# CMP461: Algorithms


## Lecture 09: Greedy Algorithms

Mohamed Alaa El-Dien Aly
Computer Engineering Department
Cairo University
Fall 2013

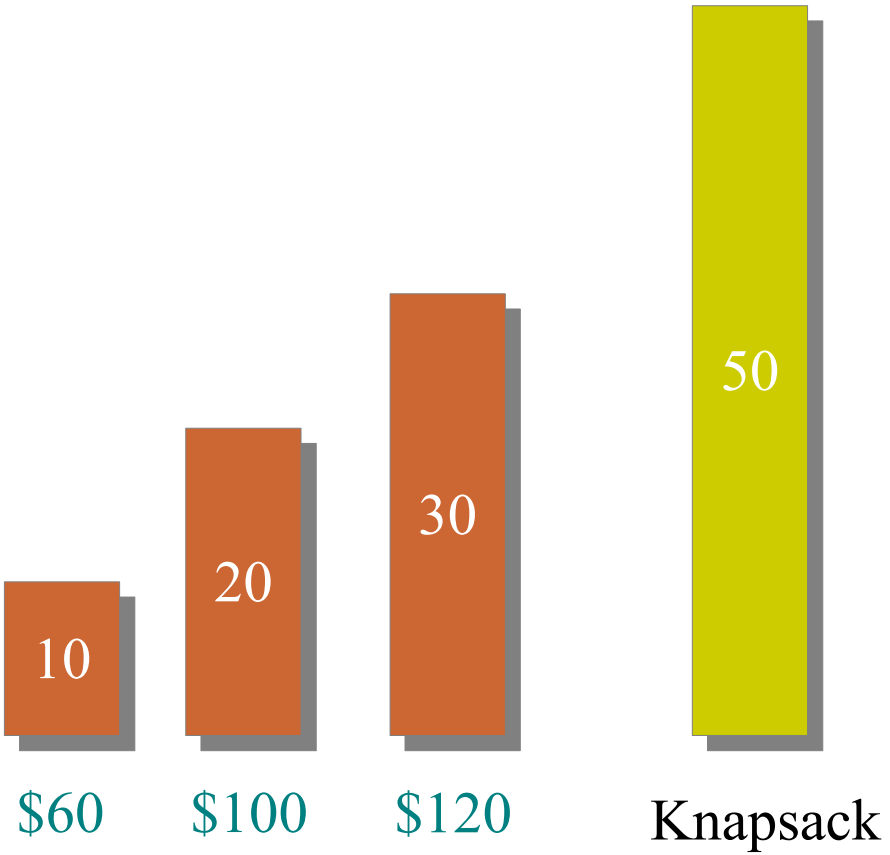
# Agenda

- Knapsack Problem
  - Dynamic Programming Algorithm
  - Greedy Algorithm
- Greedy Algorithm Properties

**Acknowledgment**
A lot of slides adapted from the slides of Erik Demaine, Charles Leiserson, and David Luebke.

# Knapsack Problem

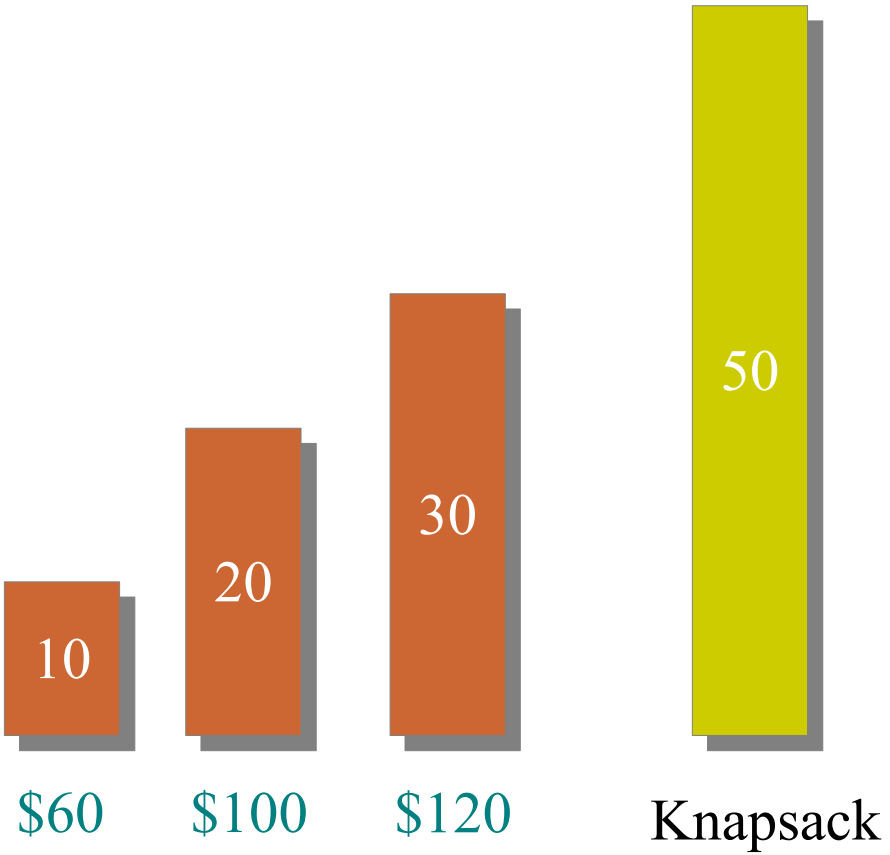
A thief has a knapsack of limited capacity *W*, and wants to pack it with items that have as much value as possible such that the total weight does not exceed the knapsack's capacity.



