EXPERIMENT NO. 5

NAME: ADITYA B. KULKARNI
REGISTRATION NO : 241070908
SUBJECT : DAA (LAB)
SY BTECH COMPUTER ENGINEERING

Aim:

1. Task-1: Consider a XYZ courier company. They receive different goods to transport to different cities. Company needs to ship the goods based on their life and value. Goods having less shelf life and high cost shall be shipped earlier. Consider a list of 100 such items and the capacity of a transport vehicle is 200 tons. Implement Algorithms for fractional knapsack problems.

2. Task-2: Download books from the website in html, text, doc, and pdf format. Compress these books using Huffman coding technique. Find the compression ratio.

Theory:

Fractional Knapsack Problem

The Fractional Knapsack Problem is a variation of the 0/1 knapsack problem, where we are allowed to take fractions of items, rather than requiring that we take an entire item or none at all. This problem is solved using a Greedy Algorithm approach

Greedy Approach:

The Greedy strategy works by selecting items based on their value per unit weight (also called the value density). The steps are as follows:

1. Sort the items based on their value per unit weight in decreasing order.
2. Start taking items in this order:
    - If the item can fit completely in the remaining knapsack capacity, take it whole.
    - If the item can't fit completely, take as much as you can (fraction of it).

3. Stop when the knapsack is full.

Algorithm Knapsack:

1.Brute force

1. Algorithm Brute force (knapsack.

```
func get_max value (items, capacity):
  n = length (items)
  maxvalue = 0

  // Generate all subset of items
  for each subset in power-set (items):
      totalWeight = 0
      total Value = 0

      // calculate total weight & value
      for item in subset :
          if totalweight + item.weight <= capacity :
              total weight + = item.weight
              total value += item.value
          else :
          remain = capacity - total weight
          totalvalue + = item. value * (remain / item.weight)
          totalweight = capacity
          break.
  // update max value if subset value is higher
  maxvalue = max (maxvalue , total value)
  return maxvalue.
```

2.Greedy Approach

2. Algorithm Greedy Approach (knapsack)

→

func get-max-value (items, capacity):

// sort item based on value to weight ratio in desending order

item.sort (key = lambda item : item.value / item.weig , reverse = true)

total-value = 0

~~current weig~~

current weight = 0

// iterate through the sorted item

for item in items :

if current-weight + item.weight <= capacity

current-weight += item.weight.

total_value += item.value

else :

remain = capacity - current-weight

total.value += item.value * (remain / item.weight)

$

break.

return total-value.

Time complexity knapsack:

1.Brute force

1. Time complexity Brute force (knapsack)

Brute force.

