Day 12

09 December 2022 22:03

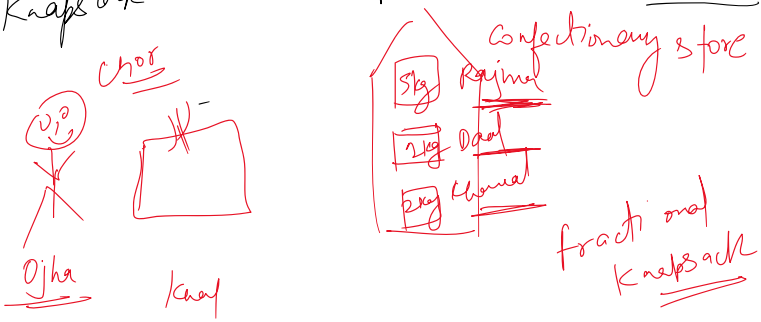
0/1 Knapsack

Knapsack Problem Types

Fractional Knapsack

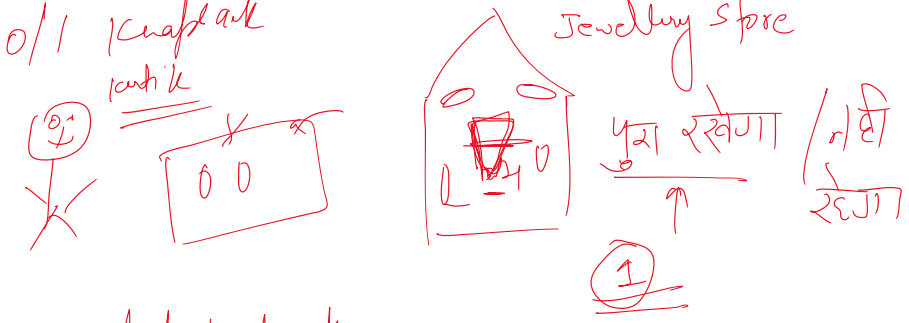
0/1 Knapsack

Unbounded Knapsack.



fractional knapsack

0/1 knapsack



Unbounded knapsack

Unbounded Knapsack

0-1 knapsack

Quantity of each item of

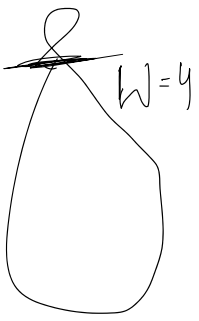
N items

we pick one item only once.

