Lecture 28: Greedy Algorithms I

2023/10/19

詹博华（中国科学院软件研究所）

Greedy Algorithms

- An alternative to dynamic programming for solving optimization problems.
- Make the choice that is **locally optimal** at each step, yielding a **globally optimal** solution.

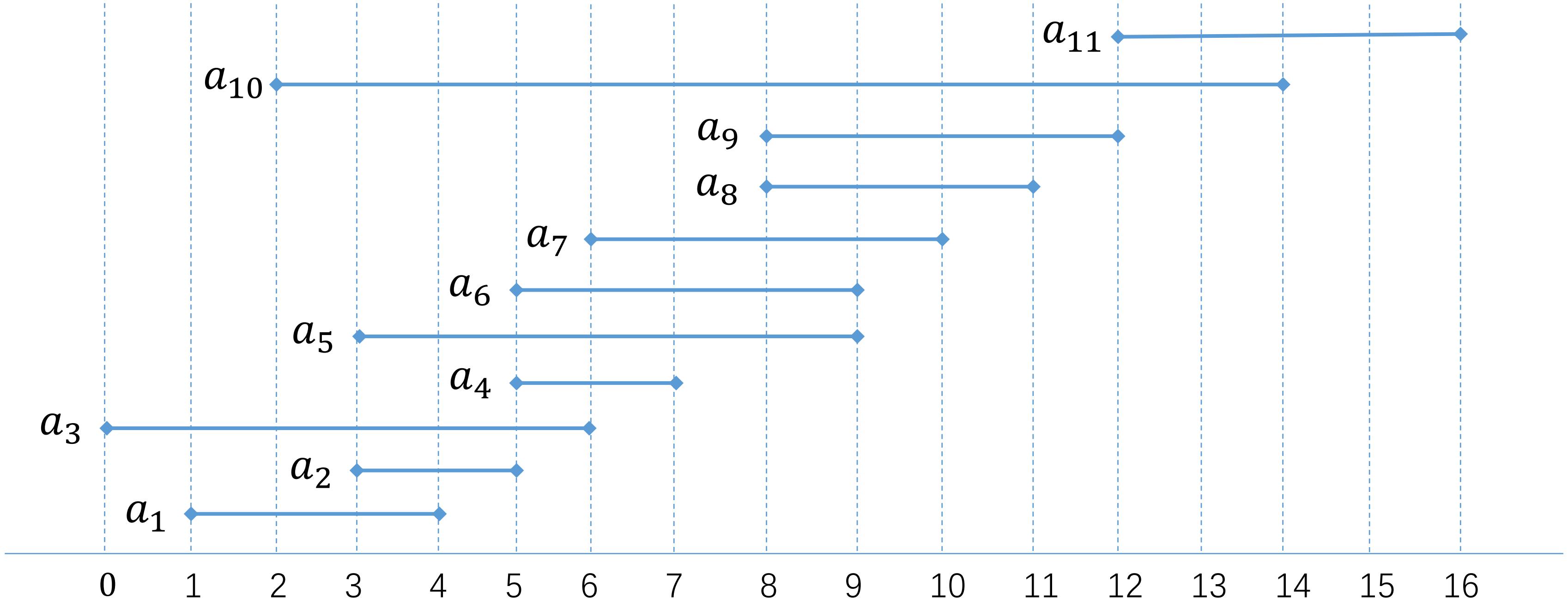
Example: Activity-selection problem

- Suppose we have a set of activities $S = \{a_1, a_2, \dots, a_n\}$. Each activity has a start time s_i and finish time f_i, where $0 \le s_i < f_i$.
- Activity a_i takes place in the half-open interval $[s_i, f_i)$.
- Two activities a_i and a_j are compatible if $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap. That is, if $s_i \ge f_j$ or $s_j \ge f_i$.
- **Activity-selection problem:** select a maximum-size subset of mutually compatible activities.
- **To start: sort the activities by their finish time.**

