Name: Vishal Shashikant Salvi.

Class: SE Comps

Batch: C

UID: **2019230069**

Aim:

Knapsack problem using Greedy approach.

Theory:

What is Greedy Strategy?

Greedy algorithms are like dynamic programming algorithms that are often used to solve optimal problems (find best solutions of the problem according to a particular criterion).

Greedy algorithms implement optimal local selections in the hope that those selections will lead to an optimal global solution for the problem to be solved. Greedy algorithms are often not too hard to set up, fast (time complexity is often a linear function or very much a second-order function). Besides, these programs are not hard to debug and use less memory. But the results are not always an optimal solution.

Greedy strategies are often used to solve the combinatorial optimization problem by building an option A. Option A is constructed by selecting each component Ai of A until complete (enough n components). For each Ai, you choose Ai optimally. In this way, it is possible that at the last step you have nothing to select but to accept the last remaining value.

There are two critical components of greedy decisions:

1. Way of greedy selection. You can select which solution is best at present and then solve the subproblem arising from making the last selection. The selection of greedy algorithms may depend on previous selections. But it cannot depend on any future selection or depending on the solutions of subproblems. The algorithm evolves in a way that makes selections in a loop, at the same time shrinking the given problem to smaller subproblems.
2. Optimal substructure. You perform the optimal substructure for a problem if the optimal solution of this problem contains optimal solutions to its subproblems.

A greedy algorithm has five components:

1. A set of candidates, from which to create solutions.
2. A selection function, to select the best candidate to add to the solution.
3. A feasible function is used to decide if a candidate can be used to build a solution.
4. An objective function, fixing the value of a solution or an incomplete solution.
5. An evaluation function, indicating when you find a complete solution.

Knapsack Problem-

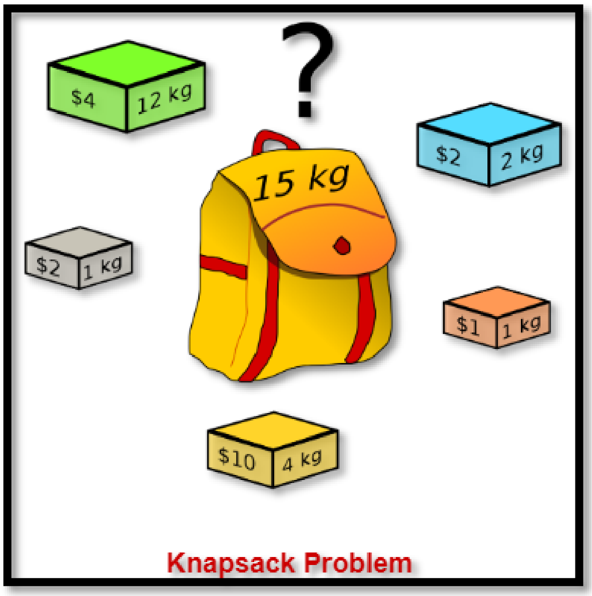
You are given the following-

- A knapsack (kind of shoulder bag) with limited weight capacity.
- Few items each having some weight and value.

The problem states-

Which items should be placed into the knapsack such that-

- The value or profit obtained by putting the items into the knapsack is maximum.
- And the weight limit of the knapsack does not exceed.



Knapsack Problem Variants-

Knapsack problem has the following two variants-

1. Fractional Knapsack Problem
2. 0/1 Knapsack Problem

Fractional Knapsack Problem-

In Fractional Knapsack Problem,

- As the name suggests, items are divisible here.
- We can even put the fraction of any item into the knapsack if taking the complete item is not possible.
- It is solved using Greedy Method.

Fractional Knapsack Problem Using Greedy Method-

Fractional knapsack problem is solved using greedy method in the following steps-

Step-01:

For each item, compute its value / weight ratio.

Step-02:

Arrange all the items in decreasing order of their value / weight ratio.

Step-03:

Start putting the items into the knapsack beginning from the item with the highest ratio.

Put as many items as you can into the knapsack.

