TL;DR

Here is the TL;DR for the Huffman assignment. You'll need to read this current document as well. See the discussion section document for April 15 as well.

Overview and Background (not required)

There are many techniques used to compress digital data. This assignment covers Huffman Coding.

Huffman coding was invented by David Huffman while he was a graduate student at MIT in 1950 when given the option of a term paper or a final exam. For details see this 1991 Scientific American Article. In an autobiography Huffman had this to say about the epiphany that led to his invention of the coding method that bears his name:

"-- but a week before the end of the term I seemed to have nothing to show for weeks of effort. I knew I'd better get busy fast, do the standard final, and forget the research problem. I remember, after breakfast that morning, throwing my research notes in the wastebasket. And at that very moment, I had a sense of sudden release, so that I could see a simple pattern in what I had been doing, that I hadn't been able to see at all until then. The result is the thing for which I'm probably best known: the Huffman Coding Procedure. I've had many breakthroughs since then, but never again all at once, like that. It was very exciting."

The Wikipedia reference is extensive as is this online material developed as one of the original Nifty Assignments. Both jpeg and mp3 encodings use Huffman Coding as part of their compression algorithms. In this assignment you'll implement a complete program to compress and uncompress data using Huffman coding.

You should read about the Huffman algorithm, you won't be able to complete project easily without the overview you'll get from this reading: https://www.cs.duke.edu/csed/poop/huff/info/ .

When you've read the description of the algorithm and data structures used you'll be ready to implement both decompression (aka uncompressing) and compression using Huffman Coding. You'll be using input and output or I/O classes that read and write 1 to many bits at a time, i.e., a single zero or one to several zeros and ones. This will make debugging your program a challenge.

Acknowledgments and History

We first gave a Huffman coding assignment at Duke in Spring of 1994. Over the years many people have worked on creating the infrastructure for the bit-reading and -writing code (as we changed from C to C++ to Java at Duke) and the GUI that drives the Huffman assignment. It was one of the original so-called "nifty assignments" (see http://nifty.stanford.edu) in 1999. In Fall of 2018 we moved away from the GUI and using a simple main and the command-line. This was done for pragmatic and philosophical reasons.

Git Repositories

Fork, clone, and import the cloned project from the file system. Use this URL from the course GitLab site: https://coursework.cs.duke.edu/201spring19/huffman-start ***. Be sure to fork first*** (see screen shot). Then Clone using the SSH URL after using a terminal window to cd into your Eclipse workspace. Recall that ***you should import using*** the File>Import>General>Existing Projects into Workspace -- then navigate to where you cloned the diyad project.

DO NOT DO NOT import using the Git open -- use General>Existing Projects Into Workspace.

Pushing to Git

When you make a series of changes you want to 'save', you'll push those changes to your GitLab repository. You should do this after major changes, certainly every hour or so of coding. You'll need to use the standard Git sequence to commit and push to GitLab:

```
git add .
git commit -m 'a short description of your commit here'
git push
```

Partners

You may work with a partner ***from your discussion section*** on this assignment. One person should fork-and-clone from the GitLab repo. That person will add the other person/partner as a collaborator on the project. For full information, see the documentation here: https://docs.gitlab.com/ee/user/project/members/

Choose Settings>Members>Invite Members. Then use the autocomplete feature to invite your partner to the project. Both of you can clone and push to this project.

1. First, one person should create the GitLab repository then add the partner as a maintainer to the project.
2. Both students should clone the same repository and import it into Eclipse.
3. After both students have cloned and imported, one person should add a comment to the **`LinkStrand.java`** class with their name in a comment at the start of the file. Commit and push this change.
4. The other partner will then use the command line and issue a **`git pull`** request. Simply use the command-line and type:

```
git pull
```

5. After this command, right-click on the project in Eclipse and choose the Refresh menu item. You should see the modified **`LinkStrand.java`** file with a new comment. Add your name in a comment, then commit and push. The other person will need to issue a **`git pull`** to get that file.

