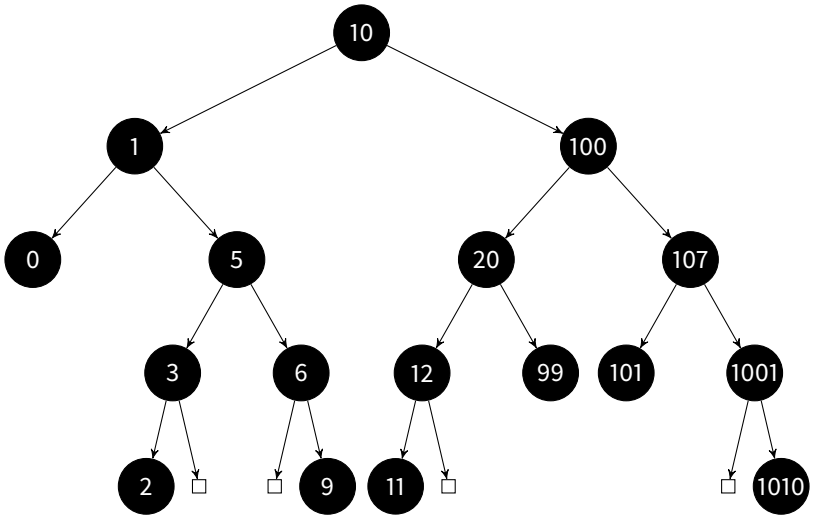
This is exactly O(log n) time complexity.

Now let's consider couple of practical real-world algorithms. One interesting thing is that if the input array is sorted, 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 search[12], and there is the `bsearch()` function in standard C/C++ library[13].

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.

[12] https://en.wikipedia.org/wiki/Binary_search_algorithm
[13] http://en.cppreference.com/w/cpp/algorithm/bsearch



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 $\log_n(number\_of\_nodes)$. We've got 17 elements in the tree at the picture, $\log_2(17) = 4.08...$, indeed, there are 5 tiers in the tree.

7 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 $log_{10}(2^{64}) = 19.2...$.

Functions like `itoa()`[14] (which converts input number to a string) can calculate output buffer size precisely, calculating common logarithm of the input number.

8 Natural logarithm

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

8.1 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 20: Python code

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

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

[14] http://www.cplusplus.com/reference/cstdlib/itoa/

```
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 21: 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):
        # 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 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
```

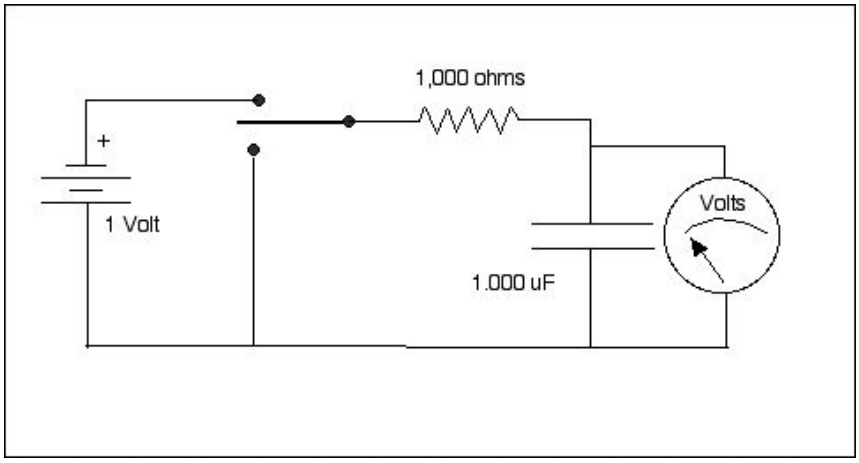
```
year= 0 month= 0 day= 0 amount 100.006944444
year= 0 month= 0 day= 1 amount 100.013889371
year= 0 month= 0 day= 2 amount 100.02083478
year= 0 month= 0 day= 3 amount 100.027780671
...
year= 39 month= 11 day= 26 amount 271.762123927
year= 39 month= 11 day= 27 amount 271.780996297
year= 39 month= 11 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 $100 \cdot e$.

8.2 Exponential decay

8.2.1 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 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, 0.128}]}, AxesLabel -> {seconds, voltage}]
```

