



Compression: Crash Course Computer Science #21

Crash Course: Computer Science

<https://youtube.com/watch?v=OtDxDvCpPL4>

<https://nerdfighteria.info/v/OtDxDvCpPL4>

===== (00:00) to (02:00) =====

===== Introduction (00:00) =====

Hi I'm Carrie Anne and welcome to Crash Course Computer Science. Last episode we talked about files, bundles of data stored on a computer that are formatted and arranged to encode information, like text, sound, or images. We even discussed some basic file formats like text (.txt) wave (.wav) and bitmap (.bmp). While these formats are perfectly fine and still used today, their simplicity also means they're not very efficient.

Ideally, we want files to be as small as possible, so we can store lots of them without filling up our hard drives and also transmit them more quickly. Nothing is more frustrating than waiting for an email attachment to download (ugh!). The answer is compression, which literally squeezes data into a smaller size.

To do this we have to encode data using fewer bits than the original representation. That might sound like magic, but it's actually computer science.

===== Uncompressed Images (00:58) =====

Let's return to our old friend from last episode, Mr. Pac Man. This image is 4 pixels by 4 pixels. As we discussed, image data is typically stored as a list of pixel values.

To know where rows end, image files have metadata, which defines properties like dimensions. But to keep it simple today, we're not going to worry about it. Each pixel's color is a combination of three additive primary colors: red, green, and blue. We store each of these values in one byte, giving us a range of 0 to 255 for each color.

If you mix full intensity red, green, and blue, that's 255 for all three values, you get the color white. If you mix full intensity red and green, but no blue at 0, you get yellow. We have 16 pixels in our image and each of those needs three bytes of color data.

That means this image's data will consume 48 bytes of storage, but we can compress the data and pack it into a smaller number of bytes than 48.

===== Run-Length Encoding(1:55) =====

One way to compress data is to reduce repeated or redundant information. The most straightforward way to do this is called run-length encoding. This takes advantage of the fact that there are often runs of identical values in files. For example, in our Pac Man image there are seven yellow pixels in a row.

Instead of encoding the redundant data, yellow pixel, yellow pixel, yellow pixel, ... and so on. We can just say that there are seven yellow pixels in a row by inserting an extra byte that identifies the length of the run, like so, and then we can eliminate the redundant data behind it. To ensure that computers don't get confused with which bytes are run-lengths and which bytes represent color, we have to be consistent in how we apply this scheme. So we need to preface all pixels with their run length. In some cases this actually adds data, but on the whole, we've dramatically reduced the number of bytes we need to encode this image. We're now at 24 bytes, down from 48. That's 50% smaller, a huge saving.

===== Lossless compression (02:40) =====

Also note that we've not lost any data. We can easily expand this back to the original form without any degradations.

A compression technique that has this characteristic is called lossless compression, because we don't lose anything. The decompressed data is identical to the original data before compression, bit for bit.

===== (02:00) to (04:00) =====

===== Dictionary Compression (2:56) =====

Let's take a look at another type of lossless compression, where blocks of data are replaced by more compact representations. This is sort of like Don't Forget to Be Awesome being replaced by DFTBA. To do this, we need a dictionary that stores the mapping from codes to data.

Let's see how this works for our example. We can view our image as not just a string of individual pixels, but also as blocks of data. For simplicity, we're going to use pixel pairs, which is six bytes long, but blocks can be any size.

In our example, there are only four pairings, white-yellow, black-yellow, yellow-yellow, and white-white. Those are the data blocks in our dictionary we want to generate compact codes for.

What's interesting is that these blocks occur at different frequencies. There are four yellow-yellow pairs, two white-yellow pairs, and one each of black-yellow and white-white.

Because yellow-yellow is the most common block, we want that to be substituted for the most compact representation. On the other hand, black-yellow and white-white can be substituted for something longer because those blocks are infrequent.

===== (04:00) to (06:00) =====

===== Huffman Trees (3:51) =====

One method for generating efficient codes is building a Huffman tree, invented by David Huffman while he was a student at MIT in the 1950's. His algorithm goes like this:

First, you lay out all the possible blocks and their frequencies. At every round, you select the two with the lowest frequencies. Here, that's black-yellow and white-white, each with a frequency of one.

