CS217 - Data Structures & Algorithm Analysis (DSAA)

Lecture #12

Greedy algorithms

Prof. Pietro S. Oliveto
Department of Computer Science and Engineering
Southern University of Science and Technology (SUSTech)

olivetop@sustech.edu.cn
https://faculty.sustech.edu.cn/olivetop

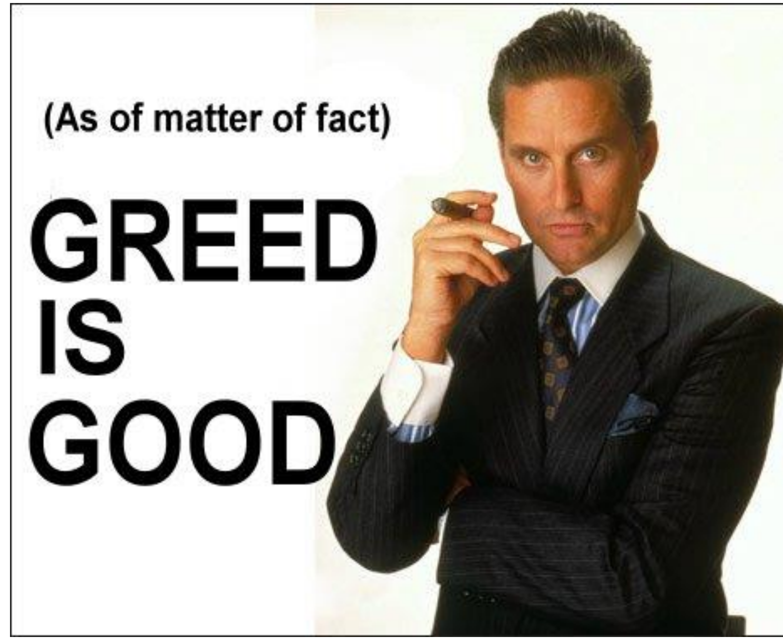
Reading: Sections 15.1 and 15.2

Aims of this lecture

- To discuss the greedy design paradigm for solving optimisation problems.
- To show how to prove correctness of greedy algorithms.
- To see examples of problems where greedy algorithms succeed, and examples of problems where the greedy approach fails.

Greedy Algorithms

- A greedy algorithm makes "greedy" – locally optimal – choices for subproblems.
- The hope is that this yields a globally optimal solution.
- Greedy algorithms work well for some problems, but may fail miserably on others.



