In [ ]:
```
import numpy as np
import docx2txt
import matplotlib.pyplot as plt
import math
from queue import PriorityQueue
%matplotlib inline
```

# *Project* 1*B*

## Implemenation of Huffman Coding and Shannon Coding

**By Shinehale(yunsong Yang) (U202115980)**

This programming project is designed for the memento to Steve Jobs, D.A. Huffman and C. E. Shannon. The objective of this programming assignment is to deeply understand the Huffman coding and Shannon coding method. First of all, let us import the data from the target file as the pretty start of this experiment.

In [ ]:
```
str_data = docx2txt.process('./Steve_Jobs_Speech.docx')
str_data = str_data.lower()
str_data = str_data.replace('\n', '')
print('the length of the English file is: ', len(str_data))
```

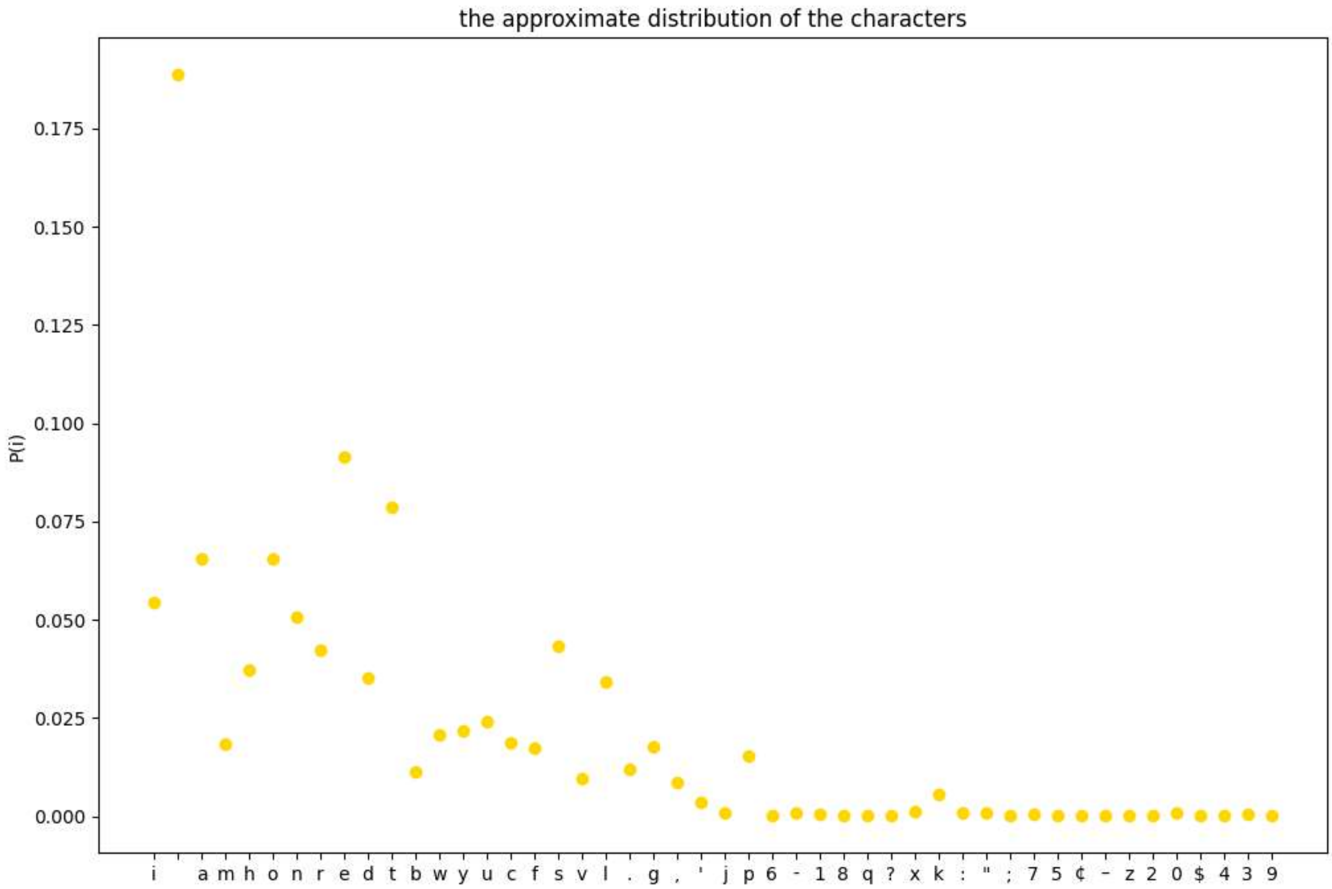
```
the length of the English file is:  11810
```

