Numerals

John Monash Science School
Co-curricular term two 2022

Tyson Jones
tyson.jones.input@gmail.com

Contents

1 Foreward

These class notes are about *numerals*; methods of notating numbers through squiggles and marks on a page. This innocuous topic reaches into all sorts of insidious areas like combinatorics, information theory, digital logic and bike theft.

These notes were tailored for precocious year ten Australian highschoolers. However, almost no prior maths is assumed (relevant topics are reviewed in Sec. 5), and the contents may prove especially elucidating for programmers and undergraduate computer scientists. If counting excites you, strap yourself in.

I finally warn that the ramblings herein are highly stylised, joke-riddled and strikingly tone-deaf, and surveyed at the reader's discretion. No offence is intended to the following parties:

- Late antiquity Indian mathematicians
- Chinese accountants
- Donald Knuth
- The Yuki, Mayan, and Sumerian peoples
- The likes of programmers
- Combination lock manufacturers and their sham-fuelled industry
- Europe
- Those presently incarcerated
- American libertarians
- Authors of the IEEE 754 technical standard
- The Australian teaching pay commission
- Victims of Russian occupation
- Subscribers to the New Testament
- Apple fans
- BIC (don't sue)

2 Integers

2.1 Base 10

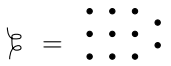
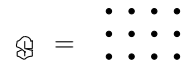
Natural numbers are all around us. My left hand has as many fingernails as there are dots below:

Due to a gnawing incident, my right foot features this many toes:

Sixth century Indians were sick of counting their missing toes in this cumbersome manner, so agreed upon these symbols:

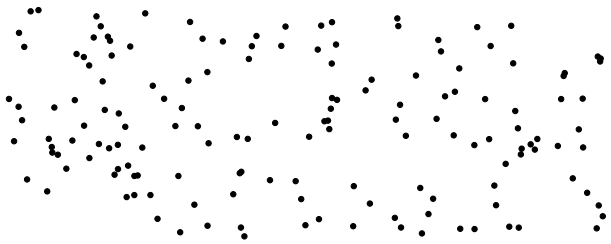
$0 =$	$5 =$
$1 =$	$6 =$
$2 =$	$7 =$
$3 =$	$8 =$
$4 =$	$9 =$

They adopted *ten* unique numeral symbols, and the Indian embassy have since ignored my emails proposing additional symbols

We are ergo stuck with a "base  " system, or in its own language, "base ten". Though it seems a comfortable choice of base for ten-fingered humans, the Mayans decided on twenty (the number of fingers and toes), the Yuki used eight (the number of spaces between fingers), and the Babylonians created sixty symbols before anybody could stop them.

The Indians soon realised ten numbers were too few; they could not denote the number of sunrises in a season, nor the number of instagram likes on a very popular Bhimbetka painting. Since their numeral symbol designer had gone missing in Mesopotamia, the Indians developed the Hindu-Arabic "positional notation" to denote big numbers using *multiple* of their existing numeral symbols, written side-by-side. For example,

$153 =$ 

In this notation, the symbols or "digits" are read right-to-left, have their values multiplied with an increasingly large number, and are finally summed. The sequence 153 is interpreted as number

$$153 \quad = \quad 3 \times 10^0 \; + \; 5 \times 10^1 \; + \; 1 \times 10^2$$

I wrote that compactly by making use of your existing understanding of symbols $+$ and $\times$, and of the numerical value $ten = 10$, and of the power notation $base^{exponent}$. To avoid all that, I could verbosely write

153 = 3 added to
5 of (my number of fingers), added to
1 of (my number of fingers of (my number of fingers))

The position notation is convenient because the represented numbers can become very big while their numerals remain concise.

2 =
22 =
222 =

The largest four-digit base-10 number is 9999. It is the choice of the *maximum* value symbol (9) for every digit. What is the value of the largest k-digit base-10 number, where $k \in \mathbb{N}$ is unspecified?

We should also make mention here of *negative* numbers, later proving indispensable by Indian economists when quantifying the contributions of the British empire. While expressions like

$$5 - 2 = 3$$

were okay, those like

$$2 - 5 = ??$$

really knocked the socks off of third century Greek mathematicians, who had neglected to read the *negative* second century Chinese literature. The spooky negative variant of an integer $x \in \mathbb{N}$ is defined as that resulting from subtraction from zero:

$$0 - x = -x$$

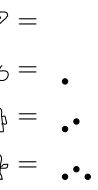
The "$-$" prefix of a negative numeral is still arbitrary notation however, and history played with many:

$$-5 \quad = \quad \bar{5} \quad = \quad \varsigma \quad = \quad \not{5} \quad = \quad 5$$

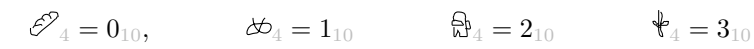
Negative numbers are very important for denoting very small positive numbers, as we will later see.

2.2 Base *B*

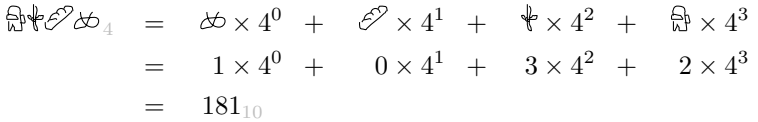
We are so well practised at using a base *ten* number system that we may mistake it as innate. But it *is* arbitrary, and no more legitimate than the base *four* system I hereby propose:



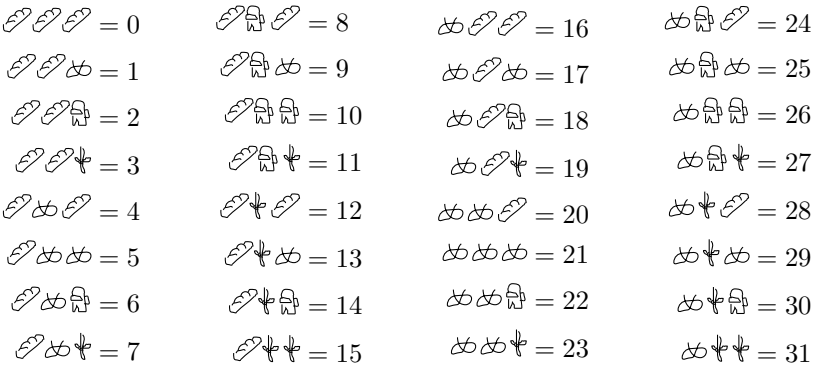
We use subscripts to indicate the base of the denoted number, in case of ambiguity. For instance, $101_{10} \neq 101_2 = 5_{10}$. To defer insanity, the subscripted numbers are *always* encoded in base-10. In our scheme,



Do not bother nominating this scheme to the Indian embassy, for they have cemented their eggs into the base ten basket. But we can anyway steal their positional notation to denote big numbers:



Counting up from zero follows the same procedure as counting in base ten: We increment the right-most ("least significant") digit. If it surpasses its maximum value symbol ("overflows"), it is reset to its minimum value symbol, and the next digit is incremented similarly (a "carry").



The largest three-digit number in our base-four scheme is

$$\begin{aligned} \text{⚘⚘⚘}_4 &= 3 \times 4^0 \ + \ 3 \times 4^1 \ + \ 3 \times 4^2 \\ &= 63_{10} \end{aligned}$$

But we didn't need to evaluate this - we could have immediately written down the answer. Recognise that the *next* integer...

$$\text{⚘⚘⚘} \ + \ \text{☙} \ = \ \text{☙🦴🦴🦴}$$

is three zeros (🦴) and a single one (☙), and ergo has has value

$$= 1 \times 4^3$$

The largest three-digit number is therefore one fewer than the smallest four-digit number:

$$\begin{aligned} \text{⚘⚘⚘}_4 &= 4^3 - 1 \\ &= 63_{10} \end{aligned}$$

Let us now consider a general base B number system, where B itself is an unspecified natural number ($B \in \mathbb{N}$). Such a system must employ B unique numeral symbols (to denote values 0, 1, ..., $B - 1$) but we will leave them unspecified too. In this way, we can study number systems of *all* bases at once, including those natural to *eleven* fingered aliens ($B = 11$) and *ninety-six* tentacled Shima monster octopi ($B = 96$). We even describe the hexadecimal system ($B = 16$) adopted by the ghastliest beast of them all: the unwashed programmer.

