CH08-320201

Algorithms and Data Structures

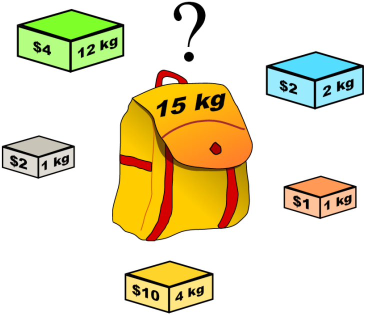
ADS

Lecture 20

Dr. Kinga Lipskoch

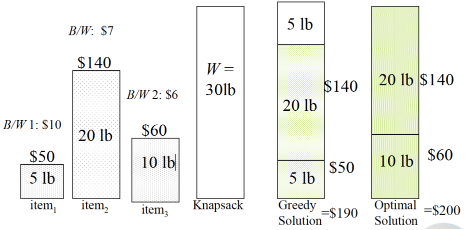
Spring 2019

Knapsack Problem (Revisited)



Knapsack Problem: Greedy Algorithm

- Greedy approaches make a locally optimal choice.
- There is no guarantee that this will lead to a globally optimal solution.
- In the 0-1 Knapsack Problem it did not.



Knapsack Problem: Dynamic Programming Approach (1)

- Let us try a dynamic programming approach.
- We need to carefully identify the subproblems.
- If items are labeled $1..n$, then a subproblem would be to find an optimal solution for $S_k = \{\text{items labeled } 1, 2, ..., k\}$.

Knapsack Problem: Dynamic Programming Approach (2)

Max weight: W = 20

$w_1 = 2$ $b_1 = 3$	$w_2 = 4$ $b_2 = 5$	$w_3 = 5$ $b_3 = 8$	$w_4 = 3$ $b_4 = 4$	

For S_4:
Total weight: 14
Maximum benefit: 20

$w_1 = 2$ $b_1 = 3$	$w_2 = 4$ $b_2 = 5$	$w_3 = 5$ $b_3 = 8$	$w_5 = 9$ $b_5 = 10$

For S_5:
Total weight: 20
Maximum benefit: 26

S_4 S_5

Item #	Weight w_i	Benefit b_i
1	2	3
2	4	5
3	5	8
4	3	4
5	9	10

Solution for S_4 is not part of the solution for S_5

Knapsack Problem: Dynamic Programming Approach (3)

- Re-define the subproblem by also considering the weight that is given to the subproblem.
- The subproblem then will be to compute $V[k, w]$, i.e., to find an optimal solution for $S_k = \{\text{items labeled } 1, 2, ..k\}$ in a knapsack of size w, with $w \leq W$.
- $V[k, w]$ denotes the overall benefit of the solution.
- Question: Assuming we know $V[i, j]$ for $i = 0, 1, 2, ..., k-1$ and $j = 0, 1, 2, ..., w$, how can we derive $V[k, w]$?
- Answer:

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

Knapsack Problem: Dynamic Programming Approach (4)

- Explanation of

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

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

Knapsack Problem: Dynamic Programming Approach (5)

Dynamic-programming algorithm:

Input: $S_n = \{(w_i, b_i) : i = 1, ..., n\}$ and maximum weight 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 = 0 to W
    if (wi > w) // i cannot be part of solution
      V[i,w] = V[i-1,w]
    else        // wi <= w
      if (V[i-1,w] > bi + V[i-1,w-wi])
        V[i,w] = V[i-1,w]
      else
        V[i,w] = bi + V[i-1,w-wi]
```

Knapsack Problem: Dynamic Programming Approach (6)

Computation time:

O(W)

O(n)

O(nW)

```
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 = 0 to W
    if (wi > w)
        V[i,w] = V[i-1,w]
    else
        if (V[i-1,w] > bi + V[i-1,w-wi])
            V[i,w] = V[i-1,w]
        else
            V[i,w] = bi + V[i-1,w-wi]
```

Overall time complexity is **O(nW)**

Pseudo-Polynomial Time

- A numeric algorithm runs in pseudo-polynomial time if its running time is a polynomial in the numeric value of the input (the largest integer present in the input) – but not necessarily in the length of the input (the number of bits required to represent it)
- Example:
  - The time complexity of the previous algorithm is $O(nW)$
  - Consider $n = 50000$ and $W = 1,000,000,000,000$
  - Binary representation of W is 1110100011010100101001010001000000000000 which requires $L = 40$ bits
  - Therefore, $O(nW) = O(50000 * 2^{40}) = O(n * 2^{L})$
  - This means that the previous algorithm runs in pseudo-polynomial time.

Pseudo-Polynomial vs. Truly Polynomial

- Consider the algorithm for adding n numbers using a loop running n times, we say, the complexity is $O(n)$
- But this n can also be written as 2^b
- Does this mean that adding n numbers is a pseudo-polynomial time algorithm?
- Adding n numbers, we implicitly say, that we are adding the sum of n of some constant c bit numbers (e.g., 32 bit integers)
- Then the size of n numbers is $c * n$
- The complexity is $O(c * n)$ with c being a constant which means that the complexity is $O(n)$, therefore it is a truly polynomial time algorithm

Knapsack Problem: Dynamic Programming Approach (7)

Example:

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

Knapsack Problem: Dynamic Programming Approach (8)

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
```

Knapsack Problem: Dynamic Programming Approach (9)