As long as partners are modifying different files, this process works seamlessly. Modifying the same file can lead to issues in resolving conflicts. Git will deal with this with your help, but it's better to take turns in working on the same file, or to work on different files within the project. ***Ideally you'll always be physically together when working on the project.***
When submitting you'll use the partner/team in Gradescope. That's described below in XXYY.

There is really only one file in this project: HuffProcessor.java. ***Work together, side-by-side, if you have a partner.***

Overview of Programming

You're given two main programs: **`HuffMainDecompress.java`** and **`HuffMainCompress.java`**. These are very similar, but call different methods in **`HuffProcessor`**: **`decompress`** and **`compress`**, respectively. For the programs to run you'll need to implement methods in HuffProcess.java. ***Do this in the order shown below!***

1. First implement **`HuffProcessor.decompress`** and verify that you can decompress three already-compressed files provided in the assignment. You'll likely implement helper methods as part of implementing **`decompress`**.
2. Second, and only after implementing **`decompress`**, implement **`HuffProcessor.compress`**. You'll likely implement helper methods as part of implementing compress.

To verify that decompress works, run **`HuffMainDecompress`** three times and specify ***hidden1.txt.hf***, ***hidden2.txt.hf***, and ***mystery.tif.hf*** as the input file for the three different runs. You can decompress these to the output files ***hidden1.txt***, ***hidden2.txt***, and ***mystery.tif*** , for example. The text files should be recognizable as text related to Compsci 201. Opening ***mystery.tif*** should show a black and white image of a frog. You can see what these files should be when uncompressed by comparing them to the files ***h1.txt***, ***h2.txt***, and ***m1.tif***. You can use the Diff.java file to see if your decompressed file is bit-for-bit identical to what's expected.

Then turn to compress --- to verify that compress works, you'll compress a file F, say to F.hf. You'll then decompress F.hf --- say to a file G. You should see that F and G are exactly the same, bit-for-bit. Since you've already verified that **`decompress`** works, this will help you finish the project. You'll find information below on using the command-line program **`diff`** to compare two files to determine if they are the same, or the Java program **`Diff.java`** to do this.

You'll start (via Git) with versions of **`compress`** and **`decompress`** that simply copy the input to the output. You'll replace these methods with working version for decompression and compression. ***You should implement decompress first***.

Implementing **`HuffProcessor.decompress`**

There are four conceptual steps in decompressing a file that has been compressed using Huffman coding:

1. Read the 32-bit "magic" number as a check on whether the file is Huffman-coded (see lines 156-159 below)
2. Read the tree used to decompress, this is the same tree that was used to compress, i.e., was written during compression (helper method call on line 161 below).
3. Read the bits from the compressed file and use them to traverse root-to-leaf paths, writing leaf values to the output file. Stop when finding **`PSEUDO_EOF`** (helper method call on line 162 below).
4. Close the output file (line 163 below).

```
154    public void decompress(BitInputStream in, BitOutputStream out){
155
156        int bits = in.readBits(BITS_PER_INT);
157        if (bits != HUFF_TREE) {
158            throw new HuffException("illegal header starts with "+bits);
159        }
160
161        HuffNode root = readTreeHeader(in);
162        readCompressedBits(root,in,out);
163        out.close();
164    }
```

You may choose to implement some of these steps using helper methods, e.g., steps two and three above. This could lead to a version of decompress similar to the one shown. You do not have to use this code, but it illustrates how to use the bit-stream classes and how to throw appropriate exceptions.

As you can see, a **`HuffException`** is thrown if the file of compressed bits does not start with the 32-bit value **`HUFF_TREE`**. Your code should also throw a **`HuffException`** if reading bits ever fails, i.e., the **`readBits`** method returns -1. That could happen in the helper methods when reading the tree and when reading the compressed bits.

