Huffman Analysis

Question 1

Suppose you want to compress two different files: `fileA` and `fileB`. Both have N total characters and M unique characters. The characters in `fileA` follow a uniform distribution, meaning each of the unique characters appears N/M times. In `fileB`, the i'th unique character appears 2^i times (and the numbers add up to N), so some characters are much more common than others. Which file should achieve a higher compression ratio? Explain your answer.

`fileB` will have a higher compression ratio. Since `fileB` has characters that occur with a higher frequency, these characters will be accessed with paths shorter than the same characters in `fileA`, where the weights are the same. Therefore, the compressed file for `fileB` will comprise less bits than the compressed file for `fileA`. Therefore, `fileB` should achieve a higher compression ratio.

Question 2

What is the asymptotic runtime complexity of `compress` as a function of N and/or M? Explain your answer, referencing the algorithm / implementation. Be sure to account for all three parts of compression in your explanation: (1) determining counts of characters, (2) creating the Huffman coding tree, and (3) writing the encoded file.

`getCounts():` loop runs `N` times. Returns an array of size 256. Therefore, it has a complexity of $O(N)$

`makeTree():` $O(\log M + M \log M + 1) = O(M \log M)$

```
private HuffNode makeTree(int[] counts) {
    PriorityQueue<HuffNode> pq = new PriorityQueue<>();
    for (int i = 0; i < counts.length; i++) { // Runs 256 times
      if (counts[i] > 0)
        pq.add(new HuffNode(i, counts[i], null, null)); // O(log M)
    }

    pq.add(new HuffNode(PSEUDO_EOF, 1, null, null)); // O(1)

    while (pq.size() > 1) { // runs M times
      HuffNode left = pq.remove();  // O(1)
      HuffNode right = pq.remove(); // O(1)
      HuffNode t = new HuffNode(0, left.weight + right.weight, left, right);
      pq.add(t); // O(log M)
    }
    HuffNode root = pq.remove(); // O(1)

    return root;
  }
```

`writeTree():` From the code, the following recurrence relation can be obtained: $T(M) = T(M/2) + O(1)$. This means that `writeTree` is $O(\log M)$.

`makeEncodings():` Results in the same recurrence relation. Thus, it is $O(\log M)$.

`compress():` Doing a line-by-line analysis of the code, we obtain that the runtime complexity of `compress` is $O(N + M \log M + \log M + 1) = O(N + M \log M)$

```
public void compress(BitInputStream in, BitOutputStream out){
    int[] counts = getCounts(in); // O(N)
    HuffNode root = makeTree(counts); // O(M log M)
    in.reset();
    out.writeBits(BITS_PER_INT, HUFF_TREE);
    writeTree(root, out); // O(log M)
    String[] encodings = new String[ALPH_SIZE+1];
    makeEncodings(root, "", encodings); // O(log M)

    int readChar = in.readBits(BITS_PER_WORD);

    while (readChar != -1) { // runs N times
```

```
      String code = encodings[readChar]; // O(1)
      out.writeBits(code.length(), Integer.parseInt(code,2)); // O(1)
      readChar = in.readBits(BITS_PER_WORD); // O(1)
    }

    String eof = encodings[PSEUDO_EOF];
    out.writeBits(eof.length(), Integer.parseInt(eof,2));

    out.close();
  }
```

Question 3

When running `decompress`, each character that is decompressed requires traversing at most M nodes in the Huffman coding tree, and there are N such characters. This analysis would at first suggest that the asymptotic runtime complexity of `decompress` is O(MN). However, you are unlikely to experience this in practice; this estimate is too simple and pessimistic. To see why, answer the following examining two different extreme cases:

- **First consider the case where, like `fileA` in question 1, every unique character appears N/M times. Then what would the asymptotic runtime complexity of `decompress` be?**

  Every leaf in a tree where each character has the same frequency has a depth of $\log M$. A leaf has to be reached for every character. There are a total of N characters. Therefore, the asymptotic runtime complexity of `decompress` would be $O(N \log M)$.

- **Now consider the case like `fileB` where the i'th unique character appears 2^i times (and the numbers add up to N). Would the runtime complexity be better or worse than for `fileA`? You do not need to derive the asymptotic runtime complexity exactly, just compare to the answer for `fileA`. *Hint*: Recall your answer to question 1 and observe the relationship between the runtime**

complexity of `decompress` and the *number of bits* in the data being decompressed.

In Question 1, it was determined that `fileB` will have a higher compression ratio because the compressed file will contain a smaller number of bits than the compressed file for `fileA`. Since the number of bits for the compressed file for `fileB` will be lower than the number of bits for the compressed file for `fileA`, the `decompress` method will have to process less bits. Thus, the runtime complexity for `fileB` would be better than for `fileA`.