**Character recognition.**

Suppose we have an array of bits:

01011001010010001

And our goal is to teach the machine to recognize some pattern, such as this:

0101

Given that our set of bits has a finite number of permutations equal to **pow(2,size\_of\_array)** , we can say that we also have a finite set of arrays in which we can encounter such a pattern.

This means that a BMP image consisting only of black and white pixels has a finite set of permutations in which we can encounter the character we want to recognize.

Our goal, then, is to provide the program with training data so that it can recognize similar patterns based on that data.

We could say that the data we provided is 100% a case of the desired character, but we want to put multiple cases and label them with a single name, such as 0 or 1.

This is similar to polynomials, where we can get one result, such as 0, from different parameters (images with our symbol).

Thus, the more data we have, the greater the degree of the polynomial. We can consider the function this way:

**(x-learning\_data1)\* (x-learning\_data2)\* (x-learning\_data3)\* (x-learning\_data4)...**

But how can we represent the image as a single number? We can do it similarly to how binary numbers do it.

We can say that our first set of bits **01011001011010010001** can be represented as **45713**.

We can define each individual bitmap differently, but the numbers we get will be awfully big.

For example, a 10x10 black square would **pow (2,100)** = 1,267,650,600,228,229,401,496,703,205,376.

But what if we want to use larger images?

Then we can create our own integers of any size we want. We just need to create an array of bools with the appropriate arithmetic operations.

Our polynomials use subtraction and multiplication, under which the integers are closed, so we don't need a floating point at all.

Consider the following case.

We want to track how the number which specifies our image would be changed while we change the image.

We have this image:

**00000000000000**

It is just white rectangle.

Then we add one 1 to the right

**00000000000001**

Our image has been changed slightly and we got a value bigger then previous only by 0ne.

So, with slight change of our image, we have slight change in a number, that’s what we want.

But look at this:

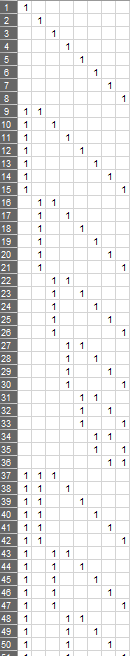
**10000000000000**

We also have changed image slightly, but our number changed astonishingly.

We can than just rearrange all possible values of the byte such that it would satisfy our demands (the less image changed, the less changed number)

I decided to count like this

(Line number specifies the actual value of the byte)

