# 0-1 Knapsack Problem using Dynamic Programming

# General Knapsack Problem

- 0-1 Knapsack Problem
- Problem Analysis
  - Divide and Conquer
  - Dynamic Solution
- Algorithm using Dynamic Programming
- Time Complexity
- Generalization, Variations and Applications
- Conclusion

# 0-1 Knapsack Problem Statement

The knapsack problem arises whenever there is resource allocation with no financial constraints

Problem Statement

- A thief robbing a store and can carry a maximal weight of W into his knapsack. There are n items and $i^{th}$ item weight is $w_i$ and worth is $v_i$ dollars. What items should thief take, not exceeding the bag capacity, to maximize value?

Assumption:

- The items may not be broken into smaller pieces, so thief may decide either to take an item or to leave it, but may not take a fraction of an item.

# 0-1 Knapsack Problem Another Statement

## Problem Statement

- You are in Japan on an official visit and want to make shopping from a store (Best Denki)
- A list of required items is available at the store
- You are given a bag (knapsack), of fixed capacity, and only you can fill this bag with the selected items from the list.
- Every item has a value (cost) and weight,
- And your objective is to seek most valuable set of items which you can buy not exceeding bag limit.

# 0-1 Knapsack Problem: Remarks

## Assumption

- Each item must be put entirely in the knapsack or not included at all that is why the problem is called 0-1 knapsack problem

## Remarks

- Because an item cannot be broken up arbitrarily, so it is its 0-1 property that makes the knapsack problem hard.
- If an item can be broken and allowed to take part of it then algorithm can be solved using greedy approach optimally

# 0-1 Knapsack Problem Construction

## Problem Construction

- You have prepared a list of n objects for which you are interested to buy, The items are numbered as $i_1, i_2, \ldots, i_n$
- Capacity of bag is W
- Each item i has value $v_i$, and weight $w_i$
- We want to select a set of items among $i_1, i_2, \ldots, i_n$ which do not exceed (in total weight) capacity W of the bag
- Total value of selected items must be maximum
- How should we select the items?

# 0-1 Knapsack Problem

## Formal Construction of Problem

- Given a list: $i_1, i_2, \ldots, i_n$, values: $v_1, v_2, \ldots, v_n$ and weights: $w_1, w_2, \ldots, w_n$ respectively
- Of course $W \geq 0$, and we wish to find a set S of items such that $S \subseteq \{i_1, i_2, \ldots, i_n\}$ that

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

subject to $$\sum_{i \in S} w_i \leq W$$

# Brute Force Solution

- Compute all the subsets of $\{i_1, i_2, . . ., i_n\}$, there will be $2^n$ number of subsets.
- Find sum of the weights of total items in each set and list only those sets whose sum does not increase by W (capacity of knapsack)
- Compute sum of values of items in each selected list and find the highest one
- This highest value is the **required solution**
- The computational cost of Brute Force Approach is exponential and not economical
- Find some other way!

# Divide and Conquer Approach

Approach

- Partition the knapsack problem into sub-problems
- Find the solutions of the sub-problems
- Combine these solutions to solve original problem

Comments

- In this case the sub-problems are not independent
- And the sub-problems share sub-sub-problems
- Algorithm repeatedly solves common sub-sub-problems and takes more effort than required
- Because this is an optimization problem and hence dynamic approach is another solution if we are able to construct problem dynamically

# Steps in Dynamic Programming

Step1 (Structure):

- Characterize the structure of an optimal solution
- Next decompose the problem into sub-problems
- Relate structure of the optimal solution of original problem and solutions of sub-problems

Step 2 (Principal of Optimality)

- Define value of an optimal solution recursively
- Then express solution of the main problem in terms of optimal solutions of sub-problems.

# Steps in Dynamic Programming

Step3 (Bottom-up Computation):

- In this step, compute the value of an optimal solution in a bottom-up fashion by using structure of the table already constructed.

Step 4 (Construction of an Optimal Solution)

- Construct an optimal solution from the computed information based on Steps 1-3.

