CS 344: Design and Analysis of Computer Algorithms

(Spring 2022 — Sections 5,6,7,8)

Lecture 10:
Dynamic Programming:
Knapsack, Longest Increasing Subsequence

Dynamic Programming:
It is just Smart Recursion

Fibonacci Numbers

- Fibonacci numbers are the following sequence:
  - 1,1,2,3,5,8,13,21,34,…
- $F(n)$: n-th Fibonacci number
- $F(1) = 1$, $F(2) = 1$
- $F(n) = F(n-1) + F(n-2)$

Recursive Algorithm

- **RecFibo**(n):
  - If n=1 or n=2, return 1
  - Otherwise, return **RecFibo**(n-1) + **RecFibo**(n-2)

Memoization

- Pick an array S[1:n] initialized with ‘undefined’ in every entry
- Whenever we compute F(i), store it in S[i]
- Next time, instead of recomputing F(i), just return S[i]

Fibonacci Numbers with Memoization

- Initialize an array S[1:n] with 'undefined' in every entry
- **MemFibo**(n):
  - If S[n] $\neq$ 'undefined', return S[n]
  - If n=1 or n=2, let S[n] = 1
  - Otherwise, let S[n] = MemFibo(n-1) + MemFibo(n-2)
  - Return S[n]

Bottom-Up Dynamic Programming

- We can literally fill up the array of stored answers ourselves
- **DynFibo**(n):
  - Create an empty array S[1:n]
  - Let $S[1] = S[2] = 1$
  - For $i = 3$ to n: let $S[i] = S[i-1] + S[i-2]$
  - Return S[n]

Elements of Dynamic Programming

Dynamic Programming?

- Write a recursive formula for the problem we want to solve
- Write a recursive function that computes the recursive formula
- Use a table to store the answer of recursive function for each recursive call and do not recompute them

Dynamic Programming?

- Write a recursive formula for the problem we want to solve
- Write a recursive function that computes the recursive formula
- Use a table to store the answer of recursive function for each recursive call and do not recompute them
- Two ways:
  - Memoization: Top-Down Dynamic Programming
  - Iterative: Bottom-Up Dynamic Programming

Writing Recursive Formula

- Step One: **Specification**
  - Answer to the question of **“What?”**
  - Describe the problem you want to solve using your formula in plain English
    - Example: “For every $1 \leq i \leq n$, the formula F(i) is supposed to be the i-th Fibonacci number”
  - Describe how the answer to the original problem can be obtained IF we have a solution to the recursive formula
    - Example: “Return F(n)”

Writing Recursive Formula

- Step Two: **Solution**
  - Answer to the question of **“How?”**
  - Give a recursive formula for solving for the problem you described in specification by solving the instances of the same problem
    - Example: “$F(1) = F(2) = 1$; for any $i > 2$, $F(i) = F(i-1)+F(i-2)$”
  - Prove that this recursive formula indeed matches the specification provided in the previous step
    - Answer to the question of **“Why?”**

Writing Recursive Formula

- Prove that this recursive formula indeed matches the specification provided in the previous step:
  - Prove that base case of recursive formula is correct
  - Prove that larger values of formula are computed correctly from the smaller values
- Note: this is just induction in disguise

Next Steps?

- Step Three:
  - Use either memoization or bottom-up dynamic programming
  - The choice is entirely up to you
    - Just remember in bottom-up dynamic programming, you have to specify the order of evaluation also

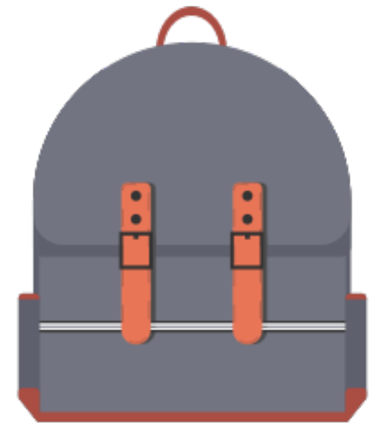
The Knapsack Problem

The Knapsack Problem

- **Input:**
  - A collection of n items with value v_i and weight w_i
  - A knapsack of size W

