Problem 2

We begin, as always, by importing required libraries followed by defining any functions we are going to create for later use.

Library Imports

For this implementation, we will require Python's "*heapq*" library so that we can create a priority que.

In [1]:
```
import heapq
```

We will also need the "*csv*" library so that we can read and write CSV files.

In [2]:
```
import csv
```

Finally, we will need "*time*" library so that we can determine how long the compression and decompression processes run

In [3]:
```
import time
```

Now that we have imported all the required libraries, we can move on to defining ***OUR*** functions and subroutines.

Function Definitions

We will need several functions of our own to both allow us to endcode/decode using Huffman codes, as a well as to make the later code easier to read and write by moving simple and/or repeated tasks to subroutines of their own. These subroutines are

- File Reader (*for text files*)
- File Writer (*for text files*)
- File Reader (*for CSV files*)
- File Writer (*for CSV files*)
- Dictionary Extractor
- N-Gram Generator
- Freqency Counter
- Huffman Tree Maker
- Huffman Code Builder

We will also need an object class for

- Huffman Nodes

File Reader (*for text files*)

We start with the **File Reader** for text files. We will name it ***fileRDR*** and its code is

In [4]:
```
def fileRDR(filename):
    with open(filename, 'r') as myTextFileIn:
        myTextIn = myTextFileIn.read();

        myTextFileIn.close()


    return myTextIn
```

Testing

In order to test it, we define a string to have the same contents as those which occur in our *TestTextFile.txt* test file

In [5]:
```
testText = "This is a test text file"
```

Then we import the file's contents to another string

In [6]:
```
testTextIn = fileRDR("TestTextFile.txt")
```

Last, we check that they are the same and print the string if they are

In [7]:
```
if testText == testTextIn:
    print(testTextIn)
else:
    print("OOOPS!!!")
```

```
This is a test text file
```

Since the text file reader works, we move on to the next subroutine.

File Writer (*for text files*)

We continue with the **File Writer** for text files. We will name it ***fileWTR*** and its code is

In [8]:
```
def fileWTR(filename, strToWrite):
    with open(filename, 'w') as myTextFileOut:
        myTextFileOut.write(strToWrite)

        myTextFileOut.close()


    return None
```

Testing

We test this subroutine by writing our previously defined string **testText** to another file *TestTextFile2.txt* and then reading that new file back in with **fileRDR** and comparing the read result with the original string. Starting with the write

In [9]:
```
fileWTR("TestTextFile2.txt", testText)
```

we then read the new file back in

In [10]:
```
testTextIn2 = fileRDR("TestTextFile2.txt")
```

and check the see that they are the same

In [11]:
```
if testText == testTextIn2:
    print(testTextIn2)
else:
    print("OOOPS!!!")
```

```
This is a test text file
```

Since the text writer works, we move on to CSV file readers and writers.

File Reader (*for CSV files*)

Now, we will create a **File Reader** for CSV files. We will name it ***csvFileRDR*** and its code is

In [12]:
```
def csvFileRDR(filename):
    csvOUT = []

    with open(filename, 'r') as myCSVfileIn:
        csvReader = csv.reader(x for x in myCSVfileIn)

        for row in csvReader:
            temp = row
            csvOUT.append(temp)

        myCSVfileIn.close()


    return csvOUT
```

Testing

In order to test our new CSV file reader, we define a string array to have the same contents as those which occur in our *testCSV.csv* test file

In [13]:
```
testCSV = [['a', '1'], ['b', '2'], ['c', '3'], ['d', '4'], ['e', '5'], ['f', '6'],
['g', '7'], ['h', '8'], ['i', '9'], ['j', '10']]
```

We now import the file's contents to another array

In [14]:
```
testCSVin = csvFileRDR('testCSV.csv')
```

finally checking if they are equal to our previously defined array and printing the array if they are

In [15]:
```
testCOND = True

rowCTR = 0
for row in testCSV:
    colCTR = 0

    for col in row:
        tmpTest = testCSVin[rowCTR]
        temp = tmpTest[colCTR]

        if temp.strip() == col.strip():
            colCTR += 1
        else:
            testCOND = False
            break

    rowCTR += 1


if testCOND:
    print(testCSVin)
else:
    print("OOOPS!!!")
```

```
[['a', '1'], ['b', '2'], ['c', '3'], ['d', '4'], ['e', '5'], ['f', '6'], ['g', '
7'], ['h', '8'], ['i', '9'], ['j', '10']]
```

Since the CSV reader works, we move on to the CSV writer.

