0/1 Knapsack Problem

Dr. Bibhudatta Sahoo
Communication & Computing Group
Department of CSE, NIT Rourkela
Email: bdsahu@nitrkl.ac.in, 9937324437, 2462358

The Knapsack Problem

A thief breaks into a museum. Fabulous paintings, sculptures, and jewels are everywhere. The thief has a good eye for the value of these objects, and knows that each will fetch hundreds or thousands of dollars on the clandestine art collector's market. But, the thief has only brought a single knapsack to the scene of the robbery, and can take away only what he can carry. What items should the thief take to maximize the haul?

Knapsack problem; introduction

- The knapsack problem is well known in operations research literature. This problem arises whenever there is a resource allocation problem with financial constraints.
- For example, given that you have a fixed budget, how do you select what things you should buy? Assume that everything has a cost and value. We seek assignments that provide the most value for a given budget.
- The term *knapsack problem* invokes the image of a backpacker who is constrained by a fixed-size knapsack and so must fill it only with the most useful items.

Knapsack problem; introduction

- The classical knapsack problem assumes that each item must be put entirely in the knapsack or not included at all. It is this 0/1 property that makes the knapsack problem difficult.
- In the 0/1 Knapsack problem, you are given a bag which can hold W kg. There is an array of items each with a different weight and price value assigned to them.
- The objective of the 0/1 Knapsack is to maximize the value you put into the bag. You are allowed to only take all or nothing of a given item.

0-1 Knapsack Problem

Informal Description: We have n items. Let v_i denote the value of the i-th item, and let w_i denote the weight of the i-th item. Suppose you are given a knapsack capable of holding total weight W.

Our goal is to use the knapsack to carry items, such that the total values are maximum; we want to find a subset of items to carry such that

- The total weight is at most W.
- The total value of the items is as large as possible.

0-1 Knapsack Problem

Formal description:

Given $W > 0$, and two n-tuples of positive numbers

$$\langle v_1, v_2, \ldots, v_n \rangle \quad \text{and} \quad \langle w_1, w_2 \ldots, w_n \rangle,$$

we wish to determine the subset $T \subseteq \{1, 2, \ldots, n\}$ (of items to carry) that

$$\text{maximizes} \quad \sum_{i \in T} v_i,$$

$$\text{subject to} \quad \sum_{i \in T} w_i \leq W.$$

Remark: This is an optimization problem. The *Brute Force* solution is to try all 2^n possible subsets T.

General Schema of a DP Solution

Step1: Structure: Characterize the structure of an optimal solution by showing that it can be decomposed into *optimal* subproblems

Step2: Recursively define the value of an optimal solution by expressing it in terms of optimal solutions for smaller problems (usually using min and/or max).

Step 3: Bottom-up computation: Compute the value of an optimal solution in a bottom-up fashion by using a table structure.

Step 4: Construction of optimal solution: Construct an optimal solution from computed information.

Developing a DP Algorithm for Knapsack

Step 1: Decompose the problem into smaller problems.

We construct an array $V[0..n, 0..W]$.
For $1 \leq i \leq n$, and $0 \leq w \leq W$, the entry $V[i, w]$ will store the maximum (combined) value of any subset of items $\{1, 2, \ldots, i\}$ of (combined) weight at most w.

That is

$$V[i, w] = \max \left\{ \sum_{j \in T} v_j \ : \ T \subseteq \{1, 2, \ldots, i\}, \sum_{j \in T} w_j \leq w \right\}.$$

If we can compute all the entries of this array, then the array entry $V[n, W]$ will contain the solution to our problem.

Note: In what follows we will say that T is a *solution* for $[i, w]$ if $T \subseteq \{1, 2, \ldots, i\}$ and $\sum_{j \in T} w_j \leq w$ and that T is an *optimal solution* for $[i, w]$ if T is a solution and $\sum_{j \in T} v_j = V[i, w]$.

Developing a DP Algorithm for Knapsack

Step 2: Recursively define the value of an optimal solution in terms of solutions to smaller problems.

Initial Settings: Set

$$V[0,w] = 0 \quad \text{for } 0 \le w \le W, \quad \text{no item}$$
$$V[i,w] = -\infty \quad \text{for } w < 0, \quad \text{illegal}$$

Recursive Step: Use

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

for $1 \le i \le n, 0 \le w \le W$.

Intuitively, an optimal solution would either choose item i is or not choose item i.