Activity-selection problem: example

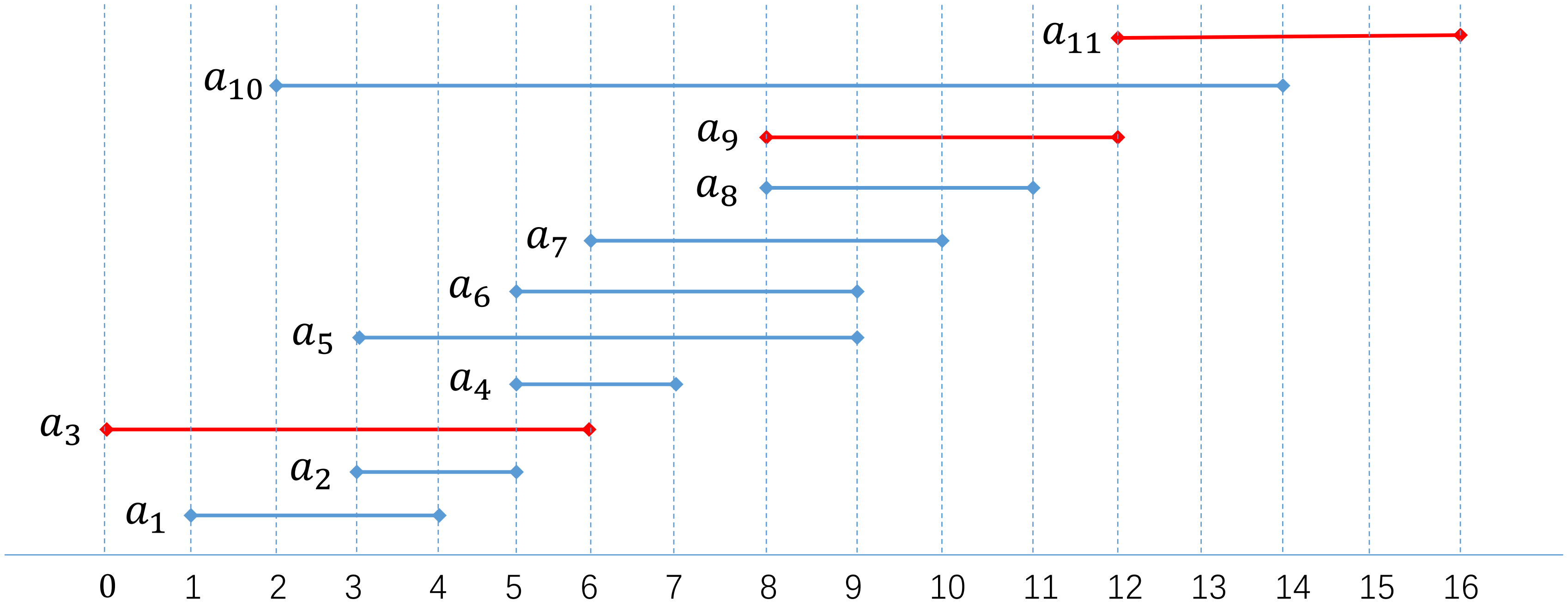
- Consider the following example:

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



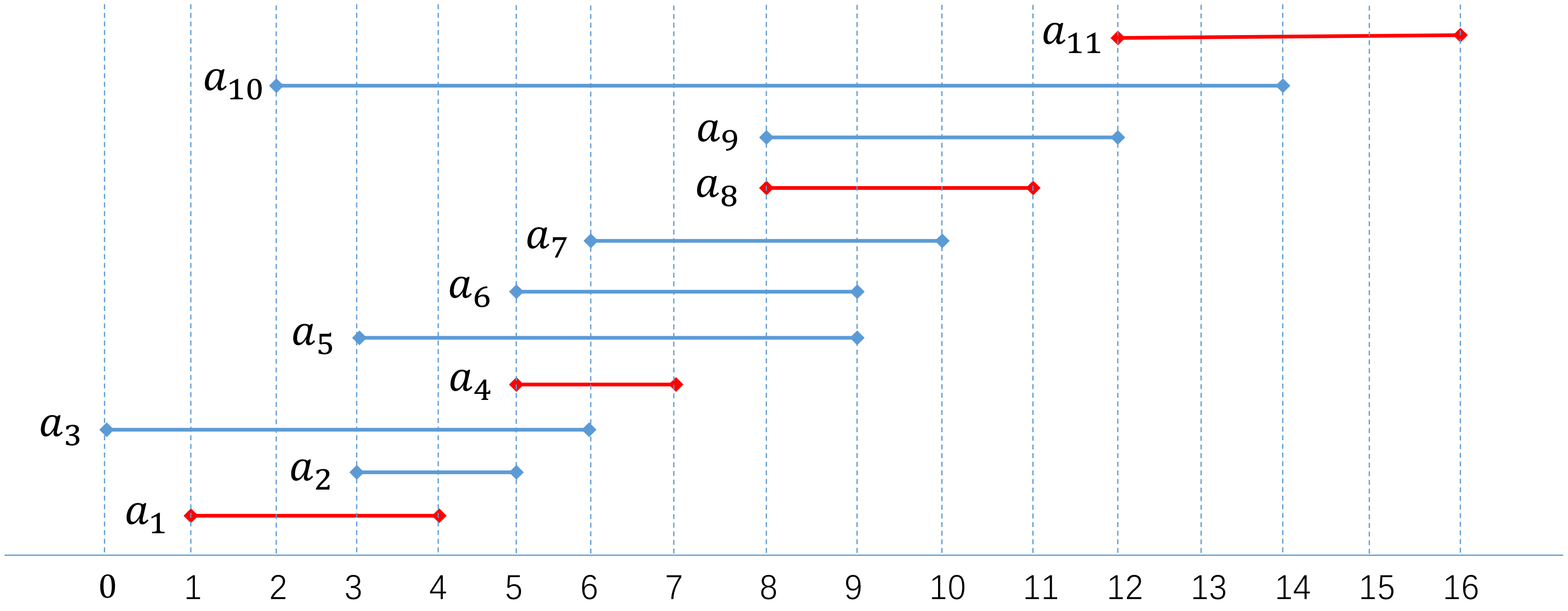
Activity-selection problem: example

- Possible choice of subset: $\{a_3, a_9, a_{11}\}$ (not maximum).



Activity-selection problem: example

- An maximum solution: $\{a_1, a_4, a_8, a_{11}\}$. Note there are several alternative maximum solutions.



