Analysis of Algorithm

Practical no 5 :

Fractional Knapsack

Code :

```
import java.util.*;
class Item {
   double weight;
   double value;
   public Item(double weight, double value) {
      this.weight = weight;
      this.value = value;
   }
   public double getValuePerWeight() {
      return value / weight;
   }
}
public class FractionalKnapsack {
   public static double knapsack(Item[] items, double capacity) {
      Arrays.sort(items, (a, b) -> Double.compare(b.getValuePerWeight(), a.getValuePerWeight()));
      double totalValue = 0.0;
      double remainingCapacity = capacity;
      for (Item item : items) {
         if (remainingCapacity == 0)
            break;
         if (item.weight <= remainingCapacity) {
            totalValue += item.value;
            remainingCapacity -= item.weight;
         }
         else {
            totalValue += item.getValuePerWeight() * remainingCapacity;
```

```
            remainingCapacity = 0;
        }
      }
      return totalValue;
   }
   public static void main(String[] args) {
      Scanner sc = new Scanner(System.in);
      System.out.println("Enter the number of items : ");
      int n = sc.nextInt();
      System.out.println("Enter the capacity of the knapsack : ");
      int capacity = sc.nextInt();
      int[] p = new int[n];
      int[] w = new int[n];
      System.out.println("Enter the Profits : ");
      for (int i = 0; i<n; i++)
         p[i] = sc.nextInt();
      System.out.println("Enter the Weights :");
      for (int i = 0; i<n; i++)
         w[i] = sc.nextInt();
      Item[] items = new Item[n];
      for (int i = 0; i<n; i++)
         items[i] = new Item(w[i],p[i]);
      double maxValue = knapsack(items, capacity);
      System.out.println("Maximum value in the knapsack = " + maxValue);
   }
}
```

Output :

```
Enter the number of items :
5
Enter the capacity of the knapsack :
40
Enter the Profits :
10
20
30
40
50
Enter the Weights :
12
53
73
129
54
Maximum value in the knapsack = 37.03703703703704
```

Analysis :

Classes and Methods

1. Item Class

The Item class represents an item with two properties:

- **weight**: the weight of the item.
- **value**: the value of the item.

The constructor initializes these two fields:

The method getValuePerWeight() calculates the value-to-weight ratio of the item:

This ratio helps determine how "valuable" an item is in comparison to its weight, which is essential for the greedy algorithm.

2. FractionalKnapsack Class

This class contains the main logic of the Fractional Knapsack problem.

knapsack Method

- **Sorting**: The first step is to **sort the items** in descending order of their value-to-weight ratio (value/weight). This is the core of the greedy approach: choose the most valuable items first. The sorting uses a lambda function to compare the ratios of two items (a and b).
- **Variables**:
    - totalValue: Keeps track of the total value accumulated in the knapsack.
    - remainingCapacity: The remaining capacity of the knapsack.
- **Loop through sorted items**:
    - If the **remaining capacity is zero**, the loop breaks (no more items can be added).

- If the **item's weight is less than or equal to the remaining capacity**, the whole item can be added to the knapsack. The value of the item is added to totalValue, and the remaining capacity is reduced by the item's weight.
- If the **item's weight is greater than the remaining capacity**, only a **fraction** of the item is added to the knapsack. The value of the fraction is calculated by multiplying the item's value-to-weight ratio by the remaining capacity.

The function then returns the total value that can be carried by the knapsack.

Main Method

- The program first asks the user to input the number of items (n) and the **capacity** of the knapsack.
- Then, it initializes two arrays p[] (profits) and w[] (weights) to store the values and weights of the items. The user is prompted to input these arrays.
- An array of Item objects is created, and each item is initialized with its weight and value.
- The knapsack method is called with the list of items and the knapsack's capacity. The result, which is the maximum value that can be carried, is printed.

Time Complexity

1. Sorting the Items

The program starts by sorting the items based on their value-to-weight ratio. Sorting takes **O(n log n)** time in all cases because the Arrays.sort method used in Java is typically implemented using a **dual-pivot quicksort** (with a worst-case time complexity of **O(n log n)**) or a **Timsort** (also **O(n log n)**).

- Sorting complexity is **O(n log n)** in the **best**, **average**, and **worst** cases.

2. Iterating Through the Items

After sorting, the program loops through the sorted array of items and either adds the full item or a fraction of it to the knapsack. This loop runs **n** times, where each operation inside the loop (like comparisons and assignments) takes constant time, **O(1)**.

- Looping through the items has a complexity of **O(n)**.

Thus, the overall **time complexity** of the program is:

- **Best Case**: **O(n log n)**
- **Average Case**: **O(n log n)**
- **Worst Case**: **O(n log n)**

Space Complexity

1. Input Arrays

The program creates two input arrays (p[] for profits and w[] for weights), each of size **n**.

- The space complexity of these arrays is **O(n)**.

2. Item Array

The program creates an array of Item objects, where each Item object contains two double values (weight and value). So, the space required for this array is **O(n)**.

3. Sorting Space

The Arrays.sort() method in Java, depending on the sorting algorithm used (typically **quicksort** or **Timsort**), requires some auxiliary space. However, Timsort usually requires **O(log n)** auxiliary space, and quicksort can require up to **O(log n)** in the worst case for recursion. Since the sorting is done in-place, the space complexity for sorting is **O(log n)**.

4. Other Variables

The program uses a few constant variables like totalValue, remainingCapacity, and iterators, all of which take constant space **O(1)**.

Thus, the overall **space complexity** of the program is:

- **Best Case**: **O(n)**
- **Average Case**: **O(n)**
- **Worst Case**: **O(n)**