First, you must make sure you ask the right question. If you just want to compress one single text, you can compress it to one bit: let the encoded version of your text "1", but every other text will become "0", followed by the unchanged original text. You decode it by looking at the first bit. If it is "1", then you put out the original text, if it is "0", just drop it and emit the rest.   
This "compression algorithm" is effective only for one text, ineffective for the others. This is typical for what you can expect: Whatever kind of compression algorithm you have, it will effectively compress some texts, and not compress, and in some cases even prolong others. (That is easily seen by the so called "counting argument")  
  
So you want to have a algorithm that compresses one class of texts well (say, all texts in English), some less well (say, texts in Latin), and some not at all (completely random text strings). To get at numbers, you have to make a model of the "machine" that produces your texts, and will produce texts that look like English with higher probability than texts that look like garbage. A good approximation for such a machine is a so called [Markov source](http://en.wikipedia.org/wiki/Markov_source). It can take finitely many states, and in any state, it selects a random letter, according to a probability distribution attached to that state, emits that letter, and changes into another state, depending on that letter. For example, assume your 129 states are the 128 ASCII characters, and one initial state. These represent the previous character you emitted. So, if you are in the state "q", you will emit a "u" as the next character with a high probability, and the other characters with much lower probability. That way, you can produce texts that look a tiny bit like English. If you want to refine, have more states that represent the last two characters you emitted, and so on.  
  
Now, if you have a Markov source, you can define limits: let **A be your alphabet, **{1,…n}the states of the Markov source, and **pia the probability that it emits the sign **a in state **i. We denote the probability that the source is in state **i with **πi. Then the we define the *entropy of the source* as **−∑****∑**∈****H=−∑i=1nπi∑a∈Apialog2⁡pia.  
  
[Shannon's source coding theorem](http://en.wikipedia.org/wiki/Shannon%27s_source_coding_theorem#Source_coding_theorem) says that under some technical conditions on the Markov source, the entropy of the source is a lower bound for the average bit length of the encoded text, divided by the length of the text. (The formula in the entropy definition is based on the idea of encoding more probable characters with less bits, hence the −**−log2⁡pia, which can be interpreted as the number of bits you assign to sign **a in state **i, and it averages over all the states and texts with the appropriate probabilities.)  
  
Often, compression algorithms align with this view: You prepare a model of your text class as a Markov source and find a way to encode this source optimally.  
The well-known [Huffman coding](https://en.wikipedia.org/wiki/Huffman_encoding) approaches the entropy bound optimally, provided you want to encode character by character, and you cannot encode characters by bit strings of fractional length; [Arithmetic coding](https://en.wikipedia.org/wiki/Arithmetic_coding) is a way to actually realize "bit strings of fractional length".   
  
The Lempel-Ziv  [LZ77 and LZ78](https://en.wikipedia.org/wiki/Lempel-Ziv) compression methods are based on a different idea (not Markov modeling and encoding characters but referencing substrings); however, it can be shown that, if they are used to compress texts emitted by a Markov source, their compression quality approaches the entropy asymptotically. In a way, the LZ algorithms work without needing a model first because they implicitly build the model themselves.