Practical-3

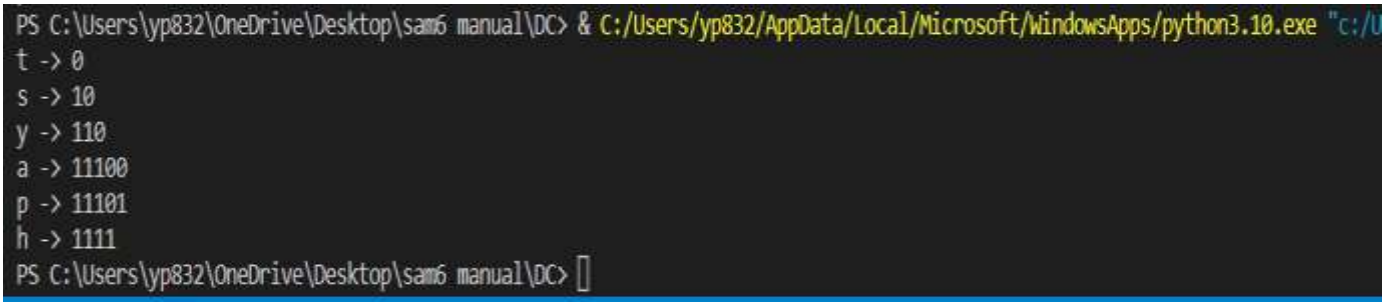
AIM: Implement Huffman Code(HC) to generate binary code when symbol and probabilities are given.

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:

```
Enter number of characters: 3
Enter probability of each character:
5
3
2
9
3
8
Symbol   Probability      Range_from       Range_to
-------------------------------------------------------
5        3                0                3
2        9                3                12
3        8                12               20

Enter text: YASH

Encoding
Symbol   Low_v    High_v   diff
Y        0        0        0
A        0        0        0
S        0        0        0
H        0        0        0
Code word for YASH is: 0

Decoding:
Code     Output   Range_from       Range_to
0        5        0                3
0        5        0                3
0        5        0                3
0        5        0                3
Text for 0 is: 5555
```

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:

```
PS C:\Users\yp832\OneDrive\Desktop\sam6 manual\DC> & C:/Users/yp832/AppData/Local/Microsoft/WindowsApps/python3.10.exe '
p/sam6 manual/DC/P6.PY"
[(0, 0, 'ABCDE'), (0, 0, 'FABCR'), (3, 2, 'DDF'), (0, 0, 'HTFDD'), (0, 1, '')]
ABCDEFABCRDEDDFHTFDDA
PS C:\Users\yp832\OneDrive\Desktop\sam6 manual\DC>
```

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:

```
PS C:\Users\yp832\OneDrive\Desktop\sam6 manual\DC> & C:/U
C/p.7.py"

Compressed:
[77, 73, 83, 83, 257, 259, 80, 73]

Decompressed:
MISSISSIPI
PS C:\Users\yp832\OneDrive\Desktop\sam6 manual\DC>
```

Practical-8

AIM: Write a program to Implement LZ78 algorithm.

CODE:

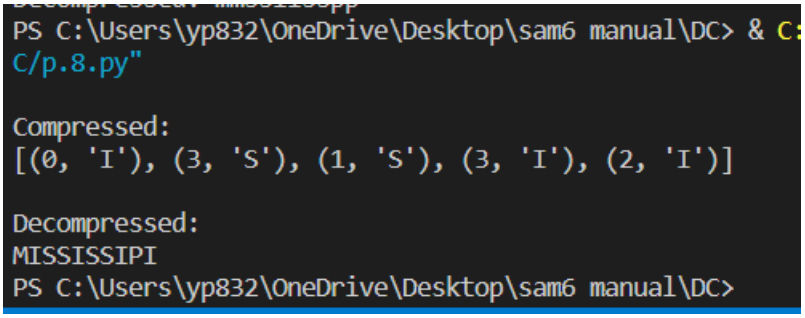
```
def lz78_compress(text):
    dictionary = {char: idx for idx, char in enumerate(set(text))}
    result = []
    i = 0
    while i < len(text):
        j = i + 1
        while j <= len(text) and text[i:j] in dictionary:
            j += 1
        if j > len(text):
            break
        dictionary[text[i:j]] = 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:

```
PS C:\Users\yp832\OneDrive\Desktop\sam6 manual\DC>   & C:/Users/yp832/AppData/Local/Micr
anual/DC/mtf.py"
Input text: panama
Move to Front Transform: [15, 1, 14, 1, 14, 1]
PS C:\Users\yp832\OneDrive\Desktop\sam6 manual\DC>
```

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:

```
yp832/OneDrive/Desktop/sam6 man
ual/DC/P.9jpg.py"
Encoded String:
 0010101111001000110001110100010111000101010111111111010111011001110101000110000000111011111111011010011011111000011001000101010001100111011101010010011001001110
PS C:\Users\yp832\OneDrive\Desktop\sam6 manual\DC> & C:/Users/yp832/AppData/Local/Microsoft/WindowsApps/python3.10.exe "c:/Users/
yp832/OneDrive/Desktop/sam6 manual/DC/P.9jpg.py"
PS C:\Users\yp832\OneDrive\Desktop\sam6 manual\DC> & C:/Users/yp832/AppData/Local/Microsoft/WindowsApps/python3.10.exe "c:/Users/
yp832/OneDriv
e/Desktop/sam6 manual/DC/P.9jpg.py"                                                                                    PS C:\U
sers\yp832\OneDrive\Desktop\sam6 manual\DC> & C:/Users/yp832/AppData/Local/Microsoft/WindowsApps/python3.10.exe "c:
/Users/yp832/OneDrive/Desktop/sam6 manual/DC/P.9jpg.py"                                                                Encoded
 String:                                                                                                          0010101111001
000110001110100010111000101010111111111010111011001110101000110000000111011111111011010011011111000011001000
101010001100111011010100100110010011100010011110100111011000110110101101111001111010000000010110111100101111000110100000110
 0010101111001000110001110100010111000101010111111111010111011001110101000110000000111011111111011010011011111000011001000101010001100111011101010010011001001110001001111010011101100011011010110111100111101000000001011011110010111100011010000011 00101011110 001010111 0
 0010101111001000110001110100010111000101010111111111010111011001110101000110000000111011111111011010011011111000011001000101010001100111011101010010011001001110001001111010011101100011011010110111100111101000000001011011110010111100011010000011010101101010111
1111111100100000010011110110010100000111100000100100100110101111
Huffman Dictionary:
 {65: '0000', 69: '0001', 154: '00100', 52: '001010', 60: '001011', 58: '00110', 62: '001110', 67: '001111', 61: '0100', 63: '010
10', 64: '01011', 66: '01100', 71: '01101', 70: '0111', 72: '100000', 75: '100001', 73: '10001', 76: '100100', 77: '100101', 78:
'100110', 83: '100111', 79: '10100', 85: '10101', 87: '101100', 88: '101101', 90: '101110', 94: '101111', 104: '11000', 106: '110
010', 109: '110011', 113: '110100', 122: '110101', 126: '110110', 144: '110111', 68: '1110', 55: '11110', 59: '11111'}
Decoded 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]]
```