values | 1 | 2 | 3 |

weights | 4 | 5 | 1 |

I_1 I_2 I_3

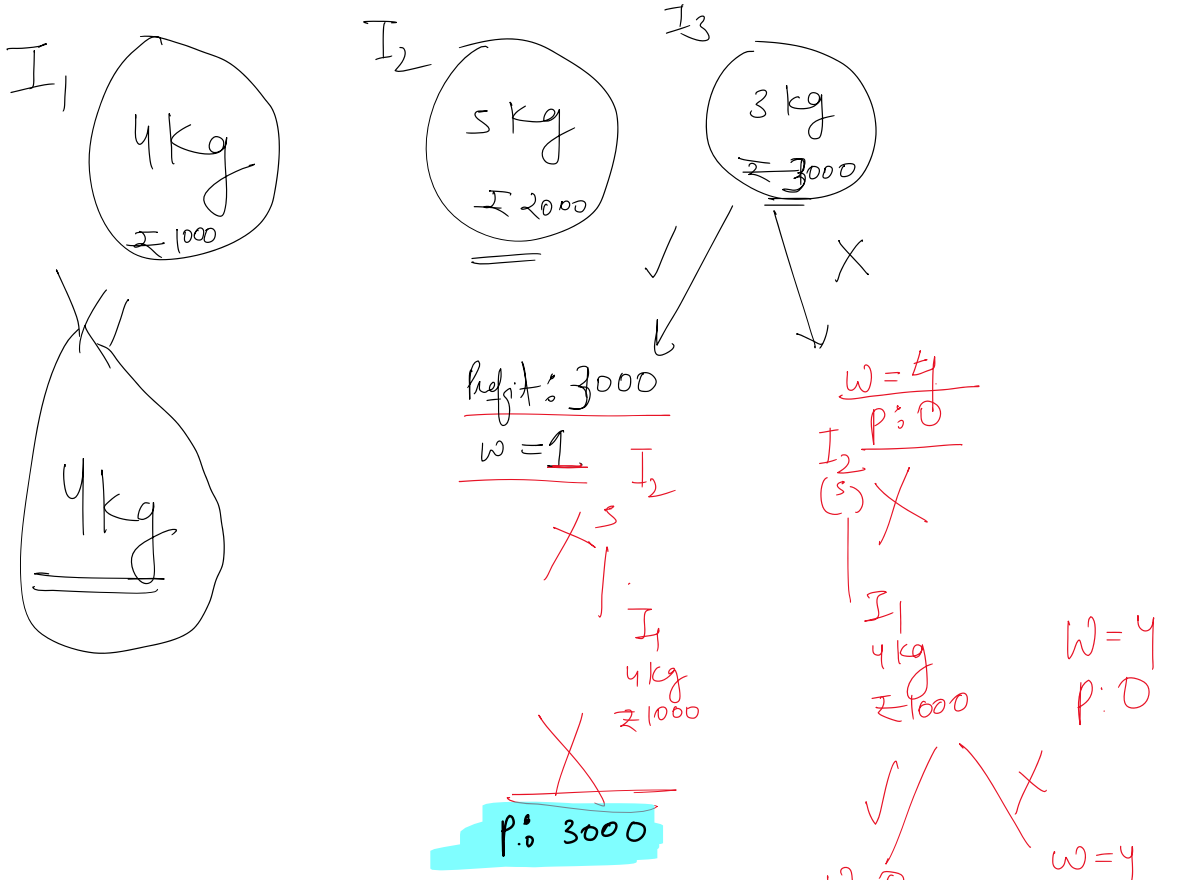


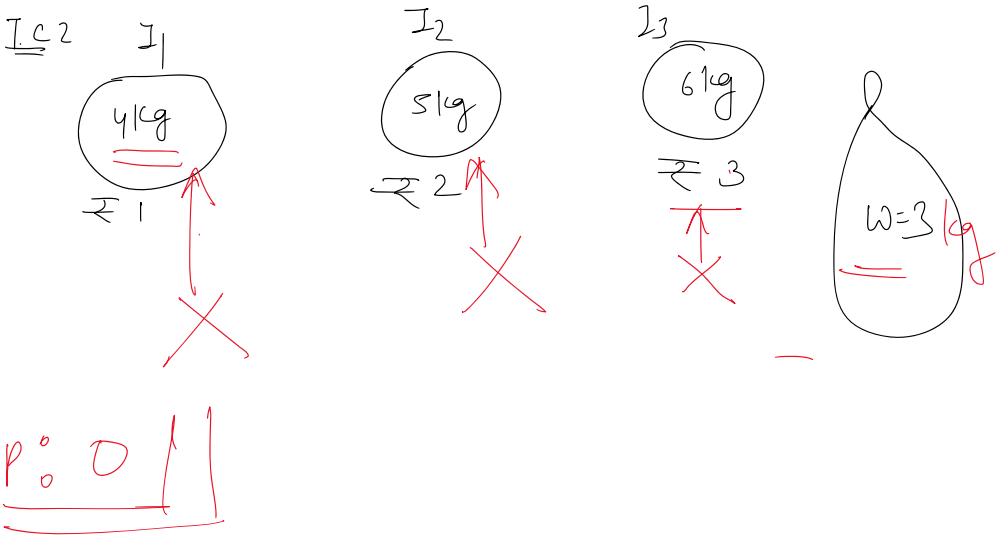
Maximum Profit

You cannot break an item

You cannot break an item

Fractional knapsack X





Choice + Decision = Recursion

Recursion (Top to Bottom)

T.C 3

	I_1	I_2	I_3	I_4
values	6	10	18	22
weight	2	4	3	5

$W = 5$

Recursive (top to bottom)