- Algorithm:

```
function fksnap ( p, w, n, m )
        n → no. of items
        p → array of profit
        w → array of weight
        M → Max Capacity of Sack.
   real Cp, cw, x(1:n)

   1> Start
   2> Sort all items in descending order of p/w
   3> cp = cw = 0
   4> for i = 1 to n do
       {
           if ( cw + w(i) < M then
            {
               x(i) = 1
               cp = cp + P(i)
               cw = cw + w(i)
            }
           else
            {
               x(i) = (M - cw) / w(i)

              cp = cp + P(i) * x(i)
              cw = M
              go to step 5
           }
       }
   5> return
```

problem:

n = 7

M = 15

n	p	w	p/w
1	10	2	5
2	5	3	1.66
3	15	5	3
4	7	7	1
5	6	1	6
6	18	4	4.5
7	3	1	3

5, 1, 6, 7, 3, 2, 1

Step 1:

consides Item 5

cp = cw = 0

∴ w(5) = 1

∴ 0 + 1 < 15

∴ Item 5 is accepted

∴ x(5) = 1

∴ cp = 0 + 6 = 6

∴ cw = 0 + 1 = 1

Step 2 :

consider Item 1

$\therefore \omega(1) = 2$

$\therefore 1 + 2 < 15$

$\therefore$ Item 1 is accepted

$\therefore x(1) = 1$

$\therefore cp = 6 + 10 = 16$

$\therefore cw = 1 + 2 = 3$

Step 3 :

consider Item 6

$\therefore \omega(6) = 4$

$\therefore 3 + 4 < 15$

$\therefore$ Item 6 is selected

$\therefore x(6) = 1$

$\therefore cp = 16 + 18 = 34$

$\therefore cw = 3 + 4 = 7$

Step 4 :

consider Item 7

$\therefore \omega(7) = 1$

$\therefore 7 + 1 < 15$

$\therefore$ Item 7 is selected

$\therefore x(7) = 1$

$\therefore cp = 34 + 3 = 37$

$\therefore cw = 7 + 1 = 8$

Step 5 :

Consider Item 3

$\therefore$ $w(3) = 5$

$\therefore$ $8 + 5 < 15$

$\therefore$ Item 3 is accepted

$\therefore$ $x(3) = 1$

$\therefore$ $cp = 37 + 15 = 52$

$\therefore$ $cw = 8 + 5 = 13$

Step 6

Consider Item 2

$\therefore$ $w(2) = 3$

$\therefore$ $13 + 3 > 15$

$\therefore$ Item 2 is partially selected

$\therefore$ $x(2) = \frac{15 - 13}{3} = \frac{2}{3}$

$\therefore$ $cp = 52 + 5 * \frac{2}{3}$

$\therefore$ $cp = 52 + \frac{10}{3}$

$\therefore$ $cp = \frac{156 + 10}{3}$

$\therefore$ $cp = \frac{166}{3}$

$\boxed{\therefore\ cp = 55.33}$

$\boxed{\therefore\ cw = 15}$

Time Complexity-

- The main time taking step is the sorting of all items in decreasing order of their value / weight ratio.
- If the items are already arranged in the required order, then while loop takes O(n) time.
- The average time complexity of **Merge Sort** is O(nlogn).
- Therefore, total time taken including the sort is O(nlogn).

Analysis :

$$T(n) = \underbrace{T(n/2)}_{\text{Left Sublist}} + \underbrace{T(n/2)}_{\text{Right Sublist}} + \underbrace{cn}_{\text{Both combine}}$$

where $n > 1 \; T(1) = 0$

i) using master theorem:

Let, the recurrence relation for merge sort is

$$T(n) = T(n/2) + T(n/2) + cn$$

i.e $T(n) = 2T(n/2) + cn$

$T(1) = 0$

As per master theorem

$T(n) = \Theta(n^d \log n)$ if $a = b$

$a = 2,\ b = 2$ and $f(n) = cn$

i.e n^d with $d = 1$

and

$a = b^d$

i.e $2 = 2^1$

This case gives us

$T(n) = \Theta(n \log_2 n)$

$\therefore \Theta(n \log n)$

Code:

```
#include<stdio.h>
struct items
{
        float profit[10],weight[10];
}item;
void knapsack(int n, float weight[], float profit[], float capacity) {
   float x[20], cp = 0,cw=0;
   int i, j, u;
   u = capacity;

   for (i = 0; i < n; i++)
    {
      if (cw+weight[i]<u)
            {
         x[i] = 1.0;
         cp = cp+ profit[i];
         cw=cw+weight[i];
      }
      else
      {
            x[i]=(u-cw)/weight[i];
            cp=cp+profit[i]*x[i];
            cw=u;
              }

    }

   printf("\nThe final solution is:- \n ");
```

```
   for (i = 0; i < n; i++)
      printf("%f\t", x[i]);

   printf("\nMaximum profit is:- %f", cp);

}

int main() {
   float capacity;
   int num, i, j;
   float ratio[20], temp;

   printf("\nEnter the no. of items:- ");
   scanf("%d", &num);
         for (i = 0; i < num; i++)
         {
                 printf("\nEnter the profits and wt of item %d:",i+1);
                 scanf("%f %f", &item.profit[i], &item.weight[i]);
    }

   printf("\nEnter the capacityacity of knapsack:- ");
   scanf("%f", &capacity);

   printf("\n");

   for (i = 0; i < num; i++) {
      ratio[i] = item.profit[i] / item.weight[i];
      printf("%f\t",ratio[i]);
   }
```

```
  printf("\n");

  for (i = 0; i < num; i++) {
    for (j = i + 1; j < num; j++) {
      if (ratio[i] < ratio[j]) {
        temp = ratio[j];
        ratio[j] = ratio[i];
        ratio[i] = temp;

        temp = item.weight[j];
                        item.weight[j] =item.weight[i];
                        item. weight[i] = temp;

        temp = item.profit[j];
        item.profit[j] = item.profit[i];
        item.profit[i] = temp;
      }
    }
  }

  knapsack(num, item.weight, item.profit, capacity);
  return(0);
}Code:
#include<stdio.h>
struct items
{
        float profit[10],weight[10];
}item;
void knapsack(int n, float weight[], float profit[], float capacity) {
  float x[20], cp = 0,cw=0;
```

```
   int i, j, u;
   u = capacity;

   for (i = 0; i < n; i++)
    {
      if (cw+weight[i]<u)
            {
         x[i] = 1.0;
         cp = cp+ profit[i];
         cw=cw+weight[i];
      }
      else
      {
            x[i]=(u-cw)/weight[i];
            cp=cp+profit[i]*x[i];
            cw=u;
              }

    }



   printf("\nThe final solution is:- ");
   for (i = 0; i < n; i++)
      printf("%f\t", x[i]);

   printf("\nMaximum profit is:- %f", cp);

}

int main() {
```

```
float capacity;
int num, i, j;
float ratio[20], temp;

printf("\nEnter the no. of items:- ");
scanf("%d", &num);
        for (i = 0; i < num; i++)
        {
                printf("\nEnter the profits and wt of item %d:",i+1);
                scanf("%f %f", &item.profit[i], &item.weight[i]);
 }

printf("\nEnter the capacityacity of knapsack:- ");
scanf("%f", &capacity);

for (i = 0; i < num; i++) {
   ratio[i] = item.profit[i] / item.weight[i];
   printf("%f",ratio[i]);
}

for (i = 0; i < num; i++) {
   for (j = i + 1; j < num; j++) {
      if (ratio[i] < ratio[j]) {
         temp = ratio[j];
         ratio[j] = ratio[i];
         ratio[i] = temp;

         temp = item.weight[j];
                        item.weight[j] =item.weight[i];
                        item. weight[i] = temp;
```

```
            temp = item.profit[j];

            item.profit[j] = item.profit[i];

            item.profit[i] = temp;

          }

       }

    }


   knapsack(num, item.weight, item.profit, capacity);

   return(0);

}
```

Output:

```
Enter the no. of items:- 7

Enter the profits and wt of item 1:
10 2

Enter the profits and wt of item 2:
5 3

Enter the profits and wt of item 3:
15 5

Enter the profits and wt of item 4:
7 7

Enter the profits and wt of item 5:
6 1

Enter the profits and wt of item 6:
18 4

Enter the profits and wt of item 7:
3 1

Enter the capacityacity of knapsack:- 15

5.000000        1.666667        3.000000        1.000000        6.000000        4.500000        3.000000

The final solution is:-
 1.000000       1.000000        1.000000        1.000000        1.000000        0.666667        0.000000
Maximum profit is:- 55.333332
--------------------------------
Process exited after 40.54 seconds with return value 0
Press any key to continue . . .
```

Conclusion: Thus, the basic idea of the greedy approach is to calculate the ratio value/weight for each item and sort the item on basis of this ratio. Then take the item with the highest ratio and add them until we can't add the next item as a whole and at the end add the next item as much as we can. The knapsack problem is in combinatorial optimization problem. It appears as a subproblem in many, more complex mathematical models of real-world problems.