Developing a DP Algorithm for Knapsack

Step 3: Bottom-up computation of $V[i, w]$ (using iteration, not recursion).

Bottom: $V[0, w] = 0$ for all $0 \leq w \leq W$.

Bottom-up computation: Computing the table using

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

row by row.

V[i,w]	w=0	1	2	3	...	...	W
i= 0	0	0	0	0	...	...	0
1	→						
2	→						
⋮	→						
n	→						

bottom

↓

up

Example of the Bottom-up computation

Let $W = 10$ and

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

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

Remarks:

- The final output is $V[4, 10] = 90$.
- The method described does not tell which subset gives the optimal solution. (It is $\{2, 4\}$ in this example).

Let $V[i, j]$ denote the maximum value of the objects that fit in the knapsack, selecting objects from 1 through i with the sack's weight capacity equal to j.

To find $V[i, j]$ we have two choices concerning the decisions made on object i (in the optimal solution for $V[i, j]$): We can either ignore object i or we can include object i.

In the **former case**, the optimal solution of $V[i, j]$ is identical to the optimal solution to using objects 1 though $i - 1$ with sack's capacity equal to W (by the definition of the V notation) .

In the **latter case**, the parts of the optimal solution of $V[i, j]$ concerning the choices made to objects 1 through $i - 1$, must be an optimal solution to $V[i - 1, j - w_i]$, an application of the principle of optimality Thus, we have derived the following recurrence for $V[i, j]$:

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

The boundary conditions are $V[0, j] = 0$ if $j \geq 0$, and $V[i, j] = -\infty$ when $j < 0$.

The problem can be solved using dynamic programming (i.e., a bottom-up approach to carrying out the computation steps) based on a tabular form when the weights are integers.

Example: There are $n = 5$ objects with integer weights $w[1..5] = \{1,2,5,6,7\}$, and values $v[1..5] = \{1,6,18,22,28\}$. The following table shows the computations leading to $V[5,11]$ (i.e., assuming a knapsack capacity of 11).

Time: O(nW)
space: O(W)

Sack's capacity		0	1	2	3	4	5	6	7	8	9	10	11
wi	*vi*												
1	1	0	1	1	1	1	1	1	1	1	1	1	1
2	6	0	1	6	7	7	7	7	7	7	7	7	7
5	18	0	1	6	7	7	18	19	24	25	25	25	25
6	22	0	1	6	7	7	18	22	24	28	29	29	40
7	28	0	1	6	7	7	18	22	28	29	34	35	**40**

$V[3, 8]$

$V[4, 8] = \max(V[3, 8], 22 + V[3, 2])$

$V[3, 8 - w_4] = V[3, 2]$

Knapsack Problem by DP (pseudocode)

1. Algorithm DPKnapsack($w[1..n]$, $v[1..n]$, W)
2. var $V[0..n,0..W]$, $P[1..n,1..W]$: int
3. for $j := 0$ to W do
4. $V[0,j] := 0$
5. for $i := 0$ to n do
6. $V[i,0] := 0$
7. for $i := 1$ to n do
8. for $j := 1$ to W do
9. if $w[i] \leq j$ and $v[i] + V[i-1,j-w[i]] > V[i-1,j]$ then
10. **$V[i,j] := v[i] + V[i-1,j-w[i]]$; $P[i,j] := j-w[i]$**
11. else
12. $V[i,j] := V[i-1,j]$; $P[i,j] := j$
13. **return $V[n, W]$ and the optimal subset by backtracing**

The Dynamic Programming Algorithm

```
KnapSack(v, w, n, W)
{
  for (w = 0 to W) V[0, w] = 0;
  for (i = 1 to n)
    for (w = 0 to W)
      if (w[i] ≤ w)
        V[i, w] = max{V[i − 1, w], v[i] + V[i − 1, w − w[i]]};
      else
        V[i, w] = V[i − 1, w];
  return V[n, W];
}
```

Time complexity: Clearly, $O(nW)$.

Suggested questions

- Write an dynamic programming algorithm to find optimal solution to 0-1 knapsack problem for n items with time complexity $O(nW)$; for the knapsack capacity W.
- Write an dynamic programming algorithm to find a solution to 0-1 knapsack problem that yields maximum profit P for n items with the knapsack capacity W.