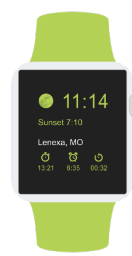
The Knapsack Problem

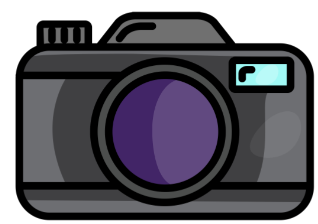
- **Input:**
  - A collection of n items with value v_i and weight w_i
  - A knapsack of size W



size: 35

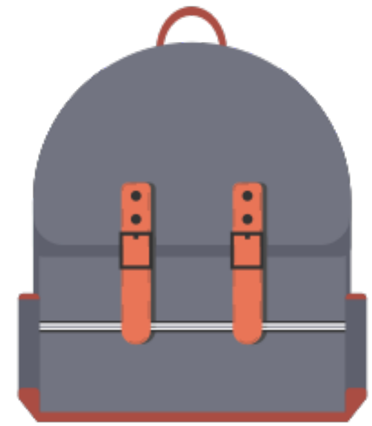
value:	**1000**	**5**	**50**	**5**	**100**
weight:	**25**	**5**	**8**	**5**	**15**





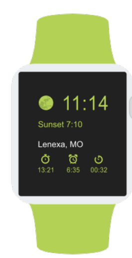
The Knapsack Problem

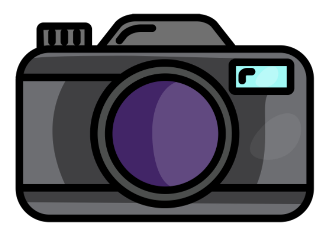
- **Output:**
  - The maximum value we can get by picking a subset S of items
  - The total weight of S should be less than Knapsack size



size: 35

value:	**1000**	**5**	**50**	**5**	**100**
weight:	**25**	**5**	**8**	**5**	**15**

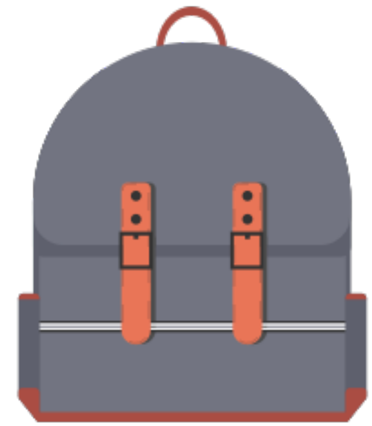



The Knapsack Problem

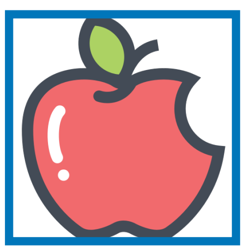
- **Output:**
  - The maximum valu
    picking a subset S of
  - The total weight of S should be less than
    Knapsack size

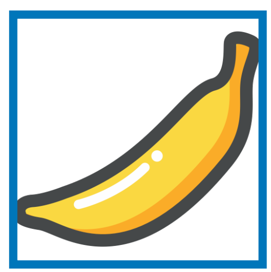
Value of 160 and weight 33

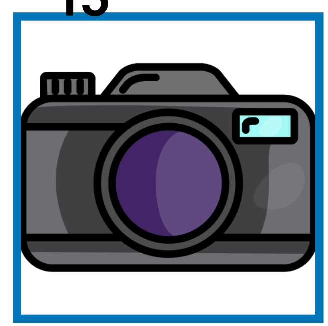


size: 35

value:	**1000**	**5**	**50**	**5**	**100**
weight:	**25**	**5**	**8**	**5**	**15**



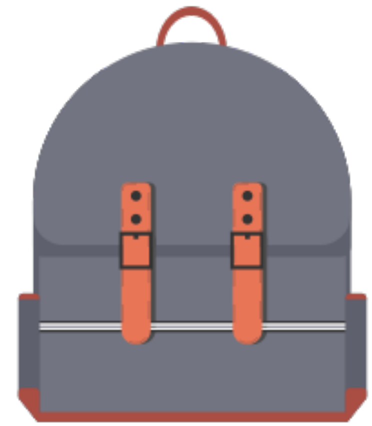




The Knapsack Problem

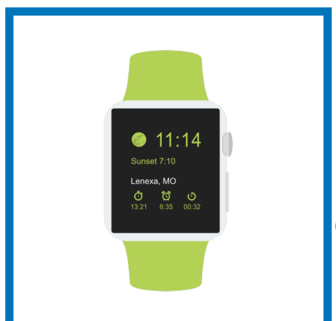
- **Output:**
  - The maximum valu
    picking a subset S of
  - The total weight of S uld be less than
    Knapsack size

