Set 6. Data Compression

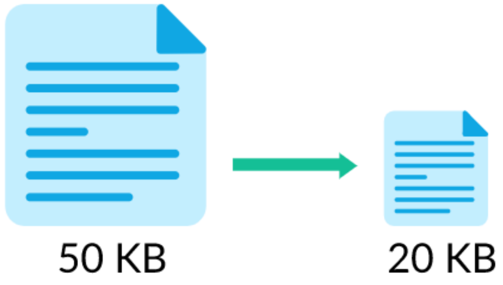
Skill 6.01: Indicate the two types of data compression
Skill 6.02: Apply lossless text compression
Skill 6.03: Apply lossless RLE image compression
Skill 6.04: Interpret a Huffman algorithm encoded sequence

Skill 6.01: Indicate the two types of data compression

Skill 6.01 Concepts

Modern computers can store increasingly large numbers of files, but file size still matters. The smaller our files are, the more files we can store.

We us **compression algorithms** to reduce the amount of space needed to represent a file



There are two types of compression**: lossless** and **lossy**

Lossless compression algorithms reduce the size of files without losing any information in the file, which means that we can reconstruct the original data from the compressed file.

Lossy compression algorithms reduce the size of files by discarding the less important information in a file, which can significantly reduce file size but also affect the file quality.

We will first explore lossless compression techniques that work for text documents, simple images, and all binary data, and then explore lossy, compression techniques for photos and audio.

Skill 6.02: Apply lossless text compression

Skill 6.02 Concepts

Lots of people compress text every day, when writing text messages. For example, if I want to say “Great, see you later!”, I could type “Gr8, see you l8r!”

The text was shortened by replacing the part that reads “ate” with “8”. We could also shorten the text by replace “see” with “c” and “you” with “u”.

“Gr8, cu l8r!”

Computers can compress text in a similar way, by finding repeated sequences and replacing them with shorter representations. They don’t need to worry about the end result sounding the same, like people do, so they can compress more efficiently.

Consider a quote from William Shakespeare:

```
to be or not to be, that is the question
```

The most obvious repeated sequences are “to” and “be”, so the computer could represent them with a character that isn’t part of the original text, like:

```
⊜ ⬗ or not ⊜ ⬗, that is the question
```

Any repeated sequence can be replaced, even if it's not a whole word, so the computer can also replace "th":

```
⊜ ⬗ or not ⊜ ⬗, ✧at is ✧e question
```

The computer also needs to store the table of replacements that it made, so that it can reconstruct the original.

replacement	original
⊜	to
⬗	be
✧	th

Skill 6.02 Exercise 1

As we saw above, some text can be compressed down quite a bit – the more repetition in a file, the more it can be compressed. Some text however, can't compressed at all, like the alphabet:

ABCDEFGHIJKLMNOPQRSTUVWXYZ

In fact, a "compressed" version of the alphabet might take more bytes to represent than the original version, depending if the algorithm puts additional metadata in the file.

Skill 6.02 Exercise 2

Skill 6.03: Apply lossless RLE image compression

Skill 6.03 Concepts

Images are all around us, from application icons to animated GIFs to photos. Image files can take up a lot of space, so computers employ a range of algorithms to compress image files.

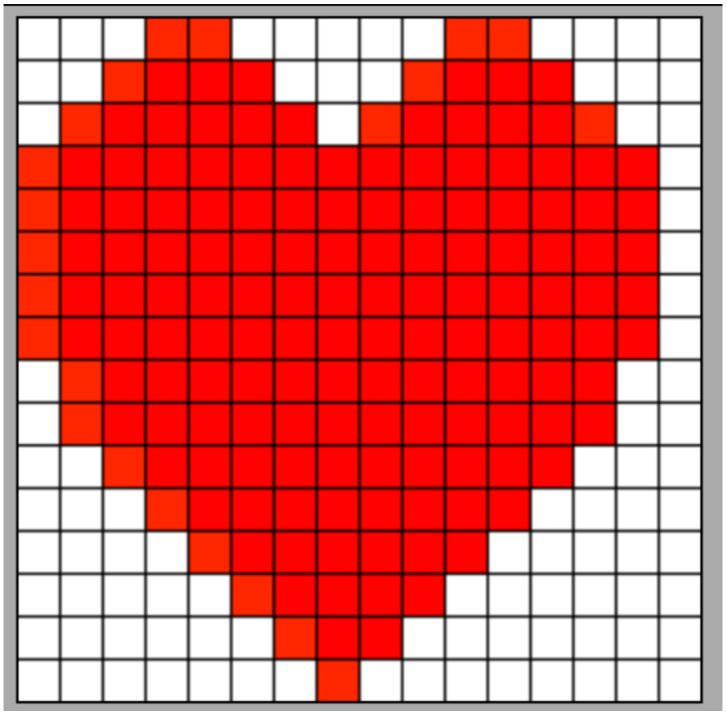
For the simplest of images, computers can use a compression algorithm called **run-length encoding (RLE)**.