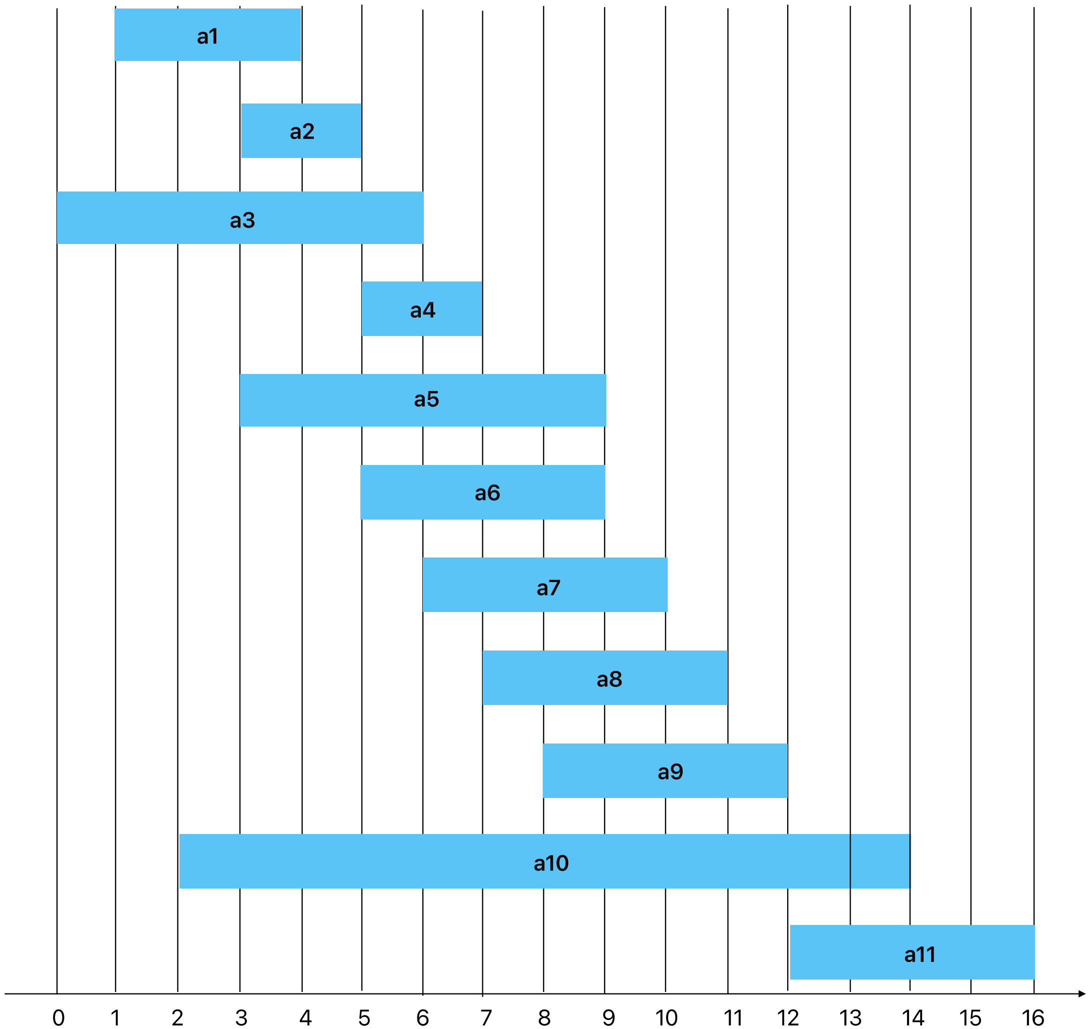
Activity Selection Problem

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	7	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

- Problem of scheduling competing activities that require exclusive use of a common resource, e.g. a lecture theatre.
- **Input:** activities $a_1, a_2, \ldots, a_n$ with start times $s_1, \ldots, s_n$ and finish times $f_1, \ldots, f_n$, where $0 \leq s_i \leq f_i < \infty$
- Activities are **compatible** if the intervals $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap.
- **Goal:** select a maximum-size set of mutually compatible activities (e.g. schedule a maximum number of lectures in a lecture theatre).
- Assume **without loss of generality** that activities are sorted according to finish time: $f_1 \leq f_2 \leq \cdots \leq f_n$

Activity Selection Problem

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	7	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16



Optimal substructure for activity selection

- Assume the optimal solution contains an activity a_k.
- By including $\boldsymbol{a_k}$, we are left with two subproblems:
  1. Selecting mutually compatible activities that end **before $\boldsymbol{a_k}$ starts**.
  2. Selecting mutually compatible activities that start **after $\boldsymbol{a_k}$ has ended**.
- The solutions to the subproblems used within the optimal solution must themselves be optimal.
- Smells like **Dynamic Programming!**
  - Try all possible a_k and solve smaller subproblems

Dynamic programming approach

- Let S_{ij}: set of activities that start after a_i finishes and finish before a_j starts;
- Suppose you want to find the max set of compatible activities in S_{ij}
- Assume the optimal solution contains an activity a_k. So:

$$c[i,j] = \begin{cases} 0 & \text{if } S_{ij} = \emptyset\,, \\ \max\{c[i,k] + c[k,j] + 1 : a_k \in S_{ij}\} & \text{if } S_{ij} \neq \emptyset\,. \end{cases}$$

  - $c[i,j] = Opt(S_{ij})$ be the optimal solution size for S_{ij}
  - Try all possible a_k and solve smaller subproblems
- Complete problem (n activities):
  - ➢ $Opt\left(S_{0(n+1)}\right) = \max\{Opt(S_{0k}) + Opt(S_{k(n+1)}) + 1, 0 < k < (n+1)\}$
  - S_{0k} denotes the set of activities that finish before a_k starts
  - $S_{k(n+1)}$ those that start after a_k finishes.

Runtime?

- Actually, a simpler approach is possible.

Greedy choice for activity selection

- **Intuition**: choose an activity that **leaves the resource available for as many other activities as possible**.
- One of the activities we choose must be the first to finish.
- Intuition: choose the activity a_1 with the **earliest finish time**, since that leaves the resource available for as many activities that follow it as possible.
- Note: there may be other activities that start before a_1, but they won't finish before time f_1.

