Dynamic Programming III: Knapsack Problem

Admin

- Exam distributed after class Friday; can be taken in any 24 hour period **ending at 10.00 am Wednesday** (must be submitted by start of Wednesday's class)
  - LaTeX template will be shared if you wish to use it
  - You may write by hand but must *clearly label answers*
    - The expectation is the same in either case: work out problem on scratch paper, write up final (clean) solution
- Friday we will meet upstairs
  - Activity to practice dynamic programming w.r.t. graphs
  - Also use Friday as a chance to ask questions about anything (including hw solutions)

Knapsack Problem

Further Reading: Chapter 6.4, KT

Knapsack Problem

Problem. Pack a knapsack to maximize the total item value

- There are n items, each with weight w_i and value v_i:

$$I = \{(v_1, w_1), \ldots, (v_n, w_n)\}$$

- Knapsack has total capacity C

- For any set of items T they fit in the Knapsack iff

$$\sum_{i \in T} w_i \leq C$$

- **Goal**: Find subset S of items that fit in the knapsack (satisfy the capacity constraint) **and maximize** the total value:

$$\sum_{i \in S} v_i$$

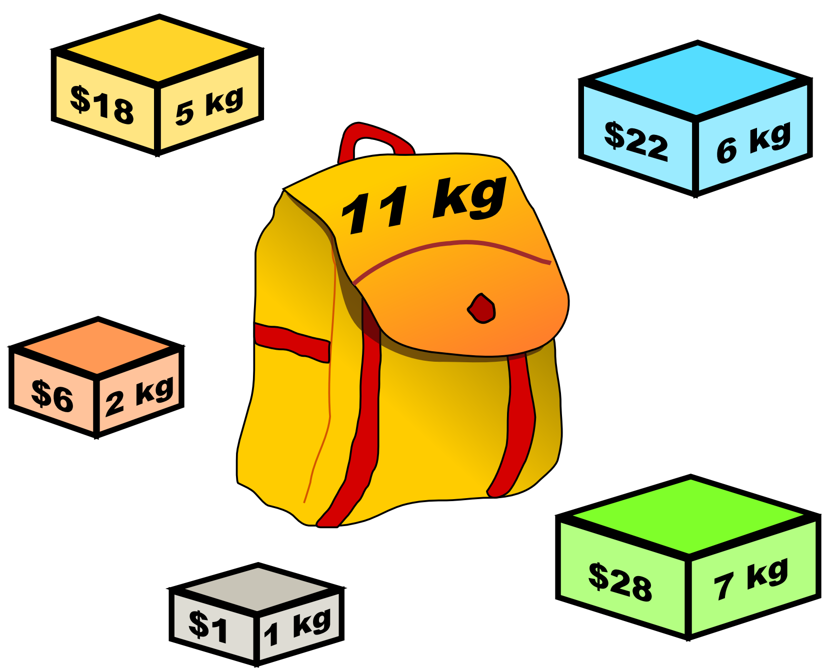
- **Assumption**. All weights and values are non-negative integers

Knapsack Problem

Let's first explore **greedy** solutions to the problem.

Consider the following problem instance:

- Ideas for what to be greedy about?



i	v_i	w_i
1	\$1	1 kg
2	\$6	2 kg
3	\$18	5 kg
4	\$22	6 kg
5	\$28	7 kg

Knapsack instance
(weight limit C = 11 kg)