Let variables a, b, c, $d \in \mathbb{N}$ represent the individual digits (in base B, increasing significance) of a particular four-digit number $x \in \mathbb{N}$.

$$x_B = [d][c][b][a]$$

Note that algebra is agnostic to the base of the numbers it abstracts, so we can write expressions like $x + y$ without writing the base subscript. However, the above equation defines a, b, c, d as the specific digits of x in base B, hence the subscript. To be valid digits (which have not overflowed), they satisfy

$$\begin{aligned} 0 \leq a \ &< B & \qquad 0 \leq b \ &< B \\ 0 \leq c \ &< B & \qquad 0 \leq d \ &< B \end{aligned}$$

This is compactly written with an integer interval

$$a, b, c, d \ \in \ [0..B)$$

Evaluating x (given base-B digits a, b, c, d) as a base-10 number according to the positional convention is as before:

$$x_{10} \quad = \quad a \times B^0 \ + \ b \times B^1 \ + \ c \times B^2 \ + \ d \times B^3$$

Consider how the following specific numbers are represented in base B:

- $$x = 0_{10}$$

  Representing zero is trivial; every digit has the minimum value symbol denoting zero, $d = c = b = a = 0_B$.

  $$\therefore\ x_B = [0][0][0][0]$$

- $$x = B\ -\ 1_{10}$$

  This happens to be the maximum value of a single digit, so only the least significant digit is non-zero:

  $$x_B = [0][0][0][B-1]$$

- $$x = B$$

  Since this satisfies $x \geq B$ (equals or exceeds the base), one or more non-least-significant digits in x_B must be non-zero.

  $$x_B = [0][0][1][0]$$

- $$x = \text{the biggest four-digit number representable}$$

  This is the number whereby every digit has its maximum value symbol (analogous to 9999_{10}). Ergo

  $$x_B = [B-1][B-1][B-1][B-1]$$
  $$\begin{aligned}\therefore\ x &= (B-1) \times B^0\ \ +\ \ (B-1) \times B^1\ \ +\ \ (B-1) \times B^2\ \ +\ \ (B-1) \times B^3 \\ &= (B-1)(B^0 + B^1 + B^2 + B^3)\end{aligned}$$

  Though once again, we should instead realise x is *one less* than the *smallest* five-digit number, so conclude

  $$x_B = [1][0][0][0][0] - [0][0][0][0][1]$$
  $$\therefore\ x = B^5 - 1$$

- $$x = \text{the biggest } k\text{-digit number representable}$$

  Using the same trick, we realise x is one less than the smallest $(k+1)$-digit number, which is a *one* followed by k *zeroes*. Hence

  $$x = B^k - 1$$

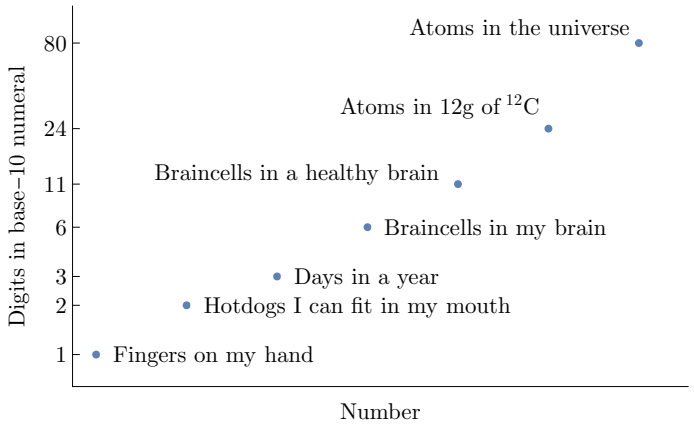
The final example above is profound. It shows that under the positional notation, the *range* of representable numbers grows *exponentially* with the number of digits.

$$\text{range of base-}B\ k\text{-digit numeral} = B^k$$

This is why our numerals are so *concise*; we need only a logarithmic number of digits to denote a given integer $x \in \mathbb{N}$.

$$\text{number of base-}B\text{ digits} = 1 + \lfloor \log_B(x) \rfloor$$

The notation $\lfloor z \rfloor$ is the *floor* of z, merely rounding z down to the next integer. Logarithms grow as slowly as novel coronaviruses proliferate rapidly in unvaccinated communities.



Of course, all of these numerals can be prepended with *zero* to technically increase their number of digits *ad nauseam*.

$$13 = 013 = 0013 = 0000000000000013$$

We usually agree not to do so. The exponentially expressive positional notation enables us to write down all kinds of wonderfully big numbers without exhausting ourselves nor running out of ink. We must thank forward thinking pioneers like tenth century Abu'l-Hasan al-Uqlidisi for foiling the BIC capitalist agenda.

2.3 Examples

I hope to have convinced you that the choices of base and symbols of a number system are arbitrary. Yet you will find that your friends don't show up when you schedule them to meet at "amogus ceiling-fan baguette pretzel" o-clock. Humans are particular about their numerals and recognise only a few with any real seriousness, recycling the Hindu-Arabic numeral symbols and the Anglo-Saxon futhorc runic alphabet.

For instance, the $B = 1$ "*unary*" tally mark scheme:

$$\begin{aligned} \text{I} &= 1_{10} & \text{𝍸 I} &= 6_{10} \\ \text{II} &= 2_{10} & \text{𝍸 II} &= 7_{10} \\ \text{III} &= 3_{10} & \text{𝍸 III} &= 8_{10} \\ \text{IIII} &= 4_{10} & \text{𝍸 IIII} &= 9_{10} \\ \text{𝍸} &= 5_{10} & \text{𝍸 𝍸} &= 10_{10} \end{aligned}$$

While verbose, tally marks are useful for incrementing or counting in settings where the previous numerals cannot be erased, such as scratches upon a prison cell wall using Joey-the-rat's canine tooth. Notice there is no established symbol for zero; thankfully nobody starts counting until the shower experience on day two.

The *hexadecimal* system ($B = 16$) extends the base-10 numerals with symbols A to F.

$$\begin{aligned} 0_{16} &= 0_{10} & 5_{16} &= 5_{10} & \text{A}_{16} &= 10_{10} & \text{F}_{16} &= 15_{10} \\ 1_{16} &= 1_{10} & 6_{16} &= 6_{10} & \text{B}_{16} &= 11_{10} & 10_{16} &= 16_{10} \\ 2_{16} &= 2_{10} & 7_{16} &= 7_{10} & \text{C}_{16} &= 12_{10} & &\vdots \\ 3_{16} &= 3_{10} & 8_{16} &= 8_{10} & \text{D}_{16} &= 13_{10} & \text{AA}_{16} &= 10 \times 16 + 10 = 170_{10} \\ 4_{16} &= 4_{10} & 9_{16} &= 9_{10} & \text{E}_{16} &= 14_{10} & \text{FF}_{16} &= 15 \times 16 + 15 = 255_{10} \end{aligned}$$

Hexadecimal numerals are used both by assembly programmers to denote the possible values of a *nibble* (four bits) and by web programmers to encode colours in order to confuse graphic artists. Since a "hex-colour" is specified as a *six* digit base-16 numeral, we know there are only 16^6 unique colours no matter what the computer monitor companies try to sell us.

