Binary

Recall that data in a computer is stored as streams of zeros and ones. For example, using 8 bits of storage to store the letter "A" using the ACII character encoding format, we use the bits 0100 0001, which is represented in hexadecimal as 0x41.

Choosing a Bit Depth

Imagine we need to store a piece of data. Let's take as an example a person's age in years. Since the oldest living people on earth are younger than 120 years, we know that at least for the next few decades, we will never need to represent a person's age to be over 150 years. So let's say we want to represent age as an integer between zero and 150 years old. What bit depth do we require to represent this data item?

The number of values, n, that a binary number with b digits can represent is given by the formula $n = 2^b$. For example, if b = 5, then $n = 2^5 = 32$. A binary number of 5 digits can represent 32 different values.

The logarithm is the inverse operation of exponentiation. Thus, in order to represent a number of values, n, the number of bits, b, that are required to is given by the formula: $b = \log_2(n)$. For a calculator that doesn't have a base-2 logarithm, you can use $b = \log_2(n) = \log(n) \div \log(2)$. For our example where we wish to represent the numbers 0 through 150, or 151 values, we get:

$$b = \log_2(151) = \frac{\log(151)}{\log(2)} \approx 7.2384$$

So 7 binary digits is insufficient to represent 151 different values, but 8 bits is more than enough. Note that when we use eight bits to represent the numbers 0 through 150, we have unused possible values 151 through 255. In a sense, we are wasting space.

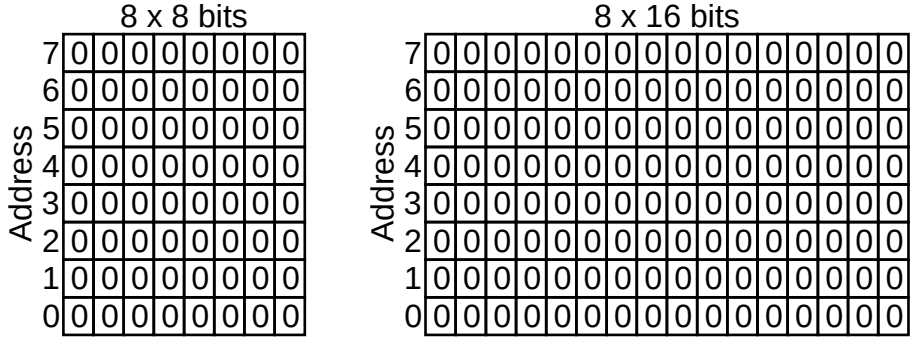
Memory Storage as an Array

While learning pseudocode and Java, we learned to use arrays. An array is a contiguous storage of the same data type. Computer storage, including computer memory can be thought of as similar to an array – a two-dimensional array of bits.

The ***width*** of memory is how many bits of data can be transferred to or from the memory at once. This is usually a multiple of two, commonly 8, 16, 32, 64, 128, or even more bits.

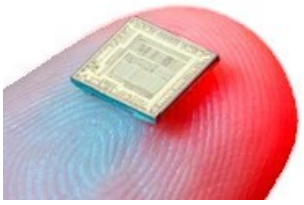
The ***capacity*** of a memory is the total amount of data that the memory can store, and is usually measured in bytes.

The diagram below shows conceptual diagrams for two memories. Both memories contain eight address locations, numbered 0 through 7. The memory on the left has a width of 8 bits. It's total capacity is 64 bits, or 8 bytes. The memory on the right has a width of 16 bits. It's total capacity is 128 bits, or 16 bytes.



When discussing memory size, the convention is to write the *address space* first, followed by *data width*. In the memory above left, the 8x16 refers to an address space with 8 different addresses, and each address has a width of 16 bits.

