93. Optimal Tree Problem: Huffman Trees and Codes

Code:

```
import heapq
from collections import defaultdict, Counter
class Node:
    def __init__(self, freq, symbol, left=None, right=None):
        self.freq = freq
        self.symbol = symbol
        self.left = left
        self.right = right

    def __lt__(self, other):
        return self.freq < other.freq

def build_huffman_tree(symbols):
    priority_queue = [Node(freq, sym) for sym, freq in symbols.items()]
    heapq.heapify(priority_queue)
    while len(priority_queue) > 1:
        left = heapq.heappop(priority_queue)
        right = heapq.heappop(priority_queue)
        merged = Node(left.freq + right.freq, None, left, right)
        heapq.heappush(priority_queue, merged)

    return heapq.heappop(priority_queue)

def generate_codes(node, prefix="", codebook={}):
    if node is not None:
        if node.symbol is not None:
            codebook[node.symbol] = prefix
        generate_codes(node.left, prefix + "0", codebook)
        generate_codes(node.right, prefix + "1", codebook)
    return codebook
```

```
def huffman_encoding(data):
    if not data:
        return "", {}
    frequency = Counter(data)


    huffman_tree = build_huffman_tree(frequency)
    huffman_codes = generate_codes(huffman_tree)
    encoded_data = ''.join(huffman_codes[symbol] for symbol in data)

    return encoded_data, huffman_codes

def huffman_decoding(encoded_data, huffman_codes):
    reverse_codebook = {v: k for k, v in huffman_codes.items()}
    decoded_data = []
    current_code = ""

    for bit in encoded_data:
        current_code += bit
        if current_code in reverse_codebook:
            decoded_data.append(reverse_codebook[current_code])
            current_code = ""

    return ''.join(decoded_data)
data = "this is an example for huffman encoding"
encoded_data, huffman_codes = huffman_encoding(data)
print("Encoded Data:", encoded_data)
print("Huffman Codes:", huffman_codes)
```

```
decoded_data = huffman_decoding(encoded_data, huffman_codes)
print("Decoded Data:", decoded_data)
```

Output:

```
Encoded Data: 0101001001001001010110010010101111100010111100111011110011100000110111101011101110010110010100110001101110100111111000101111000001111110010101110010001
Huffman Codes: {'n': '000', 's': '0010', 'm': '0011', 'h': '0100', 't': '01010', 'd': '01011', 'r': '01100', 'l': '01101', 'x': '01110', 'c': '01111', 'p': '10000', 'g': '10001', 'i': '1001', ' ': '101', 'u': '11000', 'o': '11001', 'f': '1101', 'e': '1110', 'a': '1111'}
Decoded Data: this is an example for huffman encoding
```

Time Complexity:

- T(n)= O((E+V)logv)