Okay, from the data above, we can have the view that the length of the string is 11810 after we replace the enter with white character. In order to calculate the entropy of the string, we may review the formula as follow:

$$H(\chi) = -\sum_i P_i log P_i$$

By using the powerful formula, we can easily write the code as bellow:

In [ ]:
```
dict_data = dict()
for each in str_data:
    if dict_data.get(each) == None:
        dict_data[each] = 1
    else:
        dict_data[each] += 1
vals_data = np.array(list(dict_data.values()))
keys_data = np.array(list(dict_data.keys()))
tot_chara = np.sum(vals_data)
P_data = vals_data / tot_chara
plt.figure(figsize=(12, 8))
plt.title('the approximate distribution of the characters')
plt.scatter(keys_data, P_data, color='gold')
plt.ylabel('P(i)')
plt.show()
```




```
In [ ]: H_data = 0.0
        for each in P_data:
            H_data += - each * math.log2(each)
        print('the entropy by using the 0-th markov chain model is %.4f bits'%(H_data))
```

```
the entropy by using the 0-th markov chain model is 4.2395 bits
```

Well, we have the entropy of the document is $H(\chi) = 4.2395 bits$, let us consider the Binary Huffman algorithm, we can let $r$ as the number of the source symbols, and the $P$ as the probability distribution of source symbols(in order to simulate, we will use the frequency as the probability). Generally, we can have iterative or recursive method to implement the method. the pseudocode of the binary huffman code algorithm as follow:

**Input:** $r$: the number of the source symbols; $P$: the probability distribution of source symbols.

**Output:** Output the Huffman codewords $w_i$ corresponding to the source symbols $s_i$.

initialization;

**if** $r == 2$ **then**

  return $s_0 \longmapsto 0$, $s_1 \longmapsto 1$

**else**

  sort $\{p_i\}$ in Descending order;

  reduce source: create a new symbol $s'$ to replace $s_{r-1}$, $s_{r-2}$ with the probability $p' = p_{r-1} + p_{r-2}$;

  Call the Huffman algorithm recursively to obtain the codes of $s_0, \ldots, s_{r-3}, s'$ as $w_0, \ldots, w_{r-3}, w'$ with the corresponding probability distribution $p_0, \ldots, p_{r-3}, p'$;

  return $s_0 \longmapsto w_0$, $s_1 \longmapsto w_1$, …, $s_{r-3} \longmapsto w_{r-3}$, $s_{r-2} \longmapsto w'_0$, $s_{r-1} \longmapsto w'_1$.

**end**

So let dive into the specific code, we will use the Iterative method by using the priority_queue for better efficiency, the time complexity is $O(nlogn)$, while we can abstract the procedure to a general aspect as

Q-ary huffman tree, the only thing need to consider is to add some redundancy to the dataset. the number of the redundancy can be calculated by

$$N = Q - (n \bmod Q)$$

and then change the number of children nodes from 2 to Q is enough. since Python is a dynamically typed language and does not perform type and count checks on variables, we don't care about the preset space corresponding to the number of children node.