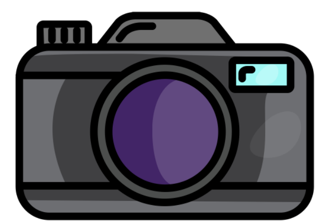
Value of 1050 and weight 32



size: 35

value:	**1000**	**5**	**50**	**5**	**100**
weight:	**25**	**5**	**8**	**5**	**15**

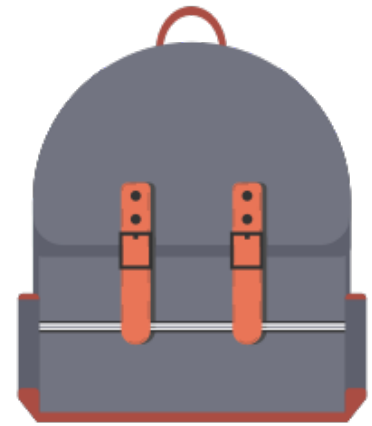




The Knapsack Problem

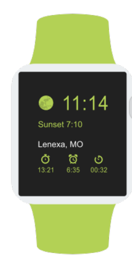
- **Output:**
  - The maximum valu[...] picking a subset S of [...]
  - The total weight of S should be les[...] than Knapsack size

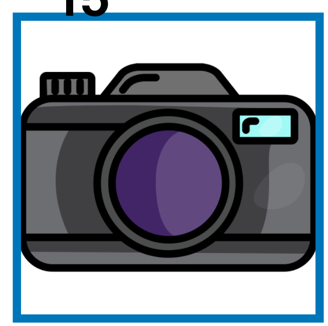
Value of 1100 and weight 40



size: 35

value:	**1000**	**5**	**50**	**5**	**100**
weight:	**25**	**5**	**8**	**5**	**15**

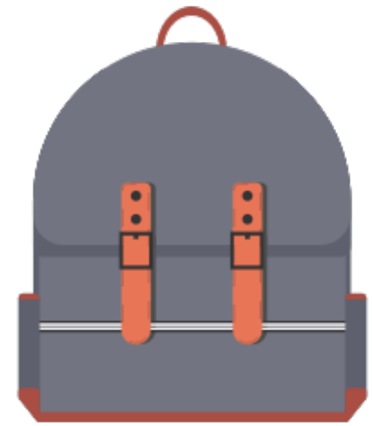




The Knapsack Problem

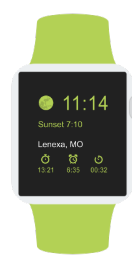
- **Output:**
  - The maximum valu
    picking a subset S of
  - The total weight of S should be les than
    Knapsack size

Value of 1100 and weight 40



size: 35

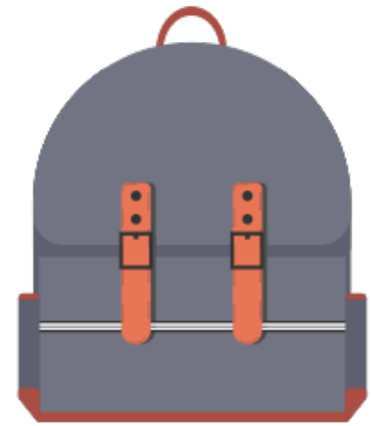
value:	**1000**	**5**	**50**	**5**	**100**
weight:	**25**	**5**	**8**	**5**	**15**



The Knapsack Problem

- **Output:**
  - The maximum valu picking a subset S of
  - The total weight of S uld be less than Knapsack size

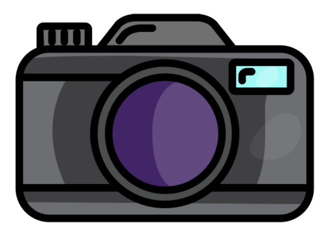
Value of 1050 and weight 32



size: 35

value:	**1000**	**5**	**50**	**5**	**100**
weight:	**25**	**5**	**8**	**5**	**15**



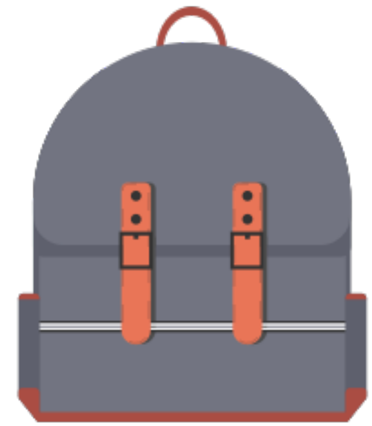



An Algorithm for Knapsack?