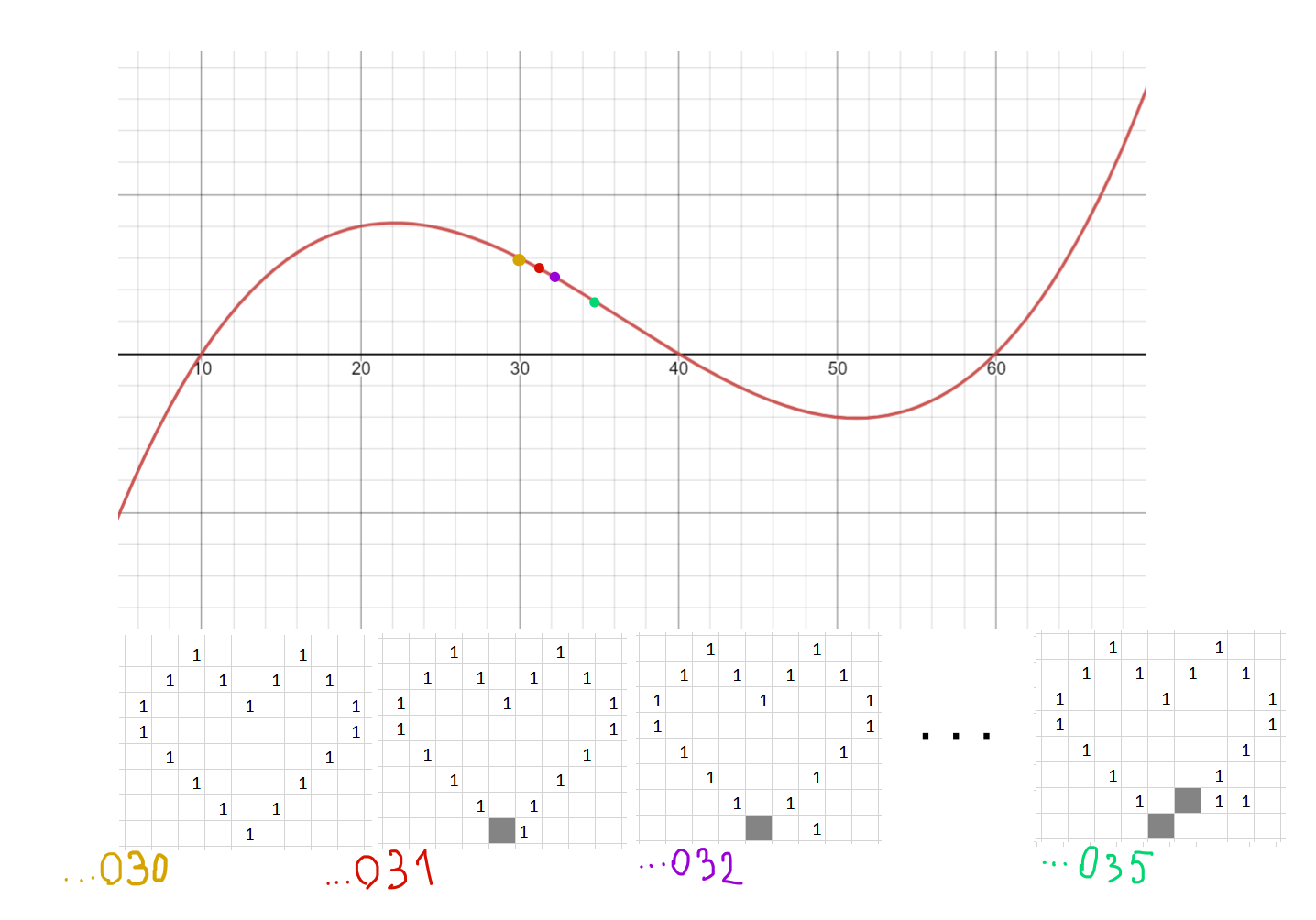


You can see that the bigger number the more ones we have in the byte.

Also, while number grows the position of most of ones is not changes.

Look at the actual image:

(Grey squares is the previous position of ones)



You can see that moving along our function line the image changes very slightly and preserve the number of ones.

I believe that there can be better approach for rearranging the byte order, but I think this way also satisfies us.

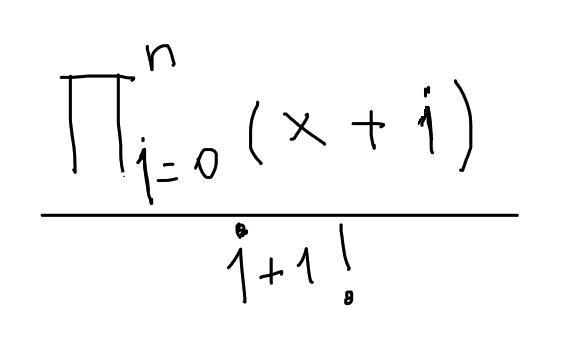
The next question is how to convert this counting system into common binary system, because it will be a had ache to do a math with this numbers.

If you will go along the number and count how much you should pass to come to the next number of ones in the byte, you’ll see the next pattern:

To pass the first part … to pass the second part … to pass the third part … the further we go, the more nested it will be.

(1+1+1+1+1+1+1+1) + (1+2+3+4+5+6+7) + ((1+2+3+4+5+6)+( 1+2+3+4+5)+( 1+2+3+4) + (1+2+3) +(1+2) + (1)) + …

There is cool formula for this:



(i - level of nesting, x – max number )

I think there is enough stuff here to figure out how to intermingle it all up ~~and get cool results~~.

After I ‘ve implemented all the aforementioned, I found one serious flaw of the above propositions.

The first and major problem is that we specify only one value in our polynomial that is 0s.

It is needed to say to the function that for example the letter “A” should give us results close to 1, letter “B” close to 2 and so on.

For that purpose, one may use Lagrange’s method for interpolation or Newton’s Divided differences algorithm.