# Knapsack Problem

Two types:

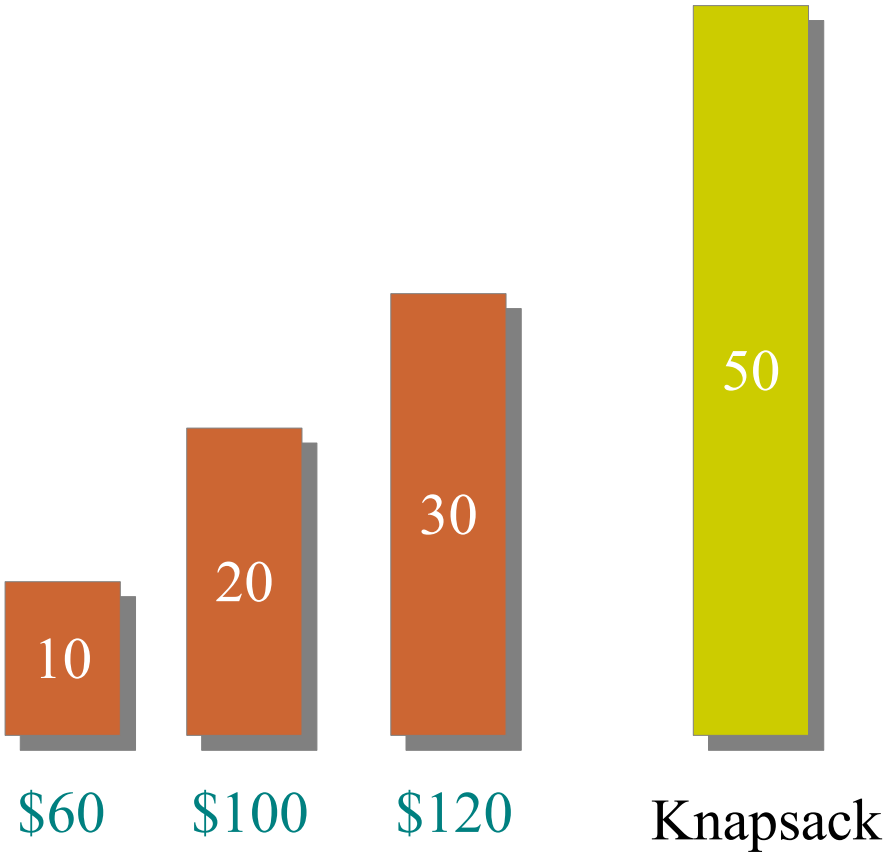
1. 0-1 Knapsack Problem: can only pack *whole* items
2. Fractional Knapsack: can pack *fractions* of items



# 0-1 Knapsack Problem

Given a knapsack with capacity $W$ and a set $S$ of $n$ items each with weight $w_i$ and value $b_i$, all integers.

Problem: Find the subset of items to put in the knapsack to achieve maximum value.

