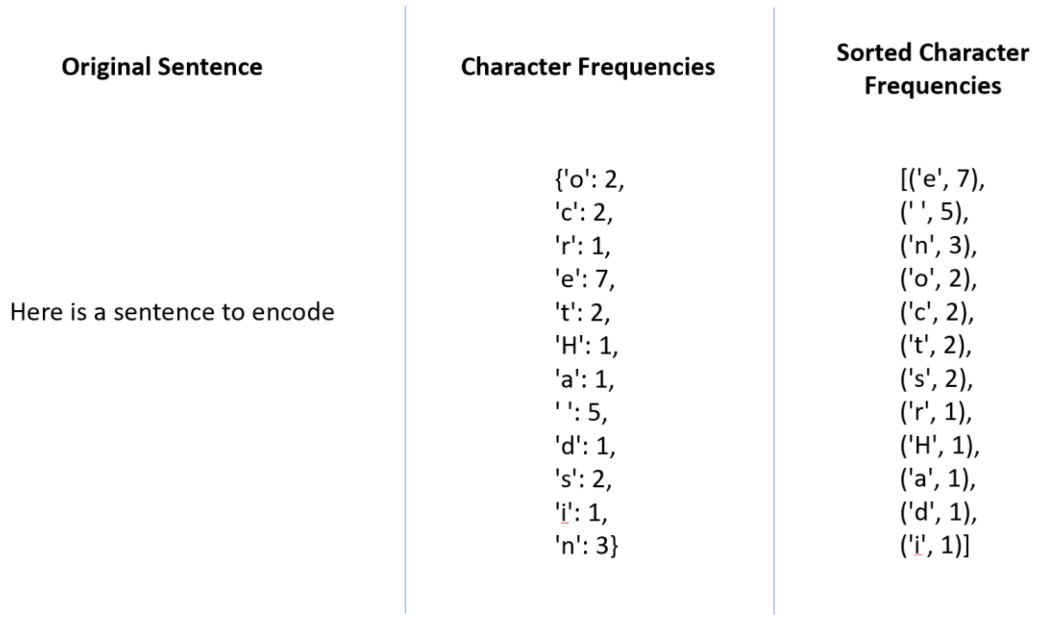
Problem 3

In order to gather the frequency of the characters, I used a dictionary where the keys are the character and the value is the frequency count of the character. Scanning through the input string to gather the frequencies is O(n). I chose to use a dictionary since getting/setting values while gathering the frequencies with a dictionary is O(1). I then sort the list so that it is easier to find the two least frequent entries. Sorting is O(n*log(n)).



Original Sentence	**Character Frequencies**	**Sorted Character Frequencies**
	{'o': 2,	[('e', 7),
	'c': 2,	(' ', 5),
	'r': 1,	('n', 3),
	'e': 7,	('o', 2),
Here is a sentence to encode	't': 2,	('c', 2),
	'H': 1,	('t', 2),
	'a': 1,	('s', 2),
	' ': 5,	('r', 1),
	'd': 1,	('H', 1),
	's': 2,	('a', 1),
	'i': 1,	('d', 1),
	'n': 3}	('i', 1)]

Figure 1: Character Frequencies

I take the two lowest frequencies, then build a node with the left and right children being those two frequencies and the node value being the sum of those two children. I store the created nodes in a dictionary and store the node's sum back into the list of frequencies so that it can be combined. Each time I store the node's sum back into the list of frequencies, I need to resort the list so that I can get the next two least frequent entries in the updated list.

