Problem 1 (a)

Definitions

Prior to beginning our work, we load the requiste packages:

In [1]:
```
import math
```

We also load all 26 letters (*in lower case*) into an array for later use. This is done by importing a text file containing these letters

In [2]:
```
with open('../LowerCaseAlphabet.txt', 'r') as myFile:
    lowerAlpha = myFile.readlines()
```

and then stripping the return characters (*\n*) from each line

In [3]:
```
for i in range(len(lowerAlpha)):
    lowerAlpha[i] = lowerAlpha[i].replace('\n','')

lowerAlphaB = ''.join(str(x) for x in lowerAlpha)
```

Finally, we check to see if the alphabet imported properly,

In [4]:
```
print(lowerAlphaB)
```

```
abcdefghijklmnopqrstuvwxyz
```

as well as create an upper case version

In [5]:
```
upperAlpha = []
for x in lowerAlpha:
    upperAlpha.append(x.upper())

upperAlphaB = ''.join(str(x) for x in upperAlpha)
```

and check it

In [6]:
```
print(upperAlphaB)
```

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

Load File

We begin our work by loading the text from the source file:

In [7]:
```
with open('../Text-Files/sawyer-ascii.txt', 'r') as myFile:
        tempData = myFile.readlines()
```

Then, we get some basic information about the data imported from the file

In [8]:
```
print(len(tempData))
```

```
8807
```

Now, convert the array of strings to a single string.

In [9]:
```
data = ''.join(str(x) for x in tempData)
```

Then find and store the length of the resulting string:

In [10]:
```
charCNT = len(data)
print(charCNT)
```

```
402665
```

Next, the length compare it to the combined length of all the strings in the initial array we got from importing the text file.

In [11]:
```
cnt = 0
for x in tempData:
    cnt = cnt + len(x)

print(cnt)
```

```
402665
```

Since the character counts are accurate, we can proceed.

Get Character List

First, we will obtain a list of all characters occuring in the text

In [12]:
```
myChars = list(set(data))

myChars2 = ''.join(str(x) for x in myChars)
print(myChars)
print(myChars2)
```

```
['?', 'P', 'O', '+', '#', 'y', 'p', 'h', ';', 'w', '2', "'", 'c', 'R', 'T', ')',
'.', '_', '(', 'o', 's', '/', '4', '>', '\n', ']', '-', 'E', 'L', 't', ':', 'B',
'0', ',', 'd', 'I', 'x', 'l', 'k', '6', '9', '5', '$', '1', 'i', 'F', 'V', '!',
'K', 'a', 'v', 'H', 'W', '&', 'N', '<', '"', 'Y', '8', '[', 'A', 'm', 'C', 'D',
'n', 'j', 'g', '~', 'U', 'u', '*', 'q', 'b', 'f', 'r', 'S', 'z', 'M', 'G', ' ',
'%', '3', '7', 'J', '@', 'e', 'Q', 'X']
?PO+#yph;w2'cRT)._(os/4>
]-ELt:B0,dIxlk695$1iFV!KavHW&N<"Y8[AmCDnjg~Uu*qbfrSzMG %37J@eQX
```

Next, we remove our non-alphabetic characters from the list of characters to eliminate

In [13]:
```
for x in lowerAlphaB:
    myChars2 = myChars2.replace(x,'')

for x in upperAlphaB:
    myChars2 = myChars2.replace(x,'')

print(len(myChars2))
print(myChars2)
```

```
37
?+#;2')._(/4>
]-:0,695$1!&<"8[~* %37@
```

and check the results

In [14]:
```
print(list(set(myChars2)))
print(len(list(set(myChars2))))
```

```
['?', '+', '#', ':', '&', '0', ';', '2', "'", '<', ',', '"', '8', '[', ' ', '6',
')', '_', '.', '*', '(', '5', '%', '$', '3', '/', '7', '1', '@', '4', '>', '\n',
'!', '~', ']', '9', '-']
37
```

Clean the Text

Now, we can eliminate these characters from the text

In [15]:
```
#create copy of imported data
data2 = data

for x in myChars2:
    data2 = data2.replace(x,'')
```

and check to make sure the lengths have changed

In [16]:
```
print(len(data))
print(len(data2))
```

```
402665
307917
```

Last, we convert all letters in the text to lowercase

In [17]:
```
data2 = data2.lower()
```

Get Frequencies and Probabilities

Frequencies

To get the Frequencies of each letter, we first create an array to store them

In [18]:
```
counts = []
```

and then go though the alphabet counting

In [19]:
```
for x in lowerAlpha:
    counts.append(data2.count(x))

print(counts)
print(len(counts))
```

```
[24352, 5221, 6873, 15302, 37080, 6270, 6841, 19997, 19642, 692, 3138, 12565, 74
44, 20959, 24325, 4950, 194, 16092, 18376, 29970, 9340, 2474, 8244, 387, 7032, 1
57]
26
```

Now, we can compute the probabilities. First, we store the number of characters in the cleaned text

In [20]:
```
charTOT = len(data2)

print(charTOT)
```

```
307917
```

which we check against the frequencies we just calculated

In [21]:
```
print(sum(counts))
```

```
307917
```

Probabilities

Again, we first create and empty array to hold the probabilities

In [22]:
```
prbs = []
```

then we loop through the list of frequencies, using them to create each probability

In [23]:
```
for x in counts:
    prbs.append(x / charTOT)
```

this gives

In [24]:
```
print(prbs)
```

```
[0.07908624726793259, 0.016955867977409497, 0.022320950126170365, 0.049695210072
844304, 0.12042206178937831, 0.02036263018930426, 0.022217026016751268, 0.064942
8255016774, 0.0637899174128093, 0.002247358866187966, 0.01019105797991017, 0.040
80645108909219, 0.02417534595361737, 0.068067044041089, 0.07899856130061023, 0.0
1607576067576652, 0.0006300399133532738, 0.05226083652412825, 0.0596784198339162
8, 0.09733142372782276, 0.03033284943669885, 0.008034632709463915, 0.02677344868
909479, 0.0012568321982872007, 0.0228373230448465, 0.0005098776618374432]
```

Entropy Estimate

We can now estimate the entropy of the converted text (*all lower case, no special characters, spaces, tabs, or returns*). To do this, we first initialize a variable to hold our value for the entropy

In [25]:
```
entropTOT = 0
```

Then we loop through all the probabilities, computing the entropy for each and adding it to the total

In [26]:
```
for x in prbs:
    entropTOT = entropTOT - x * math.log2(x)
```

to get

In [27]:
```
print(entropTOT)
```

```
4.184820826080936
```

Problem 1 (b)