Note:

- Some time people, combine the steps 3 and 4
- Step 1-3 form basis of dynamic problem
- Step 4 may be omitted if only optimal solution of the problem is required

# Mathematical Model: Dynamic Programming

Step1 (Structure):

- Decompose problem into smaller problems
- Construct an array V[0..n, 0..W]
- V[i, w] = maximum value of items selected from {1, 2,. . ., i}, that can fit into a bag with capacity w,
- where 1 < i < n, 1 < w < W
- V[n, W] = contains maximum value of the items selected from {1,2,…,n} that can fit into the bag with capacity W storage
- Hence V[n, W] is the required solution for our knapsack problem

# Mathematical Model: Dynamic Programming

Step 2 (Principal of Optimality)

- Recursively define value of an optimal solution in terms of solutions to sub-problems

Base Case: Since

- $V[0, w] = 0, 0 < w < W$, no items are available
- $V[0, w] = -\infty, w < 0$, invalid
- $V[i, 0] = 0, 0 < i < n$, no capacity available

Recursion:

$$V[i, w] = \max(V[i-1, w], v_i + V[i-1, w - w_i])$$

$$\text{for } 1 < i < n, 0 < w < W$$

# Algorithm : Dynamic Programming

```
KnapSack (v[], w[], n, W)
for (i = 1 to n), V[i, 0] = 0; //Total value with capacity 0
for (j = 0 to W), V[0, j] = 0; //Total value by selecting 0 item
for (i = 1 to n) //selecting 1, 2, 3, …, n items
    for (j = 1 to W) //selecting capacity to be 1, 2, …, W
      if (w(i) ≤ j)
            V[i, j] = max(V[i-1, j], v_i + V[i-1, j – w_i]);
      else
            V[i, j] = V[i-1, j]; //if the i^th item is larger than j
Return V[n, W]
```

Time Complexity O(n.W)

# Developing Algorithm for Knapsack

| i | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| $v_i$ | 10 | 40 | 30 | 50 |
| $w_i$ | 5 | 4 | 6 | 3 |

Capacity = 10

- V[1, 1] = 0,
- V[1, 2] = 0
- V[1, 3] = 0,
- V[1, 4] = 0

- V[i, j] = max(V[i-1, j], $v_i$ + V[i-1, j – $w_i$]);
- V[1, 5] = max(V[0, 5], $v_1$ + V[0, 5 – $w_1$]);

  = max(V[0, 5], 10 + V[0, 5 - 5])

  = max(V[0, 5], 10 + V[0, 0])

  = max(0, 10 + 0) = max(0, 10) = 10

Keep(1, 5) = 1

# Developing Algorithm for Knapsack

- $V[i, j] = \max(V[i-1, j], v_i + V[i-1, j - w_i]);$

- $V[1, 6] = \max(V[0, 6], v_1 + V[0, 6 - w_1]);$

$= \max(V[0, 6], 10 + V[0, 6 - 5])$

$= \max(V[0, 6], 10 + V[0, 1])$

$= \max(0, 10 + 0) = \max(0, 10) = 10,$

$V[1, 7] = \max(V[0, 7], v_1 + V[0, 7 - w_1]);$

$= \max(V[0, 7], 10 + V[0, 7 - 5])$

$= \max(V[0, 7], 10 + V[0, 2])$

$= \max(0, 10 + 0) = \max(0, 10) = 10$

Keep(1, 6) = 1; Keep(1, 7) = 1

# Problem: Developing Algorithm for Knapsack

- $V[i, j] = \max(V[i-1, j], v_i + V[i-1, j - w_i]);$
- $V[1, 8] = \max(V[0, 8], v_1 + V[0, 8 - w_1]);$

  $= \max(V[0, 8], 10 + V[0, 8 - 5])$

  $= \max(V[0, 8], 10 + V[0, 3])$

  $= \max(0, 10 + 0) = \max(0, 10) = 10$
- $V[1, 9] = \max(V[0, 9], v_1 + V[0, 9 - w_1]);$

  $= \max(V[0, 9], 10 + V[0, 9 - 5])$

  $= \max(V[0, 7], 10 + V[0, 4])$

  $= \max(0, 10 + 0) = \max(0, 10) = 10$

Keep(1, 8) = 1; Keep(1, 9) = 1

# Problem: Developing Algorithm for Knapsack

- $V[i, j] = \max(V[i-1, j], v_i + V[i-1, j - w_i]);$

- $V[1, 10] = \max(V[0, 10], v_1 + V[0, 10 - w_1]);$

$= \max(V[0, 10], 10 + V[0, 10 - 5])$

$= \max(V[0, 10], 10 + V[0, 5])$

$= \max(0, 10 + 0) = \max(0, 10) = 10$

Keep(1, 10) = 1;