Bitmaps

Before we explore image compression, let's see how we can represent an image in binary *without* any compression.

Here's a simple image, a 16x16 heart icon:

Let's zoom in and overlay a grid on top, so that it's easy to see exactly which pixels are red and which pixels are white:



The heart icon is made up of only two colors, red and white, so a computer could represent it in binary by mapping red pixels to 111 and white pixels to 000. This is called a **bitmap**, since it's mapping pixels to bits.

Using this method, the heart icon would be represented like so:

```
0001100000110000
0011110001111000
0111111011111100
1111111111111110
1111111111111110
1111111111111110
1111111111111110
1111111111111110
0111111111111100
0011111111111000
0001111111110000
0000111111100000
0000011111000000
0000001110000000
0000000100000000
```

Imagine that you had to read the bits above out to someone who was copying them down. After a while, you might say things like "five zeroes" instead of "zero zero zero zero zero". Well, the computer can do that too...

RLE compression algorithm

In run-length encoding (RLE), the computer replaces each row with numbers that say how many consecutive pixels are the same color, *always starting with the number of white pixels*.

For example, the first row contains 3 white pixels, 2 red pixels, 5 white pixels, 2 red pixels, then 4 white pixels:

```
0001100000110000
```

This would be represented as follows:

```
3,2,5,2,4
```

The fourth row is interesting because it starts with a *red* pixel. Run-length encodings start with the number of white pixels, so this is how it'd be represented:

```
0,15,1
```

Skill 6.03 Exercise 1

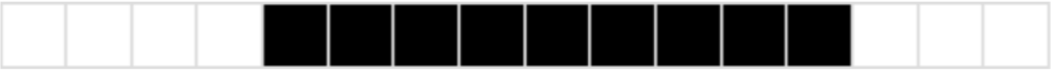
RLE Decompression

When a computer uses run length encoding, it should be able to perfectly recreate the image from the compressed representation—and so should we, if we follow the computer's strategy.

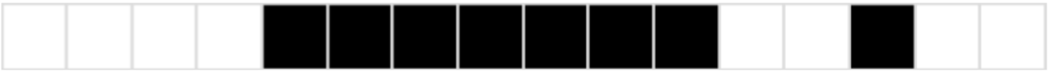
Let's try it. Here's a representation of a black & white icon using RLE:

```
4, 9, 3
4, 7, 2, 1, 2
4, 7, 2, 1, 2
4, 9, 3
4, 7, 5
4, 7, 5
5, 5, 6
0, 15, 1
1, 13, 2
```

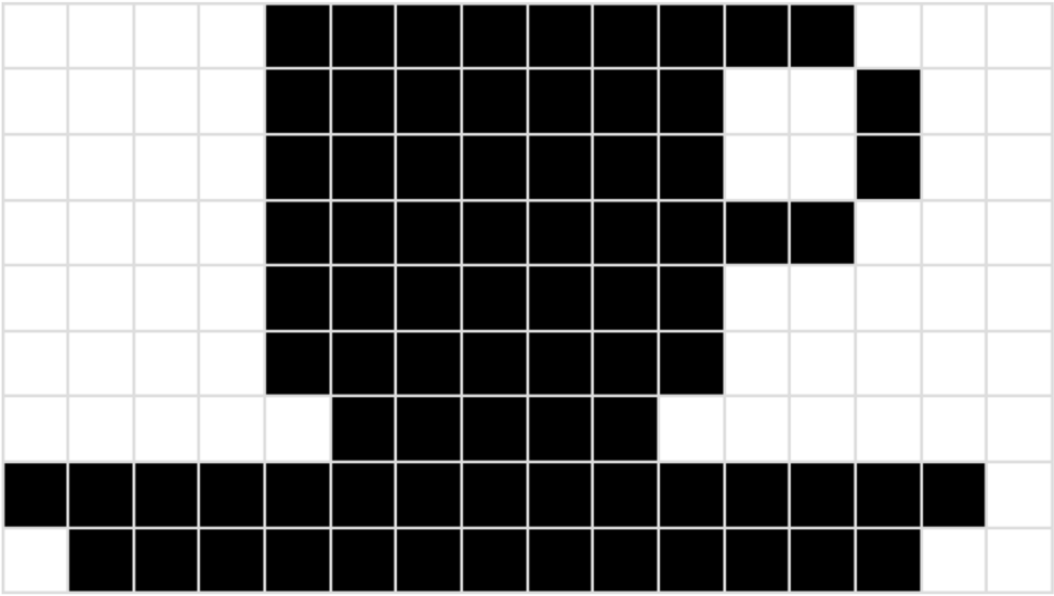
The first row has 4 white pixels, then 9 black pixels, then 3 white pixels. That looks like:



The next row has 4 white pixels, then 7 black, 2 white, 1 black, and 2 white. That looks like:



When we keep going, the final icon is a cup and saucer:



Skill 6.03 Exercise 2

Compression ratio

We've claimed that run-length encoding can save us space when storing simple images—but *how much* space? The table below summarizes the results on three heart icons of increasing size:

Image	Dimensions	Uncompressed	After RLE	Space savings
	16x16	256	228	10.9%
	32x32	1024	532	48.0%
	128x128	16384	2898	82.3%

Take a look at that final column, the space savings. Notice a pattern? We save much more space as the size increases since the runs are much longer.

What about images of the same size? This table summarizes the results of compressing three large icons with RLE:

© Pluska

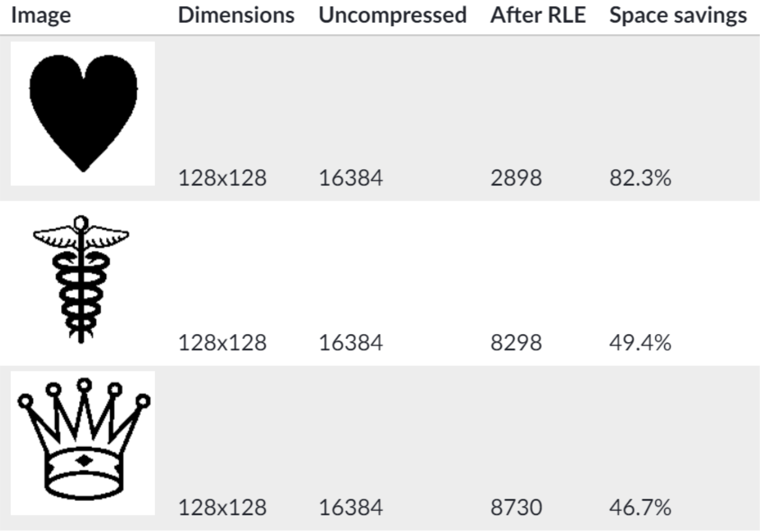


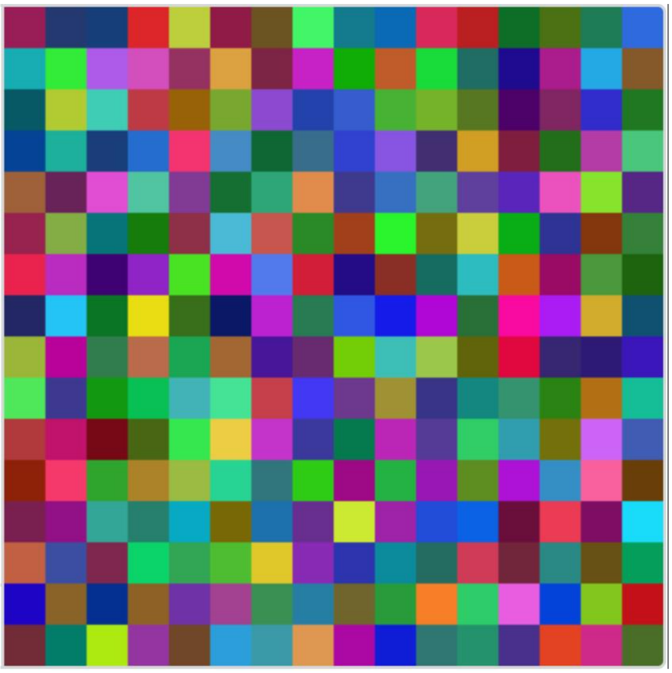
Image	Dimensions	Uncompressed	After RLE	Space savings
	128x128	16384	2898	82.3%
	128x128	16384	8298	49.4%
	128x128	16384	8730	46.7%

Now you can see why I picked a heart icon as my example: it compresses very well, thanks to its many runs of black or white. RLE compression still halves the size of the other icons, but it doesn't save nearly as much space. In fact, sometimes RLE can't save any space at all...

Limits of RLE

How about this 16x16 icon?