Correctness of the greedy choice

- Define S_k as the set of activities that start after a_k finishes.
- **Theorem 15.1:** Consider any nonempty subproblem S_k, and let a_m be an activity in S_k with the earliest finish time. Then a_m is included in **some** maximum-size subset of mutually compatible activities of S_k.
  - In other words: there is a maximum-size set that includes the activity with earliest finish time (greedy choice).
  - When applying the greedy choice we are **still on track for finding a maximum-size set of activities**.
  - Hence the greedy choice is always **safe**.

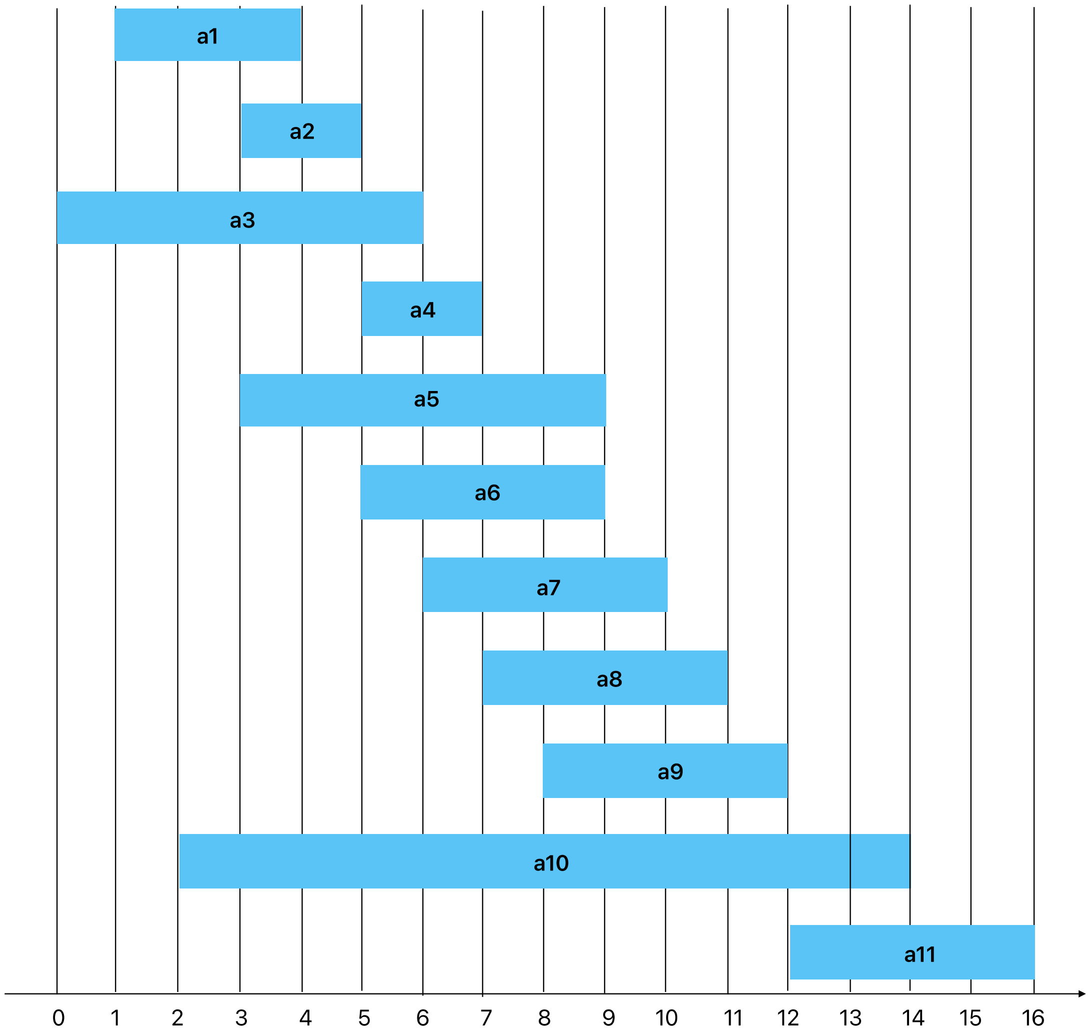
Proof of Theorem 15.1

Theorem 15.1: Consider any nonempty subproblem S_k, and let a_m be an activity in S_k with the earliest finish time. Then a_m is included in some maximum-size subset of mutually compatible activities of S_k.