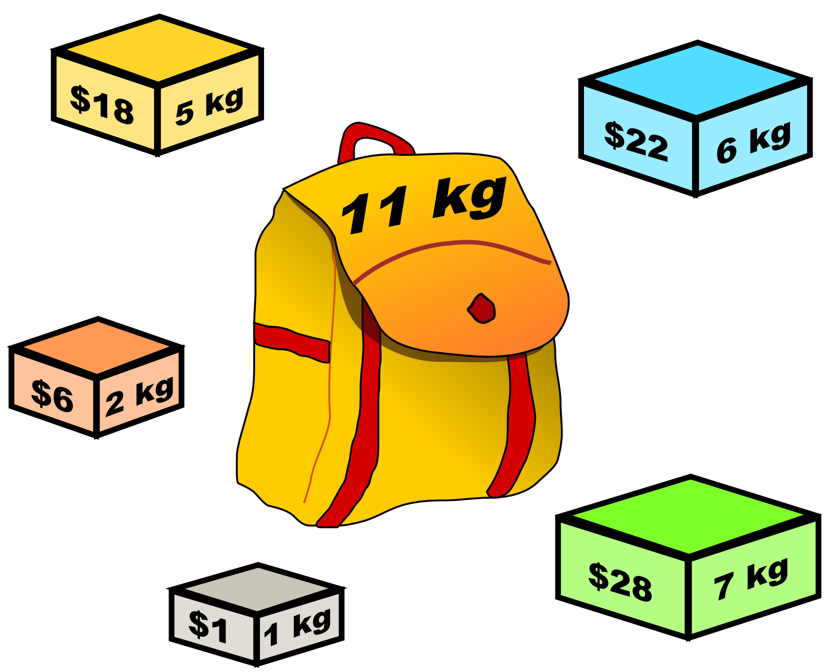
Knapsack Problem

Idea 1: Pick the most expensive stuff we can!

- **Algorithm**: greedily pick the highest value item that fits.

Total value: $35

Utilized capacity: 10 kg



i	v_i	w_i
1	\$1	1 kg
2	\$6	2 kg
3	\$18	5 kg
4	\$22	6 kg
5	\$28	7 kg

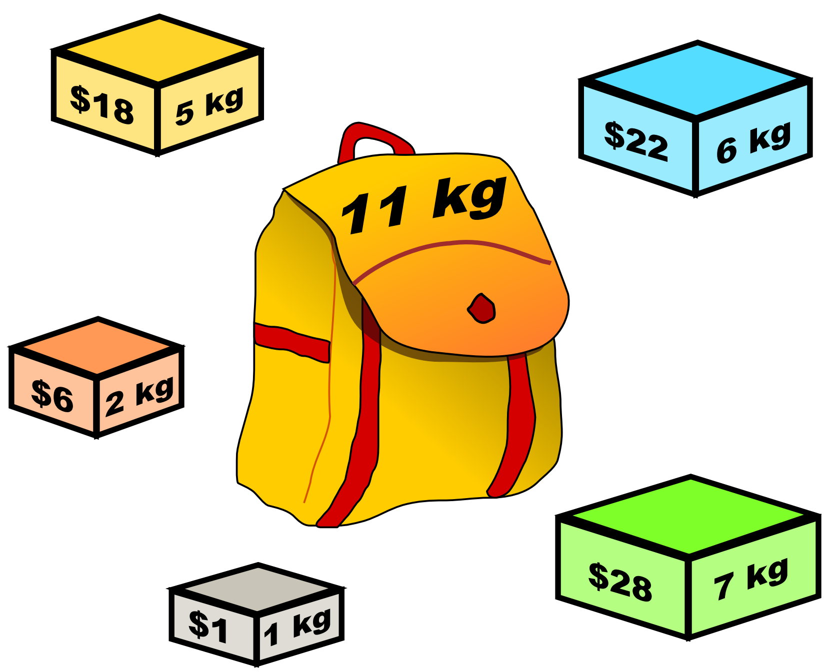
Knapsack instance

(weight limit C = 11 kg)

Knapsack Problem

Idea 2: Pick the lightest stuff we can!

- **Algorithm**: greedily pick the lowest weight item that fits.