- $V[2, 1] = 0;$
- $V[2, 2] = 0;$
- $V[2, 3] = 0;$

| | | | | |
|---|---|---|---|---|
| i | 1 | 2 | 3 | 4 |
| $v_i$ | 10 | 40 | 30 | 50 |
| $w_i$ | 5 | 4 | 6 | 3 |

Capacity = 10

# Problem: Developing Algorithm for Knapsack

- $V[i, j] = \max(V[i-1, j], v_i + V[i-1, j - w_i]);$

- $V[2, 4] = \max(V[1, 4], v_2 + V[1, 4 - w_2]);$

  $= \max(V[1, 4], 40 + V[1, 4 - 4])$

  $= \max(V[1, 4], 40 + V[1, 0])$

  $= \max(0, 40 + 0) = \max(0, 40) = 40$

- $V[2, 5] = \max(V[1, 5], v_2 + V[1, 5 - w_2]);$

  $= \max(V[1, 5], 40 + V[1, 5 - 4])$

  $= \max(V[1, 5], 40 + V[1, 1])$

  $= \max(10, 40 + 0) = \max(0, 40) = 40$

Keep(2, 4) = 1; Keep(2, 5) = 1

# Problem: Developing Algorithm for Knapsack

- $V[i, j] = \max(V[i-1, j], v_i + V[i-1, j - w_i]);$

- $V[2, 6] = \max(V[1, 6], v_2 + V[1, 6 - w_2]);$

  $= \max(V[1, 6], 40 + V[1, 6 - 4])$

  $= \max(V[1, 6], 40 + V[1, 2])$

  $= \max(10, 40 + 0) = \max(10, 40) = 40$

- $V[2, 7] = \max(V[1, 7], v_2 + V[1, 7 - w_2]);$

  $= \max(V[1, 7], 40 + V[1, 7 - 4])$

  $= \max(V[1, 7], 40 + V[1, 2])$

  $= \max(10, 40 + 0) = \max(10, 40) = 40$

# Problem: Developing Algorithm for Knapsack

- $V[i, j] = \max(V[i-1, j], v_i + V[i-1, j - w_i]);$

- $V[2, 8] = \max(V[1, 8], v_2 + V[1, 8 - w_2]);$

  $= \max(V[1, 8], 40 + V[1, 8 - 4])$

  $= \max(V[1, 8], 40 + V[1, 4])$

  $= \max(10, 40 + 0) = \max(10, 40) = 40$

- $V[2, 9] = \max(V[1, 9], v_2 + V[1, 9 - w_2]);$

  $= \max(V[1, 9], 40 + V[1, 9 - 4])$

  $= \max(V[1, 9], 40 + V[1, 5])$

  $= \max(10, 40 + 10) = \max(10, 50) = 50$

# Problem: Developing Algorithm for Knapsack

- $V[i, j] = \max(V[i-1, j], v_i + V[i-1, j - w_i]);$
- $V[2, 10] = \max(V[1, 10], v_2 + V[1, 10 - w_2]);$

  $= \max(V[1, 10], 40 + V[1, 10 - 4])$

  $= \max(V[1, 10], 40 + V[1, 6])$

  $= \max(10, 40 + 10) = \max(10, 50) = 50$

- $V[3, 1] = 0;$
- $V[3, 2] = 0;$
- $V[3, 3] = 0;$

| i | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| $v_i$ | 10 | 40 | 30 | 50 |
| $w_i$ | 5 | 4 | 6 | 3 |