File Writer (*for CSV files*)

Now, we will create a **File Writer** for CSV files. We will name it ***csvFileWTR*** and its code is

In [16]:
```
def csvFileWTR(filename, arrayToWrite):
    with open(filename, 'w', newline='') as myCSVfileOut:
        csvWriter = csv.writer(myCSVfileOut, delimiter=',',
                               quotechar=' ', quoting=csv.QUOTE_MINIMAL)
                                  #dialect='excel')
                                  #delimiter=',', quotechar='|', quoting=csv.QUOTE_MI
NIMAL)

        for row in arrayToWrite:
            csvWriter.writerow(row)

        myCSVfileOut.close()

    return None
```

Testing

In order to test our new CSV file writer, we will use our previously defined **testCSV** string array and have the CSV file writer write it to the new file *testCSV2.csv*. Then we will read this newly written file in and compare it to the oringinal **testCSV**. Writing the new file

In [17]:
```
csvFileWTR('testCSV2.csv', testCSV)
```

Reading the newly written file into the new string array **testCSVin2**.

In [18]:
```
testCSVin2 = csvFileRDR("testCSV2.csv")
```

Finally checking if the **testCSV** and **testCSVin2** string arrays match, element for element.

In [19]:
```
testCOND = True

rowCTR = 0
for row in testCSV:
    colCTR = 0

    for col in row:
        tmpTest = testCSVin2[rowCTR]
        temp = tmpTest[colCTR]

        if temp.strip() == col.strip():
            colCTR += 1
        else:
            testCOND = False
            break

    rowCTR += 1


if testCOND:
    print(testCSVin2)
else:
    print("OOOPS!!!")
```

```
[['a', '1'], ['b', '2'], ['c', '3'], ['d', '4'], ['e', '5'], ['f', '6'], ['g', '
7'], ['h', '8'], ['i', '9'], ['j', '10']]
```

Dictionary Extractor

We will also need a subroutine to extract a dictionary of all the characters used by a specified text. Thus, we create the **dictExtractr** sub-routine to extract a character dictionary from the input String provided to it.

In [20]:
```
def dictExtractr(textIn):
    dictOut = list(set(textIn))

    return dictOut
```

Testing

To test our **dictExtractr** function, we will provide it with the previously defined string, **testText**, and a new string **testGophers**, which is defined as

In [21]:
```
testGophers = "go go gophers"
```

Testing on **testText** gives

In [22]:
```
print(testText)

testDict = dictExtractr(testText)
print(testDict)
```

```
This is a test text file
['x', 's', 'i', 'h', 't', 'a', 'f', 'T', ' ', 'l', 'e']
```

While testing on **testGophers** gives

In [23]:
```
print(testGophers)

testDict2 = dictExtractr(testGophers)
print(testDict2)
```

```
go go gophers
['g', 's', 'h', 'o', ' ', 'r', 'p', 'e']
```

Since the dictionary exractor works, we will new move on to the N-Gram Generator.

N-Gram Generator

Since we may wish to encode based on Bi-Grams, Tri-Grams, or some other type of N-Grams (*instead of just characters*), we need to write a routine to create N-Grams of the specified dimension (*N*) from a specified character dictionary. We call this function **nGramBuilder** and its code is

In [24]:
```
def nGramBuilder(nIn, dictIn):
    gramsOut = []

    if nIn == 1:
        gramsOut = dictIn
    else: #if nIn > 1:
        nOut = nIn - 1

        tempGrams = nGramBuilder(nOut, dictIn)

        for letter in dictIn:
            for gram in tempGrams:
                gramsOut.append(letter + gram)


    return gramsOut
```

Our character dictionary is used by one more function which we will define next

Frequency Counter

We need to know the frequency of characters from a given dictionary in a given document. Thus, we create the **freqCTR** sub-routine to determine these frequencies.

