|  |
| --- |
| **Experiment No. 6** |
| **Fraction Knapsack** |
| Date of Performance:14/3/24 |
| Date of Submission:21/3/24 |

## Experiment No. 6

**Title:** Fractional Knapsack

**Aim:** To study and implement Fractional Knapsack Algorithm

**Objective:** To introduce Greedy based algorithms

#### Theory:

Greedy method or technique is used to solve Optimization problems. A solution that can be maximized or minimized is called Optimal Solution.

The knapsack problem states that − given a set of items, holding weights and profit values, one must determine the subset of the items to be added in a knapsack such that, the total weight of the items must not exceed the limit of the knapsack and its total profit value is maximum.

The knapsack problem or rucksack problem is a problem in combinatorial optimization: Given a set of items, each with a mass and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible. It derives its name from the problem faced by someone who is constrained by a fixed size knapsack and must fill it with the most valuable items. The most common problem being solved is the 0-1 knapsack problem, which restricts the number xi of copies of each kind of item to zero or one.

In Knapsack problem we are given:

1. n objects
2. Knapsack with capacity m.
3. An object i is associated with profit Wi.
4. Object i is associated with profit Pi.
5. Object i is placed in knapsack we get profit Pi Xi .

Here objects can be broken into pieces (Xi Values) The Objective of Knapsack problem is to maximize the profit.

**Example:**

Find an optimal solution for fractional Knapsack problem.

Where,

Number of objects = 7

Capacity of Knapsack = 15

P1,P2,P3,P4,P5,P6,P7 = (10,5,15,7,6,18,3)

W1,W2,W3,W4,W5,W6,W7 = (2,3,5,7,1,4,1)

**Solution:**

Arrange the objects in decreasing order of Pi/Wi ratio.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Object** | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| **Pi** | 10 | 5 | 15 | 7 | 6 | 18 | 3 |
| **Wi** | 2 | 3 | 5 | 7 | 1 | 4 | 1 |
| **Pi/Wi** | 5 | 1.67 | 3 | 1 | 6 | 4.5 | 3 |

Select the objects with maximum Pi/Wi ratio:

|  |  |  |  |
| --- | --- | --- | --- |
| **Object** | **Profit (Pi)** | **Weight (Wi)** | **Remaining Weight** |
| - | - | - | 15 |
| 5 | 6 | 1 | 14 |
| 1 | 10 | 2 | 12 |
| 6 | 18 | 4 | 8 |
| 3 | 15 | 5 | 3 |
| 7 | 3 | 1 | 2 |
| 2 | 3.33 | 2 | 0 |
| **Total** | **55.33** | | |

**So, the maximum profit is 55.33 units.**

**Algorithm:**

Fractional Knapsack Problem:

Here,

N- Total No. of Objects

M- Capacity of Knapsack

P- Initial profit. P=0

Pi- Profit of ith object

Wi- Weight of ith Object

**Step 1:**

For i=1 to N

**O(n)**

**O(n)**

**O(n.logn)**

Calculate Profit / Weight Ratio (i.e. Pi/Wi)

**Step 2:**

Sort objects in decreasing order of Profit / Weight Ratio

**Step 3: // Add all the profit by considering the weight capacity of fractional knapsack.**

For i=1 to N

if M > 0 AND Wi <= M

M = M –Wi

P = P + Pi

else

break

if M > 0 Then

P = P + Pi \* (M/Wi)

**Step 4:**

Display Total Profit

**Time Complexity** = O(n) + O(n.logn) + O(n)

= Max(O(n),O(n.logn),O(n))

= O(n.logn)

**Code:**

#include <stdio.h>

// Structure to represent an item

struct Item {

int weight;

int profit;

float ratio; // Profit-to-weight ratio

};

// Function to perform bubble sort on the items array based on the profit-to-weight ratio

void bubbleSort(struct Item items[], int n) {

for (int i = 0; i < n - 1; i++) {

for (int j = 0; j < n - i - 1; j++) {

if (items[j].ratio < items[j + 1].ratio) {

// Swap items[j] and items[j+1]

struct Item temp = items[j];

items[j] = items[j + 1];

items[j + 1] = temp;

}

}

}

}

// Knapsack function with items sorted in descending order of profit-to-weight ratio

int knapsack(int wt[], int pr[], int w, int n) {

struct Item items[n];

// Initialize items array with weight, profit, and profit-to-weight ratio

for (int i = 0; i < n; i++) {

items[i].weight = wt[i];

items[i].profit = pr[i];

items[i].ratio = (float)pr[i] / wt[i];

}

// Sort items based on profit-to-weight ratio

bubbleSort(items, n);

int total\_weight = 0;

int max\_profit = 0;

// Fill knapsack with items in descending order of profit-to-weight ratio

for (int i = 0; i < n; i++) {

if (total\_weight + items[i].weight <= w) {

total\_weight += items[i].weight;

max\_profit += items[i].profit;

} else {

int remaining\_weight = w - total\_weight;

max\_profit += items[i].ratio \* remaining\_weight;

break; // Knapsack capacity full

}

}

return max\_profit;

}

int main() {

int n;

printf("Enter the number of items: ");

scanf("%d", &n);

int wt[n];

int pr[n];

printf("Enter the weight array: ");

for (int i = 0; i < n; i++) {

scanf("%d", &wt[i]);

}

printf("Enter the profit array: ");

for (int i = 0; i < n; i++) {

scanf("%d", &pr[i]);

}

int w;

printf("Enter the capacity of knapsack: ");

scanf("%d", &w);

int optimal\_profit = knapsack(wt, pr, w, n);

printf("Optimal profit: %d\n", optimal\_profit);

printf("Total weight is :%d\n",w);

return 0;

}

**Output:**

Enter the number of items: 5

Enter the weight array: 3 5 4 3 6

Enter the profit array: 18 25 27 10 15

Enter the capacity of knapsack: 12

Optimal profit: 70 . Total weight is :12

**Conclusion:**

**In conclusion, the fractional knapsack problem aims to maximize the total value of items selected while considering their weights and a given capacity. Unlike the 0-1 knapsack problem, fractional knapsack allows fractions of items to be taken. The problem can be efficiently solved using a greedy strategy, where items with the highest value-to-weight ratio are chosen first. This approach ensures an optimal solution and has a time complexity of O(n log n), where n is the number of items. Overall, the fractional knapsack algorithm provides an effective solution for optimizing resource allocation in scenarios where items can be divided.**