In [ ]:
```
# for the inner value we use the google style like XXX_ as the XXX inner the class
class node:
    ## define a tree node class for huffman code
    def __init__(self, prob, val, children=None):
        self.prob_ = prob
        self.char_ = val
        self.children_ = children

    # define the rule for comparing in the priority queue.
    def __lt__(self, other):
        return self.prob_ < other.prob_


# recursive trace the path we can have the codes
def dfs_huf(node, codes, dict_haff):
    if node.prob_ == 0:                   ## filter the redundancy
        return
    if node.char_ == None:             ## this means the tree node is not a leaf node.
        for i in range(len(node.children_)):
            dfs_huf(node.children_[i], codes + str(i), dict_haff)       ## go to the children
    else:
        dict_haff[node.char_] = codes
```

In [ ]:
```
def Q_ary_huffman(datas, Q):
    dict_rv = dict()
    que = PriorityQueue()
    r = len(datas)

    ## put all the character into the priority queue
    for i in range(r):
        que.put(node(datas[i][1], datas[i][0]))

    ## add the redundancy to the priority queue
    N = Q - (r % Q)
    for i in range(N):
        que.put(node(0, None))

    ## construct the Q-ary huffman tree
    while True:
        if que.qsize() == 1:
            break
        nodes = []
        for i in range(Q):
            nodes.append(que.get())                        ## get the first Q nodes to const

        newProb = 0.0
        for i in range(Q):
            newProb += nodes[i].prob_                      ## each time we sum the first Q n

        que.put(node(newProb, None, children=nodes))       ## then put the new node back int

    root = que.get()               # the left one in the priority queue is the root
    dfs_huf(root, "", dict_rv)
    return dict_rv
```

```
In [ ]: # generate the dataset
        r = len(keys_data)
        datas = []
        for i in range(r):
            datas.append((keys_data[i], P_data[i]))

        ## call the method for getting the dict
        dict_haff = Q_ary_huffman(datas, 2)
```

```
In [ ]: # plot all the codes in a table
        cases1 = []
        cases2 = []
        cols = ['char', 'codes', 'prob']
        count = 0
        for i in range(r):
            if count >= r / 2:
                cases2.append([keys_data[i], dict_haff[keys_data[i]], P_data[i]])
            else:
                cases1.append([keys_data[i], dict_haff[keys_data[i]], P_data[i]])
            count += 1

        fig = plt.figure(figsize=(25, 10))
        fig.suptitle('Binary huffman codes', fontsize = 32)

        ax1 = fig.add_subplot(1, 2, 1)
        tab = plt.table(cellText=cases1,
                        colLabels=cols,
                        loc='center',
                        cellLoc='center',
                        rowLoc='center')
        tab.scale(1, 2)
        ax1.axis('off')

        ax2 = fig.add_subplot(1, 2, 2)
        tab = plt.table(cellText=cases2,
                        colLabels=cols,
                        loc='center',
                        cellLoc='center',
                        rowLoc='center')
        tab.scale(1, 2)
        ax2.axis('off')

        plt.show()
```

Binary huffman codes

| char | codes | prob |
|---|---|---|
| i | 0110 | 0.05436071126164268 |
|  | 00 | 0.188653683319221 |
| a | 1000 | 0.06536833192209991 |
| m | 101110 | 0.018289585097375105 |
| h | 11010 | 0.03725656223539373 |
| o | 1001 | 0.06536833192209991 |
| n | 0101 | 0.05055038103302286 |
| r | 11100 | 0.04216765453005927 |
| e | 1111 | 0.09136325148179508 |
| d | 10110 | 0.03530906011854361 |
| t | 1100 | 0.0785774767146486 |
| b | 010010 | 0.01117696867061812 |
| w | 110110 | 0.020745131244707875 |
| y | 01000 | 0.02176121930567316 |
| u | 01110 | 0.023962743437764607 |
| c | 101111 | 0.018543607112616427 |
| f | 101010 | 0.017442845046570704 |
| s | 11101 | 0.043268416596104996 |
| v | 1101110 | 0.009737510584250635 |
| l | 10100 | 0.034038950042337 |
| . | 010011 | 0.012023708721422523 |
| g | 101011 | 0.017527519051651144 |
| , | 0111111 | 0.008552074513124472 |
| ' | 01111100 | 0.003386960203217612 |