In [25]:
```
def freqCTR(dictIn, textIn):
    counts = []

    for x in dictIn:
        counts.append(textIn.count(x))


    return x
```

In [26]:
```
def freqCTR(dictIn, textIn):
    counts = {}

    for x in dictIn:
        counts[x] = textIn.count(x)


    return counts
```

Testing

We test this with the previously obtained **testDict** and **testDict2** dictionaries,

In [27]:
```
print(testDict)
print(testDict2)
```

```
['x', 's', 'i', 'h', 't', 'a', 'f', 'T', ' ', 'l', 'e']
['g', 's', 'h', 'o', ' ', 'r', 'p', 'e']
```

which were obtained from the previously defined **testText** and **testGophers** Strings,

In [28]:
```
print(testText)
print(testGophers)
```

```
This is a test text file
go go gophers
```

Testing on the **testDict** and **testText** pair, we have

In [29]:
```
testFreqs = freqCTR(testDict, testText)
print(testFreqs)
```

```
{'x': 1, 's': 3, 'i': 3, 'h': 1, 't': 4, 'a': 1, 'f': 1, 'T': 1, ' ': 5, 'l': 1,
'e': 3}
```

as expected. Similarly, testing on the **testDict2** and **testGophers** pair, we have

In [30]:
```
testFreqs2 = freqCTR(testDict2, testGophers)
print(testFreqs2)
```

```
{'g': 3, 's': 1, 'h': 1, 'o': 3, ' ': 2, 'r': 1, 'p': 1, 'e': 1}
```

as expected.

Now, before continuing with subroutines and functions, we need to define an object class for Huffman Nodes (*nodes in our Huffman tree(s)*).

Huffman Nodes

Our object for representing Huffman Nodes will be called the **HuffmanNode** class and must have the properties

- The character it represents: ***myChar***
  - *Data-Type:* ***char***
  - (*Default* = ***None***)
- The frequency of the character it represents: ***myFreq***
  - *Data-Type:* ***int***
  - (*Default = Not specified*)
- The left child of the node: ***myLeft***
  - *Data-Type:* ***HuffmanNode***
  - (*Default* = ***None***)
- The right child of the node: ***myRight***
  - *Data-Type:* ***HuffmanNode***
  - (*Default* = ***None***)

The **HuffmanNode** class must also have a method for comparing it to other instances of **HuffmanNode** and another method to allow an instance of **HuffmanNode** to determine if it is a leaf in a tree (*myLeft = None* ***and*** *myRight = None*). With all this in mind, we define our **HuffmanNode** class as follows

In [31]:
```
class HuffmanNode(object):
    def __init__(self, theFreq, theChar=None, theLeft=None, theRight=None):
        self.myChar = theChar
        self.myFreq = theFreq
        self.myLeft = theLeft
        self.myRight = theRight

    def __lt__(self, other):
        return self.myFreq < other.myFreq

    def isLeaf(self):
        return (self.myLeft == None  and  self.myRight == None)
```

Testing

We will test to ensure that our **HuffmanNode** class does the following

- Returns the proper values for
  - **myChar**
  - **myFreq**
  - **myLeft**
  - **myRight**
- Properly compares two instances of the class
- Properly determines if an instance is or is not a leaf

Starting with the value testing for **myChar** and **myFreq**, we define the values

In [32]:
```
aChar1 = "a"
aFreq1 = 10

aChar2 = "b"
aFreq2 = 6

aChar3 = "f"
aFreq3 = 4
```

which we use to define the following three instances of the **HuffmanNode** class

In [33]:
```
aHnode1 = HuffmanNode(aFreq1, aChar1)
aHnode2 = HuffmanNode(aFreq2, aChar2)
aHnode3 = HuffmanNode(aFreq3, aChar3)
```

We now have these three instances of the **HuffmanNode** class recall their specified values for *myChar* and *myFreq*

In [34]:
```
print(aHnode1.myChar)
print(aHnode1.myFreq)
```

```
a
10
```

In [35]:
```
print(aHnode2.myChar)
print(aHnode2.myFreq)
```

```
b
6
```

In [36]:
```
print(aHnode3.myChar)
print(aHnode3.myFreq)
```

```
f
4
```

Since these instances return their specified values correctly, we move on to checking if they compare themselves amongst each other properly

In [37]:
```
print(aHnode1.myFreq < aHnode2.myFreq)
print(aHnode1 < aHnode2)
```

```
False
False
```

In [38]:
```
print(aHnode1.myFreq < aHnode3.myFreq)
print(aHnode1 < aHnode3)
```

```
False
False
```

In [39]:
```
print(aHnode2.myFreq > aHnode3.myFreq)
print(aHnode2 > aHnode3)
```

```
True
True
```

Lastly, we need to check the *leaf* properties and the *isLeaf* method. To do this, we assign the second and third nodes as leaves of the first

In [40]:
```
aHnode1.myLeft = aHnode3
aHnode1.myRight = aHnode2
```

This allows us to first check if the first instances properly returns its leaves with