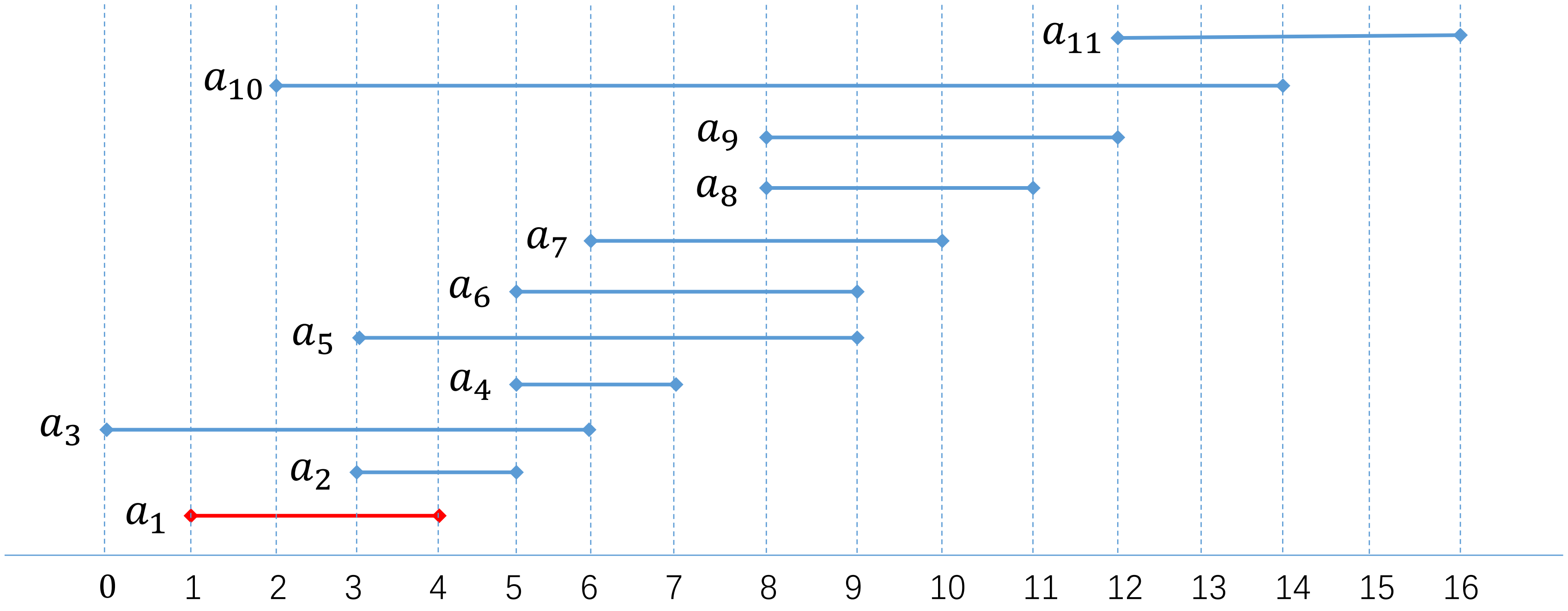
Greedy solution

- At each step, choose the activity with the earliest finishing time that can be added to the subset.

```
GREEDY-ACTIVITY-SELECTOR(s, f)
n = s.length
A = {a_1}
k = 1
for m = 2 to n
    if s[m] ≥ f[k]
        A = A ∪ {a_m}
        k = m
return A
```

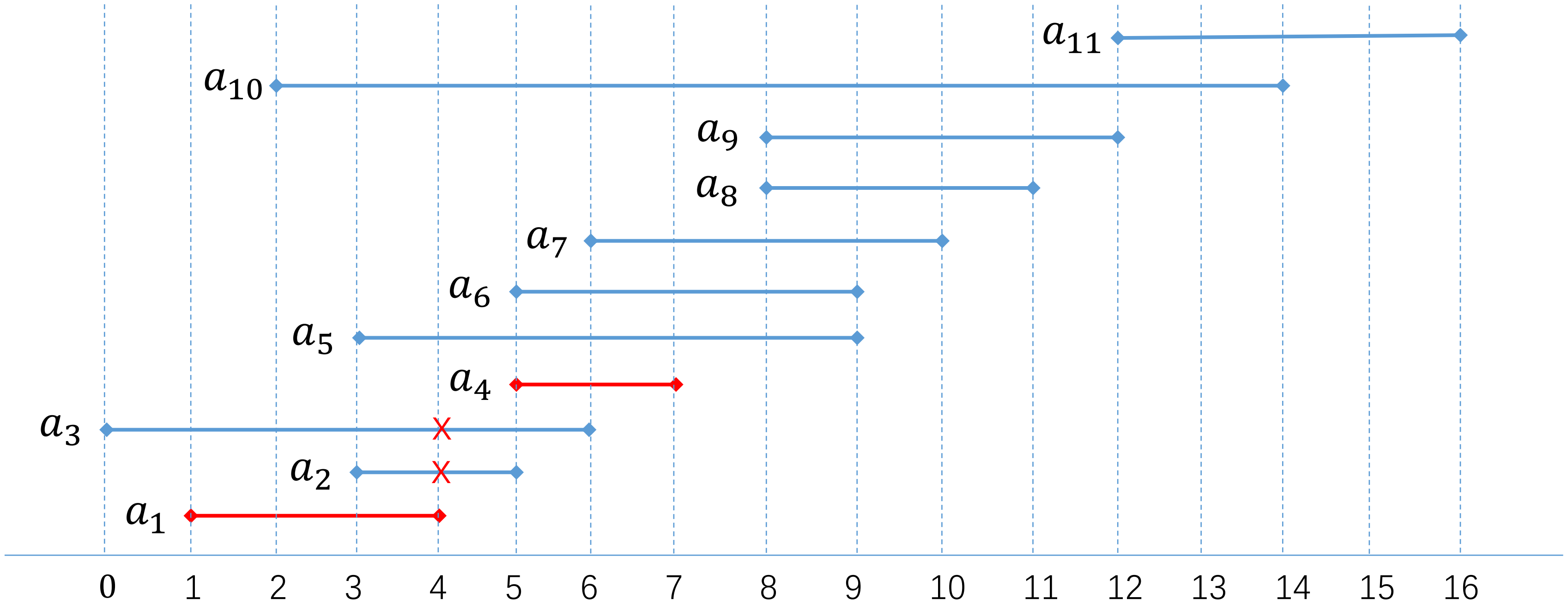
Greedy solution

- Step 1: add a_1, which finishes at time 4.