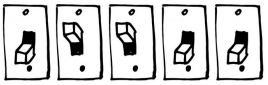
And a *bit* (a 0 or 1) is itself a single digit of a *binary* ($B = 2$) numeral.

$$\begin{aligned} 000_2 &= 0_{10} \\ 001_2 &= 1_{10} \\ 010_2 &= 2_{10} \\ 011_2 &= 3_{10} \\ 100_2 &= 4_{10} \end{aligned}$$

Binary is special because it is the *minimum* integer base which can express exponentially many numbers. By having only *two* symbols, it is very naturally instantiated in physical systems, especially dichotomous ones. For example, by labelling the state of a lightswitch...

 $= 0_2$  $= 1_2$

we can count at the expense of mum's electricity bill.

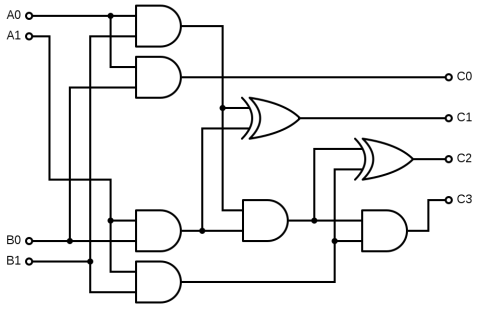


$$\begin{aligned} &= 10011_2 \\ &= 1 \times 2^0 \ + \ 1 \times 2^1 \ + \ 1 \times 2^4 \\ &= 19_{10} \end{aligned}$$

This is why the modern digital computer employs a binary encoding; it is easy to reliably produce and interpret two physically distinct states, like whether or not a memory cell contains a non-negligible electric charge. Binary numerals also allow us to repurpose the mathematical field of *Boolean algebra* in order to evaluate arithmetic expressions. The below electric circuit[1] uses electronic components which can perform logic gates...

 = AND  = XOR

in order to evaluate the product (with bits $[C3][C2][C1][C0]$) of two input binary numbers (with bits/digits $[A1][A0]$ and $[B1][B0]$).



Understanding that a computer is ultimately representing its numbers in binary allows a programmer to make use of all sorts of "*bit-twiddling tricks*".

$$\begin{aligned} (26 \times 2^3)_{10} &= 00011010_2 \ll 3_{10} \qquad (\ll \text{ means "left shift bits"}) \\ &= 11010000_2 \\ &= 208_{10} \end{aligned}$$

[1] This is a actually just an abstract *logic diagram* but is trivial to translate into an electric circuit due to convenient electrical phenomena like parallel currents.

2.4 Combinations

I gave late antiquity mathematicians a lot of credit for the positional notation whereby

$$\ldots[d][c][b][a]_B \;=\; a + b\,B + c\,B^2 + d\,B^3 + \ldots$$

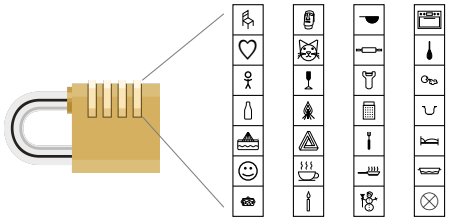
I will now demonstrate it is a natural notation admitted by combinatorics, and is much better than some alternatives I just made up like

$$8^{(4)}\ 5^{(1)}\ 2^{(0)} \;=\; 80052_{10}$$

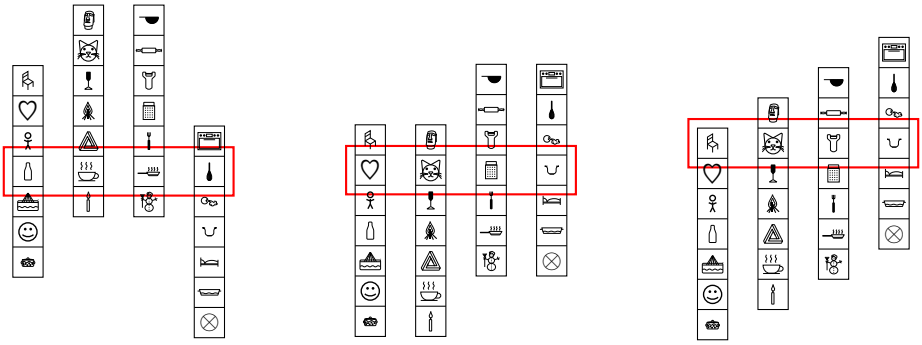
$${}^{4}_{2}3 \;= 324_{10}$$

$$\;=\; 308_{12} = 440_{10}$$

Consider a combination lock with *four* dials, each containing *seven* unique symbols.



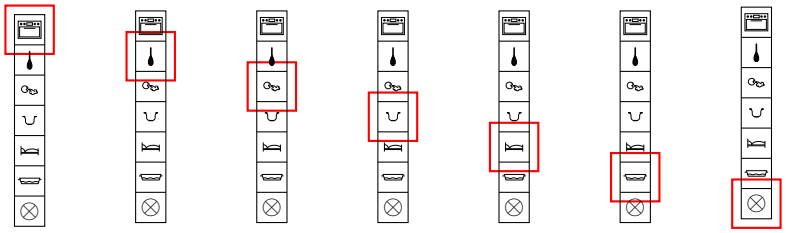
Imagine that this whimsical lock secures a stranger's bicycle which we intend to steal despite leaving our bolt cutters in our other trousers. So we test a few random combinations, hoping to guess the unlock sequence.



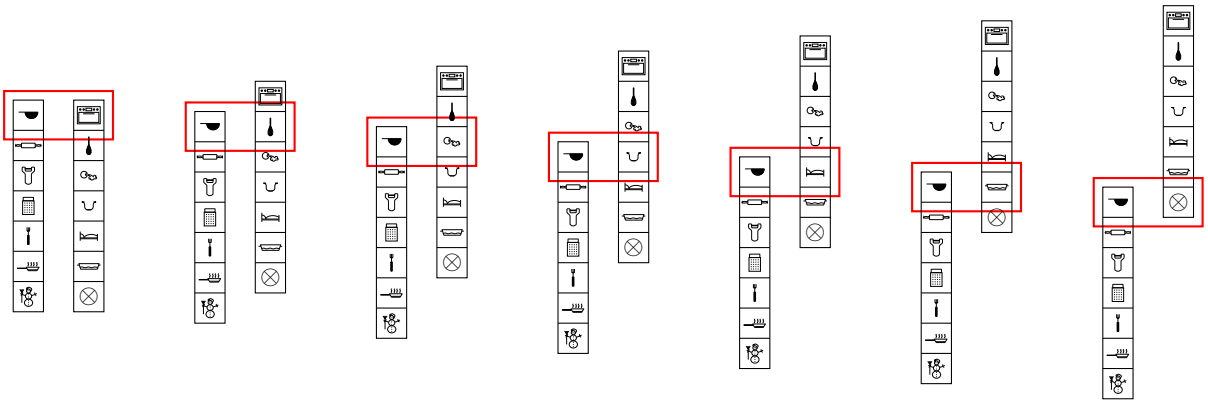
No luck: the lock remains fastened, and some onlookers glance suspiciously in our direction. Perhaps we can hurriedly try *all* combinations in succession before

the bobbies show up. Exactly how many combinations are possible? Let's work it out (*and quickly*).

