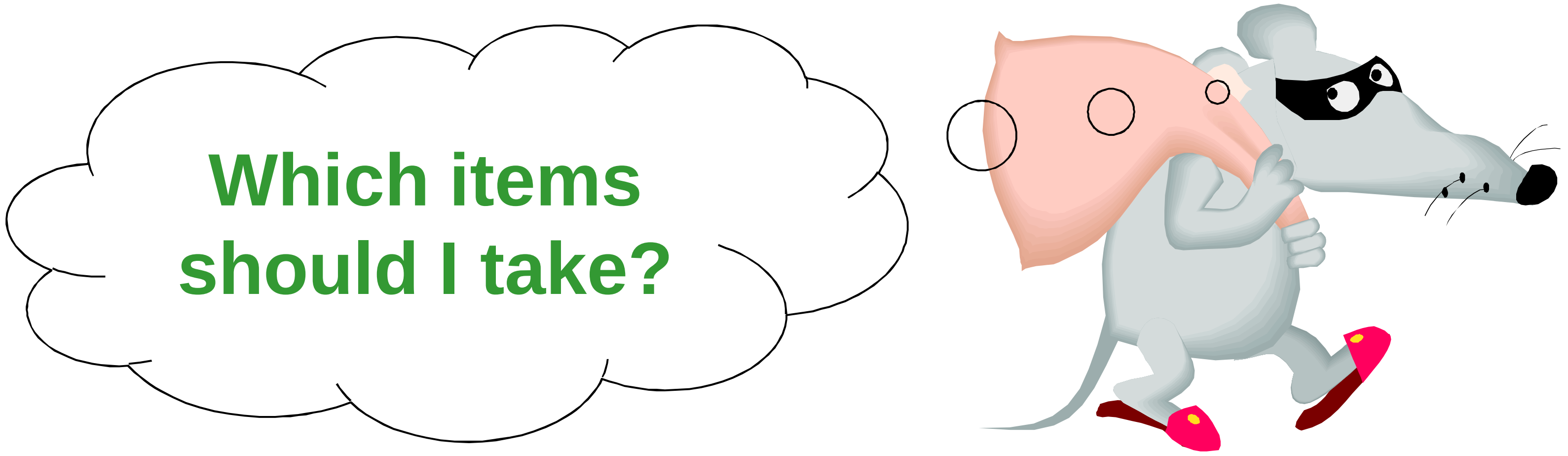
Dynamic Programming:

0-1 Knapsack Problem

The Knapsack Problem

The famous *Knapsack Problem*:

A thief breaks into a museum. Fabulous paintings, sculptures, and jewels are everywhere. The thief has a good eye for the value of these objects, and knows that each will fetch hundreds or thousands of dollars on the clandestine art collector's market. But, the thief has only brought a single knapsack to the scene of the robbery, and can take away only what he can carry. What items should the thief take to maximize the haul?



The Knapsack Problem

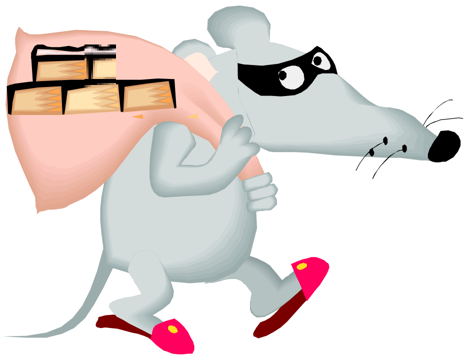
There are two versions of the problem:

(1) "0-1 knapsack problem"

Items are indivisible: you either take an item or not. Solved with *dynamic programming*.

(2) "Fractional knapsack problem"

Items are divisible: you can take any fraction of an item. Solved with a *greedy algorithm*.



0-1 Knapsack problem

- Thief has a knapsack with maximum capacity *W*, and a set *S* consisting of *n* items
- Each item *i* has some weight w_i and benefit value v_i (all w_i , v_i and *W* are integer values)
- Problem: How to pack the knapsack to achieve maximum total value of packed items?
- Goal:

find x_i such that for all $x_i \in \{0, 1\}$, $i = 1, 2, .., n$

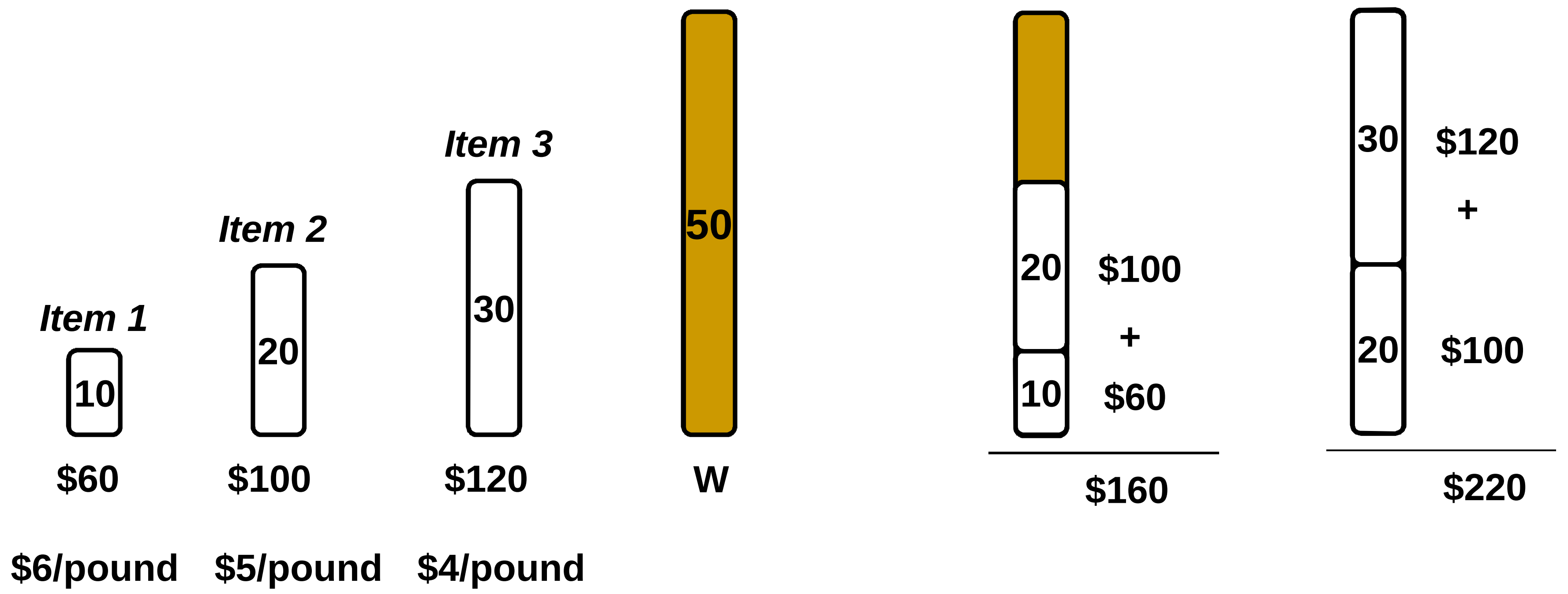
$\sum w_i x_i \leq W$ and

$\sum x_i v_i$ is maximum

Optimal Substructure Property

- Both problems exhibit the optimal substructure property.
- To show this for both the problems, consider the most valuable load weighing at most W pounds
  - Q: If we remove item j from the load, what do we know about the remaining load?
  - A: The remaining load must be the most valuable load weighing at most $W - w_j$ that the thief could take from the n-1 original items excluding item j.

0-1 Knapsack - Greedy Strategy Fails



0-1 Knapsack: Brute-Force Approach

- Since there are n items, there are 2^n possible combinations of items.
- We go through all combinations and find the one with the most total value and with total weight less or equal to W.
- Running time will be $O(2^n)$.

0-1 Knapsack - Dynamic Programming

- P(i, w) – the maximum profit that can be obtained from items 1 to i, if the knapsack has size w
- Case 1: thief takes item i

