**Practical-3**

**AIM: Implement Huffman Code(HC) to generate binary code when symbol and probabilities are given.**

**CODE:**

import heapq

class node:

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

self.freq = freq

self.symbol = symbol

self.left = left

self.right = right

self.huff = ''

def \_\_lt\_\_(self, nxt):

return self.freq < nxt.freq

def printNodes(node, val=''):

newVal = val + str(node.huff)

if(node.left):

printNodes(node.left, newVal)

if(node.right):

printNodes(node.right, newVal)

if(not node.left and not node.right):

print(f"{node.symbol} -> {newVal}")

chars = ['y', 'a', 's', 'h', 'p', 't']

freq = [ 25, 6, 50, 20, 10, 95]

nodes = []

for x in range(len(chars)):

heapq.heappush(nodes, node(freq[x], chars[x]))

while len(nodes) > 1:

left = heapq.heappop(nodes)

right = heapq.heappop(nodes)

left.huff = 0

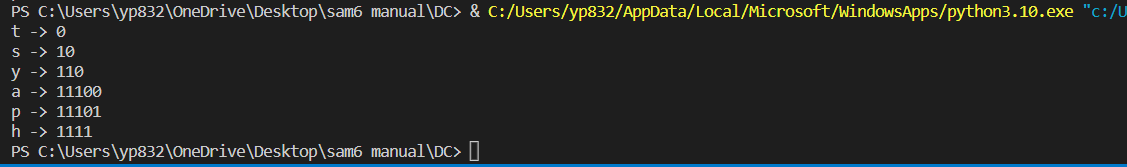
right.huff = 1

newNode = node(left.freq+right.freq, left.symbol+right.symbol, left, right)

heapq.heappush(nodes, newNode)

printNodes(nodes[0])

**OUTPUT:**



**Practical-4**

**AIM: Implement Huffman code which can compress given file and**

**decompress compressed file.**

**CODE:**

class Nodes:

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

self.probability = probability

self.symbol = symbol

self.left = left

self.right = right

self.code = ''

def CalculateProbability(the\_data):

the\_symbols = dict()

for item in the\_data:

if the\_symbols.get(item) == None:

the\_symbols[item] = 1

else:

the\_symbols[item] += 1

return the\_symbols

the\_codes = dict()

def CalculateCodes(node, value = ''):

newValue = value + str(node.code)

if(node.left):

CalculateCodes(node.left, newValue)

if(node.right):

CalculateCodes(node.right, newValue)

if(not node.left and not node.right):

the\_codes[node.symbol] = newValue

return the\_codes

def OutputEncoded(the\_data, coding):

encodingOutput = []

for element in the\_data:

encodingOutput.append(coding[element])

the\_string = ''.join([str(item) for item in encodingOutput])

return the\_string

def TotalGain(the\_data, coding):

beforeCompression = len(the\_data) \* 8

afterCompression = 0

the\_symbols = coding.keys()

for symbol in the\_symbols:

the\_count = the\_data.count(symbol)

afterCompression += the\_count \* len(coding[symbol])

print("Space usage before compression (in bits):", beforeCompression)

print("Space usage after compression (in bits):", afterCompression)

def HuffmanEncoding(the\_data):

symbolWithProbs = CalculateProbability(the\_data)

the\_symbols = symbolWithProbs.keys()

the\_probabilities = symbolWithProbs.values()

print("symbols: ", the\_symbols)

print("probabilities: ", the\_probabilities)

the\_nodes = []

for symbol in the\_symbols:

the\_nodes.append(Nodes(symbolWithProbs.get(symbol), symbol))

while len(the\_nodes) > 1:

the\_nodes = sorted(the\_nodes, key = lambda x: x.probability)

right = the\_nodes[0]

left = the\_nodes[1]

left.code = 0

right.code = 1

newNode = Nodes(left.probability + right.probability, left.symbol + right.symbol, left, right)

the\_nodes.remove(left)

the\_nodes.remove(right)

the\_nodes.append(newNode)

huffmanEncoding = CalculateCodes(the\_nodes[0])

print("symbols with codes", huffmanEncoding)

TotalGain(the\_data, huffmanEncoding)

encodedOutput = OutputEncoded(the\_data,huffmanEncoding)

return encodedOutput, the\_nodes[0]

def HuffmanDecoding(encodedData, huffmanTree):

treeHead = huffmanTree

decodedOutput = []

for x in encodedData:

if x == '1':

huffmanTree = huffmanTree.right

elif x == '0':

huffmanTree = huffmanTree.left

try:

if huffmanTree.left.symbol == None and huffmanTree.right.symbol == None:

pass

except AttributeError:

decodedOutput.append(huffmanTree.symbol)

huffmanTree = treeHead

string = ''.join([str(item) for item in decodedOutput])

return string

the\_data = input("enter the string")

print(the\_data)

encoding, the\_tree = HuffmanEncoding(the\_data)

print("Encoded output", encoding)

print("Decoded Output", HuffmanDecoding(encoding, the\_tree))