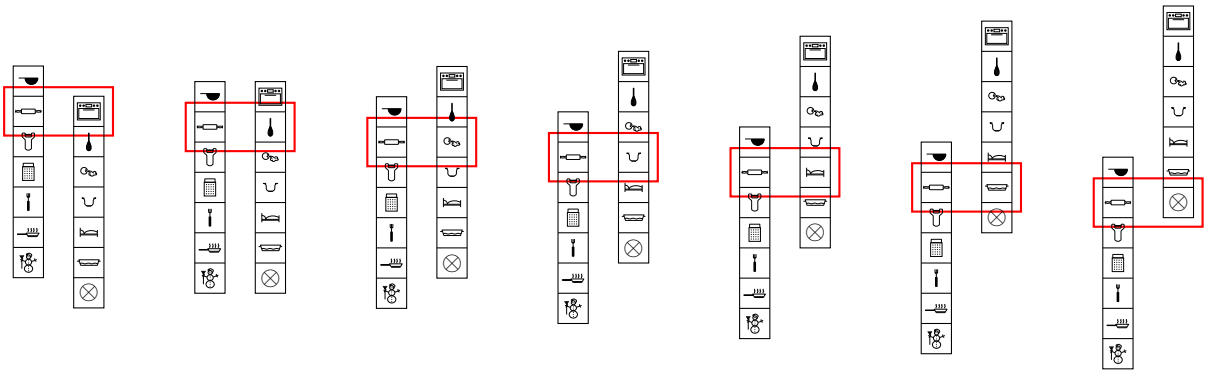
To try a combination, we must decide the position of each dial in-turn. Let us start with the rightmost dial. There are *seven* choices.



Now introduce the next adjacent dial. For each position of the rightmost dial, we can independently set the new adjacent dial to any of its seven positions. Start at its first value () while trying each of the seven rightmost dial values.

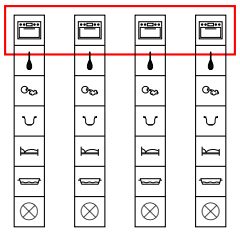


Increment the adjacent dial to , reset the rightmost dial to , and try again.

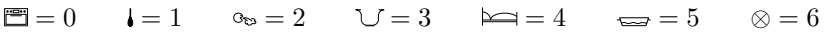


We can repeat this process for every of the seven values of the adjacent dial. There are hence $7 \times 7 = 49$ possible combinations of the two dials. Introducing yet another dial means each of these 49 two-dial combinations can be the suffix of a three-dial combination, where the new dial is set to any of its seven values. We continue, realising that the introduction of a new dial of $B = 7$ symbols increases the total number of combinations by a factor B. The full four-dial lock ergo has $B^4 = 2401$ possible combinations.

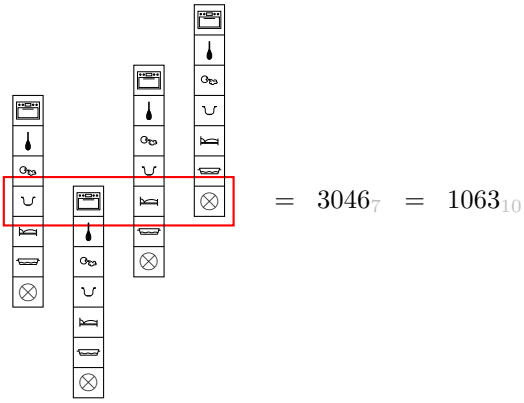
Trying one combination every second, this means it would take 40 minutes to systematically exhaust all possible combinations and yoink that beautiful bike. But the more important lesson here is the natural ordering of the tested combinations. Notice the actual symbols etched upon the lock dials were of no importance. They served only to distinguish the positions possible of an individual dial. Since the dials have a strict ordering that cannot be confused (we cannot accidentally physically swap two dials during our tampering), they may as well use the same symbol set:



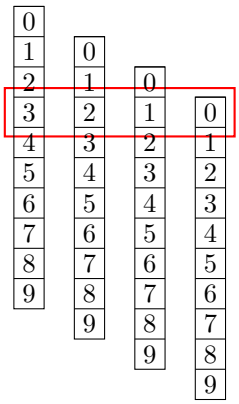
Nothing is changed. Yet, an analogy between these dials and numerals is forming. If we associate each symbol with its index on the dial...



then a given lock combination is merely a numeral in a base-7 number scheme!



The analogy is perfectly valid for our beloved base-10 numeral system.



$$= \quad 3210_{10}$$

We can now appreciate that the positional notation, whereby the nth digit (from the right, indexing from 0) of a base-B numeral has prefactor $B^n \ldots$

$$\begin{aligned} 0001_B &= B^0 \\ 0010_B &= B^1 \\ 0100_B &= B^2 \\ 1000_B &= B^3 \\ &\vdots \end{aligned}$$

is a very natural notation. This factor is the number of sub-combinations possible of the digits/dials to its right.

The analogy between dials of a combination lock and digits of a numeral is simple and apparent, but combination locks do not grow on trees and did not appear in Mesopotamia until the thirteenth century. Why am I making such a fuss about them? Because using combinatorics to denote numerals achieves "*closure*": every combination of digits you write down forms a valid numeral, interpretable as a valid number. For instance, select any number of the rightmost lock dial symbols (allowing duplicates) and write them down in any order. The result is always a valid numeral.

2.5 Extensions

There are many more ways to denote natural numbers, some better than those we saw above for certain situations.

- *Scientific notation* is useful when only the left-most digits of a large number are non-zero.

$$5 \times 10^9 = 5000000000$$

- *Tetration* is an instance of Donald Knuth's up-arrow notation, useful for denoting *huge* powers of a base.

$$B \uparrow\uparrow 4 = B^{B^{B^B}}$$

- *Factorial base* (also known as *factoradic numerals*) is (apparently) useful in combinatorics and cryptography, and replaces the nth digit value in positional notation from B^n to $(n+1)!$

$$\begin{aligned} 1 &= 1! \\ 10 &= 2! \\ 100 &= 3! \\ 1000 &= 4! \\ &\vdots \end{aligned}$$

Each digit is still restricted to be in 0-9, and a general numeral is evaluated as

$$\begin{aligned} \ldots[d][c][b][a]_! &= a \times 1! + b \times 2! + c \times 3! + d \times 4! + \ldots \\ &= a + 2\,b + 6\,c + 24\,d \end{aligned}$$

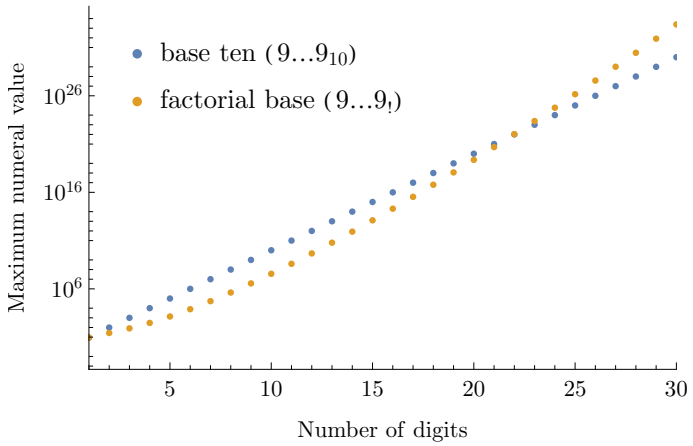
For example,

$$\begin{aligned} 244020_! &= 0 \times 1! + 2 \times 2! + 0 \times 3! + 4 \times 4! + 4 \times 5! + 2 \times 6! \\ &= 2020_{10} \end{aligned}$$

Given the same digits, do you think factorial-base or base-10 can denote a larger number? You may be tricked by looking at some examples...

$$\begin{aligned} 99_! &= 27_{10} \\ 9,999_! &= 297_{10} \\ 999,999,999_! &= 3,682,017_{10} \end{aligned}$$

It looks like base-10 is winning, because the maximum k-digit factorial-base numerals (for k above) require fewer than k digits when expressed in base-10. This may puzzle you if you subscribe to the doctrine that "factorials grow faster than exponentials", meaning $n!$ grows faster than B^n. Why aren't we seeing that? Because we are not looking at sufficiently big numbers! It turns out that for $B = 10$, the factorial becomes bigger than the exponential for $n \geq 25$.



