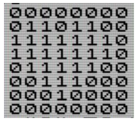
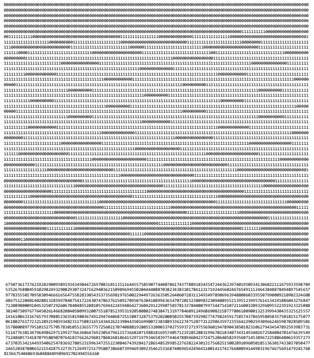
More *bits* of CS

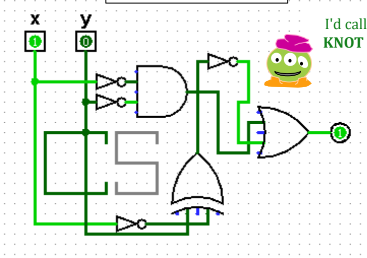




Too many bits? Compress!

Below binary: ***physical circuits***

Circuit design, part 1



I'd call this a **KNOT** gate...

Hw #4

pr0 (reading) A bug and a crash!

pr1 (lab) binary ~ decimal

pr2 conversion + compression

extra image processing...

Lots of tutoring hrs - join in... !

P/F vs T/F

Reasoning, *bit by bit*

<<	>>	&	|
left-shift	right-shift	and	or

and (both) **&** **|** or (either)

bitwise and

```
5:  101
6:  110
&   100
```

```
5 & 6
4
```

bitwise and

```
11: 1011
 5: 0101
&
```

```
11 & 5
```

bitwise or

```
5:  101
6:  110
|   111
```

```
5 | 6
7
```

bitwise or

```
11: 1011
 5: 0101
|
```

```
11 | 5
```

At its base ...

(case)

in

42

```
ntb( 42 )
ntb( 21 ) + '0'
ntb( 10 ) + '1'
ntb( 5 ) + '0'
ntb( 2 ) + '1'
ntb( 1 ) + '0'
ntb( 0 ) + '1'
''
```

out

'101010'

```
def numToBin( N      ):
    """ converts a decimal int to a binary string
    """
    if N==0:      return ''
    else:         return numToBin(       ) + str(       )
```

What if you wanted base-3 output?! ***base-B output?***

At its base ...

(case)

in

'101010'

```
btn('101010')
2*btn('10101') + 0
2*btn('1010') + 1
2*btn('101') + 0
2*btn('10') + 1
2*btn('1') + 0
2*btn('') + 1
0
```

2*1

2*2*2*1

2*2*2*2*2*1

out

42

```
def binToNum( S     ):
    """ converts a binary string to a decimal int
    """
    if S=='':  return 0
    else:      return 2*binToNum(       ) + int(        )
```

saves the need for another `if`

What if you wanted base-3 input?! ***base-B input?***

Bits & Base-2

value	representation
	64 32 16 8 4 2 1
0	0
1	1
2	10
5	101
10	1010
15	1111
16	10000
21	10101
42	101010
127	1111111
	64 32 16 8 4 2 1

What are 3 and 4 in base-2?

7?
8?

What does *shifting right* do? >> 1

What's **1000** ?

What does *shifting left* do? << 1

What's **101000** ?

What's the max value **4** bits can hold?

What's the max value **7** bits can hold?

What's the max value **N** bits can hold?

Extra: How could bits represent *negative* numbers? *Fractions?*

How *far* can we count…?

I can see some patterns here – even *with one eye closed!*

with		max count
1 bit	1	1
2 bits	11	3
3 bits	111	7
4 bits	1111	15
7 bits	1111111	127
8 bits	11111111	255
N bits	1111...1111	

8-bit **signed** values
-128 to 127

8-bit **unsigned** values
0 to 255

How could bits represent ***negative*** numbers?

15 bits

31 bits

Fractions?

How *far* can we count…?

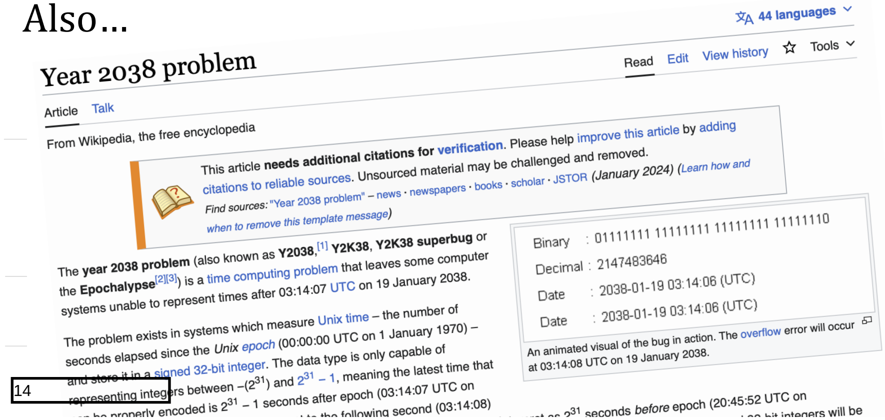
in 2015?