Definitions

Prior to beginning our work, we load the requiste packages:

In [1]:
```
import math
import time

t0 = time.time()
```

We also load all 26 letters (*in lower case*) into an array for later use. This is done by importing a text file containing these letters

In [2]:
```
with open('../LowerCaseAlphabet.txt', 'r') as myFile:
    lowerAlpha = myFile.readlines()
```

and then stripping the return characters (*\n*) from each line

In [3]:
```
for i in range(len(lowerAlpha)):
    lowerAlpha[i] = lowerAlpha[i].replace('\n','')

lowerAlphaB = ''.join(str(x) for x in lowerAlpha)
```

Finally, we check to see if the alphabet imported properly,

In [4]:
```
print(lowerAlphaB)
```

```
abcdefghijklmnopqrstuvwxyz
```

as well as create an upper case version

In [5]:
```
upperAlpha = []
for x in lowerAlpha:
    upperAlpha.append(x.upper())

upperAlphaB = ''.join(str(x) for x in upperAlpha)
```

and check it

In [6]:
```
print(upperAlphaB)
```

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

Load File

We begin our work by loading the text from the source file:

In [7]:
```
with open('../Text-Files/sawyer-ascii.txt', 'r') as myFile:
        tempData = myFile.readlines()
```

Then, we get some basic information about the data imported from the file

In [8]:
```
print(len(tempData))
```

```
8807
```

Now, convert the array of strings to a single string.

In [9]:
```
data = ''.join(str(x) for x in tempData)
```

Then find and store the length of the resulting string:

In [10]:
```
charCNT = len(data)
print(charCNT)
```

```
402665
```

Next, the length compare it to the combined length of all the strings in the initial array we got from importing the text file.

In [11]:
```
cnt = 0
for x in tempData:
    cnt = cnt + len(x)

print(cnt)
```

```
402665
```

Since the character counts are accurate, we can proceed.

Get Character List

First, we will obtain a list of all characters occuring in the text

In [12]:
```
myChars = list(set(data))

myChars2 = ''.join(str(x) for x in myChars)
print(myChars)
print(myChars2)
```

```
['m', 'r', 'q', '6', 'A', '"', 'i', '[', 'L', ':', 'K', 'D', 'b', 'R', 'X', 'y',
'0', 'e', 'T', '-', 'p', 'd', 'C', '$', 'H', "'", '2', '~', 'J', 'F', 'V', '/',
's', '+', 'U', '%', 'g', ' ', '*', '8', 'Y', '<', '9', 'Q', '5', ']', '7', 'P',
'v', 'l', 'S', '?', '.', 'z', 't', 'N', '\n', 'n', '>', ';', '&', 'o', '@', 'f',
'h', '(', ',', 'E', 'B', 'M', '!', 'W', 'a', 'O', 'I', 'w', ')', '4', '_', 'c',
'#', 'G', 'k', 'u', 'x', '3', 'j', '1']
mrq6A"i[L:KDbRXy0eT-pdC$H'2~JFV/s+U%g *8Y<9Q5]7PvlS?.ztN
n>;&o@fh(,EBM!WaOIw)4_c#Gkux3j1
```

Next, we remove our non-alphabetic characters from the list of characters to eliminate

In [13]:
```
for x in lowerAlphaB:
    myChars2 = myChars2.replace(x,'')

for x in upperAlphaB:
    myChars2 = myChars2.replace(x,'')

print(len(myChars2))
print(myChars2)
```

```
37
6"[:0-$'2~/+% *8<95]7?.
>;&@(,!)4_#31
```

and check the results

In [14]:
```
print(list(set(myChars2)))
print(len(list(set(myChars2))))
```

```
['!', '$', "'", '2', '~', '6', '"', '[', '\n', ':', '/', '>', '+', ')', '%', ' '
, '*', '8', '<', '9', ';', '4', '5', '_', '#', '&', '0', ']', '7', '@', '3', '('
, ',', '1', '?', '-', '.']
37
```

Clean the Text

Now, we can eliminate these characters from the text

In [15]:
```
#create copy of imported data
data2 = data

for x in myChars2:
    data2 = data2.replace(x,'')
```

and check to make sure the lengths have changed

In [16]:
```
print(len(data))
print(len(data2))
```

```
402665
307917
```

Last, we convert all letters in the text to lowercase

In [17]:
```
data2 = data2.lower()
```

Generate N-Grams

For the next part, we will need lists of Bi-Grams and Tri-Grams based off the lowercase english alphabet. We start with Bi-Grams

Bi-Grams

We begin with the array of lowercase alphabetic characters for the english language; however, we must first define an empty array to hold the Bi-Grams that we will generate. After which, we loop through the lowercase alphabet twice, joining each pair and appending the new pair to our array.

In [18]:
```
myBiGrams = []

for xx in lowerAlpha:
    for yy in lowerAlpha:
        temp = ''.join(str(xyz) for xyz in [xx,yy])
        myBiGrams.append(temp)

print(len(myBiGrams))
```

```
676
```

Checking that the number of Bi-Grams we generated is correct, we have

In [19]:
```
print(26*26)
```

```
676
```

Tri-Grams

Again, we begin with the array of lowercase alphabetic characters for the english language; however, we must first define an empty array to hold the Tri-Grams that we will generate. After which, we loop through the lowercase alphabet three times, joining each triplet and appending the new triplet to our array.

In [20]:
```
myTriGrams = []

for xx in lowerAlpha:
    for yy in lowerAlpha:
        for zz in lowerAlpha:
            temp = ''.join(str(xyz) for xyz in [xx,yy,zz])
            myTriGrams.append(temp)

print(len(myTriGrams))
```

```
17576
```

Checking that the number of Tri-Grams we generated is correct, we have

In [21]:
```
print(26*26*26)
```

```
17576
```

Quad-Grams

Again, we begin with the array of lowercase alphabetic characters for the english language; however, we must first define an empty array to hold the Quad-Grams that we will generate. After which, we loop through the lowercase alphabet four times, joining each quadruplet and appending the new quadruplet to our array.

