Project Part-I

CSM-322: Information and Coding Theory

September 26, 2022

Gandharv Jain
20124018

Question 1

```python
def parseInput(d):
        codewords = list()
        pairs = d.strip().strip('{}').split(',')
        for i in pairs: codewords.append(i.split(':')[1].strip().strip('\''))
        return codewords

codewords = parseInput(input("Enter encoding (Ex - {A: '1', B: '0'}): "))
codewords.sort(key = len)
n = len(codewords)

for i in range(n):
        for j in range(i + 1, n):
                c1 = codewords[i]
                c2 = codewords[j]
                if c2.startswith(c1):
                        print("No, {} has prefix {}".format(c2, c1))
                        exit()
print("Yes")
```

```
Input: {A: '1', B: '0'}
Output: Yes
Input: {A: '1', B: '11'}
Output: No, 11 has prefix 1
```

Question 2

```python
def parseInput(d):
        codewords = list()
        message, encoding = d.split(',', 1)
        message = message.strip().strip('\'')
        pairs = encoding.strip().strip('{}').split(',')
        for i in pairs: codewords.append(i.split(':')[1].strip().strip('\''))
        return message, codewords

message, codewords = parseInput(input("Enter word and encoding (Ex - '01', {A: '1', B: '0'}): "))
count = 0
def checkCodewordPrefix(msg):
        global count
        if (len(msg) == 0):
                count = count + 1
                return
        for c in codewords:
                if msg.startswith(c): checkCodewordPrefix(msg[len(c):])

checkCodewordPrefix(message)
print(count)
```

```
Input: '01', {A: '1' , B: '0'}
Output: 1
Input: '11', {A: '1', B: '11'}
Output: 2
```

Question 3

```python
import queue as Q

def parseInput(d):
        source = dict()
        pairs = d.strip().strip('{}').split(',')
        for i in pairs:
                pair = i.split(':')
                source[pair[0].strip().strip('\'')] = float(pair[1].strip().strip('\''))
        return source

class node:
        def __init__(self, frq, sym, left=None, right=None):
                self.frq = frq
                self.sym = sym
                self.left = left
                self.right = right
        def __lt__(self, other): return self.frq < other.frq

def assignCodes(node, code=''):
        global huffman_code
        if not node.left and not node.right: huffman_code[node.sym] = code
        if node.left: assignCodes(node.left, code + '0')
        if node.right: assignCodes(node.right, code + '1')

source = parseInput(input("Enter symbols and frequencies (Ex - {A: 12, B: 5}): "))
pq = Q.PriorityQueue()
huffman_code = dict()

for alphabet, frequency in source.items(): pq.put(node(frequency, alphabet))
while pq.qsize() > 1:
        left = pq.get()
        right = pq.get()
        pq.put(node(left.frq + right.frq, left.sym + right.sym, left, right))

assignCodes(pq.get())
print(huffman_code)
```

```
Input: {A: 12, B: 5}
Output: {'B': '0', 'A': '1'}
Input: {S1: 0.3, S2: 0.2, S3: 0.2, S4: 0.2, S5: 0.1}
Output: {'S4': '00', 'S2': '01', 'S1': '10', 'S5': '110', 'S3': '111'}
```