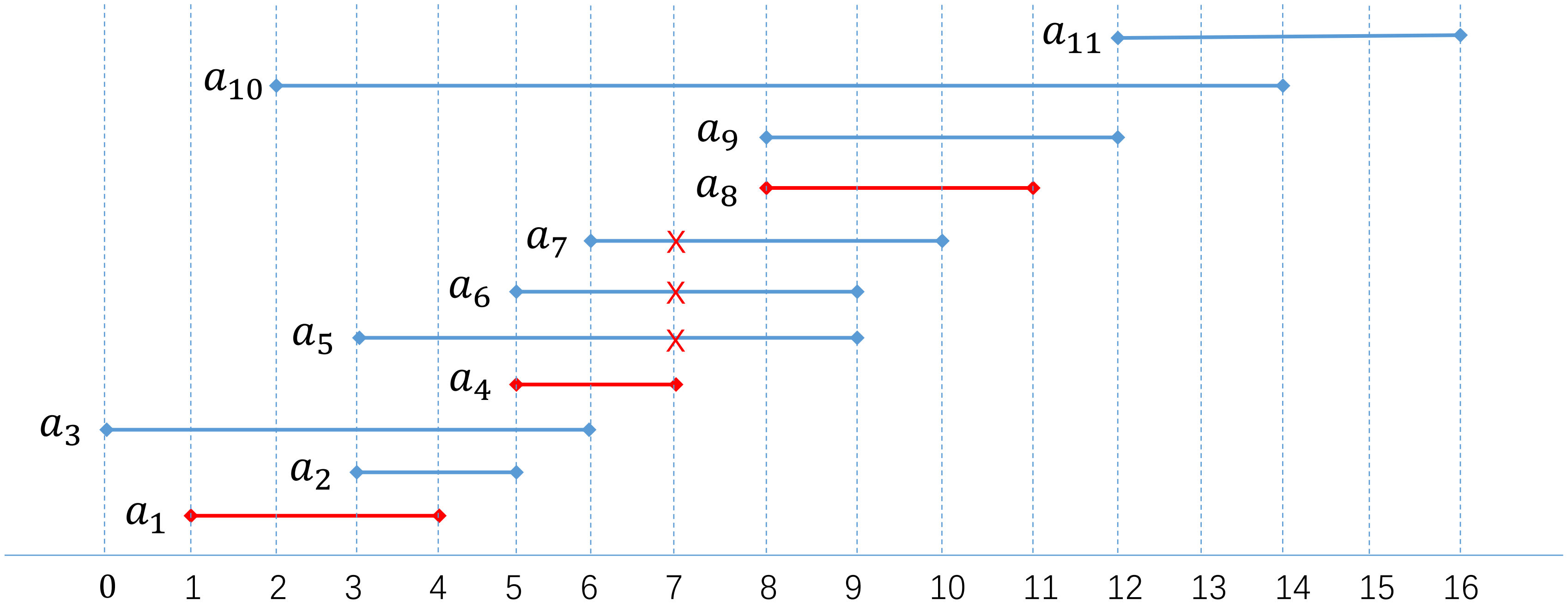
Greedy solution

- Step 2: neither a_2 and a_3 can be added. The next task that can be added is a_4, which finishes at time 7.



