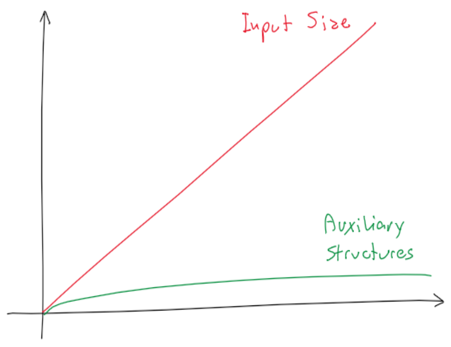
Problem 3: Huffman Coding

The algorithm input n will correspond to the number of characters in the original data to be encoded.

Since we analyze the time and space complexity relative to input growth, we may have two different situations:

- Best case: An input much bigger than the size of the alphabet part (or the number of all different symbols) encountered in the input itself.
- Worst case: A short input string with size comparable to the number of symbols encountered. Being a noticeably short input of not many repeating patterns.

This is relevant since program creates and iterates over some auxiliary data structures which size is more related to the number of symbols found in the input than to the input itself. The size of these auxiliary structures tends to be constant as the input grows. Their size is limited by the alphabet ***t*** size. For most real-world data compression applications with large inputs, the alphabet size will grow and approximate to a constant limit (the size of ASCII or UTF8 tables, for example). At this point, it will remain constant and independent of input size as shown in the picture below.



We may consider that the poorer performance in the worst-case scenario do not lead to a time-consuming process since the input size is short, after all.

1) Encoding Function

The encoding function is addressed by a sequence of five steps, which will be separately described and analyzed.

Create and fill a Char-Frequency Map

The algorithm firstly traverses the input data and creates a dictionary that maps the frequency of each symbol occurrence. Each character in the input requires a ***get()*** and a ***set()*** dictionary ***O(1)*** operations. Thus, resulting in a ***O(n)*** time complexity for this part.

For space complexity, the map is limited by the alphabet ***t*** size. Thus, a best case ***O(1)*** for space complexity. The algorithm will only perform a ***O(n)*** with short length inputs with lack of repeating patterns.

Create and fill a min-heap based Priority List

In here it is created a priority queue, based on **[1]**, being the highest priority those characters with minimum frequency. The queue is created by using a binary minimum heap tree data structure. This structure allows logarithm time complexity for recovery from a ***pop()*** followed by a ***sift-down()*** operation. The insertion operation also have a logarithmic behavior by ***push()*** followed by ***sift-up()*** operations. Both ***sift-down()*** and ***sift-up()*** are required to rearrange the structure to maintain the heap constraints. They both need a tree traversal which explains the log behavior. Such operations will be performed for node in the priority list.

Nevertheless, to build the priority list the algorithm only needs to traverse the Char-Frequency Map dictionary, which will be limited by its ***t*** size (according to the alphabet size), which tend to be constant for large inputs as explained before. Thus, this part of the algorithm can be categorized by a ***O(1)*** time complexity in best case for large inputs. It will only perform a ***O(n log n)*** with short length inputs of no repeating patterns.

For space complexity, the priority queue size will accompany the behavior of Char-Frequency Map construction since they must have the same number of nodes/items. Thus a best case ***O(1)*** for large inputs and ***O(n)*** with short inputs.

Construct a Huffman Tree by traverse the Priority List

The construction of the Huffman tree requires traversing the Priority List popping and pushing data from and to it, which will perform similarly to the previous part, regarding to the time complexity. Thus, ***O(1)*** in best case for large inputs and ***O(n log n)*** with short inputs.

For space complexity, the Huffman tree size will be proportional to the priority queue size. Thus a best case ***O(1)*** for large inputs and ***O(n)*** with short inputs.

Visit Huffman Tree nodes recursively and build up a Huffman Code Table

To build up a Huffman code table the algorithm needs to perform a walk across every edge in the Huffman Tree. This is performed recursively by a ***visit()*** function and the number of times this function is executed is equal to the size of the Huffman tree. Thus, ***O(1)*** in best case for large data inputs and ***O(n)*** with short inputs.

For space complexity, the Huffman code table size will have the same length of the first Char-Frequency Map dictionary. Thus a best case ***O(1)*** for large inputs and ***O(n)*** with short inputs.

Encode Data

Once built the code table the algorithm traverses the entire input again to correlate each character to its corresponding code. Thus, this part has ***O(n)*** time complexity.

The structure size needed to handle the encoding process is simply the size of one character symbol, since the characters are encoded one by one. Thus, a space complexity of ***O(1)***.

The overall complexity associated with the 5-step encoding process will correspond to the maximum of the complexity encountered in each step.

- Best case: ***O(n)*** time complexity and ***O(1)*** space complexity;
- Worst case: ***O(n log n)*** time complexity and ***O(n)*** space complexity.

2) Decoding Function

The decoding function is addressed by simply parsing the encoded data and decode it based on the Huffman Tree walk.

Each decoded character will need a walk from root to leaf in the Huffman tree. The number of steps to decode back each input character is log ***t***, being ***t*** all possible encoded characters (or the number of leaves in the Huffman tree). Although it is expected to walk deeper in the Huffman tree only for less frequent characters. Since the tree size tends to be constant for bigger inputs, each tree traversal tends to ***O(1)*** in this best scenario or ***O(log n)*** with short inputs.

The total encoded data size will be proportional to the input size by the achieved algorithm compression rate. Thus, giving a resulting time complexity of ***O(n)*** in best case and ***O(n log n)*** with short inputs.

The data structure size needed for handling the decode operation corresponds to the Huffman tree received as parameter and a pair of auxiliaries. Thus a best case ***O(1)*** for large inputs and ***O(n)*** with short inputs.

References

[1] Heap (data structure) - https://en.wikipedia.org/wiki/Heap_(data_structure), accessed in 05/29/2021.