You combine these into a little tree, which have a combined frequency of two so we record that. And now, one step of the algorithm is done.

Now we repeat the process. This time we have three things to choose from. Just like before, we select the two with the lowest frequency, put them into a little tree, and record the new total frequency of all the sub-items.

Ok, we're almost done. This time, it is easy to select the two items with the lowest frequency because there are only two things left to pick. We combine these into a tree and now we're done.

Our tree looks like this, and it has a very cool property. It's arranged by frequency, with less common items lower down.

===== Dictionary Creation (4:43) =====



Compression: Crash Course Computer Science #21

Crash Course: Computer Science

<https://youtube.com/watch?v=OtDxDvCpPL4>

<https://nerdfighteria.info/v/OtDxDvCpPL4>

So now we have a tree, but you may be wondering how this gets us to a dictionary. Well, we use our frequency- sorted tree to generate the codes we need by labeling each branch with a 0 or a 1, like so.

With this, we can write out our code dictionary. Yellow-yellow is encoded as just a single 0, white-yellow is encoded as 10, black-yellow is 110, and finally white-white is 111.

The really cool thing about these code words is that there's no way to have conflicting codes, because each path down the tree is unique. This means our codes are prefix-free, that is, no code starts with another complete code.

===== Image Compression (5:15) =====

Now let's return to our image data and compress it. Our first pixel pair, white-yellow, is substituted for the bits 10. The next pair is black-yellow, which is substituted for 110. Next is yellow-yellow, with the incredibly compact substitution of just 0, and this process repeats for the rest of the image.

So instead of 48 bytes of image data, this process has encoded it into 14 bits, not bytes, bits. That's less than two bytes of data, but don't break out the Champagne quite yet. This data is meaningless unless we also save our code dictionary, so we'll need to append it to the front of the image data like this.

Now including the dictionary, our image data is 30 bytes long. That's still a significant improvement over 48 bytes.

===== (06:00) to (08:00) =====

===== (5:56) =====

The two approaches we discussed, removing redundancies and using more compact representations, are often combined and underlie almost all lossless compressed file formats like gif, png, pdf, and zip files.

Both run-length encoding and dictionary coders are lossless compression techniques. No information is lost; when you decompress, you get the original file.

That's really important for many types of files. Like, it'd be very odd if I zipped up a word document to send to you, and when you decompressed it on your computer, the text was different.

But there are other types of files where we can get away with little changes, perhaps by removing unnecessary or less important information, especially information that human perception is not good at detecting, and this trick underlies most lossy compression techniques. These tend to be pretty complicated, so we're going to attack this at a conceptual level.

Let's take sound as an example. You're hearing is not perfect. We can hear some frequencies of sound better than others and there are some we can't hear at all, like ultrasound, unless you're a bat.

Basically, if we make a recording of music and there's data in the ultrasonic frequency range, we can disregard it because we know that humans can't hear it. On the other hand, humans are very sensitive to frequencies in the vocal range, like people singing. It's best to preserve quality there as much as possible. Deep bass is somewhere in between. Humans can hear it, but we're less attuned to it. We mostly sense it.

Lossy audio compressors take advantage of this and encode

different frequency bands at different precisions. Even if the result is rougher, it's likely that users won't perceive the difference, or at least it doesn't dramatically affect the experience. And here comes the hate-mail from the audiophiles. You encounter this type of audio compression all the time.

(On the phone) It's one of the reasons you sound different on a cell phone versus in person.

The audio data is being compressed, allowing more people to take calls at once.

As the signal quality or bandwidth gets worse, compression algorithms remove more data, further reducing precision, which is why Skype calls sometimes sound like robots talking. Compared to an uncompressed audio format, like a wav or flac, there we go, got the audio files back, compressed audio files like mp3's are often 10x smaller. That's a huge saving, and is why I've got a killer music collection on my retro iPod. Don't judge.

===== (08:00) to (10:00) =====

This idea of discarding or reducing precision in a manner that aligns with human perception is called "perceptual coding" and it relies on models of human perception, which comes from a field of study called "psychophysics."

