Design and Analysis of Algorithms

CSE 5311

Lecture 16 Greedy algorithms

Junzhou Huang, Ph.D.

Department of Computer Science and Engineering

Overview

- A greedy algorithm always makes the choice that looks best at the moment
  - Make a locally optimal choice in hope of getting a globally optimal solution
  - Example: Play cards, Invest on stocks, etc.
- Do not always yield optimal solutions
- They do for some problems with optimal substructure (like Dynamic Programming)
- Easier to code than DP

An Activity-Selection Problem

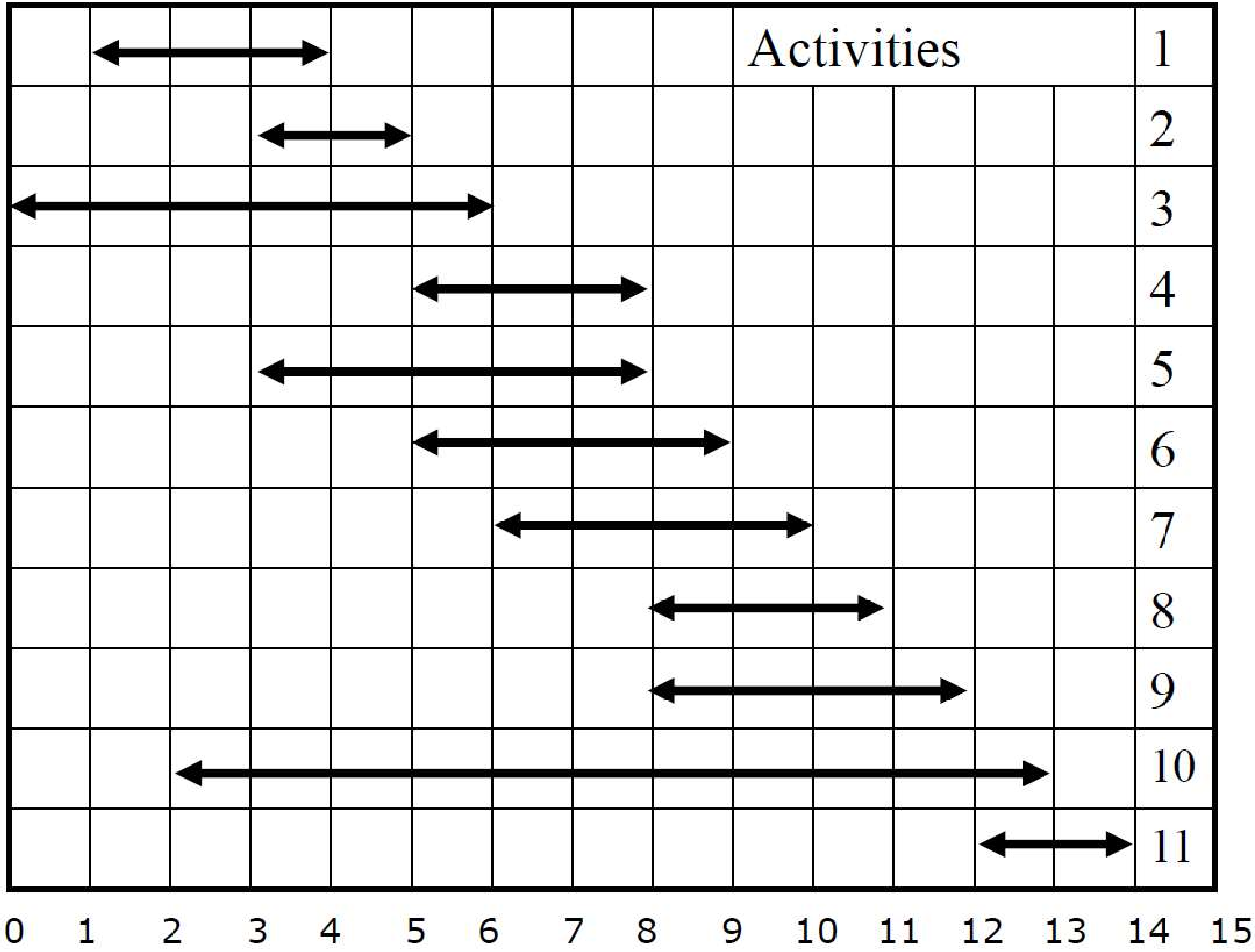
- **Input**: Set S of n activities, $a_1, a_2, \ldots, a_n$.
  - Activity a_i starts at time s_i and finishes at time f_i
  - Two activities are compatible, if their intervals don't overlap.
- **Output:** A subset of maximum number of mutually compatible activities.
  - **Assume:** activities are sorted by finishing times $f_1 \leq f_2 \leq \ldots \leq f_n$
  - **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	8	8	9	10	11	12	13	14

  - Possible sets of mutually compatible activities
  - $\{a_3, a_9, a_{11}\}$
  - $\{a_1, a_4, a_8, a_{11}\}$
  - $\{a_2, a_4, a_9, a_{11}\}$

Largest subsets

An Activity-Selection Problem



Optimal Substructure

- Suppose an optimal solution includes activity $\mathbf{a_k}$.
  Two subproblems:

$\mathbf{a_1, \ldots, a_{k-1},}$ $\mathbf{a_{k,}}$ $\mathbf{a_{k+1,} \ldots, a_n}$