Reading the Tree

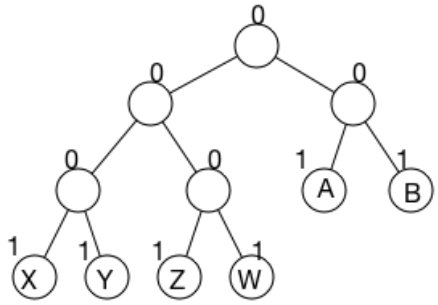
Reading the tree using a helper method is required since reading the tree, stored using a pre-order traversal, requires recursion. You don't have to use the names or parameters described above, though you can.

In this assignment, interior tree nodes are indicated by the single bit 0 and leaf nodes are indicated by the single bit 1. No values are written for internal nodes and a 9-bit value is written for a leaf node. This leads to the following pseudocode to read the tree.

```
public HuffNode readTree(...) {
     bit = read a single bit
     if (bit == -1) throw exception
     if (bit == 0) {
         left = readTree(...)
         right = readTree(...)
         return new HuffNode(0,0,left,right);
     }
     else {
         value = read nine bits from input
         return new HuffNode(value,0,null,null);
     }
}
```

For example, consider this bit sequence

`0001X1Y01Z1W01A1B`



Each letter represents a 9-bit sequence to be stored in a leaf as shown in the tree to the right. You'll read these 9-bit chunks with an appropriate call of **`readBits`**. Rather than use 9, you should use **`BITS_PER_WORD + 1`**. A 9-bit sequence represents a "character" stored in each leaf. This character, actually a value between

0-255, will be written to the output stream when uncompressing. One leaf stores **`PSEUDO_EOF`**, this won't be printed, but will stop the process of reading bits.

Note that when you read the first 0, you know it's an internal node (it's a 0), you'll create such a node, and recursively read the left and right subtrees as shown in the pseudocode above. The left subtree call will read the bits 001X1Y01Z1W as the left subtree of the root and the right subtree recursive call will read 01A1B as the right subtree. Note that in the bit-sequence representing the tree, a single bit of 0 and 1 differentiates INTERNAL nodes from LEAF nodes, not the left/right branch taken in uncompressing that comes later. The internal node that's the root of the left subtree of the overall root has its own left subtree of 01X1Y and a right subtree of 01Z1W. When you read the single 1-bit, your code will need to read 9-bits to obtain the value stored in the leaf. See the pseudocode for details.

Reading Compressed Bits

Once you've read the bit sequence representing tree, you'll read the remaining bits from the **`BitInputStream`** representing the compressed file one bit at a time, traversing the tree from the root and going left or right depending on whether you read a zero or a one. This is represented in the pseudocode for **`decompress`** by the helper method **`readCompressedBits`**.

The pseudocode from https://www.cs.duke.edu/csed/poop/huff/info/ is reproduced here, this is the same code shown in that document -- it's not perfect Java, hence pseudocode. (Note: you break when reaching **`PSEUDO_EOF`**, so when you write a leaf value to the output stream, you write 8, or **`BITS_PER_WORD`** bits).

(continued on following page)

```
HuffNode current = root;
while (true) {
    int bits = input.readBits(1);
    if (bits == -1) {
        throw new HuffException("bad input, no PSEUDO_EOF");
    }
    else {
        if (bits == 0) current = current.left;
        else current = current.right;

        if (current is a leaf node) {
            if (current.value == PSEUDO_EOF)
                break;   // out of loop
            else {
                write 8-bits for current.value;
                current = root; // start back after leaf
            }
        }
    }
}
```

Implementing `HuffProcessor.compress`

There are five conceptual steps to compress a file using Huffman coding. You do not need to use helper methods for these steps, but for some steps helper methods are useful.

