6.4 Knapsack Problem

http://tvtropes.org/pmwiki/pmwiki.php/Main/DouweDabbert

Knapsack Problem

Knapsack problem.

- Given n objects and a "knapsack."
- Item i weighs $w_i > 0$ kilograms and has value $v_i > 0$.
- Knapsack has capacity of W kilograms.
- Goal: fill knapsack so as to maximize total value.

Q. What is the maximum value here?

W = 11

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

TU Delft

Knapsack Problem

Knapsack problem.

- Given n objects and a "knapsack."
- Item i weighs $w_i > 0$ kilograms and has value $v_i > 0$.
- Knapsack has capacity of W kilograms.
- Goal: fill knapsack so as to maximize total value.

Q. What is the maximum value here?

A. { 3, 4 } attains 40

W = 11

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Q. What could be a reasonable greedy algorithm?

A. Repeatedly add item with maximum ratio v_i / w_i.

Q. Is this greedy algorithm optimal?

Ex: { 5, 2, 1 } achieves only value = 35 $\Rightarrow$ greedy not optimal.

TU Delft

Dynamic Programming: False Start

Recursively define value of optimal solution:

Def. OPT(i) = max profit subset of items 1, …, i.

- Case 1: OPT does not select item i.
  - OPT selects best of { 1, 2, …, i-1 }

- Case 2: OPT selects item i.
  - accepting item i does not immediately imply that we will have to reject other items; this depends on the remaining weight!
    (does not only depend on best of { 1, 2, …, i-1 })

Conclusion. Need more sub-problems!

Q. What are the relevant parameters of a sub-problem?

Q. And how to express the optimal value of a set of items and a capacity in terms of these sub-problems? (1 min)

TU Delft

Dynamic Programming: Adding a New Variable

Recursively define value of optimal solution:

Def. OPT(i, w) = max profit subset of items 1, …, i with weight limit w.

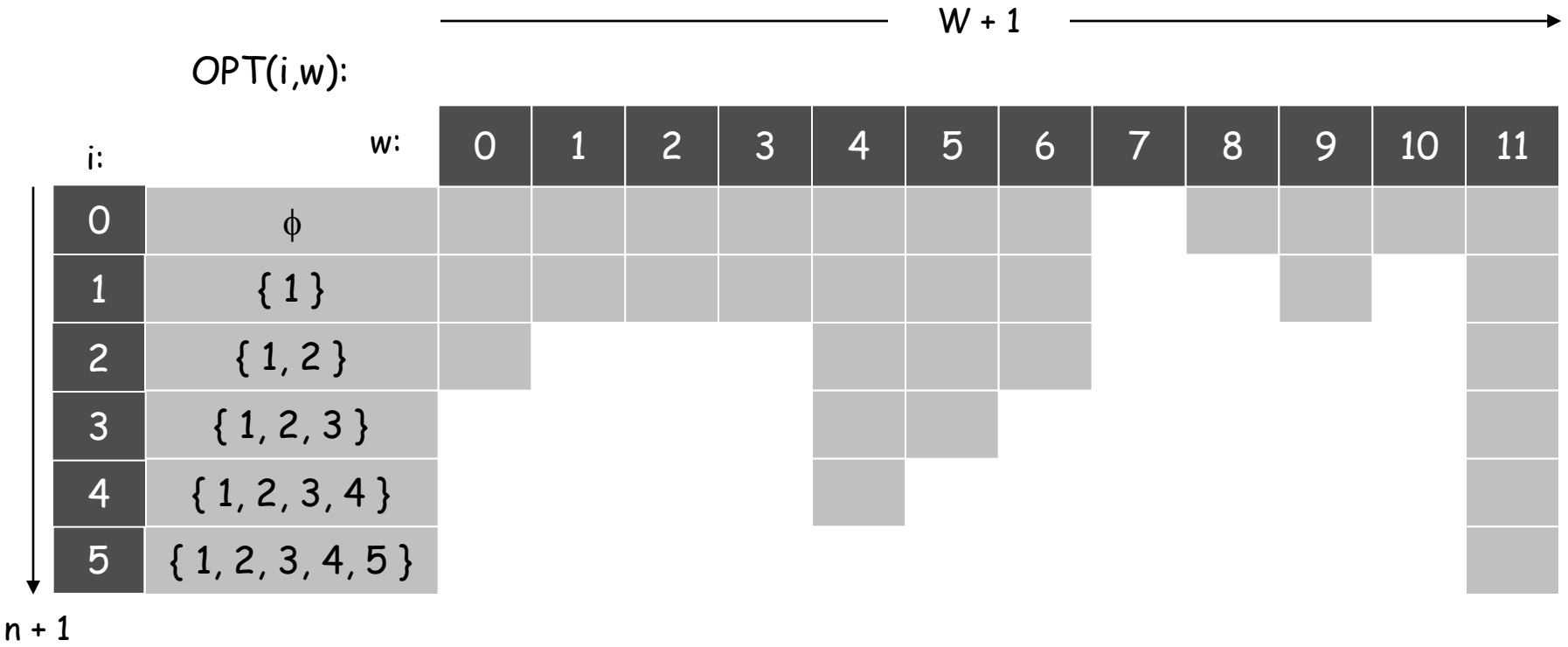
- Case 1: OPT does not select item i.
  - OPT selects best of { 1, 2, …, i-1 } using weight limit w