In [22]:
```
myQuadGrams = []

for ww in lowerAlpha:
    for xx in lowerAlpha:
        for yy in lowerAlpha:
            for zz in lowerAlpha:
                temp = ''.join(str(wxyz) for wxyz in [ww,xx,yy,zz])
                myQuadGrams.append(temp)

print(len(myQuadGrams))
```

```
456976
```

Checking that the number of Tri-Grams we generated is correct, we have

In [23]:
```
print(26*26*26*26)
```

```
456976
```

Get Frequencies and Probabilities

Frequencies

To get the Frequencies of each letter, we first create an arrays to store them for Bi-Grams and Tri-Grams

In [24]:
```
counts = []
cntsBI= []
cntsTRI = []
cntsQUAD = []
```

and then go through the alphabet counting

In [25]:
```
for x in lowerAlpha:
    counts.append(data2.count(x))

print(len(counts))
print(counts)
```

```
26
[24352, 5221, 6873, 15302, 37080, 6270, 6841, 19997, 19642, 692, 3138, 12565, 74
44, 20959, 24325, 4950, 194, 16092, 18376, 29970, 9340, 2474, 8244, 387, 7032, 1
57]
```

and then go though the Bi-Grams counting

In [26]:
```
for x in myBiGrams:
    cntsBI.append(data2.count(x))

print(len(cntsBI))
```

```
676
```

and then the Tri-Grams counting

In [27]:
```
for x in myTriGrams:
    cntsTRI.append(data2.count(x))

print(len(cntsTRI))
```

```
17576
```

and then the Quad-Grams counting

In [28]:
```
for x in myQuadGrams:
    cntsQUAD.append(data2.count(x))

print(len(cntsQUAD))
```

```
456976
```

Character and N-Gram totals

Now, we can compute the probabilities. First, we store the number of characters in the cleaned text

In [29]:
```
charTOT = len(data2)

print(charTOT)
```

```
307917
```

which we check against the frequencies we just calculated

In [30]:
```
print(sum(counts))
```

```
307917
```

Then we compute the number of Bi-, Tri-, and Quad- Grams based on the total number of characters in the text

In [31]:
```
biTOT = math.floor(charTOT / 2)
triTOT = math.floor(charTOT / 3)
quadTOT = math.floor(charTOT / 4)
```

which gives

In [32]:
```
print(biTOT)
print(triTOT)
print(quadTOT)
```

```
153958
102639
76979
```

Probabilities

Again, we first create and empty array to hold the probabilities for single characters, as well as all our N-Grams

In [33]:
```
prbs = []
biPRBS = []
triPRBS = []
quadPRBS = []
```

then we loop through the list of frequencies, using them to create each probability

In [34]:
```
for x in counts:
    prbs.append(x / charTOT)

for x in cntsBI:
    biPRBS.append(x / biTOT)

for x in cntsTRI:
    triPRBS.append(x / triTOT)

for x in cntsQUAD:
    quadPRBS.append(x / quadTOT)
```

this gives

In [35]:
```
print(prbs)
```

```
[0.07908624726793259, 0.016955867977409497, 0.022320950126170365, 0.049695210072
844304, 0.12042206178937831, 0.02036263018930426, 0.022217026016751268, 0.064942
8255016774, 0.0637899174128093, 0.002247358866187966, 0.01019105797991017, 0.040
80645108909219, 0.02417534595361737, 0.068067044041089, 0.07899856130061023, 0.0
1607576067576652, 0.0006300399133532738, 0.05226083652412825, 0.0596784198339162
8, 0.09733142372782276, 0.03033284943669885, 0.008034632709463915, 0.02677344868
909479, 0.0012568321982872007, 0.0228373230448465, 0.00050987766183744 32]
```

for the single characters and array sizes for the others as

In [36]:
```
print(len(biPRBS))
print(len(triPRBS))
print(len(quadPRBS))
```

```
676
17576
456976
```

Entropy Estimate

Single Characters

We can now estimate the entropy of the converted text (*all lower case, no special characters, spaces, tabs, or returns*). To do this, we first initialize a variable to hold our value for the entropy

In [37]:
```
entropTOT = 0
```

Then we loop through all the probabilities, computing the entropy for each and adding it to the total

In [38]:
```
for x in prbs:
    entropTOT = entropTOT - x * math.log2(x)
```

to get

In [39]:
```
print(entropTOT)
```

```
4.184820826080936
```

Bi-Grams

For Bi-Grams, we first initialize a variable to hold our value for the entropy

In [40]:
```
biEntropTOT = 0
```

Then we loop through all the Bi-Gram Probabilities

In [41]:
```
testI = 0
for x in biPRBS:
    testI += 1

    if x == 0.0:
        biEntropTOT = biEntropTOT
    else:
        biEntropTOT = biEntropTOT - x * math.log2(x)


print(testI)
print(biEntropTOT)
```

```
676
13.561376307716296
```

Tri-Grams

For Tr-Grams, we first initialize a variable to hold our value for the entropy

In [42]:
```
triEntropTOT = 0
```

Then we loop through all the Tri-Gram Probabilities

In [43]:
```
testI = 0
for x in triPRBS:
    testI += 1

    if x == 0.0:
        triEntropTOT = triEntropTOT
    else:
        triEntropTOT = triEntropTOT - x * math.log2(x)


print(testI)
print(triEntropTOT)
```

```
17576
27.92124606372456
```

Quad-Grams

For Quad-Grams, we first initialize a variable to hold our value for the entropy

In [44]:
```
quadEntropTOT = 0
```

Then we loop through all the Qaud-Gram Probabilities

In [45]:
```
testI = 0
for x in quadPRBS:
    testI += 1

    if x == 0.0:
        quadEntropTOT = quadEntropTOT
    else:
        quadEntropTOT = quadEntropTOT - x * math.log2(x)


print(testI)
print(quadEntropTOT)
```

```
456976
45.63916839392134
```

Times

Overall, it took

In [46]:
```
t1 = time.time()
print(t1-t0)
```

```
103.23156189918518
```

Problem 1 (c)

If the entropy is estimated using Tri-Grams instead of Bi-Grams, the estimate of the entropy will be higher than the estimate calcuated using Bi-Grams. This is due to the fact that the alphabet size is larger for Tri-Grams compared with Bi-Grams.

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
['h', ' ', 'f', 'x', 'a', 'l', 'i', 'e', 'T', 't', 's']
```

While testing on **testGophers** gives

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

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

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

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)
```

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

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)
```

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

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

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

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

```
{'h': 1, ' ': 5, 'f': 1, 'x': 1, 'a': 1, 'l': 1, 'i': 3, 'e': 3, 'T': 1, 't': 4,
's': 3}
{'h': 1, ' ': 2, 'g': 3, 'o': 3, 'e': 1, 'r': 1, 'p': 1, 's': 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', 'x': '0001', 'i': '001', ' ': '01', 'e': '100', 's': '101', 'l': '
11000', 'f': '11001', 'h': '11010', 'a': '11011', 't': '111'}
```

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

```
{'g': '00', 's': '010', 'p': '0110', 'r': '0111', 'o': '10', 'h': '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)
```

```
00001101000110101001101011101101111100101111011111000001111011100100111000100
```

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