- Let A_k be a maximum-size subset of mutually compatible activities in S_k, and let a_j be the activity in A_k with the earliest finish time.
  - a_m is the first-finishing activity in the whole subproblem (greedy choice)
  - a_j is the first-finishing activity selected in A_k, so $f_m \leq f_j$.
- To prove: there is a maximum-size compatible subset that includes a_m.
- If A_k includes the greedy choice a_m (that is, $a_j = a_m$), we're done.
- Otherwise, let's swap a_j for greedy choice a_m: $A_k' = A_k \setminus \{a_j\} \cup \{a_m\}$.
- Since $f_m \leq f_j$ and a_j is first-finishing, no incompatibilities are created.
- Since all activities in A_k were compatible, they are compatible in A_k'.
- As $|A_k'| = |A_k|$, A_k' is a maximum-size subset of compatible activities.

Activity Selection Problem

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	7	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16



Correctness of the greedy choice (2)

- **General scheme** for correctness of greedy algorithms:
  1. Cast the optimisation problem as one in which we make a choice and are left with one subproblem to solve.
  2. Prove that there is always an optimal solution to the original problem that makes the greedy choice, so that the greedy choice is always safe.

Idea behind Theorem 15.1:

- Consider an optimal solution *A*.
- If *A* contains the greedy choice, we're done.
- Otherwise, change *A* into *A'* such that A' contains the greedy choice and show that *A'* is also an optimal solution.

Greedy algorithm for activity selection

```
GREEDY-ACTIVITY-SELECTOR(s, f, n)
1  A = {a_1}
2  k = 1
3  for m = 2 to n
4      if s[m] ≥ f[k]           // is a_m in S_k?
5          A = A ∪ {a_m}         // yes, so choose it
6          k = m                 // and continue from there
7  return A
```

有一个默认前提：按照了结束时间的先后进行排序

- Pick first activity a_1 (earliest finish time) *[line 1]*
- Ignore activities starting before f_1 finishes *[line 4]*
- Pick first activity that starts after f_1 finishes (it has lowest *f*) *[line 5]*
- Iterate with remaining activities (*k* gives index of last activity added) *[line 6]*
- **Runtime?**

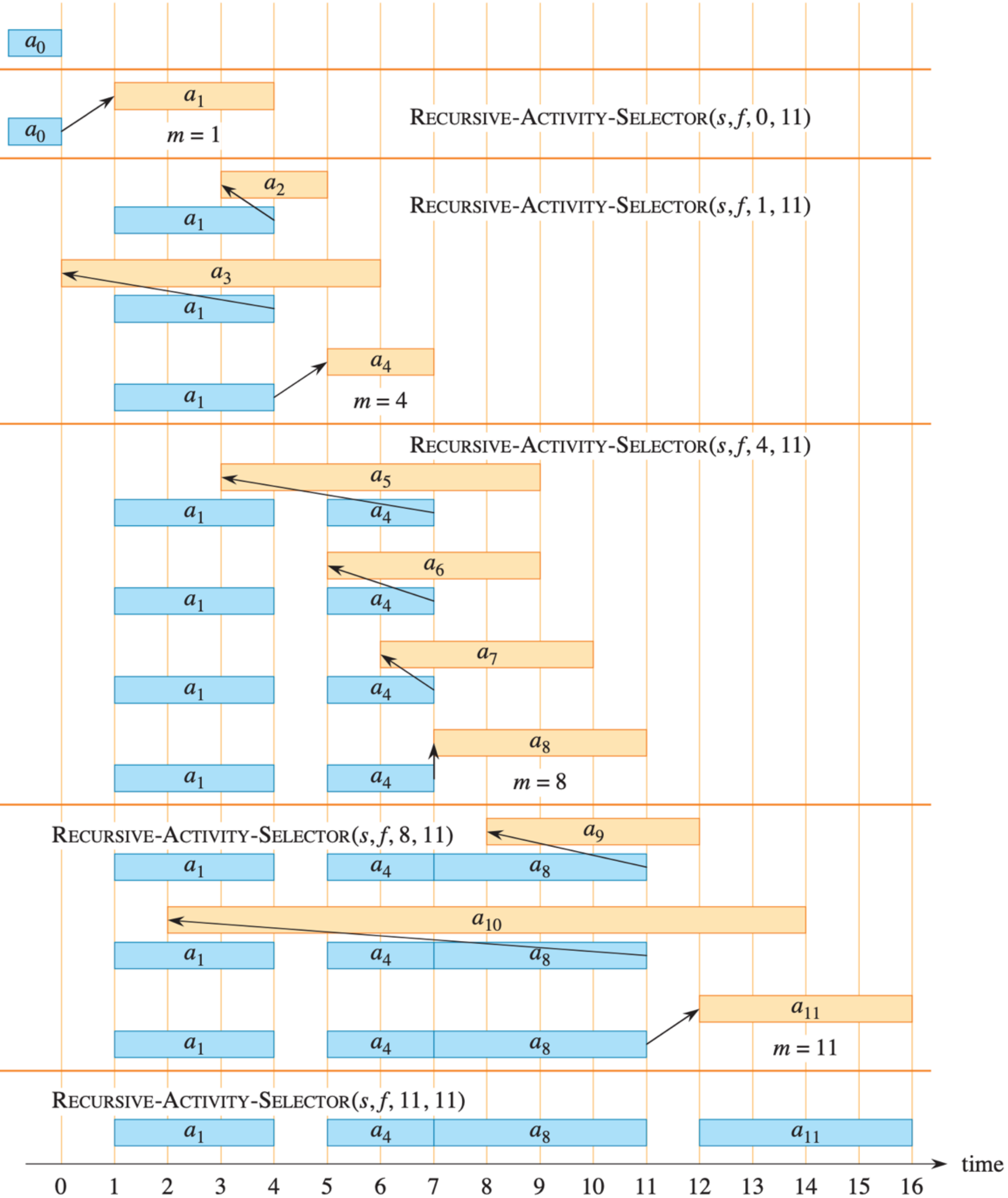
Recursive version

```
RECURSIVE-ACTIVITY-SELECTOR(s, f, k, n)
m = k + 1
while m ≤ n and s[m] < f[k]        // find the first activity in S_k to finish
    m = m + 1
if m ≤ n
    return {a_m} ∪ RECURSIVE-ACTIVITY-SELECTOR(s, f, m, n)
else return ∅
```