Lo!

$$999,999,999,999,999,999,999,999,999_! = 5,827,302,643,226,110,604,462,817_{10}$$

- *Chinese numerals* avoid a big vulnerability in positional notation. Imagine that after a night of ~~heavy drinking~~ hydrating responsibly, you hand your friend an I Owe You note for \$45 and head to bed. They arrive the next morning to collect their debt, which has miraculously grown to \$45,000. It is no wonder you cannot remember spending so much of their money, they assert, since you drank ten thousand times the lethal limit of ~~vodka~~ cranberry juice! You repay your debt, not noticing your friend's hasty handwriting, and vow to never ~~drink~~ get carried away in the juice aisle again.

  The positional notation allows mischievous cretin to append additional digits to a written number in order to increase its value.

  $$\text{i o u} \quad \$45,000$$

  Fraud of this kind is rampant in ~~neoliberal hellscapes~~ developed countries with tipping cultures and paper bills. Chinese financial numerals avoid this - but I cannot understand why, so ask your Chinese friends!

3 Decimals

We now know how to denote any integer using a concise base-B positional numeral. For example

$$10111110111011110110100 1_2 \quad = \quad 12513129_{10} \quad = \quad \text{BEEF69}_{16}$$

But what about *decimal* or non-integer numbers? How does one write down the portion of two fish divided between 5000 starving pilgrims, or the modest hourly wage of a meek JMSS co-curricular teacher?

3.1 Dot point

We can introduce a *decimal separator* to the positional notation in order to denote rational numbers. The Scottish fancied the dot point

$$12 + 5 \div 10 = 12.5$$

while much of Europe sinned with an awful comma

$$= 12,5$$

The digits after the decimal separator correspond to *negative* powers of the base.

$$153.48_{10} = 1 \times 10^2 + 5 \times 10^1 + 3 \times 10^0 + 4 \times 10^{-1} + 8 \times 10^{-2}$$

These are the natural continuation of the powers. In fact, the role of the decimal separator is simply to denote which digit corresponds to the zero power.

$$\begin{array}{cccc} & 10^0 & 10^{-1} & \\ & \downarrow & \downarrow & \\ 3 & . & 7 & {}_{10} \end{array}$$

Of course one can think up all kinds of crappier alternatives.

$$39.01 \quad = \quad \overset{\overset{\text{zero power}}{\swarrow}}{3901} \quad = \quad 3901^{(-2)}$$

Negative powers after a decimal separator is not specific to base ten. The convention holds for a general base $B \in \mathbb{N}$.

$$\begin{aligned} 0.1_B &= B^{-1} \\ 0.01_B &= B^{-2} \\ 0.001_B &= B^{-3} \\ &\vdots \end{aligned}$$

A general base B decimal numeral denotes

$$\ldots [c][b][a].[\alpha][\beta][\gamma]_B \ldots = \cdots + c\,B^2 + b\,B^1 + a\,B^0 + \alpha\,B^{-1} + \beta\,B^{-2} + \gamma\,B^{-3} + \ldots$$

This is the recipe for interpreting decimal numerals that you have likely been unknowingly using for your entire life! Just like for integer numerals, all variables in this decimal numeral are restricted to $0 \leq a, b, c, \alpha, \beta, \gamma \leq B - 1$.

Let's consider the numbers we can denote using strictly three decimal digits in base B.

$$x_B = 0.[\alpha][\beta][\gamma]_B \qquad \alpha, \beta, \gamma \ \in \ [0..B)$$

Here are some important things to notice.

- The smallest (non-zero) number is $x = B^{-3}$, corresponding to $\alpha = \beta = 0$ and $\gamma = 1$.

$$0.[0][0][1]_B$$

- The next smallest number is $x = 2 \times B^{-3}$, achieved by incrementing the right-most digit. The right-most digit can be repeatedly incremented (adding value B^{-3}) until it reaches its maximum:

$$0.[0][0][B-1]_B \;=\; (B-1)B^{-3}.$$

  The next representable number...

$$\begin{aligned} 0.[0][1][0]_B &= B^{-2} \\ &= (B-1)B^{-3} + B^{-3} \end{aligned}$$

  is again just an increment of B^{-3}. The *gap* between each representable number is always B^{-3}, even when overflow occurs!

  Generally, the gap between each number representable by k decimal base-B digits is the smallest non-zero number, B^{-k}. This is consistent with the positional notation for integers in the previous chapter, where the smallest non-zero number was simply $B^0 = 1$.

- The biggest number is achieved when the digits are set to their maximum $\alpha = \beta = \gamma = B - 1$. In that case

$$\begin{aligned} \max\; x_B &= 0.[B-1][B-1][B-1] \\ \max\; x &= (B-1)B^{-1} + (B-1)B^{-2} + (B-1)B^{-3} \end{aligned}$$

  Once again, we do not need to evaluate this. We know that the *next* number (if we allowed the non-decimal digits to change) is simply

$$1.[0][0][0]_B \;=\; 1 \times B^0 \;= 1$$

  Since the gap between representable numbers is fixed at B^{-3}, we know

$$\max\; x \;=\; 1 - B^{-3}$$

3.2 Rounding error

How should a computer store a decimal number? We saw that binary (i.e. base $B = 2$ positional notation) worked well for storing integers. Using k binary digits ("*bits*"), a computer can represent all integers between 0 and $2^k - 1$ (inclusive). We could simply slap on another set of m bits to store the negative-power digits.

Let's build the world's worst accounting computer to process my finances. We will dedicate $k = m = 4$ bits to represent a single decimal number, where the 4 rightmost bits correspond to the negative powers of two.

$$\boxed{0}\boxed{0}\boxed{0}\boxed{0}\,\boxed{0}\boxed{0}\boxed{0}\boxed{0}_2 = 0.0_{10}$$

The bits are ordered in the usual positional notation.

$$0010\ 0101_2 \ = \ 2^0 + 2^{-2} + 2^{-4} \ = \ 2.3125_{10}$$

The smallest non-zero number is $B^{-m} = 2^{-4}$.

$$0000\ 0001_2 \ = \ 0.0625_{10}$$

That's a pretty pathetic minimum but we'll continue. The next smallest representable number is

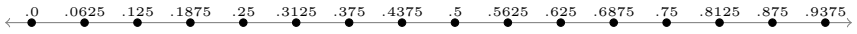
$$0000\ 0010_2 \ = \ 0.125_{10}$$

How would this computer store my total teaching earnings, about $12.10?

It simply cannot! There is no assignment of the 8 bits which will encode the number 12.10_{10}. The gap between representable decimals (the "*machine epsilon*") is $\epsilon = B^{-m} = 0.0625_{10}$. The closest number to my true earnings that the computer can store is

$$1100\ 0010_2 \ = \ 12.125_{10}$$

Just through trying to store the number, the computer will overestimate my total earnings by 2.5 cents. This is called *rounding error*, since the true number was rounded to the nearest representable number. The crummy computer can only store the below decimal numbers (integer factors of ϵ):



An error smaller than ϵ may not seem a huge problem; 2.5 cents won't even buy my daily lunch of a single gummy bear, especially not after the inflation of World War 3. But when our computer performs *calculations* on these 8 bit decimals, things will go downhill faster than the credibility of the Putin government.

