Arithmetic LZW (ALZW)

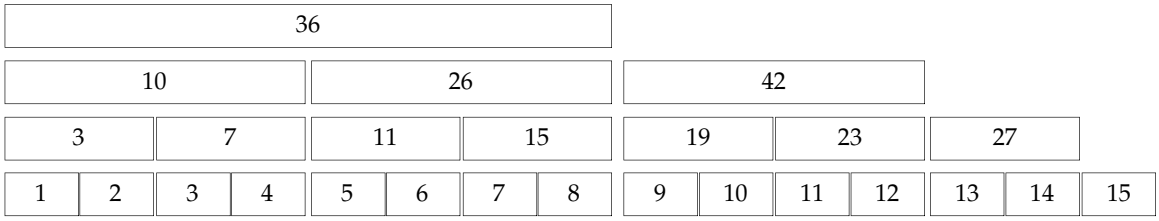
Run Xuan Yang (2017280387)

This project makes LZW compression encoding more compact by combining it with arithmetic coding, plus other additional improvements, including adaptive frequency count with multi-alphabet, forget procedure, and forbidden region.

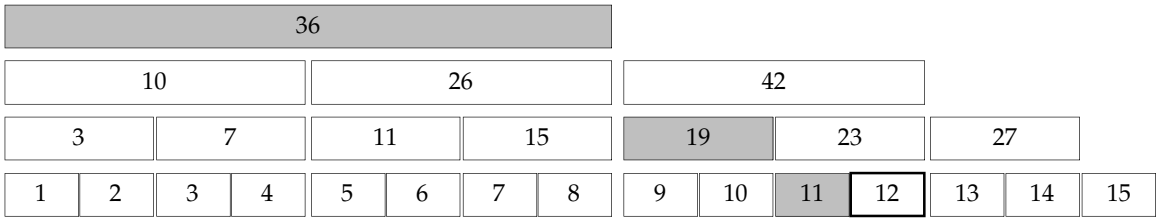
One of the problem with LZW is that each compressed alphabet has to be represented by an integer number of bits. Let's say we already have 8 entries in the dictionary, each entry is indexed from $(000)_2$ to $(111)_2$, adding a new entry $(1000)_2$ will increase the number of digits by 1, while the entries from $(1001)_2$ to $(1111)_2$ are not used until sometime later. This is essentially equivalent to creating 16 entries while not using 7 of them, leaving almost half of the entries being wasted. By encoding the entry indices using arithmetic coding, we can equally split the interval into only 9 equal ranges, this way we eliminate the wasted spaces by only adding 1 entry each time.

Another issue with LZW is that, some entries are used more often than other entries, but the encoding size does not vary according to their frequencies. ALZW addresses this by using adaptive encoding, namely the weight of an entry will be set to n_0 when it's newly added, and increased by n every time it's used. According to [1], the encoding achieve the best performance when n_0 and n are set to 1 and 2 respectively. The entire interval will be equally split into smaller intervals whose number is the sum of all weights, and each entry will have its interval length according to its weight.

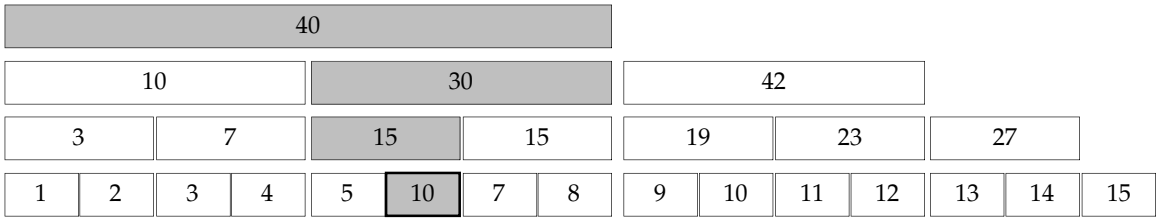
This gives another problem that, since the interval size of an entry is constantly changing, it's time-costly to update the offset of every entry that comes after it. Alternatively, we can only store the interval size of each entry and calculate the offset of a certain entry upon required, which is the approach used in [1], but this requires summing up all entries before it. Either way is an $O(n)$ operation. This is solved by using a binary tree whose nodes always keep track of the sum of their two children, and makes this and other operations work in $O(\log n)$ time. For example, suppose we have 20 entries in the dictionary, each with weight from 1 to 15 respectively, then the tree looks like this:



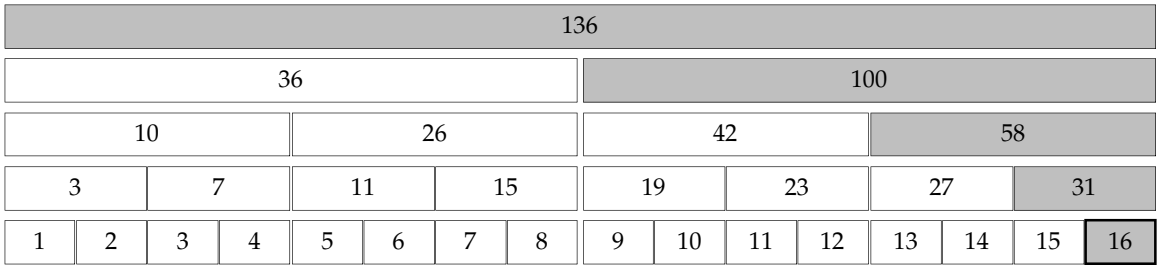
Finding the offset of an element, say 12, only requires calculating $36 + 19 + 11 = 66$, as shown below.



Updating an existing value, say changing 6 to 10, will also recursively update its parents.



Adding a new entry to the end, say 16, will also add necessary parent nodes, as well as new layers when needed.



Removing an existing entry is not optimized by this data structure, and fortunately we don't need this kind of optimization as it will only happen in batch during forget procedure, as we will show later in the text.

And finally during decoding, querying which interval contains a certain value, say 60, works top-down as follows:

- In the 1st layer, $60 \geq 36$ so it's also larger than both of its children.
- In the 2nd layer, $60 - 36 < 42$ so it lies within one of its children.
- In the 3rd layer, $60 - 36 \geq 19$ so it's also larger than both of its children.
- In the 4th layer, $60 - 36 - 19 < 11$ so it lies within the 11th range.

Also, as previous examples have probably already shown, the length of each layer, say layer k (one-based), is always at least 2^{k-1} and at most $2^k - 1$. This makes sure that the extra memory is always at most n plus some constant, where n is the total number of entries.

Another problem with LZW is that, once the dictionary gets full, new entries will not be added any more. This may potentially affect compression ratio if the entries in the dictionary no longer represent the characteristics of the input. This is partially addressed in [1] by assigning more weights to entries with longer words. ALZW uses a different approach by implementing a forget-procedure, such that every entry will have its weight divided by 2 each time the number of bytes encoded reaches a multiple of a certain number, called block size, and entries with weight 1 will just get removed. This also makes sure that the number of entries in the dictionary is never greater than twice the block size.

However, removing an entry that is prefix of another entry will cause problem when trying to match input with the longest word in the dictionary. For example, both "afte" and "after" exist in the dictionary with weights 1 and 5 respectively. If we remove "afte" from the dictionary, it might be difficult to tell whether "after" still exists or not. One solution to this problem is to always keep track of the length of the longest word, and always try to match input until that length, but this could take a lot of time if one of the words gets very long. ALZW took another approach, which is to only remove entries that are not prefix of other entries, and also never remove single-alphabet entries. This somewhat nullifies the property that the dictionary size is at most twice the block size, but in reality it didn't cause an observable defect in memory consumption.

ALZW uses a variant of arithmetic coding called range coding in order to have better control over precision and rounding. The general idea is to equally split the whole interval into 4 regions, and whenever the current interval intersects with only 2 regions, we expand the current interval and pop the corresponding bit. This requires the dictionary to always have less entries than the size of one region so that each word is assigned to an interval of length at least 1. ALZW uses interval size 2^{28}, There are other variants that pops a whole byte at a time, which gives slightly better performance but requires finer precision control.

ALZW also reserves an interval of size 1/10000 that denotes EOF, as well as a variable-sized interval for forbidden region that enables error checking - If the encoding falls into the forbidden region during decoding, the program will crash. The default size of the forbidden is 30/10000, which increase the output size by 0.3%.

The following table records the performance of ALZW encoding over different files using different block size. All tests are run with 0.3% forbidden region.

	File name	bible.txt	mspaint.exe	SegoeWP.ttf
	File size (bytes)	4587478	6532096	492656
Block size 2^{12}	**Output (bytes)**	1670501	3213086	336470
	Duration (s)	10.6450602	42.8610479	1.1633891
Block size 2^{16}	**Output (bytes)**	1417758	3114780	352496
	Duration (s)	12.4178512	42.1171956	1.3357028
Block size 2^{20}	**Output (bytes)**	1365730	3237937	363415
	Duration (s)	10.0417823	32.7183894	0.8152188
Block size 2^{24}	**Output (bytes)**	1387973	3329250	363415
	Duration (s)	7.1860892	27.6820071	0.9602554

References:

[1]: Y. Perl, V. Maram and N. Kadakuntla, "The cascading of the LZW compression algorithm with arithmetic coding," [1991] Proceedings. Data Compression Conference, Snowbird, UT, 1991, pp. 277-286.