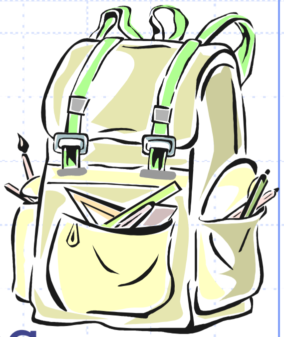
Presentation for use with the textbook, Algorithm Design and Applications, by M. T. Goodrich and R. Tamassia, Wiley, 2015

# Dynamic Programming: 0/1 Knapsack

Civil War Knapsack. U.S. government image. Vicksburg National Military Park. Public domain.

# The 0/1 Knapsack Problem

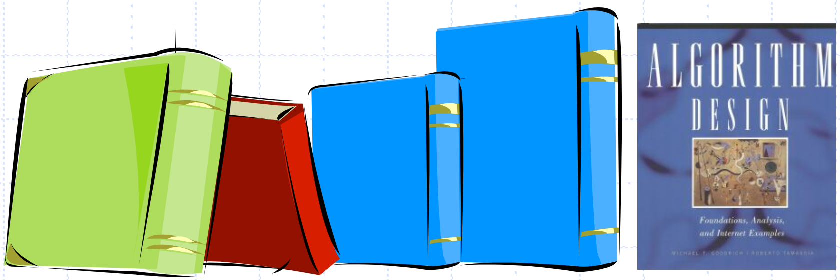


- Given: A set S of n items, with each item i having
  - $w_i$ - a positive weight
  - $b_i$ - a positive benefit
- Goal: Choose items with maximum total benefit but with weight at most W.
- If we are **not** allowed to take fractional amounts, then this is the **0/1 knapsack problem**.
  - In this case,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

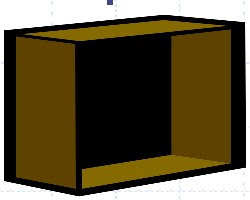
- Given: A set S of n items, with each item i having
  - $b_i$ - a positive “benefit”
  - $w_i$ - a positive “weight”
- Goal: Choose items with maximum total benefit but with weight at most W.

Items:




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

“knapsack”

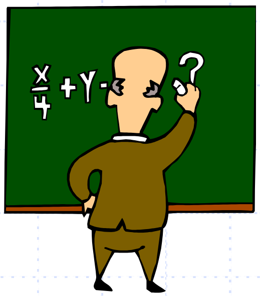


box of width 9 in

Solution:

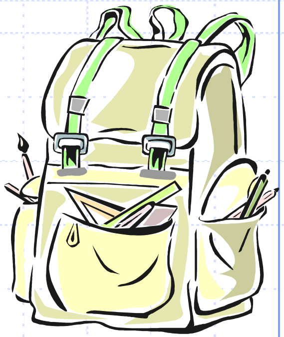
- item 5 (\$80, 2 in)
- item 3 (\$6, 2in)
- item 1 (\$20, 4in)

# 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).

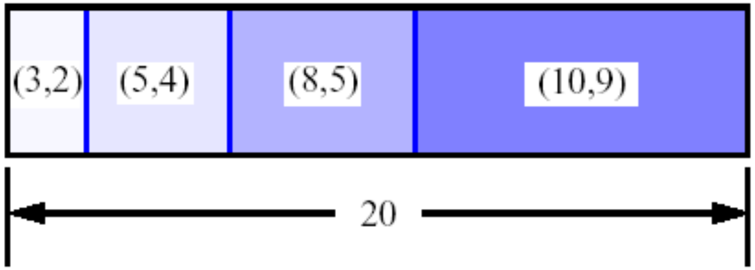
# A 0/1 Knapsack Algorithm, First Attempt



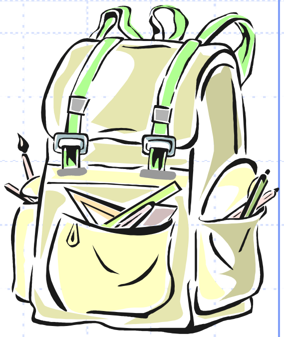
- $S_k$: Set of items numbered 1 to k.
- Define B[k] = best selection from $S_k$.
- Problem: does not have subproblem optimality:
  - Consider set S={(3,2),(5,4),(8,5),(4,3),(10,9)} of (benefit, weight) pairs and total weight W = 20

Best for $S_4$:


Best for $S_5$:



# A 0/1 Knapsack Algorithm, Second (Better) Attempt



- $S_k$: Set of items numbered 1 to k.
- Define B[k,w] to be the best selection from $S_k$ with weight at most w
- Good news: this does have subproblem optimality.

$$B[k,w] = \begin{cases} B[k-1,w] & \text{if } w_k > w \\ \max\{B[k-1,w], B[k-1,w-w_k]+b_k\} & \text{else} \end{cases}$$

- I.e., the best subset of $S_k$ with weight at most w is either
  - the best subset of $S_{k-1}$ with weight at most w or
  - the best subset of $S_{k-1}$ with weight at most $w-w_k$ plus item k

# 0/1 Knapsack Algorithm

$$B[k,w] = \begin{cases} B[k-1,w] & \text{if } w_k > w \\ \max\{B[k-1,w],\ B[k-1,w-w_k]+b_k\} & \text{else} \end{cases}$$

- Recall the definition of B[k,w]
- Since B[k,w] is defined in terms of B[k−1,*], we can use two arrays of instead of a matrix
- Running time: O(nW).
- Not a polynomial-time algorithm since W may be large
- This is a pseudo-polynomial time algorithm

**Algorithm** *01Knapsack*(*S*, *W*):
    **Input:** set $S$ of $n$ items with benefit $b_i$ and weight $w_i$; maximum weight $W$
    **Output:** benefit of best subset of $S$ with weight at most $W$

```
let A and B be arrays of length W + 1
for w ← 0 to W do
    B[w] ← 0
for k ← 1 to n do
    copy array B into array A
    for w ← w_k to W do
        if A[w−w_k] + b_k > A[w] then
            B[w] ← A[w−w_k] + b_k
return B[W]
```