15 March 2021 10:09

General method →

* Greedy Algo solves problem which have n inputs and requires us to obtain subset of soln, that satisfies the
* constraints, this subset is known as **feasible soln**

We need to find feasible soln that either maximizes or minimizes the objective function, such feasible soln is called optimal solution

The greedy algorithm works in stages such that only one input is considered at a time The decision will be made at each stage whether the particular input is optimal or not.

* If the input results into infeasible soln cycle then the input is dropped.

```
Greedy (a, n)
{  // a[1... n] contains n inputs
    Solution = ∅
    for( i = 1 to n)
    {   x = select (a[i])
        if ( feasible (solution, x)) then
            Solution = union (solution, x);
    }
```

1. There are n inputs
2. Take one input at a time and add to soln set (initially Empty)
3. If the added inputs results into feasible soln

Solution = union(solution, x).

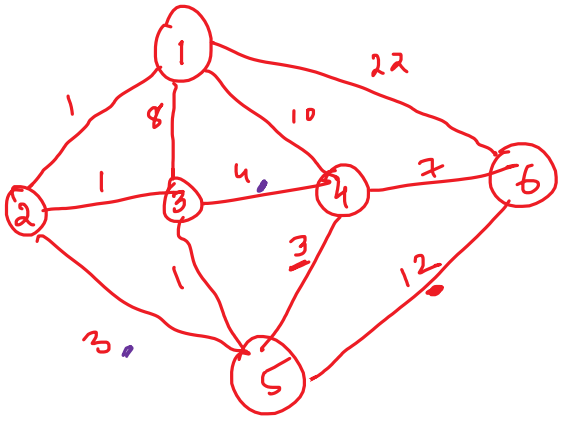
}

return solution

}

| into feasible soln accept it
else consider next input

Q)



Shortest path from current 1 → dest^n 6

current = 1 (source)

d[current] = d[1] = 0

visited[1] = 1

dc = 0

d

1	2	3	4	5	6
0	1	8	10	∞	22

p

1	2	3	4	5	6
0	1	1	1	0	1

visited

1	2	3	4	5	6
1	0	0	0	0	0

current = 2, dc = d[2] = 1

visited[2] = 1

1	2	3	4	5	6
0	1	2	10	4	22

1	2	3	4	5	6
0	1	2	1	2	1

1	2	3	4	5	6
1	1	0	0	0	0

current = 3 dc = d[3] = 2, visited[3] = 1

1	2	3	4	5	6
0	1	2	6	3	22

1	2	3	4	5	6
0	1	2	3	2	1

1	2	3	4	5	6
1	1	1	0	0	0

current = 5 dc = d[5] = 3 visited[5] = 1

1	2	3	4	5	6
0	1	2	6	3	15

1	2	3	4	5	6
0	1	2	3	3	5

1	2	3	4	5	6
1	1	1	0	1	0

current = 4 dc = d[4] = 6 visited[4] = 1

1	2	3	4	5	6
0	1	2	6	3	15

1	2	3	4	5	6
0	1	2	3	3	5

1	2	3	4	5	6
1	1	1	1	1	0

Since current == dest we will stop

} source to

[Since current == dest we will stop

Total cost = d[dest] = d[4] = 6

Path :-> 1 - 2 - 3 - 4 (shortest path from Source to dest^n)] } Source to dest^n

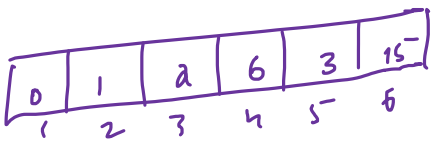
Qf

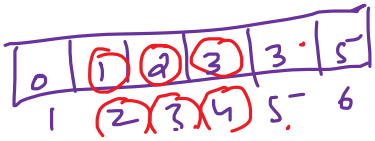
Find Shortest distance from Source to all the nodes :->