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
```

Knapsack Problem: Dynamic Programming Approach (10)

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

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

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

Knapsack Problem: Dynamic Programming Approach (11)

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

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

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

Knapsack Problem: Dynamic Programming Approach (12)

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					

$i=1$
$b_i=3$
$w_i=2$
$w=3$
$w-w_i=1$

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

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

Knapsack Problem: Dynamic Programming Approach (13)

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					

$i=1$

$b_i=3$

$w_i=2$

$w=4$

$w-w_i=2$

Items:

$\boxed{1: (2,3)}$

2: (3,4)

3: (4,5)

4: (5,6)

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

Knapsack Problem: Dynamic Programming Approach (14)

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					

$i=1$
$b_i=3$
$w_i=2$
$w=5$
$w-w_i=3$

Items:
$\boxed{1: (2,3)}$
2: (3,4)
3: (4,5)
4: (5,6)

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

Knapsack Problem: Dynamic Programming Approach (15)

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					

$i=2$

$b_i=4$

$w_i=3$

$w=1$

$w-w_i=-2$

Items:

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

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

Knapsack Problem: Dynamic Programming Approach (16)

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					

$i=2$

$b_i=4$

$w_i=3$

$w=2$

$w-w_i=-1$

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

```
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
```

Knapsack Problem: Dynamic Programming Approach (17)

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					

$i=2$

$b_i=4$

$w_i=3$

$w=3$

$w-w_i=0$

Items:

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

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

Knapsack Problem: Dynamic Programming Approach (18)

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					

$i=2$
$b_i=4$
$w_i=3$
$w=4$
$w-w_i=1$

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

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

Knapsack Problem: Dynamic Programming Approach (19)

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					

$i=2$

$b_i=4$

$w_i=3$

$w=5$

$w-w_i=2$

Items:

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

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

Knapsack Problem: Dynamic Programming Approach (20)

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					

$i=3$
$b_i=5$
$w_i=4$
$w=1..3$

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

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

Knapsack Problem: Dynamic Programming Approach (21)

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					

$i=3$

$b_i=5$

$w_i=4$

$w=4$

$w-w_i=0$

Items:

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

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

Knapsack Problem: Dynamic Programming Approach (22)

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					

$i=3$
$b_i=5$
$w_i=4$
$w=5$
$w-w_i=1$

Items:

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

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

Knapsack Problem: Dynamic Programming Approach (23)

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**	

$i=4$

$b_i=6$

$w_i=5$

$w=1..4$

Items:

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

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

Knapsack Problem: Dynamic Programming Approach (24)

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**

$i=4$
$b_i=6$
$w_i=5$
$w=5$
$w-w_i=0$

Items:

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

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

Knapsack Problem: Dynamic Programming Approach (25)

- This algorithm only finds the maximally possible value that can be carried in the knapsack, i.e., the value of $V[n, W]$.
- To know the items that are put together to reach this maximum value, an addition to this algorithm is necessary that is based on traversing the table in a post-processing step.
- Algorithm:

```
i=n, k=W
while (i > 0 and k > 0)
  if (V[i,k] != V[i-1,k])
    add item i to knapsack
    i = i-1, k = k-wi
  else // item i is not in the knapsack
    i = i-1
```

Knapsack Problem: Dynamic Programming Approach (26)

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

Items:

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

```
i=n, k=W
while (i > 0 and k > 0)
    if (V[i,k] ≠ V[i-1,k])
        mark the ith item as in the knapsack
        i = i-1, k = k-wi
    else
        i = i-1
```

Knapsack Problem: Dynamic Programming Approach (27)

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

Items:

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

```
i=n, k=W
while (i > 0 and k > 0)
    if (V[i,k] ≠ V[i-1,k])
        mark the ith item as in the knapsack
        i = i-1, k = k-wi
    else
        i = i-1
```

Knapsack Problem: Dynamic Programming Approach (28)

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

Items:

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

```
i=n, k=W
while (i > 0 and k > 0)
    if (V[i,k] ≠ V[i-1,k])
        mark the ith item as in the knapsack
        i = i-1, k = k-wi
    else
        i = i-1
```

Knapsack Problem: Dynamic Programming Approach (29)

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

Items:

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

```
i=n, k=W
while (i > 0 and k > 0)
    if (V[i,k] ≠ V[i-1,k])
        mark the ith item as in the knapsack
        i = i-1, k = k-wi
    else
        i = i-1
```

Knapsack Problem: Dynamic Programming Approach (30)

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

Items:

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

```
i=n, k=W
while (i > 0 and k > 0)
    if (V[i,k] ≠ V[i-1,k])
        mark the ith item as in the knapsack
        i = i-1, k = k-wi
    else
        i = i-1
```

Knapsack Problem: Dynamic Programming Approach (31)

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}

Items:

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

```
i=n, k=W
while (i > 0 and k > 0)
    if (V[i,k] ≠ V[i-1,k])
        mark the ith item as in the knapsack
        i = i-1, k = k-wi
    else
        i = i-1
```

Summary

We have discussed 3 algorithmic concepts:

1. Divide & Conquer Method
   Splits problem into multiple subproblems, solves them recursively, and combines the solutions.
2. Greedy Algorithms
   Makes a locally best choice to reduce the problem to a subproblem and iteratively solves the subproblem in the hope to find a globally best solution.
3. Dynamic Programming
   Computes subproblems in a bottom-up fashion and stores (intermediate) solutions to subproblems in a table.