An arithmetic calculation in our computer's CPU will take in a binary decimal, and output a binary decimal. We may ask it to compute

$$2_{10} \ \div \ 10_{10}$$

which gets encoded as

$$0010\ 0000_2 \ \div \ 1010\ 0000_2$$

As regrettably intelligent humans, we know the answer is 0.2_{10}. But this number cannot be represented in our binary scheme, so the CPU rounds it to the closest binary number and proudly outputs:

$$0000\ 0011_2 \ = \ 0.1875_{10}$$

The maximum error of this single calculation is always bounded by the machine epsilon $\epsilon = B^{-m} = 2^{-4} = 0.0625_{10}$, because this is the gap between representable numbers. But now imagine that the output is being fed into another

calculation, like multiplication with the number $8.9375_{10} = 1000\ 1111_2$. The *correct* full calculation is

$$(2_{10} \div 10_{10}) \times 8.9375_{10} = 1.7875_{10}.$$

The closest representable binary number to this happens to be

$$0001\ 1101_2 = 1.8125_{10}$$

But our lousy computer won't even output that! It will perform

$$\begin{aligned} x &= (0010\ 0000_2 \div 1010\ 0000_2) \times 1000\ 1111_2 \\ &= (0000\ 0011_2) \times 1000\ 1111_2 && \text{(rounded)} \\ &= 0000\ 1011_2 && \text{(rounded)} \\ &= 1.6875_{10} \end{aligned}$$

The error between its output and the true value is 0.1_{10} which *exceeds* the machine epsilon ϵ. This is because each intermediate step of the calculation must be stored as a binary number and ergo experience rounding error. So the computer actually multiplied number 0.1875_{10} (instead of 0.2_{10}) with 8.9375_{10}, which *should* yield erroneous result 1.67578_{10}, but this is finally rounded to binary-representable number 1.6875_{10}. Sequences of arithmetic operations *compound* the rounding error. The longer the calculation, the progressively more inaccurate the output becomes!

*Holy smokes, computers **SUCK**! Everybody **PANIC**!*

Woah there partner. We were imagining a lousy machine using only $m = 4$ decimal bits. The machine epsilon vanishes exponentially quickly as $\epsilon = 2^{-m}$. *Your* computer running Python likely uses $m = 64$ decimal bits, so it's epsilon is

$$\epsilon = 2^{-64} \approx 5.42101 \times 10^{-20}$$

That's so tiny that we probably do not even see the effects of rounding errors in our daily uses. And surely not with simple numbers like 0.1_{10} and 0.3_{10} even though they do not have exact binary representations, right? Let's reassure ourselves and ask Python on your computer to evaluate

$$x = 0.1_{10} + 0.1_{10} + 0.1_{10} - 0.3_{10}$$

It outputs:

$$5.551115123125783 \times 10^{-17}$$

Your computer believes $x \neq 0$. A belief like that can easily blow up a spacecraft full of astronauts.

*Holy smokes, computers definitely DO **SUCK**! Everybody **PANIC**!*

Let's not be *too* critical though, since you yourself have rounding error. You have likely before committed:

$$2_{10} \div 3_{10} \approx 0.666666667_{10}$$

It just so happens that 2/3 cannot be expressed in a finite base ten numeral. *Any* choice of base B will still restrict a k digit decimal to a finite set of numbers (B^k of them), despite the existence of uncountably infinitely many real numbers in between. In fact, you cannot write down the *ridiculously extreme majority* of all existing numbers. One should endeavour to live their life peacefully despite this.

3.3 Floating point

Your computer is at least doing something a bit more clever than the binary decimal system I described above. Our imagined 8-bit computer with 4 decimal bits could only store numbers as small as 2^{-4}, but nothing smaller (besides zero).

$$0000\ 0001_2 = 0.0625_{10}$$

If this computer was performing a series of arithmetic calculations, and an intermediate operation produced a number smaller than half of the above minimum number, it would round to zero. This would make subsequent multiplication with any other number zero, even if the other number is *huge*. As such, the whole calculation would likely spit out an absurdly incorrect number. Having such an unimpressive minimum decimal number makes our machine almost useless. To support smaller numbers, we would need to add more decimal bits.

Your actual computer probably represents numbers using 64 bits. If all of them were decimal bits then our binary decimal system could store a number as small as

$$000\ldots0001_2 = 2^{-64} \approx 5.4 \times 10^{-20}$$

That's certainly *small*, but nowhere near as small as the Planck constant

$$\hbar \approx 1.05 \times 10^{-34}$$

which quantum physicists have no problem loudly multiplying away in their MacBooks at the back of Starbucks. How has Apple achieved this feat? How can a computer store a number smaller than 2^{-64} while using no more than 64 bits? Or in other words, how can one denote a decimal as small as 2^{-110} using fewer than 110 bits?

The same way I just did! `:^)`
Scientific notation allows us to avoid nonsense like this:

$$m_e \approx 0.000000000000000000000000000167_{10} = 1.67 \times 10^{-27}$$

or like this:

$$N_A \approx 602200000000000000000000.0_{10} = 6.022 \times 10^{23}$$

We do not waste precious digits on long prefixes or suffixes of zeros; we instead *shift* the non-zero digits right up to the left and denote the size of the shift as an exponent of the base. This compactifying trick works in any base.

$$\begin{aligned} 0.0000001_B &= 1 \times B^{-7} \\ 1FA0000000.0_{16} &= (1.FA)_{16} \times (16^9)_{10} \\ 101000000_2 &= (1.01)_2 \times (2^8)_{10} \end{aligned}$$

If we are working with a fixed specific base which we do not need to write down, then all identifying information of a number in scientific notation is captured in a handful of digits. The mass of an electron in kilograms (in base ten) is:

$$\begin{aligned} \text{mantissa}: &\ 1.67 \\ \text{exponent}: &\ -27 \end{aligned}$$

This is how your computer is encoding decimal numbers in binary. It is using base-two scientific notation to represent the electron's mass as

$$m_e \approx 1.615_{10} \times 2^{-89}$$

and is storing it as *two* bit sequences.

$$\begin{aligned} \text{mantissa}: &\ 1.10011101_2 && (= 1.613281_{10}) \\ \text{exponent}: &\ -1011001_2 && (= -89_{10}) \end{aligned}$$

This encoding is called "*floating point*" because changing the stored exponent has the effect of moving the decimal separator (the dot point) in the binary numeral of the stored number. The dot point is "floating":

$$1.011 \times 2^3 = 10.11 \times 2^2 = 101.1 \times 2^1$$

By convention however, the exponent is always chosen such that the dot point comes after the *first* one-bit, like in the left example. This is called "normalisation".

$$\begin{aligned} \text{mantissa}: &\ 0.1100_2; \\ \text{exponent}: &\ -100_2 \end{aligned}$$

$$\begin{aligned} \text{mantissa}: &\ 1.1000_2; \\ \text{exponent}: &\ -101_2 \end{aligned}$$

Normalisation means the first bit of the mantissa is not actually stored, because it is *always* implicitly 1. We get to use that extra bit on the decimal digits, yay! The mass of an electron can then be stored as in as few as 17 bits! (including the signs + and −)

$$\begin{aligned} \text{mantissa}: &\quad +\,1.10011101_2 \\ \text{exponent}: &\quad -\,1011001_2 \end{aligned}$$

Your computer is likely using the "IEEE 754 double-precision" 64-bit convention where the number of bits in the mantissa is fixed at 52 (with an extra bit for its sign, 0 = + and 1 = −), and the exponent uses 10 bits (also with an extra bit for its sign[2]). The bit sequences are plopped right next to each other, like so:

Here is how our approximation to the electron mass would appear:

0 1 0001011001 10011101000000000000000000000000000000000000000000000000

Let's consider how to encode some specific numbers, when we use m bits for the mantissa, and p bits for the exponent, excluding the signs.

- What is the *largest* number our computer can store? It has the form

$$\max\ x = +\,(\text{largest mantissa}) \times 2^{+(\text{largest exponent})}$$

  We are by now quite practised at this. The mantissa has an implicit 1.0 followed by m decimal digits all with value 1. Meanwhile the exponent is the maximum integer of p digits.

$$\begin{aligned} \max\ x &= (1.11\cdots111_2) \times 2^{(111\cdots111_2)} \\ &= \left(1. \ +\ 1 - 2^{-m}\right) \times 2^{(2^p-1)} \\ &= \left(2 - 2^{-m}\right) \times 2^{(2^p-1)} \end{aligned}$$

  For $m = 52$ and $p = 10$, this is about

$$\max\ x \approx 1.8 \times 10^{308}$$

  or a *Googol cubed*. Quite large; imagine every atom in our universe was duplicated into its own entire universe. Then the same process happened between all those combined atoms twice more. The unimaginable collection of atoms in the final ensemble of universes is about as large as this number.

