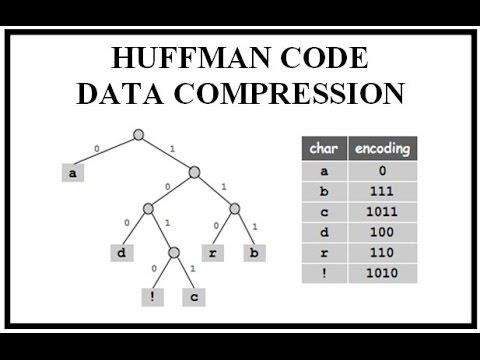
## Practical 6

### **Aim:** Write a program to implement Huffman’s code algorithm

**Theory:**

1. Huffman coding is a lossless data compression algorithm.
2. The idea is to assign variable-length codes to input characters, lengths of the assigned codes are based on the frequencies of corresponding characters.
3. The most frequent character gets the smallest code and the least frequent character gets the largest code.
4. Huffman coding is a method of data compression that is independent of the data type, that is, the data could represent an image, audio or spreadsheet.
5. This compression scheme is used in JPEG and MPEG-2. Huffman coding works by looking at the data stream that makes up the file to be compressed.

**Example:**



**Algorithm:**

Huffman(C)

1. n = |C|
2. Q = C
3. for i = 1 to n - 1
4. allocate a new node z
5. z.left = x = EXTRACT-MIN(Q)
6. z.right = y = EXTRACT-MIN(Q)
7. z.freq = x.freq + y.freq
8. INSERT(Q,z)
9. return EXTRACT-MIN(Q) // return the root of the tree

**Code:**

# Huffman Coding in python

string = 'abcdef'

# Creating tree nodes

class NodeTree(object):

def \_\_init\_\_(self, left=None, right=None):

self.left = left

self.right = right

def children(self):

return (self.left, self.right)

def nodes(self):

return (self.left, self.right)

def \_\_str\_\_(self):

return '%s\_%s' % (self.left, self.right)

# Main function implementing huffman coding

def huffman\_code\_tree(node, left=True, binString=''):

if type(node) is str:

return {node: binString}

(l, r) = node.children()

d = dict()

d.update(huffman\_code\_tree(l, True, binString + '0'))

d.update(huffman\_code\_tree(r, False, binString + '1'))

return d

# Calculating frequency

freq = {}

for c in string:

if c in freq:

freq[c] += 1

else:

freq[c] = 1

freq = sorted(freq.items(),

key=lambda x:

x[1], reverse=True)

nodes = freq

while len(nodes) > 1:

(key1, c1) = nodes[-1]

(key2, c2) = nodes[-2]

nodes = nodes[:-2]

node = NodeTree(key1, key2)

nodes.append((node, c1 + c2))

nodes = sorted(nodes, key=lambda x: x[1], reverse=True)

huffmanCode = huffman\_code\_tree(nodes[0][0])

print(' Char | Huffman code ')

print('----------------------')

for (char, frequency) in freq:

print(' %-4r |%12s' % (char, huffmanCode[char]))

print("Neeraj Appari 021")

**Output**:

Char | Huffman code

----------------------

'a' | 101

'b' | 100

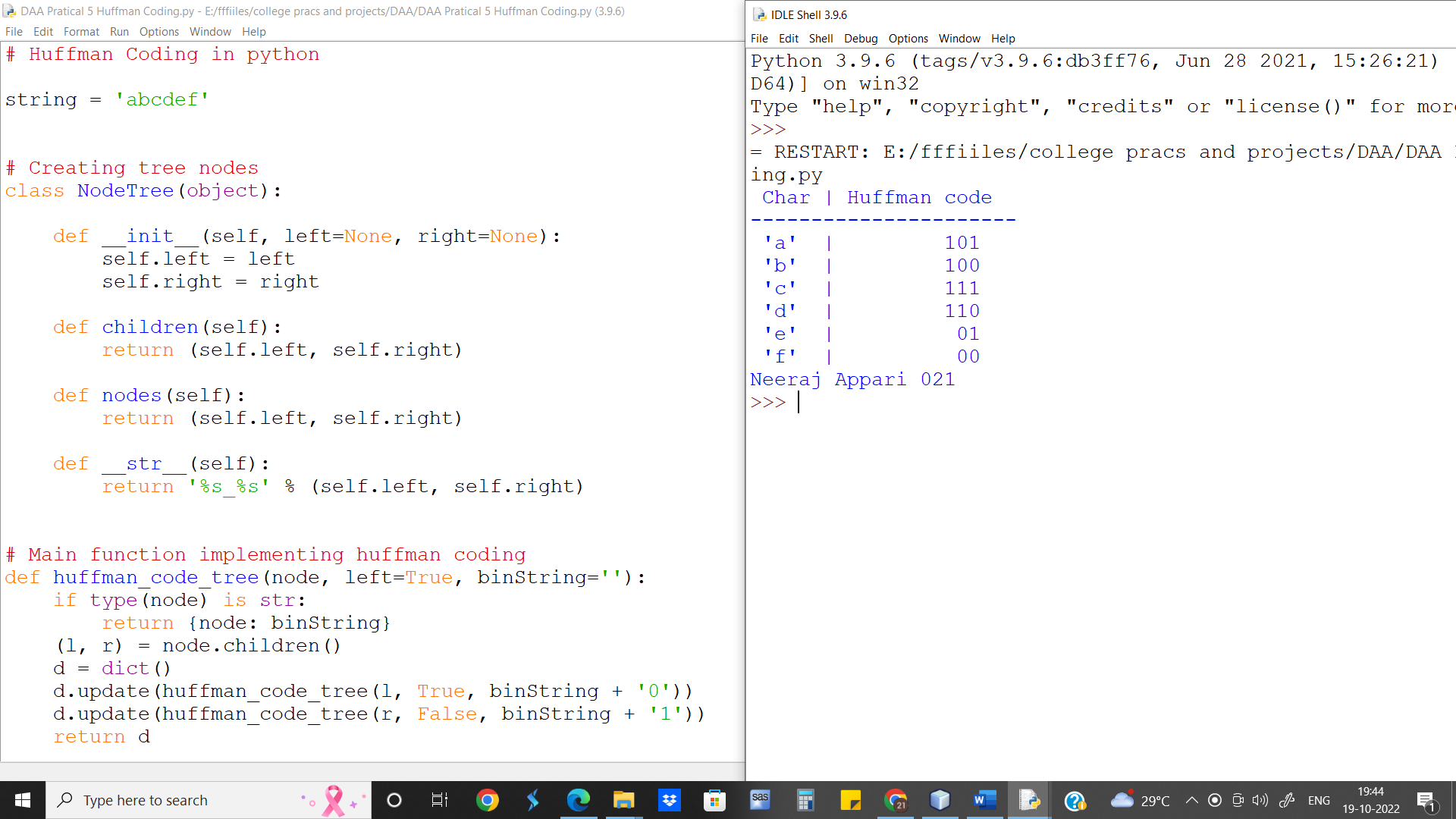
'c' | 111

'd' | 110

'e' | 01

'f' | 00

Neeraj Appari 021



**Runtime for Longest Common Subsequence Recursive approach is O(n logn)**

**Conclusion:** We implement Huffman Code and find the encoded form of strings and analysed its complexity which is O(n logn)