CSE3318: Dynamic Programming
0-1 Knapsack Problem
by
Dr. Bhanu Jain

Unauthorized copying, distribution, or reproduction of this material is **strictly prohibited**.

All slides are based on: ***Introduction to Algorithms***, by Thomas H. Cormen, Charles E. Leiserson, Ronald E. Rivest, Clifford Stein,3rd edition (CLRS)

© Bhanu Jain 2025

Dynamic Programming

- **Divide and Conquer method:**
  - solve problems by combining the solutions to subproblems
  - partition the problem into disjoint subproblems
  - solve the subproblems recursively, and then combine their solutions to solve
  - does more work than necessary, repeatedly solving the common subproblems
- **Dynamic programming:**
  - applies when the subproblems overlap —that is, when subproblems share subproblems.
  - solves each subproblem just once and then saves its answer in a table
  - avoiding the work of recomputing the answer every time it solves each subproblem
  - applied to optimization problems
  - such subproblems can have many possible solutions
  - such solution has a value, find a solution with the optimal (minimum or maximum) value
  - call such a solution *an* optimal solution to the problem, as opposed to *the* optimal solution

© Bhanu Jain 2025

Dynamic Programming

- Dynamic-programming algorithm, follows a sequence of four steps:
  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 a bottom-up fashion.
  4. Construct an optimal solution from computed information.

  Steps 1–3 form the **basis** of a dynamic-programming solution to a problem
  - If we need only the **value of an optimal solution**, and **not the solution itself**, then we can omit step 4
  - When we do perform **step 4**, we sometimes **maintain additional information** during step 3 so that we can easily **construct an optimal solution**

- Applies to a problem that at first seems to require a lot of time (possibly exponential), provided we have:
  - **Subproblem optimality:** the global optimum value can be defined in terms of optimal subproblems
  - **Subproblem overlap:** the subproblems are not independent, but instead they overlap (hence, should be constructed bottom-up).

© Bhanu Jain 2025

The 0/1 Knapsack Problem

- **Given:** A set S of n items, with each item i having
  - w_i - a positive weight
  - b_i - a positive benefit
- **Goal**: Choose items with maximum total benefit but with at most weight W.
- If we are not allowed to take fractional amounts, then this is the 0/1 knapsack problem.
  - In this case, we let T denote the set of items we take
  - **Objective:** maximize

$$\sum_{i \in T} b_i$$

  - **Constraint**:

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

© Bhanu Jain 2025

The 0/1 Knapsack Problem

Solution Approaches:

1.Brute Force: Try all subsets (exponential time complexity, O(2^n)).

2.Dynamic Programming: Uses a table to store solutions to subproblems (O(nW) complexity).

3.Greedy Approach (not optimal for 0-1 case): Works well for Fractional Knapsack but not for 0-1 Knapsack.

© Bhanu Jain 2025

The 0/1 Knapsack Problem: Brute Force Approach

Brute Force: Try all subsets (exponential time complexity, O(2^n)).

- With n items, there are $\mathbf{2^n}$ possible ways to select items.
- Check all combinations to find the one with the highest value while keeping the total weight ≤ W.
- This results in a time complexity of $\mathbf{O(2^n)}$—exponential and inefficient.

Can We Optimize This?

- Yes! Dynamic programming offers a more efficient solution.
- The key is to break the problem into manageable subproblems.

Defining Subproblems

- Suppose items are labeled 1 to n.
- A natural subproblem is:
  Find the optimal solution for the subset S_k = {items 1, 2, …, k}.
- This seems reasonable, but can we express the final solution S_n in terms of these subproblems S_k?

A Challenge

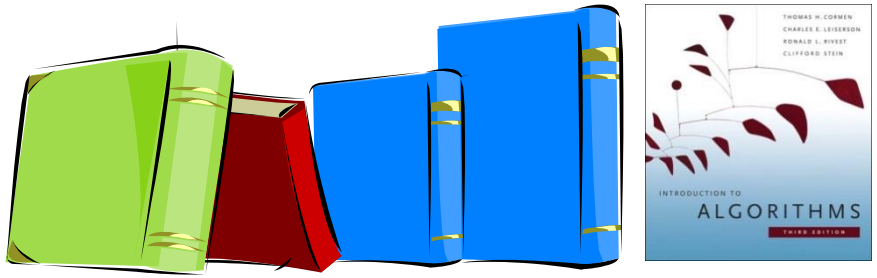
- This approach doesn't directly work—we need a better way to define our subproblems.