1. Select compatible activities that finish before a_k starts

2. Select compatible activities that finish after a_k starts

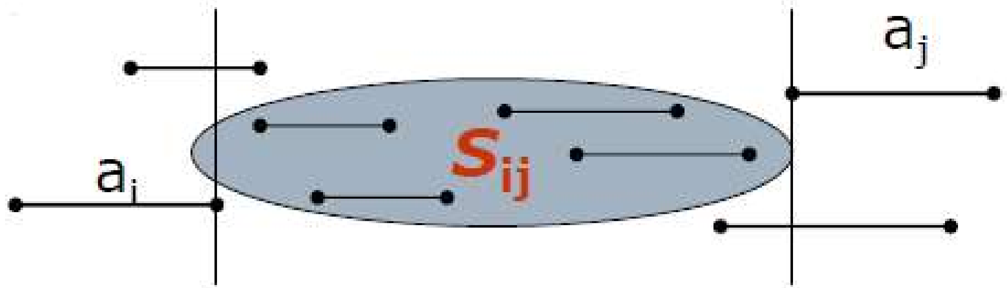
- The solution to the two subproblems must be optimal (prove using cut-and-paste argument)

Cut-and-Paste

- The "cut and paste" technique is a way to prove that a problem has the property.
  - In particular, you want to show that when you come up with an optimal solution to a problem, you have necessarily used optimal solutions to the constituent subproblems.
- The proof is by contradiction.
  - Suppose you came up with an optimal solution to a problem by using suboptimal solutions to subproblems.
  - Then, if you were to replace ("cut") those suboptimal subproblem solutions with optimal subproblem solutions (by "pasting" them in), you would improve your optimal solution.
  - But, since your solution was optimal by assumption, you have a contradiction.

Recursive Solution

- S_{ij} : subset of activities that start after $\mathbf{a_i}$ and finish before $\mathbf{a_j}$ starts



- Subproblems: find c[i, j], maximum number of mutually compatible activities from
- Recurrence:

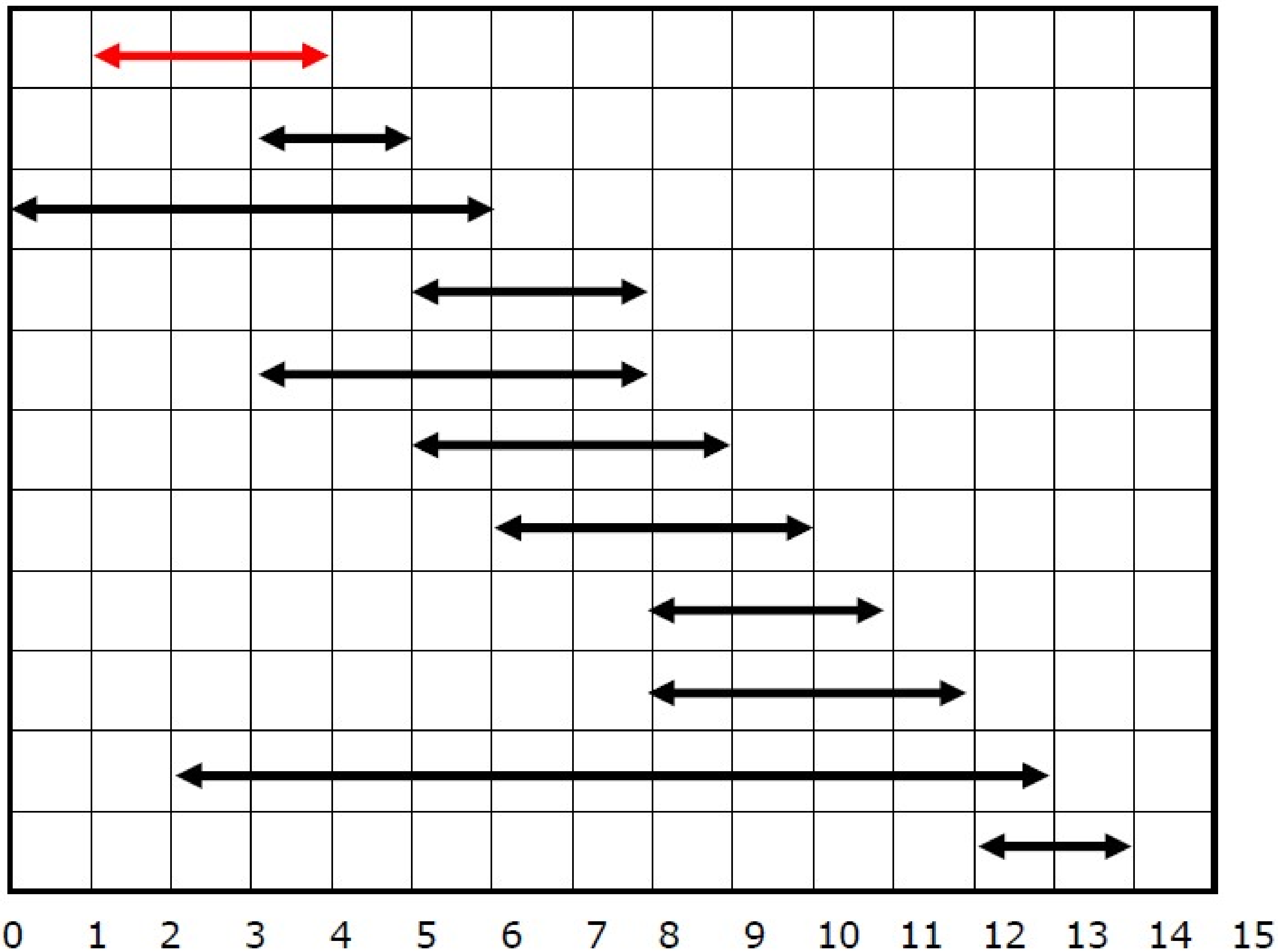
$$c[i,j] = \begin{cases} 0 & \text{if } S_{ij} = \emptyset \\ \max\limits_{a_k \in S_{ij}} \{c[i,k] + c[k,j] + 1\} & \text{if } S_{ij} \neq \emptyset \end{cases}$$