Capacity = 10

# Problem: Developing Algorithm for Knapsack

- $V[i, j] = \max(V[i-1, j], v_i + V[i-1, j - w_i]);$

- $V[3, 4] = \max(V[2, 4], v_3 + V[2, 4 - w_3]);$

  $= \max(V[2, 4], 30 + V[2, 4 - 6])$

  $= \max(V[2, 4], 30 + V[2, -2]) = V[2, 4] = 40$

- $V[3, 5] = \max(V[2, 5], v_3 + V[2, 5 - w_3]);$

  $= \max(V[2, 5], 30 + V[2, 5 - 6])$

  $= \max(V[2, 5], 30 + V[2, -1])$

  $= V[2, 5] = 40$

# Problem: Developing Algorithm for Knapsack

- $V[i, j] = \max(V[i-1, j], v_i + V[i-1, j - w_i]);$

- $V[3, 6] = \max(V[2, 6], v_3 + V[2, 6 - w_3]);$

  $= \max(V[2, 6], 30 + V[2, 6 - 6])$

  $= \max(V[2, 6], 30 + V[2, 0])$

  $= \max(V[2, 6], 30 + V[2, 0])$

  $= \max(40, 30) = 40$

- $V[3, 7] = \max(V[2, 7], v_3 + V[2, 7 - w_3]);$

  $= \max(V[2, 7], 30 + V[2, 7 - 6])$

  $= \max(V[2, 7], 30 + V[2, 1])$

  $= \max(V[2, 7], 30 + V[2, 1])$

  $= \max(40, 30) = 40$

# Problem: Developing Algorithm for Knapsack

- $V[i, j] = \max(V[i-1, j], v_i + V[i-1, j - w_i]);$

- $V[3, 8] = \max(V[2, 8], v_3 + V[2, 8 - w_3]);$

  $= \max(V[2, 8], 30 + V[2, 8 - 6])$

  $= \max(V[2, 8], 30 + V[2, 2])$

  $= \max(V[2, 8], 30 + V[2, 2])$

  $= \max(40, 30 + 0) = 40$

- $V[3, 9] = \max(V[2, 9], v_3 + V[2, 9 - w_3]);$

  $= \max(V[2, 9], 30 + V[2, 9 - 6])$

  $= \max(V[2, 9], 30 + V[2, 3])$

  $= \max(V[2, 9], 30 + V[2, 3])$

  $= \max(50, 30 + 0) = 50$

# Problem: Developing Algorithm for Knapsack

- $V[i, j] = \max(V[i-1, j], v_i + V[i-1, j - w_i]);$

- $V[3, 10] = \max(V[2, 10], v_3 + V[2, 10 - w_3]);$

  $= \max(V[2, 10], 30 + V[2, 10 - 6])$

  $= \max(V[2, 10], 30 + V[2, 4])$

  $= \max(V[2, 10], 30 + V[2, 4])$

  $= \max(50, 30 + 40) = 70$

- $V[4, 1] = 0;$
- $V[4, 2] = 0;$

| i | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| $v_i$ | 10 | 40 | 30 | 50 |
| $w_i$ | 5 | 4 | 6 | 3 |

Capacity = 10

# Problem: Developing Algorithm for Knapsack

- $V[i, j] = \max(V[i-1, j], v_i + V[i-1, j - w_i]);$

- $V[4, 3] = \max(V[3, 3], v_4 + V[3, 3 - w_4]);$

  $= \max(V[3, 3], 50 + V[3, 3 - 3])$

  $= \max(V[3, 3], 50 + V[3, 3 - 3])$

  $= \max(V[3, 3], 50 + V[3, 0]) = \max(0, 50) = 50$

- $V[4, 4] = \max(V[3, 4], v_4 + V[3, 4 - w_4]);$

  $= \max(V[3, 4], 50 + V[3, 4 - 3])$

  $= \max(V[3, 4], 50 + V[3, 4 - 3])$

  $= \max(V[3, 4], 50 + V[3, 1])$

  $= \max(40, 50) = 50$

# Problem: Developing Algorithm for Knapsack

- $V[i, j] = \max(V[i-1, j], v_i + V[i-1, j - w_i]);$

- $V[4, 5] = \max(V[3, 5], v_4 + V[3, 5 - w_4]);$

  $= \max(V[3, 5], 50 + V[3, 5 - 3])$

  $= \max(V[3, 5], 50 + V[3, 5 - 3])$

  $= \max(V[3, 5], 50 + V[3, 2])$

  $= \max(40, 50) = 50$

- $V[4, 6] = \max(V[3, 6], v_4 + V[3, 6 - w_4]);$

  $= \max(V[3, 6], 50 + V[3, 6 - 3])$

  $= \max(V[3, 6], 50 + V[3, 6 - 3])$

  $= \max(V[3, 6], 50 + V[3, 3])$

  $= \max(40, 50) = 50$

# Problem: Developing Algorithm for Knapsack

- $V[i, j] = \max(V[i-1, j], v_i + V[i-1, j - w_i]);$
- $V[4, 7] = \max(V[3, 7], v_4 + V[3, 7 - w_4]);$