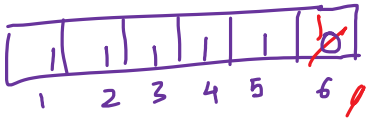
Now we will not stop when current = dest, Instead we will visited all the nodes

Now we will continue the Sol^n

Current = 6 dc = d[6] = 15 , visited[6] = 1







Now Shortest path from

Path

1 -> 2 = d[2] = 1 [1 - 2]

1 -> 3 = d[3] = 2 [1-2-3]

1 -> 4 = d[4] = 6 [1-2-3-4]

1 -> 5 = d[5] = 3 [1-2-3-5]

1 - 6 = d[6] = 15 [1-2-3-5-6]

Gives Shortest Path from Single Source 1 to All other Vertex.

Algo (g, V, E)

```
{   current = source
    d[current] = 0
    visited[current] = 1
        dc = 0
    c = 1
while( c != v)
{
  for( i=1 to v)
  {
    if ( g[current][i] != 0  && visited[i] != 1)
    {
      if( g[current][i] + dc < d[i])
      {  d[i] = g[current][i] + dc
         p[i] = current;
      }
    }
  }
current = Node from Unvisited at at Smallest distance
  visited[current] = 1
   dc = d[current]
   c = c + 1.
} // while
```

To display Cost:

```
for(i=1 to v)
print ("Shortest distance from" source "to" i "is"
                                              d[i])
```

Knapsack Problem

We are given an empty knapsack of capacity 'm', We are given n different objects weights of object i is represented as w[i] profit on object i is represented as p[i].

We want to fill knapsack with total capacity 'm' such that profit always remains maximum

Let $x[i] = 0$, if i^{th} object is not added in knapsack.

$x[i] = 1$, if i^{th} object is added in knapsack

The problem is to maximize $\sum_{i=1}^{n} p_i * x_i$

Subjected to $\sum_{i=1}^{n} w_i * x_i \leq m$

Problem can be solved by Considering objects in Decreasing order of Profit by weight

Hence, the problem is solved by Greedy Approach.

Q) Consider $m=15$ $n=7$

P_1	P_2	P_3	P_4	P_5	P_6	P_7
10	5	15	7	6	18	3

w_1	w_2	w_3	w_4	w_5	w_6	w_7
2	3	5	7	1	4	1

Ans → Type 1) Fractional Knapsack.

Step 1) find P/W of each object [Per Unit Cost]

	Object	Profit	Weight	P/W
(2)	1	10	2	5
(3)	2	5	3	1.67
(4)	3	15	5	3
(6)	4	7	7	1
(1) ✓	5	6	1	6
(2)	6	18	4	4.5
(4)	7	3	1	3

Step 2 → Arrange objects in Descending order of P/W

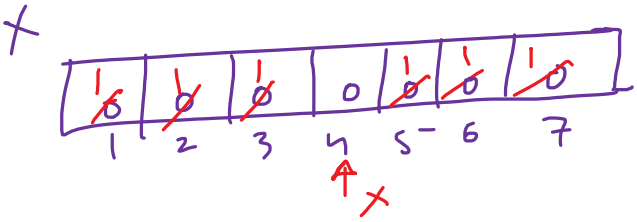
Object	Profit	Weight	P/W
5 ✓	6.	1.	6
1	10.	2.	5
6.	18.	4.	4.5
3	15	5.	3
7	3.	1.	3
2	5	3.	1.67
4	7	7.	1

Step 3 → $m=15$

Step 3 → m = 15

Object Added	Weight	Remaining Capacity	Profit Earned
5	1	14	6
1	2	12	6+10=16
6	4	8	16+18=34
3	5	3	34+15=49
7	1	2	49+3=52
2	2	0	52+1.67×2 = 55.34

Total Profit = 55.34



Fractional Knapsack → We are Allowed to take fraction of object that will Satisfy the capacity

Note' There is Concept of 0/1 Knapsack
0 = object is not considered
1 = " " considered
} Either you will take object with all its weight or will not take at all