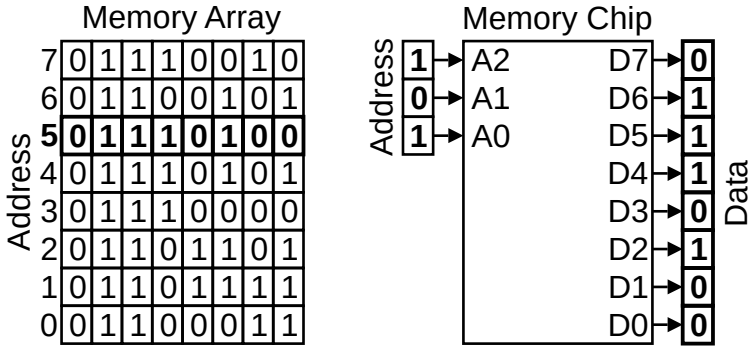
Memory Storage Chips



Computers and computer memory are physically implemented on silicon wafers that are cut into dies. An example of a silicon die is shown in the picture to the left. A die is packaged to become a "chip".

When reading data from memory, an address is given to the memory chip, and the memory chip responds with the data bits that are stored at that address in the memory array.

The diagram below shows an example of reading from memory. The memory array is shown on the left and the inputs and outputs to the memory chip are shown on the right. The read request is for the data at address 5 (binary 101), and when this address is input to the address bits of the memory, the memory outputs the data found at address 5 to the output data lines.



It is most common to write the *most significant bit* on the left hand side when written horizontally, and on the top when written vertically. In the memory array, address 5 contains the bits 0111 0100 (0x74).

The address lines together are called the ***address bus***, and the data lines together are called the ***data bus***. Notice that we only need 3 address lines to access 8 different addresses ($2^3 = 8$).

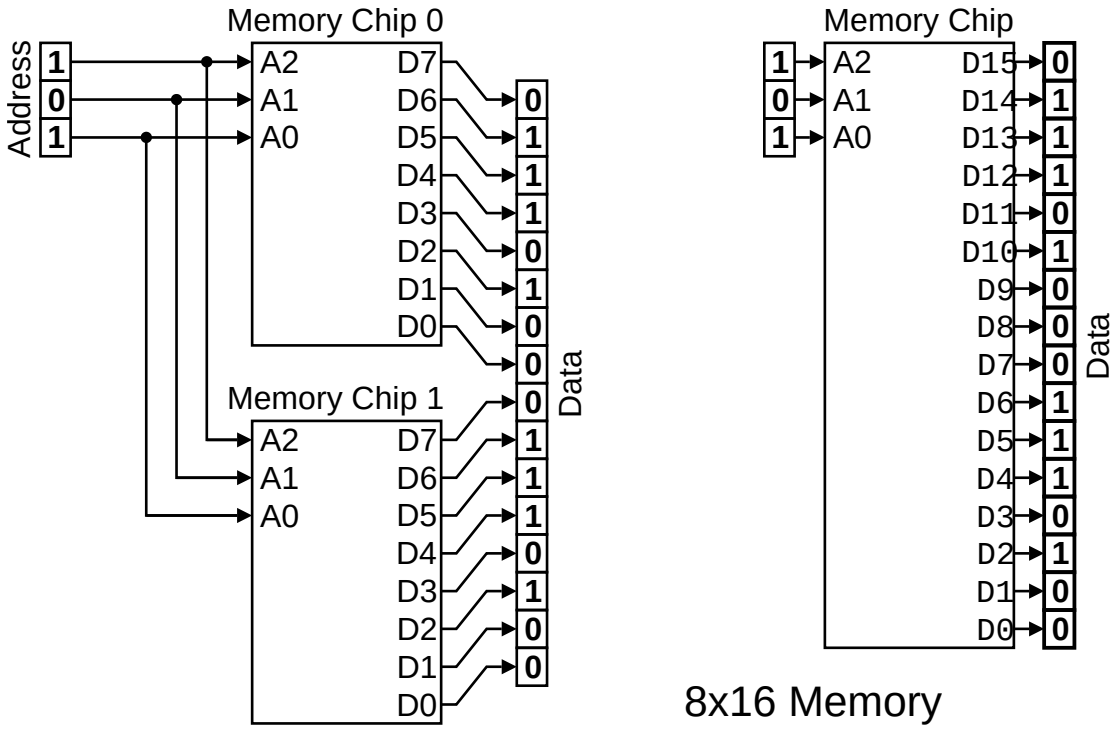
A memory chip will also have a few other lines, called ***control lines,*** that are not show. These lines inform the chip whether an operation is intended for the memory chip, whether it is read operation or write operation, etc.

