The 0/1 Knapsack Problem

If we limit the x_i to only 1 or 0 (take it or leave it), this results in the 0/1 Knapsack problem.

Optimization Problem: find $x_1, x_2, ..., x_n$, such that:

$$\begin{cases} \text{maximize:} & \sum_{i=1}^{n} p_i \cdot x_i \\ \text{subject to:} & \sum_{i=1}^{n} w_i \cdot x_i \leq m \\ & x_i \in \{0,1\}, 1 \leq i \leq n \end{cases}$$

The Greedy method does not work for the 0/1 Knapsack Problem!

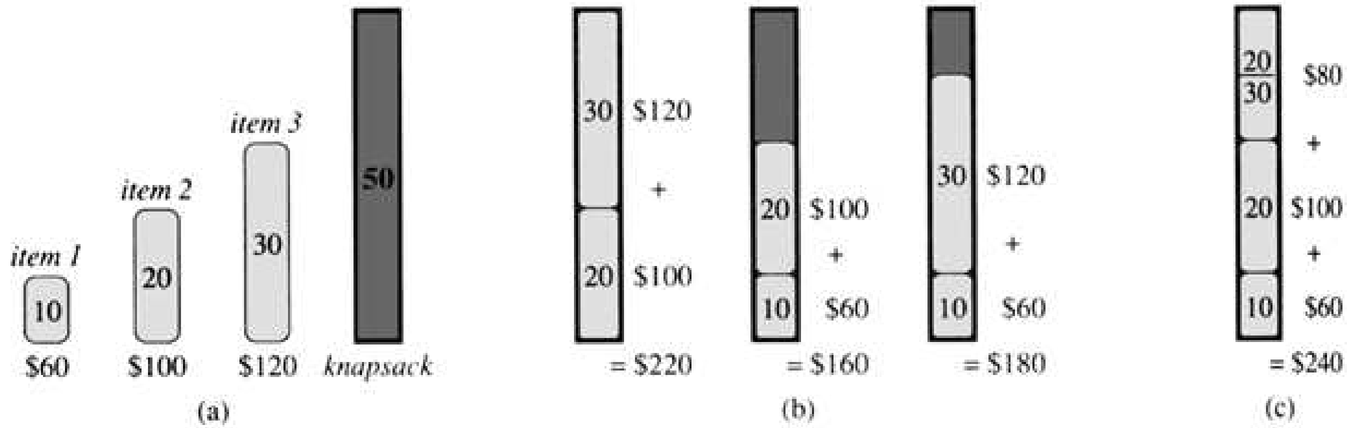


Figure 17.2 The greedy strategy does not work for the 0-1 knapsack problem. **(a)** The thief must select a subset of the three items shown whose weight must not exceed 50 pounds. **(b)** The optimal subset includes items 2 and 3. Any solution with item 1 is suboptimal, even though item 1 has the greatest value per pound. **(c)** For the fractional knapsack problem, taking the items in order of greatest value per pound yields an optimal solution.

The Knapsack Problem

There are two versions of the problem:

1. "Fractional" knapsack problem.
2. "0/1" knapsack problem.

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

2 Item are indivisible; you either take an item or not. Solved with **dynamic programming**.

0/1 Knapsack problem: the brute-force approach

Let's first solve this problem with a straightforward algorithm:

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

Can we do better?

- Yes, with an algorithm based on **dynamic programming**.
- Two key ingredients of optimization problems that lead to a dynamic programming solution:
  - **Optimal substructure:** an optimal solution to the problem contains within it optimal solutions to subproblems.
  - **Overlapping subproblems:** same subproblem will be visited again and again (i.e., subproblems share subsubproblems).

Optimal Substructure of 0/1 Knapsack problem

- Let KNAP(**1, n, M**) denote the 0/1 Knapsack problem, choosing objects from $[\mathbf{1..n}]$ under the capacity constraint of **M**.

- If $(x_1, x_2, ..., x_n)$ is an optimal solution for the problem KNAP(1, n, M), then:

  1 If $x_n = 0$ (we do not pick the n-th object), then $(x_1, x_2, ..., x_{n-1})$ must be an optimal solution for the problem KNAP(1, n-1, M).

  2 If $x_n = 1$ (we pick the n-th object), then $(x_1, x_2, ..., x_{n-1})$ must be an optimal solution for the problem KNAP(1, n-1, M - w_n).

