Compression

Compression saves:

- space: use less memory --> going to save I/O requests
- time: save time by saving space, which is to save I/O requests by using more CPU time

> I/O is so much slower than CPU time on modern machines
> We are saving real time by spending CPU time --> more energy cost

There are two big ideas used in compression programs like zlib, etc.

(1) Huffman Coding

The **BEST** possible algorithm **under its constraints**:

1. Input is a sequence of symbols taken from a known alphabet.
2. We know the **popularity** of each input symbol. (say a byte)
3. Each input symbol corresponds to some bit stream. (byte into bit stream, and package the stream into byte)

When you have these constraints, you can assemble a table mapping input symbols to output bit strings, with **lengths chosen based on how frequent they are**. With ASCII characters in the context of the English language as an example:

Symbol	Bit String
(space)	00
e	010
t	011
a	100
o	1010
...	...
^Q	111011011101

Notice that the least popular symbols could map to a representation that takes up even more space than their original form, but that's okay because they occur so scarcely that on average it does not hinder the total length by much.

Then, compression is simply a matter of iterating over the input symbols and performing a lookup.

Decompressing

- **OPTION A:** You have a pre-computed table shared by the compressor and de-compressor. Sender computes the table and sends it first. This introduces some overhead in transmission, but more importantly, it has to read all of the input first.

- **OPTION B: Dynamic (Adaptive) Hoffman tables**.

  > The problem of `Option A` is its assumption:
  >
  > > knowing the probability of each symbol in advance, but in many cases you don't

  1. Both sender and recipient start off with a table in their head that's perfectly balanced (each leaf node has the same distance with the root).
  2. Sender then sends the first byte and updates its Hoffman table according to the byte that it already sent.
  3. The recipient does the same, and because both sides can detect the same popularity of the data they are sending/receiving, their tables are kept in sync.

  This is a little slower because they have to update the tables dynamically as they go but not by much. You also have to agree on what to do when there's a tie in popularity. Both sides need to be able to stay in lock step with each other.

- **For example:**

  Following the most left path, we can eventually reach the null node of `00000000`, and the right most we have the leaf `11111111`
  Assuming each leaf node has weight of 1 occurrence

  - NOTE that one byte has 8 bits, which has 256 probable values, so we have 256 leafs here for each byte

  But now, assume `a` has twice occurrences of others, then it has twice the weight, say one less bit than others
  For an extreme example, `a` may finally have only one bit, which the others take more.
  In such case, it takes only one bit to send an `a` character

  The BEST Huffman Coding can do is to compress 1 byte to 1 bit

Table Generation Algorithm

Suppose you computed a table of probabilities (possibly by counting the frequencies of each symbol in some sample text):

Symbol	Probability
(space)	0.1
e	0.07
t	0.05
...	...
SUM:	1.0

You build a **Huffman tree** to determine the bit string to assign. Review your MATH 61 notes lol.

(2) Dictionary Coding

An approach that can be more optimal than Huffman coding by relaxing a constraint. Instead of mapping individual symbols to bit strings, you map sequences of symbols to bit strings.

For example:

Sequence	Encoding
the	1
a	2
an	3
or	4
...	...

You read the text, divide it into *words* (hence "dictionary" coding), and then for each word, you assign a number to it. Of course, *words* here refer to any byte streams, so they likely include punctuation marks or spaces for efficiency.

If the sender and recipient know this table, then a string can be compressed much more efficiently than when using Huffman coding.

Decompressing

There is also **dynamic dictionary coding**.

1. Start with a trivial dictionary with 256 entries, where each word is length 1 and represented with some number from 0-255.
2. As data is transmitted, the tables on both side are updated dynamically and in the same way such that by the end, they can use dictionary coding.

- **Example 1:**
  Suppose you already sent a substantial chunk of data. You can send data in terms of what you already sent. For example, if you're about to send the word "French", and you've sent "French" 97500 bytes ago, you can send a pair of integers, offset and length, like (97500, 6), which the recipient can decode with their copy of the data.

- **Example 2:**
  It works like the below:

  Sender: `|` is where the sender now pointing at `already sent umbrella (500 bytes msg) | umbrella askfhkjasfh`

  Recipient: `|` is where the sender now pointing at `already sent asfsf umbrella (500 bytes msg) |`

  And now the sender sends a msg to the recipient: `508,8`, which means go back 508 bytes and read the next 8 bytes, `umbrella`

> There is also the **sliding window algorithm** where you have a moving range of the sent data that you can use for this approach.

> This is **NECESSARY** because RAM has a limit. This is the case when data is large like `50T`

zlib

zlib uses both of these approaches. It uses dictionary coding to output a sequence of numbers, and then it uses Huffman coding on that sequence.

- **Example:**
  Dictionary takes a bunch of string of bytes and turns them into small bytes, (the msg mentioned above)
  Huffman Coding then takes these small bytes and generates a Huffman Tree to encode these msg

  In such case, both the sender and the recipient keep a fixed dynamic sliding window and a Huffman Tree on sync

Does compression guarantee you get something smaller than what you started off with?

No. If this were true, than you can compress an indefinite number of times, all the way until 1 byte, which would then have to be compressed to 0 bytes under this assumption. This is obviously impossible to do without losing information.

> **ASIDE:** A **general purpose compression algorithm** has to be able to work with *any* byte sequence. Certain algorithms can be specialized for certain types of input, but software like Git needs something general purpose like zlib because it has to be able to work with any blob.

Compression Quality measures:

- shrinkage: such as .21 of decompress (this may depends on languge to encode)
- usage of RAM (compressor, and decompressor)
- CPU time
- energy cost ...