  $= \max(V[3, 7], 50 + V[3, 7 - 3])$

  $= \max(V[3, 7], 50 + V[3, 7 - 3])$

  $= \max(V[3, 7], 50 + V[3, 4])$

  $= \max(40, 50 + 40) = 90$
- $V[4, 8] = \max(V[3, 8], v_4 + V[3, 8 - w_4]);$

  $= \max(V[3, 8], 50 + V[3, 8 - 3])$

  $= \max(V[3, 8], 50 + V[3, 8 - 3])$

  $= \max(V[3, 8], 50 + V[3, 5])$

  $= \max(40, 50 + 40) = 90$

# Problem: Developing Algorithm for Knapsack

- $V[i, j] = \max(V[i-1, j], v_i + V[i-1, j - w_i]);$
- $V[4, 9] = \max(V[3, 9], v_4 + V[3, 9 - w_4]);$

  $= \max(V[3, 9], 50 + V[3, 9 - 3])$

  $= \max(V[3, 9], 50 + V[3, 9 - 3])$

  $= \max(V[3, 9], 50 + V[3, 6])$

  $= \max(50, 50 + 40) = 90$
- $V[4, 10] = \max(V[3, 10], v_4 + V[3, 10 - w_4]);$

  $= \max(V[3, 10], 50 + V[3, 10 - 3])$

  $= \max(V[3, 10], 50 + V[3, 10 - 3])$

  $= \max(V[3, 10], 50 + V[3, 7])$

  $= \max(70, 50 + 40) = 90;$ Keep(4, 10) = 1

# Optimal Value: Entire Solution

Let W = 10

Final Solution: V[4, 10] = 90

Items selected = {2, 4}

| i | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| $v_i$ | 10 | 40 | 30 | 50 |
| $w_i$ | 5 | 4 | 6 | 3 |

| V[i, w] | W = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| i = 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| i = 1 | 0 | 0 | 0 | 0 | 0 | 10 | 10 | 10 | 10 | 10 | 10 |
| i = 2 | 0 | 0 | 0 | 0 | 40 | 40 | 40 | 40 | 40 | 50 | 50 |
| i = 3 | 0 | 0 | 0 | 0 | 40 | 40 | 40 | 40 | 40 | 50 | 70 |
| i = 4 | 0 | 0 | 0 | 50 | 50 | 50 | 50 | 90 | 90 | 90 | 90 |

# Algorithm : Dynamic Programming

```
KnapSack (v, w, n, W)
for (i = 1 to n), V[i, 0] = 0;
for (j = 0 to W), V[0, j] = 0;
for (i = 1 to n)
      for (j = 1 to W)
         if (w(i) ≤ j)
               V[i, j] = max(V[i-1, j], v_i + V[i-1, j – w_i]);
         else
               V[i, j] = V[i-1, j];
Return V[n, W]
```

Time Complexity O(n.W)

# Elements: Knapsack Algorithm

How do we use all values *keep[i, w]*, to determine a subset S of items having the maximum value?

- If *keep[n, w]* is 1, then $n \in S$, and we can repeat for *keep[n-1, W - $w_n$]*
- If *keep[n, w]* is 0, then $n \notin S$ and we can repeat for *keep[n-1, W]*
- Following is a partial program for this output elements

```
K = W;
for (i = n down to 1)
        if keep[i, K] = = 1
                output i
                K = K – w_i
```

# Constructing Optimal Solution

- $V[i, j] = \max(V[i-1, j], v_i + V[i-1, j - w_i]);$
- i = 4

  V[4, 10] = max(70, 50 + 40) = 90; Keep(4, 10) = 1
- i = 3

  V[3, 10 - 3] = V[3, 7] = max(40, 30) = 40 Keep(3, 7) = 0
- i = 2

  V[2, 7] = max(10, 40) = 40 Keep(2, 7) = 1
- i = 1

  V[1, 7-4] = V[1, 3] = 0 Keep(1, 3) = 0

# Complete: Dynamic Programming Algorithm

```
KnapSack(v, w, n, W)
for (w = 0 to W), V[0, w] = 0;  for (i = 1 to n), V[i, 0] = 0;
for (i = 1 to n)
        for (w = 1 to W)
            if ((w(i) ≤ w) and (v_i + V[i-1,w – w_i] > V[i-1,w]))
                    V[i, w] = (v_i + V[i-1,w – w_i];
                    keep[i, w] = 1;
            else
                    V[i, w] = V[i-1,w];
                    keep[i, w] = 0;

K = W;
            for (i = n down to 1)
                    if keep[i, K] = = 1
                            output i
                            K = K – w_i
Return V[n, W]
```