List of most viewed YouTube videos

From Wikipedia, the free encyclopedia

in 2022?

Also…



Year 2038 problem

From Wikipedia, the free encyclopedia

This article **needs additional citations for verification**. Please help improve this article by adding citations to reliable sources. Unsourced material may be challenged and removed.
Find sources: "Year 2038 problem" – news · newspapers · books · scholar · JSTOR *(January 2024)* (*Learn how and when to remove this template message*)

The **year 2038 problem** (also known as **Y2038**,[1] **Y2K38**, **Y2K38 superbug** or the **Epochalypse**[2][3]) is a time computing problem that leaves some computer systems unable to represent times after 03:14:07 UTC on 19 January 2038.

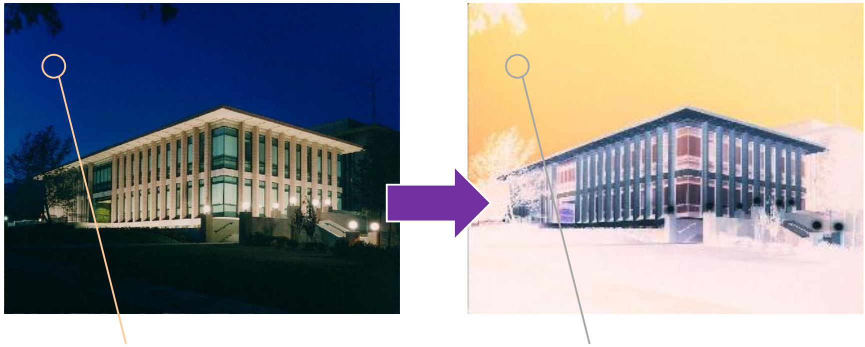
The problem exists in systems which measure Unix time – the number of seconds elapsed since the *Unix epoch* (00:00:00 UTC on 1 January 1970) – and store it in a signed 32-bit integer. The data type is only capable of representing integers between $-(2^{31})$ and $2^{31} - 1$, meaning the latest time that can be properly encoded is $2^{31} - 1$ seconds after epoch (03:14:07 UTC on

```
Binary  : 01111111 11111111 11111111 11111110
Decimal : 2147483646
Date    : 2038-01-19 03:14:06 (UTC)
Date    : 2038-01-19 03:14:06 (UTC)
```

An animated visual of the bug in action. The overflow error will occur at 03:14:08 UTC on 19 January 2038.

Hw4: *images are just bits, too!*

hw4pr3 *(extra)*



<u>old</u> pixel at 42,42 has

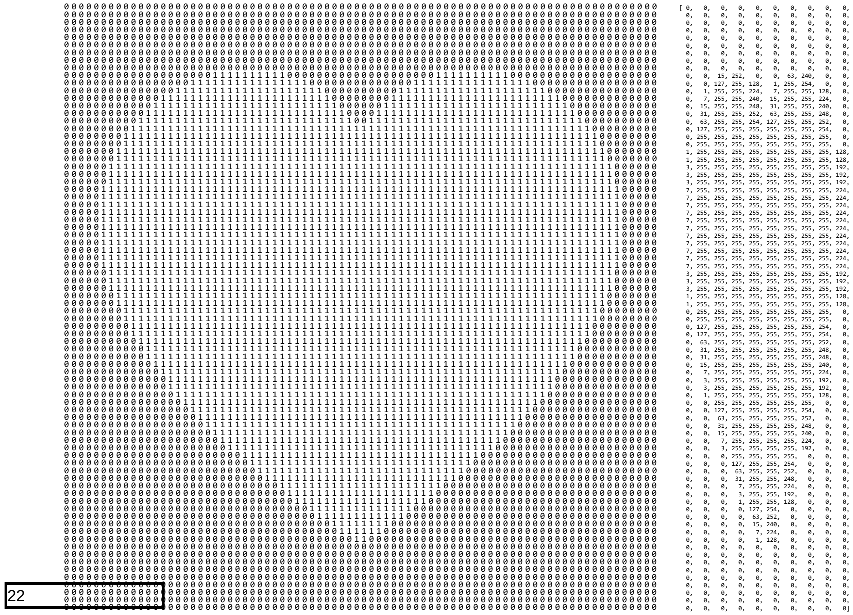
red = 1	(out of 255)
green = 36	(out of 255)
blue = 117	(out of 255)