  You can chuck such numbers into Python (with syntax `2.4E6` = 2.4×10^6) to verify my claims.

[2] This is a lie to defer some nuance. What a nuisance those computer scientists have made to my document structure.

```
>>> 1.13E308 + 0.2E308
1.33e+308

>>> 1.13E308 + 0.7E308
inf
```

- What is the *smallest* number (non-zero and positive) that our computer can store? It has the form

$$\begin{aligned}\min\ x &= +\left(\text{smallest mantissa}\right) \times 2^{-(\text{largest exponent})} \\ &= +\left(1.000\cdots00_2\right) \times 2^{-(111\cdots111_2)} \\ &= 2^{-(2^p-1)}\end{aligned}$$

  The mantissa bits can all be set to zero, but the interpreted number is still prefixed with an implicit 1, as per normalisation.

  For $m = 52$ and $p = 10$, this is about

$$\min\ x \approx 1.1 \times 10^{-308}$$

  This is indeed what Python claims to us:

```
>>> import sys
>>> sys.float_info
sys.float_info(..., min=2.2250738585072014e-308, ...)
```

  However Python can actually squeeze even *smaller* numbers out through "*denormalisation*". It simply relaxes the requirement and assumption that the first mantissa digit is *one*, allowing it to be *zero*. The minimum number in this *ad-hoc* encoding is

$$\begin{aligned}\min_{\text{denorm}}\ x &= +\left(\text{smallest unnormalised mantissa}\right) \times 2^{-(\text{largest exponent})} \\ &= +\left(0.000\cdots01_2\right) \times 2^{-(111\cdots111_2)} \\ &= \left(2^{-m}\right) \times 2^{-(2^p-1)}\end{aligned}$$

  Now we can play with "*subnormal numbers*" as small (substituting $m = 52$ and $p = 10$) as

$$\min_{\text{denorm}}\ x\ \approx\ 2.5 \times 10^{-324}$$

  though I pray you never have to.

```
>>> 3E-324
5e-324

>>> 2E-324
0.0
```

- How *many* unique floating-point numbers are representable? Since a number is ultimately encoded using 64 bits, we can immediately upperbound it by the number of unique 64-bit sequences, i.e.

$$\leq 2^{64}$$

Are we counting any particular numbers twice?

The astute reader, having survived all that, may find themselves quite confused. Something bugs them. They may ask: *How can a normalised floating-point system, wherein the first digit of the mantissa is always **one**, represent the simplest number of them all; **zero**? How does it encode* 0.0_{10}? Dear reader, it cannot. All this time I have tortured you, and I must apologise. My sinister personality has caused me to lie about an important nuance of the IEEE 754 double-precision standard, which I now remedy.

The exponent doesn't really dedicate a bit to its sign. Instead, it offsets its 11-bit values by 2^{10} so that there is a unique zero encoded by sequence 01111111111_2. Anything less than this is negative, and larger than this is positive. This permits 2^{11} unique exponents. *However*, the extreme values 00000000000_2 and 11111111111_2 are reserved to encode special values.

When the exponent is zero and the mantissa is zero, the encoded floating-point number is zero.

$$0.0_{10} = 0\ 00000000000\ 0000000000000000000000000000000000000000000000000000$$

The mantissa is still signed, so there *is* a negative zero too. Neat huh?

$$-0.0_{10} = 1\ 00000000000\ 0000000000000000000000000000000000000000000000000000$$

```
>>> - 0.0
-0.0
```

When the exponent is zero but the mantissa digits are *not* all zero, the encoded number is subnormal. The exponent implicitly becomes its minimum (-2^{10}) and the mantissa will be assumed unnormalised (the first digit will not be assumed 1). This was Python's trick for handling numbers smaller than 10^{-308}.

$$(-0.13\times10^{-323})_{10} \approx -2^{-2^{10}-52} = 1\ 00000000000\ 0000000000000000000000000000000000000000000000000001$$

Finally, the other special value of the exponent encodes the whacky family of invalid numbers, like infinity and `NaN`.

$$\texttt{inf} = 0\ 11111111111\ 0000000000000000000000000000000000000000000000000000$$

$$\text{NaN} = 0\ 11111111111\ 1100001000000010000011000000111000000000100001000001$$

Yes indeed, this means there are 2^{53} (four quadrillion) unique `NaN` types. Be afraid programmers, be very afraid.

3.4 Extensions

We went a little off the rails about computers, but this chapter was really about ways to write down decimal numerals. That is, how to leverage the great success of the positional notation system for denoting integers, in order to denote fractional quantities.

There are many more ways humans have devised to write down decimals, and you've likely already met a few.

- *Repeated numerals* have all sorts of disputed syntax.

$$\frac{1}{3} \;=\; 0.\dot{3} \;=\; 0.\overline{3} \;=\; 0.333\ldots \;\approx 0.333333$$

  They allows rational numbers with infinite but periodic decimal digits to be compactly notated.

$$\frac{1}{11} \;=\; 0.\dot{0}\dot{9} \;=\; 0.\overline{09} \;\approx 0.090909$$
$$\frac{3}{7} \;=\; 0.\dot{4}28571\dot{} \;=\; 0.\overline{428571} \;\approx 0.428571428571428571$$

- *Mixed numerals* are composed of an integer and an adjacent proper fraction.

$$9\tfrac{3}{4}$$

  They can represent any rational number concisely and finitely, but arithmetic becomes a bit of a ballache.

- *Complex numbers* $\mathbb{C}$ are a whole new set of numbers beyond the scope of this already hyper-distracted document. Those familiar may be interested in Knuth's *Quater-imaginary base* system, which is simply our beloved positional notation with base $B = 2\mathrm{i}$, and digits 0, 1, 2 and 3.

$$\begin{aligned} 302.11_{2\mathrm{i}} &= 3 \cdot (2\,\mathrm{i})^2 + 0 \cdot (2\,\mathrm{i})^1 + 2 \cdot (2\,\mathrm{i})^0 + 1 \cdot (2\,\mathrm{i})^{-1} + 1 \cdot (2\,\mathrm{i})^{-2} \\ &= -10.25_{10} - 0.5_{10}\,\mathrm{i} \end{aligned}$$

  Please direct all questions about practical applications to Donald Knuth.

4 Review questions

Got all that? Try these questions

1. What is 125032222_7 when denoted in base 10?
2. How many digits would Avagadro's constant have if written in base 6? Hint: you don't need to determine the numeral.
3. What is the result of
$$45A9F12_{16} \times (16^4)_{10}$$
when expressed in Hexadecimal?
4. An alien species uses the below symbols in their positional numeral system:

Their freaky alien telephone numbers are 5 digits long, and can contain any of these symbols in any order. You wish to call your alien friend Sxlahmdog but have forgotten their number, so you dial 5 random symbols. What are the chances you dialled correctly?

5. Your call to Sxlahmdog has earned you a telephone bill for

 (AUD)

You have no idea of the integer value of each symbol. What is the *maximum* possible value (in base 10) of this fee?

6. Consider a base-B decimal number composed of k digits (no scientific notation). The decimal separator occurs after the second left-most digit. What is the maximum value of this number?
7. Under the IEEE 754 double-precision standard (albeit with an explicit sign bit for the exponent, given below in orange), what number (in base ten) does the below bit sequence encode?

1 1 0011001100 11000000000000000000000000000000000000000000000000000011

The solutions to these questions are at the bottom of this document. But have a genuine attempt at before you look!

