# Lab 6

**Objectives:**

The objective of this lab is to implement and understand the Huffman Coding algorithm, a fundamental technique in data compression. Through the construction of a Huffman Tree and generation of prefix-free binary codes based on character frequencies, the lab aims to demonstrate how data can be efficiently encoded to reduce storage space. This exercise also helps reinforce concepts of greedy algorithms, priority queues (heaps), and binary tree traversal. By the end of the lab, students should be able to apply Huffman coding in practical scenarios and understand its significance in areas such as file compression and network data transmission.

**Theory: Huffman Coding**

Huffman Coding is a popular algorithm used for lossless data compression. It is based on the idea of assigning shorter binary codes to more frequent characters and longer codes to less frequent ones. This ensures that the total number of bits used to represent a message is minimized.

The algorithm works by building a binary tree, called the Huffman Tree, using the frequencies of each character in the input. Characters with the lowest frequencies are placed lower in the tree, resulting in longer codes, while more frequent characters are closer to the root, resulting in shorter codes. This is an example of a greedy algorithm.

Steps of Huffman Coding:

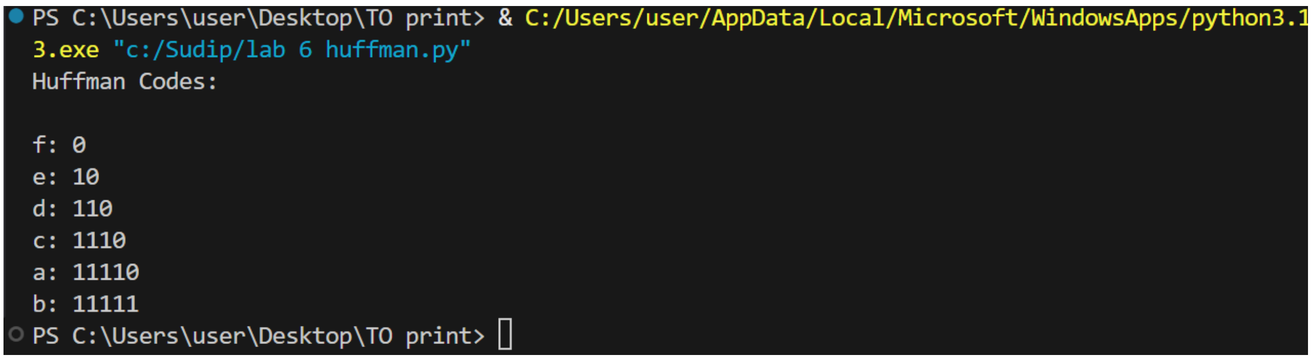
a) Count the frequency of each character.
b) Create a priority queue (min-heap) with all characters as leaf nodes.
c) Remove two nodes with the lowest frequencies and combine them.
d) Repeat step 3 until only one node (the root) remains — this is the Huffman Tree.
e) Traverse the tree to assign binary codes: go left (0) and right (1).

**Source 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, other):
        return self.freq < other.freq
def print_huffman_codes(node, code=''):
    new_code = code + str(node.huff)
    if node.left:
        print_huffman_codes(node.left, new_code)
    if node.right:
        print_huffman_codes(node.right, new_code)
    if not node.left and not node.right:
        print(f"{node.symbol}: {new_code}")

chars = ['a', 'b', 'c', 'd', 'e', 'f']
freqs = [5, 9, 10, 15, 30, 45]
# Create priority queue
nodes = [Node(freqs[i], chars[i]) for i in range(len(chars))]
heapq.heapify(nodes)
# Build Huffman Tree
while len(nodes) > 1:
    left = heapq.heappop(nodes)
    right = heapq.heappop(nodes)
    left.huff = 0
    right.huff = 1
    merged = Node(left.freq + right.freq, left.symbol + right.symbol, left, right)
    heapq.heappush(nodes, merged)
print("Huffman Codes:\n")
print_huffman_codes(nodes[0])
```

**Output:**



```
PS C:\Users\user\Desktop\TO print> & C:/Users/user/AppData/Local/Microsoft/WindowsApps/python3.1
3.exe "c:/Sudip/lab 6 huffman.py"
Huffman Codes:

f: 0
e: 10
d: 110
c: 1110
a: 11110
b: 11111
PS C:\Users\user\Desktop\TO print> 
```

**Conclusion:**

In conclusion, the implementation of Huffman Coding successfully demonstrated how character frequencies can be used to construct an optimal binary tree for efficient data compression. By using a priority queue to build the tree and assigning shorter codes to more frequent characters, we achieved a prefix-free binary encoding scheme. This lab not only reinforced the theoretical understanding of greedy algorithms and binary trees but also highlighted the real-world relevance of Huffman coding in file compression and digital communication. The practical experience gained through coding and testing the algorithm deepened our grasp of data structures and algorithmic efficiency.