Question 4

```python
import queue as Q
from math import log

def parseInput(d):
        source = dict()
        pairs = d.strip().strip('{}').split(',')
        for i in pairs:
                pair = i.split(':')
                source[pair[0].strip().strip('\'')] = float(pair[1].strip().strip('\''))
        return source

class node:
        def __init__(self, frq, sym, left=None, right=None):
                self.frq = frq
                self.sym = sym
                self.left = left
                self.right = right
        def __lt__(self, other): return self.frq < other.frq

def assignCodes(node, code=''):
        global huffman_code
        if not node.left and not node.right: huffman_code[node.sym] = code
        if node.left: assignCodes(node.left, code + '0')
        if node.right: assignCodes(node.right, code + '1')

source = parseInput(input("Enter symbols and frequencies (Ex - {A: 12, B: 5}): "))
pq = Q.PriorityQueue()
huffman_code = dict()

for alphabet, frequency in source.items(): pq.put(node(frequency, alphabet))
while pq.qsize() > 1:
        left = pq.get()
        right = pq.get()
        pq.put(node(left.frq + right.frq, left.sym + right.sym, left, right))
assignCodes(pq.get())

f_total = sum(source.values())
for k in source: source[k] /= f_total
huffman_avg_word_len = sum(len(v) * source[k] for k, v in huffman_code.items())
source_entropy = sum(-1 * p * log(p, 2) for p in source.values())

print(f"Entropy of Huffman code: {round(huffman_avg_word_len, 8)}")
print(f"Entropy of optimal structure: {round(source_entropy, 8)}")
```

```
Input: {A: 12, B: 5}
Output:
    Entropy of Huffman code: 1.0
    Entropy of optimal structure: 0.87398105
Input: {S1: 0.3, S2: 0.2, S3: 0.2, S4: 0.2, S5: 0.1}
Output:
    Entropy of Huffman code: 2.3
    Entropy of optimal structure: 2.24643934
```

Question 5

```python
def parseInput(d):
        elements = d.strip().strip('{}').split(',')
        codewords = list()
        for i in elements: codewords.append(i.strip())
        return codewords

def HammingDistance(x, y):
        ans = 0
        for i in range(32):
                if (x>>i)&1 != (y>>i)&1: ans += 1
        return ans

codewords = parseInput(input("Enter encoding (Ex - {1, 3, 5}): "))
n = len(codewords)
s = sum(HammingDistance(i, j) for i in range(n) for j in range(n))

print(f"Sum of Hamming distances: {s}")
```

```
Input: {1, 3, 5}
Output: Sum of Hamming distances: 8
Input: {1, 2, 3, 4, 5}
Output: Sum of Hamming distances: 32
```

Question 6

```python
def parseInput(d):
        d, recv_word = d.strip().rsplit(',', 1)
        recv_word = recv_word.strip()
        elements = d.strip().strip('{}').split(',')
        codewords = list()
        for i in elements: codewords.append(i.strip())
        return recv_word, codewords

def HammingDistance(x, y):
        ans = 0
        for i in range(len(x)):
                if x[i] != y[i]: ans += 1
        return ans

recv_word, codewords = parseInput(input("Enter encoding (Ex - {000, 111}, 110): "))
p = 0.05
sent_word = ""
max_fwd_prob = -1
for c in codewords:
        t = HammingDistance(recv_word, c)
        fwd_prob = (p ** t) * ((1 - p) ** (len(recv_word) - t))
        if max_fwd_prob < fwd_prob:
                sent_word = c
                max_fwd_prob = fwd_prob

print(f"Most likely codeword sent using MLD rule with p = {p}: {sent_word}")
```

```
Input: {000, 111}, 110
Output: Most likely codeword sent using MLD rule with p = 0.05: 111
Input: {000, 111}, 110
Output: Most likely codeword sent using MLD rule with p = 0.95: 000
```

Question 7

```python
def parseInput(d):
        d, recv_word = d.strip().rsplit(',', 1)
        recv_word = recv_word.strip()
        elements = d.strip().strip('{}').split(',')
        codewords = list()
        for i in elements: codewords.append(i.strip())
        return recv_word, codewords

def HammingDistance(x, y):
        ans = 0
        for i in range(len(x)):
                if x[i] != y[i]: ans += 1
        return ans

recv_word, codewords = parseInput(input("Enter encoding (Ex - {000, 111}, 110): "))

sent_word = ""
min_hamming_distance = len(recv_word) + 1
for c in codewords:
        t = HammingDistance(recv_word, c)
        if min_hamming_distance > t:
                sent_word = c
                min_hamming_distance = t

print(f"Most likely codeword sent using MDD rule: {sent_word}")
```

```
Input: {000, 111}, 110
Output: Most likely codeword sent using MDD rule: 111
Input: {000, 111}, 100
Output: Most likely codeword sent using MDD rule: 000
```