**OUTPUT:**

****

**Practical-5**

**AIM: Implement adaptive Huffman program to compress decompressed file.**

**CODE:**

class Nodes:

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

self.probability = probability

self.symbol = symbol

self.left = left

self.right = right

self.code = ''

def CalculateProbability(the\_data):

the\_symbols = dict()

for item in the\_data:

if the\_symbols.get(item) == None:

the\_symbols[item] = 1

else:

the\_symbols[item] += 1

return the\_symbols

the\_codes = dict()

def CalculateCodes(node, value = ''):

newValue = value + str(node.code)

if(node.left):

CalculateCodes(node.left, newValue)

if(node.right):

CalculateCodes(node.right, newValue)

if(not node.left and not node.right):

the\_codes[node.symbol] = newValue

return the\_codes

def OutputEncoded(the\_data, coding):

encodingOutput = []

for element in the\_data:

encodingOutput.append(coding[element])

the\_string = ''.join([str(item) for item in encodingOutput])

return the\_string

def TotalGain(the\_data, coding):

beforeCompression = len(the\_data) \* 8

afterCompression = 0

the\_symbols = coding.keys()

for symbol in the\_symbols:

the\_count = the\_data.count(symbol)

afterCompression += the\_count \* len(coding[symbol])

print("Space usage before compression (in bits):", beforeCompression)

print("Space usage after compression (in bits):", afterCompression)

def HuffmanEncoding(the\_data):

symbolWithProbs = CalculateProbability(the\_data)

the\_symbols = symbolWithProbs.keys()

the\_probabilities = symbolWithProbs.values()

print("symbols: ", the\_symbols)

print("probabilities: ", the\_probabilities)

the\_nodes = []

for symbol in the\_symbols:

the\_nodes.append(Nodes(symbolWithProbs.get(symbol), symbol))

while len(the\_nodes) > 1:

the\_nodes = sorted(the\_nodes, key = lambda x: x.probability)

right = the\_nodes[0]

left = the\_nodes[1]

left.code = 0

right.code = 1

newNode = Nodes(left.probability + right.probability, left.symbol + right.symbol, left, right)

the\_nodes.remove(left)

the\_nodes.remove(right)

the\_nodes.append(newNode)

huffmanEncoding = CalculateCodes(the\_nodes[0])

print("symbols with codes", huffmanEncoding)

TotalGain(the\_data, huffmanEncoding)

encodedOutput = OutputEncoded(the\_data,huffmanEncoding)

return encodedOutput, the\_nodes[0]

def HuffmanDecoding(encodedData, huffmanTree):

treeHead = huffmanTree

decodedOutput = []

for x in encodedData:

if x == '1':

huffmanTree = huffmanTree.right

elif x == '0':

huffmanTree = huffmanTree.left

try:

if huffmanTree.left.symbol == None and huffmanTree.right.symbol == None:

pass

except AttributeError:

decodedOutput.append(huffmanTree.symbol)

huffmanTree = treeHead

string = ''.join([str(item) for item in decodedOutput])

return string

the\_data = input("enter the string")

print(the\_data)

encoding, the\_tree = HuffmanEncoding(the\_data)

print("Encoded output", encoding)

print("Decoded Output", HuffmanDecoding(encoding, the\_tree))

**OUTPUT:**

****

**Practical-2**

**AIM: Write a program to generate binary code in case of arithmetic coding.**

**CODE:**

#include<iostream>

#include<unordered\_map>

#include<vector>

using namespace std;

struct node{

double prob, range\_from, range\_to;};

double encoding(unordered\_map<char, node> arr, string s){

cout<<"\nEncoding\n";

double low\_v=0.0, high\_v=1.0, diff= 1.0;

cout<<"Symbol\tLow\_v\tHigh\_v\tdiff\n";

for(int i=0; i<s.size(); i++){

high\_v= low\_v+ diff\* arr[s[i]].range\_to;

low\_v= low\_v+ diff\* arr[s[i]].range\_from;

diff= high\_v- low\_v;

cout<<s[i]<<"\t"<<low\_v<<"\t"<<high\_v<<"\t"<<diff<<endl; }

return low\_v;}

string decoding(unordered\_map<char, node> arr, double code\_word, int len){

cout<<"\nDecoding: \n";

char ch;

string text= "";

int j=0;

unordered\_map<char, node>:: iterator it;

cout<<"Code\tOutput\tRange\_from\tRange\_to\n";

while(j<len){

cout<<code\_word<<"\t";

for(it= arr.begin(); it!=arr.end(); it++){

char i= (\*it).first;

if(arr[i].range\_from<= code\_word && code\_word< arr[i].range\_to){

ch= i;

code\_word= (code\_word-arr[i].range\_from)/(arr[i].range\_to- arr[i].range\_from);

break;