Early Finish Greedy

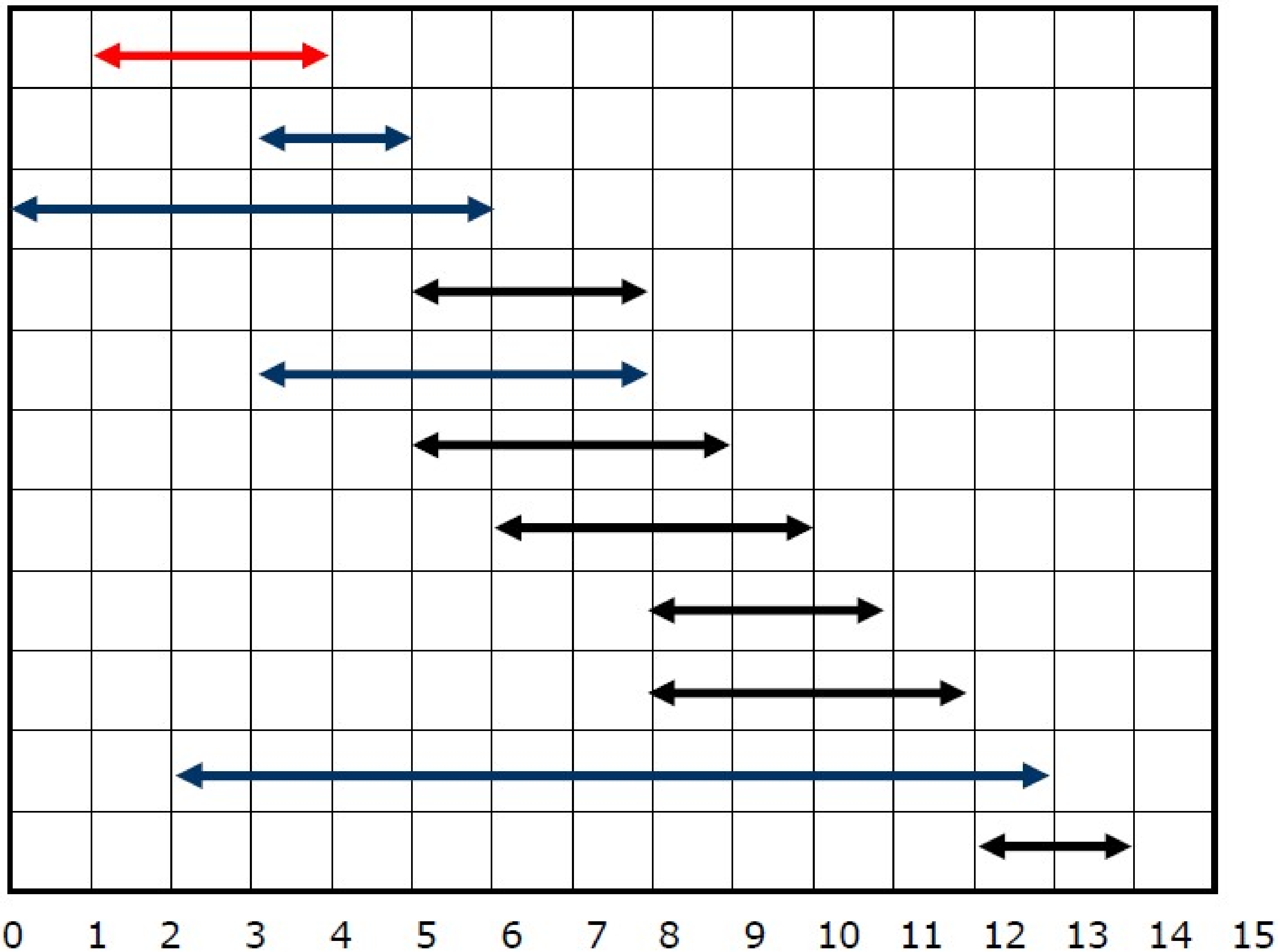
1. **while**(activities)
2. Select the activity with the earliest finish
3. Remove the activities that are not compatible
4. **end while**

- **Greedy in the sense that:**
  - It leaves as much opportunity as possible for the remaining activities to be scheduled.
  - Maximizes the amount of unscheduled time remaining