Total value: $25
Utilized capacity: 9 kg

i	v_i	w_i
1	$1	1 kg
2	$6	2 kg
3	$18	5 kg
4	$22	6 kg
5	$28	7 kg

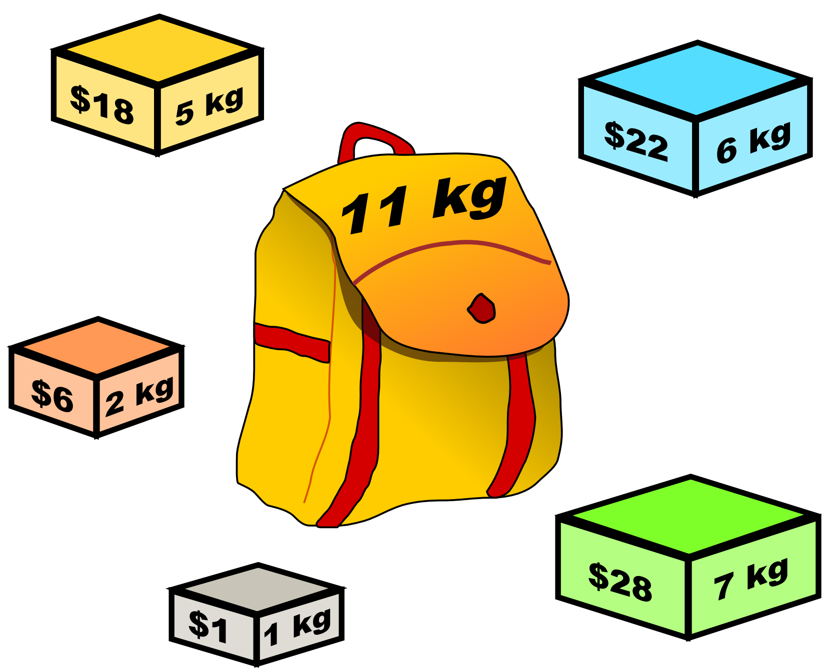
Knapsack instance
(weight limit C = 11 kg)

Knapsack Problem

Idea 1: Pick the most expensive stuff we can!

- **Algorithm**: greedily pick the highest weight item that fits.

Total value: $35
Utilized capacity: 10 kg



i	v_i	w_i
1	\$1	1 kg
2	\$6	2 kg
3	\$18	5 kg
4	\$22	6 kg
5	\$28	7 kg

Knapsack instance
(weight limit C = 11 kg)

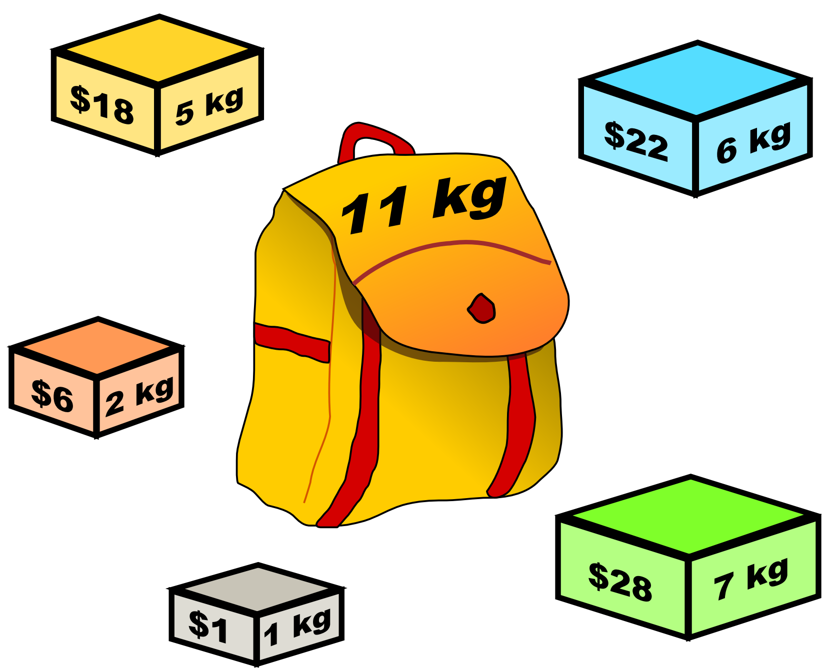
Knapsack Problem

Other ideas?

Spoiler: Greedy doesn't work! What is optimal in this instance?