- Does the following work:
  - pick the most valued item that still fits the knapsack
  - Repeat

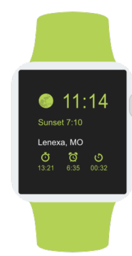
An Algorithm for Knapsack?

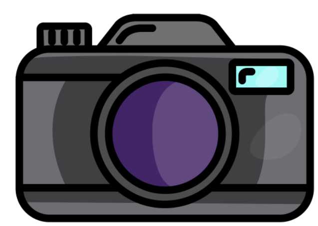
- Does the following work:
  - pick the most valued item that still fits the knapsack
  - Repeat



size: 35

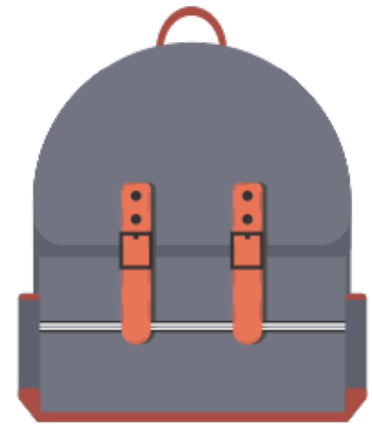
value:	**1000**	**5**	**50**	**5**	**100**
weight:	**25**	**5**	**8**	**5**	**15**

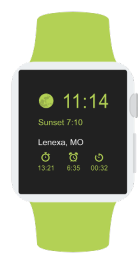
An Algorithm for Knapsack?

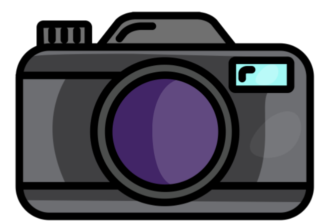
- Does the following work:
  - pick the most valued item that still fits the knapsack
  - Repeat



size: 35

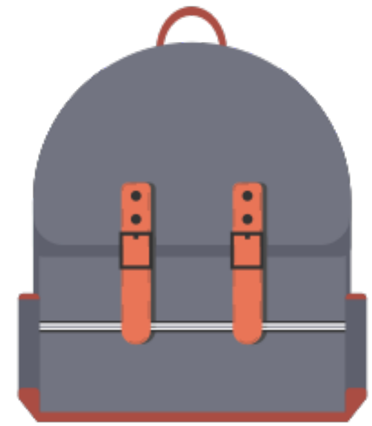
value:	**1000**	**5**	**50**	**5**	**100**
weight:	**25**	**5**	**8**	**5**	**15**





An Algorithm for Knapsack?

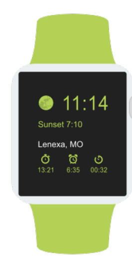
- Does the following work:
  - pick the most valued item that still fits the knapsack
  - Repeat

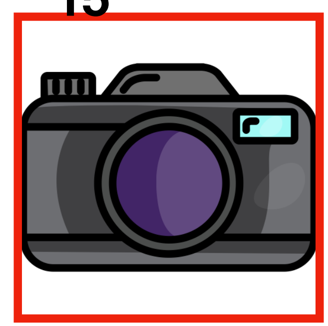


size: 35

value:	**1000**	**5**	**50**	**5**	**100**
weight:	**25**	**5**	**8**	**5**	**15**

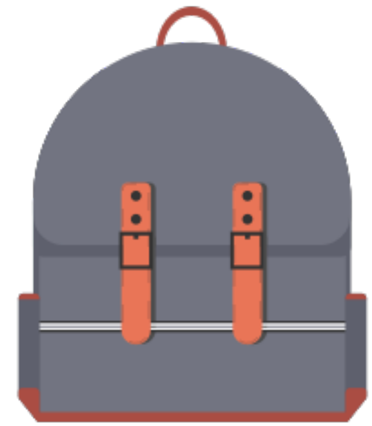




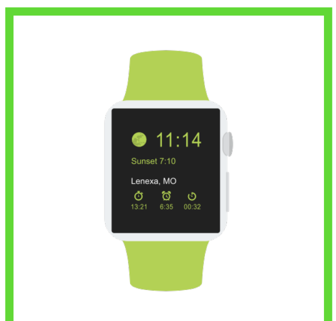
An Algorithm for Knapsack?