```
00101110010111001001101100110101110 10
```

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
```

and stopping the timer

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)
```

```
0.13590407371520996
```

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
```

and stopping the timer

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)
```

```
1.0406601428985596
```

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)
```
```
0.5515129566192627
```

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)
```

```
6.176233291625977
```

Problem 3

We have three parts:

- Imports, global function definitions, and text import.
  - Import required libraries (*i.e. time, csv, etc*)
  - Define any required global functions
  - Import the text to work with from its .txt file
- Work for the LZ-Compression part
- Work for the LZ-DEcompression part

Thus, we now move to imports, global function definitions, and text import.

Imports, Global Function Definitions, and Text-File Import.

We start with library imports.

Library Imports

We require the **time**

In [1]:
```
import time
```

the **csv**

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

and the **requests** libraries

In [3]:
```
import requests
```

Then we move on to global function definitions.

Global Function Definitions

We require four global functions. These are

- A text-file reader
- A text-file writer
- A csv-file reader
- A csv-file writer

Text-File Reader

We start with a text-file reader

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

        myTextFileIn.close()


    return myTextIn
```

Text-File Writer

We continue with a text-file writer

In [5]:
```
def fileWTR(filename, strToWrite):
    with open(filename, 'w') as myTextFileOut:
        myTextFileOut.write(strToWrite)

        myTextFileOut.close()


    return None
```

CSV-File reader

We follow that with a csv-file reader

In [6]:
```
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
```

CSV-File Writer

We finish with a csv-file writer

In [7]:
```
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
```

Text Import

We will now import two different text-files for use in this problem.

Tom Sawyer

The first, is "*Tom Sawyer*"

In [8]:
```
textInA = fileRDR('../Text-Files/sawyer-ascii.txt')
```

King James Bible

The second in the "*King James version of the Bible*"

In [9]:
```
textInB = fileRDR('../Text-Files/kingJames-ascii.txt')
```

Having finished importing libraries, defining global functions, and importing the files we are going to work with, we move on to the work for LZ-Compression

LZ-Compression

We will first define a method for compression a text file using the Lempel-Ziv Compression routine specified in the book.

In [10]:
```
def lzCompr(textIn):

    myDict = dict()
    myAns = []
    myResult = ''
    myComp = ''


    posCTR = 1
    word = ''
    for char in textIn:
        wordNchar = word + char

        if (wordNchar in myDict):
            word = wordNchar
        else:
            myDict[wordNchar] = posCTR
            posCTR += 1

            if (len(wordNchar) == 1):
                myAns.append([0, char])
            else:
                anINT = myDict[word]
                myAns.append([anINT, char])

            word = ''


    if (word):
        anINT = myDict[word]
        myAns.append([anINT])


    for row in myAns:
        for col in row:
            myResult = myResult + str(col)


    for row in myAns:
        tst = 0

        for col in row:

            if tst == 0:
                myComp = myComp + str(col) + ','
            else:
                myComp = myComp + str(col) + ';'

            tst += 1


    myOut = [myAns, myResult, myComp]

    return myOut
```

Alternate Function

We also define this alternate, and slightly simpler, function

In [11]:
```
def lzComprA(textIn):

    myDict = dict()
    myAns = []


    posCTR = 1
    word = ''
    for char in textIn:
        wordNchar = word + char

        if (wordNchar in myDict):
            word = wordNchar
        else:
            myDict[wordNchar] = posCTR
            posCTR += 1

            if (len(wordNchar) == 1):
                myAns.append([0, char])
            else:
                anINT = myDict[word]
                myAns.append([anINT, char])

            word = ''


    if (word):
        anINT = myDict[word]
        myAns.append([anINT])


    return myAns
```

Tom Sawyer

Now, we can run and time the compression routine for *Tom Sawyer*.

Initial Verion

By using the first routine we defined

In [12]:
```
t0 = time.time()
compTextArrA = lzCompr(textInA)
t1 = time.time()
```

To get the results

In [13]:
```
formCompText = compTextArrA[2]
noFormCompText = compTextArrA[1]

tTot = t1 - t0
origTextLEN = len(textInA)
formCompTextLEN = len(formCompText)
noFormCompTextLEN = len(noFormCompText)

perCnoF = noFormCompTextLEN / origTextLEN
perCisF = formCompTextLEN / origTextLEN

print('\n')
print('# of characters in original version :  ' + str(origTextLEN))
print('# of characters in formatted compressed version :  ' + str(formCompTextLEN)
)
print('# of characters in NON-formatted compressed version :  ' + str(noFormCompTe
xtLEN))
print('compression ratio of formatted compressed version :  ' + str(perCisF))
print('compression ratio of NON-formatted compressed version :  ' + str(perCnoF))
print('total runtime :  ' + str(tTot) + ' sec')
print('DONE DONE DONE !!!')
print('\n')
```

```
# of characters in original version :  402665
# of characters in formatted compressed version :  540258
# of characters in NON-formatted compressed version :  393457
compression ratio of formatted compressed version :  1.341705884544224
compression ratio of NON-formatted compressed version :  0.9771323556802801
total runtime :  2.377018928527832 sec
DONE DONE DONE !!!
```

Alternate Version

And then compress *Tom Sawyer* again, this time using the alternate version of the routine.

In [14]:
```
t0 = time.time()
compTextArrAa = lzComprA(textInA)
t1 = time.time()
```

To get the results

In [15]:
```
formCompText = compTextArrAa[2]
noFormCompText = compTextArrAa[1]

tTot = t1 - t0
origTextLEN = len(textInA)
formCompTextLEN = len(formCompText)
noFormCompTextLEN = len(noFormCompText)

perCnoF = noFormCompTextLEN / origTextLEN
perCisF = formCompTextLEN / origTextLEN

print('\n')
print('# of characters in original version :  ' + str(origTextLEN))
print('# of characters in formatted compressed version :  ' + str(formCompTextLEN)
)
print('# of characters in NON-formatted compressed version :  ' + str(noFormCompTe
xtLEN))
print('compression ratio of formatted compressed version :  ' + str(perCisF))
print('compression ratio of NON-formatted compressed version :  ' + str(perCnoF))
print('total runtime :  ' + str(tTot) + ' sec')
print('DONE DONE DONE !!!')
print('\n')
```

```
# of characters in original version :  402665
# of characters in formatted compressed version :  2
# of characters in NON-formatted compressed version :  2
compression ratio of formatted compressed version :  4.9669079756124815e-06
compression ratio of NON-formatted compressed version :  4.9669079756124815e-06
total runtime :  0.1863260269165039 sec
DONE DONE DONE !!!
```

King James Version of the Bible

Now, we can run and time the compression routine for *King James Version of the Bible*.