Let's zoom in, so we can visualize each pixel:



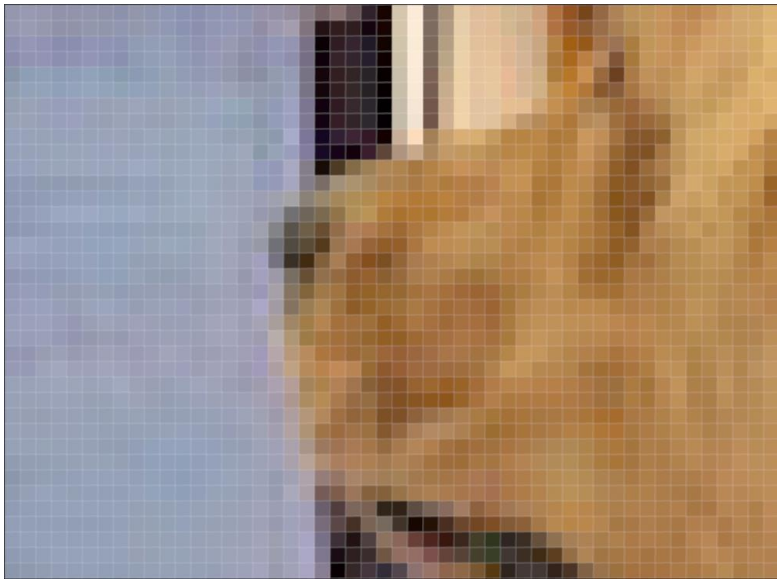
Every pixel in that icon is a **different color**, and there are **no runs**.

Run-length-encoding can't compress an image like that at all. That's an example that I made up just for this article so it might not seem that common. As it turns out, photographic images are similar to that icon—the real world is filled with details that interrupt runs.

Consider this picture of a dog looking at a computer screen,

At normal resolution, it looks like there are blocks of similar color, like in the dog's fur or the grey in the computer screen.

Let's zoom in to the pixels:



Now you can see that even the seemingly simple computer screen is a vast array of similar-but-not-quite-the-same colors. A run-length-encoding of the pixels won't do much to reduce the file size.

Uses for RLE

RLE compression was a very popular technique when most computer images were icons with limited color palettes. These days, our images are more complex and don't contain as many runs of the same color.

Where is RLE used today? Fax machines still use RLE for compressing the faxed documents, since they only need to represent black and white letters. JPEG images do use RLE during a final stage of compression, but they first use a more complex algorithm to compress the photographic details. We'll explore that more soon.

Skill 6.03 Exercise 3

Skill 6.04: Interpret a Huffman algorithm encoded sequence

Skill 6.04 Concepts

Computers represent all data in binary, so all types of files, from text to images to videos, are ultimately sequences of bits. Regardless of whether the bits represent a document or a GIF, computers can use a bit compression technique called **Huffman coding**.

Let's see how it works with a simple textual example. This example language uses only 4 different characters, and yet is incredibly important to us: it's the language used to represent DNA and is made up of sequences of four characters A, C, G and T.

For example, the 4.6 million characters representing an E.coli DNA sequence happens to start with:

```
agcttttcattct
```

Since we need to represent four characters, a computer would typically represent each character using 2 bits, such as:

character	binary code
a	00
c	01
g	10
t	11

The 13 characters above would be written using 26 bits as follows - notice that we don't need gaps between the codes for each bits.

```
00100111111111010011110111
```

But we can do better than this. In the short sample text above the letter "t" is more common than the other letters ("t" occurs 7 times, "c" 3 times, "a" twice, and "g" just once). If we give a shorter code to "t", then we'd be using less space 54% of the time (7 out of 13 characters). For example, we could use the codes:

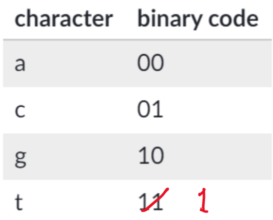
character	binary code
a	010
c	00
g	011
t	1

Then our 13 characters would be coded as:

0100110011110001011001

That's just 22 bits, four less bits than our original encoding. That may not seem like a lot, but imagine if we used an optimization like that on the entire 4.6 million characters of the DNA!

But the encoding scheme above is just one optimal case of several. And, how can we be sure it can be decoded? For example, if we just reduced the length for “t” like this:



character	binary code
a	00
c	01
g	10
t	1~~1~~ 1

We would be unable to decode the following message: 110011

Is it ttatt or tgct?

The Huffman algorithm comes up the optimal bit patterns based on how frequently each character appears in the sequence. And the beauty of the Huffman algorithm is that it gives us a way to come up with a set of binary codes for a given sequence that ensures the data can be reconstructed unambiguously and reliably. Although the Huffman algorithm is beyond the scope of this course, it is interesting! And you can learn more about it here, http://csfieldguide.org.nz/en/chapters/coding-compression/huffman-coding

Skill 6.04 Exercise 1