- Set $f_0 = 0$ and first recursive call for $(s, f, 0, n)$
- Looks for the first **compatible** activity to finish in S_k
- Recurse with remaining activities (*m* gives index of last activity added)
- **Runtime?**

Solution of example instance

i	s_i	f_i
0	–	0
1	1	4
2	3	5
3	0	6
4	5	7
5	3	9
6	5	9
7	6	10
8	7	11
9	8	12
10	2	14
11	12	16

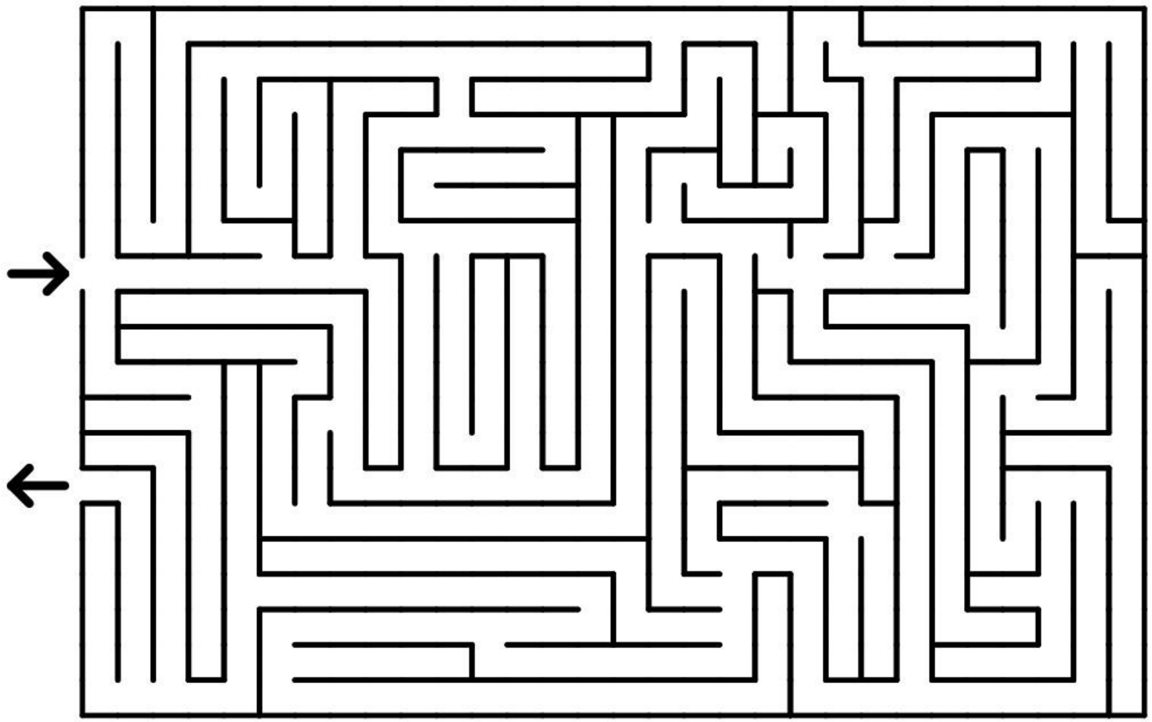


Coin Changing Problem