Increasing Memory Capacity

The amount of memory that can be stored on a single chip keeps increasing. However, so does demand for capacity. It is often the case that a single memory chip will not have enough capacity to store the amount of data desired. To solve this, there are two ways to expand the capacity of memory beyond what is available on a single chip: *linear expansion* and *parallel expansion*.

Parallel Expansion of Memory

Parallel expansion of memory increases the bit ***width*** of memory. Examine the diagram below.



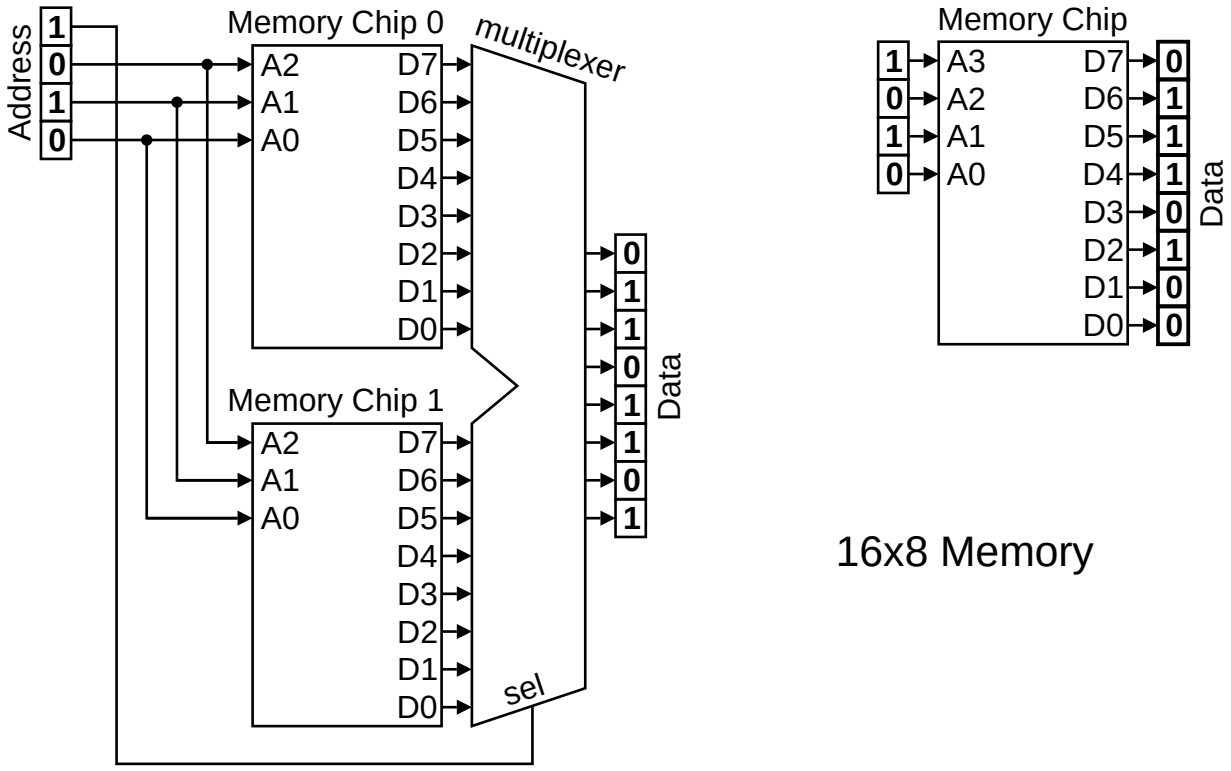
The left side of the diagram shows how an address bus can be connected to two memory chips, each memory chip containing 3 bits of address, so allowing for 8 memory locations. The data bus width is doubled in size – from 8 bits for each chip to 16 bits for the combined chip – by combining the output of both chips in parallel.

The right side of the diagram shows what the combined chip logically looks like to a computer accessing the memory. The number of chips required to implement the capacity and width of memory is made transparent to the computer.

Notice that the total capacity of memory has doubled by doubling the width. Each of the original chips has 8x8 bits of memory, for 64 bits total memory capacity. The combined circuit has 8x16 bits, for 128 bits total memory capacity.