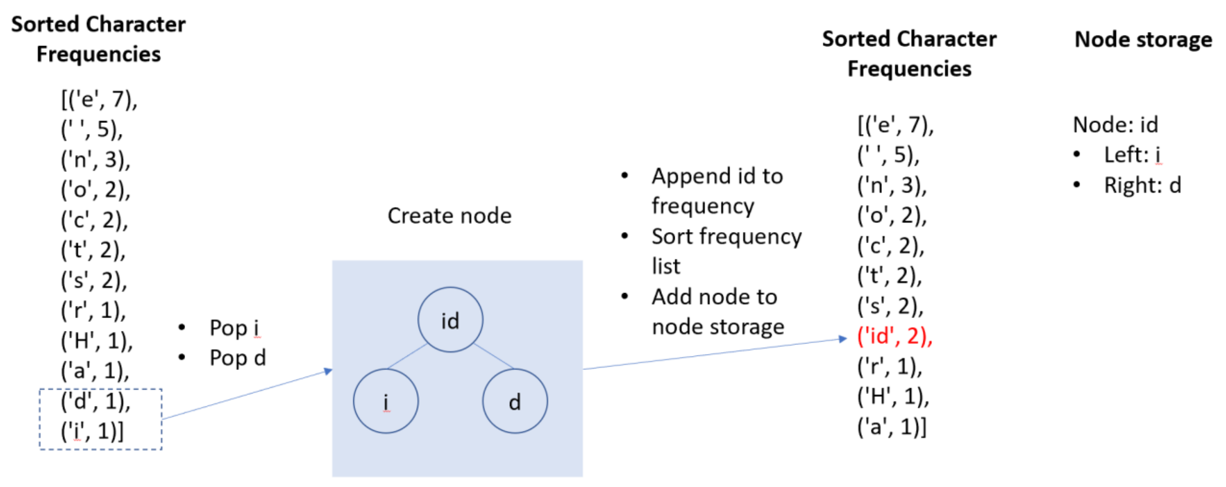


Figure 2: Creating nodes from frequencies

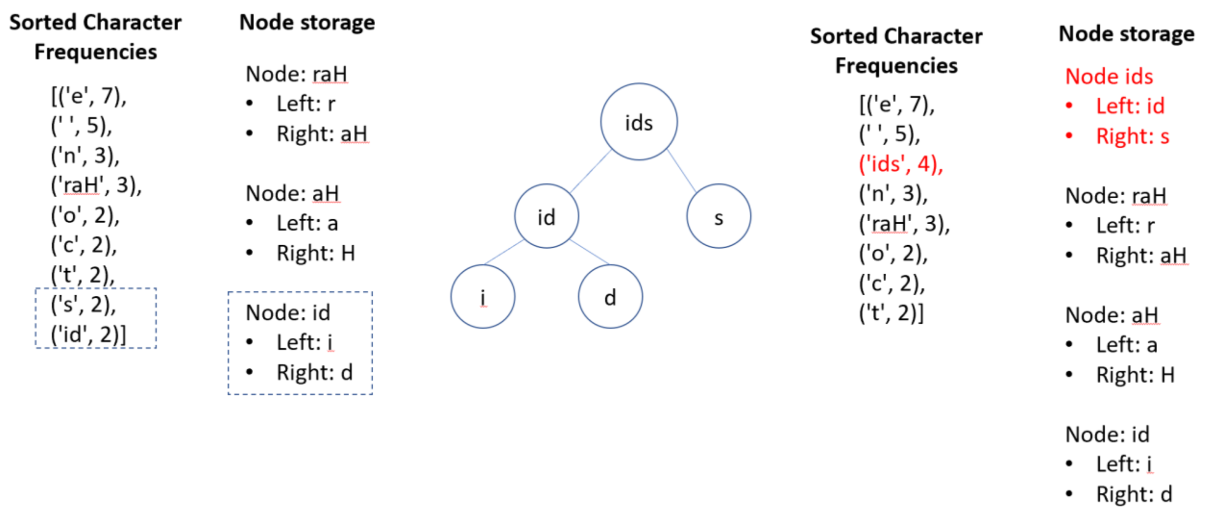


Figure 3: Continuing to create nodes from frequencies

The sorting happens with each loop so I believe this would be O(n^2*log(n)), which I believe is the worst-case efficiency for this code

Eventually, there will be one node left that will become the root of the encoding tree.

The encoding is generated by traversing the tree in a breadth-first approach and adding a code based on if choosing the left or right node during the traversal. In order to implement the breadth-first approach, a queue is used to correctly order the nodes for traversal. I chose to use breadth-first because it would help order the list so that you would have characters with shorter codes first and characters with longer codes as you traverse deeper in the tree. I thought it would be good to find/replace the longer codes first when decoding. In order to encode, I do a string replace for each character code I built. For decoding, I do the same process but in reverse, replacing the code with a character. I initially had issues here with decoding but giving the root (01), left (0) and right (11) nodes unique codes helped with the decoding so that I wasn’t incorrectly replacing the values with the wrong characters.

In terms of space complexity, I store the characters with their frequencies as a dictionary, which will depend on the number of unique characters within the sentence to be encoded. While running the algorithm, I am also storing the created nodes in a dictionary, which is also dependent on the number of unique characters in the original sentence. Storing these intermediate values was helpful in executing this algorithm, which is dependent on results from previous steps.