Question 1

- File B should have a significantly higher compression ratio than file A. This is because the Huffman compression algorithm is more effective on text documents which have an uneven distribution of characters. Huffman compression assigns shorter binary codes to the most frequent characters, and the tradeoff for this is longer binary codes for the least common characters. These tradeoffs are always favorable for the compression ratio. In the example of File A, the characters each have a uniform distribution, so there will not be any meaningful differences in character encoding lengths and no data saving character length tradeoffs.

Question 2

```
65      public void compress(BitInputStream in, BitOutputStream out){
66
67
68          int[] counts = getCounts(in);
69          HuffNode root = makeTree(counts);
70          in.reset();
71          out.writeBits(BITS_PER_INT, HUFF_TREE);
72          writeTree(root, out);
73          String[] encodings = new String[ALPH_SIZE + 1];
74          makeEncodings(root, path: "", encodings);
75
76          writeCompressedBits(encodings, in, out);
77
78
79          out.close();
80      }
```

- We will examine the runtime complexity of (1) determining counts of characters, (2) creating the Huffman coding tree, and (3) writing the encoded file in order to find the overall runtime complexity of compress.

```
68          int[] counts = getCounts(in);
69          HuffNode root = makeTree(counts);
70          in.reset();
71          out.writeBits(BITS_PER_INT, HUFF_TREE);
72          writeTree(root, out);
73          String[] encodings = new String[ALPH_SIZE + 1];
74          makeEncodings(root, path: "", encodings);
75
76          writeCompressedBits(encodings, in, out);
77
78
79          out.close();
80      }
81
82      //Helper method
83      private int[] getCounts(BitInputStream in){
84          int[] count = new int[ALPH_SIZE + 1];
85          for (int i = 0; i < count.length; i++) {
86              count[i] = 0;
87          }
88          while (true) {
89              int bits = in.readBits(BITS_PER_WORD);
90              if (bits == -1) {
91                  break;
92              } else {
93                  count[bits]++;
94              }
95
96          }
97          return count;
98
99      }
```

- For part 1, getCounts() is called once. Inside the method, an integer array is made and in constant time. Then a forloop runs on line 85 M times. Line 86 assigns a value for an index of an array in constant time. The while loop started on line 88 runs N times. readBits() runs in constant time. The code on lines 91 and 93 is also constant time. The resulting runtime complexity is O(N + M) for getCounts().

```
101     //helper method
102     private HuffNode makeTree(int[] counts){
103         PriorityQueue<HuffNode> pq = new PriorityQueue<>();
104         for(int i = 0; i < counts.length; i++) {
105             if (counts[i] > 0) {
106                 pq.add(new HuffNode(i,counts[i],ltree: null,rtree: null));
107             }
108         }
109         pq.add(new HuffNode(PSEUDO_EOF,count: 1,ltree: null,rtree: null)); // account for PSEUDO_EOF having
110
111         while (pq.size() > 1) {
112         HuffNode left = pq.remove();
113         HuffNode right = pq.remove();
114         // create new HuffNode t with weight from
115         // left.weight+right.weight and left, right subtrees
116         HuffNode t = new HuffNode(left.value + right.value, left.weight + right.weight, left, right);
117         pq.add(t);
118         }
119         return pq.remove();
120     }
```

- For part 2, makeTree() is called with the parameter counts. A priority queue is initialized in constant time. A forloop runs M times and inside the loop, a value is added to the priority queue which is log(M) complexity. Then another value is added in log(M) complexity once. Then a while loop runs ~M times, and inside of it values are added and removed from a priority queue, which have log(M) complexity. Finally a node is removed, which has log(M) complexity. The resulting runtime complexity of makeTree() is O(Mlog(M)).

```java
//helper method
private void writeTree(HuffNode root, BitOutputStream out){
    if (root == null) return;
    if (root.left == null && root.right == null){
        out.writeBits(numBits: 1, value: 1);
        out.writeBits(BITS_PER_WORD+1, root.value);
        return;
    }

    out.writeBits(numBits: 1,value: 0);
    writeTree(root.left, out);
    writeTree(root.right, out);

}
```

```java
//helper method
private void makeEncodings(HuffNode root, String path, String[] encodings){
    if (root == null) {
        return;
    }
    if (root.right == null && root.left == null) {
        encodings[root.value] = path;
        return;
    }

    makeEncodings(root.left, path + "0", encodings);
    makeEncodings(root.right, path + "1", encodings);
}
```

```java
private void writeCompressedBits(String[] encodings, BitInputStream in, BitOutputStream out){
    while(true){
        int bit = in.readBits(BITS_PER_WORD);
        if(bit == -1){
            break;
        }

        String code = encodings[bit];
        out.writeBits(code.length(), Integer.parseInt(code, radix: 2));
    }
    String pseudo = encodings[PSEUDO_EOF];
    out.writeBits(pseudo.length(), Integer.parseInt(pseudo, radix: 2));
}
```

- For part 3, the runtime complexity of writeTree() is O(M), because it runs for each unique value in the tree, and the runtime complexity of the non-recursive methods is O(1). Make encodings also has runtime complexity O(M), because it encodes M unique characters, and the non-recursive methods are O(1). The writeCompressedBits() method contains a while loop that loops through all N characters, and all of the code inside of the loop is O(1).
- The resulting big-O runtime complexity of the compress method is O(N+Mlog(M)).

Question 3

- In a file where each unique character has an even distribution, the huffman tree is symmetrical. The decompress method will then have to loop through log(M) nodes N times. The frequency of each unique character is N/M, so the big-O runtime complexity for fileA would be O(Nlog(N)).
- In fileB, there is a very uneven distribution of unique characters, so half of the characters would have an encoding that is one edge away from the root node, a quarter would have an encoding two edges away and so on. This means that the vast majority of characters in the compressed

string will be decompressed in significantly less than M constant time operations. This means that the overall runtime complexity on this file will be significantly better than O(NM).