- Optimal packing is $\{i_3, i_4\}$: value $40 (and weight 11)

How many packings muse we consider in an **exhaustive** search?



i	v_i	w_i
1	\$1	1 kg
2	\$6	2 kg
3	\$18	5 kg
4	\$22	6 kg
5	\$28	7 kg

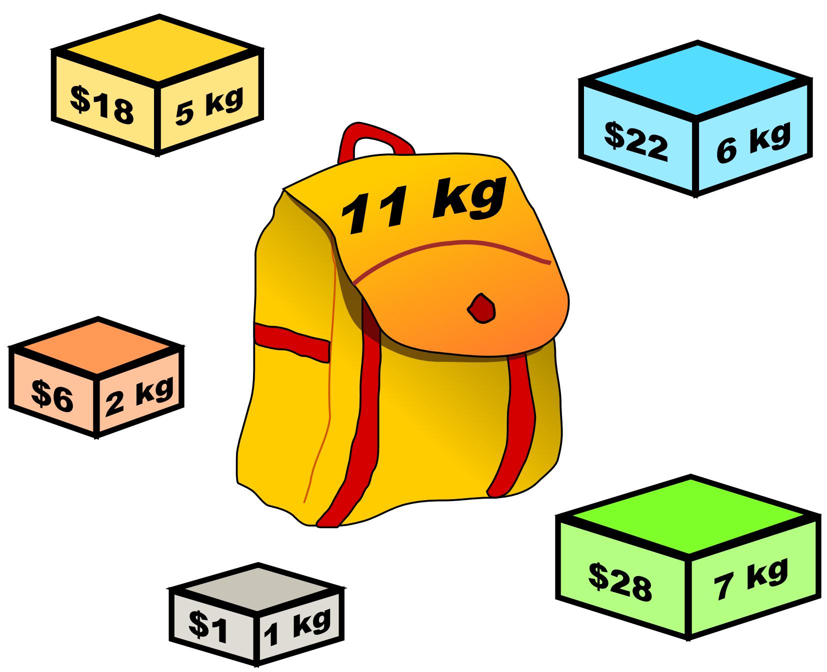
knapsack instance
(weight limit W = 11)

Exponential Possibilities

Given S items, how many subsets of items are there in total?

- 2^S: there are an exponential number of possibilities

- Dynamic programming trades of space for time, and through memoization, we get an (interestingly) efficient solution!



i	v_i	w_i
1	\$1	1 kg
2	\$6	2 kg
3	\$18	5 kg
4	\$22	6 kg
5	\$28	7 kg

knapsack instance
(weight limit W = 11)

Recipe for a Dynamic Program

- **Formulate the right subproblem.** The subproblem must have an optimal substructure
- **Formulate the recurrence.** Identify how the result of the smaller subproblems can lead to that of a larger subproblem
- **State the base case(s)**. The subproblem thats so small we know the answer to it immediately!
- **State the final answer.** (In terms of the subproblem)
- **Choose a memoization data structure.** Where are you going to store already computed results? (Usually a table)
- **Identify evaluation order.** Identify the dependencies: which subproblems depend on which ones? Using these dependencies, identify an evaluation order
- **Analyze space and running time.** As always!