Initial Verion

By using the first routine we defined

In [16]:
```
t0 = time.time()
compTextArrB = lzCompr(textInB)
t1 = time.time()
```

To get the results

In [17]:
```
formCompText = compTextArrB[2]
noFormCompText = compTextArrB[1]

tTot = t1 - t0
origTextLEN = len(textInB)
formCompTextLEN = len(formCompText)
noFormCompTextLEN = len(noFormCompText)

perCnoF = noFormCompTextLEN / origTextLEN
perCisF = formCompTextLEN / origTextLEN

print('\n')
print('# of characters in original version :  ' + str(origTextLEN))
print('# of characters in formatted compressed version :  ' + str(formCompTextLEN)
)
print('# of characters in NON-formatted compressed version :  ' + str(noFormCompTe
xtLEN))
print('compression ratio of formatted compressed version :  ' + str(perCisF))
print('compression ratio of NON-formatted compressed version :  ' + str(perCnoF))
print('total runtime :  ' + str(tTot) + ' sec')
print('DONE DONE DONE !!!')
print('\n')
```

```
# of characters in original version :  4351875
# of characters in formatted compressed version :  4598217
# of characters in NON-formatted compressed version :  3499160
compression ratio of formatted compressed version :  1.0566059457130548
compression ratio of NON-formatted compressed version :  0.8040580209679736
total runtime :  352.93959760665894 sec
DONE DONE DONE !!!
```

Alternate Version

And then compress *The King James Version of the Bible* again, this time using the alternate version of the routine.

In [18]:
```
t0 = time.time()
compTextArrBa = lzComprA(textInB)
t1 = time.time()
```

To get the results

In [19]:
```
formCompText = compTextArrBa[2]
noFormCompText = compTextArrBa[1]

tTot = t1 - t0
origTextLEN = len(textInB)
formCompTextLEN = len(formCompText)
noFormCompTextLEN = len(noFormCompText)

perCnoF = noFormCompTextLEN / origTextLEN
perCisF = formCompTextLEN / origTextLEN

print('\n')
print('# of characters in original version :  ' + str(origTextLEN))
print('# of characters in formatted compressed version :  ' + str(formCompTextLEN)
)
print('# of characters in NON-formatted compressed version :  ' + str(noFormCompTe
xtLEN))
print('compression ratio of formatted compressed version :  ' + str(perCisF))
print('compression ratio of NON-formatted compressed version :  ' + str(perCnoF))
print('total runtime :  ' + str(tTot) + ' sec')
print('DONE DONE DONE !!!')
print('\n')
```

```
# of characters in original version :  4351875
# of characters in formatted compressed version :  2
# of characters in NON-formatted compressed version :  2
compression ratio of formatted compressed version :  4.595720235530662e-07
compression ratio of NON-formatted compressed version :  4.595720235530662e-07
total runtime :  2.0846071243286133 sec
DONE DONE DONE !!!
```

With the compression part done, we move on to LZ Decompression

LZ-Decompression

We will first define a method for decompression a compressed| text file using the Lempel-Ziv Compression routine specified in the book.

In [ ]:
```
def lzDEcompr(inArray):
    mySize = len(inArray)
    mySize1 = mySize - 1

    myEndSize = len(inArray[mySize - 1])

    deCompText = ""

    for i in range(mySize1):
        row = inArray[i]

        numNow = row[0]
        charNow = row[1]

        if (numNow == 0):
            deCompText += charNow
        else:
            wordNow = charNow

            while (numNow != 0):
                newRow = inArray[numNow]

                charNow = newRow[1]
                wordNow = charNow + wordNow

                numNow = newRow[0]

            deCompText += wordNow


    row = inArray[mySize1]
    if (myEndSize == 1):
        tmpNum = row[0]

        rowNow = inArray[tmpNum]

        charNow = rowNow[1]
        numNow = rowNow[0]

        if (numNow == 0):
            deCompText += charNow
        else:
            wordNow = charNow

            while (numNow != 0):
                newRowNow = inArray[numNow]

                charNow = newRowNow[1]
                wordNow = charNow + wordNow

                numNow = newRowNow[0]

            deCompText += wordNow

    else:
        charNow = row[1]
        numNow = row[0]

        if (numNow == 0):
            deCompText += charNow
        else:
            wordNow = charNow

            while (numNow != 0):
```

Run on Compressed Tom Sawyer

First, we run the decompression routine on the compressed *Tom Sawyer*.

In [ ]:
```
t0 = time.time()
tomDecomp = lzDEcompr(compTextArrAa)
t1 = time.time()
```

Then compare the lengths

In [ ]:
```
print(len(textInA))
print(len(tomDecomp))
```

and the texts themselves

In [ ]:
```
tomDecomp == textInA
```

and, finally, and elapsed time

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

Run on Compressed King James Version of the Bible

First, we run the decompression routine on the compressed *King James Version of the Bible*.

In [ ]:
```
t0 = time.time()
bibleDecomp = lzDEcompr(compTextArrBa)
t1 = time.time()
```

Then compare the lengths

In [ ]:
```
print(len(textInB))
print(len(bibleDecomp))
```

and the texts themselves

In [ ]:
```
textInB == bibleDecomp
```

and, finally, and elapsed time

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

Problem 3 (*alt1*)

We have three parts:

- Imports, global function definitions, and text import.
    - Import required libraries (*i.e. time, csv, etc*)
    - Define any required global functions
    - Import the text to work with from its .txt file
- Work for the LZ-Compression part
- Work for the LZ-DEcompression part

Thus, we now move to imports, global function definitions, and text import.

Imports, Global Function Definitions, and Text-File Import.

We start with library imports.

Library Imports

We require the **time**

In [1]:
```
import time
```

the **csv**

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

and the **requests** libraries

In [3]:
```
import requests
```

Then we move on to global function definitions.