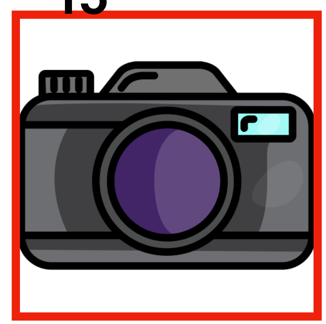
- Does the following work:
  - pick the most valued item that still fits the knapsack
  - Repeat



size: 35

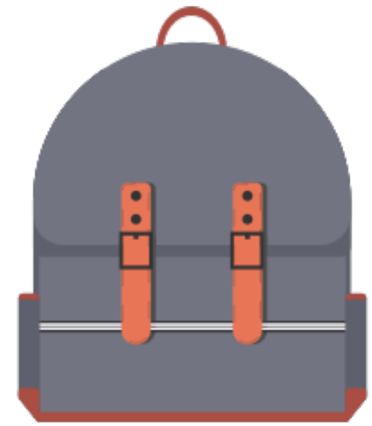
value:	**1000**	**5**	**50**	**5**	**100**
weight:	**25**	**5**	**8**	**5**	**15**





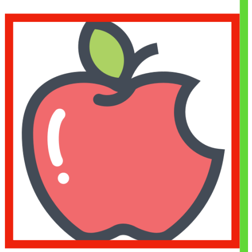
An Algorithm for Knapsack?

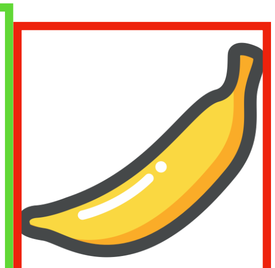
- Does the following work:
  - pick the most valued item that still fits the knapsack
  - Repeat

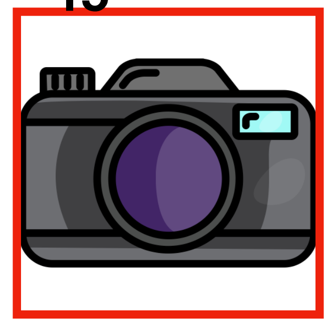


size: 35

value:	**1000**	**5**	**50**	**5**	**100**
weight:	**25**	**5**	**8**	**5**	**15**

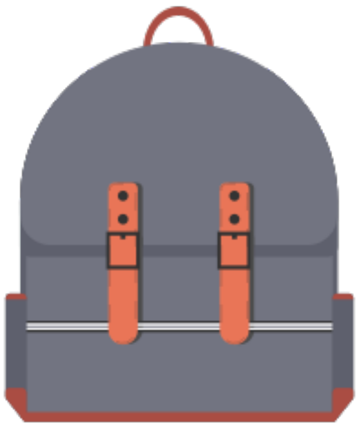






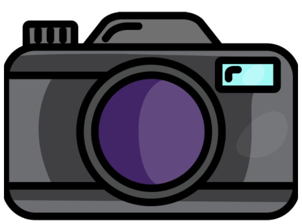
An Algorithm for Knapsack?

- Does the following work:
  - pick the most valued item that still fits the knapsack
  - Repeat



size: 35

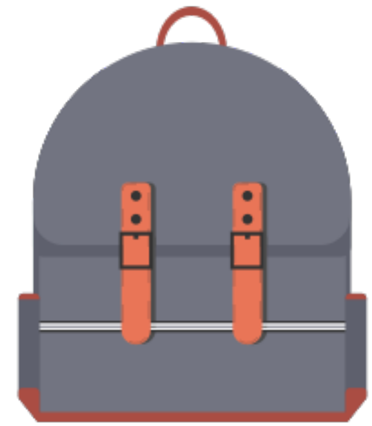
value:	**1000**	**5**	**50**	**5**	**100**
weight:	**25**	**5**	**8**	**5**	**15**



An Algorithm for Knapsack?