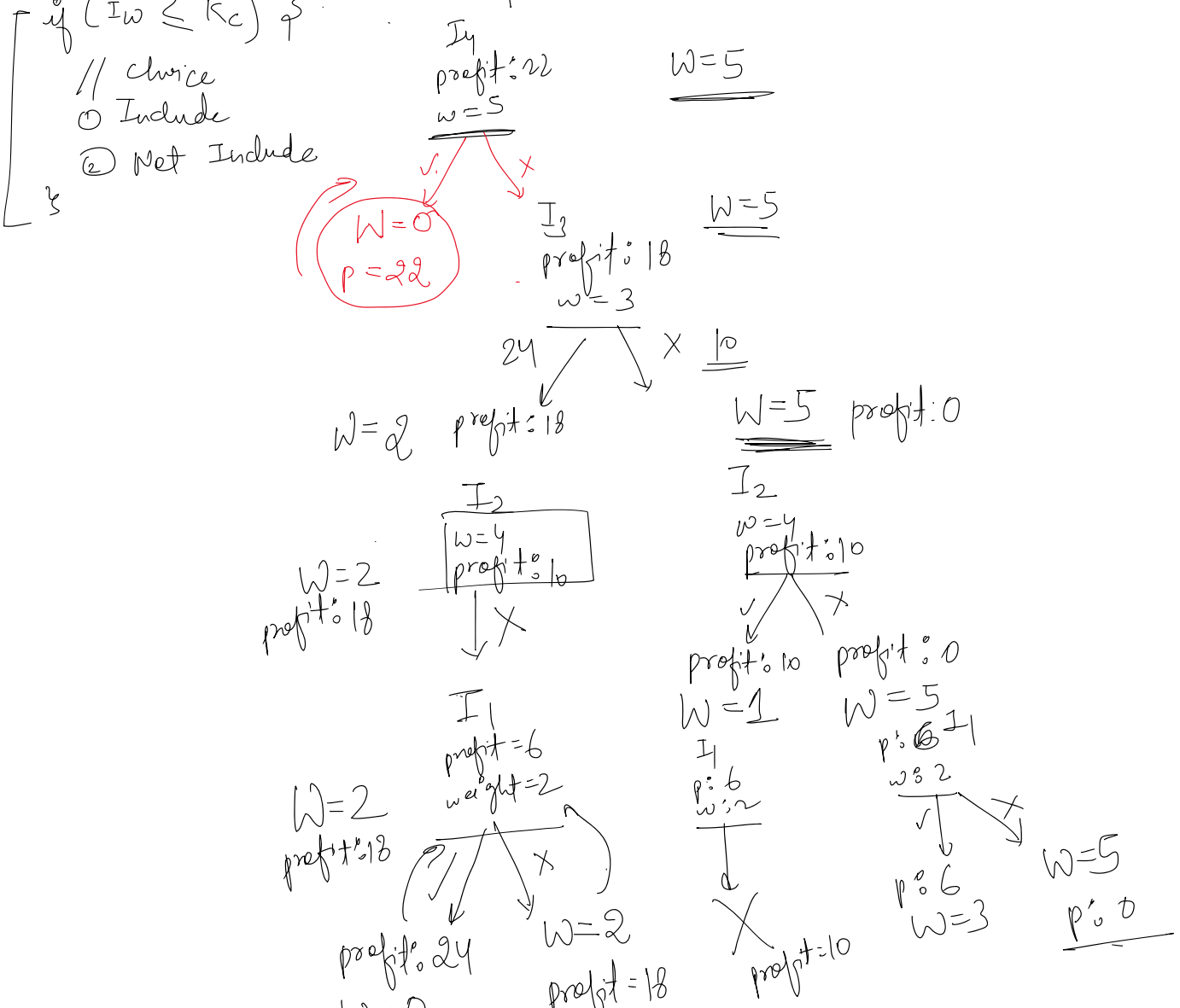
if ($I_w \leq K_c$) {

// choice

① Include

② Not Include

}



profit: 24

Choice Diagram

if(Iw <= Kc) {

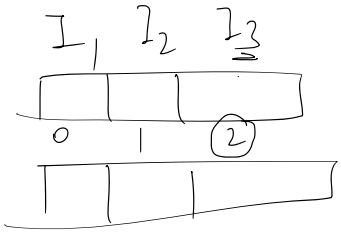
// Include

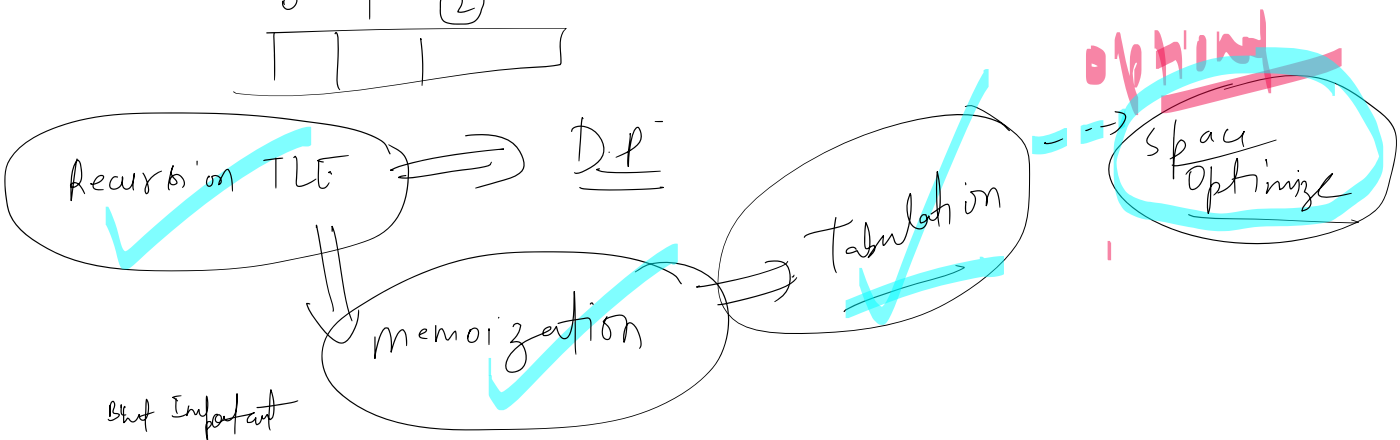
// Not Include

} else {

// Not Include

}





But Important

1. Changing variable

2-D → 1D → (1 - changing)

→ 2D → (2 - changing)

Recursive calls

f(2,1)

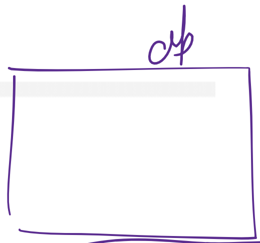
```
class Solution
{
  public:
  //Function to return max value that can be put in knapsack of capacity W.
  int solve(int Knapsack_Capacity,int Item_Weights[], int Item_Profit[], int No_of_Item){
     // base case
     if(No_of_Item == 0 || Knapsack_Capacity == 0) return 0;
     // choice diagram
     int maxi = 0;
     if(Item_Weights[No_of_Item-1] <= Knapsack_Capacity){
        // include
        maxi = max(maxi,solve(Knapsack_Capacity-Item_Weights[No_of_Item-1],Item_Weights,Item_Profit,No_of_Item-1) +
Item_Profit[No_of_Item-1]);

        // not include
         maxi =  max(maxi,solve(Knapsack_Capacity,Item_Weights,Item_Profit,No_of_Item-1));
     }else{
        // not include
```

```
        // not include
        maxi = max(maxi,solve(Knapsack_Capacity,Item_Weights,Item_Profit,No_of_Item-1));
      }
      return maxi;

   }
   int knapSack(int Knapsack_Capacity, int Item_Weights[], int Item_Profit[], int No_of_Items)
