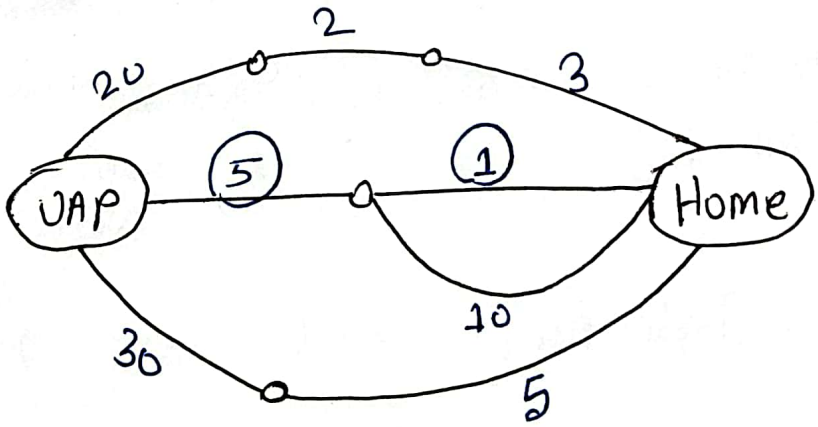
Greedy Algorithm —

- Makes locally optimal choice
- Once a decision made never reconsidered.
- do not always give optimal solution.



Example : 0/1 knapsack

Item	value	weight
1	$100	20
2	$120	30
3	$60	10

Capacity 50.

Objective : Max profit

constraint : capacity

Step-1 : Calculate value by weight for each item

Step-2 : Sort items based on the above ratio in descending order.

Step-3 : Take item or not (0/1).

Solution ⇒

Item	Value	weight	Value/weight
1	$100	20	5
2	$120	30	4
3	$60	10	6

After sorting in descending order –

Item	Value	Weight	Value/weight
3	$60	10	6
1	$100	20	5
2	$120	30	4

If weight <= capacity, pick item.
Then find remaining capacity.

(1) Item 3 → 10 < 50

pick item 3. capacity = 50 − 10 = 40

Profit = $60.

(2) Item 1 → 20 < 40

pick item 1. capacity = 40 − 20 = 20

Profit = $60 + $100 = ~~$150~~ $160

(3) Item 2 → 30 > 20.

Item 2 rejected.

∴ Max Profit = Item 3 + Item 1

= $60 + $100

= $160 [Ans]

Here, we can see that, ~~maximum~~ if we took item 1 and 2, then ~~we~~ the maximum profit would be $220 within the given capacity. If we took item 2 and 3, then the maximum profit would be $180 within given capacity. $220 and $180 are greater than $160. So, we can say that greedy algorithm doesn't give optimal solution of 0/1 knapsack.

Fractional knapsack -

এখানে Example এর ক্ষেত্রে,

Item - 2 rejected হবে না। সম্পূর্ণ capacity পাবে সম্পূর্ণ নেয়া হবে।

Item - 2 ⇒ Fraction ⇒ 20.

Max Profit ⇒

= Item 3 + Item 1 + Item 2 $\left(\frac{20}{30}\right)$

= 160 + 100 + $120^{4} \times \frac{20}{30}$

= $240

Example –

Capacity = 15

Item	1	2	3	4	5	6	7
Value	11	6	16	8	7	19	4
Weight	3	4	6	8	2	5	2
Value/weight	3.67	1.5	2.67	1	3.5	3.8	2
	2	6	4	7	3	1	5

$$x\begin{pmatrix} 1 & 0 & 5/6 & 0 & 1 & 1 & 0 \\ x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7 \end{pmatrix}$$

Item	Value	Weight	Remaining capacity	Total Profit
6	19	5	15-5=10	19
1	11	3	10-3=7	19+11=30
5	7	2	7-2=5	30+7=37
3	$\frac{5}{6}\times 16 = 13.3$	$\frac{5}{6}\times 6 = 5$	5-5=0	37+13.3 = 50.3