Global Function Definitions

We require four global functions. These are

- A text-file reader
- A text-file writer
- A csv-file reader
- A csv-file writer

Text-File Reader

We start with a text-file reader

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

        myTextFileIn.close()


    return myTextIn
```

Text-File Writer

We continue with a text-file writer

In [5]:
```
def fileWTR(filename, strToWrite):
    with open(filename, 'w') as myTextFileOut:
        myTextFileOut.write(strToWrite)

        myTextFileOut.close()


    return None
```

CSV-File reader

We follow that with a csv-file reader

In [6]:
```
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
```

CSV-File Writer

We finish with a csv-file writer

In [7]:
```
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
```

Text Import

We will now import two different text-files for use in this problem.

Tom Sawyer

The first, is "*Tom Sawyer*"

In [8]:
```
textInA = fileRDR('../Text-Files/sawyer-ascii.txt')
```

King James Bible

The second in the "*King James version of the Bible*"

In [9]:
```
textInB = fileRDR('../Text-Files/kingJames-ascii.txt')
```

Having finished importing libraries, defining global functions, and importing the files we are going to work with, we move on to the work for LZ-Compression

LZ-Compression

We will first define a method for compression a text file using the Lempel-Ziv Compression routine specified in the book.

In [10]:
```
def lzCompr(textIn):

    myDict = dict()
    myAns = []
    myResult = ''
    myComp = ''


    posCTR = 1
    word = ''
    for char in textIn:
        wordNchar = word + char

        if (wordNchar in myDict):
            word = wordNchar
        else:
            myDict[wordNchar] = posCTR
            posCTR += 1

            if (len(wordNchar) == 1):
                myAns.append([0, char])
            else:
                anINT = myDict[word]
                myAns.append([anINT, char])

            word = ''


    if (word):
        anINT = myDict[word]
        myAns.append([anINT])


    for row in myAns:
        for col in row:
            myResult = myResult + str(col)


    for row in myAns:
        tst = 0

        for col in row:

            if tst == 0:
                myComp = myComp + str(col) + ','
            else:
                myComp = myComp + str(col) + ';'

            tst += 1


    myOut = [myAns, myResult, myComp]

    return myOut
```

Alternate Function

We also define this alternate, and slightly simpler, function

In [11]:
```
def lzComprA(textIn):

    myDict = dict()
    myAns = []


    posCTR = 1
    word = ''
    for char in textIn:
        wordNchar = word + char

        if (wordNchar in myDict):
            word = wordNchar
        else:
            myDict[wordNchar] = posCTR
            posCTR += 1

            if (len(wordNchar) == 1):
                myAns.append([0, char])
            else:
                anINT = myDict[word]
                myAns.append([anINT, char])

            word = ''


    if (word):
        anINT = myDict[word]
        myAns.append([anINT])


    return myAns
```

Tom Sawyer

Now, we can run and time the compression routine for *Tom Sawyer*.

Initial Verion

By using the first routine we defined

In [12]:
```
t0 = time.time()
compTextArrA = lzCompr(textInA)
t1 = time.time()
```

To get the results

In [13]:
```
formCompText = compTextArrA[2]
noFormCompText = compTextArrA[1]

tTot = t1 - t0
origTextLEN = len(textInA)
formCompTextLEN = len(formCompText)
noFormCompTextLEN = len(noFormCompText)

perCnoF = noFormCompTextLEN / origTextLEN
perCisF = formCompTextLEN / origTextLEN

print('\n')
print('# of characters in original version :  ' + str(origTextLEN))
print('# of characters in formatted compressed version :  ' + str(formCompTextLEN)
)
print('# of characters in NON-formatted compressed version :  ' + str(noFormCompTe
xtLEN))
print('compression ratio of formatted compressed version :  ' + str(perCisF))
print('compression ratio of NON-formatted compressed version :  ' + str(perCnoF))
print('total runtime :  ' + str(tTot) + ' sec')
print('DONE DONE DONE !!!')
print('\n')
```

```
# of characters in original version :  402665
# of characters in formatted compressed version :  540258
# of characters in NON-formatted compressed version :  393457
compression ratio of formatted compressed version :  1.341705884544224
compression ratio of NON-formatted compressed version :  0.9771323556802801
total runtime :  2.5548479557037354 sec
DONE DONE DONE !!!
```

Alternate Version

And then compress *Tom Sawyer* again, this time using the alternate version of the routine.

In [14]:
```
t0 = time.time()
compTextArrAa = lzComprA(textInA)
t1 = time.time()
```

To get the results

In [ ]:
```
formCompText = compTextArrAa[2]
noFormCompText = compTextArrAa[1]

tTot = t1 - t0
origTextLEN = len(textInA)
formCompTextLEN = len(formCompText)
noFormCompTextLEN = len(noFormCompText)

perCnoF = noFormCompTextLEN / origTextLEN
perCisF = formCompTextLEN / origTextLEN

print('\n')
print('# of characters in original version :  ' + str(origTextLEN))
print('# of characters in formatted compressed version :  ' + str(formCompTextLEN)
)
print('# of characters in NON-formatted compressed version :  ' + str(noFormCompTe
xtLEN))
print('compression ratio of formatted compressed version :  ' + str(perCisF))
print('compression ratio of NON-formatted compressed version :  ' + str(perCnoF))
print('total runtime :  ' + str(tTot) + ' sec')
print('DONE DONE DONE !!!')
print('\n')
```

```
# of characters in original version :  402665
# of characters in formatted compressed version :  2
# of characters in NON-formatted compressed version :  2
compression ratio of formatted compressed version :  4.9669079756124815e-06
compression ratio of NON-formatted compressed version :  4.9669079756124815e-06
total runtime :  0.30546998977661133 sec
DONE DONE DONE !!!
```

King James Version of the Bible

Now, we can run and time the compression routine for *King James Version of the Bible*.

Initial Verion

By using the first routine we defined

In [ ]:
```
t0 = time.time()
compTextArrB = lzCompr(textInB)
t1 = time.time()
```

To get the results

In [ ]:
```
formCompText = compTextArrB[2]
noFormCompText = compTextArrB[1]

tTot = t1 - t0
origTextLEN = len(textInB)
formCompTextLEN = len(formCompText)
noFormCompTextLEN = len(noFormCompText)

perCnoF = noFormCompTextLEN / origTextLEN
perCisF = formCompTextLEN / origTextLEN

print('\n')
print('# of characters in original version :  ' + str(origTextLEN))
print('# of characters in formatted compressed version :  ' + str(formCompTextLEN)
)
print('# of characters in NON-formatted compressed version :  ' + str(noFormCompTe
xtLEN))
print('compression ratio of formatted compressed version :  ' + str(perCisF))
print('compression ratio of NON-formatted compressed version :  ' + str(perCnoF))
print('total runtime :  ' + str(tTot) + ' sec')
print('DONE DONE DONE !!!')
print('\n')
```

Alternate Version

And then compress *The King James Version of the Bible* again, this time using the alternate version of the routine.

In [ ]:
```
t0 = time.time()
compTextArrBa = lzComprA(textInB)
t1 = time.time()
```

To get the results

In [ ]:
```
formCompText = compTextArrBa[2]
noFormCompText = compTextArrBa[1]

tTot = t1 - t0
origTextLEN = len(textInB)
formCompTextLEN = len(formCompText)
noFormCompTextLEN = len(noFormCompText)

perCnoF = noFormCompTextLEN / origTextLEN
perCisF = formCompTextLEN / origTextLEN