Example

Let's run our algorithm on the following data:

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

Example (2)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1						
2						
3						
4						

for w = 0 to W

V[0,w] = 0

Example (3)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0					
2	0					
3	0					
4	0					

for i = 1 to n

V[i,0] = 0

Example (4)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0				
2	0					
3	0					
4	0					

i=1

b_i=3

w_i=2

w=1

$w-w_i$ =-1

if w_i <= w // item i can be part of the solution
    if $b_i + V[i-1,w-w_i] > V[i-1,w]$
        $V[i,w] = b_i + V[i-1,w- w_i]$
    else
        $V[i,w] = V[i-1,w]$
else $V[i,w] = V[i-1,w]$ // $w_i > w$

Example (5)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3			
2	0					
3	0					
4	0					

i=1

b_i=3

w_i=2

w=2

w-w_i =0

if w_i <= w // item i can be part of the solution
    if b_i + V[i-1,w-w_i] > V[i-1,w]
        V[i,w] = b_i + V[i-1,w- w_i]
    else
        V[i,w] = V[i-1,w]
else V[i,w] = V[i-1,w] // w_i > w

Example (6)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3		
2	0					
3	0					
4	0					

i=1

b_i=3

w_i=2

w=3

w-w_i =1

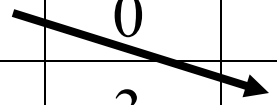
if w_i <= w // item i can be part of the solution
    if b_i + V[i-1,w-w_i] > V[i-1,w]
        V[i,w] = b_i + V[i-1,w- w_i]
    else
        V[i,w] = V[i-1,w]
else V[i,w] = V[i-1,w] // w_i > w

Example (7)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	
2	0					
3	0					
4	0					



i=1

b_i=3

w_i=2

w=4

w-w_i =2

if w_i <= w // item i can be part of the solution

    if b_i + V[i-1,w-w_i] > V[i-1,w]

        V[i,w] = b_i + V[i-1,w- w_i]

    else

        V[i,w] = V[i-1,w]

else V[i,w] = V[i-1,w] // w_i > w

Example (8)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0					
3	0					
4	0					

i=1

b_i=3

w_i=2

w=5

w-w_i =3

if w_i <= w // item i can be part of the solution
    if b_i + V[i-1,w-w_i] > V[i-1,w]
        V[i,w] = b_i + V[i-1,w- w_i]
    else
        V[i,w] = V[i-1,w]
else V[i,w] = V[i-1,w] // w_i > w

Example (9)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0				
3	0					
4	0					

i=2

b_i=4

w_i=3

w=1

$w-w_i$ =-2

if w_i <= w // item i can be part of the solution
    if $b_i + V[i-1,w-w_i] > V[i-1,w]$
        $V[i,w] = b_i + V[i-1,w- w_i]$
    else
        V[i,w] = V[i-1,w]
else V[i,w] = V[i-1,w] // $w_i > w$

Example (10)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3			
3	0					
4	0					

i=2

b_i=4

w_i=3

w=2

w-w_i =-1

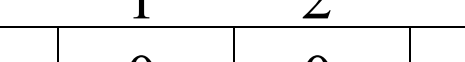
if w_i <= w // item i can be part of the solution
    if b_i + V[i-1,w-w_i] > V[i-1,w]
        V[i,w] = b_i + V[i-1,w- w_i]
    else
        V[i,w] = V[i-1,w]
else V[i,w] = V[i-1,w] // w_i > w

Example (11)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4		
3	0					
4	0					



$i=2$

$b_i=4$

$w_i=3$

$w=3$

$w-w_i=0$

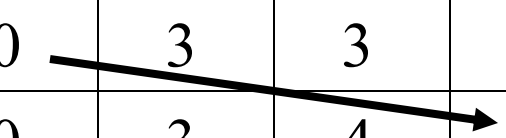
```
if w_i <= w // item i can be part of the solution
        if b_i + V[i-1,w-w_i] > V[i-1,w]
            V[i,w] = b_i + V[i-1,w- w_i]
        else
            V[i,w] = V[i-1,w]
else V[i,w] = V[i-1,w]  // w_i > w
```

Example (12)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	
3	0					
4	0					



i=2

b_i=4

w_i=3

w=4

w-w_i =1

if w_i <= w // item i can be part of the solution
    if b_i + V[i-1,w-w_i] > V[i-1,w]
        V[i,w] = b_i + V[i-1,w- w_i]
    else
        V[i,w] = V[i-1,w]
else V[i,w] = V[i-1,w] // w_i > w