| char | codes | prob |
|---|---|---|
| j | 11011111101 | 0.0007620660457239627 |
| p | 011110 | 0.015495342929720575 |
| 6 | 1101111100010 | 0.00016934801016088062 |
| - | 11011111110 | 0.0007620660457239627 |
| 1 | 01111101111 | 0.0005927180355630821 |
| 8 | 110111110010111 | 8.467400508044031e-05 |
| q | 110111110000 | 0.00033869602032176124 |
| ? | 110111111001 | 0.00033869602032176124 |
| x | 1101111101 | 0.001354784081287045 |
| k | 11011110 | 0.00550381033022862 |
| : | 11011111111 | 0.0007620660457239627 |
| " | 0111110110 | 0.0010160880609652837 |
| ; | 1101111100100 | 0.00016934801016088062 |
| 7 | 01111101000 | 0.0004233700254022015 |
| 5 | 011111010010 | 0.00016934801016088062 |
| ¢ | 11011111001010 | 8.467400508044031e-05 |
| – | 011111010011 | 0.0002540220152413209 |
| z | 110111110011 | 0.00033869602032176124 |
| 2 | 1101111110001 | 0.00016934801016088062 |
| 0 | 0111110101 | 0.0009314140558848434 |
| $ | 11011111100000 | 8.467400508044031e-05 |
| 4 | 11011111100001 | 8.467400508044031e-05 |
| 3 | 01111101110 | 0.0005080440304826418 |
| 9 | 1101111100011 | 0.00016934801016088062 |

From the above code, we can have a look at the huffman code method, and apply it into a specific document, finally calculate the codes for each character while there is another famous method for coding named shannon code, which is generated by the information theory godfather Shannon, the pseudocode of the Shannon code algorithm as follow:

**Input:** $r$: the number of the source symbols;
$P = \{p(s_i)\}, i = 1, \ldots, r$: the probability distribution of source symbols.
**Output:** Output the Shannon codewords $w_i$ corresponding to the source symbols $s_i$.
initialization;
sort $\{p(s_i)\}$ in Descending order; **for** $i = 1 \to r$ **do**
  $F(s_i) \leftarrow \sum_{k=1}^{i-1} p(s_k)$;
  $l_i \leftarrow \left\lceil \log \frac{1}{p(s_i)} \right\rceil$;
  Code $F(s_i)$ using binary;
  Take $l_i$ digits after the dot as the codeword for the source symbols $s_i$.
**end**

In [ ]:
```
#self define a function for converting the val to binary presentation
def convert_binary(val, leng):
    each, tot = 1, 0.0
    rv = ""
    for i in range(leng):
        each = each * 0.5
        if tot + each <= val:
            tot += each
            rv += "1"
        else:
            rv += "0"
    return rv
```

In [ ]:
```
P_tuples = []
for i in range(r):
    P_tuples.append(tuple([keys_data[i], P_data[i]]))

P_tuples = sorted(P_tuples, key=lambda x: -x[1])          # use the lambda function for sor

dict_shannon = dict()

tot_F = 0.0
for i in range(r):                                         # just like what we do in the pse
    prob = P_tuples[i][1]
    length = math.ceil(math.log2(1 / prob))
    codes = convert_binary(tot_F, leng=length)
    dict_shannon[P_tuples[i][0]] = codes
    tot_F += prob
```

In [ ]:
```
# plot all the codes in a table
cases1 = []
cases2 = []
cols = ['char', 'codes', 'prob']
count = 0
for i in range(r):
    if count >= r / 2:
        cases2.append([keys_data[i], dict_shannon[keys_data[i]], P_data[i]])
    else:
        cases1.append([keys_data[i], dict_shannon[keys_data[i]], P_data[i]])
    count += 1

fig = plt.figure(figsize=(25, 10))
fig.suptitle('Shannon codes', fontsize = 32)
```

```python
ax1 = fig.add_subplot(1, 2, 1)
tab = plt.table(cellText=cases1,
                colLabels=cols,
                loc='center',
                cellLoc='center',
                rowLoc='center')
tab.scale(1, 2)
ax1.axis('off')

ax2 = fig.add_subplot(1, 2, 2)
tab = plt.table(cellText=cases2,
                colLabels=cols,
                loc='center',
                cellLoc='center',
                rowLoc='center')
tab.scale(1, 2)
ax2.axis('off')

plt.show()
```

