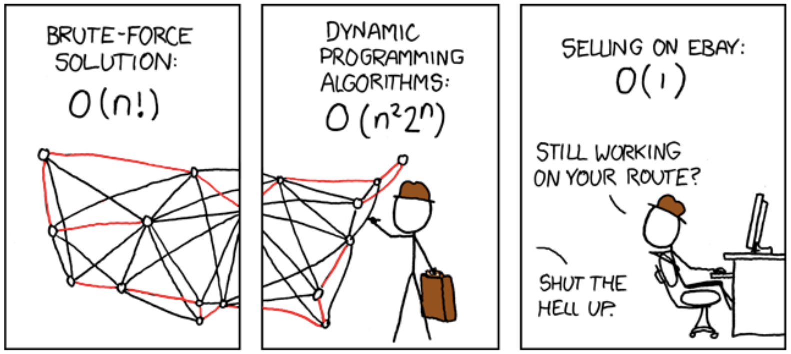
Solving Problems with Optimal Substructure



Xkcd.org

Slides adapted from Ran Libeskind-Hadas,David Kauchak,CS 460 JHU

Problems with Optimal Substructure

- Combining optimal solutions to subproblems leads to globally optimal solution
- Used for optimization problems:
  - Find *a* solution with *the* optimal value
  - Minimization and maximization

Dynamic Programming: The Goal

- Solve each subproblem once
- Save solution in a table and refer back any time we revisit the subproblem
- "Store, don't recompute" → Time-memory trade-off
- Two basic approaches: top-down with memoization and bottom up

Identifying a dynamic programming problem

The solution can be defined with respect to solutions to subproblems

The subproblems created are *overlapping*, that is **we see the same subproblems repeated**

Creating a dynamic programming solution

1. Characterize the structure of an optimal solution
2. Recursively define the value of an optimal solution
3. Compute the value of an optimal solution, typically in bottom-up fashion
4. Construct an optimal solution from computed information

```
FIBONACCI-DP(n)
  fib[1] ← 1
  fib[2] ← 1
  for i ← 3 to n
      fib[i] ← fib[i − 1] + fib[i − 2]
  return fib[n]
```

Creating a dynamic programming solution

Step 1: Identify a solution to the problem with respect to **smaller** subproblems (pretend like you have a solver, but it only works on smaller problems):

- F(n) = F(n-1) + F(n-2)

Step 2: **bottom up** - start with solutions to the smallest problems and build solutions to the larger problems

```
FIBONACCI-DP(n)
  fib[1] ← 1
  fib[2] ← 1
  for i ← 3 to n
      fib[i] ← fib[i − 1] + fib[i − 2]
  return fib[n]
```

use an array to store solutions to subproblems

Important Questions to ask about the DP Table:

- **Meaning?**
  - What do the cells mean?
- **Want?**
  - What cell do you want?
- **Easy?**
  - What cells can you fill out (easily)?
- **Rule?**
  - What rule helps fill out other cells?

The DP table should include the possible inputs to the recursive call

```
FIBONACCI-DP(n)
  fib[1] ← 1
  fib[2] ← 1
  for i ← 3 to n
      fib[i] ← fib[i − 1] + fib[i − 2]
  return fib[n]
```

Elements of a DP (revisited)

- Optimal substructure
  - A solution to a problem consists of making a choice/computation that will lead to an optimal solution
  - Given this choice/computation, determine which subproblems arise and how to characterize the resulting space of subproblems.
  - Solutions to the sub-problems used within the optimal solution must themselves be optimal. Otherwise, we'd see "cut-and-paste" error:
    - Suppose that one of the subproblem solutions is not optimal
    - Cut it out
    - Paste in an optimal solution
    - Get a better solution to the original problem. Contradicts the optimality of problem solutions
- Overlapping subproblems

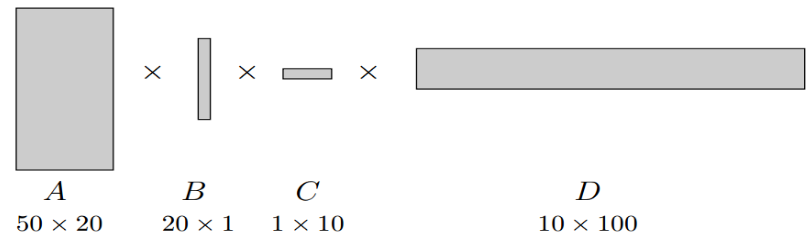
How does this differ from greedy?