© Bhanu Jain 2025

The 0/1 Knapsack Problem

- Given: A set S of n items, with each item i having
  - b_i - a positive "benefit"
  - w_i - a positive "width"
- Goal: Choose items with maximum total benefit but with **w** at most **W.**

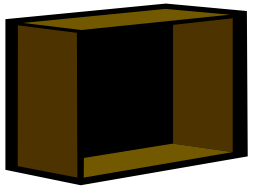
Items:



1 2 3 4 5

Width:	4 in	2 in	2 in	6 in	2 in
Benefit:	$20	$3	$6	$25	$80

"knapsack"



box of width 9 in

Solution:

- item 5 ($80, 2 in)
- item 3 ($6, 2in)
- item 1 ($20, 4in)

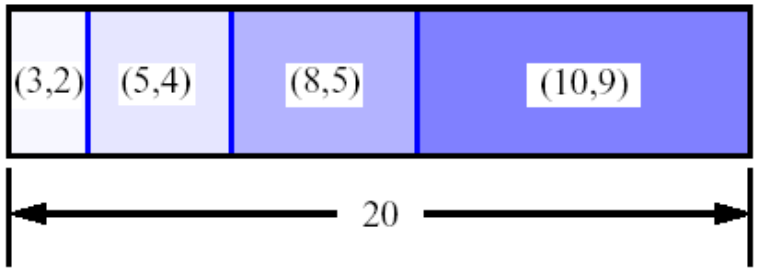
© Bhanu Jain 2025

The 0/1 Knapsack Problem

- **S_k: Set of items numbered 1 to k.**
- Define B[k] = best selection from S_k.
- Problem: does not have subproblem optimality:
  - Consider set S={(3,2),(5,4),(8,5),(4,3),(10,9)} of (**benefit, weight**) pairs and total weight **W = 20**

Best for S_4:

Best for S_5:



A problem has Optimal Substructure Property if the optimal solution can be computed by using the optimal solution to its subproblems without having to compute all possible subproblems

Solution for S_4 is not part of the solution for S_5!!!

© Bhanu Jain 2025

The 0/1 Knapsack Problem: Recursive formulation

Defining the Problem

- Let S_k represent the set of items numbered **1 to k**.
- Define **B[k, w]** as the optimal selection of items from S_k with a total weight of at most **w**.
- **Good news:** This problem follows the principle of **subproblem optimality**.

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

Finding the Best Subset

- The optimal subset of S_k with weight ≤ **w** is determined by:
  1. The **best subset** of S_{k-1} that fits within weight **w**, OR
  2. The **best subset** of S_{k-1} with weight at most **w - w_k**, plus item **k**.

Considering Two Cases

- Case 1: If $w_k > w$, item k is too heavy and cannot be included.
- Case 2: If $w_k \leq w$, item k can be included, and we choose the option that gives the **higher total value**.

© Bhanu Jain 2025

The 0/1 Knapsack Problem: Algorithm

```
// Initialize base cases
for w = 0 to W:   // O(W)
   B[0, w] = 0

for i = 1 to n:   //O(n)
   B[i, 0] = 0

// Fill the DP table
for i = 1 to n:          //O(n)
   for w = 0 to W:     //O(W)
      if w_i <= w // Item i can be included
         if b_i + B[i-1,w-w_i] > B[i-1,w]
                  B[i,w] = b_i + B[i-1,w- w_i]
            else
                  B[i,w] = B[i-1,w]
      else:
         B[i, w] = B[i-1, w]  // Item i is too heavy, exclude it
```

Case 2: If $\mathbf{w_k \le w}$, item **k** can be included, and we choose the option that gives the **higher total value**.

Case 1: If $\mathbf{w_k > w}$, item **k** is too heavy and cannot be included.