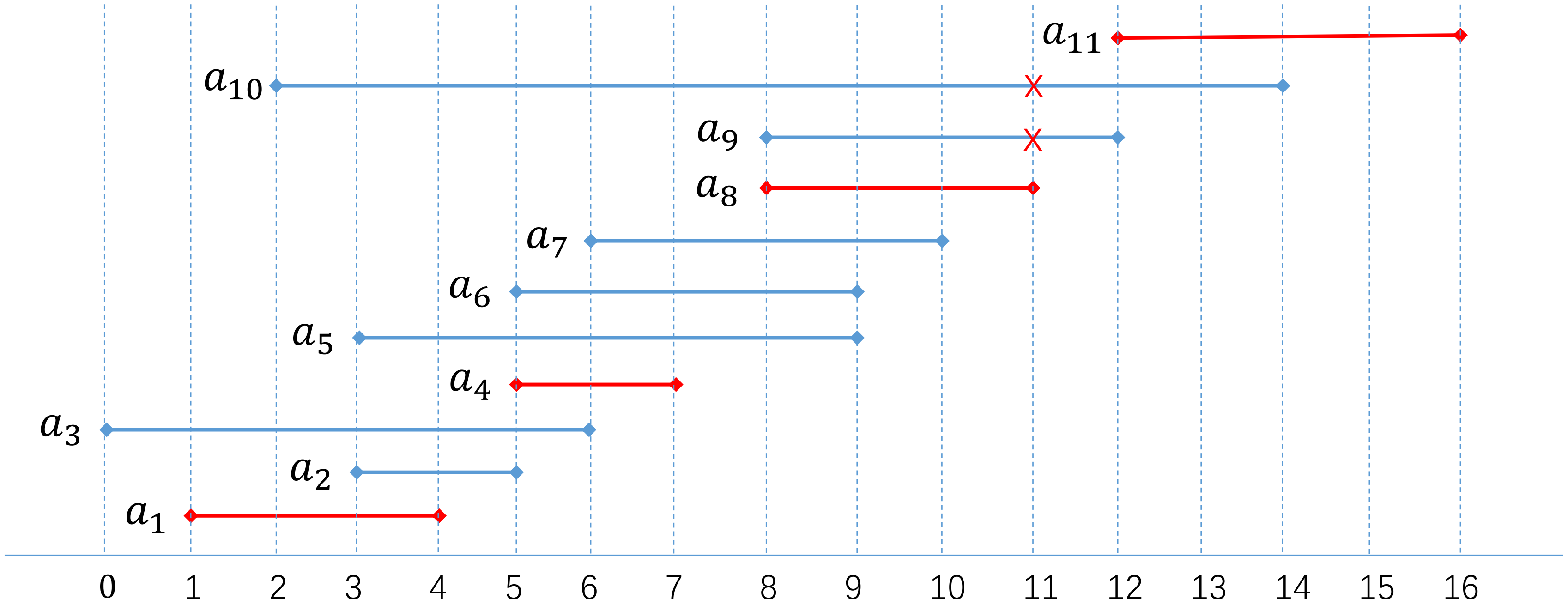
Greedy solution

- Step 3: neither a_5, a_6, a_7 cannot be added. The next task that can be added is a_8, which finishes at time 11.



Greedy solution

- Step 4: neither a_9 and a_{10} can be added, so the final task to be added is a_{11}.

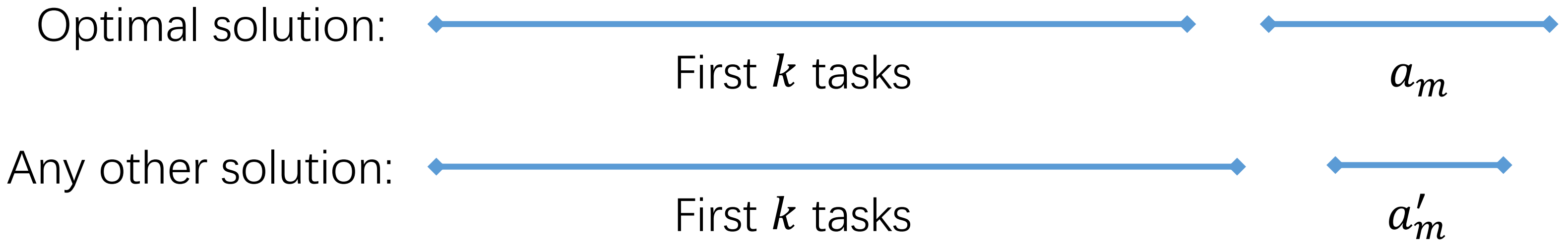


Proof of correctness

- Why is the above algorithm correct?
- Consider the subproblem: what is the earliest time to finish n tasks, for each $n \geq 1$? Denote this by t_n.
- Solution for $n = 1$: pick the task that finishes earliest (that is, a_1).

Proof of correctness

- Solution for $n = k + 1$: start from the solution for $n = k$, then pick the next task (according to finish time) that can be added. Suppose this is $a_m = [s_m, f_m)$.
- If any other solution can do better, then its last task $a_{m'} = [s_{m'}, f_{m'})$ must satisfy $s_{m'} \geq t_k$ and $f_{m'} < f_m$. But then $a_{m'}$ would be chosen rather than a_m in the algorithm, contradiction.



But then, a'_m should be chosen instead of a_m in the algorithm.