  $P(i, w) = v_i + P(i - 1, w - w_i)$
- Case 2: thief does not take item i

  $P(i, w) = P(i - 1, w)$

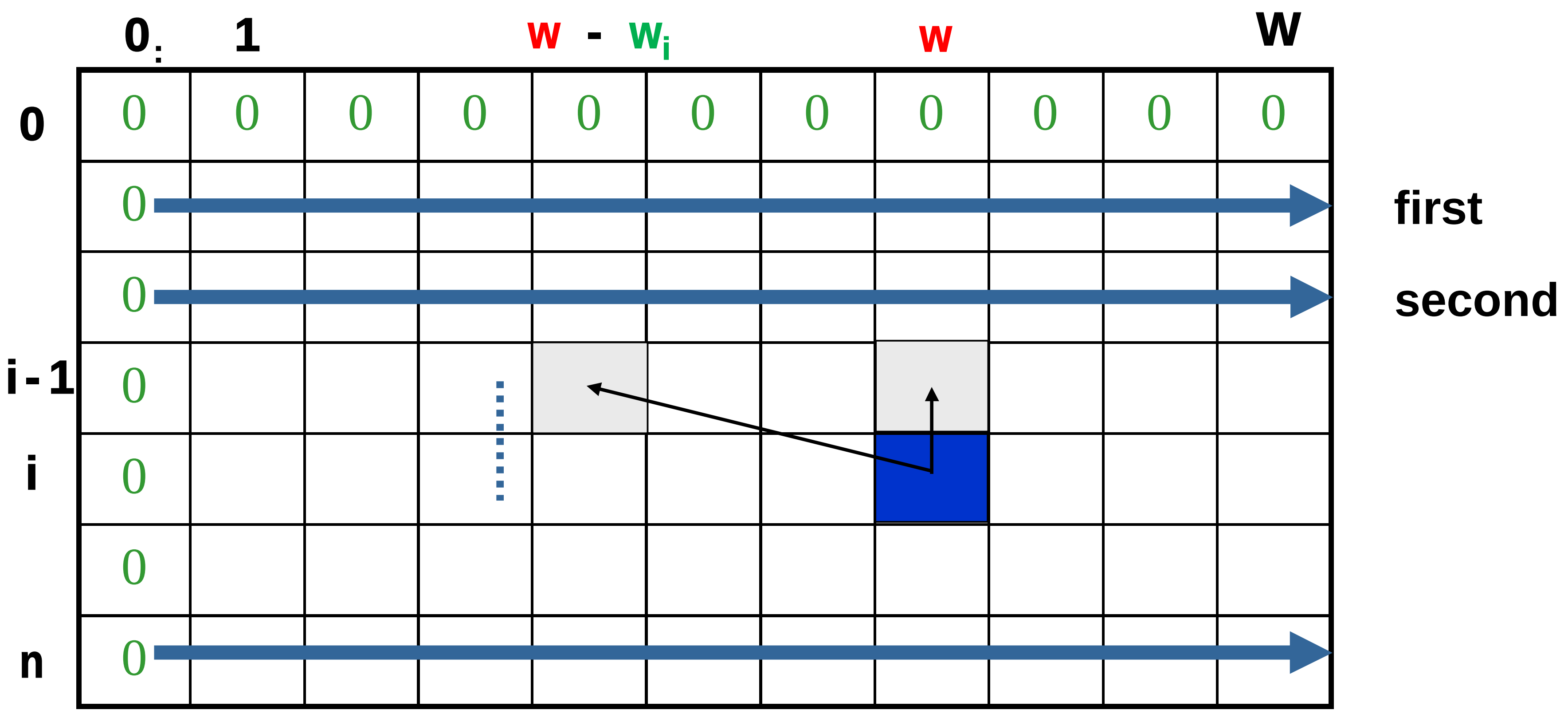
Recursive Formula

$$P[i,w] = \begin{cases} P[i-1,w] & \text{if } w < w_i \\ \max\{v_i + P[i-1, w-w_i], P[i-1,w]\} & \text{else} \end{cases}$$

- The best subset that has the total weight *w*, either contains item *i* or not.
- First case: $w < w_i$. Item *i* can't be part of the solution, since if it was, the total weight would be > *w*, which is unacceptable.
- Second case: $w >= w_i$. Then the item *i* can be in the solution, and we choose the case with greater value.

Dependencies among Subproblems

$$P(i, w) = \max \{\underbrace{v_i + P(i-1, w-w_i)}_{\text{Item } i \text{ was taken}},\ \underbrace{P(i-1, w)}_{\text{Item } i \text{ was not taken}}\}$$



Overlapping Subproblems

$P(i, w) = \max \{v_i + P(i - 1, w-w_i), P(i - 1, w) \}$

	0	1				w					W
0	0	0	0	0	0	0	0	0	0	0	0
	0										
	0										
i-1	0					■					
i	0					▨	▨	▨	▨	▨	▨
	0										
n	0										