5 Appendices

- A "*set*" is a collection of numbers. Here is the set of my favourites:

$$S = \{42,\ 101,\ 666,\ 42069\}$$

Notation $42 \in S$ means "42 is an element of set S".
Notation $7 \notin S$ means "7 is not an element of set S".
(get that stanky 7 outta here)

- The "*natural numbers*" ($\mathbb{N}$) is the set of positive whole numbers including zero.

$$\mathbb{N} = \{0,\ 1,\ 2,\ 3,\ 4,\ \dots\}$$

Some pedants insist this set should exclude zero. Please direct them to any of notations

$$\mathbb{N}^+ = \mathbb{N}^* = \mathbb{N}_{>0} = \mathbb{N}_{\neq 0} = \mathbb{N}_1 = \mathbb{N} \setminus \{0\}$$

- The "*integers*" ($\mathbb{Z}$) contain all the natural numbers, and their negative forms.

$$\mathbb{Z} = \{\dots -4,\ -3,\ -2,\ -1,\quad 0,\quad 1,\quad 2,\quad 3,\quad 4,\ \dots\}$$

- The "*reals*" ($\mathbb{R}$) contain every number you have ever seen[3]. It contains all the integers (like 2 and 3), and every non-whole number in between them (like 2.151942069).

$$2 \in \mathbb{R}$$
$$2.151942069 \in \mathbb{R}$$
$$\pi \in \mathbb{R}$$

- The "*rationals*" ($\mathbb{Q}$) contain only the real numbers which can be expressed as fractions of integers:

$$\frac{a}{b} \quad \text{where} \quad a, b \in \mathbb{Z}$$

This obviously includes all integers.

$$-3 = \frac{-3}{1}$$

Anything you write down as a finite decimal can always be expressed as such a fraction.

$$4.7513 = \frac{47513}{10000} \in \mathbb{Q}$$

[3] (until taking your first class in complex numbers)

In fact, even infinite but *repeating* decimals (like $5.6666666\ldots$ or $8.123123123\ldots$) are always rational. "*Irrational*" numbers are simply those not included in the rational set, like $\sqrt{2} \notin \mathbb{Q}$. Fans of irrationals must go elsewhere.

$$\pi, e, \phi \quad \notin \mathbb{Q}$$

- "*Exponentiation*" or "*raising a number to the power of another*" means repeated multiplication with itself.

$$a^3 \;=\; a \times a \times a$$

The exponent doesn't need to be positive:

$$a^{-3} \;=\; \frac{1}{a^3} \;=\; \frac{1}{a \times a \times a}$$

The exponent doesn't even need to be an integer...

$$4^{-\pi} \approx 0.0128402$$

... but its definition is a closely guarded secret[4].

An "*exponential*" refers to a function of a variable which is an exponent.

$$f(x) = 7^x$$

- The "*logarithm*" is the inverse function of the exponential. It can be simply understood as the function which, when given values a and b related by

$$a = b^c$$

will return the unknown power c

$$c = \log_b(a)$$

- The "*factorial*" of a natural number is the product of itself and all smaller natural numbers (excluding zero)

$$\begin{aligned} 5! \;&=\; 5 \times 4 \times 3 \times 2 \times 1 \\ n! \;&=\; n \times (n-1) \times (n-2) \times \cdots \times 3 \times 2 \times 1 \end{aligned}$$

We also define

$$0! = 1$$

just because ¯_(ツ)_/¯

[4] (until your first class in calculus)

Factorials grow *very fast* meaning that $x!$ (for a big integer $x \in \mathbb{N}$) will typically be much bigger than a polynomial (like $2\,x^3$) or an exponential (like 4^x). For example, when $x = 30$,

$$\begin{aligned} 2\,x^3 &= 54,000 \\ 4^x &= 1,152,921,504,606,846,976 \\ x! &= 265,252,859,812,191,058,636,308,480,000,000 \end{aligned}$$

- A "*real interval*" is a subset of the real set written as

$$[a, b] \quad \subset \mathbb{R}$$

and contains all numbers between $a \in \mathbb{R}$ and $b \in \mathbb{R}$ (inclusive). For example

$$\begin{aligned} 5.2 &\in [4, 6.5] \\ 3 &\notin [4, 6.5] \end{aligned}$$

Round brackets indicate that end number is not included in the subset, but every real number right up to it remains included.

$$\begin{aligned} 6.4 &\in [4, 6.5) \\ 6.4999999 &\in [4, 6.5) \\ 6.5 &\notin [4, 6.5) \\ 4 &\notin (4, 6.5) \end{aligned}$$

- An "*integer interval*" is a subset of the integers

$$[a..b] \;=\; \{a,\ a+1,\ a+2,\ a+3,\ \ldots,\ b-2,\ b-1,\ b\}$$

where $a, b \in \mathbb{Z}$ (and assuming b is the larger integer). For example

$$\begin{aligned} [3..8] &= \{3,\ 4,\ 5,\ 6,\ 7,\ 8\} \\ [3..8) &= \{3,\ 4,\ 5,\ 6,\ 7\} \\ (3..8) &= \{4,\ 5,\ 6,\ 7\} \end{aligned}$$

This notation is less ubiquitous than that for real intervals, so is sometimes instead expressed as an intersection of a real interval and the integers.

$$[a..b) \;=\; [a, b) \cup \mathbb{N}$$

6 Review answers

1.

$$\begin{aligned} 125032222_{7} &= 1 \cdot 7^8 + 2 \cdot 7^7 + 5 \cdot 7^6 + 0 \cdot 7^5 + 3 \cdot 7^4 + 2 \cdot 7^3 2 \cdot 7^2 + 2 \cdot 7^1 + 2 \cdot 7^0 \\ &= 8008135_{10} \end{aligned}$$

2.

$$\begin{aligned} \text{number of base 6 digits} &= 1 + \lfloor \log_6(n_A) \rfloor \\ &= 1 + \lfloor \frac{\log_{10}(n_A)}{\log_{10}(6)} \rfloor \\ &= 1 + \lfloor \frac{\log_{10}(6.022 \times 10^{23})}{\log_{10}(6)} \rfloor \\ &= 31 \end{aligned}$$

3.

$$45A9F12_{16} \times (16^4)_{10} = 45A9F120000_{16}$$

because multiplication with a power of the base merely shifts the digits by the power.

4. The aliens use 18 unique numeral symbols, implying a base $B = 18$ system. The number of unique $k = 5$ digit numerals (i.e. phone numbers) is then

$$B^k = 18^5 = 1889568_{10}$$

The odds of calling the correct number are thus

$$\frac{1}{1889568} \approx 5.3 \times 10^{-7}$$

5. The great potential value of numeral

 $_{18}$

assumes the five unique symbols above are the five largest value symbols, ordered in decreasing value. I.e.

 $_{18} = 17_{10}$

 $_{18} = 16_{10}$

 $_{18} = 15_{10}$

 $_{18} = 14_{10}$

 $_{18} = 13_{10}$

The number then encodes number

 $_{18} = 17 \cdot 18^5 + 16 \cdot 18^4 + 16 \cdot 18^3 + 15 \cdot 18^2 + 14 \cdot 18^1 + 13 \cdot 18^0$

$= 33,900,709_{10}$

6. The maximum k-digit base-B numeral, where the decimal seperator occurs after the second left-most digit, has value:

$$[B-1][B-1].[B-1][B-1][B-1]\cdots[B-1]_B$$

where there are $k-2$ digits after the dot point. This means that the epsilon, or the gap between k-digit numbers of this form, is

$$\epsilon = B^{k-2}.$$

The *next* numeral from that above is $k+1$ digits:

$$[1][0][0].[0][0][0]\cdots[0]_B$$

and has value B^2. Hence, the original numeral encodes value:

$$B^2 - B^{k-2}$$

7.

$$\begin{aligned}
&1\ 1\ 0011001100\ \ 11000000000000000000000000000000000000000000000000000011\\
&= -1.11000000000000000000000000000000000000000000000000000011_2 \times 2^{-0011001100_2}\\
&= -\left(2^0 + 2^{-1} + 2^{-2} + 2^{-51} + 2^{-52}\right) \times 2^{-\left(2^2+2^3+2^6+2^7\right)}\\
&\approx \left(-6.80642 \times 10^{-62}\right)_{10}
\end{aligned}$$