}}

cout<<ch<<"\t"<<arr[ch].range\_from<<"\t\t"<<arr[ch].range\_to<<endl;

text+= ch;

j++;

}

return text;

}

int main(){

int n;

cout<<"Enter number of characters: ";

cin>>n;

unordered\_map<char, node> arr;

vector<char> ar;

double range\_from= 0;

cout<<"Enter probability of each character:\n";

for(int i=0; i<n; i++){

char ch;

cin>>ch;

ar.push\_back(ch);

cin>>arr[ch].prob;

arr[ch].range\_from= range\_from;

arr[ch].range\_to= range\_from+ arr[ch].prob;

range\_from= arr[ch].range\_to;

}

cout<<"Symbol\tProbability\tRange\_from\tRange\_to\n";

cout<<"----------------------------------------------------\n";

for(int i=0; i<ar.size(); i++){

char ch= ar[i];

cout<<ch<<"\t"<<arr[ch].prob<<"\t\t"<<arr[ch].range\_from<<"\t\t"<<arr[ch].range\_to<<endl;

}

cout<<endl;

string s;

cout<<"Enter text: ";

cin>>s;

double code\_word= encoding(arr, s);

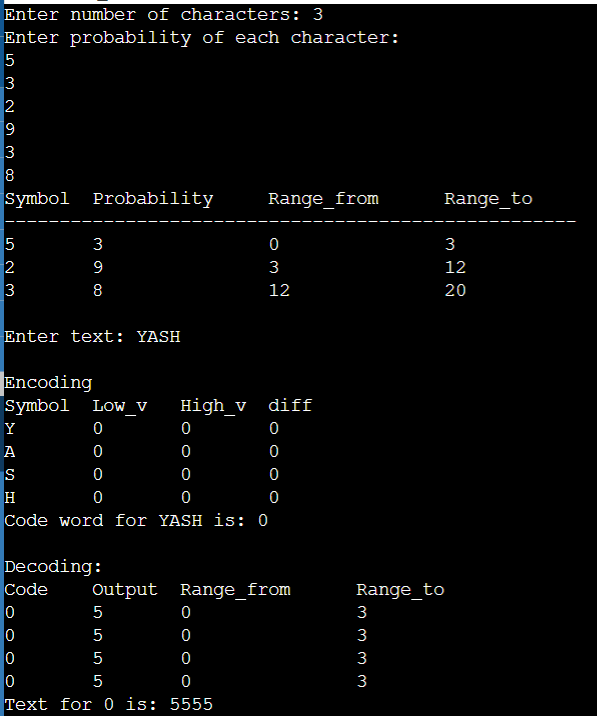
cout<<"Code word for "<<s<<" is: "<<code\_word<<endl;

string text= decoding(arr, code\_word, s.size());

cout<<"Text for "<<code\_word<<" is: "<<text<<endl;

}

**OUTPUT:**

****

**Practical-6**

**AIM: Write a program to Implement LZ77 algorithm.**

**CODE:**

def compress(data):

compressed\_data = []

search\_buffer = ""

lookahead\_buffer = ""

search\_buffer\_size = 10

lookahead\_buffer\_size = 5

for i in range(len(data)):

lookahead\_buffer += data[i]

if len(lookahead\_buffer) == lookahead\_buffer\_size or i == len(data) - 1:

best\_match\_length = 0

best\_match\_position = 0

for j in range(len(search\_buffer)):

match\_length = 0

for k in range(len(lookahead\_buffer)):

if search\_buffer[j+k] != lookahead\_buffer[k]:

break

match\_length += 1

if match\_length > best\_match\_length:

best\_match\_length = match\_length

best\_match\_position = j

compressed\_data.append((best\_match\_position, best\_match\_length, lookahead\_buffer[best\_match\_length:]))

search\_buffer += lookahead\_buffer

search\_buffer = search\_buffer[-search\_buffer\_size:]

lookahead\_buffer = ""

return compressed\_data

def decompress(compressed\_data):

decompressed\_data = ""

search\_buffer = ""

for i in range(len(compressed\_data)):

match\_position, match\_length, c = compressed\_data[i]

decompressed\_data += search\_buffer[match\_position:match\_position+match\_length]

search\_buffer += c

decompressed\_data += c

return decompressed\_data

data = "ABCDEFABCRDEDDFHTFDDD"