print('\n')
print('# of characters in original version :  ' + str(origTextLEN))
print('# of characters in formatted compressed version :  ' + str(formCompTextLEN)
)
print('# of characters in NON-formatted compressed version :  ' + str(noFormCompTe
xtLEN))
print('compression ratio of formatted compressed version :  ' + str(perCisF))
print('compression ratio of NON-formatted compressed version :  ' + str(perCnoF))
print('total runtime :  ' + str(tTot) + ' sec')
print('DONE DONE DONE !!!')
print('\n')
```

With the compression part done, we move on to LZ Decompression

LZ-Decompression

We will first define a method for decompression a compressed| text file using the Lempel-Ziv Compression routine specified in the book.

In [ ]:
```
def lzDEcompr(inArray):
    mySize = len(inArray)
    mySize1 = mySize - 1

    myEndSize = len(inArray[mySize - 1])

    deCompText = ""

    for i in range(mySize1):
        row = inArray[i]

        numNow = row[0]
        charNow = row[1]

        if (numNow == 0):
            deCompText += charNow
        else:
            wordNow = charNow

            while (numNow != 0):
                newRow = inArray[numNow]

                charNow = newRow[1]
                wordNow = charNow + wordNow

                numNow = newRow[0]

            deCompText += wordNow


    row = inArray[mySize1]
    numNow = row[0]
    charNow = ''
    if (myEndSize == 1):
        rowNow = inArray[numNow]

        charNow = rowNow[1]
        numNow = rowNow[0]
    else:
        charNow = row[1]

    if (numNow == 0):
        deCompText += charNow
    else:
        wordNow = charNow

        while (numNow != 0):
            newRowNow = inArray[numNow]

            charNow = newRowNow[1]
            wordNow = charNow + wordNow

            numNow = newRowNow[0]

        deCompText += wordNow


    return deCompText
```

Run on Compressed Tom Sawyer

First, we run the decompression routine on the compressed *Tom Sawyer*.

In [ ]:
```
t0 = time.time()
tomDecomp = lzDEcompr(compTextArrAa)
t1 = time.time()
```

Then compare the lengths

In [ ]:
```
print(len(textInA))
print(len(tomDecomp))
```

and the texts themselves

In [ ]:
```
tomDecomp == textInA
```

and, finally, and elapsed time

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

Run on Compressed King James Version of the Bible

First, we run the decompression routine on the compressed *King James Version of the Bible*.

In [ ]:
```
t0 = time.time()
bibleDecomp = lzDEcompr(compTextArrBa)
t1 = time.time()
```

Then compare the lengths

In [ ]:
```
print(len(textInB))
print(len(bibleDecomp))
```

and the texts themselves

In [ ]:
```
textInB == bibleDecomp
```

and, finally, and elapsed time

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

Problem 4 (a)

First, import the **math** library

In [1]:
```
import math
import scipy.special
```

Then move on to the formal math

Define R.V.s

We define the random variables X and Y as

- X - The outcome of a series
- Y - The number of games in a series

The R.V. X has the probabilities

- **A** wins in 5 : $(0.5)^5 = 0.03125$ with multiplicity 1
- **A** wins in 6 : $(0.5)^5(0.5)^1 = 0.015625$ with multiplicity $\binom{5}{4}$
- **A** wins in 7 : $(0.5)^5(0.5)^2 = 0.0078125$ with multiplicity $\binom{6}{4}$
- **A** wins in 8 : $(0.5)^5(0.5)^3 = 0.00390625$ with multiplicity $\binom{7}{4}$
- **A** wins in 9 : $(0.5)^5(0.5)^4 = 0.00195313$ with multiplicity $\binom{8}{4}$
- **B** wins in 5 : $(0.5)^5 = 0.03125$ with multiplicity 1
- **B** wins in 6 : $(0.5)^5(0.5)^1 = 0.015625$ with multiplicity $\binom{5}{4}$
- **B** wins in 7 : $(0.5)^5(0.5)^2 = 0.0078125$ with multiplicity $\binom{6}{4}$
- **B** wins in 8 : $(0.5)^5(0.5)^3 = 0.00390625$ with multiplicity $\binom{7}{4}$
- **B** wins in 9 : $(0.5)^5(0.5)^4 = 0.00195313$ with multiplicity $\binom{8}{4}$

Thus, we define the probability and multiplicity arrays for X

In [2]:
```
probX = []
probX.append((0.5)**5)
probX.append(((0.5)**5)*((0.5)**1))
probX.append(((0.5)**5)*((0.5)**2))
probX.append(((0.5)**5)*((0.5)**3))
probX.append(((0.5)**5)*((0.5)**4))
probX.append((0.5)**5)
probX.append(((0.5)**5)*((0.5)**1))
probX.append(((0.5)**5)*((0.5)**2))
probX.append(((0.5)**5)*((0.5)**3))
probX.append(((0.5)**5)*((0.5)**4))

print(probX)
print(len(probX))
```

```
[0.03125, 0.015625, 0.0078125, 0.00390625, 0.001953125, 0.03125, 0.015625, 0.007
8125, 0.00390625, 0.001953125]
10
```

In [3]:
```
multX = []
multX.append(1)
multX.append(scipy.special.binom(5, 4))
multX.append(scipy.special.binom(6, 4))
multX.append(scipy.special.binom(7, 4))
multX.append(scipy.special.binom(8, 4))
multX.append(1)
multX.append(scipy.special.binom(5, 4))
multX.append(scipy.special.binom(6, 4))
multX.append(scipy.special.binom(7, 4))
multX.append(scipy.special.binom(8, 4))

print(multX)
print(len(multX))
```

```
[1, 5.0, 15.0, 35.0, 70.0, 1, 5.0, 15.0, 35.0, 70.0]
10
```

Now, the R.V. Y has the probabilities

- Series lasts 5 : $2(0.5)^5 = 0.0625$
- Series lasts 6 : $2\binom{5}{4}(0.5)^5(0.5)^1 = 0.15625$
- Series lasts 7 : $2\binom{5}{4}(0.5)^5(0.5)^2 = 0.234375$
- Series lasts 8 : $2\binom{5}{4}(0.5)^5(0.5)^3 = 0.273438$
- Series lasts 9 : $2\binom{5}{4}(0.5)^5(0.5)^4 = 0.273438$

Thus, we define the probability array for Y

In [4]:
```
probY = []
probY.append(2*probX[0]*multX[0])
probY.append(2*probX[1]*multX[1])
probY.append(2*probX[2]*multX[2])
probY.append(2*probX[3]*multX[3])
probY.append(2*probX[4]*multX[4])