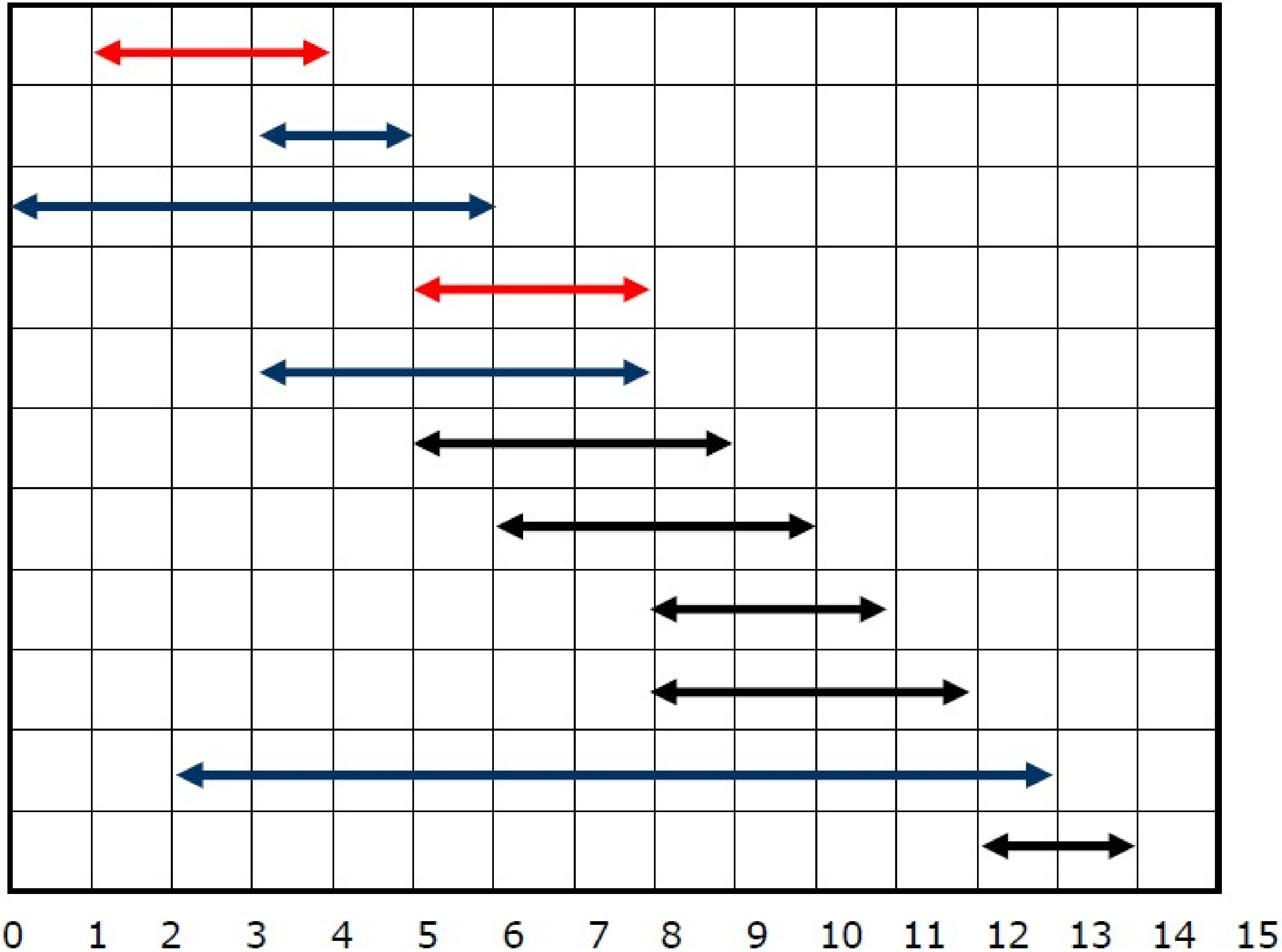
Example



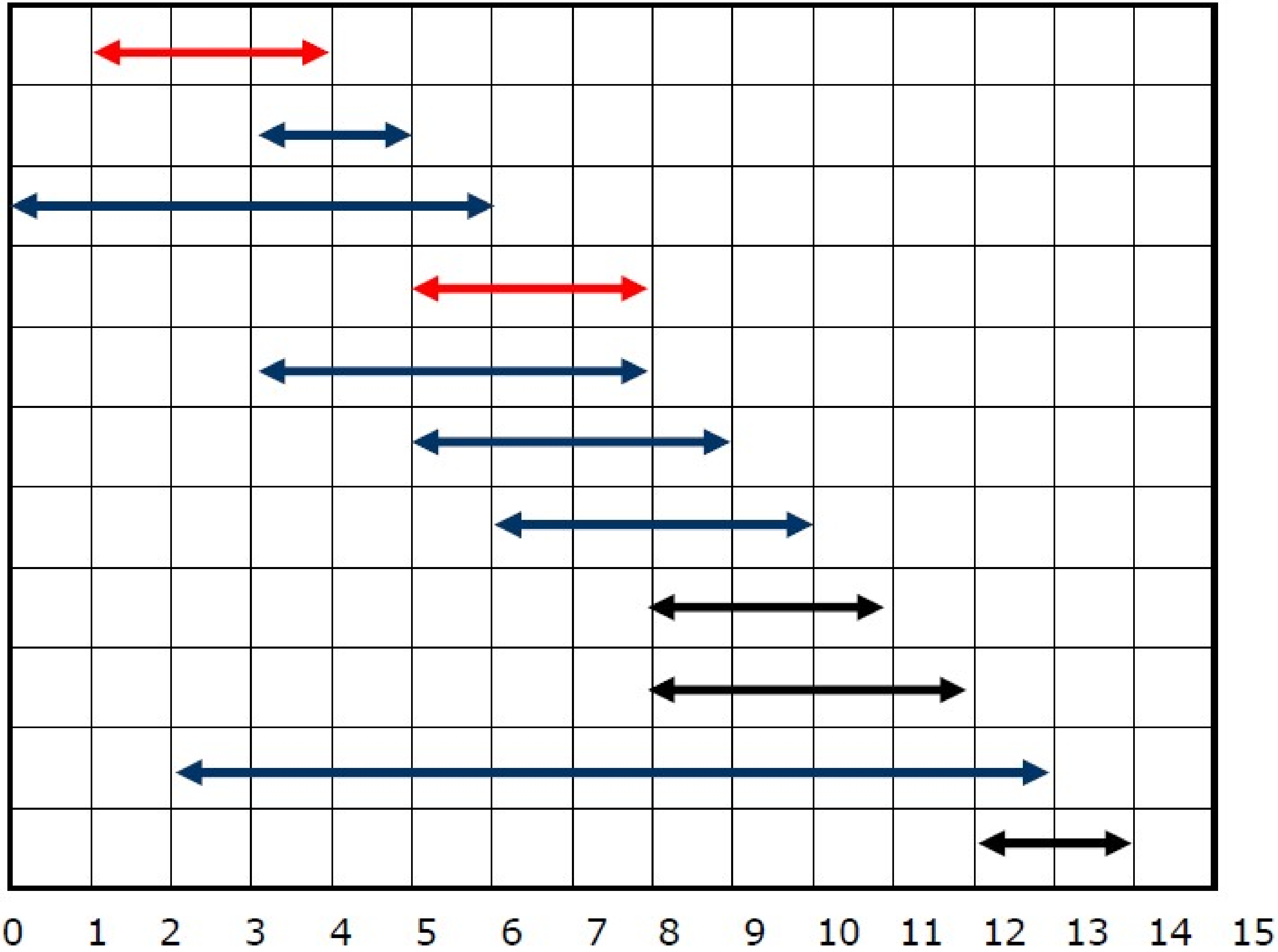
Example



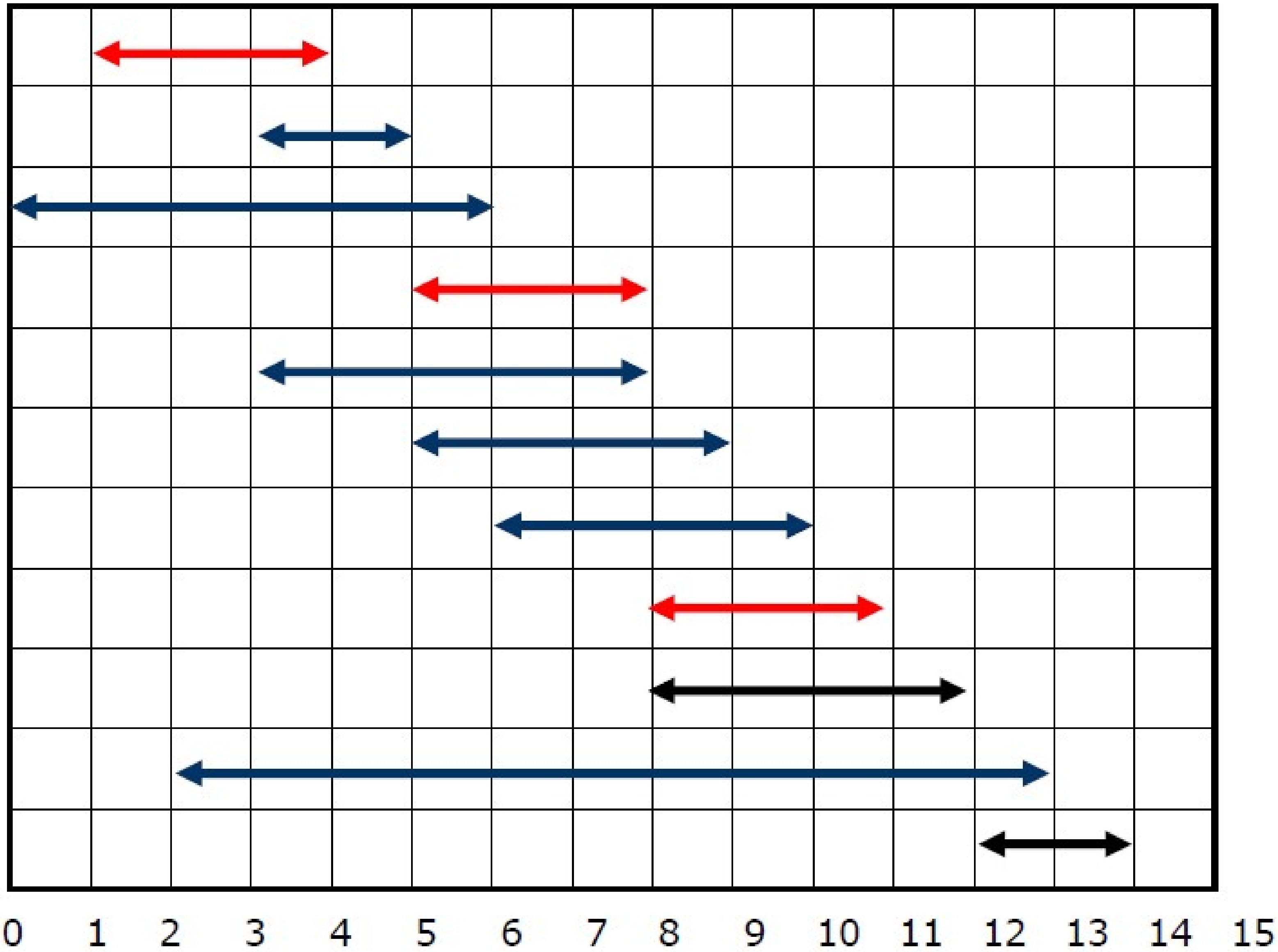
Example



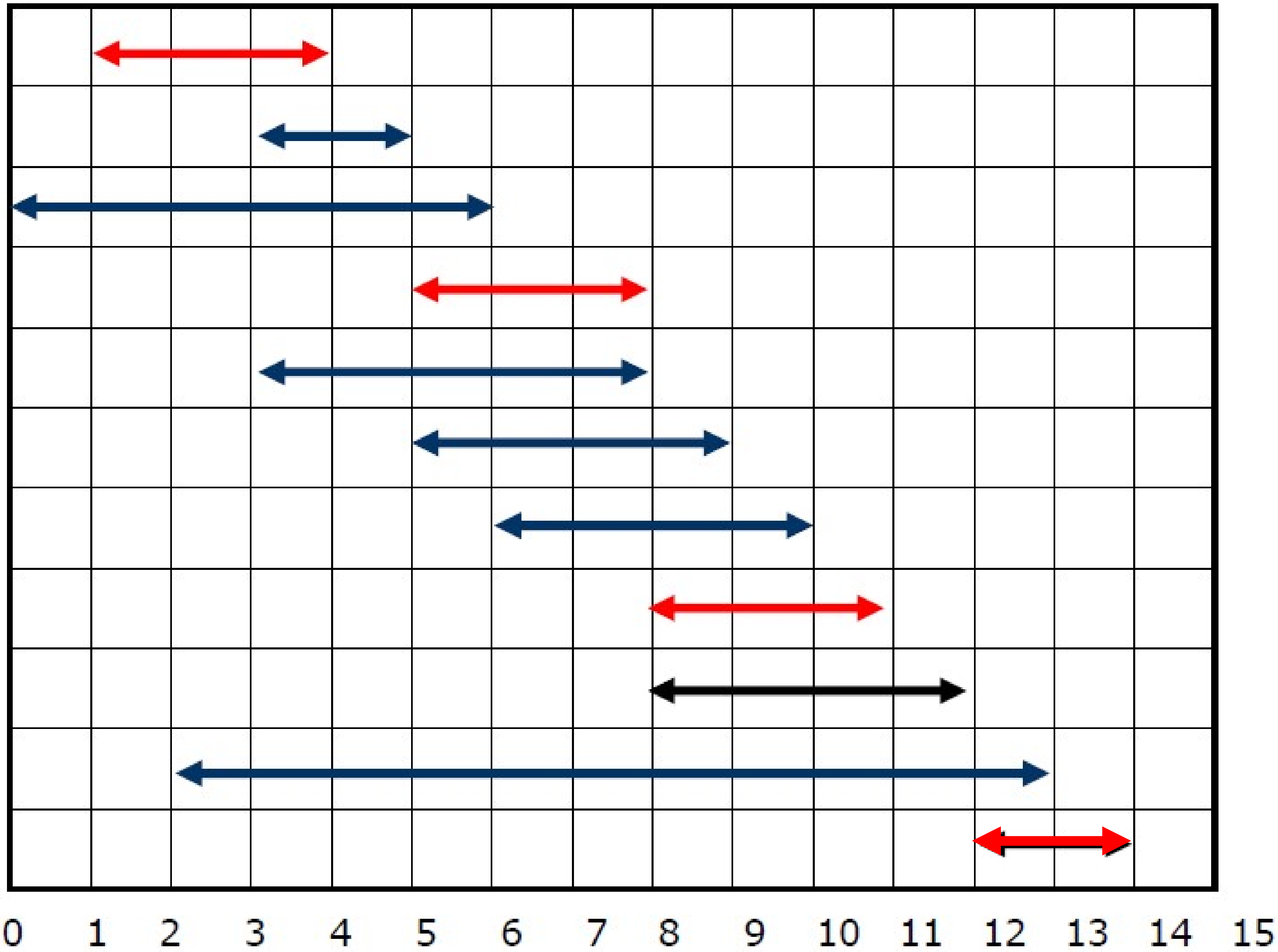
Example



Example



Example



Greedy Choice Property

- **Locally optimal choice, we then get globally optimal solution**
  - **Them 16.1**: if S is an non-empty activity-selection subproblem, then there exists optimal solution A in S such that a_{s1} in A, where a_{s1} is the earliest finish activity in A
  - **Sketch of proof**: if there exists optimal solution B that does not contain a_{s1} , can always replace the first activity in B with a_{s1}. Same number of activities, thus optimal.