Shannon codes

| char | codes | prob |
|---|---|---|
| i | 01111 | 0.05436071126164268 |
|  | 000 | 0.188653683319221 |
| a | 0101 | 0.06536833192209991 |
| m | 110111 | 0.018289585097375105 |
| h | 10101 | 0.03725656223539373 |
| o | 0110 | 0.06536833192209991 |
| n | 10001 | 0.05055038103302286 |
| r | 10100 | 0.04216765453005927 |
| e | 0011 | 0.09136325148179508 |
| d | 10110 | 0.03530906011854361 |
| t | 0100 | 0.0785774767146486 |
| b | 1111001 | 0.01117696867061812 |
| w | 110101 | 0.020745131244707875 |
| y | 110011 | 0.02176121930567316 |
| u | 110010 | 0.023962743437764607 |
| c | 110110 | 0.018543607112616427 |
| f | 111010 | 0.017442845046570704 |
| s | 10011 | 0.043268416596104996 |
| v | 1111011 | 0.009737510584250635 |
| l | 11000 | 0.034038950042337 |
| . | 1111000 | 0.012023708721422523 |
| g | 111000 | 0.017527519051651144 |
| , | 1111100 | 0.008552074513124472 |
| ' | 111111001 | 0.003386960203217612 |

| char | codes | prob |
|---|---|---|
| j | 11111110011 | 0.0007620660457239627 |
| p | 1110110 | 0.015495342929720575 |
| 6 | 1111111110110 | 0.00016934801016088062 |
| - | 11111110100 | 0.0007620660457239627 |
| 1 | 11111110111 | 0.0005927180355630821 |
| 8 | 11111111111010 | 8.467400508044031e-05 |
| q | 111111110101 | 0.00033869602032176124 |
| ? | 111111110111 | 0.00033869602032176124 |
| x | 1111110110 | 0.001354784081287045 |
| k | 11111011 | 0.00550381033022862 |
| : | 11111110110 | 0.0007620660457239627 |
| " | 1111110111 | 0.0010160880609652837 |
| ; | 1111111110111 | 0.00016934801016088062 |
| 7 | 111111110100 | 0.0004233700254022015 |
| 5 | 1111111111001 | 0.00016934801016088062 |
| ¢ | 11111111111011 | 8.467400508044031e-05 |
| – | 111111111010 | 0.0002540220152413209 |
| z | 111111111000 | 0.00033869602032176124 |
| 2 | 1111111111010 | 0.00016934801016088062 |
| 0 | 11111110001 | 0.0009314140558848434 |
| $ | 11111111111101 | 8.467400508044031e-05 |
| 4 | 11111111111110 | 8.467400508044031e-05 |
| 3 | 11111111001 | 0.0005080440304826418 |
| 9 | 1111111111011 | 0.00016934801016088062 |

Yeah, the shannon codes has shown above in the table format, finnally we should calculate the average code length for huffman and shannon codes method respectively, and calculate their own encoding effeciency with the entropy

In [ ]:
```python
codes_huffman = 0
for i in range(r):
    codes_huffman += len(dict_haff[keys_data[i]]) * P_data[i]
print('the average code length of huffman is %.4f bits' %codes_huffman)
print('shannon code efficiency is %.4f%%' %(H_data / codes_huffman * 100))

codes_shannon = 0
for i in range(r):
    codes_shannon += len(dict_shannon[keys_data[i]]) * P_data[i]
print('the average code length of shannon is %.4f bits' %codes_shannon)
print('shannon code efficiency is %.4f%%' %(H_data / codes_shannon * 100))
```

```
the average code length of huffman is 4.2809 bits
shannon code efficiency is 99.0316%
the average code length of shannon is 4.6638 bits
shannon code efficiency is 90.9028%
```

From the data above, we can find that

- the average code length of huffman is $4.2809 bits$, the effeciency is $99.0316\%$

- the average code length of shannon is $4.6638 bits$, the effeciency is $90.9028\%$