Linear Expansion of Memory

Linear expansion of memory increases the *depth* of memory. The diagram below shows two memory chips joined together as two memory banks.



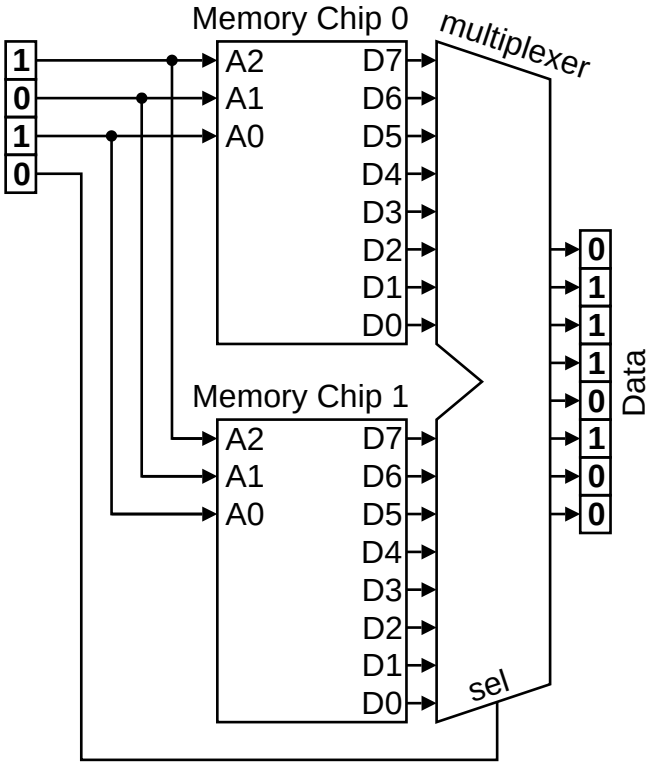
The left side of the diagram shows an address bus with four bits of address, which results in 16 possible address locations. Since each memory contains only 3 bits of memory (8 possible memory address locations), a single memory chip does not have the capacity to store the capacity that is required by a 4-bit address. Examine which address lines are connected to each memory chip. The lower three address lines are connected to each address chip, while the *most significant bit* of the address line is not connected to the memory chips.

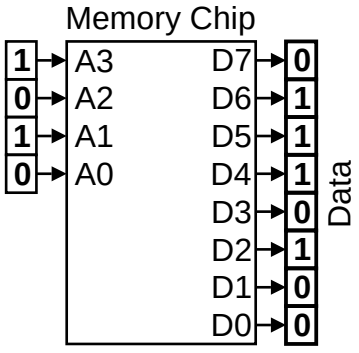
This means that for each memory chip, there are two different addresses that can be placed on the address bus that will request the same address location from each chip. For example, but address 0 and address 8 on the address bus will be seen as address 0 by both memory chips. Similarly with address 1 (binary 0001) and address 9 (binary 1001).

The *most significant bit* of the address will then select which ***bank*** of memory should finally be presented to be read. If the upper address bit is zero, it will select the data output from memory chip 0. If the upper address bit is one, it will select the data output from memory chip 1.

The right side of the diagram shows how the circuit on the left looks logically to the computer accessing the memory. Again, the total capacity of the memory has doubled, but this time the memory data width has remained the same while the number of addresses has doubled from 8 addresses to 16 addresses. Our final memory capacity is 16x8, for a total of 128 bits. This is double the memory of a single chip.

Notice that addresses 0 through 7 will access memory chip 0, while addresses 8 through 15 will access memory chip 1. Examine the difference in how the address lines are connected on the next diagram and, before reading further, determine how the addresses will be allocated to memory chip 0 versus memory chip 1.





16x8 Memory

Rather than being *banked*, this memory is referred to as *interleaved*. Every even address will be output by memory chip 0, while every odd address will be output by memory chip 1.

Notice the diagram to the right it unchanged. Whether memory chips are internally banked or interleaved (or even connected in some other way) is logically transparent to the computer reading the memory.

Doubling Memory Capacity

Understanding the structure of memory as described above should help you understand a curious question about memory. Why does memory storage often come in a multiple of two? For example, USB storage devices or storage for a phone might be 126GB or 256GB or 512GB, but we don't typically see a storage device with 350GB or 500GB – numbers that would seem more rounded to us.