1. Determine the frequency of every eight-bit character/chunk in the file being compressed (see line 64 below).
2. From the frequencies, create the Huffman trie/tree used to create encodings (see line 65 below).
3. From the trie/tree, create the encodings for each eight-bit character chunk (see line 66 below).
4. Write the magic number and the tree to the beginning/header of the compressed file (see lines 68-69 below).
5. Read the file again and write the encoding for each eight-bit chunk, followed by the encoding for PSEUDO_EOF, then close the file being written (see lines 71-73 below).

```
62    public void compress(BitInputStream in, BitOutputStream out){
63
64        int[] counts = readForCounts(in);
65        HuffNode root = makeTreeFromCounts(counts);
66        String[] codings = makeCodingsFromTree(root);
67
68        out.writeBits(BITS_PER_INT, HUFF_TREE);
69        writeHeader(root,out);
70
71        in.reset();
72        writeCompressedBits(codings,in,out);
73        out.close();
74    }
```

You won't need to throw exceptions for the steps outlined. A brief description of each step follows. More details can be found in the explanation of the Huffman algorithm https://www.cs.duke.edu/csed/poop/huff/info/.

Determining Frequencies

Create an integer array that can store 257 values (use **`ALPH_SIZE + 1`**). You'll read 8-bit characters/chunks, (using **`BITS_PER_WORD`** rather than 8), and use the read/8-bit value as an index into the array, incrementing the frequency. The code you start with in compress (and decompress) illustrates how to read until the sentinel -1 is read to indicate there are no more bits in the input stream. ***You'll need explicitly set `freq[PSEUDO_EOF] = 1` for the array to indicate there is one occurrence of the value `PSEUDO_EOF`.***

Making Huffman Trie/Tree

You'll use a greedy algorithm and a priority queue of **`HuffNode`** objects to create the trie. Since **`HuffNode`** implements **`Comparable`** (using weight) the code you write will remove the minimal-weight nodes when **`pq.remove()`** is called as shown in the pseudocode below.

```
PriorityQueue<HuffNode> pq = new PriorityQueue<>();

for(every index such that freq[index] > 0) {
    pq.add(new HuffNode(index,freq[index],null,null);
}
```

```
while (pq.size() > 1) {
    HuffNode left = pq.remove();
    HuffNode right = pq.remove();
    // create new HuffNode t with weight from
    // left.weight+right.weight and left, right subtrees
    pq.add(t);
}
HuffNode root = pq.remove();
```

If you stored a frequency/count of 1 for **PSEUDO_EOF** in the array of frequencies the code above will create a trie in which **PSEUDO_EOF** is stored in some leaf node. You'll need to ***be sure that's the case, that PSEUDO_EOF is represented in the tree.*** As shown above, you should only add nodes to the priority queue for indexes/8-bit values that occur, i.e., that have non-zero weights.

Make Codings from Trie/Tree

For this you'll need a recursive helper method, similar to code you've seen in class for the LeafTrails APT problem. As shown in the example of compress above, this method returns an array of Strings such that **a[val]** is the encoding of the 8-bit chunk **val**. See the debugging runs at the end of this write-up for details. As with the LeafTrails APT, the recursive helper method will have the array of encodings as one parameter, a node that's the root of a subtree as another parameter, and a string that's the path to that node as a string of zeros and ones. The first call of the helper method might be as shown, e.g., in the helper method **makeCodingsFromTree**.

```
String[] encodings = new String[ALPH_SIZE + 1];
codingHelper(root,"",encodings);
```

In **codingHelper**, if root is a leaf, an encoding for the value stored in the leaf is added to the array, e.g.,

```
if (root is leaf) {
    encodings[root.myValue] = path;
    return;
}
```

If the root is not a leaf, you'll need to make recursive calls adding "0" to the path when making a recursive call on the left subtree and adding "1" to the path when making a

recursive call on the right subtree. Every node in a Huffman tree has two children. ***Be sure that you only add a single "0" for left-call and a single "1" for right-call. Each recursive call has a String path that's one more character than the parameter passed, e.g., path + "0" and path + "1".***

Writing the Tree

Writing the tree is similar to the code you wrote to read the tree when decompressing. If a node is an internal node, i.e., not a leaf, write a single bit of zero. Else, if the node is a leaf, write a single bit of one, followed by nine bits of the value stored in the leaf. This is a pre-order traversal: write one bit for the node, then make two recursive calls if the node is an internal node. No recursion is used for leaf nodes. You'll need to write 9 bits, or **`BITS_PER_WORD + 1`**, because there are possibly 257 values including **`PSEUDO_EOF`**.