Dynamic Programming

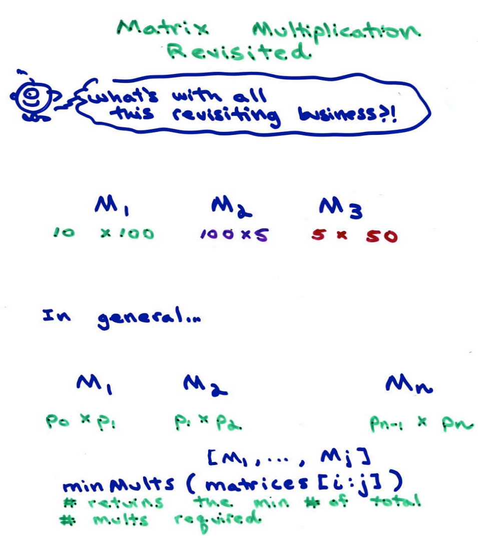
- Not a specific algorithm, but a technique (like divide-and-conquer).
- Developed back in the day when "programming" meant "tabular method" (like linear programming). Doesn't really refer to computer programming.
- Used for optimization problems:
  - Find *a* solution with *the* optimal value
  - Minimization and maximization

Order Matters

http://www.cs.berkeley.edu/~vazirani/algorithms/chap6.pdf

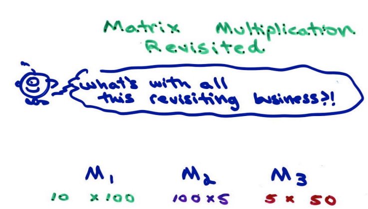


Parenthesization	Cost computation	Cost
$A \times ((B \times C) \times D)$	$20 \cdot 1 \cdot 10 + 20 \cdot 10 \cdot 100 + 50 \cdot 20 \cdot 100$	120, 200
$(A \times (B \times C)) \times D$	$20 \cdot 1 \cdot 10 + 50 \cdot 20 \cdot 10 + 50 \cdot 10 \cdot 100$	60, 200
$(A \times B) \times (C \times D)$	$50 \cdot 20 \cdot 1 + 1 \cdot 10 \cdot 100 + 50 \cdot 1 \cdot 100$	7, 000



Ran++ Conventions

```
matrices[1:3]
matrices[1]:  p[0] x p[1]
matrices[2]:  p[1] x p[2]
matrices[3]:  p[2] x p[3]
```




```
matrices[1:3]
matrices[1]:  p[0] x p[1]
matrices[2]:  p[1] x p[2]
matrices[3]:  p[2] x p[3]
```

```
minMults(matrices[i:j])
  if i == j: return 0 # one matrix, no mults!
  else:
     best = Infinity  # best cost so far
     for k from i to j-1:  # where shall we split?
       left = minMults[i:k]
       right = minMults[k+1:j]
       lastMult = p[i-1]*p[k]*p[j]
       total = left + right + lastMult
       if total < best: best = total
  return best
```


```
matrices[1:3]
matrices[1]:  p[0] x p[1]
matrices[2]:  p[1] x p[2]
matrices[3]:  p[2] x p[3]
```

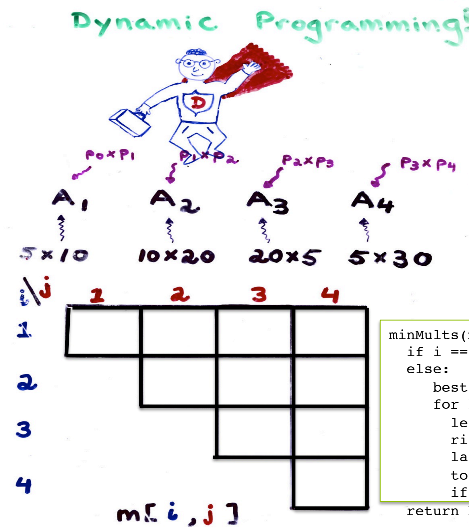
```
minMults(matrices[i:j])
  if i == j: return 0 # one matrix, no mult
  else:
     best = Infinity  # best cost so far
     for k from i to j-1:  # where shall we
       left = minMults[i:k]
       right = minMults[k+1:j]
       lastMult = p[i-1]*p[k]*p[j]
       total = left + right + lastMult
       if total < best: best = total
  return best
```

Meaning?
- What do the cells mean?

