Algorithms and Data Structures

The Last Lesson

Marius Kloft

This Course

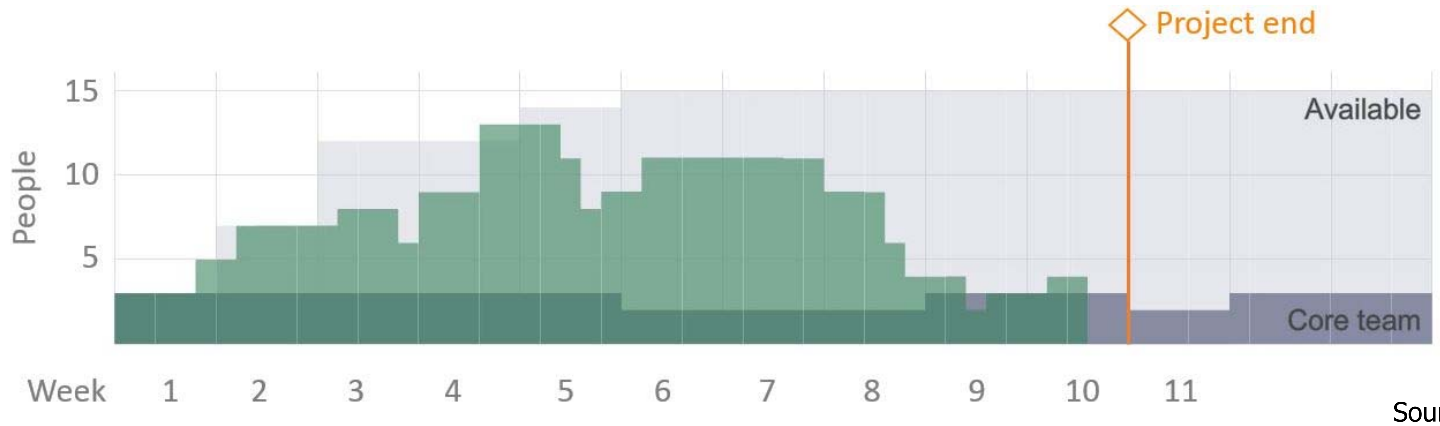
• Introduction	2
• Complexity analysis	1
• Abstract Data Types	1
• Styles of algorithms	1
• Lists, stacks, queues	2
• Sorting (lists)	3
• Searching (in lists, PQs, SOL)	5
• Hashing (to manage lists)	2
• Trees (to manage lists)	4
• Graphs (no lists!)	4
• The End	1
• Sum	**26/26**

Content of this Lecture

- Knapsack – hard, but "approximable"
  - The Problem
  - Dynamic programming solution
  - Approximation
- Your Feedback
- What's next

Real-world Motivation

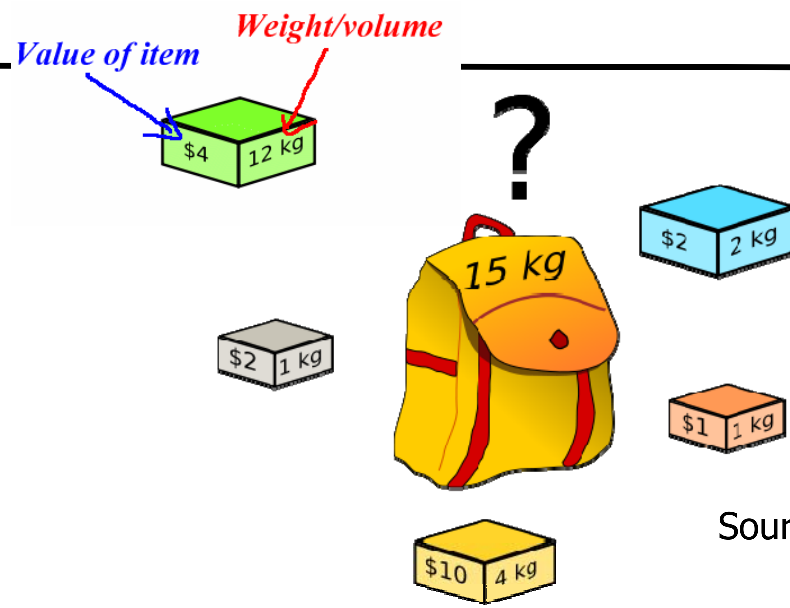
- Project management: Resource Allocation



Source: Wikipedia.org

- Maximize success of project
  - Given a maximal budget or personal resources

Knapsack Problem



Source: Wikipedia.de

- Given a set S of items, |S|=n, with weights w_i and value v_i and a maximal weight M; find the subset $T \subseteq S$ such that

$$\sum_{i \in T} w_i \leq M \quad \text{and} \quad \sum_{i \in T} v_i = \max$$

How to find best set of items T ??