Proof: Cut-and-Paste.

Solution in terms of subproblems

Based on the optimal substructure, we can write down the solution for the 0/1 Knapsack problem as follows:

- Let C[**n, M**] be the value (total profits) of the optimal solution for KNAP(1, **n, M**).

C[n, M] = max (profits for case 1,
profits for case 2)
= max (C[n-1, M], C[n-1, M - w_n] + p_n).

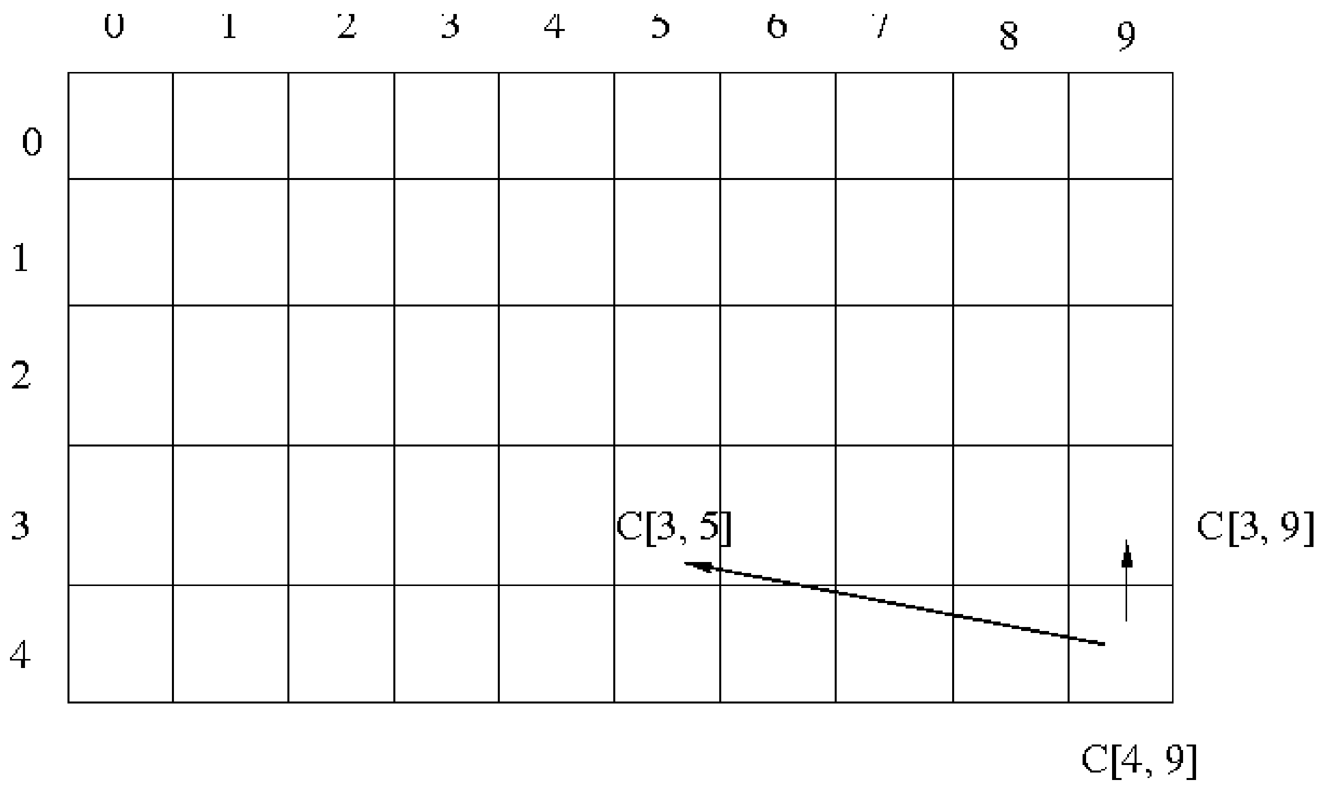
Similarly

C[n-1, M] = max (C[n-2, M], C[n-2, M - w_{n-1}] + p_{n-1}).
C[n-1, M - w_n] = max (C[n-2, M - w_n],
C[n-2, M - w_n - w_{n-1}] + p_{n-1}).

