DAA Assignment 5

Ishaan Shaikh
231070063
SY CE

Q. 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 an algorithm for fractional knapsack problem.

Algorithm:

Brute Force:

```
// Input: Items array and max capacity
// Output: Total value of items carried
BruteForceKnapsack(items, W):
    S = [ ]
    total_value = 0
    n = items.length

    for i = 1 to 2^n
        subset = [ ]
        remaining = W
        for j = 1 to n and remaining > 0
            if items[j] in subset : // (i & 1<<j)
                fraction = min(items[j].weight/remaining, 1)
                subset.append((items[j].value, fraction))
                remaining = remaining - fraction * items[j].weight
        valueSum = sum(item[1] * item[2] for item in subset)

        total_value = max(total_value, valueSum)

    return total_value
```

Greedy:

Algorithm :

Fractional Knapsack (items):
// Input: Array of class item containing items available for transport.
// Output: Total value of maximum shipment

```
for each item in items:
    item.ratio = item.value / (item.life * item.weight)

sort (items, by = item.ratio)
total_weight = 200
total_value = 0

for each item in items:
    if total_weight == 0
        break
    if item.weight <= total.weight
        total_value += item.value
        total_weight -= item.weight
    else:
        fraction = total_weight / item.weight
        total_value += item.value * fraction
        total_weight = 0

return total_value
```

Test cases:

Test Cases:

1) Input: [Item('A', 100, 500, 2), Item('B', 100, 700, 1)]
Output: Value : 1200

2) Input: [Item('C', 150, 750, 1), Item('D', 70, 280, 2), Item('E', 50, 400, 3)]
Output: Value : 950

3) Input: [Item('F', 180, 1200, 1), Item('G', 50, 300, 3)]
Output: Value : 1320

4) Input: [Item("J", 50, 0, 1), Item('K', 100, 0, 2)]
Output: All items have zero value

5) Input: [ ]
Output: No items available to load

6) Input: [Item('O', 0, 500, 1), Item('P', 100, 700, 2)]
Output: One or more items have zero weight

Time Complexity:

Brute Force:

Brute Force:

Input: Array of items and its properties

Basic operation: Check value of each subset

Input size: n

Let $C_{worst}(n)$ be the T.C of worst case of algorithm

$$C_{worst}(n) = \sum_{i=1}^{2^n} i$$

$$= 2^n - \cancel{1} + \cancel{1}$$

$$= 2^n$$

$\therefore$ T.C is $O(2^n)$

Greedy:

Time Complexity:

Input size : n (file size)

Basic operation : Build a Huffman Tree and assign Huffman codes to each character

1) Counting frequency of each character

Let Cworst(n) be the T.C

$C_{worst}(n) = \sum_{i=1}^{n} 1 = n-1+1 = n$

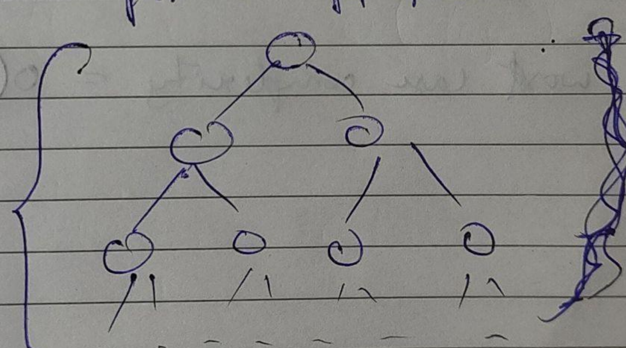
$\therefore T.C = O(n)$

2) Building Huffman Tree

Let m be the no. of unique characters in the file of size n (m <= n)

It involves push and pop operations in the heap

log m



There are m unique characters

$\therefore$ There will be m nodes in the Huffman Tree

~~$\therefore$ Tree will~~

$\therefore$ m elements will be pushed and popped from the tree

$\therefore$ T.C is $O(m \log m)$

3) Encoding each character
Each node is visited and there are m nodes
∴ $O(m)$ is the T.C.