Since the address lines use binary numbers to send the address to the storage device, it is much easier and more symmetric to design chips and circuitry using multiples of two. Using the techniques described in the previous sections, we can easily take two 128GB chips to create a 256GB memory circuit, or either two 256GB chips or four 128GB chips to design a 512GB memory circuit.

You can try for yourself to design a circuit that can take two memory chips that can each store 10 bytes (10x8) to create a memory storage device with double the address space (20x8) or perhaps increase by ten times (100x8). The address lines should be binary. How many address lines would be required for the 10x8 chip? How many address lines for the 20x8 circuitry or 100x8 circuitry? Which addresses would be unused for the 10x8 chip? Even though you could probably accomplish the task, it will almost certainly not as simple and elegant as circuitry that will make complete use of the address space (i.e.: increase the address space by a factor of two).

Measuring Storage Capacity

You should be familiar with the SI system that uses a ***decimal prefix***, such as k for kilo, meaning 1000. For example, 1000 grams (g) is equal to 1 kilogram (kg). The decimal prefixes use powers of 10.

Computer data storage is generally available in amounts that are a multiple of 2. It is rather unusual to store exactly 1000 bytes in a storage array, but this is a close to power of two – 2^{10} is equal to 1024. Thus it became common to round down and say that a storage that could store 1024 bytes of data stores 1 kilobyte (KB) of data. (Note also the non-SI use of capital K rather than lower case k).

The difference between the ***binary prefix***, where kilo means 2^{10}, or 1024, and the *decimal prefix*, where kilo means 10^3, or 1000, is only 2.4%. However, as you may notice in the table below, as the storage capacity becomes larger, the difference in capacity also becomes greater. This lead to the International Electrotechnical Commission (IEC) standard that allows for distinguishing decimal and binary prefixes.

Although sometimes SI decimal prefixes may still, in practice, be used to refer to storage sizes that are actually a multiple of two, it is more correct to use the proper binary prefixes.

The binary prefixes are formed by taking the first two digits of the SI decimal prefix, and appending the first two digits of the word binary. Thus “kilo-binary” becomes the prefix “kibi-” and “mega-binary” becomes the prefix “mebi-”. The symbol for each is formed by the first letter, capitalized, followed by a lower case “i”. For example, the binary prefix kibi- is given the symbol Ki, and a kibibyte is given the symbol “KiB”. Study the table below and become familiar with the different prefixes and their corresponding values.

International System of Units (SI)			**International Electrotechnical Commission (IEC)**				
Unit	**Symbol**	**Size**	**Unit**	**Symbol**	**Size**		**Difference**
kilobyte	KB	10^3	kibibyte	KiB	2^{10}	1,024	102%
megabyte	MB	10^6	mebibyte	MiB	2^{20}	1,048,576	105%
gigabyte	GB	10^9	gibibyte	GiB	2^{30}	1,073,641,824	107%
terabyte	TB	10^{12}	tebibyte	TiB	2^{40}	1,099,511,627,776	110%
petabyte	PB	10^{15}	pebibyte	PiB	2^{50}	1,125,899,906,842,624	113%
exabyte	EB	10^{18}	exbibyte	EiB	2^{60}	$\approx 1.152{,}921{,}505 \times 10^{18}$	115%
zettabyte	ZB	10^{21}	zebibyte	ZiB	2^{70}	$\approx 1.180{,}591{,}621 \times 10^{21}$	118%
yottabyte	YB	10^{24}	yobibyte	YiB	2^{80}	$\approx 1.208{,}925{,}820 \times 10^{24}$	121%
ronnabyte[1]	RB	10^{27}					
quettabyte[1]	QB	10^{30}					

[1] Official SI prefixes added in 2022

Measuring Transmission Capacity

While storage capacity is generally measured in bytes and is usually expressed in a multiple of two – this because of how data is stored on chips – transmission capacity does not follow these conventions. Transmission rates are generally expressed in bits per second, and use decimal prefixes.

Let us take the example of transmission of the first digital phone calls. Although human hearing is said to range from 20 Hz up to 20 kHz, Although the sound is often described as “tinny”, the human voice can still be understood well when the frequency range is restricted to a maximum of 4 kHz.