In [41]:
```
print(aHnode3.myChar)
print(aHnode1.myLeft.myChar)
aHnode3 == aHnode1.myLeft
```

```
f
f
```

Out[41]: `True`

for the left and

In [42]:
```
print(aHnode2.myChar)
print(aHnode1.myRight.myChar)
aHnode2 == aHnode1.myRight
```

```
b
b
```

Out[42]: `True`

for the right. Lastly, we check if the instances return the proper responses for the *isLeaf* method

In [43]:
```
aHnode1.isLeaf()
```

Out[43]: `False`

In [44]:
```
aHnode2.isLeaf()
```

Out[44]: `True`

In [45]:
```
aHnode3.isLeaf()
```

Out[45]: `True`

Since the **HuffmanNode** object class tests out properly, we move on to creating a method to build a Huffman Tree.

Huffman Tree Maker

Given some frequency data about the occurance of character in some text, where the data is in the form *{**char** (or str): **count**}*, we want a method which will create a corresponding Huffman Tree. Thus, the method must first convert each element of the provided data to its own instance of the **HuffmanNode** class; after which it sequentially builds a HuffmanTree (*itself a Huffman Node*) by successively combining the two nodes with the lowest frequency values into a new instance of a **HuffmanNode** which has these two nodes as children and a frequency equal to the sum of the frequencies of its children. Since we are working from the "*bottom*" of the pile of frequencies up, we will utilize the **heapify**, **heappop**, and **heappush** methods from the "*heapq*" library to allow us to implement this as a reverse priority que. Thus, we define the method **hTreeMakr** as follows

In [46]:
```
def hTreeMakr(theFreqData):
    myFreqData = theFreqData

    hNodes = []

    for char in myFreqData:
        hNodes.append(HuffmanNode(myFreqData[char], char))

    heapq.heapify(hNodes)

    while(len(hNodes) > 1):
        leftLeaf = heapq.heappop(hNodes)
        rightLeaf = heapq.heappop(hNodes)

        newFreq = leftLeaf.myFreq + rightLeaf.myFreq

        newNode = HuffmanNode(newFreq, theLeft = leftLeaf, theRight = rightLeaf)

        heapq.heappush(hNodes, newNode)


    return None if hNodes == [] else heapq.heappop(hNodes)
```

Testing

We will test this method on the **testFreqs** and **testFreqs2** frequency data dictionaries we already have

In [47]:
```
print(testFreqs)
print(testFreqs2)
```

```
{'x': 1, 's': 3, 'i': 3, 'h': 1, 't': 4, 'a': 1, 'f': 1, 'T': 1, ' ': 5, 'l': 1,
'e': 3}
{'g': 3, 's': 1, 'h': 1, 'o': 3, ' ': 2, 'r': 1, 'p': 1, 'e': 1}
```

In [48]:
```
testHtree = hTreeMakr(testFreqs)
```

In [49]:
```
print(testHtree.myLeft.myLeft.myChar)
print(str(testHtree.myLeft.myLeft.myChar).upper())
```

```
None
NONE
```

In [50]:
```
testHtree2 = hTreeMakr(testFreqs2)
```

In [51]:
```
print(testHtree2.myLeft.myLeft.myChar)
print(str(testHtree2.myLeft.myLeft.myChar).upper())
```

```
g
G
```

Since the **hTreeMakr** method tests properly, we move on to our *last* sub-routine.

Code Creator

The last *sub-routine* we need is one which will convert **HuffmanTrees** into a code index (*dictionary*). We call this method **codeFromHtree** and its code is

In [52]:
```
def codeFromHtree(hTree):
    code = dict()

    def bldCode(hNode, codeNow = ''):

        if (hNode == None):
            return

        if (hNode.myLeft == None and hNode.myRight == None):
            code[hNode.myChar] = codeNow

        bldCode(hNode.myLeft, codeNow + "0")
        bldCode(hNode.myRight, codeNow + "1")


    bldCode(hTree)


    return code
```

Testing

To test our **codeFromHtree** method, we will use the previously defined **testHtree** and **testHtree2**

In [53]:
```
testCode = codeFromHtree(testHtree)
print(testCode)
```

```
{'T': '0000', 'h': '0001', 'i': '001', ' ': '01', 's': '100', 'e': '101', 'f': '
11000', 'a': '11001', 'x': '11010', 'l': '11011', 't': '111'}
```

In [54]:
```
testCode2 = codeFromHtree(testHtree2)
print(testCode2)
```

```
{'g': '00', 's': '010', 'p': '0110', 'h': '0111', 'o': '10', 'r': '1100', 'e': '
1101', ' ': '111'}
```

Since this sub-routine correctly built all the codes in its tests, we can now move on to the actually programs for encoding and decoding.