Complexity

- Brute-force approach: Enumerate all possible $T \subseteq S$
- For each T, computing its value and weight is in O(n)
- How many different T exist?
  - Every item from S can be part of T or not
  - This gives $2*2*2* \ldots *2=2^n$ different options
- Bottom line: brute-force $O(2^n)$
  - Actually cannot do better in complexity:
    The knapsack problem is NP-complete

Variations

- Our formulation is called 0/1 knapsack problem
  - Every item can be in the set at most once – no copies
- In the unbounded knapsack problem, a number x_i of copies of each item (w_i, v_i) may be used:

$$\sum_{i \in T} x_i * w_i \leq M \qquad \sum_{i \in T} x_i * v_i = \max$$

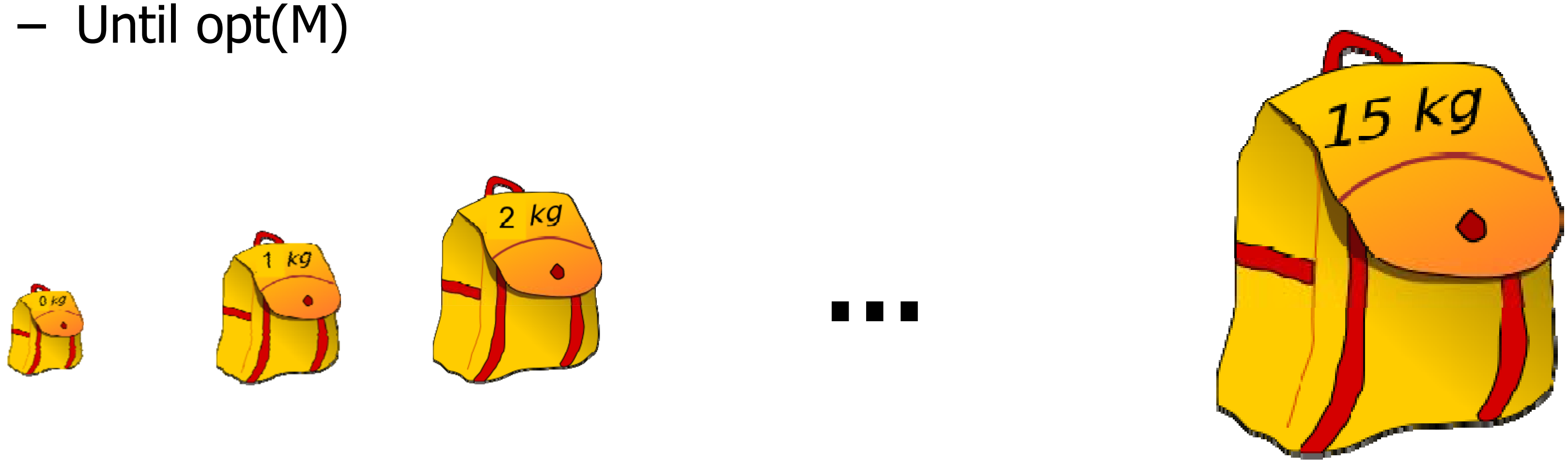
  - All x_i must be integer
  - Bounded Knapsack has an upper bound on each x_i as additional constraint
- Also NP-complete
- But …

Idea

- Consider unbounded case and $\forall i{:}w_i>0$, $M>0$ and $\exists i{:}w_i \leq M$
- Let opt(m) be the optimal solution for some m with $m \leq M$:

$$\mathrm{opt}(m) := \max\left(\sum_{i \in T} x_i * v_i\right) \quad \text{under} \quad \sum_{i \in T} x_i * w_i \leq m$$

- Idea: Use dynamic programming
  - Find solution for opt(0), then for opt(1), then for opt(2), …
  - Until opt(M)



Dynamic Programming Solution

- Assume we know opt(0), …, opt(m-1) for some m≤M
- We can use this knowledge to construct a solution for m
  - The "new" knapsack has 1 kg of weight more capacity
  - An optimal solution either fills this 1 kg or not
  - If it does fill it, this kg can be used only by exactly one item
- Thus:

$$opt(m) = \max(opt(m-1), \max_{i:w_i<m}(v_i + opt(m-w_i)))$$

we do not fill the additional 1kg

we use one item extra (with weight w_i), thus there is only $m-w_i$ weight left

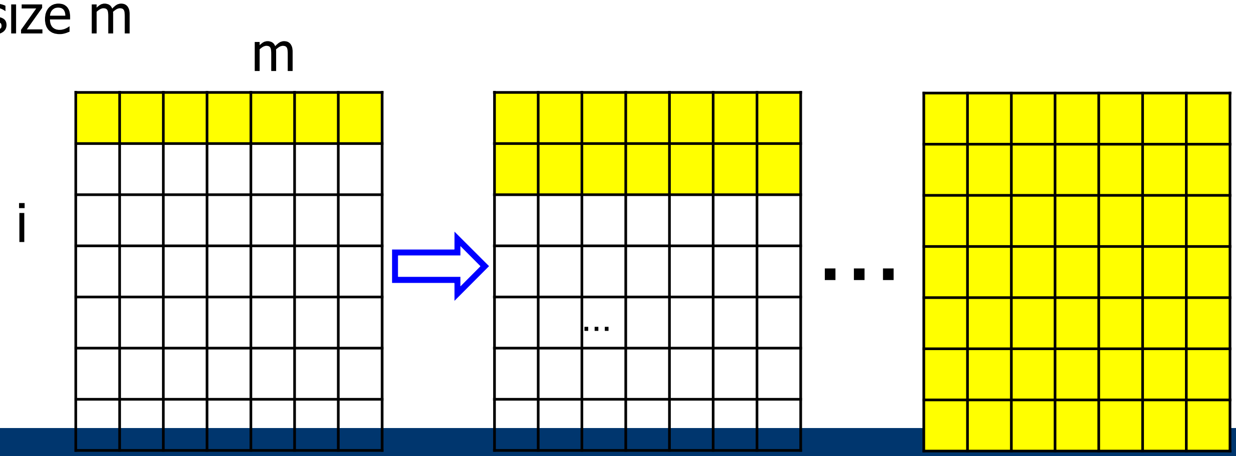
Analysis

$$opt(m) = \max(opt(m-1), \max_{i:w_i<m}(v_i + opt(m-w_i)))$$

- Computing opt(M) requires bottom-up computation of opt(0), opt(1), … opt(M-1), opt(M)
  - In every step, we consider at most n different items
- Together: O(n*M)
  - No contradiction to NP-completeness:
    $O(n*M) = O(n*2^{\underbrace{\#\text{bits}(M)}_{\text{length of input}}})$
  - Good runtime for small M

DP for 0/1 Knapsack (Sketch)

- A similar DP approach works for 0/1 Knapsack
- Define opt(m,i) := optimal value under budget m when using only the first i items
- Can show: if $w_i \leq m$, then
  $$opt(m,i) = \max(\, opt(m,i-1),\ v_i + opt(m-w_i,i-1)\,)$$
- Thus can again use a dynamic programming approach
  - First loop over #items i
  - Then loop over size m
  - Fills opt(m,i)



Content of this Lecture

- Knapsack: Outlook
  - The Problem
  - Dynamic programming solution
  - Approximation
- Your Feedback
- What's next

Greedy Algorithm

- Consider the unbounded knapsack problem
- Intuitively, we want items with high value and low weight
- We compute items' relative value $p_i = v_i/w_i$

p_i high

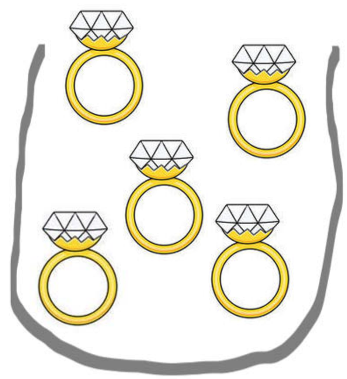
p_i low

Source: http://www.vectorhq.com/premium/cartoon-diamond-ring-374494

Source: http://www.seilnacht.com/Lexikon/bleisenk.JPG

Greedy Algorithm

- We sort items by their relative value $p_i = v_i/w_i$
- Iteratively choose items for T as follows:
  - Always use the item next with the best relative value that fits
  - Always put as many copies of an item as fit into the knapsack



- First step is in O(n*log(n))
- Never look back, never withdraw a previously put item
- But: How good are the solutions it computes?

Example

- Let M=20kg and S = { (\$5,7kg), (\$11,17kg) }
- Relative values: 5/7=0,71 and 11/17=0,64
- Greedy packs 2*item 1 (value=10, weight=14)
- Packing 1*item 2 would be better: value=11, weight=17

- How bad can greedy get?

Example

- Let M=20kg and S = { ($5,7kg), ($11,17kg) }
- Relative values: 5/7=0,71 and 11/17=0,64
- Greedy packs 2*item 1 (value=10, weight=14)
- Packing 1*item 2 would be better: value=11, weight=17

- How bad can greedy get?

Approximate Solution

- Let A be an algorithm computing solutions A(I) for instances I of an optimization problem P. Let OPT(I) be a function computing the optimal solution for I
- Assume P is a maximization problem: Large is good
- If, for all instances I, $OPT(I)/A(I) \leq \varepsilon$, then A is called an ε-approximation algorithm for P
  - ε is called the relative performance guarantee of A for P
- We are interested in polynomial algorithms A for hard problems P with small ε

Greedy Knapsack is a 2-Approximation

- Proof sketch
  - Only look at the best item i=(v, w) that fits into M (first iteration)
  - Assume i fits k times into M
  - If $k*w \leq M/2$, another i would fit. If follows: $k*w > M/2$
  - Assume rest=M-k*w is filled exactly by i'=(v',w') using k' instances
    - i' cannot be better than i, or we had used it for the first iteration
    - Assume i' is almost as good as i
  - It follows that $k*v > k'*v'$
  - Since k*v+rest-value=OPT, we have $OPT/k*v \leq 2$