Recursive Algorithm

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 Ø
```

- **Initial call:** RECURSIVE-ACTIVITY-SELECTOR$(s, f, 0, n)$.
- **Complexity:** $\Theta(n)$
- **Straightforward to convert to an iterative one**

Iterative Algorithm

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
```

- **Initial call:** GREEDY-ACTIVITY-SELECTOR(s, f)
- **Complexity:** $\Theta(n)$

Elements of the greedy strategy

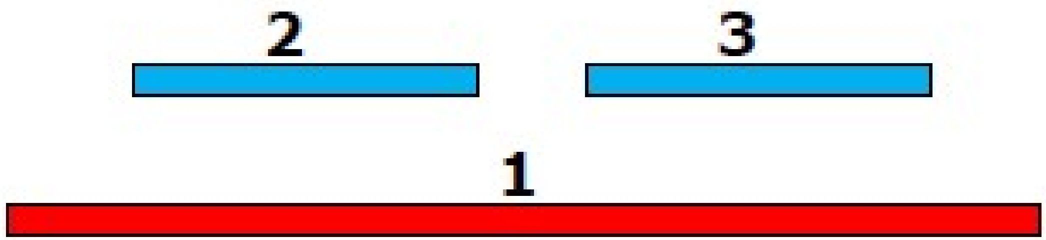
- Determine the optimal substructure
- Develop the recursive solution
- Prove that if we make the greedy choice, only one subproblem remains
- Prove that it is safe to make the greedy choice
- Develop a recursive algorithm that implements the greedy strategy
- Convert the recursive algorithm to an iterative one.

Not All Greedy are Equal

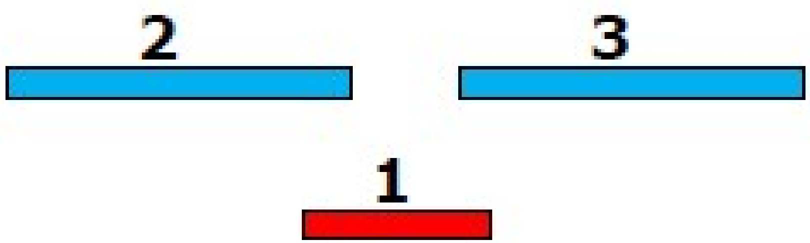
- **Earliest finish time:** Select the activity with the earliest finish time ⇐ **Optimal**
- **Earliest start time:** Select activity with the earliest start time
- **Shortest duration:** Select activity with the shortest duration $\mathbf{d_i = f_i - s_i}$
- **Fewest conflicts:** Select activity that conflicts with the least number of other activities first
- **Last start time:** Select activity with the last start

