|  |
| --- |
| Experiment No. 5 |
| Fractional Knapsack using Greedy Method |
| Date of Performance: |
| Date of Submission: |

**Experiment No. 5**

**Title:** Fraction Knapsack

**Aim:** To study and implement Fraction 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 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) An object i is associated with profit Pi , 5) when an object i is placed in knapsack we get profit Pi Xi .

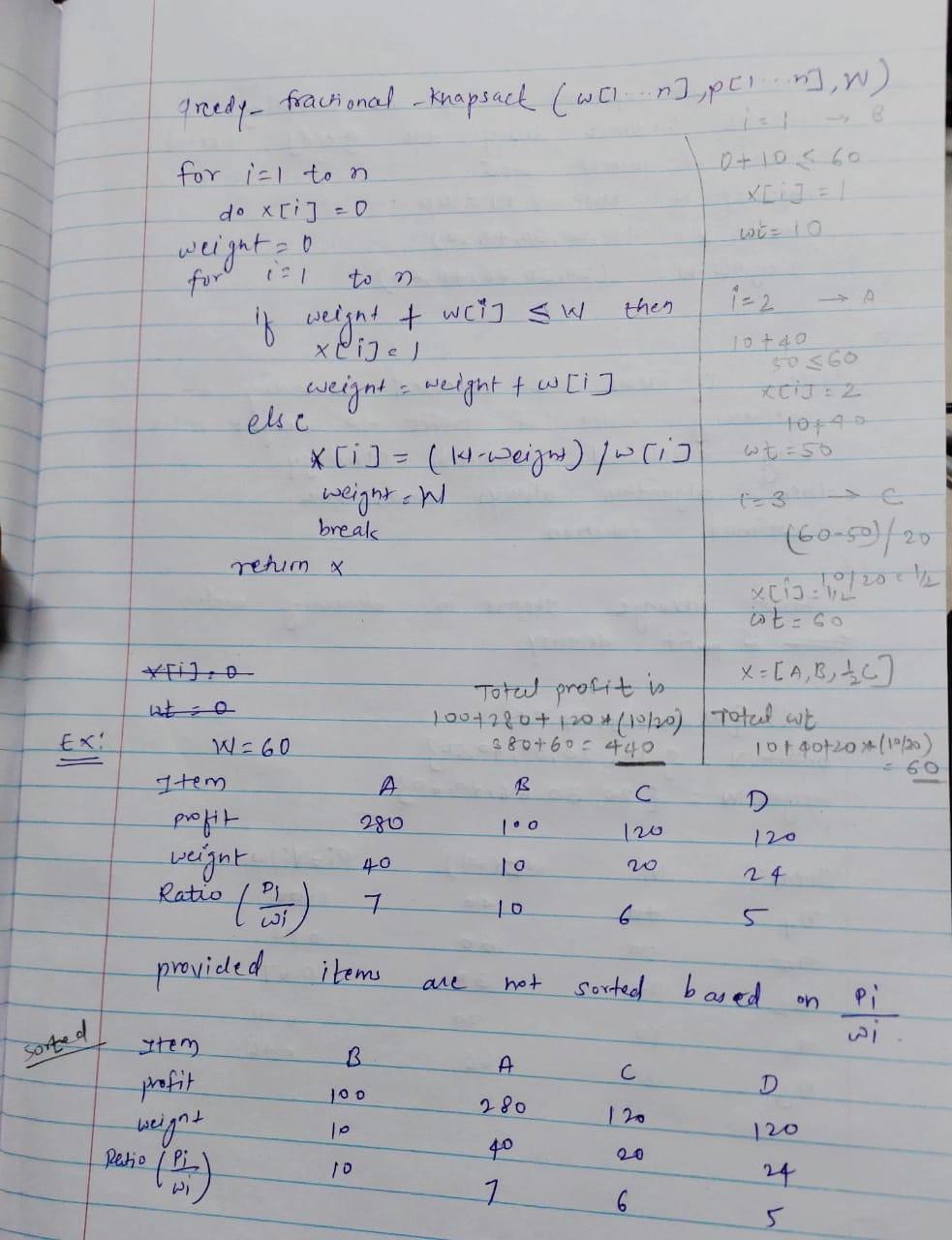
Here objects can be broken into pieces (Xi Values) The Objective of Knapsack problem is to maximize the profit.

