Algorithms Analysis and Design

Chapter 7

Dynamic Programming
Part 2

0/1 knapsack problem

Knapsack problem

- **Recall:** There are two versions of the problem:

1) 0/1 knapsack problem
2) Fractional knapsack problem.

1) Items are indivisible; you either take an item or not. Solved with dynamic programming.
2) Items are divisible: you can take any fraction of an item. Solved with a greedy algorithm.

0/1 Knapsack problem

- Thief has a knapsack with maximum capacity W, and a set S consisting of n items
- Each item i has some weight w_i and benefit value v_i (all w_i, v_i and W are integer values) Integer Knapsack problem
- Problem: How to pack the knapsack to achieve maximum total value of packed items?
- Goal:
  - find x_i such that for all $x_i = \{0, 1\}$, $i = 1, 2, .., n$

    $\sum w_i x_i \leq W$ and

    $\sum x_i v_i$ is maximum

The constraint here is we can either put an item completely into the bag or cannot put it at all [It is not possible to put a part of an item into the bag]

0/1 Knapsack problem

- Example:

Input: *N = 3, W = 4, v[] = {1, 2, 3}, weight[] = {4, 5, 1}*

- *There are two items which have weight less than or equal to 4.*
- *If we select the item with weight 4, the possible profit is 1.*
- *And if we select the item with weight 1, the possible profit is 3.*
- *So the maximum possible profit is 3.*

Note that we cannot put both the items with weight 4 and 1 together as the capacity of the bag is 4.

0/1 Knapsack problem: brute force and greedy approaches

- Recall:
- Brute-force approach : Running time will be O(2^n) Exponential Time!
- Greedy approach: Does not guarantee the optimal solution

A simple solution is to consider all subsets of items and calculate the total weight and profit of all subsets. Consider the only subsets whose total weight is smaller than W. From all such subsets, pick the subset with maximum profit.

Solved with dynamic programming.

Top Down [Recursion Approach]

Optimal substructure

> ***Optimal substructure***
>
> ***An optimal solution to a problem (instance) contains optimal solutions to subproblems.***

In the 0/1 knapsack problem, the optimal substructure refers to the fact that the maximum value achievable for a given capacity and a subset of items can be obtained by considering the optimal solutions for smaller capacities and subsets of items.

Top Down [Recursion Approach]

Optimal substructure

To consider all subsets of items, there can be two cases for every item.

Case 1: The item is included in the optimal subset.

Case 2: The item is not included in the optimal set.

The maximum value obtained from 'N' items is the max of the following two values.

- ✓ Maximum value obtained by N-1 items and W weight (excluding n^{th} item)
- ✓ Value of nth item plus maximum value obtained by N-1 items and (W – weight of the Nth item) [including N^{th} item].
- ✓ If the weight of the 'N^{th}' item is greater than 'W', then the Nth item cannot be included and **Case 1** is the only possibility.

Top Down [Recursion Approach]

Optimal substructure

The maximum value obtained from 'N' items is the max of the following two values.

- ✓ Maximum value obtained by N-1 items and W weight (excluding n^{th} item)
- ✓ Value of nth item plus maximum value obtained by N-1 items and (W – weight of the Nth item) [including N^{th} item].
- ✓ If the weight of the 'N^{th}' item is greater than 'W', then the Nth item cannot be included and **Case 1** is the only possibility.

Top Down [Recursion Approach]

```
Algorithm knapsack(W, w, v, n):
    if n == 0 or W == 0:
       return 0                          // Base Case

    if w[n] > W:
       return knapsack(W, w, v, n-1)    // the item is not included

    else:
      profit1 = knapsack(W, w, val, n-1)
      profit2 = v[n] + knapsack(W-w[n], w, v, n-1)
      return max(profit1, profit2)
```

// Return the maximum of two cases:
// (1) nth item not included
// (2) included

The function knapsack takes four parameters:

- **W**: the maximum capacity of the knapsack.
- **w** : an array of item weights
- **v** : an array of item values
- **n** :the number of items.

Notes:

- n=0 there are no more items
- W=0 no remaining capacity

Why this approach is inefficient???