- Case 2: OPT selects item i.
  - new weight limit = $w - w_i$
  - OPT selects best of { 1, 2, …, i–1 } using this new weight limit

$$OPT(i, w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1, w) & \text{if } w_i > w \\ \max \{ OPT(i-1, w), \; v_i + OPT(i-1, w - w_i) \} & \text{otherwise} \end{cases}$$

TU Delft

Knapsack Algorithm: Recursive



W = 11

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

$$OPT(i, w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1, w) & \text{if } w_i > w \\ \max\{ OPT(i-1, w),\ v_i + OPT(i-1, w-w_i) \} & \text{otherwise} \end{cases}$$

Knapsack Algorithm: Recursive

W + 1

OPT(i,w):

i:	w:	0	1	2	3	4	5	6	7	8	9	10	11
0	ϕ	0	0	0	0	0	0	0		0	0	0	0
1	{ 1 }	0	1	1	1	1	1	1			1		1
2	{ 1, 2 }	0				7	7	7					7
3	{ 1, 2, 3 }					7	18						25
4	{ 1, 2, 3, 4 }					7							40
5	{ 1, 2, 3, 4, 5 }												40

n + 1

W = 11

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

$$OPT(i, w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1, w) & \text{if } w_i > w \\ \max\{ OPT(i-1, w),\ v_i + OPT(i-1, w-w_i) \} & \text{otherwise} \end{cases}$$

Knapsack Algorithm: Bottom-Up

W + 1

OPT(i,w):

i:	w:	0	1	2	3	4	5	6	7	8	9	10	11
0	ϕ												
1	{ 1 }												
2	{ 1, 2 }												
3	{ 1, 2, 3 }												
4	{ 1, 2, 3, 4 }												
5	{ 1, 2, 3, 4, 5 }												

n + 1

W = 11

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

$$OPT(i, w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1, w) & \text{if } w_i > w \\ \max\{ OPT(i-1, w),\ v_i + OPT(i-1, w-w_i) \} & \text{otherwise} \end{cases}$$

Knapsack Algorithm: Bottom-Up

W + 1

OPT(i,w):

i: \ w:	0	1	2	3	4	5	6	7	8	9	10	11
0	ϕ	0	0	0	0	0	0	0	0	0	0	0
1	{ 1 }	0	1	1	1	1	1	1	1	1	1	1
2	{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7
3	{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25	25	25	25
4	{ 1, 2, 3, 4 }	0	1	6	7	7	18	22	24	28	29	29	40
5	{ 1, 2, 3, 4, 5 }	0	1	6	7	7	18	22	28	29	34	34	40

n + 1

W = 11

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

$$OPT(i, w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1, w) & \text{if } w_i > w \\ \max\{ OPT(i-1, w),\ v_i + OPT(i-1, w-w_i) \} & \text{otherwise} \end{cases}$$

Knapsack Problem: Bottom-Up

Compute value of optimal solution iteratively.

Knapsack. Fill up an n-by-W array.

```
Input: n, w_1,…,w_N, v_1,…,v_N

for w = 0 to W
   M[0, w] = 0

for i = 1 to n
   for w = 0 to W
      if (w_i > w)
         M[i, w] = M[i-1, w]
      else
         M[i, w] = max {M[i-1, w], v_i + M[i-1, w-w_i ]}

return M[n, W]
```

Q. What is the running time? (1 min)

A.

Knapsack Problem: Bottom-Up

Compute value of optimal solution iteratively.

Knapsack. Fill up an n-by-W array.

```
Input: n, w_1,…,w_N, v_1,…,v_N

for w = 0 to W
   M[0, w] = 0

for i = 1 to n
   for w = 0 to W
      if (w_i > w)
         M[i, w] = M[i-1, w]
      else
         M[i, w] = max {M[i-1, w], v_i + M[i-1, w-w_i ]}

return M[n, W]
```

Q. What is the running time? (1 min)

A. $\Theta(n\,W)$

Knapsack Problem: Running Time

Running time. $\Theta(n\ W)$.

- Not polynomial in input size!
- "Pseudo-polynomial."
- Decision version of Knapsack is NP-complete.
  (Complexity theory, 3rd year)

Knapsack approximation algorithm. There exists a polynomial algorithm that produces a feasible solution that has value within 0.01% of optimum.
[Section 11.8, Master course on Advanced Algorithms]

Knapsack Problem: Finding a Solution

Construct optimal solution from computed information.

```
Run Knapsack()
Run Find-Solution(n,W)

Find-Solution(i,w) {
   if (i = 0 or w = 0)
      output nothing
   else if ( M[i,w] = M[i-1, w] )
         Find-Solution(i-1,w)
   else
         print i
         Find-Solution(i-1,w-w_i)
}
```