Example (13)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0					
4	0					

i=2

b_i=4

w_i=3

w=5

w-w_i =2

if w_i <= w // item i can be part of the solution
    if b_i + V[i-1,w-w_i] > V[i-1,w]
        V[i,w] = b_i + V[i-1,w- w_i]
    else
        V[i,w] = V[i-1,w]
else V[i,w] = V[i-1,w] // w_i > w

Example (14)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4		
4	0					

i=3

b_i=5

w_i=4

w= 1..3

```
if w_i <= w // item i can be part of the solution
        if b_i + V[i-1,w-w_i] > V[i-1,w]
            V[i,w] = b_i + V[i-1,w- w_i]
        else
            V[i,w] = V[i-1,w]
else V[i,w] = V[i-1,w]  // w_i > w
```

Example (15)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	
4	0					

i=3

b_i=5

w_i=4

w= 4

w- w_i=0

if w_i <= w // item i can be part of the solution
    if b_i + V[i-1,w-w_i] > V[i-1,w]
        V[i,w] = b_i + V[i-1,w- w_i]
    else
        V[i,w] = V[i-1,w]
else V[i,w] = V[i-1,w] // w_i > w

Example (16)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0					

i=3

b_i=5

w_i=4

w= 5

w- w_i=1

if w_i <= w // item i can be part of the solution
    if b_i + V[i-1,w-w_i] > V[i-1,w]
        V[i,w] = b_i + V[i-1,w- w_i]
    else
        V[i,w] = V[i-1,w]
else V[i,w] = V[i-1,w] // w_i > w

Example (17)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	

i=4

b_i=6

w_i=5

w= 1..4

```
if w_i <= w // item i can be part of the solution
        if b_i + V[i-1,w-w_i] > V[i-1,w]
            V[i,w] = b_i + V[i-1,w- w_i]
        else
            V[i,w] = V[i-1,w]
else V[i,w] = V[i-1,w]  // w_i > w
```

Example (18)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

$i=4$

$b_i=6$

$w_i=5$

$w= 5$

$w- w_i=0$

```
if w_i <= w // item i can be part of the solution
        if b_i + V[i-1,w-w_i] > V[i-1,w]
            V[i,w] = b_i + V[i-1,w- w_i]
        else
            V[i,w] = V[i-1,w]
else V[i,w] = V[i-1,w]  // w_i > w
```

Exercise

1. 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 $W = 6$.

Comments

- This algorithm only finds the max possible value that can be carried in the knapsack
  - i.e., the value in V[n,W]
- To know the items that make this maximum value, an addition to this algorithm is necessary

Constructing the Optimal Solution

- The algorithm for computing $V[i, w]$ described in the previous slide does not record which subset of items gives the optimal solution.

- To compute the actual subset, we can add an auxiliary boolean array $keep[i, w]$ which is 1 if we decide to take the i-th file in $V[i, w]$ and 0 otherwise.

Question: How do we use all the values $keep[i, w]$ to determine the subset T of files having the maximum computing time?

Constructing the Optimal Solution

Question: How do we use the values $keep[i, w]$ to determine the subset T of items having the maximum computing time?

If keep$[n, W]$ is 1, then $n \in T$. We can now repeat this argument for keep$[n-1, W-w_n]$.
If keep$[n, W]$ is 0, the $n \notin T$ and we repeat the argument for keep$[n-1, W]$.

Therefore, the following partial program will output the elements of T:

```
K = W;
for (i = n downto 1)
    if (keep[i, K] == 1)
    {
        output i;
        K = K - w[i];
    }
```

The Complete Algorithm for the Knapsack

```
KnapSack(v, w, n, W)
{
    for (w = 0 to W) V[0, w] = 0;
    for (i = 1 to n)
        for (w = 0 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 downto 1)
        if (keep[i, K] == 1)
        {
            output i;
            K = K − w[i];
        }
    return V[n, W];
}
```

How to find actual Knapsack Items

- All of the information we need is in the table.
- $V[n,W]$ is the maximal value of items that can be placed in the Knapsack.
- Let i=n and k=W

if $V[i,k] \neq V[i-1,k]$ then

mark the i^{th} item as in the knapsack

$i = i-1$, $k = k\text{-}w_i$

else

$i = i-1$ // Assume the i^{th} item is <u>not</u> in the knapsack

// Could it be in the optimally packed knapsack?