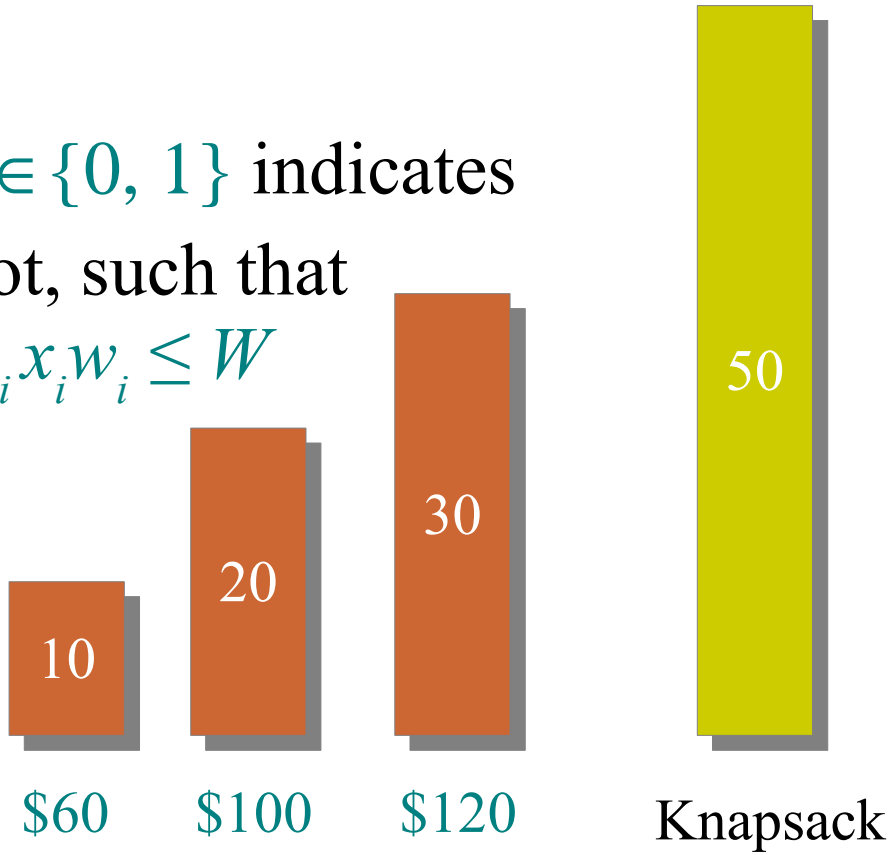


# 0-1 Knapsack Problem

Given a knapsack with capacity $W$ and a set $S$ of $n$ items each with weight $w_i$ and value $b_i$, all integers.

Problem: Find the subset of items to put in the knapsack to achieve maximum value.

Find $x = [x_1, x_2, \ldots x_n]$ where $x_i \in \{0, 1\}$ indicates whether item $i$ is included or not, such that $x = \text{argmax}_x \sum_i x_i b_i$ subject to $\sum_i x_i w_i \leq W$