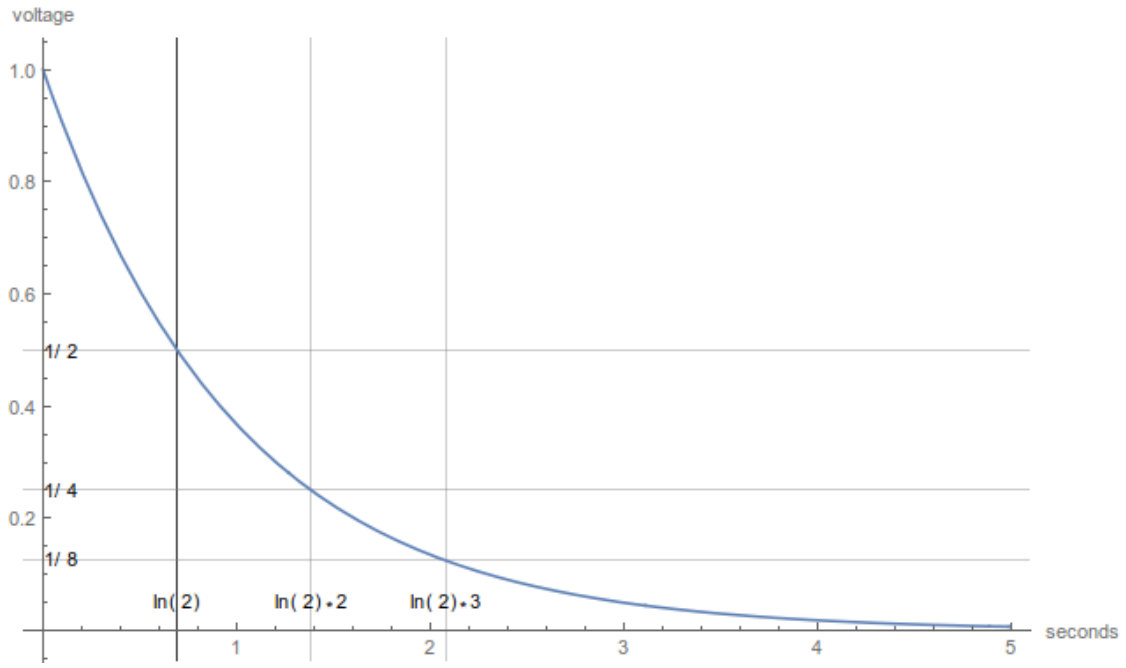


Figure 9: Capacitor voltage during discharge

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 interesting 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 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}]
```

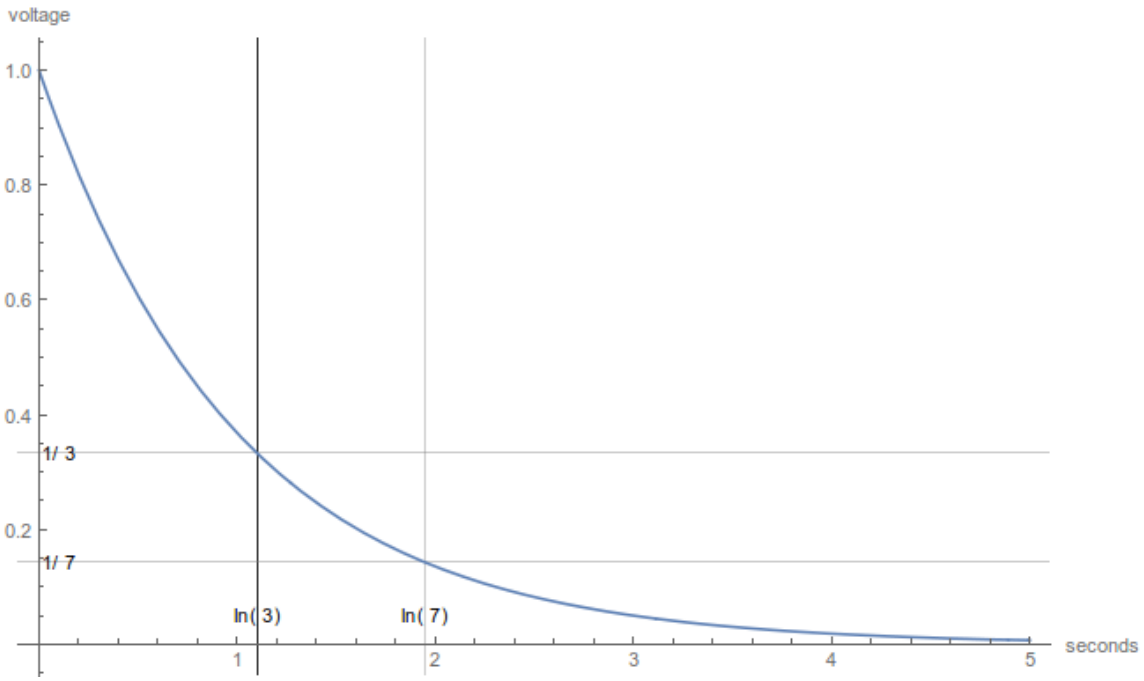


Figure 10: Capacitor voltage during discharge

…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.

8.2.2 Radioactive decay

Radioactive decay is also exponential decay. Let's take Polonium 210 as an example[15]. 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 lead[16]). 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 25: Wolfram Mathematica

