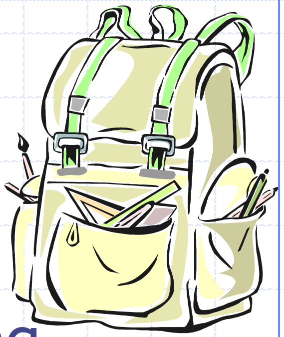
Dynamic Programming:

0-1 Knapsack

The 0/1 Knapsack Problem

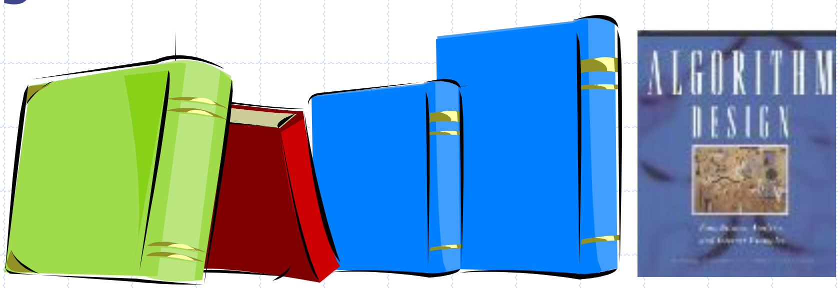


- Given: A set S of n items, with each item i having
  - w_i - a positive weight
  - b_i - a positive benefit
- Goal: Choose items with maximum total benefit but with weight at most W.
- If we are **not** allowed to take fractional amounts, then this is the **0/1 knapsack problem**.
  - In this case, we let T denote the set of items we take
  - Objective: maximize $$\sum_{i \in T} b_i$$
  - Constraint: $$\sum_{i \in T} w_i \leq W$$

Example

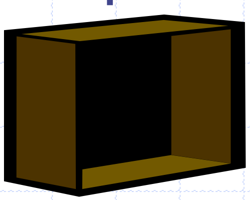
- Given: A set S of n items, with each item i having
  - b_i - a positive "benefit"
  - w_i - a positive "weight"
- Goal: Choose items with maximum total benefit but with weight at most W.

Items:



Items:	1	2	3	4	5
Weight:	4 in	2 in	2 in	6 in	2 in
Benefit:	\$20	\$3	\$6	\$25	\$80

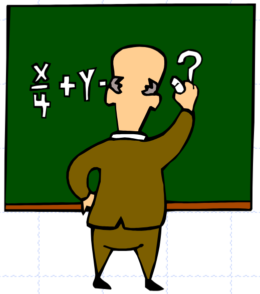
"knapsack"



box of width 9 in

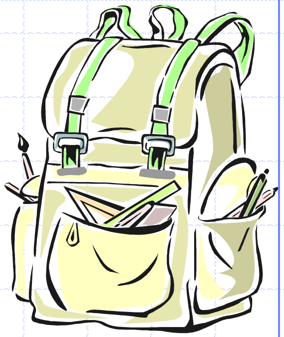
Solution:
- item 5 (\$80, 2 in)
- item 3 (\$6, 2in)
- item 1 (\$20, 4in)

The General Dynamic Programming Technique



- Applies to a problem that at first seems to require a lot of time (possibly exponential), provided we have:
  - **Simple subproblems:** the subproblems can be defined in terms of a few variables, such as j, k, l, m, and so on.
  - **Subproblem optimality:** the global optimum value can be defined in terms of optimal subproblems
  - **Subproblem overlap:** the subproblems are not independent, but instead they overlap (hence, should be constructed bottom-up).

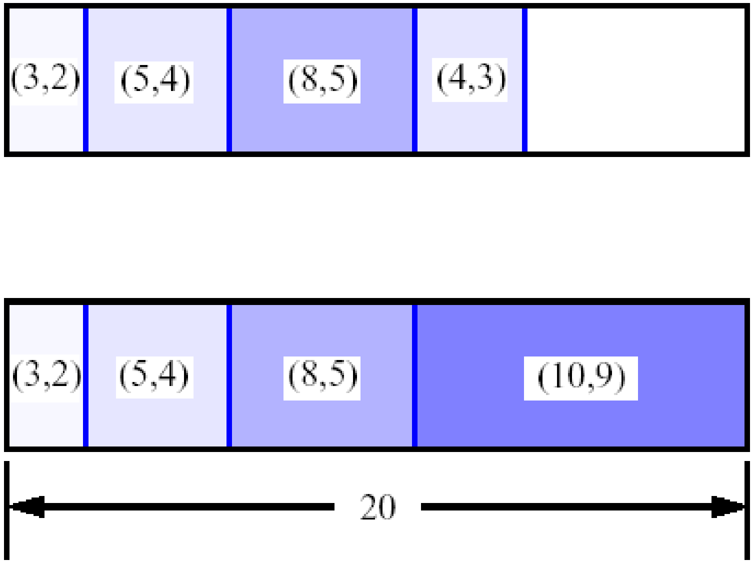
A 0/1 Knapsack Algorithm, First Attempt



- S_k: Set of items numbered 1 to k.
- Define B[k] = best selection from S_k.
- Problem: does not have subproblem optimality:
  - Consider set S={(3,2),(5,4),(8,5),(4,3),(10,9)} of (benefit, weight) pairs and total weight W = 20

Best for S_4:

Best for S_5:



0-1 Knapsack Better 2nd Attempt

- Items cannot be subdivided – take the entire item or do not take it
- At every iteration of the algorithm, for each item i make a binary choice to include the item or not
- If the *i-th* item is included, then
  - the benefit value increases
  - the availability capacity in the knapsack is reduced by the weight of the *i-th* item

- $M[i, w]$ = maximum benefit that can be obtained by selecting (some of) the items from 1, 2,…, i with the knapsack capacity of $w \leq W$, where w is the residual capacity in the knapsack and W is the maximum weight, or threshold, that the knapsack can hold
- $b(i)$ is the benefit (or price in our example) of including item i

0-1 Knapsack Algorithm from Lecture

Algo: 0-1 Knapsack (S, W)
Input: Set S of n $\geq$ 0 items, each with benefit b(i) $\geq$ 0 and weight w(i) $\geq$ 0, and knapsack capacity W. Assume w(i), n and W are integers and 0$\leq$ i$\leq$ n with i representing the i-*th* item
Output: Maximum benefit that can be obtained from S with a knapsack of capacity W

```
M[n, W] ← new n x W matrix
for i ← to n do
   for w ←0 to W // all possible residual capacities
      M[i, w] ← 0  // base case and initialization

for i ← 1 to n do
   for w ← 1 to W //assume weights are integers
      if w(i) ≤ w then
         M[i, w] ← b(i) + M[i-1, w - w(i)]
      M[i, w] ← max{ M[i, w], M[i-1, w] }
return M[n, W]
```