Finding the Items

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

i=4

k= 5

b_i=6

w_i=5

$V[i,k] = 7$

$V[i-1,k] = 7$

i=n, k=W

while i,k > 0

if $V[i,k] \neq V[i-1,k]$ then

mark the i^{th} item as in the knapsack

$i = i-1$, $k = k-w_i$

else

$i = i-1$

Finding the Items (2)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

i=4

k= 5

b_i=6

w_i=5

V[i,k] = 7

V[i–1,k] =7

i=n, k=W

while i,k > 0

if *V[i,k]* ≠ *V[i–1,k]* then

mark the i^{th} item as in the knapsack

$i = i–1$, $k = k\text{-}w_i$

else

$i = i–1$

Finding the Items (3)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

i=3

k= 5

b_i=5

w_i=4

V[i,k] = 7

V[i–1,k] =7

i=n, k=W

while i,k > 0

if *V[i,k]* ≠ *V[i–1,k]* then

mark the i^{th} item as in the knapsack

$i = i–1$, $k = k\text{-}w_i$

else

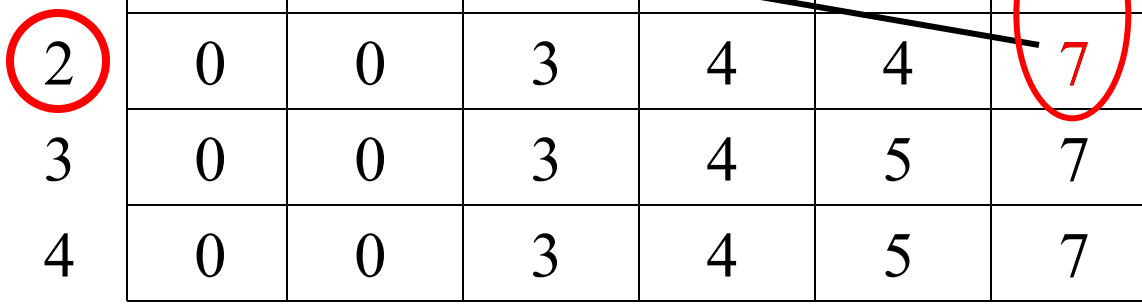
$i = i–1$

Finding the Items (4)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7



i=2

k= 5

b_i=4

w_i=3

V[i,k] = 7

V[i–1,k] =3

k – w_i=2

i=n, k=W

while i,k > 0

if *V[i,k]* ≠ *V[i–1,k]* then

mark the i^{th} item as in the knapsack

$i = i–1$, $k = k\text{-}w_i$

else

$i = i–1$

Finding the Items (5)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

i=1

k= 2

b_i=3

w_i=2

$V[i,k] = 3$

$V[i-1,k] = 0$

$k - w_i = 0$

i=n, k=W

while i,k > 0

if $V[i,k] \neq V[i-1,k]$ then

mark the i^{th} item as in the knapsack

$i = i-1$, $k = k-w_i$

else

$i = i-1$

Finding the Items (6)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

i=0
k= 0

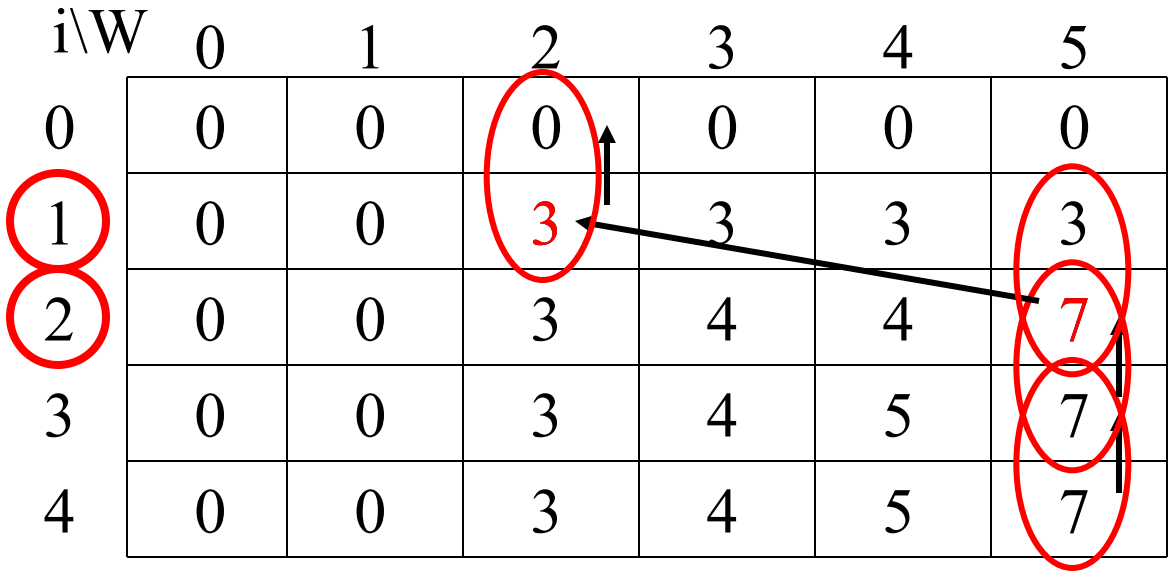
The optimal knapsack should contain {1, 2}