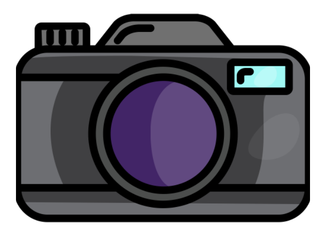
- Does the following work:
  - pick the most valued item that still fits the knapsack
  - Repeat

Seems to work on this example



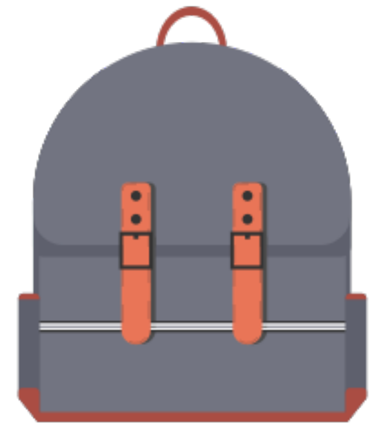
size: 35

value:	1000	5	50	5	100
weight:	25	5	8	5	15



An Algorithm for Knapsack?

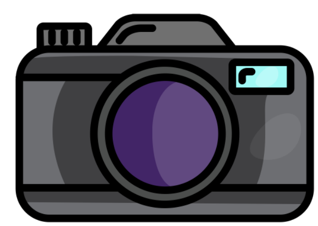
- Does the following work:
  - pick the most valued item that still fits the knapsack
  - Repeat



size: 35

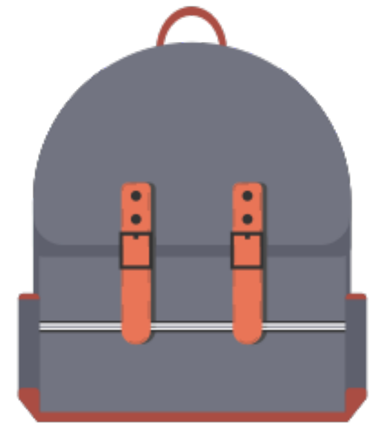
value:	**1000**	**5**	**950**	**5**	**100**
weight:	**25**	**5**	**15**	**5**	**15**



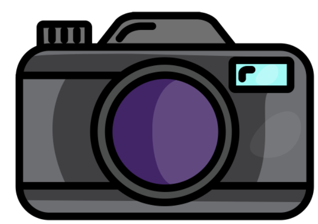
An Algorithm for Knapsack?

- Does the following work:
  - pick the most valued item that still fits the knapsack
  - Repeat



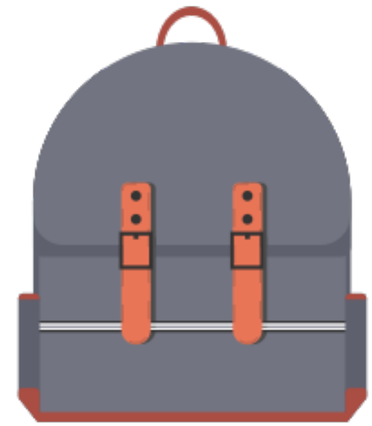
size: 35

value:	**1000**	**5**	**950**	**5**	**100**
weight:	**25**	**5**	**15**	**5**	**15**



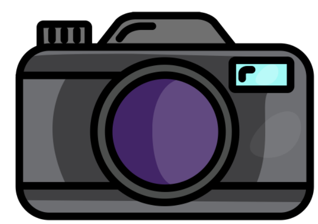
An Algorithm for Knapsack?

- Does the following work:
  - pick the most valued item that still fits the knapsack
  - Repeat



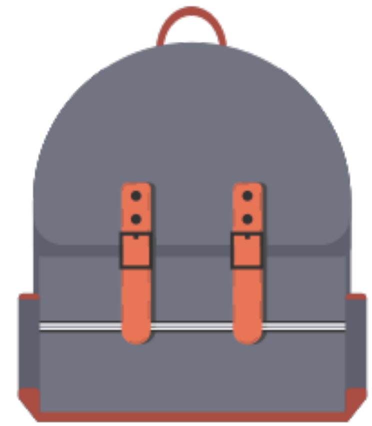
size: 35

value:	**1000**	**5**	**950**	**5**	**100**
weight:	**25**	**5**	**15**	**5**	**15**



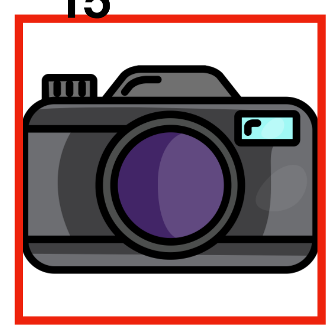
An Algorithm for Knapsack?