Want?
- What cell do you want?

Easy?
- What cells can you fill out (easily)?

Rule?
- What rule helps fill out other cells?

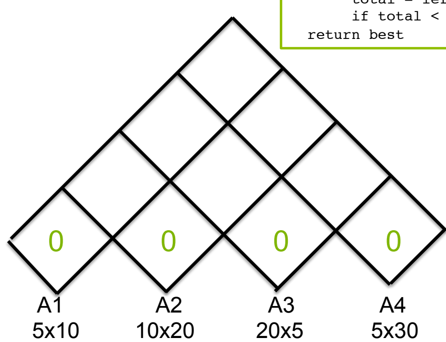


Fill this in in your notes!

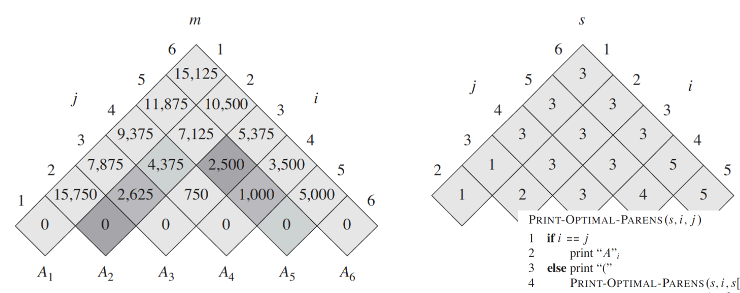

```
minMults(matrices[i:j])
  if i == j: return 0 # one matrix, no mults!
  else:
     best = Infinity  # best cost so far
     for k from i to j-1:  # where shall we split?
       left = minMults[i:k]
       right = minMults[k+1:j]
       lastMult = p[i-1]*p[k]*p[j]
       total = left + right + lastMult
       if total < best: best = total
  return best
```



```
minMults(matrices[i:j])
  if i == j: return 0 # one matrix, no mults!
  else:
     best = Infinity  # best cost so far
     for k from i to j-1:  # where shall we split?
       left = minMults[i:k]
       right = minMults[k+1:j]
       lastMult = p[i-1]*p[k]*p[j]
       total = left + right + lastMult
       if total < best: best = total
  return best
```

Another example: (CRLS)




```
PRINT-OPTIMAL-PARENS(s, i, j)
  if i == j
      print "A"_i
  else print "("
      PRINT-OPTIMAL-PARENS(s, i, s[i, j])
      PRINT-OPTIMAL-PARENS(s, s[i, j] + 1, j)
      print ")"
```

Figure 15.5 The m and s tables computed by MATRIX-CHAIN-ORDER for $n = 6$ and the following matrix dimensions:

matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimension	30×35	35×15	15×5	5×10	10×20	20×25

Longest increasing subsequence

Given a sequence of numbers $X = x_1, x_2, \ldots, x_n$ find the longest increasing *subsequence*

$(i_1, i_2, \ldots, i_k)$, that is a subsequence where numbers in the sequence increase.

5 2 8 6 3 6 9 7

Step 1: Define the problem with respect to subproblems

5 2 8 6 3 6 9 7

Two options:
Either 5 is in the
LIS or it's not

Step 1: Define the problem with respect to subproblems

5 2 8 6 3 6 9 7

include 5

5 + LIS(8 6 3 6 9 7)

What is this function exactly?

longest increasing sequence of the numbers

longest increasing sequence of the numbers starting with 8

Step 1: Define the problem with respect to subproblems

5 2 8 6 3 6 9 7

include 5

5 + LIS'(8 6 3 6 9 7)

longest increasing sequence of the numbers starting with 8

Do we need to consider anything else for subsequences starting at 5?

Step 1: Define the problem with respect to subproblems

5 2 8 6 3 6 9 7

include 5

5 + LIS'(8 6 3 6 9 7)

5 + LIS'(6 3 6 9 7)

5 + LIS'(6 9 7)

5 + LIS'(9 7)

5 + LIS'(7)

Step 1: Define the problem with respect to subproblems

5 2 8 6 3 6 9 7

don't include 5

LIS(2 8 6 3 6 9 7)

Anything else?