A digital sample rate of 8 kHz is required to accurately sample the frequencies of an analog signal with a maximum frequency of 4 kHz. (According to the Nyquist theorem, the digital sample rate must be twice the maximum analog signal frequency).

For the telephone standard, 8 bits per sample was chosen. Thus the bit rate required to transmit the digital signal can be calculated:

$$\begin{aligned} & 8000\ \frac{samples}{second} \times 8\ \frac{bits}{sample} \\ = \ & 64000\ \frac{\cancel{samples}}{second} \times \frac{bits}{\cancel{sample}} \\ = \ & 64000\ \frac{bits}{second} = \mathbf{64\ kbps} \end{aligned}$$

Early digital telephone calls transmitted data at a rate of 64 kilobits per second. (The shorthand for “bits per second” is “bps”.)

In the calculation above, the units were retained for the duration of the calculation, cancelling out units when appropriate. This is called ***dimensional analysis***. Understand this and perform dimensional analysis in all your calculations for this course. It helps to catch errors in the formula, or when trying to calculate with values that are not of compatible units (such as inches and centimeters, or even hours and seconds).

Bandwidth versus Throughput

Bandwidth is the maximum theoretical data transfer rate of a network or connection, measured in bits per second. In practice, however, many networks cannot actually maintain the theoretical maximum speed over time.

Throughput is the actual data transfer rate, which at most can be equal to the bandwidth, but is generally significantly lower. For example, if you subscribe to an internet connection that claims a bandwidth of 100 Mbps, and you test the download speed, you might find it to top out around 60 Mbps. Your throughput is 60 Mbps. The throughput is less due to thing such as network congestion, interference, data errors, etc. – topics we will discuss later in the course.

Encoding Data

To ***encode*** data is to change the format of the data for storage or transmission. Generally, to be useful the data is ***decoded*** – changed back to the original format – after being retrieved or received.

We have already discussed encoding text data using the ASCII encoding format, and encoding images using the bitmap format and vector graphics formats.

Data Compression

The ASCII characters and the bitmap files we worked with encoded that data in an ***uncompressed*** data format. It is more efficient if we store and transmit data if the data is reduced in size by using ***compression*** – the encoding of information more efficiently by using fewer bits than the original representation.

Compression may be ***lossless compression***, meaning the decoded information is an exact duplication of the original data, or ***lossy compression***, where the decoded information has lost some information, but ideally has retained the most important information. Lossy compression includes JPEG image compression and MP3 audio encoding, and we may discuss these later.

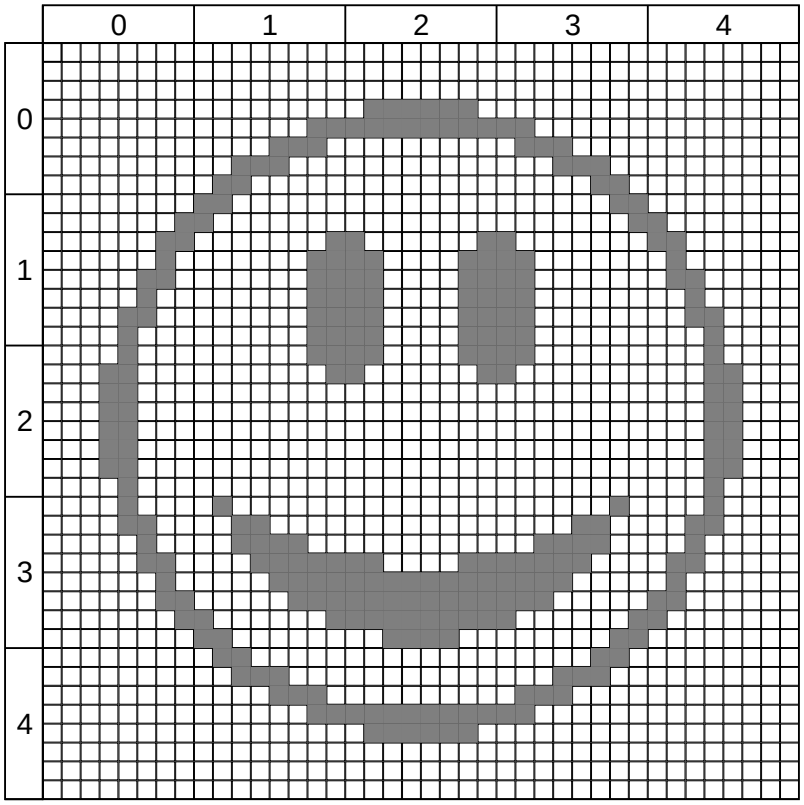
We will now look at an example of lossless encoding.