4) Encoding the given file
There are n characters in the file. Each one is replaced
∴ $O(n)$ is the T.C

∴ Total T.C $= O(m) + O(n) + O(m\log m) + O(n)$

The worst case will occur when $m = n$
(no. of unique characters = no. of total characters)

∴ Total T.C $= O(n) + O(n) + O(n\log n) + O(n)$

$= O(n\log n) + 3\,O(n)$

~~= O(nloa~~

∴ Overall worst case complexity $= O(n\log n)$

<u>Program:</u> PEP 08 Coding style for python is used

```
class Item:
    """Class to represent an item with a name, weight, value, and shelf
life."""


    def __init__(self, name, weight, value, shelf_life):
        """Initialize an item with given properties.
```

```
        Args:
```

```
            name (str): Name of the item.
            weight (float): Weight of the item.
            value (float): Value of the item.
            shelf_life (int): Shelf life of the item in days.
        """
        self.name = name
        self.weight = weight
        self.value = value
        self.shelf_life = shelf_life
        # Avoid division by zero if weight is zero
        self.value_per_weight = value / weight if weight > 0 else
float('inf')

    def __repr__(self):
        return (f"Item(name={self.name}, weight={self.weight}, "
                f"value={self.value}, shelf_life={self.shelf_life})")


def fractional_knapsack(items, max_capacity=200):
    """Calculates the maximum value obtainable within given weight
capacity.

    Args:
        items (list[Item]): List of items to be considered.
        max_capacity (float): Maximum weight capacity of the knapsack.

    Returns:
        float or str: Maximum total value achievable with given items, or
                      an error message if inputs are invalid.
    """
    # Negative Test Case 1: No items available
    if not items:
        return "Error: No items available to load."
```

```python
    # Negative Test Case 2: All items have zero value
    if all(item.value == 0 for item in items):
        return "Error: All items have zero value."

    # Negative Test Case 3: Item(s) with zero weight
    if any(item.weight == 0 for item in items):
```

```python
        return "Error: One or more items have zero weight"

    # Sort items by shelf life (ascending) and value-to-weight ratio
(descending)
    items.sort(key=lambda item: (item.shelf_life, -item.value_per_weight))

    total_value = 0
    for item in items:
        if max_capacity <= 0:
            break
        if item.weight <= max_capacity:
            max_capacity -= item.weight
            total_value += item.value
        else:
            total_value += item.value_per_weight * max_capacity
            max_capacity = 0
    return total_value


def test_fractional_knapsack():
    """Runs positive and negative test cases for the fractional_knapsack
function."""

    # Positive Test Case 1: Items exactly fill the vehicle capacity with
maximum value
    items1 = [Item("A", 100, 500, 2), Item("B", 100, 700, 1)]
    print(f"Test 1 Value: {fractional_knapsack(items1)}")
```

```python
    items4 = [Item("J", 50, 0, 1), Item("K", 100, 0, 2)]
    print(f"Test 4: {fractional_knapsack(items4)}")

    # Negative Test Case 2: No items to load
    items5 = []
    print(f"Test 5: {fractional_knapsack(items5)}")

    # Negative Test Case 3: One item has zero weight (invalid case)
    items6 = [Item("O", 0, 500, 1), Item("P", 100, 700, 2)]
    print(f"Test 6: {fractional_knapsack(items6)}")


# Run the tests
if __name__ == "__main__":
    test_fractional_knapsack()
    # Positive Test Case 2: Items taken in fractional parts to maximize value
    items2 = [Item("C", 150, 750, 1), Item("D", 70, 280, 2), Item("E", 50, 400, 3)]
    print(f"Test 2 Value: {fractional_knapsack(items2)}")

    # Positive Test Case 3: High-value item with low shelf life is partially included
    items3 = [Item("F", 180, 1200, 1), Item("G", 50, 300, 3)]
    print(f"Test 3 Value: {fractional_knapsack(items3)}")

    # Negative Test Case 1: All items have zero value
```

Output:

```
Test 1 Value: 1200
Test 2 Value: 950.0
Test 3 Value: 1320.0
Test 4: Error: All items have zero value.
Test 5: Error: No items available to load.
Test 6: Error: One or more items have zero weight
```

Conclusion: Hence, we have studied the algorithm of Fractional Knapsack. We have implemented the program using greedy technique. Greedy technique here ensures that we have the maximum possible value of the items we will be carrying in the knapsack.