Technically, this is fine, but now we have LIS and LIS' to worry about.

Can we rewrite LIS in terms of LIS'?

Step 1: Define the problem with respect to subproblems

$$LIS(X) = \max_i \{LIS'(i)\}$$

Longest increasing sequence for X is the longest increasing sequence starting at any element

$$LIS'(i) = \max_{i:i>1 \text{ and } x_i > x_1} \{1 + LIS'(X_{i\ldots n})\}$$

Longest increasing sequence starting at i

Step 2: build the solution from the bottom up

$$LIS'(i) = \max_{i:i>1 \text{ and } x_i > x_1} \{1 + LIS'(X_{i\ldots n})\}$$

LIS' :	3	4	2	2	3	2	1	1
	5	2	8	6	3	6	9	7

$$LIS(X) = \max_i \{LIS'(i)\}$$

Step 2: build the solution from the bottom up

$$LIS'(i) = \max_{i:i>1 \text{ and } x_i > x_1} \{1 + LIS'(X_{i \ldots n})\}$$

What does my data structure for storing answers look like?

Step 2: build the solution from the bottom up

$$LIS'(i) = \max_{i:i>1 \text{ and } x_i > x_1} \{1 + LIS'(X_{i \ldots n})\}$$

1-D array: only one thing changes for recursive calls, i

Step 2: build the solution from the bottom up

```
LIS(X)
n ← LENGTH(X)
create array lis with n entries
for i ← n to 1
        max ← 1
        for j ← i + 1 to n
                if X[j] > X[i]
                        if 1 + lis[j] > max
                                max ← 1 + lis[j]
        lis[i] ← max
max ← 0
for i ← 1 to n
        if lis[i] > max
                max ← lis[i]
return max
```

Another solution

Can we use LCS to solve this problem?

5 2 8 6 3 6 9 7

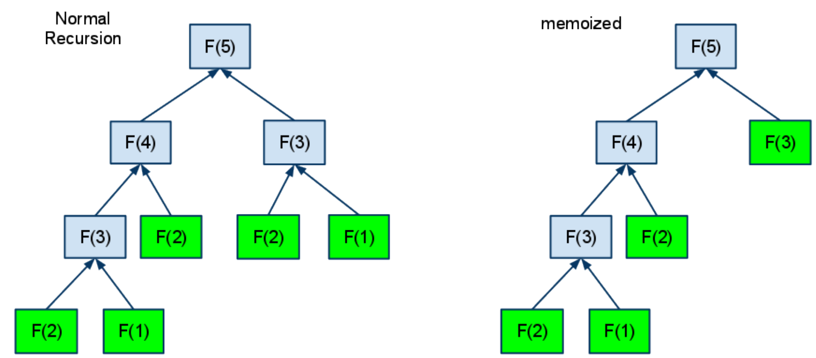
2 3 5 6 6 7 8 9

LCS

Top-down Alternative: Memoization

- Memoizing is remembering what we have computed previously.
- Solve recursively (top-down)
  - "Store, don't recompute"
  - Make a table indexed by subproblem
  - When solving subproblem (top-down):
    - Look-up in table
    - If answer is there, use it
    - Else, compute answer and store it.
- Bottom-up DP goes a step further: first determine the order in which the table would be accessed, and fill it in that way

The top-down approach



http://www.agillo.net/getting-groovy-with-fibonacci/

Memoization

Sometimes it can be a challenge to write the function in a bottom-up fashion

Memoization:

- Write the recursive function top-down
- Alter the function to check if we've already calculated the value
- If so, use the pre-calculate value
- If not, do the recursive call(s)

Memoized fibonacci

```
FIBONACCI(n)
if n = 1 or n = 2
        return 1
else
        return FIBONACCI(n − 1) + FIBONACCI(n − 2)
```

```
FIBONACCI-MEMOIZED(n)
fib[1] ← 1
fib[2] ← 1
for i ← 3 to n
        fib[i] ← ∞
return FIB-LOOKUP(n)
```

```
FIB-LOOKUP(n)
if fib[n] < ∞
        return fib[n]
fib[n] ← FIB-LOOKUP(n − 1) + FIB-LOOKUP(n − 2)
return fib[n]
```

