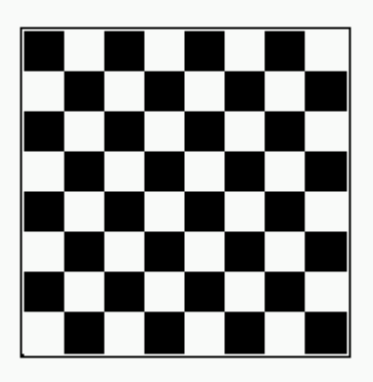
# Image Compression

Ultimately, all data in a computer is represented with 0's and 1's. We've explored how symbols can be represented as sequences of 0's and 1's; In this problem we'll explore the representation of images using 0's and 1's.

Let's begin by considering just 8-by-8 black-and-white images such as the one below:



Each cell in the image is called a "pixel". A white pixel is represented by the digit 0 and a black pixel is represented by the digit 1. The first digit represents the pixel at the top left corner of the image. The next digit represents the pixel in the top row and the second column. The eighth bit represents the pixel at the right end of the top row. The next bit represents the leftmost pixel in the second row and so forth. Therefore, the image above is represented by the following binary string of length 64:

```
'1010101001010101101010100101010110101010010101011010101001010101'
```

Of course, another way to represent that same string in python is

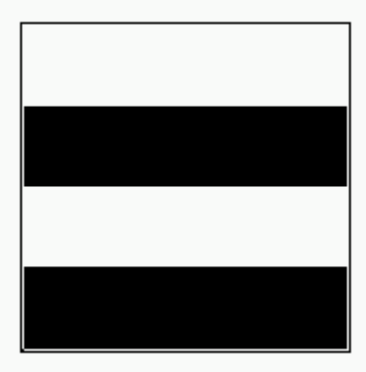
```
'1010101001010101'*4
```

(recall that this means 4 copies of the string `'1010101001010101'`).

## Backstory (also optional!)

So now what? Here's the gratuitous background story: You've been hired by NASA. NASA has a deep-space satellite that takes 8-by-8 black-and-white images and sends them back to Earth as binary strings of 64 bits as described above. In order to save precious energy required for transmitting data, NASA would like to "compress" the images sent into a format that uses as few bits as possible. One way to do this is to use the **run-length-encoding algorithm**.

Imagine that we have an image that looks like this, for example:



Using our standard sequence of 64 bits, this image is represented by a binary string beginning with 16 consecutive 0's (for two rows of white pixels) followed by 16 consecutive 1's (for two rows of black pixels) followed by 16 consecutive 0's followed by 16 consecutive 1's.

Run-length encoding (which, by the way, is used as part of the JPEG image compression algorithm) says: Let's represent that image with the code "16 white, 16 black, 16 white, 16 black". That's a much shorter description than listing out the sequence of 64 pixels "white, white, white, white, ...".

In general, run-length coding represents an image by a sequence (called a "run-length sequence") of numbers: X1, X2, ..., XN where X1 is the number of consecutive 0's until the first 1. X2 is the number of consecutive 1's until the next 0, etc. until we're done. So, for our simple image above, we'd have the sequence 16, 16, 16, 16. Notice that, by convention, the first number in the sequence is the number of consecutive 0's. Therefore, if the image/string starts with a 1, the first number in the run-length sequence would be 0 to indicate that the image begins with zero 0's.

How do we convert the run-length sequence into a binary sequence? After all, the satellite must send a sequence of 0's and 1's. One possibility is that we represent each term X1, X2, X3 in the run length sequence with a base-2 number with a fixed number, `k`, of bits. That way, we know that the first `k` bits correspond to the base 2 representation of X1. The next `k` bits correspond to the base 2 representation of X2, and so forth. (What is the right value of `k` to use? That's up to you, but you'll want to think about your choice so as not to use too few or too many bits.)

Notice that this run-length encoding will probably result in a relatively small number of bits to represent the 4-stripe image above. However, it will probably do very badly (in terms of the number of bits that it uses) in representing the checkerboard image that we looked at first. In general, run-length encoding does a good job "compressing" images that have large blocks of solid color. Fortunately, this is true of many real-world images (such as the images that NASA gets, which are mostly white with a few black spots representing celestial bodies).

Whew! So here's your job:

- Write a function called `compress(S)` that takes a binary string `S` of length 64 as input and returns another binary string as output. The output binary string should be a run-length encoding of the input string.

- Write a function called `uncompress(C)` that "inverts" or "undoes" the compressing in your `compress` function. That is, `uncompress(compress(S))` should give back `S`.
- Your `compress` function may sometimes give output that is actually longer than its input. **In a comment,** explain what is the largest number of bits that your `compress` algorithm could possibly use to encode a 64-bit string/image.
- Write `compression(S)` to return the ratio of the compressed size to the original size for image `S`.
- Test your compression algorithm on several test images of your own design. **In a comment,** describe the tests that you conducted and the compression ratios that you found. You may find it useful to write some additional functions to help automate the testing of your `compress` algorithm. Here are a few test "images" that we are providing :
    - **Penguin**: "00011000"+"00111100"*3 + "01111110"+"11111111"+"00111100"+"00100100"
    - **Smile**: "0"*8 + "01100110"*2 + "0"*8 + "00001000" + "01000010" + "01111110" + "0"*8
    - **Five**: "1"*9 + "0"*7 + "10000000"*2 + "1"*7 + "0" + "00000001"*2 + "1"*7 + "0"
- Professor I. Lai from the Pasadena Institute of Technology (P.I.T.) has made the following claim to NASA: "I have developed a new image compression algorithm `Laicompress(S)` that takes a 64-bit string and *always* outputs a *shorter* string that represents that image. That is, every image is compressed at least somewhat by my algorithm. Of course, I also have `Laiuncompress`that inverts the `Laicompress` algorithm so that `Laiuncompress(Laicompress(S))` gives back `S`. **In a comment,** argue to NASA that Professor Lai is Lai-ing—such an algorithm cannot exist! Try to make your reasoning as convincing and water-tight as possible. (In essence, you are proving that such an algorithm cannot exist.)

For all of your functions, you should think about your code before writing it. NASA will evaluate your code based on two criteria: How well it compresses random images that are fairly sparse (either lots of white with a little black or vice versa) **and** how clean and elegant your code looks. In particular, try to write as few helper functions as possible, and keep those that you write short and simple. Try to use built-in higher-order functions such as `map` and `reduce` to do much of the "heavy lifting." (Short and simple code is easier to prove correct and easier to modify.) Make sure to test your functions carefully and to document them with docstrings and comments. (*Note*: In the spirit of having short and elegant code, you *may* find yourself wanting to write a function that returns two things. How can you do that? Have your function return a list of the two elements that you want!)