   {
     return solve( Knapsack_Capacity,Item_Weights, Item_Profit, No_of_Items);
   }
};
```

T.C - O(2^N)   S.C - O(N)

↑ No.of Items

```
int solve(int Knapsack_Capacity,int Item_Weights[], int Item_Profit[], int No_of_Item,vector<vector<int>>& dp){
    // base case
    if(No_of_Item == 0 || Knapsack_Capacity == 0) return 0;
     // check for recomputation
     if(dp[Knapsack_Capacity][No_of_Item] != -1) return dp[Knapsack_Capacity][No_of_Item];
    // choice diagram

    int maxi = 0;
    if(Item_Weights[No_of_Item-1] <= Knapsack_Capacity){
        // include
        maxi = max(maxi,solve(Knapsack_Capacity-Item_Weights[No_of_Item-1],Item_Weights,Item_Profit,No_of_Item-1,dp) + Item_Profit[No_of_Item-1]);

        // not include
         maxi =  max(maxi,solve(Knapsack_Capacity,Item_Weights,Item_Profit,No_of_Item-1,dp));
    }else{
        // not include
        maxi = max(maxi,solve(Knapsack_Capacity,Item_Weights,Item_Profit,No_of_Item-1,dp));
    }
    return dp[Knapsack_Capacity][No_of_Item] = maxi;

}
int knapSack(int Knapsack_Capacity, int Item_Weights[], int Item_Profit[], int No_of_Items)
{
    // knapsack capacity
    // NO_of_items
    vector<vector<int>> dp(Knapsack_Capacity + 1, vector<int>(No_of_Items +1,-1));

   return solve( Knapsack_Capacity,Item_Weights, Item_Profit, No_of_Items,dp);
}
```



