EECS3101 (Section E) Summer 2026
Tutorial 9
Dynamic Programming with Two-Dimensional States
Solving the 0/1 Knapsack Problem

Chen-Wei Wang

Release Date: Thursday, July 30
No Submission Required: Complete for Learning & Test Prep

Contents

1 Background

- **Inputs**:
  - an array *weights*, where $weights[i]$ denotes the weight of item i;
  - an array *values*, where $values[i]$ denotes the value of item i; and
  - a positive integer *capacity*, which denotes the **maximum** total weight the knapsack can carry.
- **Output**: Find **maximum** total value of items that can fit into the knapsack.
- **Rules**:
  - The **"0/1" rule** for choosing items:
    - For each item, you either take it (1) or do not take it (0).
    - You cannot take a fractional amount of an item.
    - You cannot take the same item multiple times.
  - The total selected weight must be at most *capacity* (i.e., $\leq capacity$).
- **Example 1**:
  - Inputs: $weights = \{2, 5, 7\}$, $values = \{20, 30, 35\}$, $capacity = 6$ Output: 30
  - Optimal strategy:
    - Choose item 1.
    - Total weight: $weights[1] = 5 \leq capacity$
    - Total value: $values[1] = 30$
- **Example 2**:
  - Inputs: $weights = \{1, 2, 3\}$, $values = \{6, 10, 12\}$, $capacity = 5$ Output: 22
  - Optimal strategy:
    - Choose items 1 and 2.
    - Total weight: $weights[1] + weights[2] = 2 + 3 = 5 \leq capacity$
    - Total value: $values[1] + values[2] = 10 + 12 = 22$
- **Example 3**:
  - Inputs: $weights = \{4, 5, 1, 3\}$, $values = \{5, 7, 1, 4\}$, $capacity = 7$ Output: 9
  - Optimal strategy:
    - Choose items 0 and 3.
    - Total weight: $weights[0] + weights[3] = 4 + 3 = 7 \leq capacity$
    - Total value: $values[0] + values[3] = 5 + 4 = 9$
- **Example 4**:
  - Inputs: $weights = \{3, 8, 4, 5, 2\}$, $values = \{15, 30, 16, 20, 10\}$, $capacity = 10$ Output: 45
  - Optimal strategy:
    - Choose items 0, 3, and 4.
    - Total weight: $weights[0] + weights[3] + weights[4] = 3 + 5 + 2 = 10 \leq capacity$
    - Total value: $values[0] + values[3] + values[4] = 15 + 20 + 10 = 45$

    Note. Item 1 (with the highest value, 30) was not chosen. Why?

2 Tasks

Given a working (yet inefficient) recursive solution, design and implement a top-down DP solution using *memo* and a bottom-up DP solution using *tab*:

- Download `EECS3101_Tutorials_09_Starter.zip` from the lectures site and import it into Eclipse.
- Complete the corresponding DP methods.

3 Hints

- Hints for DP in general
  - Define the *subproblem* carefully before thinking about code.
  - This is a classic *include-or-exclude* recurrence.
  - The table is 2D because one state variable is not enough: you must track both **which items are under consideration**, and **how much capacity remains**.
  - The hardest part is usually deciding what each *state* means.

    $\Rightarrow$ Study these two state variables from the recursive helper method `knapsackHelper`, which has already been implemented for you. **Trace examples to observe patterns in the recursion trees, like how we did for the problem of Minimum Cost Climbing.**
- Hints for Top-Down DP (using memoization)
  - A recursive state needs two parameters, not one. A natural definition is:

    $K(i, c)$ = *maximum value achievable using items from index* $\boldsymbol{i}$ *onward with remaining capacity* $\boldsymbol{c}$
  - Then ask:
    - what if we choose to exclude item i?
    - what if we choose to include item i, if it fits?
  - $memo[i][c]$ should store the answer for some subproblem.
    - Before computing a state, check whether it has already been solved (and thus stored).
    - Use a special initial value like -1 to mean "not computed yet".
- Hints for Bottom-Up DP (using tabulation)
  - Think about what $tab[i][c]$ means before filling the table.
  - A natural meaning is:

    $tab[i][c]$ = *maximum value achievable using the first* $\boldsymbol{i}$ *items with capacity* $\boldsymbol{c}$
  - Then each entry comes from two possibilities:
    - item $i-1$ excluded
    - item $i-1$ included, if it fits
  - Compute smaller subproblems first.
  - Build the 2D *tab* row by row, where each row uses results from the previous row.