- Does the following work:
  - pick the most valued item that still fits the knapsack
  - Repeat



size: 35

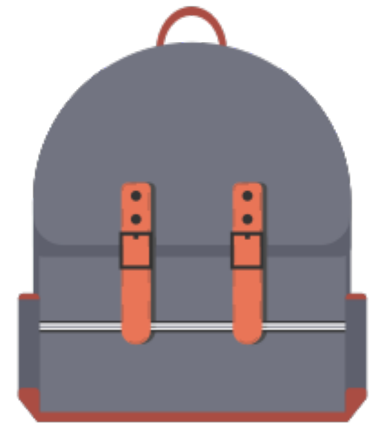
value:	**1000**	**5**	**950**	**5**	**100**
weight:	**25**	**5**	**15**	**5**	**15**



An Algorithm for Knapsack?

- Does the following work:
  - pick the most valued item that still fits the knapsack
  - Repeat

Total value: 1010



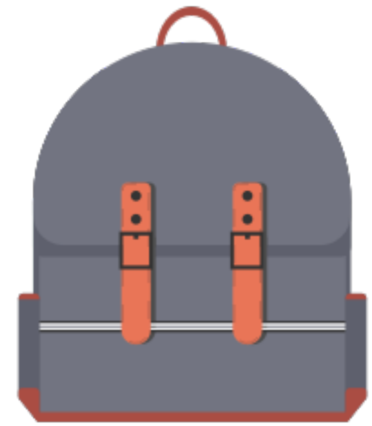
size: 35

value:	**1000**	**5**	**950**	**5**	**100**
weight:	**25**	**5**	**15**	**5**	**15**

An Algorithm for Knapsack?

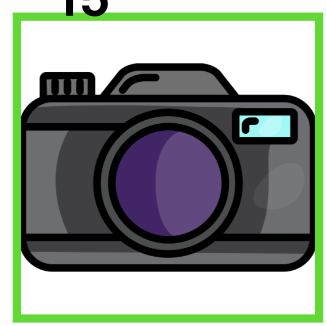
- Does the following work:
  - pick the most valued item that still fits the knapsack
  - Repeat

Total value: 1050



size: 35

value:	**1000**	**5**	**950**	**5**	**100**
weight:	**25**	**5**	**15**	**5**	**15**



A Dynamic Programming Algorithm

- Step one: **Specification**
  - For every $0 \leq i \leq n \quad , \quad 0 \leq j \leq W$ define:
  - $K(i, j)$: the maximum value we get by picking a subset of items from the first i items when we have a knapsack of size j
  - How to compute the final answer?
    - Return $K(n, W)$

A Dynamic Programming Algorithm

- Step two: **Solution**
- $$K(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ K(i-1,j) & \text{if } w_i > j \\ \max\{K(i-1,j-w_i) + v_i\ ,\ K(i-1,j)\} & \text{otherwise} \end{cases}$$
- Proof?

A Dynamic Programming Algorithm

- Pick a 2-dimensional array D[1:n][1:W] initialized with 'undefined'
- **MemKnap**(i,j):
  - If D[i][j] $\neq$ undefined return D[i][j]
  - If i=0 or j=0, return D[i][j] = 0
  - If $w_i > j$ let D[i][j] = **MemKnap**$(i-1,j)$
  - Else let D[i][j] = max{**MemKnap**$(i-1,j-w_i) + v_i$, **MemKnap**$(i-1,j)$}
  - Return D[i][j]

A Dynamic Programming Algorithm

- Proof of correctness? This is the same formula as $K(i, j)$ so nothing else to prove
- Runtime?
  - There are $(n+1) \cdot (W+1)$ subproblems
  - Each takes $O(1)$ time
  - So total runtime is $O(nW)$