This same idea is the basis of lossy compressed image formats, most famously, jpgs. Like hearing, the human visual system isn't perfect, we're really good at detecting sharp contrast, like the edges of objects, but our perceptual system isn't so hot with subtle color variations. Jpeg takes advantage of this by breaking images into blocks of 8x8 pixels, then throwing away a lot of the high frequency spatial data. For example, take this photo of our director's dog, Noodle. So cute!

Let's look at a patch of 8x8 pixels. Pretty much every pixel is different from its neighbor, making it hard to compress with lossless techniques because there's just a lot going on. Lots of little details. But human perception doesn't register all those details, so we can discard a lot of it and replace it with a simplified patch like this. This maintains the visual essence, but might only use 10% of the data. We can do this for all the patches in the image and get this result. You can still see it's a dog, but the image is rougher.

So that's an extreme example, going from a slightly compressed jpg to a highly compressed one, 1/8th the original file size. Often, you can get away with a quality somewhere in between, and perceptually, it's basically the same as the original.

The one on the left is 1/3rd the file size of the one on the right. That's big savings for essentially the same thing. Can you tell the difference between the two? Probably not, but I should mention that video compression plays a role in that, too, since I'm literally being compressed right now. Videos are really just really long sequences of images, so a lot of what I said about them applies here too, but videos can do some extra-clever stuff because between frames a lot of pixels are going to be the same, like this whole background behind me. This is called temporal redundancy. We don't need to retransmit those pixels every frame of the video; we can just copy patches of data forward.

When there are small pixel differences, like the read out on this frequency generator behind me, most video formats send data that encode just the difference between patches, which is more efficient than retransmitting all the pixels afresh, again taking advantage of interframe similarity.



Compression: Crash Course Computer Science #21

Crash Course: Computer Science

<https://youtube.com/watch?v=OtDxDvCpPL4>

<https://nerdfighteria.info/v/OtDxDvCpPL4>

===== (10:00) to (12:00) =====

The fanciest video compression formats go one step further. They find patches that are similar between frames, and not only copy them forward, with or without differences, but can also apply simple effects to them, like a shift or rotation. They can also lighten or darken a patch between frames. So if I move my hand side to side like this, the video compressor will identify the similarity, capture my hand in one or more patches, then just move these patches around between frames. You're actually seeing my hand from the past. Kind of freaky, but it uses a lot less data.

MPEG-4 videos, a common standard, are often 20 to 200 times smaller than the original, uncompressed file. However, encoding frames as translations and rotations of patches from previous frames can go horribly wrong when you compress too heavily and there isn't enough space to update pixel data inside of the patches. The video player will forge ahead, applying the right motions, even if the patch data is wrong. And this leads to some hilarious and trippy effects, which I'm sure you've seen.

Overall, it's extremely useful to have compression techniques for all the types of data I discussed today. I guess our imperfect vision and hearing are useful, too. And it's important to know about compression because it allows users to store pictures, music, and videos in efficient ways. Without compression, streaming your favorite Carpool Karaoke videos on YouTube would be nearly impossible, due to bandwidth and the economics of transmitting that volume of data for free. And now when your Skype calls sound like they're being taken over by demons, you'll know what's really going on. I'll see you next week.

Hey guys, this week's episode was brought to you by CuriosityStream, which is a streaming service full of documentaries and nonfiction titles from some really great filmmakers, including exclusive originals. Now I normally give computer science recommendations - since this is CrashCourse Computer Science and all - and CuriosityStream has a ton of great ones. But you absolutely have to check out "Miniverse" starring everyone's favorite space-station-singing-Canadian astronaut, Chris Hadfield, as he takes on a road trip across the Solar System scaled down to the size of the United States. It's basically 50 minutes of Chris and his passengers geeking out about our amazing planetary neighbors and you don't want to miss it.

===== (12:00) to (12:48) =====

So get unlimited today, and your first two months are free if you sign up at curiositystream.com/crashcourse and use the promo code "crashcourse" during the sign up process.

CrashCourse Computer Science is produced in association with PBS Digital Studios. At their channel you can check out a playlist of shows like PBS Idea Channel, PhysicsGirl, and It's Okay to Be Smart.

This episode was filmed at the Chad and Stacey Emigholz Studio in Indianapolis, Indiana, and it was made with the help of all these nice people and our wonderful graphics team, Thought Cafe. Thanks for watching. I'll CPU later.