- How to give make change for *n* pence with the fewest number of coins?

- What's a greedy strategy here?
  - Pick the largest coin of value $a_i \leq n$ and add $\lfloor n/a_i \rfloor$ coins.
  - Iterate with remaining value.
- Does it always work for Sterling?
- Does it always work for every currency?

When Greed is not Good



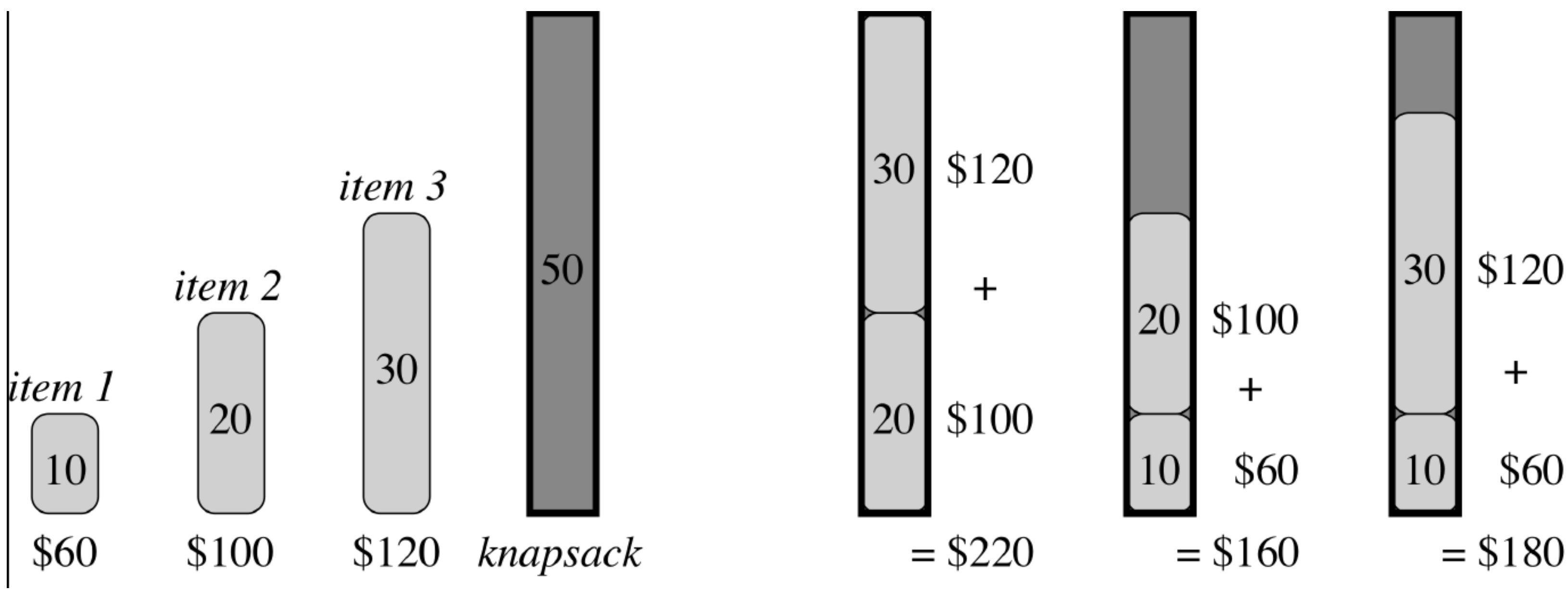
When Greed is not Good (2)