**Example:**

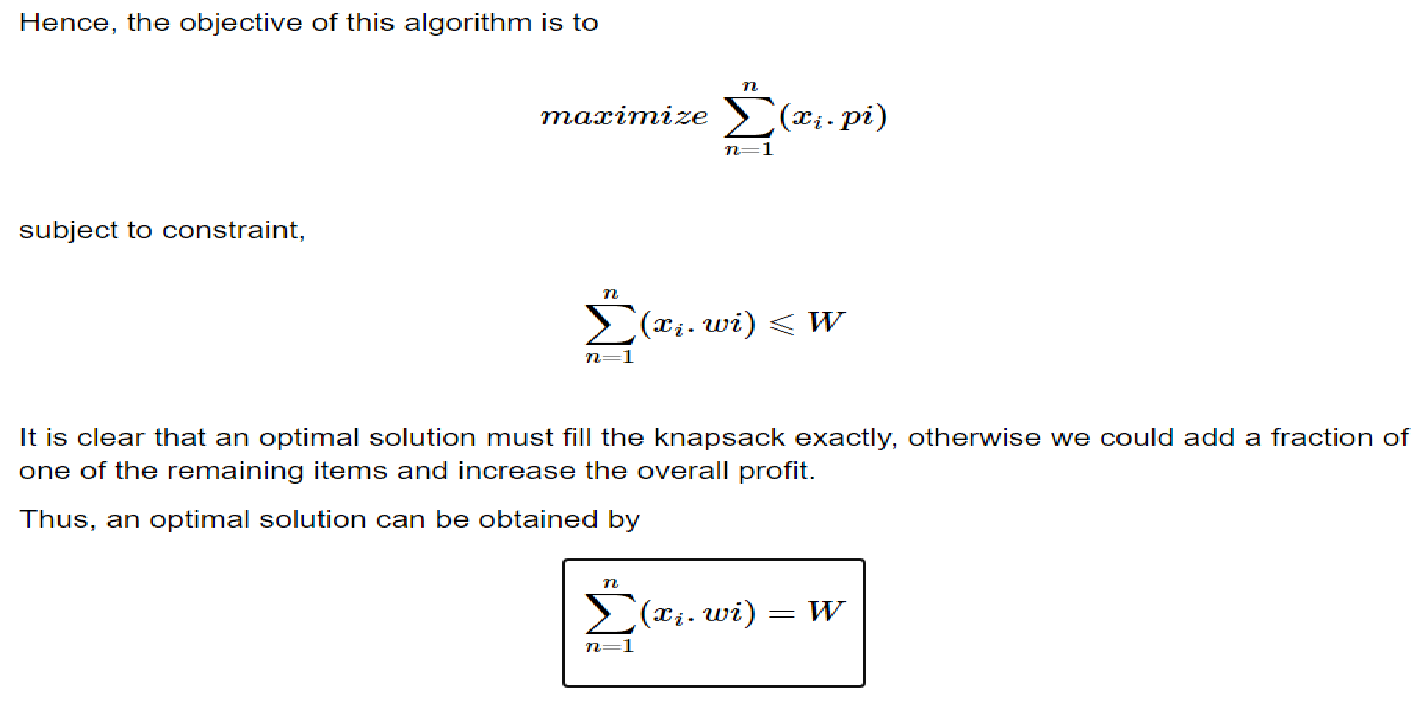
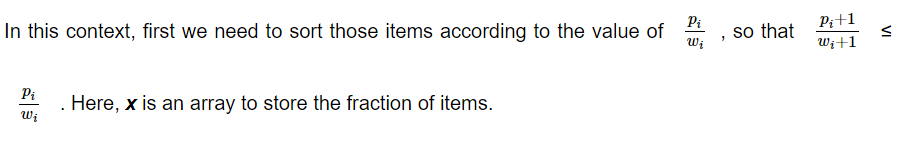
In this version of Knapsack problem, items can be broken into smaller pieces. So, the thief may take only a fraction *xi* of ith item.

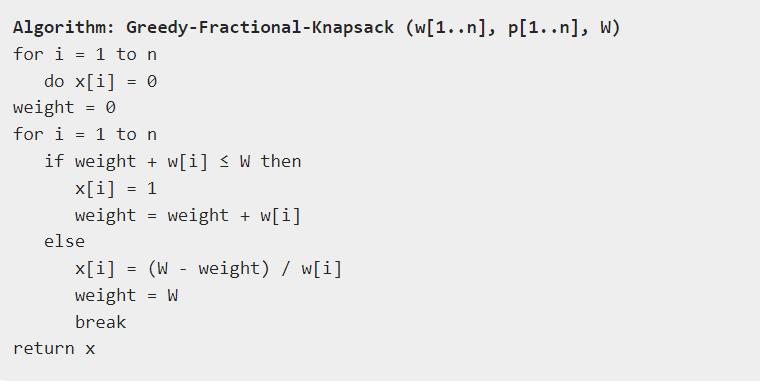
0⩽xi⩽1

The ith item contributes the weight xi.wi to the total weight in the knapsack and profit xi.pi to the total profit**.**