Top Down [Recursion Approach]

```
Algorithm knapsack(W, w, v, n):
    if n == 0 or W == 0:
        return 0                              // Base Case

    if w[n] > W:
        return knapsack(W, w, v, n-1)    // the item is not included

    else:
       profit1 = knapsack(W, w, val, n-1)
       profit2 = v[n] + knapsack(W-w[n], w, v, n-1)
       return max(profit1, profit2)
```

// Return the maximum of two cases:
// (1) nth item not included
// (2) included

Can you derive the running time???

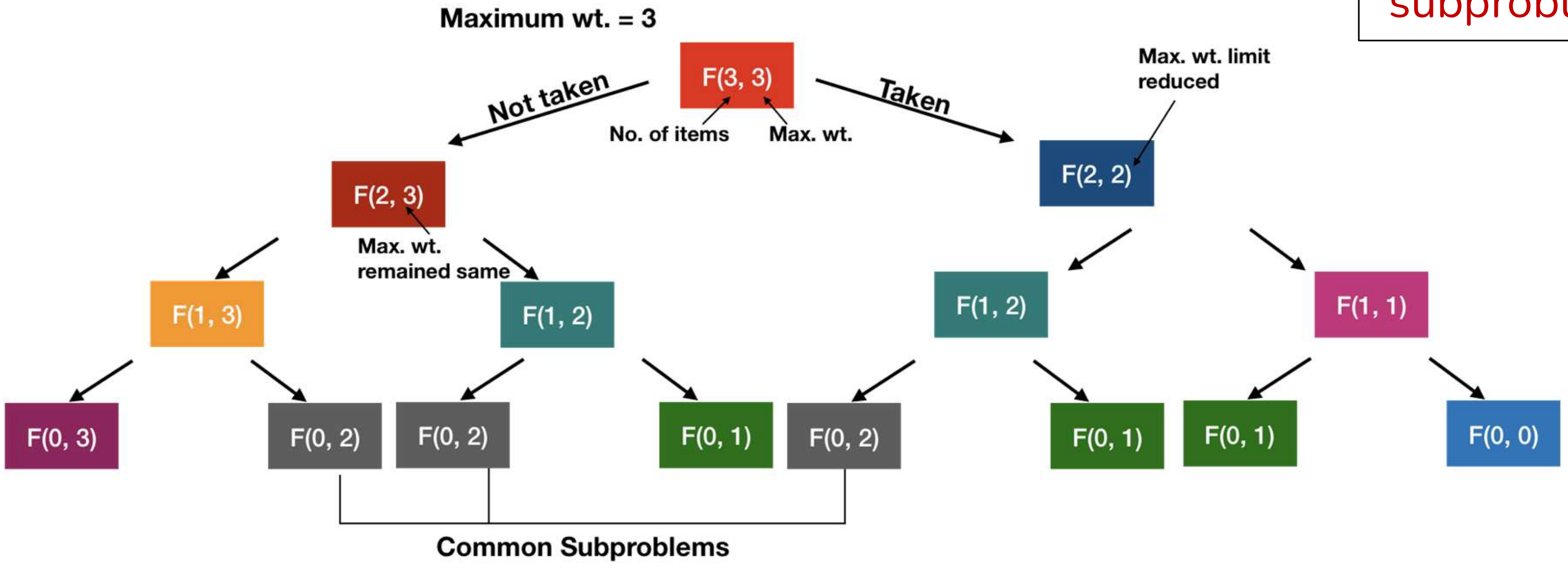
Top Down [Recursion Approach]

Overlapping subproblems ??

Time Complexity: O(2^N)

The algorithms using recursion computes the same subproblems again and again

i	1	2	3
Wi	1	1	1
Vi	100	200	50



Optimized Top Down [Recursion with memoization]

```
Algorithm knapsack(W, w, v, n, memo):
    if n == 0 or W == 0:
        return 0
    if memo[n][W] !=-1:           // Check if the current subproblem is already computed
        return memo[n][W]         // If so, return the memorized result directly

   if w[n] > W:
        memo[n][W] = knapsack(W, w, v, n-1)      // Store the calculated result
        return memo[n][W]
    else:
      profit1 = knapsack(W, w, val, n-1)
      profit2 = v[n] + knapsack(W-w[n], w, v, n-1)
      memo[n][W] = max(profit1, profit2)         // Store the calculated result
      return memo[n][W]
```

- **memo**: The memorization table.

Time Complexity: O(N * W). As redundant calculations are avoided.