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

Running time: **O(n*W)**

Brute-force algorithm: $O(2^n)$

© Bhanu Jain 2025

The 0/1 Knapsack Problem: Finding the selected items

Extracting the Solution from the Table

- The **table contains all necessary information** to determine the optimal selection.
- **B[n, W]** represents the **maximum value** achievable within the knapsack's weight limit.

Reconstructing the Optimal Solution

SelectedItems_Solution(B, n, W):

Let i=n and k=W
selected_items ={} //Initialize an empty list
//B[i,k] stores the maximum benefits for the first i items and capacity W.
for i = n to 1 do: // Iterate backward through items
if $B[i,k] \neq B[i-1,k]$ then mark the *i*th item as in the knapsack
Selected_items ← *item i*
$i = i-1, k = k-w_i$ *//w_i weight of ith item*
else
i = i-1 // Assume the *i*th item is not in the knapsack
// Could it be in the optimally packed knapsack?
return selected_items

© Bhanu Jain 2025

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

The 0/1 Knapsack Problem: Algorithm

i/W	0	1	2	3	4	5
(wt,Ben): 0	0	0	0	0	0	0
1: (2,3) 1	0	0	3	3	3	3
2: (3,4) 2	0	0	3	4	4	7
3: (4,5) 3	0	0	3	4	5	7
4: (5,6) 4	0	0	3	4	5	7

S_k: Set of items numbered 1 to k.
B[k,w] -the best selection from S_k with weight at most w
B[k,w]: Benefit or value
b_i : benefit or value of the ith item
w_k: Weight of the kth element

```
// Initialize base cases
for w = 0 to W:   // O(W)
   B[0, w] = 0

for i = 1 to n:   //O(n)
   B[i, 0] = 0

// Fill the DP table
for i = 1 to n:          //O(n)
   for w = 0 to W:     //O(W)
      if w_i <= w // Item i can be included
         if b_i + B[i-1,w-w_i] > B[i-1,w]
                  B[i,w] = b_i + B[i-1,w- w_i]
            else
                  B[i,w] = B[i-1,w]
      else:
         B[i, w] = B[i-1, w] //Item i is too heavy, exclude it
```


© Bhanu Jain 2025

Summary:0/1 Knapsack Problem & Dynamic Programming

- **Dynamic Programming (DP)** is a powerful technique for solving optimization problems efficiently by breaking them into overlapping subproblems.
- Instead of recomputing solutions, DP stores previously computed results (memoization) and reuses them, significantly reducing redundant calculations.
- The 0/1 Knapsack problem is an ideal use case for DP since its solution can be described recursively using previously solved subproblems.

Algorithm	Time Complexity	Approach
Naïve Recursive/Brute Force	**$O(2^n)$**	Tries all possible subsets (exponential time)
Dynamic Programming	**$O(W \times n)$**	Uses a DP table to store intermediate results (pseudo-polynomial time)

© Bhanu Jain 2025

Summary:0/1 Knapsack Problem & Dynamic Programming

Pseudo-Polynomial Time Complexity: An algorithm runs in pseudo-polynomial time if its running time is polynomial in the numerical values of the input, but not necessarily in the size of the input (number of bits required to represent it).

- A problem is truly polynomial if its time complexity is a polynomial function of the input size (number of bits needed to represent the input).
- A problem is pseudo-polynomial if its time complexity depends on the actual numerical value of the input, rather than just the number of bits.

Example: 0/1 Knapsack Problem

The **dynamic programming solution** for the 0/1 Knapsack problem runs in **O(W × n)**,

- **Where W** = total weight capacity of the knapsack & **n** = number of items.

This is **not polynomial in the input size** because:

- The size of W in bits is log(W) (since W is represented in binary).
- If W is large, the algorithm takes time proportional to W, which can be much larger than log(W).

Thus, O(W × n) is polynomial in W, not in log(W), making it pseudo-polynomial**.**

Why Does This Matter?

- Pseudo-polynomial algorithms seem efficient but can become infeasible for very large numerical inputs.
- Many NP-hard problems, like Subset Sum and 0/1 Knapsack, have pseudo-polynomial time DP solutions but no known truly polynomial-time solutions.

© Bhanu Jain 2025

The END!

© Bhanu Jain 2025