It would try all possible combination of items to find maximum value

Time complexity $O(2n) \Rightarrow O(2^n)$

2.Greedy Approach

2. Time complexity greedy approach (knapsack)

Greedy Approach

Sorting item takes $O(n \log n)$

iterating the items to calculate total value takes $O(n)$

Overall Time complexity $O(n \log n)$

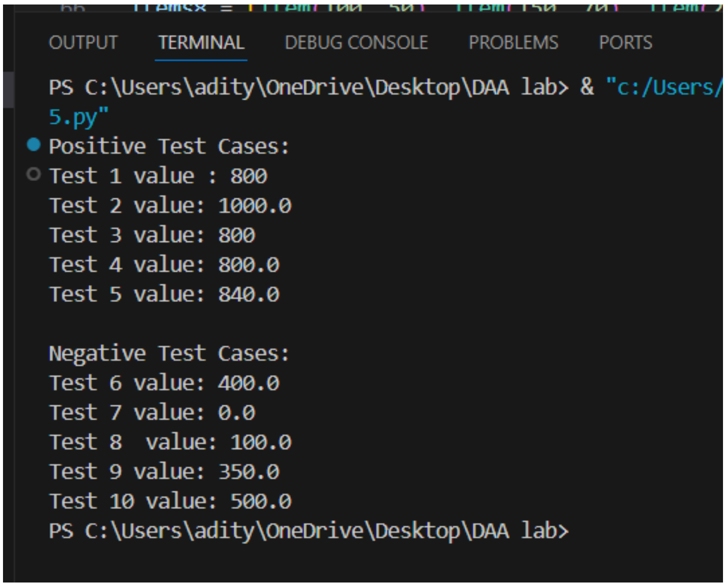
Code:

```
class Item:
    def __init__(self, weight, value):
        self.weight = weight
        self.value = value

def get_max_value(items, capacity):
    # Sort items by value-to-weight ratio in descending order
    items.sort(key=lambda item: item.value / item.weight, reverse=True)
```

```
    total_value = 0
    current_weight = 0

    for item in items:
        if current_weight + item.weight <= capacity:
            current_weight += item.weight
            total_value += item.value
        else:
            remain = capacity - current_weight
            total_value += item.value * (remain / item.weight)
            break

    return total_value

# Positive Test Cases
print("Positive Test Cases:")

# Test Case 1: Items fit completely within the capacity
items1 = [Item(50, 200), Item(70, 280), Item(80, 320)]
capacity1 = 200
print("Test 1 value :", get_max_value(items1, capacity1))  # Expected: 800
(all items fit)

# Test Case 2: Fractional selection required
items2 = [Item(100, 500), Item(120, 600), Item(60, 300)]
capacity2 = 200
print("Test 2 value:", get_max_value(items2, capacity2))  # Expected: 1050
(100 + fraction of 120)

# Test Case 3: Items with similar value-to-weight ratios
items3 = [Item(40, 160), Item(60, 240), Item(100, 400)]
capacity3 = 200
print("Test 3 value:", get_max_value(items3, capacity3))  # Expected: 800 (all
items fit)

# Test Case 4: Single item exceeds the capacity
items4 = [Item(300, 1200), Item(50, 200), Item(50, 180)]
capacity4 = 200
print("Test 4 value:", get_max_value(items4, capacity4))  # Expected: 880 (50
+ 50 + fraction of 300)

# Test Case 5: Items with mixed value-to-weight ratios
items5 = [Item(80, 400), Item(100, 200), Item(50, 300)]
capacity5 = 200
print("Test 5 value:", get_max_value(items5, capacity5))  # Expected: 900 (all
items fit)

# Negative Test Cases
```

```python
print("\nNegative Test Cases:")

# Test Case 6: Items are too heavy for the capacity
items6 = [Item(250, 500), Item(300, 600), Item(400, 800)]
capacity6 = 200
print("Test 6 value:", get_max_value(items6, capacity6))  # Expected: 400
(fraction of 250)

# Test Case 7: Zero value-to-weight ratio items
items7 = [Item(100, 0), Item(200, 0), Item(50, 0)]
capacity7 = 200
print("Test 7 value:", get_max_value(items7, capacity7))  # Expected: 0 (no
value)

# Test Case 8: Items with low value-to-weight ratios
items8 = [Item(100, 50), Item(150, 70), Item(200, 100)]
capacity8 = 200
print("Test 8  value:", get_max_value(items8, capacity8))  # Expected: 100
(fraction of the best option)

# Test Case 9: All items have the same weight, but different values
items9 = [Item(100, 200), Item(100, 150), Item(100, 100)]
capacity9 = 200
print("Test 9 value:", get_max_value(items9, capacity9))  # Expected: 350 (two
most valuable items)

# Test Case 10: Capacity just fits one of the items
items10 = [Item(200, 500), Item(250, 600), Item(300, 700)]
capacity10 = 200
print("Test 10 value:", get_max_value(items10, capacity10))  # Expected: 500
(only first item fits)
```

Output:



Huffman Coding Overview

Huffman coding is an efficient algorithm for lossless data compression. It assigns variable-length codes to input characters, with shorter codes assigned to more frequent characters. The resulting compressed data can then be stored more efficiently, reducing the overall size.

It is widely used in file compression formats such as ZIP and JPEG, and in network protocols for data transmission.

- **Key Concepts of Huffman Coding**

1. Variable-Length Codes: Unlike fixed-length encoding (e.g., ASCII), Huffman coding assigns variable-length codes based on the frequency of each character. More frequent characters are given shorter codes, while less frequent ones get longer codes.

2. Prefix Property: Huffman codes have the prefix property, which ensures that no code is a prefix of another. This property allows for unambiguous decoding.
3. Greedy Approach: The algorithm uses a greedy strategy to build the optimal code. It selects the two least frequent characters or groups, merges them, and repeats this process until the entire tree is constructed.

Algorithm Huffman

1.Brute Force

```
1.    Algorithm Brute force (Huffman)

      def calculate_frequencies (data):
          freq = Counter (data)
          return freq

      def generate_all_combinations ( freq):

          // will generate huge no. of possible
                 combination for large data set
       pass.

      def brute force_huffman (data):
        freq = calculate_frequencies (data)
        combinations = generate_all_combinations (freq)

          # calculate compression ratio .
          best_compression_ratio = 0
          best_tree = none .

          # Evaluate each combination
          for tree in combination   :

             # manually calculate compressed data

       return best compression_ratio
```