- **Travelling Salesman Problem (TSP):** given n cities and distances $d_{i,j}$ between each two cities i, j, find a shortest tour that visits all cities exactly once.
- What's a greedy strategy?
  - Always visit the nearest unseen city.
- Does it always work?
- Consider the following instance: $d_{1,2}=d_{2,3}=d_{3,4}=\ldots=d_{n-1,n}=1$ but $d_{n,1}=M$ for some arbitrarily large cost M. Let $d_{i,j}=2$ for all other edges.
  - Greedy algorithm picks all edges of weight 1, but is then forced to pick weight M. Solution can be arbitrarily bad!
  - Optimal tour has length $n+2$, e.g. $1, 2, 3, \ldots, n\text{-}2, n, n\text{-}1, 1$

0-1 Knapsack problem

- A thief robbing a store finds *n* items. The *i*-th item is worth v_i Yuan (元) and weighs w_i Grams (all integers). The thief can only carry at most *W* grams in his knapsack. Which items should he take to maximise profit?
- Called 0-1 because the thief can either take or leave items.
- What would a greedy approach look like?
  1. Sort items according to value per gram.
  2. Try to add items to the knapsack in this order.
- Have a guess: does this greedy approach always work?

0-1 Knapsack problem: greedy fails



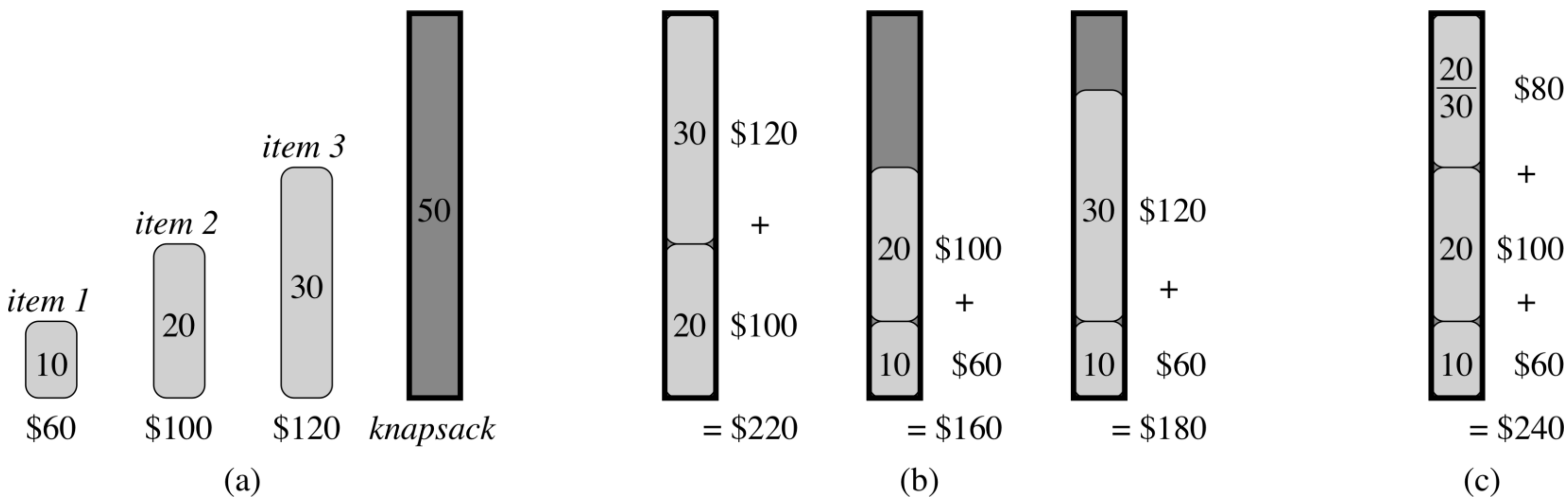
Fractional Knapsack problem

- Assume the thief can take fractions of items (e.g. stealing cheese)

- Will the greedy strategy work?

Greedy works for fractional knapsack

- Greedy algorithm takes the best possible value per weight.



Summary

- Greedy algorithms make “greedy” local choices that hopefully lead to globally optimal solutions.
- Greedy algorithms work well for activity selection, coin changing, fractional knapsack and many other problems (more examples coming up later).
- Greedy algorithms may fail badly. For the Travelling Salesman Problem (TSP) we saw an instance class where the solution quality can be arbitrarily bad.
- Greedy fails for 0-1 Knapsack, but works for the (easier) fractional knapsack problem.