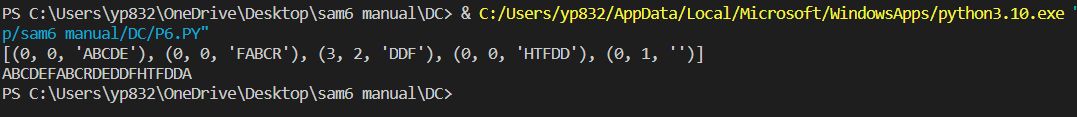
compressed\_data = compress(data)

print(compressed\_data)

decompressed\_data = decompress(compressed\_data)

print(decompressed\_data)

**OUTPUT:**



**Practical-7**

**AIM: Write a program to Implement LZW algorithm.**

**CODE:**

def lzw\_compress(text):

dictionary = {chr(i): i for i in range(256)}

next\_index = 256

result = []

i = 0

while i < len(text):

j = 1

while j <= len(text) - i:

if text[i:i+j] in dictionary:

j += 1

else:

break

index = dictionary.get(text[i:i+j-1], None)

if index is None:

index = ord(text[i])

result.append(index)

dictionary[text[i:i+j]] = next\_index

next\_index += 1

i += j - 1

return result

def lzw\_decompress(compressed):

dictionary = {i: chr(i) for i in range(256)}

next\_index = 256

result = ""

string = chr(compressed[0])

result += string

for index in compressed[1:]:

if index in dictionary:

next\_string = dictionary[index]

elif index == next\_index:

next\_string = string + string[0]

else:

raise ValueError("Compressed data is corrupted")

result += next\_string

dictionary[next\_index] = string + next\_string[0]

next\_index += 1

string = next\_string

return result

text = "MISSISSIPI"

compressed = lzw\_compress(text)

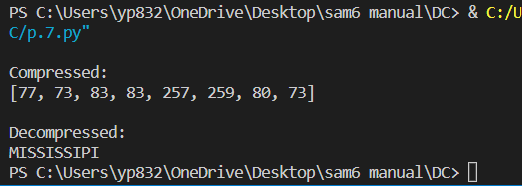
print("\nCompressed:")

print(compressed)

print("\nDecompressed:")

print(lzw\_decompress(compressed))

**OUTPUT:**

****

**Practical-8**

**AIM: Write a program to Implement LZ78 algorithm.**

**CODE:**

def lz78\_compress(text):

dictionary = {char: idx for idx, char in enumerate(set(text))}

result = []

i = 0

while i < len(text):

j = i + 1

while j <= len(text) and text[i:j] in dictionary:

j += 1

if j > len(text):

break

dictionary[text[i:j]] = len(dictionary) + 1

result.append((dictionary[text[i:j - 1]], text[j - 1]))

i = j

return result

def lz78\_decompress(compressed):

dictionary = {idx: char for idx, char in enumerate(set(text))}

result = ""

for code, next\_char in compressed:

if code in dictionary:

result += dictionary[code] + next\_char

dictionary[len(dictionary) + 1] = dictionary[code] + next\_char

else:

result += next\_char

dictionary[len(dictionary) + 1] = next\_char

return result

text = "MISSISSIPI"

compressed = lz78\_compress(text)

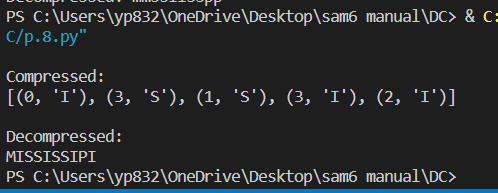
print("\nCompressed:")

print(compressed)

print("\nDecompressed:")

print(lz78\_decompress(compressed))

**OUTPUT:**



**Practical-1**

**AIM: Write a program to Implement BWT algorithm.**

**CODE:**

a = input("Enter a string:")

words = list(a)

list = []

for i in range(len(words)):

word = a[-1] + a[:-1]

new = ''.join(word)

a = new

list.append(new)

i += 1

print(list)

sort = sorted(list)

print(sort)

for i in range(len(words)):