T.C - O(knap_Capacity x No_of_Item)

S.C - O(No_of_Item X knap_capacity)

⟹ knap Capacity x No of Item

① DP (-1)||0

② Base Case (Recursion) ⟹ Initialization

③ 1st find value set start करना loop लगाना

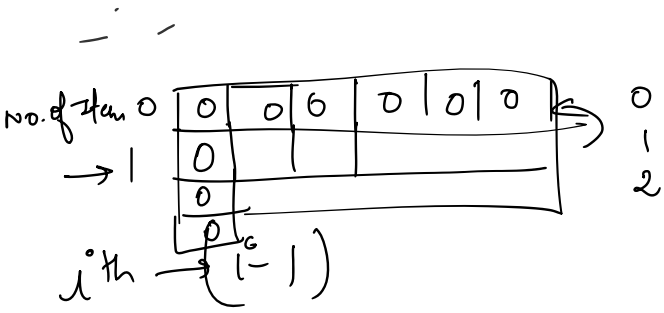
④ Recursive ⟹ DP[i][j]

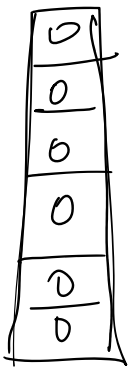
```cpp
int knapSack(int Knapsack_Capacity, int Item_Weights[], int Item_Profit[], int No_of_Items)
{
    // knapsack capacity
    // NO_of_items

    vector<vector<int>> dp(No_of_Items +1, vector<int>(Knapsack_Capacity + 1));
    // dp[i][j]
    // i--> NO_of_Items
    // j --> knapsack_capacity
    // intitilization
    for(int i = 0;i<Knapsack_Capacity + 1;i++){
        // no_of_item == 0
        dp[0][i] = 0;
    }
    for(int i = 0;i<No_of_Items +1;i++){
        // knapsack == 0
        dp[i][0] = 0;
    }

    for(int i = 1;i < No_of_Items +1;i++){
        for(int j= 1;j < Knapsack_Capacity + 1;j++){
            int maxi = 0;
            if(Item_Weights[i-1] <= j){
                // include
                maxi = max(maxi,dp[i-1][j-Item_Weights[i-1]] + Item_Profit[i-1]);

                // not include
                 maxi =  max(maxi,dp[i-1][j]);
            }else{
                // not include
               maxi =  max(maxi,dp[i-1][j]);
            }
            dp[i][j] = maxi;
        }
    }
//    return solve( Knapsack_Capacity,Item_Weights, Item_Profit, No_of_Items,dp);
    return dp[No_of_Items][Knapsack_Capacity];
}
};
```

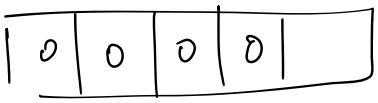
Press Esc to exit full screen

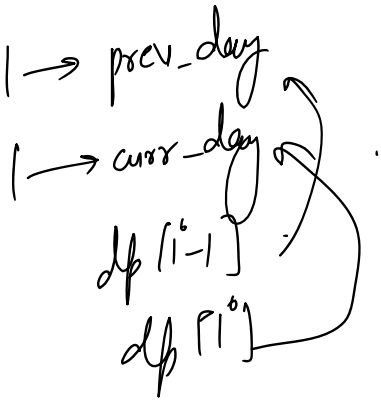
T.C - O(knap_capacity * No. of Items)

S.C - O(knap_capacity * No. of Items)









```
vector<int> curr_item(Knapsack_Capacity + 1,0);
for(int i = 1;i < No_of_Items +1;i++){
    auto prev_item = curr_item;
    for(int j= 1;j < Knapsack_Capacity + 1;j++){
        int maxi = 0;
        if(Item_Weights[i-1] <= j){
            // include
            maxi = max(maxi,prev_item[j-Item_Weights[i-1]] + Item_Profit[i-1]);

            // not include
             maxi =  max(maxi,prev_item[j]);
        }else{
            // not include
           maxi =  max(maxi,prev_item[j]);
        }
        curr_item[j] = maxi;
    }
}
 return solve( Knapsack_Capacity,Item_Weights, Item_Profit, No_of_Items,dp);
return curr_item[Knapsack_Capacity];
```

T.C-(kp x No of Item) S.C-O(kC)