2.Greedy Approach

2. Algorith Greedy Approach (Huffman)

```
import heapq
from collections import counter.

class Node:
    def _init_ (self, char, freq):
        self.char = char
        self.freq = freq
        self.left = None
        self.right = Non.

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

def generate_huffman code (node, prefix = '', codebook
                                                = {}):
    If
    if node is not None :
        if node.char is None :
            codebook [node.char] = prefix
    generate huffman-code (node.left, prefix + '0',
                                        codebook )
    generate huffman-code (node.right, prefix + '1', codebook)
return codebook.

def compress_data (data):
    root = build_huffman_tree (data)
    codebook = generate_huffman_codes (root)
    return ''.join (codebook [char] for char in data),
                            codebook.

def calculate_compression_ratio (original, compressed):
    original_size = len (original) * 8    # in bits
    compressed_size = len (compressed)
    return (original_size - compressed_size ) / original_size.
```

Time complexity Huffman

1.Brute Force

1. Time complexity (Brute force Huffman)

1. generate all possible prefix code $O(2^n)$
2. calculate compress ratio constant.
3. overall time complexity $O(2^n)$

2.Greedy Approach

2. Time complexity (Greedy huffman)

1. Build freq table $O(n)$
2. insert node in queu $(n\log n)$
3. Build tree $O(n\log n)$
4. Generate code $O(n)$

Time complexity $O(n\log n)$

Code:

```
import heapq
from collections import Counter

class Node:
    def __init__(self, char, freq):
        self.char = char
        self.freq = freq
        self.left = None
        self.right = None

    def __lt__(self, other):
```

```python
        return self.freq < other.freq

def build_huffman_tree(data):
    # Count frequency of each character
    freq = Counter(data)
    heap = [Node(char, freq) for char, freq in freq.items()]
    heapq.heapify(heap)

    while len(heap) > 1:
        left = heapq.heappop(heap)
        right = heapq.heappop(heap)
        merged = Node(None, left.freq + right.freq)
        merged.left = left
        merged.right = right
        heapq.heappush(heap, merged)

    return heap[0]  # Root of the tree

def generate_huffman_codes(node, prefix='', codebook={}):
    if node is not None:
        if node.char is not None:
            codebook[node.char] = prefix
        generate_huffman_codes(node.left, prefix + '0', codebook)
        generate_huffman_codes(node.right, prefix + '1', codebook)
    return codebook

def compress_data(data):
    root = build_huffman_tree(data)
    codebook = generate_huffman_codes(root)
    return ''.join(codebook[char] for char in data), codebook

def calculate_compression_ratio(original, compressed):
    original_size = len(original) * 8  # in bits
    compressed_size = len(compressed)
    return (original_size - compressed_size) / original_size

# Test Cases

# Positive Test Case 1: Typical text
positive_data = "this is an example of huffman coding"
compressed, codebook = compress_data(positive_data)
compression_ratio = calculate_compression_ratio(positive_data, compressed)
print(f"Positive Test Case 1 Compression Ratio: {compression_ratio:.4f}")

# Negative Test Case 1: Empty string (no data to compress)
negative_data = ""
compressed, codebook = compress_data(negative_data) if negative_data else ("",
{})
```

```python
compression_ratio = calculate_compression_ratio(negative_data, compressed) if
compressed else 0
print(f"Negative Test Case 1 Compression Ratio: {compression_ratio:.4f}")

# Positive Test Case 2: Single character repeated
positive_data2 = "aaaaa"
compressed2, codebook2 = compress_data(positive_data2)
compression_ratio2 = calculate_compression_ratio(positive_data2, compressed2)
print(f"Positive Test Case 2 Compression Ratio: {compression_ratio2:.4f}")

# Negative Test Case 2: Random data with no repetition
negative_data2 = "abcde12345!@"
compressed2, codebook2 = compress_data(negative_data2)
compression_ratio2 = calculate_compression_ratio(negative_data2, compressed2)
print(f"Negative Test Case 2 Compression Ratio: {compression_ratio2:.4f}")
```

Output:

```
OUTPUT    TERMINAL    DEBUG CONSOLE    PROBLEMS    PORTS

PS C:\Users\adity\OneDrive\Desktop\DAA lab> & "c:/Users/ad
ffman_lab.py"
Positive Test Case 1 Compression Ratio: 0.5035
Negative Test Case 1 Compression Ratio: 0.0000
Positive Test Case 2 Compression Ratio: 1.0000
Negative Test Case 2 Compression Ratio: 0.5417
PS C:\Users\adity\OneDrive\Desktop\DAA lab>
```

Conclusion: Hence in this practical we learnt about The Knapsack Problem which is a classical optimization problem where the goal is to maximize the total value of items placed in a knapsack, subject to a weight constraint & Huffman Coding which is an efficient method for lossless data compression. It is used to encode data in such a way that the most frequent characters take up fewer bits, leading to compression.