Not All Greedy are Equal

- **Earliest start time:** Select activity with the earliest start time

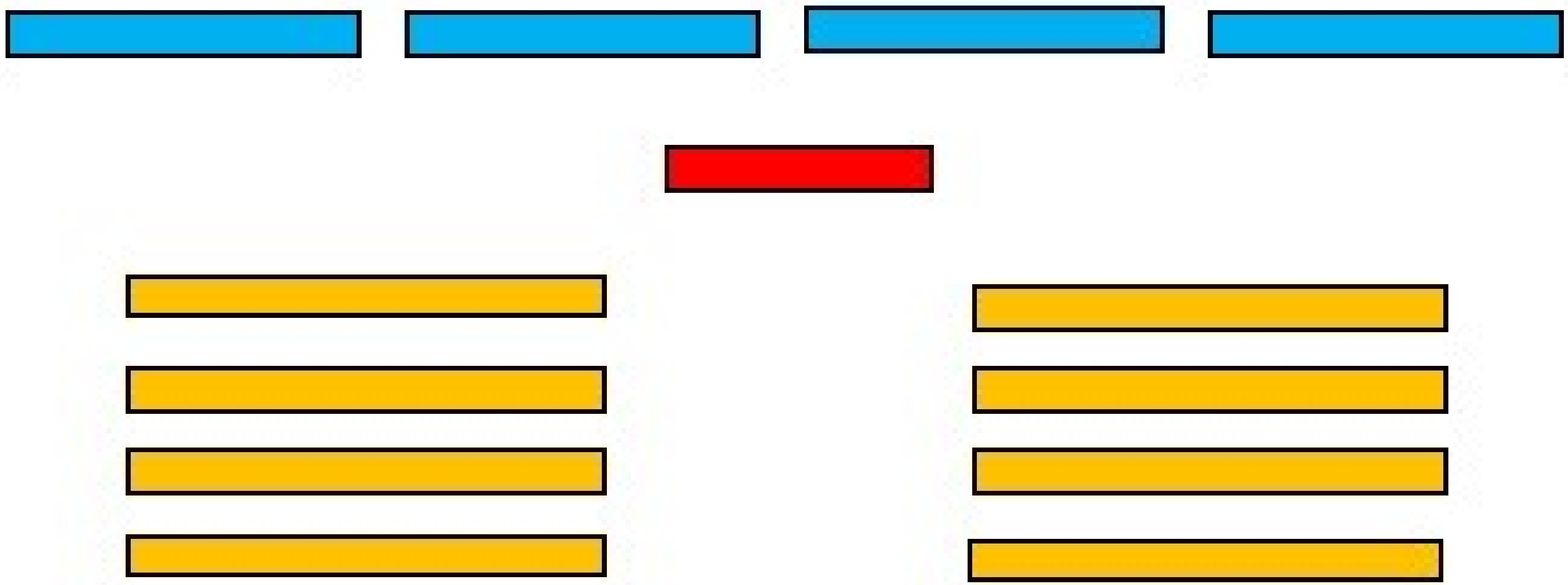


- **Shortest duration:** Select activity with the shortest duration $\mathbf{d_i = f_i - s_i}$



Not All Greedy are Equal

- **Fewest conflicts:** Select activity that conflicts with the least number of other activities first



Dynamic Programming vs. Greedy Algorithms

- **Optimization problems**
  - Dynamic programming, but overkill sometime.
  - Greedy algorithm:
    - Being greedy for local optimization with the hope it will lead to a global optimal solution, not always, but in many situations, it works.

Example: An Activity-Selection Problem

- Suppose A set of activities S={a_1, a_2,…, a_n}
  - They use resources, such as lecture hall, one lecture at a time
  - Each a_i, has a start time s_i, and finish time f_i, with $0\leq s_i< f_i<\infty$.
  - a_i and a_j are compatible if $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap
- **Goal:** select maximum-size subset of mutually compatible activities.
- Start from dynamic programming, then greedy algorithm, see the relation between the two.

DP solution –step 1

- Optimal substructure of activity-selection problem.
  - Furthermore, assume that $f_1 \leq \ldots \leq f_n$.
  - Define $S_{ij}=\{a_k: f_i \leq s_k < f_k \leq s_j\}$, i.e., all activities starting after a_i finished and ending before a_j begins.
  - Define two fictitious activities a_0 with $f_0=0$ and a_{n+1} with $s_{n+1}=\infty$
    - So $f_0 \leq f_1 \leq \ldots \leq f_{n+1}$.
  - Then an optimal solution including a_k to S_{ij} contains within it the optimal solution to S_{ik} and S_{kj}.

DP solution –step 2

- A recursive solution
- Assume c[n+1,n+1] with c[i,j] is the number of activities in a maximum-size subset of mutually compatible activities in S_{ij}. So the solution is $c[0,n+1]=S_{0,n+1}$.
- $$C[i,j]=\begin{cases} 0 & \text{if } S_{ij}=\varnothing \\ \max\limits_{i<k<j \text{ and } a_k\in S_{ij}}\{c[i,k]+c[k,j]+1\} & \text{if } S_{ij}\neq\varnothing \end{cases}$$

Converting DP Solution to Greedy Solution

- **Theorem 16.1:** consider any nonempty subproblem S_{ij}, and let a_m be the activity in S_{ij} with earliest finish time: $f_m = \min\{f_k : a_k \in S_{ij}\}$, then

1. Activity a_m is used in some maximum-size subset of mutually compatible activities of S_{ij}.
2. The subproblem S_{im} is empty, so that choosing a_m leaves S_{mj} as the only one that may be nonempty.

- Proof of the theorem:

Top-Down Rather Than Bottom-Up

- To solve S_{ij}, choose a_m in S_{ij} with the earliest finish time, then solve S_{mj}, (S_{im} is empty)
- It is certain that optimal solution to S_{mj} is in optimal solution to S_{ij}.
- No need to solve S_{mj} ahead of S_{ij}.
- Subproblem pattern: $S_{i,n+1}$.