<u>new</u> pixel at 42,42 has

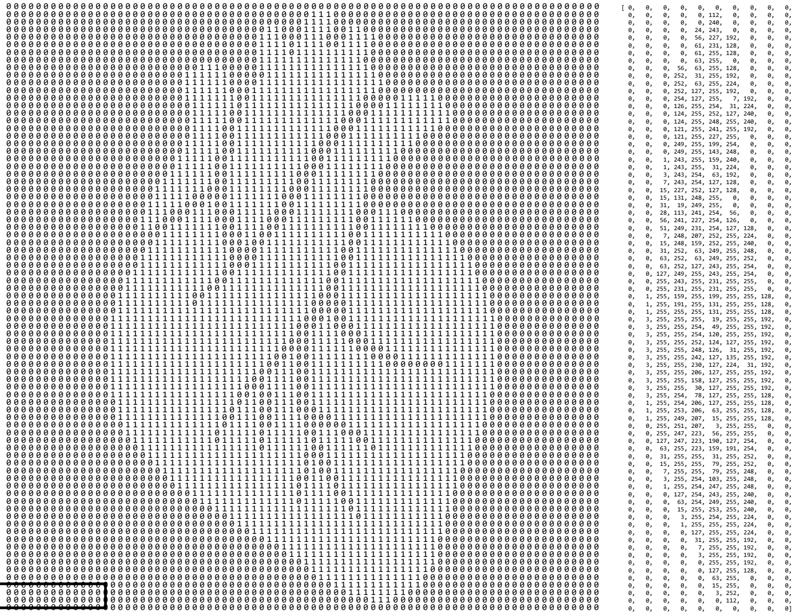
any guesses as to what this transformation was?

how many <u>bits</u> are being used for each color channel?

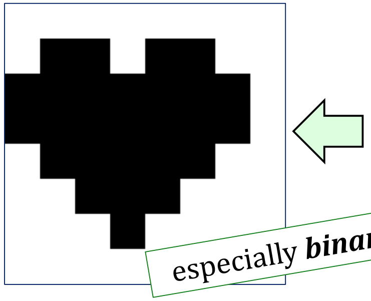
Images are Just Bits!



Images are Just Bits!



Hw4: *images are just bits, too*!



especially ***binary*** images

00000000
01101100
11111110
11111110
01111100
00111000
00010000
00000000

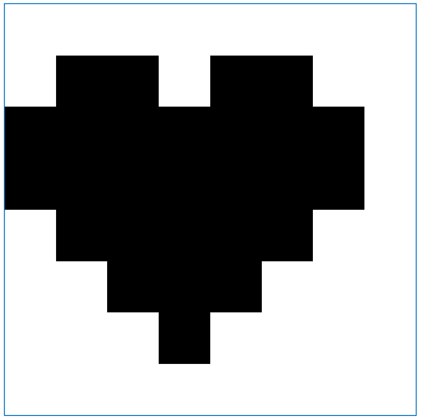
Binary Image

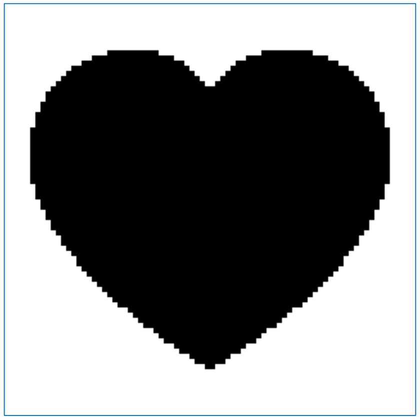
Encoding as raw bits

one big string of 64 characters

```
"0000000001101100111111101111111001111100001110000001000000000000"
```

What's the difference?





Lossy *compression vs.* ***Lossless***

Where in the World?

*Think of three situations where **Lossy** compression is better.*

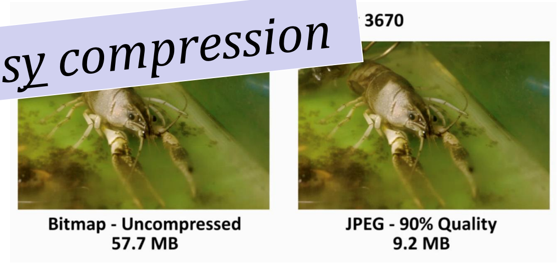
*Think of three situations where **Lossless** compression is better.*

Does this also apply to numbers?

and...

