Dynamic Programming
0/1 Knapsack

(Class 21)

- In previous class we studied matrix multiplication problem.
- Here is the dynamic programming based algorithm for computing the minimum cost of chain matrix multiplication.

```
MATRIX-CHAIN (p,N)
for i=1 to N
   do m[i, i] ← 0
for L=2 to N
   for i=1 to n-L+1
      do j ← i+L-1
      m[i, j] ← ∞
      for k=1 to j-1
         do t ← m[i, k] + m[k+1, j] + ($p_{i-1}$ * $p_k$ * $p_j$)
         if (t < m[i, j])
            then m[i, j] ← t; s[i, j] ← k
```

Running Time Analysis

- There are three nested loops.
- Each loop executes a maximum n times.
- Total time is thus $O(n^3)$.
- The s matrix stores the values k.
- The s matrix can be used to extracting the order in which matrices are to be multiplied.

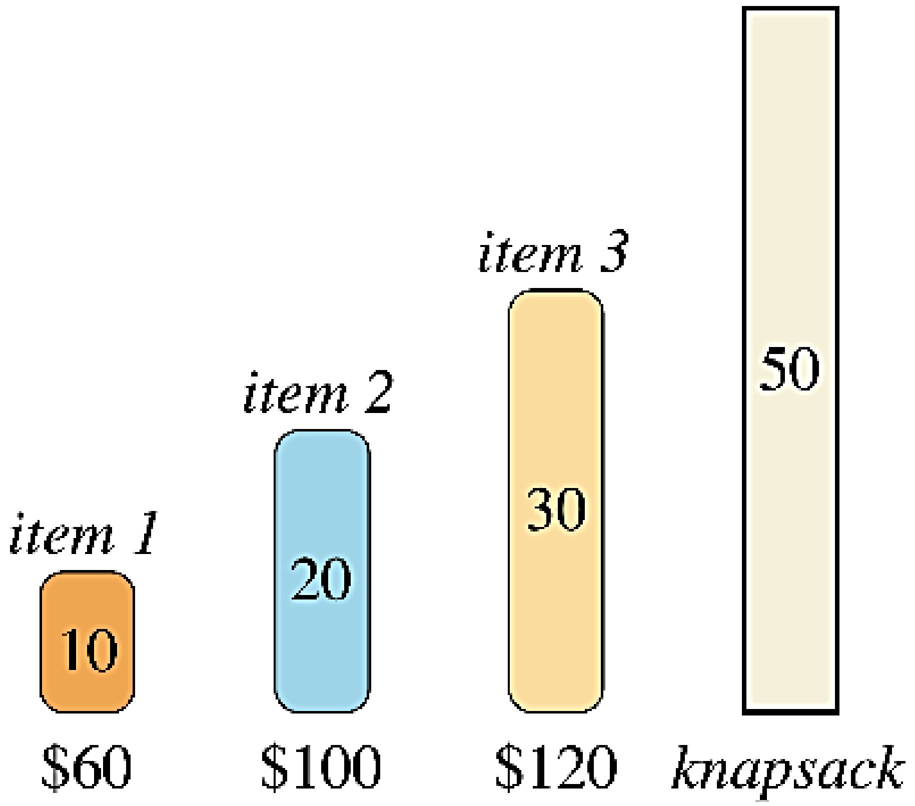
- Here is the algorithm that carries out the matrix multiplication to compute $A_{i\ldots j}$:

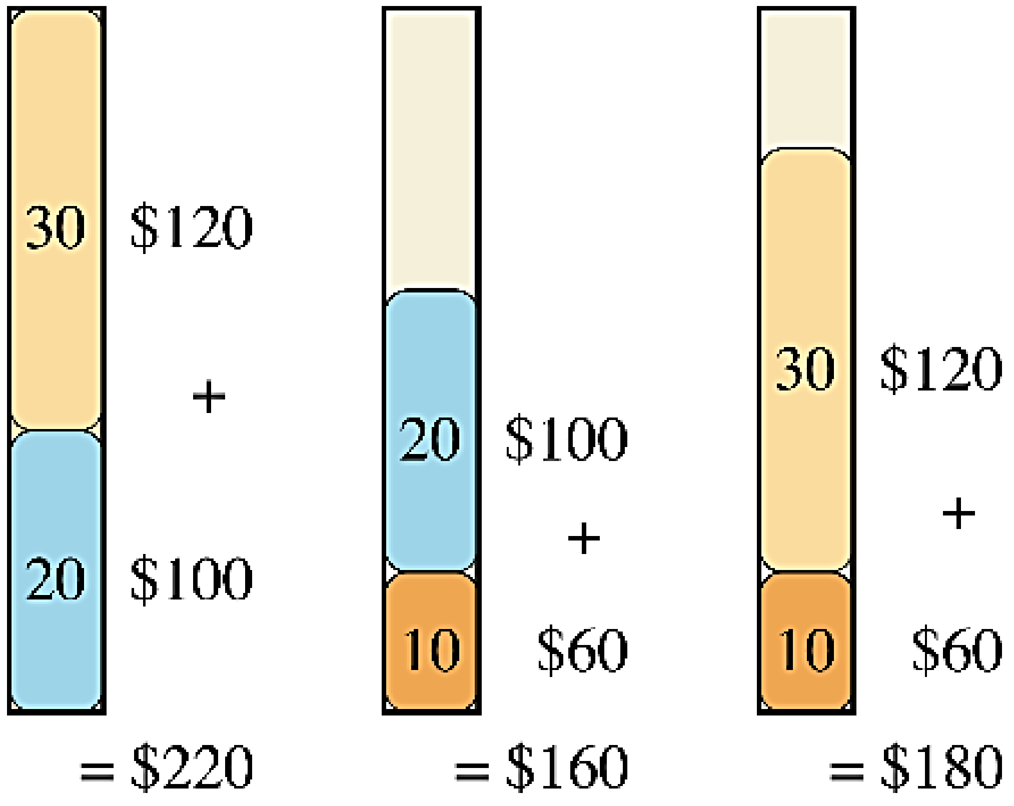
```
MULTIPLY (i, j)
if (i = j)
    return A[i]
else k ← s[i, j]
    X ← MULTIPLY(i, k)
    Y ← MULTIPLY(k+1, j)
    return X·Y
```

0/1 Knapsack Problem

- A thief goes into a jewelry store to steal jewelry items.
- He has a knapsack (a bag) that he would like to fill up.
- The bag has a limit on the total weight of the objects placed in it.
- If the total weight exceeds the limit, the bag will tear open.

- The value of the jewelry items varies for cheap to expensive.
- The thief’s goal is to put items in the bag such that the value of the items is maximized, and the weight of the items does not exceed the weight limit of the bag.
- Another limitation is that an item can either be put in the bag or not - fractional items are not allowed.
- The problem is: what jewelry should the thief choose that satisfy the constraints?


item 1
item 2
item 3
10
20
30
50
$60
$100
$120
knapsack
(a)


30 $120
+
20 $100
= $220
20 $100
+
10 $60
= $160
30 $120
+
10 $60
= $180
(b)

- Formally, the problem can be stated as follows:
    - Given a knapsack with maximum capacity W, and a set S consisting of n items.
    - Each item i has some weight w_i and value v_i (all w_i , v_i and W are integer values).
    - How to pack the knapsack to achieve maximum total value of packed items?

- For example, consider the following scenario:

Knapsack can hold $W = 20$

Item i	Weight w_i	Value v_i
1	2	3
2	3	4
3	4	5
4	5	8
5	9	10

- The knapsack problem belongs to the domain of optimization problems.
- Mathematically, the problem is:

$$maximize \sum_{i \in T} v_i$$

$$subject\ to \sum_{i \in T} w_i \leq W$$

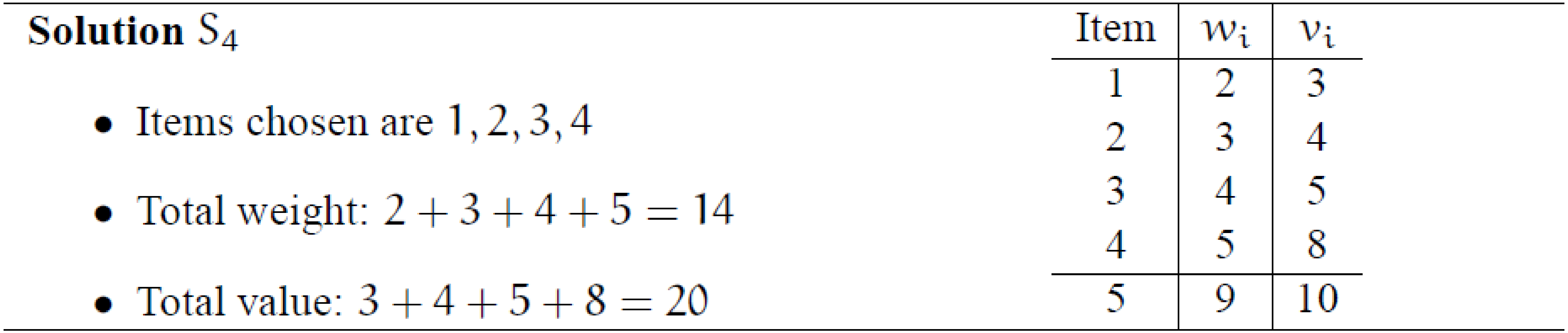
- The problem is called a “0-1” problem, because each item must be entirely accepted or rejected.
- How do we solve the problem.
- We could try the brute-force solution:
  - Since there are n items, there are 2^n possible combinations of the items (an item either chosen or not).
  - We go through all combinations and find the one with the most total value and with total weight less or equal to W

- Clearly, the running time of such a brute-force algorithm will be $O(2^n)$.
- Can we do better?
- The answer is “yes”, with an algorithm based on dynamic programming.