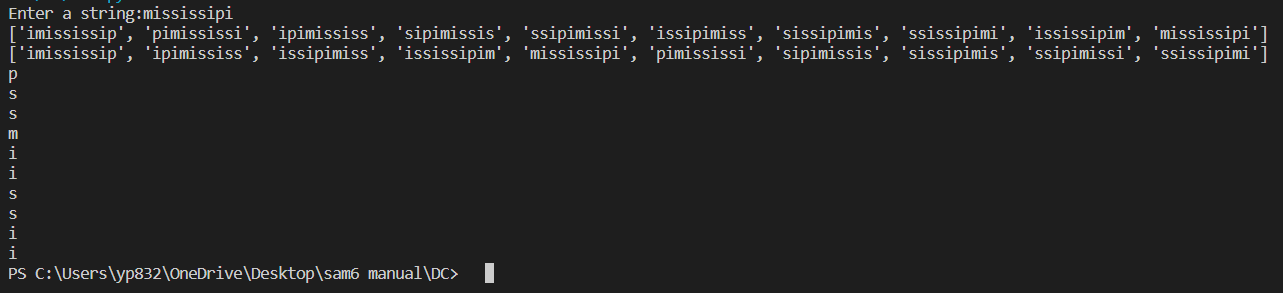
element = sort[i]

last = element[- 1]

i = i + 1

print(last)

**OUTPUT:**



**Practical-10**

**AIM: Write a program to Implement MTF algorithm.**

**CODE:**

def search(input\_char, char\_list):

for i in range(len(char\_list)):

if char\_list[i] == input\_char:

return i

def move\_to\_front(curr\_index, char\_list):

char = char\_list.pop(curr\_index)

char\_list.insert(0, char)

def mtf\_encode(input\_text, char\_list):

output\_arr = []

for char in input\_text:

output\_arr.append(search(char, char\_list))

move\_to\_front(output\_arr[-1], char\_list)

return output\_arr

if \_\_name\_\_ == '\_\_main\_\_':

input\_text = "panama"

char\_list = list("abcdefghijklmnopqrstuvwxyz")

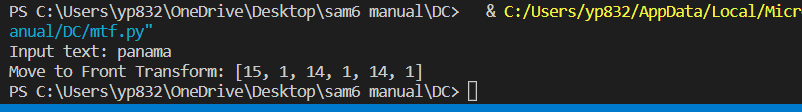
print(f"Input text: {input\_text}")

print("Move to Front Transform:", end=" ")

mtf\_transform = mtf\_encode(input\_text, char\_list)

print(mtf\_transform)

**OUTPUT:**



**Practical-9**

**AIM: Write a program which performs JPEG compression, process step bystep**

**for given 8x8 block and decompression also.**

**CODE:**

from heapq import heappush, heappop, heapify

from collections import defaultdict

import numpy as np

def huffman\_encoding(arr):

arr\_list = arr.flatten().tolist()

freq = defaultdict(int)

for symbol in arr\_list:

freq[symbol] += 1

heap = [[wt, [sym, ""]] for sym, wt in freq.items()]

heapify(heap)

while len(heap) > 1:

lo = heappop(heap)

hi = heappop(heap)

for pair in lo[1:]:

pair[1] = '0' + pair[1]

for pair in hi[1:]:

pair[1] = '1' + pair[1]

heappush(heap, [lo[0] + hi[0]] + lo[1:] + hi[1:])

huffman\_dict = dict(heappop(heap)[1:])

encoded\_list = [huffman\_dict[symbol] for symbol in arr\_list]

encoded\_str = "".join(encoded\_list)

return encoded\_str, huffman\_dict

def huffman\_decoding(encoded\_str, huffman\_dict):

reverse\_dict = {v: k for k, v in huffman\_dict.items()}

code = ""

decoded\_list = []

for bit in encoded\_str:

code += bit

if code in reverse\_dict:

symbol = reverse\_dict[code]

decoded\_list.append(symbol)

code = ""

decoded\_arr = np.array(decoded\_list).reshape((8,8))

return decoded\_arr

arr = np.array([[52, 55, 61, 66, 70, 61, 64, 73],

[63, 59, 55, 90, 109, 85, 69, 72],

[62, 59, 68, 113, 144, 104, 66, 73],

[63, 58, 71, 122, 154, 106, 70, 69],

[67, 61, 68, 104, 126, 88, 68, 70],

[79, 65, 60, 70, 77, 68, 58, 75],

[85, 71, 64, 59, 55, 61, 65, 83],

[87, 79, 69, 68, 65, 76, 78, 94]])

encoded\_str, huffman\_dict = huffman\_encoding(arr)

print("Encoded String:\n", encoded\_str)

print("Huffman Dictionary:\n", huffman\_dict)

decoded\_arr = huffman\_decoding(encoded\_str, huffman\_dict)

print("Decoded Array:\n", decoded\_arr)

**OUTPUT:**