Top Down [Recursion Approach]

- P(i, w) : The maximum profit that can be obtained from items 1 to i, if the knapsack has size w.

if $w_i > w$ P(i, w) = $P[i-1, w]$ (Will not select the item)

Otherwise, we have two choices:

- Case 1: thief takes item i

  $P(i, w) = v_i + P(i - 1, w - w_i)$

- Case 2: thief does not take item i

  $P(i, w) = P(i - 1, w)$

0/1 Knapsack problem: DP approach

Since subproblems are evaluated again, this problem has Overlapping Subproblems property. So the 0/1 Knapsack problem has both properties of a dynamic programming problem (Overlapping Subproblems, and optimal substructure). Like other typical Dynamic Programming(DP) problems, re-computation of the same subproblems can be avoided by **constructing table** (array) in a bottom-up manner.

0/1 Knapsack problem: DP approach

- $P(i, w)$: The maximum profit that can be obtained from items 1 to i, if the knapsack has size w.

if $w_i > w$ $P(i, w) = P[i-1, w]$ (Will not select the item)

Otherwise, we have two choices:

- Case 1: thief takes item i

$$P(i, w) = v_i + P(i - 1, w - w_i)$$

- Case 2: thief does not take item i

$$P(i, w) = P(i - 1, w)$$

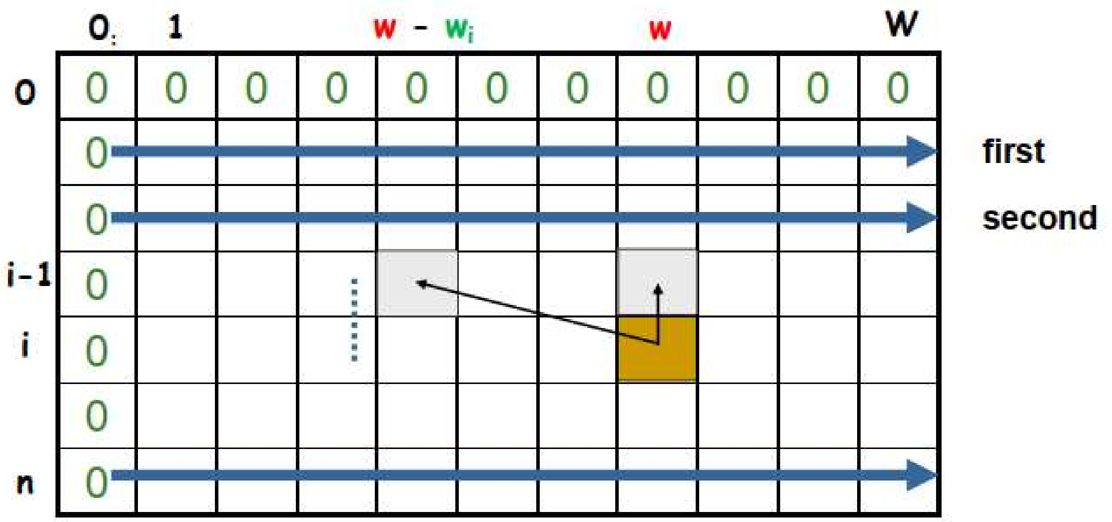
Recursive formula

$$P[i,w] = \begin{cases} P[i-1,w] & \text{if } w_i > w \\ \max\{v_i + P[i-1, w-w_k], P[i-1,w]\} & \text{else} \end{cases}$$

- The best subset that has the total weight *w,* either contains item *i* or not.
- First case: $w_i > w$. Item i can't be part of the solution, since if it was, the total weight would be $> w$, which is unacceptable
- Second case: $w_i \le w$. Then the item *i* can be in the solution, and we choose the case with greater value.

0/1 Knapsack problem: Dynamic Programming

$$P(i, w) = \max \{\underbrace{v_i + P(i-1, w-w_i)}_{\text{Item } i \text{ was taken}}, \underbrace{P(i-1, w)}_{\text{Item } i \text{ was not taken}}\}$$



Example

Apply the **bottom-up dynamic programming** algorithm to the following instance of the knapsack problem:

W (Capacity) = 6

w	1	2	3
v	10	15	40

P(i,w): The maximum profit that can be obtained from items 1 to i, if the knapsack has size w.