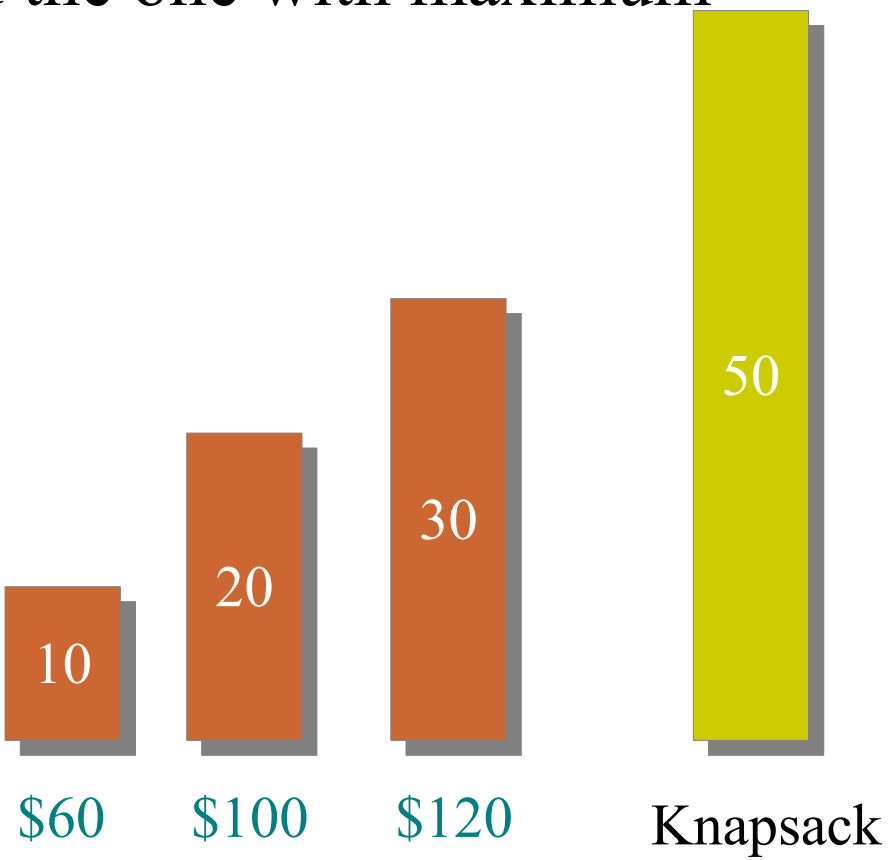


# Brute Force Approach

Since we have $n$ items, how many possible subsets?
We have $2^n$ possible subsets

Algorithm:
Enumerate all subsets, and find the one with maximum value that fits in the knapsack

Takes time $O(2^n)$



# Dynamic Programming Approach

Remember, for dynamic programming we need:

1. Optimal substructure

Solution to the problem contains optimal solutions to subproblems

2. Overlapping subproblems

Few distinct subproblems that need to be solved over and over again

# Subproblem 1

Define $S_k$ to be the subset of items labeled 1 to $k$.

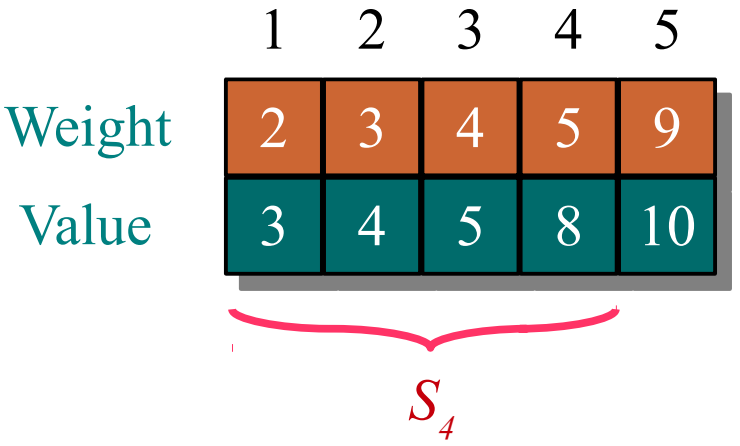
The subproblem would be to find the subset of $S_k$ with *maximum* value $B[k]$ that fits in the knapsack.

The solution to the main problem is the solution of $S_n$.

Question:
Can we describe the solution to the main problem in terms of solutions to subproblems (i.e. optimal substructure)?

Answer: No!

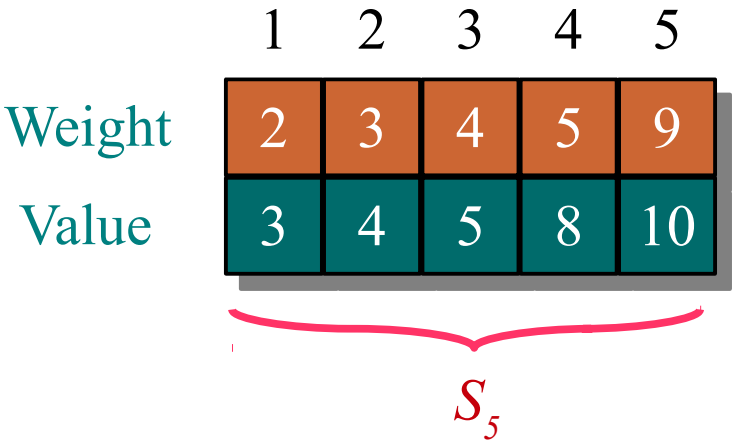
# Subproblem 1





Optimal solution for $S_4$ contains items $\{1, 2, 3, 4\}$ with *weight* 14 and *value* $B[4] = 20$

# Subproblem 1





Optimal solution for $S_4$ contains items $\{1, 2, 3, 4\}$
with *weight* 14 and *value* $B[4] = 20$

Optimal solution for $S_5$ contains items $\{1, 3, 4, 5\}$
with *weight* 20 and *value* $B[5] = 26$

Solution to $S_5$ does not contain solution to $S_4$ !!
Need another definition for the subproblem.

