Archie Ross
3/20/23
Kloub CSE 3500

This file is attached to my HuskyCT submission but here is a screenshot in case of any problems or for ease of reference.

```python
import heapq

def frequency(file):
    # create a dictionary to efficiently store the frequency of each nucleotide
    frequency_dictionary = {'A': 0, 'C': 0, 'G': 0, 'T': 0}

    # open the text file and read its contents
    with open(file, 'r') as f:
        contents = f.read().upper()

    # iterate through each nucleotide in the contents, and increment the count for its corresponding key in the dictionary
    for nucleotide in contents:
        if nucleotide in frequency_dictionary:
            frequency_dictionary[nucleotide] += 1

    return frequency_dictionary


def huffman(dictionary):
    # create a heap list of the frequency dictionary items, where each item is a list [frequency, [nucleotide, code]]
    heap = [[amount, [nucleotide, ""]] for nucleotide, amount in dictionary.items()]

    # heapify the list so that it can be used to create the Huffman tree
    heapq.heapify(heap)

    # repeatedly extract the two lowest frequency items from the heap, merge them into a new tree node, and re-insert the tree node into the heap
    while len(heap) > 1:
        low = heapq.heappop(heap)
        high = heapq.heappop(heap)

        for pair in low[1:]:
            pair[1] = '0' + pair[1]

        for pair in high[1:]:
            pair[1] = '1' + pair[1]

        heapq.heappush(heap, [low[0] + high[0]] + low[1:] + high[1:])

    # the last item remaining in the heap is the Huffman tree, represented as a list where the first element is the total frequency
    # and the rest of the elements are the nucleotides and their binary codes
    tree = heapq.heappop(heap)

    # create a dictionary from the binary codes in the tree, where the nucleotides are the keys and the codes are the values
    bin = dict(tree[1:])

    return tree, bin


def encode(sequence, codes):
    # create an empty string to store the binary encoding of the sequence
    encoded = ""

    # iterate through each nucleotide in the sequence, and append its binary code to the encoded string
    for nucleotide in sequence:
        encoded += codes[nucleotide]

    return encoded


def compress_file(input_file, output_file):
    # get the frequency of nucleotides in the input file
    frequency_of = frequency(input_file)

    # create the Huffman tree and binary codes from the frequency dictionary
    tree, bin = huffman(frequency_of)

    # open the input file and read its contents
    with open(input_file, 'r') as i:
        sequence = i.read().upper()

        # encode the sequence using the binary codes
        encoded = encode(sequence, bin)

    # write the encoded binary string to the output file
    with open(output_file, 'w') as o:
        o.write(encoded)


if __name__ == '__main__':
    compress_file("dna.txt", "compressed.txt")
```

This is the output of the compressed.txt file when the code is ran with dna.txt as the inputted file:

000100101011011010010101101111000100101011011010010101101111000100101011011010
010101101111000100101011011010010101101111000100101011011010010101101111000100
101011011010010101101111000100101011011010010101101111000100101011011010010101
101111000100101011011010010101101111000100101011011010010101101111000100101011
011010010101101111000100101011011010010101101111000100101011011010010101101111
000100101011011010010101101111000100101011011010010101101111000100101011011010
010101101111000100101011011010010101101111000100101011011010010101101111000100
101011011010010101101111000100101011011010010101101111000100101011011010010101
101111000100101011011010010101101111000100101011011010010101101111000100101011
011010010101101111000100101011011010010101101111000100101011011010010101101111
000100101011011010010101101111000100101011011010010101101111000100101011011010
010101101111000100101011011010010101101111

If you used fixed-length code to encode the message, how many bits do you need per character? For a fixed length code we would only really need 2 bits to represent 4 possibilities. In basic binary this makes the most sense but written out it’s simple, A: 00, C: 01, G: 10, T: 11. Thus for any example we’d encode an entire message in n*2 total bits.

If you used fixed-length code to encode the message, how many bits do you need to encode this message? Using the idea we just described, we’d only need n*2 bits to encode the message, this means that for the total characters (510) we’d need 1020 bits.

What are the codes for each character after using Huffman coding?
For the sequence, I believe the codes come out to A: 0, C: 100, G: 101, T: 11 where the most frequent characters get shorter binary codes.

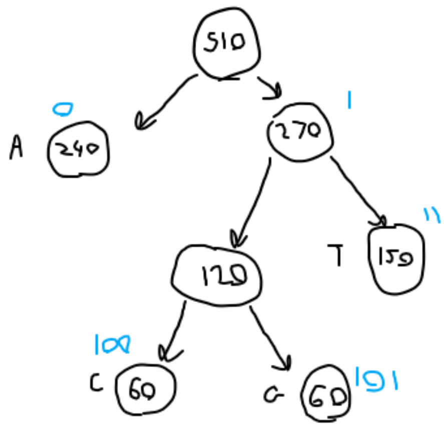
What is the average number of bits per character after using Huffman coding? Show your work.
So for the average number of bits per character we have to take the total number of bits being used divided by the values they uphold. We can basically allude this to 9/4 which gives us an average of 2.25. This response gives us a general consensus of the average we’d get given we don’t know the frequency truly occurring in the input. Yet, we have access to dna.txt which means we also can verify the frequency of the occurrences. So if we take (240/510) for A, (60/510) for C, (60/510) for G, (150/510) for T, and multiply those by the number of bits they use to represent themselves, we get the equation

1*(240/510) + 3*(60/510) + 3*(60/510) + 2*(150/510)

which gives us approximately 1.76 bits per character.

How many bits do you need to encode this message? Show your work.
Using what we found in the previous question we can find the number of bits to encode the message. We simply multiply the average times the total number of characters encoded (1.76*510) which gives us 898 which is relatively close to the 900 characters that were developed when the code was ran (in compressed.txt)

Create a Huffman encoding tree for these four characters. Show your work.



A = 0
C = 100
G = 101
T = 11

510 - MOST FREQ = 270 - MOST FREQ (NOT A) = 120 - MOST FREQ

240 (A) 150 (T) C = G