Greedy vs. Dynamic Programming

- When to use greedy algorithm vs. dynamic programming?
- Use greedy algorithm when it works: when it can be shown that making locally optimal choices gives the optimal global choice.
- Otherwise, consider dynamic programming.
- Whether greedy algorithm works can be very subtle.

Example: knapsack problem

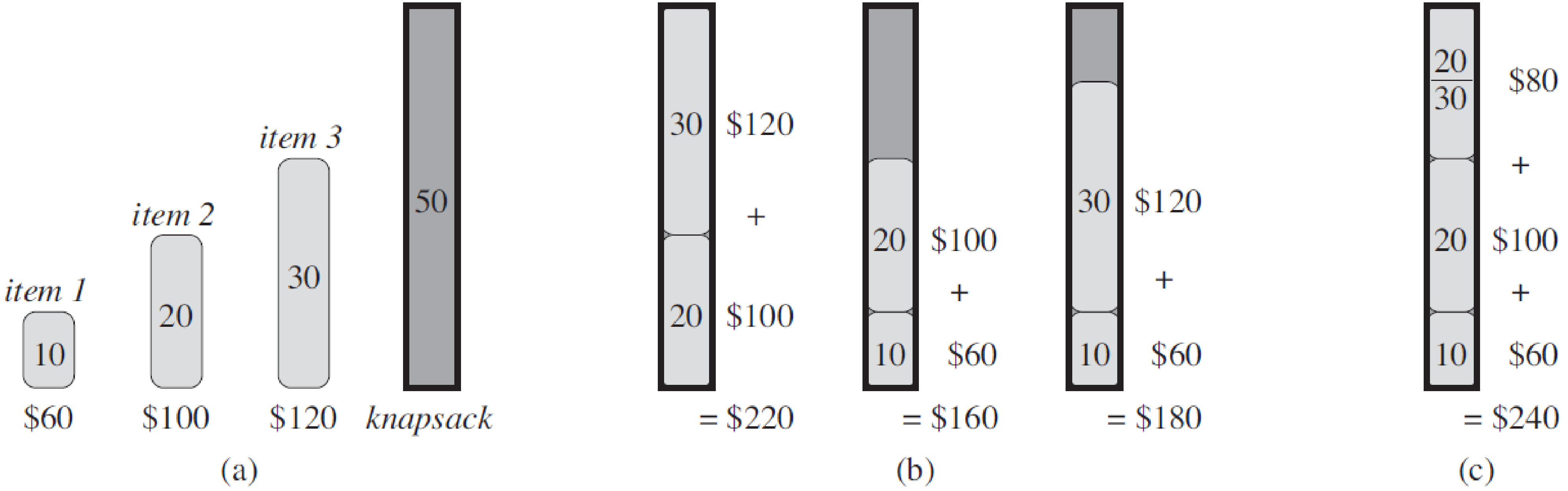
- Given a list of n items, the i^{th} item is worth v_i dollars and weights w_i pounds. You can carry a bag (a *knapsack*) with at most W pounds. What is the most value you can carry?
- 0-1 knapsack problem: either take the item or leave it behind.
- Fractional knapsack problem: can take a fraction of an item.

Knapsack problem: examples

(a): available items.

(b): optimal solution and two suboptimal solutions for 0-1 knapsack problem.

(c): optimal solution for the fractional knapsack problem.



Fractional knapsack problem: greedy solution

- Sort the items by their value per pound.
- Start filling the bag with most valuable item per pound, then the next, etc.
- Example: with the items given previously:

  ($\$60$, 10lbs), ($\$100$, 20lbs), ($\$120$, 30lbs), knapsack: 50lbs
- Their values per pound are:

  $\$6$/lbs, $\$5$/lbs, $\$4$/lbs
- So fill in item 1 first, then item 2, and use the remaining weight for part of item 3, yielding the solution shown in (c).

0-1 knapsack problem: greedy does not work

- For the 0-1 knapsack problem, we cannot follow the same approach.
- Example: with the items given previously:

  ($\$60$, 10lbs), ($\$100$, 20lbs), ($\$120$, 30lbs), knapsack: 50lbs
- The optimal solution is to put the second and third item, even though the first item has the best value per pound.
- Dynamic programming should be used for the 0-1 knapsack problem (obtain algorithm with complexity $O(nW)$).