Let's do both!

Lossy compression



Bitmap - Uncompressed
57.7 MB

JPEG - 90% Quality
9.2 MB

Lossless compression

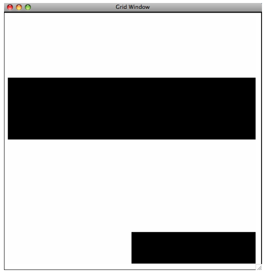
Bitmap – uncompressed
26.8 MB

PNG – compressed
14.5 MB

starting here ...

... with *BINARY* images!

Hw4: ***lossless*** binary image compression



Binary Image

```
00000000
00000000
11111111
11111111
00000000
00000000
00000000
00001111
```

same-data streaks

Encoding as raw bits

one big string of 64 characters

Hw4: lossless image compression

64 bits

"0000000000000000111111111111111100000000000000000000000000001111"

0 is the first digit

There are 16 of them.

1 is the next digit

Again, there are 16 of them.

0 is the next digit

There are 28

1 is the final digit

There are 4

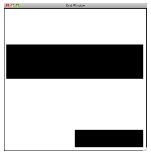
0100001100000011100**1**100

```
0100001100000011100 1100
```

0100001100000011100 1100

22 bits !!!

```
00000000
00000000
11111111
11111111
00000000
00000000
00000000
00001111
```

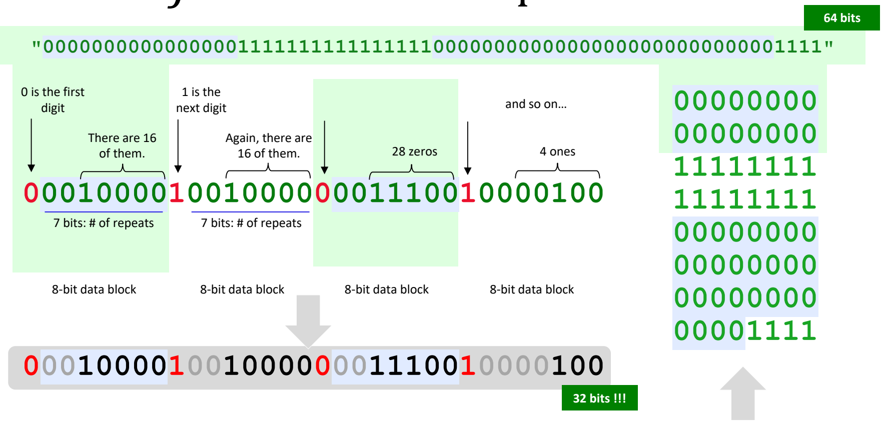


One possible algorithm:

`bit #repeats`

Any problems with this?

fixed-width compression


64 bits

"0000000000000000111111111111111100000000000000000000000000001111"

0 is the first digit

There are 16 of them.

1 is the next digit

Again, there are 16 of them.

28 zeros

and so on...

4 ones

0 0010000 1 0010000 0 0011100 1 0000100

7 bits: # of repeats

7 bits: # of repeats

8-bit data block

8-bit data block

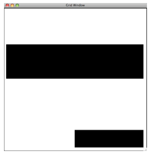
8-bit data block

8-bit data block

00010000100100000001110010000100

32 bits !!!

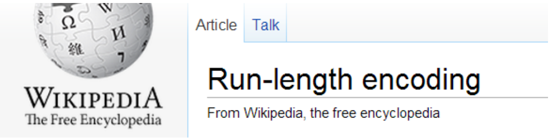
00000000
00000000
11111111
11111111
00000000
00000000
00000000
00001111



We need ***fixed-width*** blocks:

bit	#repeats
1 bit fill	7 bits for the # of repeats

← **8-bits in each "block"** →



Run-length encoding

From Wikipedia, the free encyclopedia

If you use ***7 bits*** to hold the # of consecutive repeats, what is the largest number of bits that *one block can represent*?

0 0010000

1 bit: the initial pixel

7 bits: # of repeats

8-bit total data block

7 bits?

B bits?

What if you need a ***larger*** *# of repeats?*

shortest compressed representation

longest compressed representation

What are the **BEST** and the **WORST** compression results you can get for an 8x8 image input (64 bits)?

?

BEST

?

WORST

How could we improve this compression algorithm so that ***all images*** compress to smaller than the originals? That is, how can we make compression always work ?