Run-Length Encoding

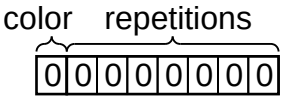
Examine the image below. Let us calculate the number of pixels, and the size of the pixel array if the image were to be stored in two-color bitmap format.

Bytes are numbered across the top and down the left-hand side. There are five bytes, or 40 bits in each dimension, and 40 × 40 gives a total of 1600 bits.

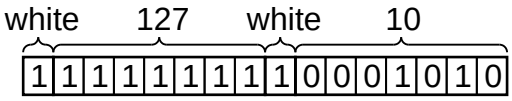
However, recall that the width of the pixel array in a Windows bitmap file needs to be a multiple of 32 bits, so each row would need to be 64 bits. This results in a pixel array size of 64 × 40, or 2560 bits.



The standard Windows bitmap file stores separate values for each pixel of the image. In contrast, ***run-length encoding*** stores the number of times that a data value is repeated. For the image above, since we have only two colors, the value of a pixel can be stored in a single bit. If we use a single byte where the first bit represents the color – let's use 1 for white and 0 for black – and the remaining 7 bits to represent the count of the number of pixels, we can represent runs of up to 127 pixels for each byte. This representation is shown diagrammatically below.



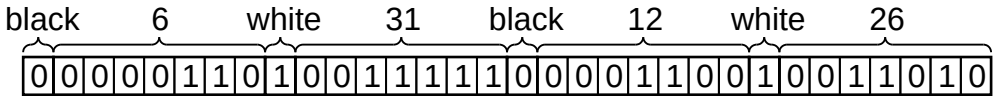
If we start to encode the smiley image above, we see three full rows of white pixels (40 pixels per row), and the fourth row contains an additional two bytes and another single pixel. This brings us to a total of 137 white pixels (40×3 + 16+1 = 137). Since we can represent a maximum of 127 pixels in a single run, it will take two bytes to represent these first rows. A run of 137 pixels can be represented as a run of 127 white pixels, followed by a run of 10 white pixels.



If we convert this to hexadecimal, we get the values 0xFF8A. What would have taken more than 12 bytes to represent as a stream of pixels we have been able to represent in only 2 bytes! This is a savings of 83%.

$$\frac{12-2}{12} = \frac{10}{12} = 0.8333...$$

Next we have 6 black pixels, another 31 white pixels, another 12 black pixels, and another 26 white pixels. This can be represented with as follows.



With these next few runs, we were able to encode a total of 38 pixels (6 + 31 + 12 + 26) into 4 bytes or 32 bytes. As we can see, when there are shorter runs of bits, we do not have such a great savings. Here we've encoded 38 bits into 32 bits, so a savings of less than 16%.

$$\frac{38-32}{38} = \frac{6}{38} = 0.157894...$$

As we can see, run-length encoding works well with images that have long strings where the same color pixel is repeated many times, but less well if the color of adjacent pixels are often not exactly the same. Consider the following array of eight pixels of alternating color.

As an array of pixels, it could be encoded in a single byte (binary 10101010, hexadecimal 0xAA), however with the run-length encoding we have devised, it would take 8 bytes! 0x8101810181018101.

With most compression methods, the amount of compression possible depends on the data that is to be compressed. The more redundancy in the data, the more compression is possible.

Run-length encoding would also likely not encode a normal photograph very well because if there are many colors in the image, adjacent pixels might be close in color, but even a slight difference cannot be encoded as a run. Other compression methods, such as the JPEG format, can better take advantage of this property of images.

Since we are able to use the run-length encoding to recreate the original image exactly, run-length encoding is a type of *lossless compression*.