# Subproblem 2

Add another parameters $w$ that defines the exact weight of the chosen subset.

Define $S_k$ to be the subset of items labeled 1 to $k$.

The subproblem would be to find the subset of $S_k$ with *maximum* value that fits into a knapsack with weight $w$.

Define $B[k, w]$ to be the maximum value attained by a subset in $S_k$ with weight $\leq w$

The solution to the main problem is $B[n, W]$.

# Recursive Solution

Assume you solved the problem for $B[k-1, w]$ and want check if item $k$ should be added to the optimal solution:

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

In other words, the optimal solution to $B[k, w]$ is either:

1. $B[k-1, w]$ if item $k$ does not fit in the knapsack with capacity $w$ and then can not be part of the solution

2. The best of $B[k-1, w]$ (without item $k$) or the value obtained by adding item $k$ to $B[k-1, w-w_k]$

Proof: exercise!

# Recursive Solution

Top-down Solution

```
Knapsack(B, k, w)
  if w[k] > w
    B[k, w] = Knapsack(B, k–1, w)
  else
    B[k, w] = max{Knapsack(B, k–1, w),
                  Knapsack(B, k–1, w–w[k]) + b[k]}
```

# Dynamic Programming Solution

Bottom-up Solution

```
Knapsack(B, n, W)
  for w = 0 to W
    B[0, w] = 0
  for k = 1 to n
    B[k, 0] = 0
    for w = 0 to W
      if w[k] > w
        B[k, w] = B[k–1, w]
      else
        B[k, w] = max(B[k–1, w], B[k–1, w – w[k]] + b[k])
```

# Dynamic Programming Solution

Bottom-up Solution

```
Knapsack(B, n, W)
  for w = 0 to W
    B[0, w] = 0
  for k = 1 to n
    B[k, 0] = 0
    for w = 0 to W
      if w[k] > w
        B[k, w] = B[k–1, w]
      else
        B[k, w] = max(B[k–1, w], B[k–1, w-w[k]] + b[k])
```

$n$ $W$

Running time? $\Theta(n\ W)$

# 0-1 Knapsack Summary

Brute-force solution: takes time $O(2^n)$

Dynamic Programming solution takes time $O(n\ W)$

What about the Fractional Knapsack problem?

Will solve it using Greedy Algorithms, a design strategy related to Dynamic Programming, that makes locally optimal (*greedy*) choices at each step hoping to arrive at a global optimum.

It works for some problems i.e. finds the global optimum, and not others!

# Greedy Algorithms

Related to Dynamic Programming, depends on two properties:

1. Optimal substructure

The optimal solution to the main problem contains optimal solutions to subproblems (like DP).

2. Greedy choice

At each step, we only need to solve one subproblem and make one choice, the *greedy* choice i.e. the best choice at the moment (unlike DP).

Finding the locally optimal solutions will get us to the global optimum.

# Greedy Solution for 0-1 Knapsack

Intuition: We want to pack the knapsack with as much value as possible. So what would be our *greedy* choice?

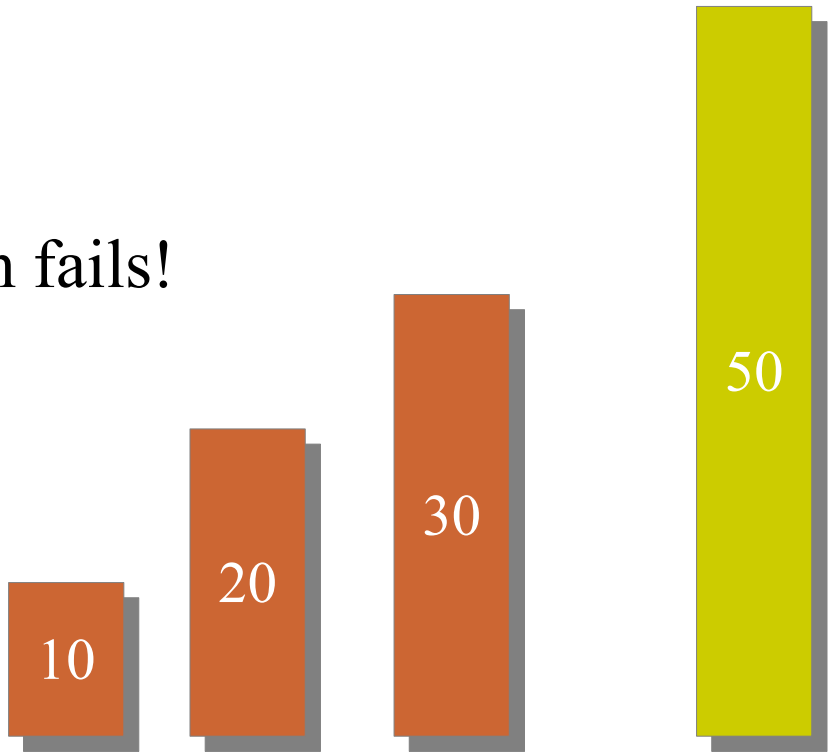
Choose the item with *maximum* value per Kg.