```
l = 0.005009157516910051; (* decay constant of Polonium 210 *)

hl = Log[2]/l
138.376

Plot[E^(-l*t), {t, 0, 1000},
 GridLines -> {{hl, hl*2, hl*3}, {0.5, 0.25, 0.125}},
 Epilog -> {Text["hl", {hl, 0.05}], Text["hl*2", {hl*2, 0.05}],
   Text["hl*3", {hl*3, 0.05}], Text["1/2", {30, 0.5}],
   Text["1/4", {30, 0.25}], Text["1/8", {30, 0.128}]},
 AxesLabel -> {days, atoms}]
```

[15] https://en.wikipedia.org/wiki/Polonium
[16] https://en.wikipedia.org/wiki/Isotopes_of_lead#Lead-206

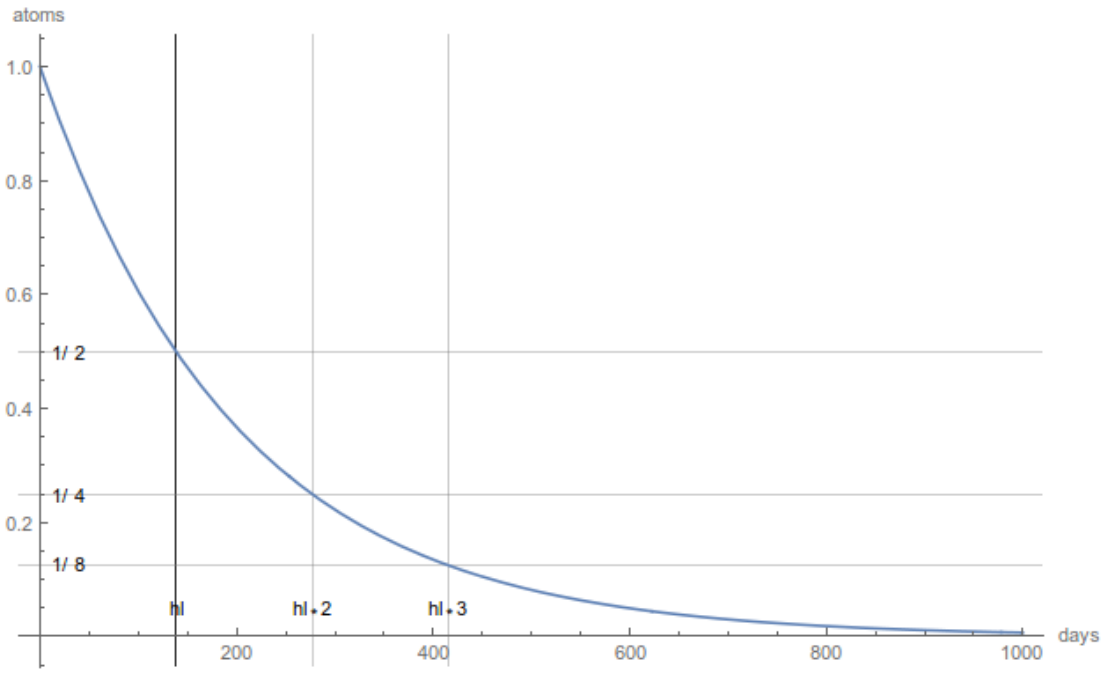


Figure 11: Exponential decay of Polonium 210

8.2.3 Beer froth

There is even the article (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.

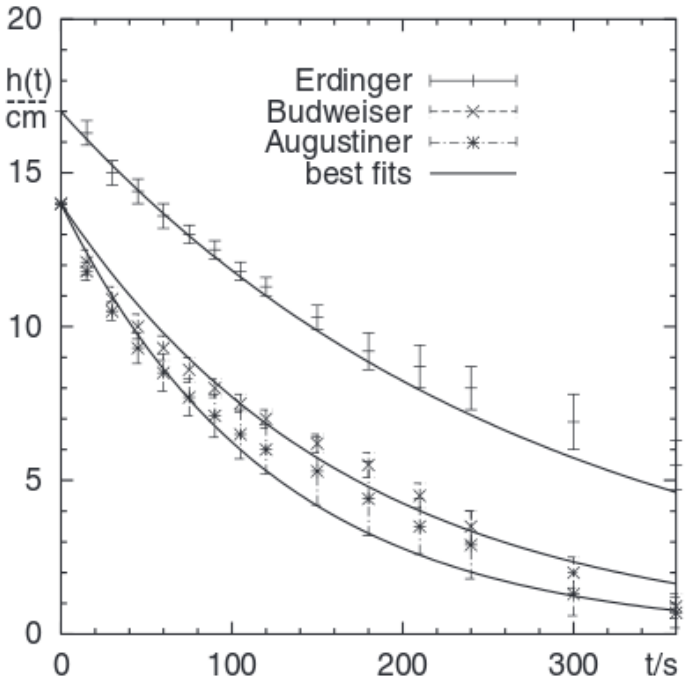


Figure 12: Results from the article

Article can be taken as a joke, nevertheless, it's a good demonstration of exponential decay.

8.2.4 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.