Towards a Subproblem

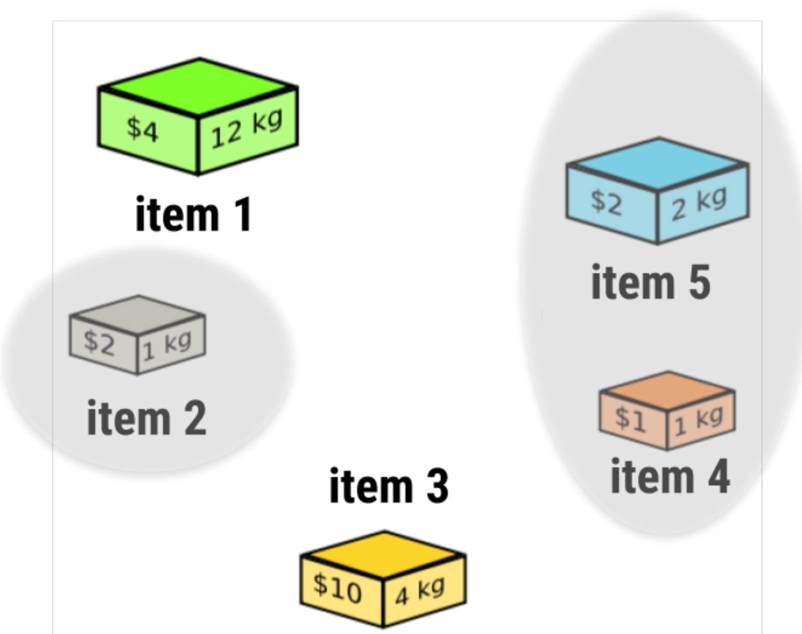
Previously, our DP has tracked a **value** instead of a **set**.

- **Idea 1**: Keep track of **current capacity**
- **Subproblem**. Let $T[c]$ denote the **value** of the optimal solution that uses capacity $\leq c$.
- **Optimal solution**: $T[C]$
- **Recurrence**: Not obvious with just capacities.
    - Why is this a challenge?

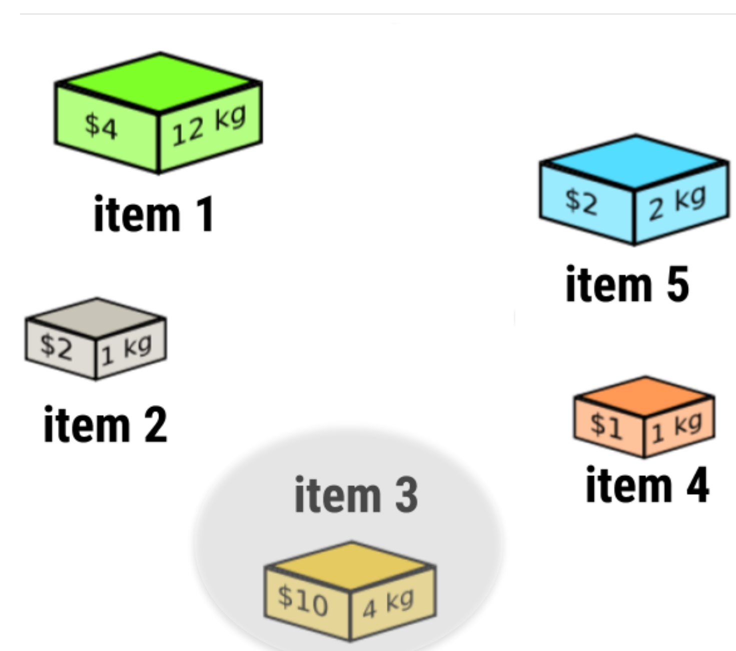
Subproblems and Optimality

When items are selected, we need to fill the remaining capacity optimally

- **Challenge**: the subproblem associated with a given remaining capacity can be solved in different ways



Partial Selection #1



Partial Selection #2

- In both cases, remaining capacity: 11 kg, but items left are different
  - Using just capacity might not be enough. Perhaps a 2D table can capture capacity *AND* items?

Subproblem:
Optimal Substructure

Subproblem

Subproblem

$\boxed{\text{OPT}(i, c)\text{: value of optimal solution using items } \{1,2,\ldots,i\} \text{ with total capacity } \leq c\text{, for } 1 \leq i \leq n,\ \ 0 \leq c \leq C}$

Final answer

$\boxed{\text{OPT}(n, C)}$

Consider all n items, consider full capacity C

Base Cases

$n \times C$: Are there any rows/columns can we fill immediately?

- What about the first column corresponding to item 1?