Memoization

Pros

- Can be more intuitive to code/understand
- Can be memory savings if you don't need answers to all subproblems

Cons

- Depending on implementation, larger overhead because of recursion (though often the functions are tail recursive)

Quick summary

- Step 1: Define the problem with respect to subproblems
  - We did this for divide and conquer too. What's the difference?
  - You can identify a candidate for dynamic programming if there is **overlap** or **repeated work** in the subproblems being created
- Step 2: build the solution from the bottom up
  - Build the solution such that the subproblems referenced by larger problems are already solved
  - Memoization is also an alternative

Efficient greedy algorithm

Once you've identified a reasonable greedy heuristic:

- Prove that it always gives the correct answer
- Develop an efficient solution

Greedy Strategy

- How do we find greedy strategies that work?

1. Cast the optimization problem as one in which we make a choice and are left with one subproblem to solve.
2. Prove that there's always an optimal solution that makes the greedy choice, so that the greedy choice is always *safe*.
3. Demonstrate optimal substructure by showing that, having made the greedy choice, combining an optimal solution to the remaining subproblem with the greedy choice gives an optimal solution to the original problem.

When is greed good?

- No general way to tell whether a problem can be solved optimally using a greedy algorithm
- Two key ingredients:
  1. Greedy-choice property – Can assemble a globally optimal solution by making locally optimal (greedy) choices.
  2. Optimal substructure – Show that optimal solution to subproblem + greedy choice → optimal solution to the problem

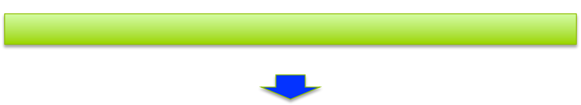
Greedy Recap

- The idea:
  - When we have a choice to make, make the one that looks the best *right now!*
  - Make a *locally optimal choice* in hopes of a *globally optimal solution*
- Key ingredients:
  1. Greedy-choice property – Can assemble a globally optimal solution by making locally optimal (greedy) choices.
  2. Optimal substructure – Show that optimal solution to subproblem + greedy choice → optimal solution to the problem

Greedy vs. divide and conquer

Divide and conquer

To solve the general problem:



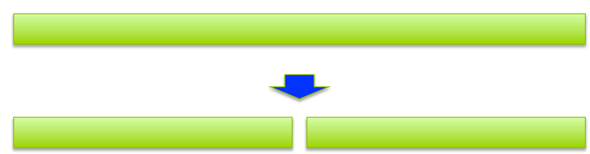
Break into sum number of sub problems, solve:

then possibly do a little work

Greedy vs. divide and conquer

Divide and conquer

To solve the general problem:

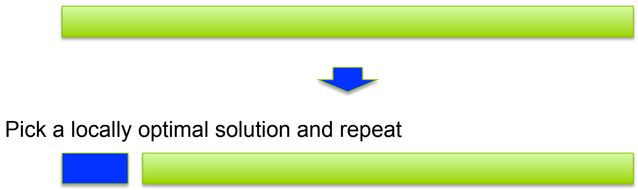


The solution to the general problem is solved with respect to solutions to sub-problems!

Greedy vs. divide and conquer

Greedy

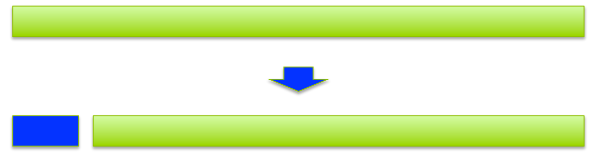
To solve the general problem:



Greedy vs. divide and conquer

Greedy

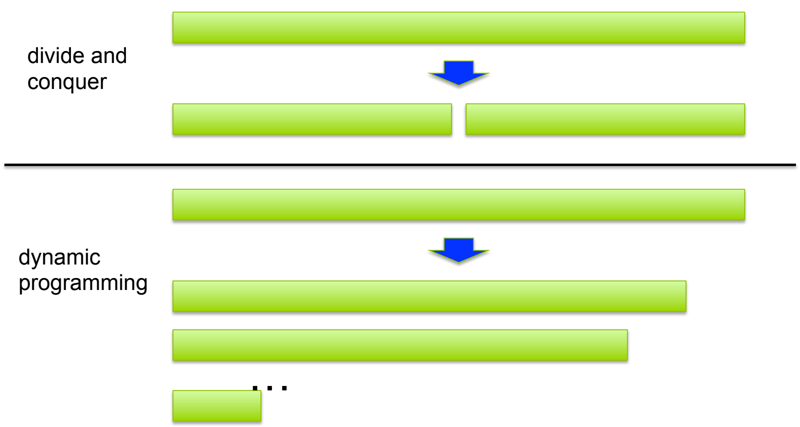
To solve the general problem:



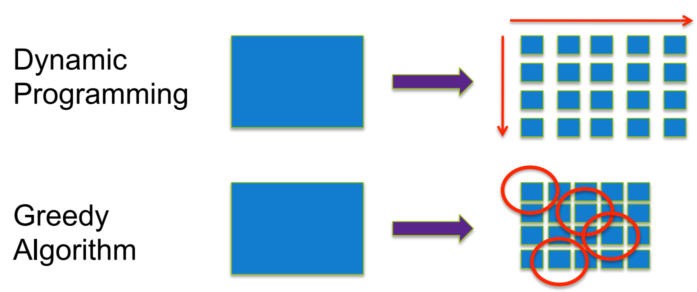
The solution to the general problem is solved with respect to solutions to sub-problems!

Slightly different than divide and conquer

D&C vs. DP: Overlapping sub-problems



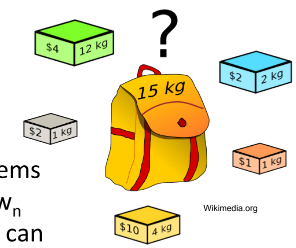
Greedy Algorithm vs Dynamic Programming



Greedy vs. DP (overview)

- With DP: solve subproblems first, then use those solutions to make an optimal choice
- With Greedy: make an optimal choice (without knowing solutions to subproblems) and then solve remaining subproblem(s)
- DP solutions are bottom up; greedy are top down
- Both apply to problems with optimal substructure: solutions to larger problems contains solutions to (1 or more) subproblems

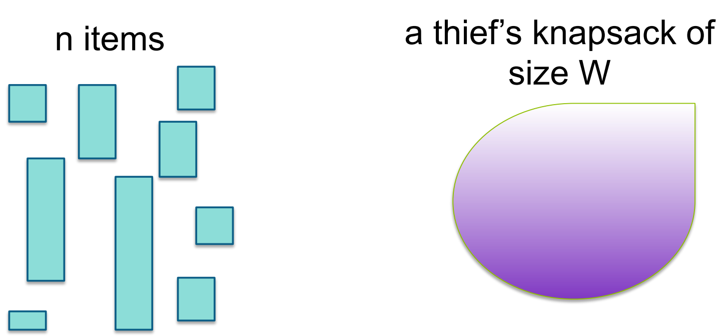
Knapsack problems: Greedy or not?



0-1 Knapsack – A thief robbing a store finds n items worth v_1, v_2, .., v_n dollars and weight w_1, w_2, …, w_n pounds, where v_i and w_i are integers. The thief can carry at most W pounds in the knapsack. Which items should the thief take if he wants to maximize value.

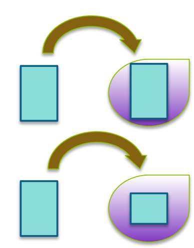
Fractional knapsack problem – Same as above, but the thief happens to be at the bulk section of the store and can carry fractional portions of the items. For example, the thief could take 20% of item i for a weight of $0.2w_i$ and a value of $0.2v_i$.

Knapsack Problem

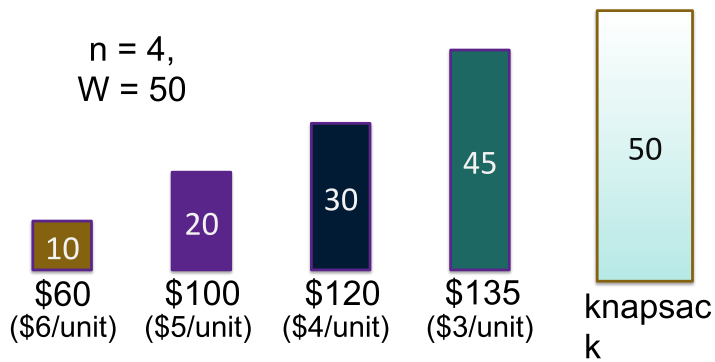


Knapsack Problem

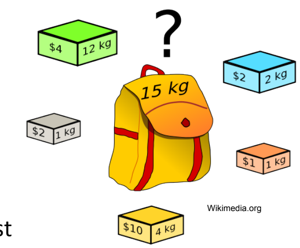
- 0-1 knapsack problem
  - Each item must be either taken or left behind.
- Fractional knapsack problem
  - The thief can take fractions of items.