Use a table to store C[·,·] and build it in a bottom up fashion

- For example, if n = 4, M = 9; $w_4 = 4$, $p_4 = 2$, then C[4, 9] = max(C[3, 9], C[3, 9 - 4] + 2).
- We can use a 2D table to contain C[·,·]; If we want to compute C[4, 9], C[3, 9] and C[3, 9 - 4] have to be ready.
- Look at the value C[n, M] = max (C[n - 1, M], C[n-1, M - w_n] + p_n), to compute C[n, M], we only need the values in the row C[n - 1,·].
- So the table C[·,·] can be built in a bottom up fashion: 1) compute the first row C[0, 0], C[0, 1], C[0, 2] ... etc; 2) row by row, fill the table.

Programming = Table



- The term "programming" used to refer to a tabular method, and it predates computer programming.

Construct the table: A recursive solution

- Let C[i, ϖ] be a cell in the table C[$\cdot$,$\cdot$]; it represents the value (total profits) of the optimal solution for the problem KNAP(1, i, ϖ), which is the subproblem of selecting items in [$1..i$] subject to the capacity constraint of ϖ.

- Then C[i, ϖ] = max(C[$i-1, \varpi$], C[$i-1, \varpi$ - w_i] + p_i).

Boundary conditions

We need to consider the boundary conditions:

- When $i = 0$; no object to choose, so C$[i, \varpi] = 0$;
- When $\varpi = 0$; no capacity available, C$[i, \varpi] = 0$;
- When $w_i > \varpi$; the current object i exceeds the capacity, definitely we can not pick it. So C$[i, \varpi] =$ C$[i-1, \varpi]$ for this case.

Complete recursive formulation

Thus overall the recursive solution is:

$$C[i,\varpi] = \begin{cases} 0 & \text{if } i = 0 \text{ or } \varpi = 0 \\ C[i-1,\varpi] & \text{if } w_i > \varpi \\ max(C[i-1,\varpi], C[i-1,\varpi - w_i] + p_i) & \text{if } i > 0 \text{ and } \varpi \geq w_i. \end{cases}$$

The solution (optimal total profits) for the original 0/1 problem KNAP(1, n, M) is in C[n, M].

Algorithm

```
DP-01KNAPSACK(p[], w[], n, M) // n: number of items; M: capacity
    for ϖ := 0 to M      C[0,ϖ ] := 0;
    for i := 0 to n      C[i, 0] := 0;

    for i := 1 to n
        for ϖ := 1 to M
            if (w[i] > ϖ)      // cannot pick item i
                C[i, ϖ] := C[i - 1, ϖ];
            else
                if ( p[i] + C[i-1, ϖ - w[i]]) > C[i-1, ϖ])
                        C[i, ϖ] := p[i] + C[i - 1, ϖ - w[i]];
                else
                        C[i, ϖ] := C[i - 1, ϖ];

    return C[n, M];
```

Complexity: $\Theta(nM)$

```
DP-01KNAPSACK(p[], w[], n, M) // n: number of items; M: capacity
    for ϖ := 0 to M     C[0,ϖ ] := 0;                 — Θ(M)
    for i := 0 to n     C[i, 0] := 0;                 — Θ(n)

    for i := 1 to n                                   — n
        for ϖ := 1 to M                               — M
            if (w[i] > ϖ)
                C[i, ϖ] := C[i - 1, ϖ];
            else
                if ( p[i] + C[i-1, ϖ - w[i]]) > C[i-1, ϖ])
                    C[i, ϖ] := p[i] + C[i - 1, ϖ - w[i]];
                else
                    C[i, ϖ] := C[i - 1, ϖ];

    return C[n, M];
```

An example

Let's run our algorithm on the following data:

n = 4 (number of items)
M = 5 (knapsack capacity = maximum weight)
(w_i, p_i): (2, 3), (3, 4), (4, 5), (5, 6)

Execution

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0					
2	0					
3	0					
4	0					

Compute C[2, 5]

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	**7**
3	0					
4	0					

Compute C[4, 5]

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	**7**

How to find the actual items in the Knapsack?

- All of the information we need is in the table.
- C[n, M] is the maximal value of items that can be placed in the Knapsack.
- Let $i =$ n and $k =$ M

if C$[i, k] \neq$ C$[i-1, k]$ then
    mark the i-th item as in the knapsack
    $i = i - 1, k = k - w_i$.
else
    $i = i - 1$

Finding the items

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

Solution: $\{1, 1, 0, 0\}$

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7