Writing Compressed Bits

After writing the tree, you'll need to read the file being compressed one more time. As shown above, the **`BitInputStream`** ***is reset, then read again to compress it***. The first reading was to determine frequencies of every 8-bit chunk. The encoding for each 8-bit chunk read is stored in the array created when encodings were made from the tree. These encodings are stored as strings of zeros and ones, e..g., "010101110101". To convert such a string to a bit-sequence you can use **`Integer.parseInt`** specifying a radix, or base of two. For example, to write the encoding for the 8-bit chunk representing 'A', which has an ASCII value of 65, you'd use:

```
String code = encoding['A']
out.writeBits(code.length(), Integer.parseInt(code,2));
```

You'll use code like this for every 8-bit chunk read from the file being compressed. You must also write the bits that encode **`PSEUDO_EOF`**, i.e.,

```
String code = encoding[PSEUDO_EOF]
out.writeBits(code.length(), Integer.parseInt(code,2));
```

You'll write these bits after writing the bits for every 8-bit chunk. The encoding for **`PSEUDO_EOF`** is used when decompressing, ***so you'll need to write the encoding bits before the output file is closed.***

The `BitInputStream` and `BitOutputStream` Classes

These two classes are provided to help in reading/writing bits in (un)compressed files. They extend the Java `InputStream` and `OutputStream` classes, respectively. They function just like `Scanner`, except instead of reading in / writing out arbitrarily delimited "tokens", they read/write specified number of bits. Note that two consecutive calls to the `readBits` method **may return different results** since `InputStream` classes maintain an internal "cursor" or "pointer" to a spot in the stream from which to read -- and once read the bits won't be read again (unless the stream is reset).

The only methods you will need to interact with are the following:

1. `int BitInputStream.readBits(int numBits)`
   This method reads from the source the specified number of bits and returns an integer. Since integers are 32-bit in Java, the parameter `numBits` must be between 1 and 32, inclusive. **It will return -1 if there are no more bits to read.**
2. `void BitInputStream.reset()`
   This method repositions the "cursor" to the beginning of the input file.
3. `void BitOutputStream.writeBits(int numBits, int value)`
   This method writes the least-significant `numBits` bits of the `value` to the output file.

Diff.java or diff

There is a mac/unix command diff that compares two files and indicates if they are the same bit-for-bit or not. You can use diff on a Windows computer if you have the Unix/bash shell.

You type

```
diff foo.txt bar.txt
```

If ***the files are the same nothing is printed***. If the files are different there's an indication of where they are different. For huff, you'll likely use debugging print statements if files aren't the same. You can also run Diff.java, provided to you for this assignment. It will ask you to select two files (from the same folder). The program prompts you to select files and displays a message if the files are the same or different. You use command-click or control-click to choose a second file on Mac/Windows.