Knapsack Problem



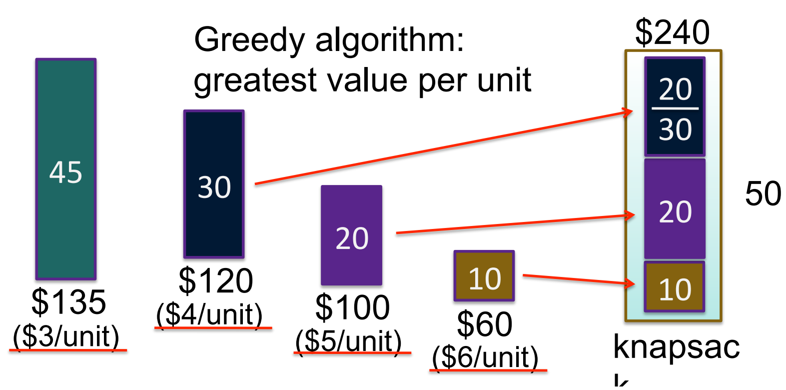
Solve and compare



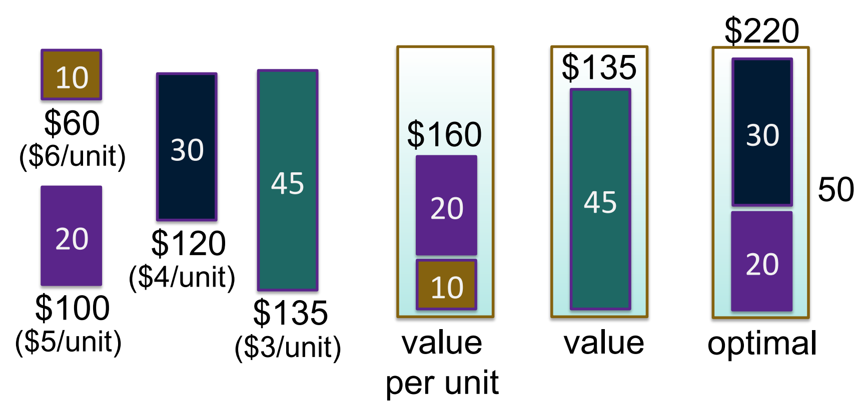
- Break into groups of at 3-4
  - Must contain someone from each row!
- Find an efficient algorithm that calculates the most valuable solution possible. Analyze:
  - Optimality – is it guaranteed to be optimal?
  - Runtime

	0-1 Knapsack	Fractional Knapsack
Greedy	Group #1	Group #2
Dynamic Programming	Group #3	Group #4

Fractional Knapsack Problem



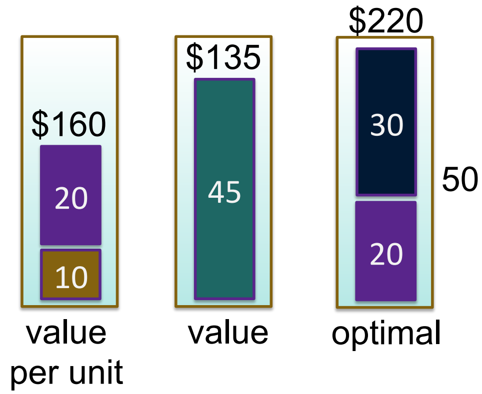
0-1 Knapsack Problem



0-1 Knapsack Problem

Difficult to get the optimal solution with a greedy strategy.

Dynamic Programming : $n \times W$



Greedy Algorithm vs Dynamic Programming

Dynamic Programming	Greedy Algorithm
Computes all subproblems	Find a local optimum
Always finds the optimal solution	May not be able to find the optimal solution
Compute all options before making choice, more memory	Typically faster, less memory