- Let us recap the steps in the dynamic programming strategy:
    - **Simple Subproblems:** We should be able to break the original problem to smaller subproblems that have the same structure.
    - **Principle of Optimality:** Recursively define the value of an optimal solution. Express the solution of the original problem in terms of optimal solutions for smaller problems.
    - **Bottom-up computation:** Compute the value of an optimal solution in a bottom-up fashion by using a table structure.
    - **Construction of optimal solution:** Construct an optimal solution from computed information.

- Let us try this: If items are labelled $1, 2, \ldots, n$, then a subproblem would be to find an optimal solution for S_k = items labelled $1, 2, \ldots, k$.
- This is a valid subproblem definition.
- The question is: can we describe the final solution S_n in terms of subproblems S_k?
- Unfortunately, we cannot do that.

- Consider the optimal solution if we can choose items 1 through 4 only.

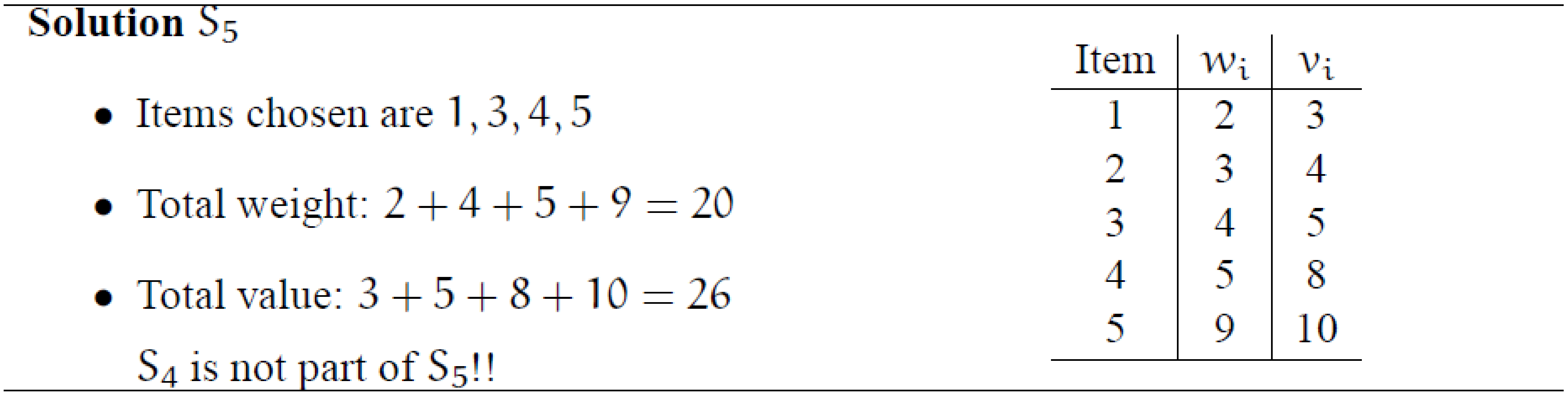


Solution S_4

- Items chosen are $1, 2, 3, 4$
- Total weight: $2 + 3 + 4 + 5 = 14$
- Total value: $3 + 4 + 5 + 8 = 20$

Item	w_i	v_i
1	2	3
2	3	4
3	4	5
4	5	8
5	9	10

- Now consider the optimal solution when items 1 through 5 are available.



Solution S_5

- Items chosen are $1, 3, 4, 5$
- Total weight: $2 + 4 + 5 + 9 = 20$
- Total value: $3 + 5 + 8 + 10 = 26$

S_4 is not part of S_5!!

Item	w_i	v_i
1	2	3
2	3	4
3	4	5
4	5	8
5	9	10

- The solution for S_4 is not part of the solution for S_5.
- So, our definition of a subproblem is flawed and we need another one.

0/1 Knapsack Problem: Dynamic Programming Approach

- For each $i \leq n$ and each $w \leq W$, solve the knapsack problem for the first i objects when the capacity is w.
- Why will this work?
- Because solutions to larger subproblems can be built up easily from solutions to smaller ones.

- We construct a matrix $V[0 \ldots n, 0 \ldots W]$.
- For $1 \leq i \leq n$, and $0 \leq j \leq W$, $V[i, j]$ will store the maximum value of any set of objects $\{1, 2, \ldots\ , i\}$ that can fit into a knapsack of weight j.
- $V[n, W]$ will contain the maximum value of all n objects that can fit into the entire knapsack of weight W.