Print Debugging Levels

The class HuffProcessor can be constructed with a debug level, e.g.,

```
HuffProcessor hp  = new HuffProcessor(4);
HuffProcessor hp = new HuffProcessor(HuffProcessor.DEBUG_HIGH);
```

You can then write printing/debugging code in your helper methods that you can "turn off" by simply constructing the **`HuffProcessor`** with no value for debugging. In my code I have statements like the following:

```
if (myDebugLevel >= DEBUG_HIGH) {
    System.out.printf("pq created with %d nodes\n", pq.size());
}
```

Which prints something about the priority queue or

```
if (root.myLeft == null && root.myRight == null) {
    codings[root.myValue] = path;
    if (myDebugLevel >= DEBUG_HIGH) {
        System.out.printf("encoding for %d is %s\n",
                          root.myValue,path);
    }
    return;
}
```

Which prints encodings for each root-to-leaf path found. For the file small.txt which consists of the single line Compsci 201: Duke the full debugging output is shown below for my program that compresses the data. You don't need to replicate this, but you should have some debugging output. This file is small, so there's not much debugging output. There would be much more for a larger or image file.

```
Huffman Compress Main
chunk      freq
10    1
32    2
48    1
49    1
50    1
```

```
58   1
67   1
68   1
99   1
101  1
105  1
107  1
109  1
111  1
112  1
115  1
117  1
256  1
pq created with 18 nodes
encoding for 117 is 0000
encoding for 49 is 0001
encoding for 10 is 0010
encoding for 256 is 0011
encoding for 32 is 010
encoding for 109 is 0110
encoding for 101 is 0111
encoding for 107 is 1000
encoding for 105 is 1001
encoding for 111 is 1010
encoding for 50 is 1011
encoding for 67 is 1100
encoding for 48 is 11010
encoding for 58 is 11011
encoding for 112 is 11100
encoding for 99 is 11101
encoding for 68 is 11110
encoding for 115 is 11111
wrote magic number -87129599
wrote leaf for tree 117
wrote leaf for tree 49
wrote leaf for tree 10
wrote leaf for tree 256
wrote leaf for tree 32
wrote leaf for tree 109
```

```
wrote leaf for tree 101
wrote leaf for tree 107
wrote leaf for tree 105
wrote leaf for tree 111
wrote leaf for tree 50
wrote leaf for tree 67
wrote leaf for tree 48
wrote leaf for tree 58
wrote leaf for tree 112
wrote leaf for tree 99
wrote leaf for tree 68
wrote leaf for tree 115
read 144 bits pre-reset
67 wrote 12 for 4 bits
111 wrote 10 for 4 bits
109 wrote 6 for 4 bits
112 wrote 28 for 5 bits
115 wrote 31 for 5 bits
99 wrote 29 for 5 bits
105 wrote 9 for 4 bits
32 wrote 2 for 3 bits
50 wrote 11 for 4 bits
48 wrote 26 for 5 bits
49 wrote 1 for 4 bits
58 wrote 27 for 5 bits
32 wrote 2 for 3 bits
68 wrote 30 for 5 bits
117 wrote 0 for 4 bits
107 wrote 8 for 4 bits
101 wrote 7 for 4 bits
10 wrote 2 for 4 bits
wrote 3 for 4 bits PSEUDO_EOF
compress from small.txt to small.txt.HF
file: 144 bits to 312 bits
read 144 bits, wrote 309 bits
bits saved = -165
```

Analysis

No analysis is required. However, you should be able to answer these questions, and they may appear on the final.

1. Why did you implement decompress first?
2. What is the purpose of PSEUDO_EOF?
3. How can a compressed have more bits than the file being compressed? When does this happen?
4. What compresses more: image file or text files, why do you think this happens?

Submitting

You'll submit the code to Gradescope after pushing your program to GitHub. This is a partner/team project in Gradescope. ***If you're working with a partner only one person will submit and then will be able to add the other as partner. DO NOT make two separate submissions and attempt to merge them together: that will not work due to a Gradescope bug.***

Refer to this document for submitting to Gradescope with a partner.

Reflect

You can access the reflect: http://bit.ly/201spring19-huff-reflect

Grading

Points are awarded equally for compression and decompression. You'll get points for decompressing and compressing text and image files. These are 10 points each, for a total of 40 points awarded as follows.

Functionality	Grade
Decompress text	D
Decompress text and image	C
Decompress and compress text	B
Decompress and compress all	A