Step 3 → m = 15

Object Added	Weight	Remaining Capacity	Profit Earned
5	1	14	6
		12	6+10=16

5	1	14	0
1	2	12	6+10=16
6	4	8.	16+18=34
3	5	3	34+15=49
7	1	2	49+3=52

Here Remaining capacity of Sack is 2

We are left with 2 objects: object 2 with weight = 3
object 4 with weight = 7

Since weights of all the remaining objects Exceeds the remaining capacity

& We are Implementing 0/1 Knapsack, so we cannot Consider any of the remaining Object

In this Case Total Profit Earned = 52
& Remaining Capacity = 2

Disadvantage of 0/1 Using Greedy ⇒ May happen that Sack is not Completely utilized
So profit is not max^m

Note 0/1 Knapsack will effectively Solved using "Dynamic Programming Approach"

Greedy Algorithm for Knapsack Problem

Given an empty knapsack of capacity 'm' and n different objects

Weights and profits for all n objects are stored in array w and p.

We want to fill the knapsack with entire capacity 'm' such that the profit earned is maximum

```
Algo KnapSack(m, n, w[], p[])
{
// Sort both p & w in decreasing order of p/w
  Let  c = m  // Initial Capacity
  Let profit = 0 // initial profit

    // Fill the Sack
for(i = 1 to n)
{   if(c - w[i] >= 0)
    {   c = c - w[i];
        profit = profit + p[i];
    Display object added with weight w[i]
      & profit earned is p[i]
    }
    else
```

```
    }
    else
      break;
  }
  if ( i <= n)
  {  profit = profit + (p[i]/w[i]) * c ;

     Display object added with weight c and
     Profit earned (p[i]/w[i])*c ;
  }
  display  max Profit = Profit
}
```

Job Sequencing with Deadline

- Given 'n' different jobs to be performed using single m/c
- Profit and deadline for each job is specified
- One job can be performed at a time.
- Every job requires one unit of time
- We want to find subset of the given job such that all the jobs in the subset are completed within their deadline and profit earned is maximum.

Ex → Consider

Job	1	2	3	4
Profit	100	15	10	27
Deadline	2	1	2	1

Possible soln → {1, 3} ⇒ 100+10 = 110
0-1 1-2

{2, 1} ⇒ 15+100 = 115
0-1 1-2

Soln → step 1: Arrange the jobs in decreasing order of profit

Job	1	4	2	3
Profit	100	27	15	10
Deadline	2	1	1	2

Job	1	4	2	3
Profit	100	27	15	10
Deadline	2	1	1	2

Step 2 → (I) Add job 1 to Soln.

$\{1\}$ (0-1) → done ⇒ Profit Earned = 100

(II) Add job 2 to Soln

$\{1, 2\}$ (0-1, 1→2) × Deadline of job 2 is till 1
but of job 1 is till 2

So we can reverse the order

$\{2, 1\}$ (0-1, 1-2) Profit = 27+100 = 127

Given the time limit, we cannot add other job

High level Description →

Algo JSD (d, J, n)

// J → set of jobs that can be completed by deadline

$J_1 = \{1\}$

for (i = 2 to n) do

{ if all job in J ∪ {i} can be

Completed by their deadline
then J = J ∪ {i}
}
}

(2) Compare Divide & Conquer & Greedy Approach

Divide & Conquer	Greedy
(1) To obtain soln to given problem	(1) Used to obtain optimal soln
(2) Problem is divided into subproblems and soln of subproblems are combined together to get soln of problem.	(2) Set of feasible sol is generated & optimal is selected.
(3) Here Duplicate soln may be obtained	(3) The optimal is selected without revisiting previous soln
(4) Less efficient as there is rework on soln	(4) more efficient than D.C but cannot guarantee optimal sol
(5) Binary Search, Merge Sort	(5) Knapsack, finding mcst