- *If no element is filled, then the possible profit is 0.*
- *if no capacity remaining, then the possible profit is 0*

	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0						
2	0						
3	0						

Example

Apply the **bottom-up dynamic programming** algorithm to the following instance of the knapsack problem:

W (Capacity) = 6

w	1	2	3
v	10	15	40

P(i,w): The maximum profit that can be obtained from items 1 to i, if the knapsack has size w.

- *Fill the first item*

	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	10	10	10	10	10	10
2	0						
3	0						

Example

Apply the **bottom-up dynamic programming** algorithm to the following instance of the knapsack problem:

W (Capacity) = 6

w	1	2	3
v	10	15	40

P(i,w): The maximum profit that can be obtained from items 1 to i, if the knapsack has size w.

- *Fill the second item considering items 1 and 2*

	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	10	10	10	10	10	10
2	0	10	15	25	25	25	25
3	0						

Example

Apply the **bottom-up dynamic programming** algorithm to the following instance of the knapsack problem:

W (Capacity) = 6

w	1	2	3
v	10	15	40

P(i,w): The maximum profit that can be obtained from items 1 to i, if the knapsack has size w.

- *Fill the third item considering items 1 , 2 and 3*

	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	10	10	10	10	10	10
2	0	10	15	25	25	25	25
3	0	10	15	40	50	55	65

Maximum profit P(3,6) = 65

Solution X = 1 1 1

Reconstructing the optimal solution

- Start at P(n, W)
- When you go left-up ⇒ item i has been taken
- When you go straight up ⇒ item i has not been taken

Maximum profit P(3,6) = 65

Solution X = 1 1 1

	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	10	10	10	10	10	10
2	0	10	15	25	25	25	25
3	0	10	15	40	50	55	65

Example

Apply the **bottom-up dynamic programming** algorithm to the following instance of the knapsack problem:

W (Capacity) = ٥

w	٢	١	٣	٢
v	١٢	١٠	٢٠	١٥

P(i,w): The maximum profit that can be obtained from items 1 to i, if the knapsack has size w.

Maximum profit P(3,6) = 37

Solution X = 1 1 0 1

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
٤	0	10	15	25	30	37

Example

Apply the **bottom-up dynamic programming** algorithm to the following instance of the knapsack problem:

W (Capacity) = ٥

w	٢	١	٣	٢
v	١٢	١٠	٢٠	١٥

P(i,w): The maximum profit that can be obtained from items 1 to i, if the knapsack has size w.