Does it find the global optimum?

No!

# Greedy Solution for 0-1 Knapsack

Greedy Algorithm fails!




Item 1 has the best value/Kg.

Greedy solution (items 1 & 2) = $160 < the optimum value of $220!

# Greedy Solution for Fractional Knapsack

Define $b[k]$ to be the value per Kg for item $k$. The problem is to find fractional weights $x[k]$ for each item to pack the knapsack such that the total weight does not exceed $W$.

Sort the items by their $b[k]$ in *increasing* order. Define $B[k,w]$ as the *maximum* value obtained from the first $k$ items that can fit a knapsack of weight $w$.

We get:

Amount we are taking from item $k$

Solution of *the* subproblem

$$B[k, w] = x[k]\, b[k] + B[k-1, w-x[k]]$$

$$x[k] = \min(w[k], w)$$

*Greedy* choice = *Maximum* amount we can take from item $k$

# Greedy Solution for Fractional Knapsack

$$B[k, w] = x[k]\, b[k] + B[k-1, w - x[k]]$$
$$x[k] = \min(w[k], w)$$

Optimal Substructure

If $B[k, w]$ is the optimal solution for $k$ and $w$, then $B[k-1, w-x[k]]$ is the optimal solution for $k-1$ and $w-x[k]$.

Proof: (Cut and paste argument)

Assume that $B[k-1, w-x[k]]$ is not optimal.
Then there exists another choice of $x[i]$ for $i = 1 \ldots k-1$ that provides bigger value. Therefore we can substitute these choices and add to them $x[k]$ to find a value better than $B[k,w]$, which is a contradiction.

# Greedy Solution for Fractional Knapsack

$$B[k, w] = x[k]\, b[k] + B[k-1, w - x[k]]$$
$$x[k] = \min(w[k], w)$$

Greedy Choice Property

If $B[k, w]$ is the optimal solution for $k$ and $w$, then the greedy choice $x[k] = \min(w[k], w)$ where $b[k]$ is largest value per Kg is part of the the optimal solution.

Proof: (Cut and paste argument)

Assume that $x[k] < \min(w[k], w)$. Since $b[k] > b[\mathrm{i}]\ \forall\ i = 1\ ..\ k-1$, we can reduce $x[k-1]$ and increase $x[k]$ by the same amount to get a larger $B[k, w]$, which is a contradiction.

# Recursive Solution for Fractional Knapsack

Recursive Solution

```
FKnapsack(k, w)
    x[k] = min(w[k], w)
    B[k, w] = x[k] b[k] + FKnapsack(k–1, w – x[k])
```

Running time? $\Theta(n \lg n)$

Time to sort + recursive calls.

# Recursive Solution for Fractional Knapsack

Iterative Solution

```
FKnapsack(n, W)
    B = 0
    for k = n downto 1
        x[k] = min(w[k], W)
        B = B + x[k] b[k]
        W = W – x[k]
```

# Fractional Knapsack Summary

Can solve with greedy algorithms in $\Theta(n \lg n)$

Two properties of Greedy Algorithms:

1. Optimal Substructure

2. Greedy Choice (one subproblem)

Next: Minimal Spanning Trees

# Recap

- Knapsack Problem
  - Dynamic Programming Algorithm
  - Greedy Algorithm
- Greedy Algorithm Properties
- Next:
  - Minimal Spanning Trees