print(probY)
print(len(probY))
```

```
[0.0625, 0.15625, 0.234375, 0.2734375, 0.2734375]
5
```

Before computing the entropies $H(x)$ and $H(y)$, we check that the probabilities we calculated are normal

In [5]:
```
xSum = 0
ySum = 0
for i in range(5):
    xSum += 2 * probX[i] * multX[i]
    ySum += probY[i]

print(xSum)
print(ySum)
```

```
1.0
1.0
```

Since the probabilities are normal, we can calculate $H(x)$ as

In [6]:
```
Hx = 0
for i in range(10):
    Hx += multX[i] * (- probX[i] * math.log2( probX[i] ))

print(Hx)
```

```
7.5390625
```

and $H(y)$ as

In [7]:
```
Hy = 0
for i in range(5):
    Hy += - probY[i] * math.log2( probY[i] )

print(Hy)
```

```
2.18206960194
```

Now, since $p(x, y)$ is

In [8]:
```
probXY = []
for i in range(10):
    xyRow = []
    for j in range(5):
        if i == j:
            xyRow.append(1/2)
        elif i % 5 == j:
            xyRow.append(1/2)
        else:
            xyRow.append(0)

    probXY.append(xyRow)


print(probXY)
```

```
[[0.5, 0, 0, 0, 0], [0, 0.5, 0, 0, 0], [0, 0, 0.5, 0, 0], [0, 0, 0, 0.5, 0], [0,
0, 0, 0, 0.5], [0.5, 0, 0, 0, 0], [0, 0.5, 0, 0, 0], [0, 0, 0.5, 0, 0], [0, 0, 0
, 0.5, 0], [0, 0, 0, 0, 0.5]]
```

and $H(x, y)$ as

In [9]:
```
Hxy = 0
for i in range(10):
    for j in range(5):
        tmpArr = probXY[i]
        tmp = tmpArr[j]

        if tmp == 0:
            Hxy = Hxy
        else:
            Hxy += - tmp * math.log2(tmp)


print(Hxy)
```

```
5.0
```

We also have the conditional probability $H(x \mid y)$ as

In [10]:
```
HxGIVy = 0
for i in range(5):
    # From player A winning in (i+5) games
    tmpA = multX[i] * (- (probX[i]*probY[i]) * math.log2(probX[i]*probY[i]))

    # From player B winning in (i+5) games
    tmpB = multX[i+5] * (- (probX[i+5]*probY[i]) * math.log2(probX[i]*probY[i]))

    tmpTOT = tmpA + tmpB
    HxGIVy += tmpTOT

print(HxGIVy)
```

```
2.29731956998
```

As well as the conditional probability $H(y \mid x)$ as

In [11]:
```
HyGIVx = 0
for i in range(10):
    HyGIVx += multX[i] * (- probY[i % 5]*probX[i] * math.log2(probY[i % 5]*probX[i
]))

print(HyGIVx)
```

```
2.29731956998
```

Using the relation $I(X;Y) = H(X) + H(Y) - H(X,Y)$, we have $I(X;Y)$ as

In [12]:
```
Ixy = Hx + Hy - Hxy
print(Ixy)
```

```
4.72113210194
```

Next, we need to define the distributions pA (for X when A wins) and qA (for Y when A wins). The distribution pA can be represented by the matrix

In [13]:
```
pA = []
for i in range(5):
    tmp = multX[i] * (probX[i] + probX[i+5])
    pA.append(tmp)

print(pA)
```

```
[0.0625, 0.15625, 0.234375, 0.2734375, 0.2734375]
```

Similarly, the distribution qA can be represented by the matrix

In [14]:
```
qA = []
for i in range(5):
    qA.append(probY[i])

print(qA)
```

```
[0.0625, 0.15625, 0.234375, 0.2734375, 0.2734375]
```

After checking these new distributions for normality, we proceed.

In [15]:
```
pAsum = 0
qAsum = 0
for i in range(5):
    pAsum += pA[i]
    qAsum += qA[i]

print(pAsum)
print(qAsum)
```

```
1.0
1.0
```

For $D(pA \;||\; X)$ we have

In [16]:
```
dPaX = 0
for i in range(5):
    dPaX += pA[i] * math.log2(pA[i] / probX[i])

print(dPaX)
```

```
5.35699289806
```

In [17]:
```
dPaX = 0
for i in range(5):
    dPaX += pA[i] * math.log2(pA[i] / (multX[i]*probX[i]))

print(dPaX)
```

```
1.0
```

and for $D(qA \;||\; Y)$ we have

In [18]:
```
dQaY = 0
for i in range(5):
    dQaY += qA[i] * math.log2(qA[i] / probY[i])

print(dQaY)
```

```
0.0
```

Problem 4 (b)

First, define the probabilities of X, Y, and Z using the arrays

In [1]:
```
pX = []
pX.append(1/2)
pX.append(1/2)

print(pX)
```

```
[0.5, 0.5]
```

In [2]:
```
pY = []
pY.append(1/2)
pY.append(1/2)

print(pY)
```

```
[0.5, 0.5]
```

In [3]:
```
pZ = []
pZ.append(1/2)
pZ.append(1/2)

print(pZ)
```

```
[0.5, 0.5]
```

Then the joint entropy, $H(X, Y, Z)$ is, at a minium, given by

In [4]:
```
import math

Hxyz = 0
for i in range(2):
    for j in range(2):
        for k in range(2):
            Hxyz += - (pX[i]*pY[j]*pZ[k]) * math.log2(pX[i]*pY[j]*
pZ[k])

print(Hxyz)
```

```
3.0
```

Problem 5

First, we sum the probabilities

$$p_n = \frac{0.1}{n}$$

for $n \in [0, 12367] \subset \mathbb{Z}^+$ to ensure the result is normal (*close to 1*)

In [1]:
```
probSumNow = 0
for i in range(12367):
    probSumNow += 0.1 / (i + 1)

print(probSumNow)
```

```
1.0000043008275796
```

Since the sum of the probabilities is reasonably close to 1, we proceed to calculate the entropies using

$$H(x) = -\sum_{n=1}^{12367} \left\{ \frac{0.1}{n} \log_2 \left[\frac{0.1}{n} \right] \right\}$$

to obtain

In [2]:
```
import math

entropySumNow = 0
for i in range(12367):
    entropySumNow += - (0.1 / (i + 1)) * math.log2(0.1 / (i + 1))

print(entropySumNow)
```

```
9.71625847652071
```

Based on this result and the results in **Problem 1** (***parts A and B***), the entropy of words is bounded above by the entropy of Bi-Grams and below by the entropy of single characters. This would seem to imply that using at most two letters for determining frequencies for Huffman encoding is probably ideal.