Huffman Coding

Compression

For this problem, a priority queue was used for the encoding part. The difference between a normal queue is that values comes out in order by priority. It takes out an element with the current highest priority. The reasoning behind this that the algorithm dequeues two nodes with the highest priority (lowest probability). Before this, the following list comprehension counts the character frequency of the input:

```
freq = [(data.count(e), e) for e in sorted(set(data))]
# Output  [(frquency, character), (frquency, character) ...]
```

The remaining node in the queue is the root of a Huffman tree. Last but not least, we need to get the Huffman codes. The function `_get_codes()` traverses the tree recursively and returns a dictionary containing those codes.

Decompression

The algorithm translates the input of prefix codes by traversing the huffman tree. It walks down the tree node by node for each bit until it reaches a leaf node. If there are more remaining bits left, it jumps back to the root of the tree and does it again.

Time and Space Complexity

A Priority queue is an abstract datastructure. Here, we use the Python's built-in queue module. According to the docs, the actual datastructure is some sort of a heap. So, the time complexity for enqueue and dequeue operations is both $O(logn)$. Also, the character frequencies needs to be sorted:

```
for f, v in sorted(char_freq): # Sorted() O(N)
        entry_count += 1
        q.put(Huffman_Node(v, f, entry_count)) # Enqueu O(log n)
```

The overall time complexity becomes $O(n) \cdot O(logn)$, which is $O(nlogn)$

The space complexity is $O(n)$

Please note that `n` is the number of characters (including whitespaces)