Optimal Solution Properties

- In DP, optimal solution depends:
  - How many subproblems to divide. (2 subproblems)
  - How many choices to determine which subproblem to use. (j-i-1 choices)
- However, the above theorem (16.1) reduces both significantly
  - One subproblem (the other is sure to be empty).
  - One choice, i.e., the one with earliest finish time in S_{ij}.
  - Moreover, top-down solving, rather than bottom-up in DP.
  - Pattern to the subproblems that we solve, $S_{m,n+1}$ from S_{ij}.
  - Pattern to the activities that we choose. The activity with earliest finish time.
  - With this local optimal, it is in fact the global optimal.

Elements of Greedy Strategy

- Determine the optimal substructure
- Develop the recursive solution
- Prove one of the optimal choices is the greedy choice yet safe
- Show that all but one of subproblems are empty after greedy choice
- Develop a recursive algorithm that implements the greedy strategy
- Convert the recursive algorithm to an iterative one.

Greedy vs. DP

- Knapsack problem: a thief robbing a store and find n items
  - I_1 (v_1,w_1), I_2 (v_2,w_2),…,$I_n(v_n,w_n)$.
  - The i-th item is worth v_i dollars and weight w_i pound
  - Given a weight W at most he can carry,
  - Find the items which maximize the values
  - Which items should the thief take to obtain the maximum amount of money?
- Fractional knapsack,
  - Fractional of items can be taken
  - Greed algorithm, O(nlogn)
- 0/1 knapsack.
  - Each item is taken or not taken
  - DP, O(nW). (Questions: 0/1 knapsack is an NP-complete problem, why O(nW) algorithm?)

Knapsack Problem

- Both exhibit the optimal-substructure property
  - **0-1**: If item j is removed from an optimal packing, the remaining packing is an optimal packing with weight at most $W\text{-}w_j$
  - **Fractional**: If w *pounds* of item j is removed from an optimal packing, the remaining packing is an optimal packing with weight at most $W\text{-}w$ that can be taken from other $n\text{-}1$ items plus $w_j - w$ of item j

Fractional Knapsack Problem

- Can be solvable by the greedy strategy
  - Compute the value per pound v_j/w_j for each item
  - Obeying a greedy strategy, take as much as possible of the item with the greatest value per pound.
  - If the supply of that item is exhausted and there is still more room, take as much as possible of the item with the next value per pound, and so forth until there is no more room
  - $O(n \lg n)$ (we need to sort the items by value per pound)
-

0-1 Knapsack Problem

- Much harder
  - Cannot be solved by the greedy strategy. Counter example?
  - We must compare the solution to the sub-problem in which the item is included with the solution to the sub-problem in which the item is excluded before we can make the choice
  - Dynamic Programming (See previous lectures)

Counter Example

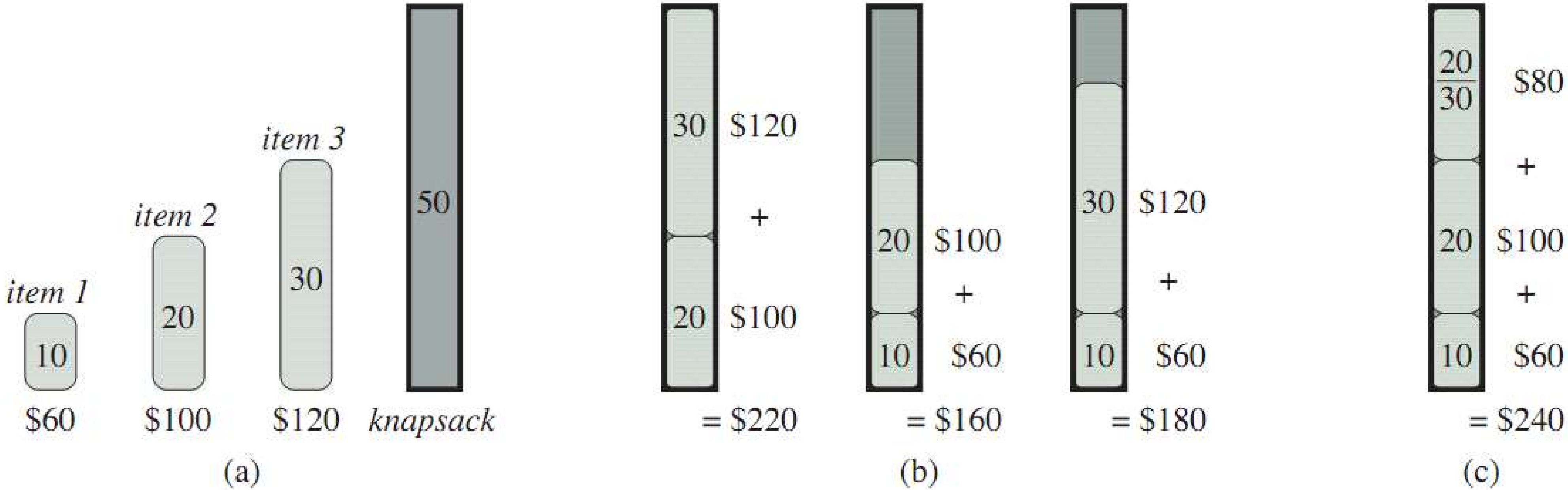


Figure 16.2 An example showing that the greedy strategy does not work for the 0-1 knapsack problem. **(a)** The thief must select a subset of the three items shown whose weight must not exceed 50 pounds. **(b)** The optimal subset includes items 2 and 3. Any solution with item 1 is suboptimal, even though item 1 has the greatest value per pound. **(c)** For the fractional knapsack problem, taking the items in order of greatest value per pound yields an optimal solution.