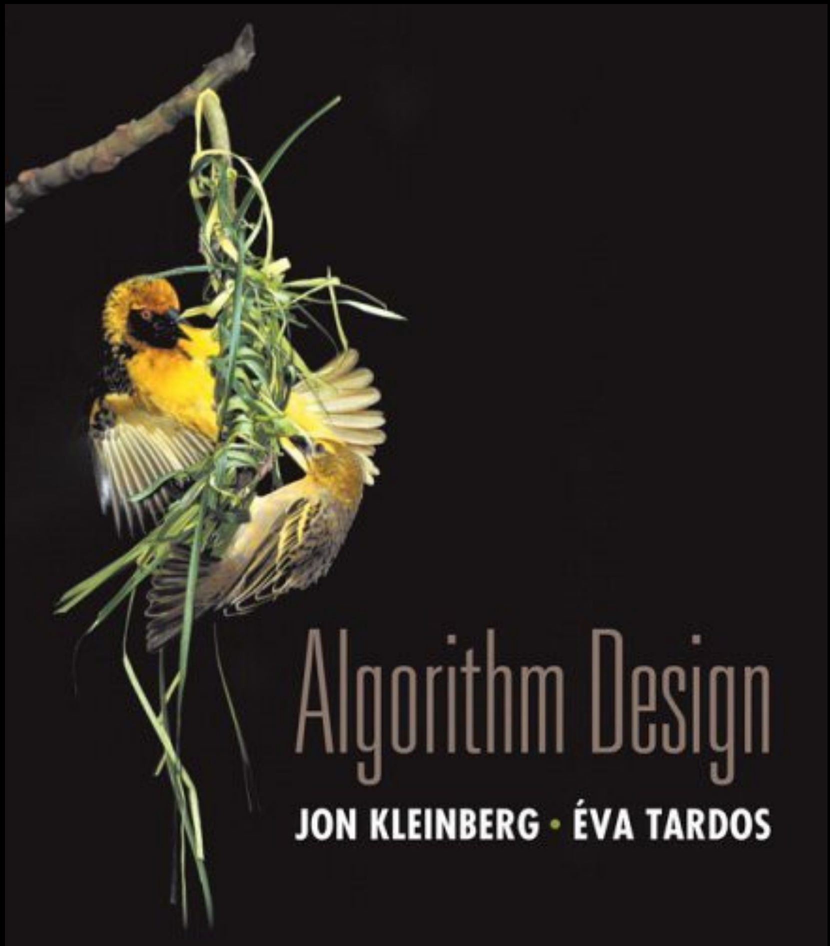

Chapter 6

Dynamic Programming

Slides by Kevin Wayne.

6.4 Knapsack Problem

Knapsack Problem

Knapsack problem.

- Given n objects $I = \{(w_i,v_i): i=1,..,n\}$ and a Knapsack
- Item i weighs $w_i > 0$ kilograms and has value $v_i > 0$.
- Knapsack has capacity of W kilograms.
- Feasible Sol: $S \subseteq I$ s.t. $\Sigma_{j \in S}\, w_i$
- Goal: fill knapsack so as to *maximize* total SUM of values: $\Sigma_{j \in S}\, V_i$

Input

Ex: $S = \{ 3, 4 \}$ has value 40.

W = 11

#	value	weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Greedy: repeatedly add item with maximum ratio v_i / w_i.

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

Dynamic Programming: 1st approach

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

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

- **Case 2:** OPT selects item i. (Which sub-problems must recursively be invoked?)
  - accepting item *i* does not immediately imply that we will have to reject ***other items*** **k < i.**
  - without knowing what other items were selected before i, we don't even know if we have enough room for i

Conclusion: Need **more** sub-problems, i.e. more parameters than just index i

Dynamic Programming: Adding a New Variable

Def. For any fixed pair $i \in I$ and $w \in \{0,1,\ldots,W\}$ consider:

$OPT(i, w)$ = **max** profit subset of items $1, \ldots, i$ with weight parameter w.

- Case 1: OPT does not select item i.
  - OPT selects **best** of sub-probl $\{ 1, 2, \ldots, i-1 \}$ using weight limit w
- Case 2: OPT selects item i
  - new weight limit = $w - w_i$
  - OPT selects **best** of $\{ 1, 2, \ldots, i-1 \}$ using **this new weight limit**

$$OPT(i,w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1,w) & \text{if } w_i > w \\ \max\{ \underbrace{OPT(i-1,w)}_{\text{Case 1}}, \underbrace{v_i + OPT(i-1, w - w_i)}_{\text{Case 2}} \} & \text{otherwise} \end{cases}$$

Q. How to fill-up the matrix $M(i, w)$, for all $i = 1..n$; $w = 0..W$??

Answer: Nice Ordering Property

In order to compute row i,
you need the values of rows $j < i$ only !

Knapsack Problem: Bottom-Up

Knapsack. Fill up an **n** x **W** array.
The **good ordering** for sub-problems

Inizialization of the First row:
no Items in the solution!

To compute M[i,w], we only need values M[i-1,w] and M[i-1,w-w_i]…
They are already there!

```
Input: n, W, w1,…,wN, v1,…,vN

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

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

return M[n, W]
```

Knapsack Algorithm

W + 1

n + 1

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

OPT: { 4, 3 }
value = 22 + 18 = 40

W = 11

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

Knapsack Problem: Running Time

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

- **Not polynomial** in **input size**!

- "***Pseudo-polynomial.***"
- Decision version of Knapsack is **NP-complete**. [Chapter 8]

Knapsack approximation algorithm. There exists a poly-time algorithm that produces a feasible solution that has value within 0.01% of optimum. [Section 11.8]

SELF TESTS at HOME

You should **write** on your notes

- Formalize the definition of Knapsack Problem
- Give a rigorous proof of the optimality of the OPT(i,w) recursive formula in the first case (when i does not belong to the optimal solution). Hint: Use Exchange argument and Contradiction
- Give a concrete instance with at least 6 items. For any given entry M[i,w] find excactly which are the (only) two previous entries required by the computation of M[i,w]
- Did you understand well why the proposed Dyn Programming for this problem is not polynomial? Give a formal argument for this issue.