5<15

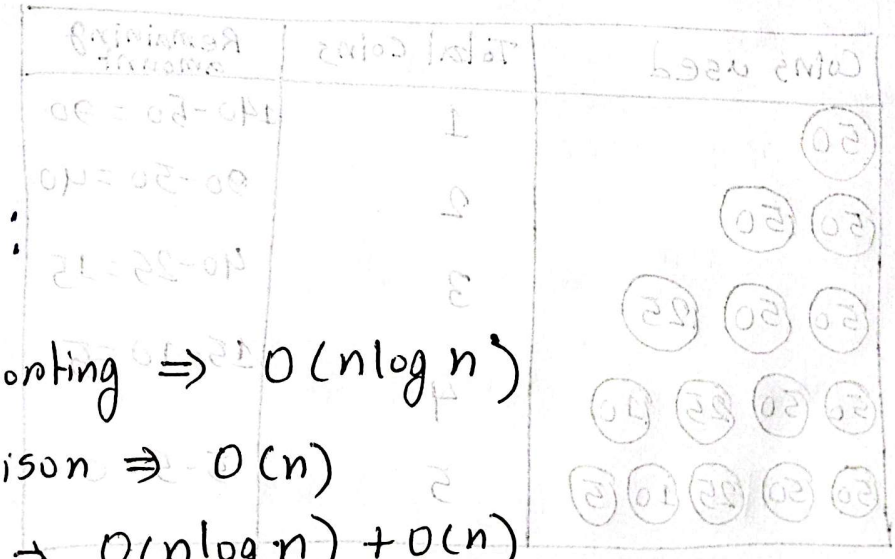
3<10

2<7

675

Algorithm:

```
for (i=0 ; i<n ; i++)
    if (arr[i].weight <= capacity)
    {
        c = c - arr[i].weight ;
        total_profit += arr[i].value ;
    }
else {
    frac = c / arr[i].weight ;
    total_profit = total_profit + frac*arr[i].value ;
}
```



Complexity:

For sorting $\Rightarrow O(n \log n)$

Comparison $\Rightarrow O(n)$

Overall $\Rightarrow O(n \log n) + O(n)$

Since $O(n \log n)$ is prominent -

Time complexity = ~~of For~~ $O(n \log n)$.

Coin Change Problem ⇒

Objective – Minimum number of coins.

Constraints – Coins are available in infinite numbers.

Example –

$n = 140$

$C = \{5, 10, 20, 25, 50\}$

After sorting –

$C = \{50, 25, 20, 10, 5\}$

If $C[i] <= n$, pick 1 coin

	Coins used	Total coins	Remaining amount
$50 \leq 140$	(50)	1	$140 - 50 = 90$
$50 \leq 90$	(50) (50)	2	$90 - 50 = 40$
$25 <= 40$	(50) (50) (25)	3	$40 - 25 = 15$
$10 <= 15$	(50) (50) (25) (10)	4	$15 - 10 = 5$
$5 = 5$	(50) (50) (25) (10) (5)	5	$5 - 5 = 0$

Number of coins used = 5.

*"Greedy doesn't give optimal solution of coin change problem every time" - Explain with example.

⇒ Assume, $n = 6$ and $C = \{1, 3, 4\}$.

After sorting, $C = \{4, 3, 1\}$

Coins	Total	Remaining
(4)	1	6-4=2
(4)(1)	2	2-1=1
(4)(1)(1)	3	1-1=0

Here, we can see that, if we took two coins of amount 3. That would be the minimum number of coins to make 6 with the available coins. But, using greedy algorithm, we found that 3 coins are needed for making 6 with the available coins. So, greedy doesn't always give optimal solution for coin change problem.

Algorithm :

```
int greedyCoinChange (int c[], int n, int i) {
    if (n == 0) return 0;
    if (c[i] <= n)
        return 1 + greedyCoinChange (c, n-c[i], i);
    else
        return greedyCoinchange (c, n, i+1);
}
```

Complexity ⇒

Array size = m

For sorting ⇒ O(m log m)

Total amount making ⇒ O(n).

Overall ⇒ O(m log m) + O(n)

Here, n >> m.

So, O(n) is prominent.

Time complexity = O(n).

Job sequencing with deadline ⇒

JobID ⟶	1	2	3	4	5
Profit ⟶	10	5	20	1	15
Deadline ⟶	1	3	2	3	2

यहां JobID का कुल दो Timeslot।

यहां Timeslot 2 only यहां Job।

यह example में Timeslot यहां - 0 to 1, 1 to 2, 2 to 3.

After sorting ⇒

JobID →	3	5	1	2	4
Profit/Value ⟶	20	15	10	5	1
Deadline ⟶	2	2	1	3	3

JobID	Profit	Deadline	Slot assigned
J_3	20	2	[1,2]
J_5	15	2	[0,1]
J_1	10	1	Rejected.
J_2	5	3	[2,3]
J_4	1	3	Rejected

Solution - (J_3, J_5, J_2)

Profit $\Rightarrow$ $20+15+5 = 40$.

Complexity $\Rightarrow$

Sorting n jobs $= O(n \log n)$

Choice $= O(n)$

Overall $= O(n) + O(n \log n)$.

Since, $O(n \log n)$ is prominent,

Time complexity $= O(n \log n)$.