i=n, k=W
while i,k > 0
    if $V[i,k] \neq V[i-1,k]$ then
        mark the n^{th} item as in the knapsack
        $i = i-1$, $k = k-w_i$
    else
        $i = i-1$

Finding the Items (7)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)



i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

The optimal knapsack should contain {1, 2}

i=n, k=W
while i,k > 0
    if $V[i,k] \neq V[i-1,k]$ then
        mark the n^{th} item as in the knapsack
        $i = i-1$, $k = k-w_i$
    else
        $i = i-1$

Memorization (Memory Function Method)

- *Goal:*
  - *Solve only subproblems that are necessary and solve it only once*
- *Memorization* is another way to deal with overlapping subproblems in dynamic programming
- With memorization, we implement the algorithm *recursively*:
  - If we encounter a new subproblem, we compute and store the solution.
  - If we encounter a subproblem we have seen, we look up the answer
- Most useful when the algorithm is easiest to implement recursively
  - Especially if we do not need solutions to all subproblems.

0-1 Knapsack Memory Function Algorithm

```
for i = 1 to n
    for w = 1 to W
        V[i,w] = -1

for w = 0 to W
    V[0,w] = 0
for i = 1 to n
    V[i,0] = 0
```

```
MFKnapsack(i, w)
    if V[i,w] < 0
        if w < w_i
            value = MFKnapsack(i-1, w)
        else
            value = max(MFKnapsack(i-1, w),
                        b_i + MFKnapsack(i-1, w-w_i))
        V[i,w] = value
    return V[i,w]
```

Review: The Knapsack Problem

- More formally, the *0-1 knapsack problem*:
  - The thief must choose among n items, where the ith item worth v_i dollars and weighs w_i pounds
  - Carrying at most W pounds, maximize value
    - Note: assume v_i, w_i, and W are all integers
    - "0-1" b/c each item must be taken or left in entirety
- A variation, the *fractional knapsack problem*:
  - Thief can take fractions of items
  - Think of items in 0-1 problem as gold ingots, in fractional problem as buckets of gold dust

Review: The Knapsack Problem And Optimal Substructure

- Both variations exhibit optimal substructure
- To show this for the 0-1 problem, consider the most valuable load weighing at most W pounds
  - *If we remove item j from the load, what do we know about the remaining load?*
  - A: remainder must be the most valuable load weighing at most W - w_j that thief could take from museum, excluding item j

Solving The Knapsack Problem

- The optimal solution to the fractional knapsack problem can be found with a greedy algorithm
  - *How?*
- The optimal solution to the 0-1 problem cannot be found with the same greedy strategy
  - Greedy strategy: take in order of dollars/pound
  - Example: 3 items weighing 10, 20, and 30 pounds, knapsack can hold 50 pounds
    - *Suppose item 2 is worth $100. Assign values to the other items so that the greedy strategy will fail*

Why 0-1 Knapsack Problem Is NP-Complete?

- The decision version of the 0-1 knapsack problem is an NP-Complete problem.
- A 'Yes' or 'No' solution to the above decision problem is NP-Complete. Solving the above inequalities is the same as solving the **Subset-Sum Problem**, which is proven to be NP-Complete. Therefore, the knapsack problem can be reduced to the Subset-Sum problem in polynomial time.
- Further, the complexity of this problem depends on the size of the input values , . That is, if there is a way of rounding off the values making them more restricted, then we'd have a polynomial-time algorithm.
- https://www.baeldung.com/cs/knapsack-problem-np-completeness

Thanks for Your Attention!

Exercises