OPT$(1, c)$: Value of optimal solution that uses item 1 and has total capacity at most c

- For $i = 1; c \in \{1,2,\ldots,C\}$ we can fill out the first column as:

$$\boxed{\begin{array}{l} \text{OPT}(1, c) = v_1 \text{ if } c \geq w_1 \\ \text{OPT}(1, c) = 0 \text{ if } c < w_1 \end{array}}$$

Item 1 fits, add its value v_1

Item 1 does not fit, value of empty knapsack is 0

Base Cases

Are there any rows/columns can we fill immediately?

- What about the first row corresponding to capacity 0?
- OPT$(i,\ 0)$: Value of optimal solution that uses first i items and has total capacity at most 0
- For $i = 1,2,\ldots,n$ we can fill out the first row as:

$$\boxed{\text{OPT}(i,\ 0) = 0}$$

Items $1\ldots i$ do not fit, value of empty knapsack is 0

Optimal Substructure

- OPT(i, c): Let us try to construct the optimal solution that uses items $\{1,2,\ldots, i\}$ and capacity at most c
- What are the possibilities for the last i^{th} item:
  - Either item i is in the optimal solution or not
  - We must consider both cases
- **Case 1.** Suppose item i is **not** in the optimal solution, what is the optimal way to solve the remaining problem?
  - OPT(i, c) $=$ OPT$(i-1, c)$

Item i is left out, use best solution that considers items $1\ldots(i-1)$ for the same capacity

Optimal Substructure

- OPT(i, c): Let us try to construct the optimal solution that uses items $\{1,2,\ldots,i\}$ and capacity at most c
- What are the possibilities for the last i^{th} item:
  - Either item i is in the optimal solution or not
  - We must consider both cases
- **Case 2.** Suppose item i **is** in the optimal solution, what is the recurrence of the optimal solution?
  - OPT(i, c) $= v_i +$ OPT$(i - 1,\ c - w_i)$
    - This case only possible if $c \geq w_i$

Final Recurrence

For $1 \leq i \leq n$ and $1 \leq c \leq C$, we have:

$$\boxed{\begin{aligned}&\text{OPT}(i, c) = \\ &\max\{\text{OPT}(i-1,\ c),\ v_i + \text{OPT}(i-1,\ c-w_i)\}\end{aligned}}$$

- **Memoization structure**: We store OPT$[i, c]$ values in a 2-D array or table using space $O(nC)$
- **Evaluation order**: In what order should we fill in the table?
    - Row-major order (row-by-row)

Running Time

- Time to fill out a single table cell? $O(1)$
- How many cells are there in our table? $O(nC)$
- Total cost? $O(nC)$

Running Time

- Is $O(nC)$ polynomial? By which I mean polynomial in the *size of the input*
- What is the input? n items, plus C
    - We need $O(n)$ size to store n items
    - How much space to store C ? $\log_2 C$ *bits*
- Is $O(nC)$ polynomial?

  One table dimension depends on value of input, not input size

    - Not polynomial in C, but polynomial in n
    - "Pseudopolynomial" - polynomial in the ***value*** of the input
- To think about: does this work if the weights are not integers?

Recipe for a Dynamic Program

- **Formulate the right subproblem.** The subproblem must have an optimal substructure
- **Formulate the recurrence.** Identify how the result of the smaller subproblems can lead to that of a larger subproblem
- **State the base case(s)**. The subproblem thats so small we know the answer to it!
- **State the final answer.** (In terms of the subproblem)
- **Choose a memoization data structure.** Where are you going to store already computed results? (Usually a table)
- **Identify evaluation order.** Identify the dependencies: which subproblems depend on which ones? Using these dependencies, identify an evaluation order
- **Analyze space and running time.** As always!

Acknowledgments

- Some of the material in these slides are taken from
    - Kleinberg Tardos Slides by Kevin Wayne (https://www.cs.princeton.edu/~wayne/kleinberg-tardos/pdf/04GreedyAlgorithmsI.pdf)
    - Jeff Erickson's Algorithms Book (http://jeffe.cs.illinois.edu/teaching/algorithms/book/Algorithms-JeffE.pdf)
    - Shikha Singh