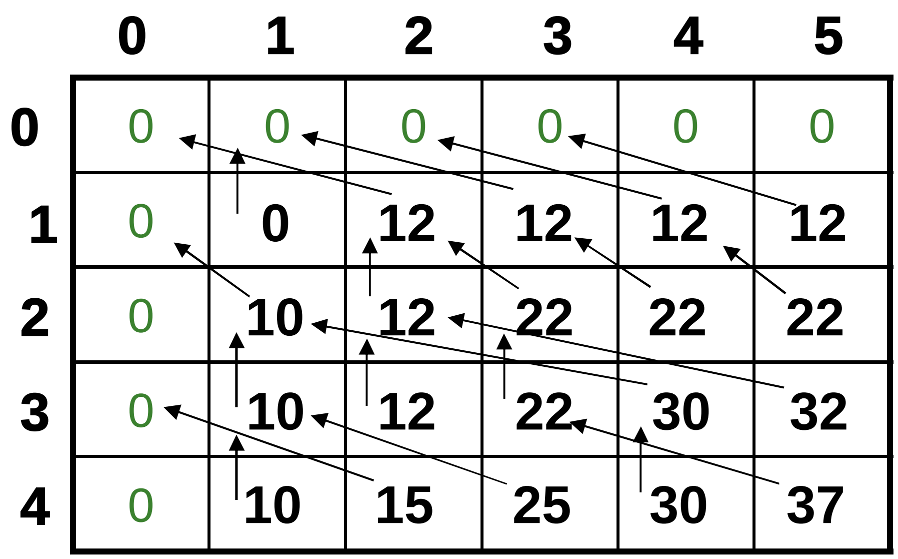
***E.g.*: all the subproblems shown in grey may depend on P(i-1, w)**

Example:

W = 5

Item	Weight	Value
1	2	12
2	1	10
3	3	20
4	2	15

$$P(i, w) = \max \{v_i + P(i - 1, w-w_i), P(i - 1, w) \}$$



	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0	10	15	25	30	37

P(1, 1) = P(0, 1) = 0

P(1, 2) = max{12+0, 0} = 12

P(1, 3) = max{12+0, 0} = 12

P(1, 4) = max{12+0, 0} = 12

P(1, 5) = max{12+0, 0} = 12

P(2, 1)= max{10+0, 0} = 10

P(2, 2)= max{10+0, 12} = 12

P(2, 3)= max{10+12, 12} = 22

P(2, 4)= max{10+12, 12} = 22

P(2, 5)= max{10+12, 12} = 22

P(3, 1)= P(2,1) = 10

P(3, 2)= P(2,2) = 12

P(3, 3)= max{20+0, 22}=22

P(3, 4)= max{20+10,22}=30

P(4, 5)= max{20+12,22}=32

P(4, 1)= P(3,1) = 10

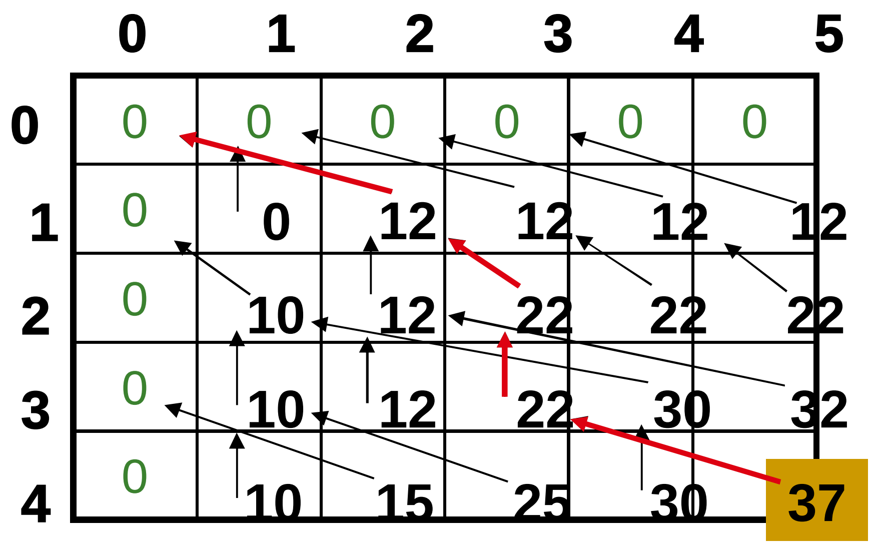
P(4, 2)= max{15+0, 12} = 15

P(4, 3)= max{15+10, 22}=25

P(4, 4)= max{15+12, 30}=30

P(4, 5)= max{15+22, 32}=37

Reconstructing the Optimal Solution



	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0	10	15	25	30	37

- Item 4
- Item 2
- Item 1

- Start at P(n, W)
- When you go left-up ⇒ item i has been taken
- When you go straight up ⇒ item i has not been taken

0-1 Knapsack Algorithm

```
for w = 0 to W
    P[0, w] = 0
for i = 0 to n
    P[i, 0] = 0
    for w = 0 to W
        if $w_i$ <= w // item i can be part of the solution
            if ($v_i$ + P[i-1, w-$w_i$] > P[i-1, w])
                P[i, w] = $v_i$ + P[i-1, w- $w_i$]
            else
                P[i, w] = P[i-1, w]
        else P[i, w] = P[i-1, w]  // $w_i$ > w
```

Running time: $O(n*W)$