****

**Algorithm:**

****



**Implementation:**

**Code:**

#include <stdio.h>

#define MAX\_ITEMS 100

void fractionalKnapsack(int n, float values[], float weights[], float capacity) {

    float x[MAX\_ITEMS];

    float totalValue = 0;

    float weight = 0;

    int i;

    for (i = 0; i < n; ++i) {

    x[i] = 0;

    }

    for (i = 0; i < n; ++i) {

    if (weight + weights[i] <= capacity) {

        x[i] = 1;

        totalValue += values[i];

        weight += weights[i];

    } else {

        x[i] = (capacity - weight) / weights[i];

        totalValue += x[i] \* values[i];

        weight = capacity;

        break;

    }

    }

    printf("Fractional Knapsack Solution:\n");

    printf("Selected fractions for each item:\n");

    for (i = 0; i < n; ++i) {

    printf("Item %d: %.2f\n", i + 1, x[i]);

    }

    printf("Total value of selected items: %.2f\n", totalValue);

}

int main() {

    int n;

    float values[MAX\_ITEMS], weights[MAX\_ITEMS], capacity;

    int i;

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

    scanf("%d", &n);

    printf("Enter the values and weights for each item:\n");

    for (i = 0; i < n; ++i) {

    printf("Item %d: ", i + 1);

    scanf("%f %f", &values[i], &weights[i]);

    }

    printf("Enter the knapsack capacity: ");

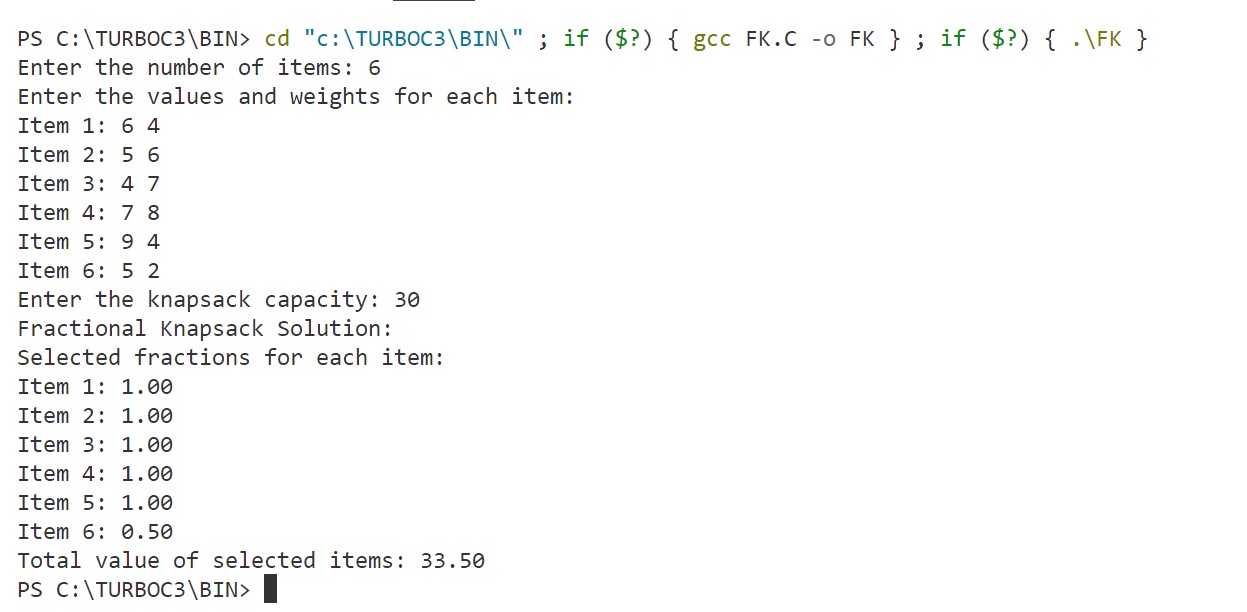
    scanf("%f", &capacity);

    fractionalKnapsack(n, values, weights, capacity);

    return 0;

}

**Output:**

****

**Conclusion:** In conclusion, the fractional knapsack algorithm efficiently solves the fractional knapsack problem, which involves selecting items of maximum value to fit within a knapsack of limited capacity. Unlike the 0/1 knapsack problem, where items must be either entirely included or excluded from the knapsack, the fractional knapsack problem allows items to be included partially, enabling fractions of items to be selected based on their value-to-weight ratio.

This algorithm iteratively selects items based on their value-to-weight ratio, prioritizing items with higher ratios to maximize the total value of items included in the knapsack. It calculates the fraction of each item to include in the knapsack, ensuring that the total weight does not exceed the knapsack's capacity.