?

Helper fun.!

`frontNum(S)` returns the # of times the first element of the input **S** appears consecutively ***at the start*** of S:

```
                   S
frontNum('1111010')
4
                    S
frontNum('00110010')
2
```

```
def frontNum(S):
```

1 base case:

```
   if  len(S) <= 1:
     return

   elif                              :
     return

   else:
    return
```

or 2 base cases:

```
len(S)== 0:
len(S)== 1:
```

or 3 tests :

```
S== '' or
S== '1' or
S== '0'
```

If the first two bits **DO** match....

If the first two bits ***DON'T*** match....

It's all bits!

images, text, sounds, data, ...

even the string `'Hi *'` is just a sequence of bits...

`'Hi *'`

4 ASCII characters
8 bits each

4*8 == 72 bits total

`01001000 01101001 00100000 00101010`

`ord('H') == 72` `ord('i') == 105` `ord(' ') == 32` `ord('*') == 42`

All computation boils down to manipulating bits!

All computation

is simply ***functions of bits***

binary inputs **S** and **T** → output, **S+T**

S	T	S+T
00	00	000
00	01	001
00	10	010
00	11	011
01	00	001
01	01	010
01	10	011
01	11	100
10	00	010
10	01	011
10	10	100
10	11	101
11	00	011
11	01	100
11	10	101
11	11	110

This week: you'll build this in Python.

bitwise addition function

addB

Next week: you'll design this with *wires!*

Adding strings?

is **circuit** *addition!*

is **syntactic** *addition!*

syntactic ~ *"meaning-free"*

Multiplying by machine:

is **circuit** *multiplying!*

is **syntactic** *multiplying!*

Doing anything by machine...

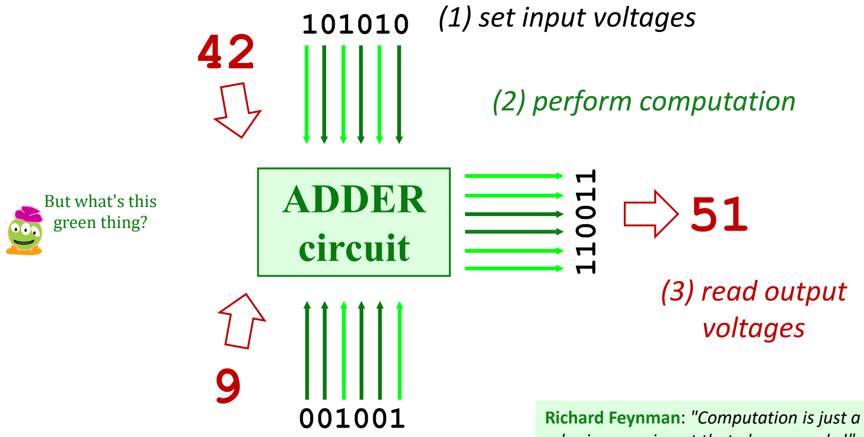
is **circuit** *interaction!*

is **syntactic** *interaction!*

means it can be done purely via ***surface syntax****, which means it can be done* ***without thinking****...*

In a computer, each bit is represented as a voltage (**1** is +5v and **0** is 0v)

Computation is simply the **deliberate combination** of those voltages!

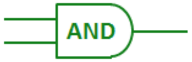


Richard Feynman: *"Computation is just a physics experiment that always works!"*

Our building blocks: ***logic gates***

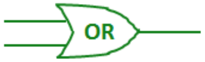
AND outputs 1 only if **ALL** inputs are 1

AND



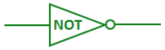
OR outputs 1 if **ANY** input is 1

OR



NOT reverses its input

NOT



These circuits are ***physical*** functions of bits…

… and *all* mathematical functions can be built from them!

Names: ______________________

```
def compress( I ):
    """ returns the RLE of the
        input binary image, I """
```

a 64-bit binary image, **I**

IQuiz

"0000000000001111111111111111111100000000000000000000011111111111"

12 zeros | 20 ones | 21 zeros | 11 ones

compress(I)

___ 1 bit ________ 7 bits ___ 1 bit ________ 7 bits ___ 1 bit ________ 7 bits ___ 1 bit ________ 7 bits

the "compressed" output, returned by compress(I)

Then, discuss

(1) What helper function would be useful for this **compress** process?

(2) What's an image **I** whose compressed output ***gets larger, not smaller***?

(2a) What are the BEST-compressible / WORST-compressible 64-bit images?

(2b) How could you ***improve*** the algorithm so that it always compresses?!!