Encoder

We will now create a method to handle the entire encoding process.

In [55]:
```
def encode(textIn):
    textDict = dictExtractr(textIn)

    freqs = freqCTR(textDict, textIn)

    myHtree = hTreeMakr(freqs)

    code = codeFromHtree(myHtree)

    encodedText = ""
    for char in textIn:
        encodedText += code[char]

    return encodedText
```

Testing

We will test with the previously defined strings **testText** and **testGophers**

In [56]:
```
testEncoded = encode(testText)
print(testEncoded)
```

```
00000001001100010011000111001011111011001110111110111010111011100000111011101
```

In [57]:
```
testEncoded2 = encode(testGophers)
print(testEncoded2)
```

```
0010111001011100100110011111011100010
```

Run on Tom Sawyer

First we start a timer

In [58]:
```
t0 = time.time()
```

Then we import the Tom Sawyer Text

In [59]:
```
tomText = fileRDR('../Text-Files/sawyer-ascii.txt')
```

followed by encoding

In [60]:
```
tomCompressed = encode(tomText)
print(len(tomCompressed))
```

```
1850008
```

In [61]:
```
t1 = time.time()
```

The compression ratio is

In [62]:
```
print(len(tomCompressed)/(8*len(tomText)))
```
```
0.574301218134181
```

and the elapsed time is

In [63]:
```
tComp = t1 - t0
print(tComp)
```
```
0.153289794921875
```

Run on the King James Version of the Bible

First we start a timer

In [64]:
```
t0 = time.time()
```

Then we import the King James Version of the Bible Text

In [65]:
```
bibleText = fileRDR('../Text-Files/kingJames-ascii.txt')
```

followed by encoding

In [66]:
```
bibleCompressed = encode(bibleText)
print(len(bibleCompressed))
```
```
19939923
```

In [67]:
```
t1 = time.time()
```

The compression ratio is

In [68]:
```
print(len(bibleCompressed)/(8*len(bibleText)))
```
```
0.5727394226626454
```

and the elapsed time is

In [69]:
```
tComp = t1 - t0
print(tComp)
```
```
1.1975061893463135
```

Decoder

Last, we will write a decoder method to decode encoded text.

In [70]:
```
def decoder(textIn, freqsIn):
    hTree = hTreeMakr(freqsIn)

    decoded = ""
    currentNode = hTree
    for compCode in textIn:
        if (compCode == "0"):
            currentNode = currentNode.myLeft
        else:
            currentNode = currentNode.myRight

        if (currentNode.isLeaf()):
            decoded += currentNode.myChar
            currentNode = hTree


    return decoded
```

Run on Tom Sawyer

Get frequencies for passing to the decoder

In [71]:
```
freqsToDecomp = freqCTR(dictExtractr(tomText), tomText)
```

Start a timer

In [72]:
```
t0 = time.time()
```

Decompress

In [73]:
```
tomDecomp = decoder(tomCompressed, freqsToDecomp)
```

Stop the timer

In [74]:
```
t1 = time.time()
```

Compare the lengths

In [75]:
```
print(len(tomDecomp))
print(len(tomText))
```

```
402665
402665
```

and contents

In [76]:
```
tomDecomp == tomText
```

Out[76]:
```
True
```

and, finally, compute the elapse time

In [77]:
```
totTime = t1 - t0
print(totTime)
```

```
0.7617809772491455
```

Run on The King James version of the Bible

Get frequencies for passing to the decoder

In [78]:
```
freqsToDecomp = freqCTR(dictExtractr(bibleText), bibleText)
```

Start a timer

In [79]:
```
t0 = time.time()
```

Decompress

In [80]:
```
bibleDecomp = decoder(bibleCompressed, freqsToDecomp)
```

Stop the timer

In [81]:
```
t1 = time.time()
```

Compare the lengths

In [82]:
```
print(len(bibleDecomp))
print(len(bibleText))
```

```
4351875
4351875
```

and contents

In [83]:
```
bibleDecomp == bibleText
```

Out[83]:
```
True
```

and, finally, compute the elapse time

In [84]:
```
totTime = t1 - t0
print(totTime)
```

```
7.724107980728149
```