	0	1	2	3	4	5
0						
1						
2						
3						
٤						

Example

W = 5

Item	Weight	Value
1	2	12
2	1	10
3	3	20
4	2	15

$$P(i, w) = \max \{v_i + P(i-1, w-w_i), P(i-1, w)\}$$

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0	10	15	25	30	37

P(1, 1) = P(0, 1) = 0

P(1, 2) = max{12+0, 0} = 12

P(1, 3) = max{12+0, 0} = 12

P(1, 4) = max{12+0, 0} = 12

P(1, 5) = max{12+0, 0} = 12

P(2, 1)= max{10+0, 0} = 10

P(2, 2)= max{10+0, 12} = 12

P(2, 3)= max{10+12, 12} = 22

P(2, 4)= max{10+12, 12} = 22

P(2, 5)= max{10+12, 12} = 22

P(3, 1)= P(2,1) = 10

P(3, 2)= P(2,2) = 12

P(3, 3)= max{20+0, 22}=22

P(3, 4)= max{20+10,22}=30

P(4, 5)= max{20+12,22}=32

P(4, 1)= P(3,1) = 10

P(4, 2)= max{15+0, 12} = 15

P(4, 3)= max{15+10, 22}=25

P(4, 4)= max{15+12, 30}=30

P(4, 5)= max{15+22, 32}=37

Example

W = 5

$$P(i, w) = \max\{v_i + P(i-1, w-w_i), P(i-1, w)\}$$

Item	Weight	Value
1	2	12
2	1	10
3	3	20
4	2	15

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0	10	15	25	30	37

P(1, 1) = P(0, 1) = 0

P(1, 2) = max{12+0, 0} = 12

P(1, 3) = max{12+0, 0} = 12

P(1, 4) = max{12+0, 0} = 12

P(1, 5) = max{12+0, 0} = 12

P(2, 1)= max{10+0, 0} = 10

P(2, 2)= max{10+0, 12} = 12

P(2, 3)= max{10+12, 12} = 22

P(2, 4)= max{10+12, 12} = 22

P(2, 5)= max{10+12, 12} = 22

P(3, 1)= P(2,1) = 10

P(3, 2)= P(2,2) = 12

P(3, 3)= max{20+0, 22}=22

P(3, 4)= max{20+10,22}=30

P(4, 5)= max{20+12,22}=32

P(4, 1)= P(3,1) = 10

P(4, 2)= max{15+0, 12} = 15

P(4, 3)= max{15+10, 22}=25

P(4, 4)= max{15+12, 30}=30

P(4, 5)= max{15+22, 32}=37

Reconstructing the optimal solution

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0	10	15	25	30	37

- Item 4
- Item 2
- Item 1

- Start at P(n, W)
- When you go left-up $\Rightarrow$ item i has been taken
- When you go straight up $\Rightarrow$ item i has not been taken

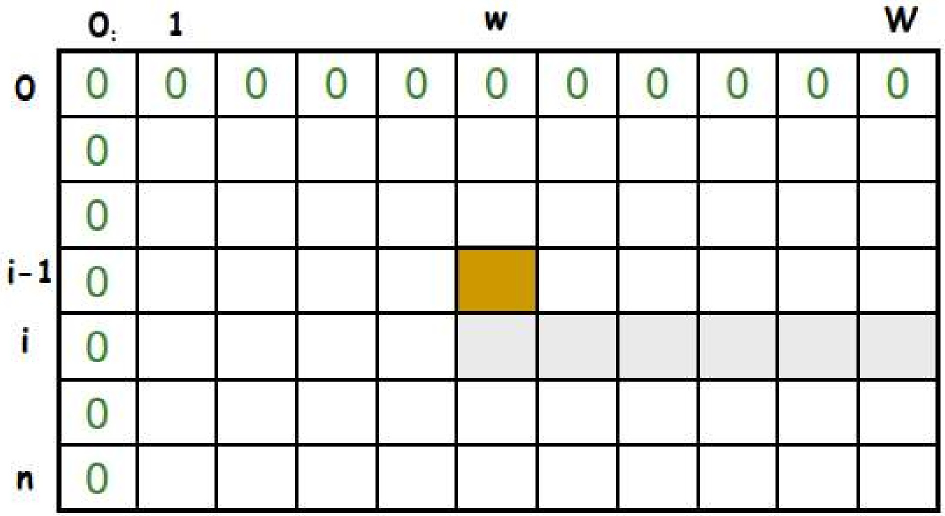
Optimal substructure

- Consider the most valuable load that weights at most W pounds
- If we remove item j from this load

$\Rightarrow$ The remaining load must be the most valuable load weighing at most $W - w_j$ that can be taken from the remaining $n - 1$ items

Overlapping subproblems

$$P(i, w) = \max \{v_i + P(i-1, w-w_i), P(i-1, w)\}$$



E.g.: all the subproblems shown in grey may depend on P(i-1, w)

0/1 knapsack DP algorithm

```
for w = 0 to W
   P[0,w] = 0
for i = 0 to n
   P[i,0] = 0
   for w = 0 to W
      if w_i <= w // item i can be part of the solution
         if v_i + P[i-1,w-w_i] > P[i-1,w]
            P[i,w] = v_i + P[i-1,w- w_i]
         else
            P[i,w] = P[i-1,w]
      else P[i,w] = P[i-1,w]  // w_i > w
```

O(n*W)

Exercise:

Run the 0/1 knapsack DP algorithm on the following data:

n = 4 (# of elements)
W = 5 (max weight)
Elements (weight, benefit):
(2,3), (3,4), (4,5), (5,6)

Exercise:

a. Apply the bottom-up dynamic programming algorithm to the following instance of the knapsack problem:

item	weight	value
1	3	$25
2	2	$20
3	1	$15
4	4	$40
5	5	$50

capacity $W = 6$.

b. How many different optimal subsets does the instance of part (a) have?

Research topics

- Finding the optimal string editing
- The version of the knapsack problem in which there are unlimited quantities of copies for each of the n item kinds given.
- Coin-row problem.
- Coin change-making problem.
- Matrix-chain multiplication
- Others ….