Lectures 20-21.
Greedy Algorithms

Introduction to Algorithms
Sungkyunkwan University

Hyunseung Choo
choo@skku.edu

Greedy Algorithms

- A *greedy algorithms* always make the choice that looks best at the moment
  - My everyday examples
    - Playing cards
    - Invest on stocks
    - Choose a university
  - The hope
    - A locally optimal choice will lead to a globally optimal solution

Introduction

- Similar to Dynamic Programming
- It applies to Optimization Problem
- When we have a choice to make, make the one that looks *best right now*
  - Make a *locally optimal choice* in hope of getting a *globally optimal solution*
- Greedy algorithms don't always yield an optimal solution, but sometimes they do
  - For many problems, it provides an optimal solution much more quickly than a dynamic programming approach

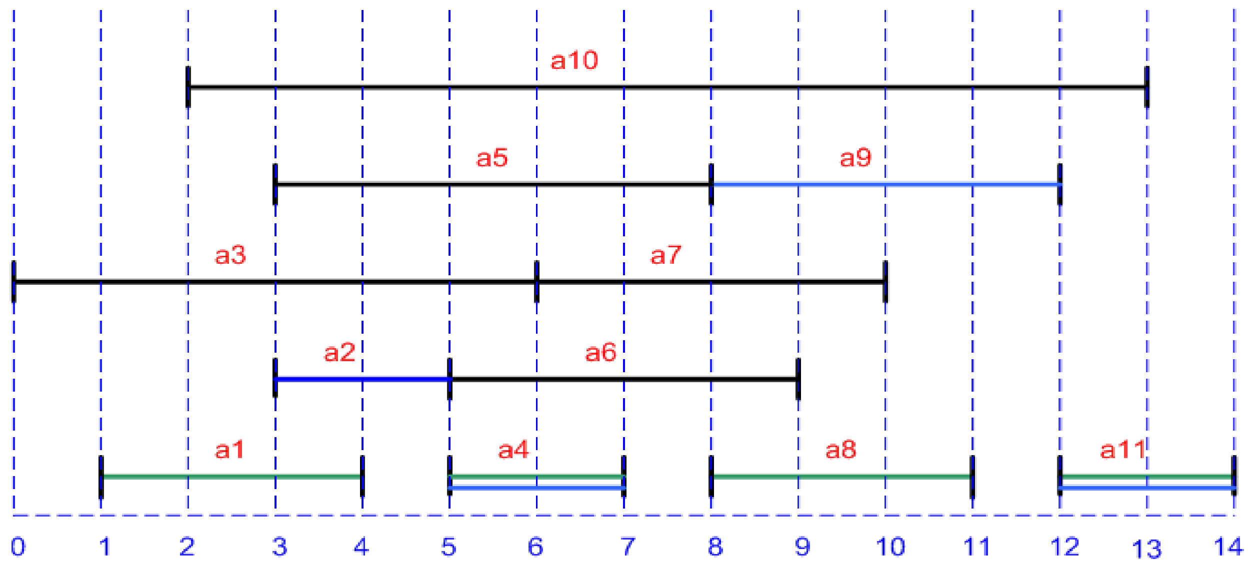
Activity Selection (AS) Problem (1/3)

- The problem of scheduling several competing activities that require exclusive use of a common resource
  - Set of activities $S = \{a_1, \ldots, a_n\}$ require exclusive use of a common resource
  - For example, scheduling the use of a classroom
- a_i needs resource during period $[s_i, f_i)$
  - […) is a half-open interval
  - s_i : *start time* and f_i: *finish time*
- ***Goal***
  - Select the largest possible set of *non-overlapping* (***mutually compatible***) activities
- ***Note:*** Could have many other objectives
  - Schedule room for longest time
  - Maximize income rental fees

Activity Selection (AS) Problem (2/3)

- Here is a set of start and finish times
  - What is the maximum number of activities that can be completed?

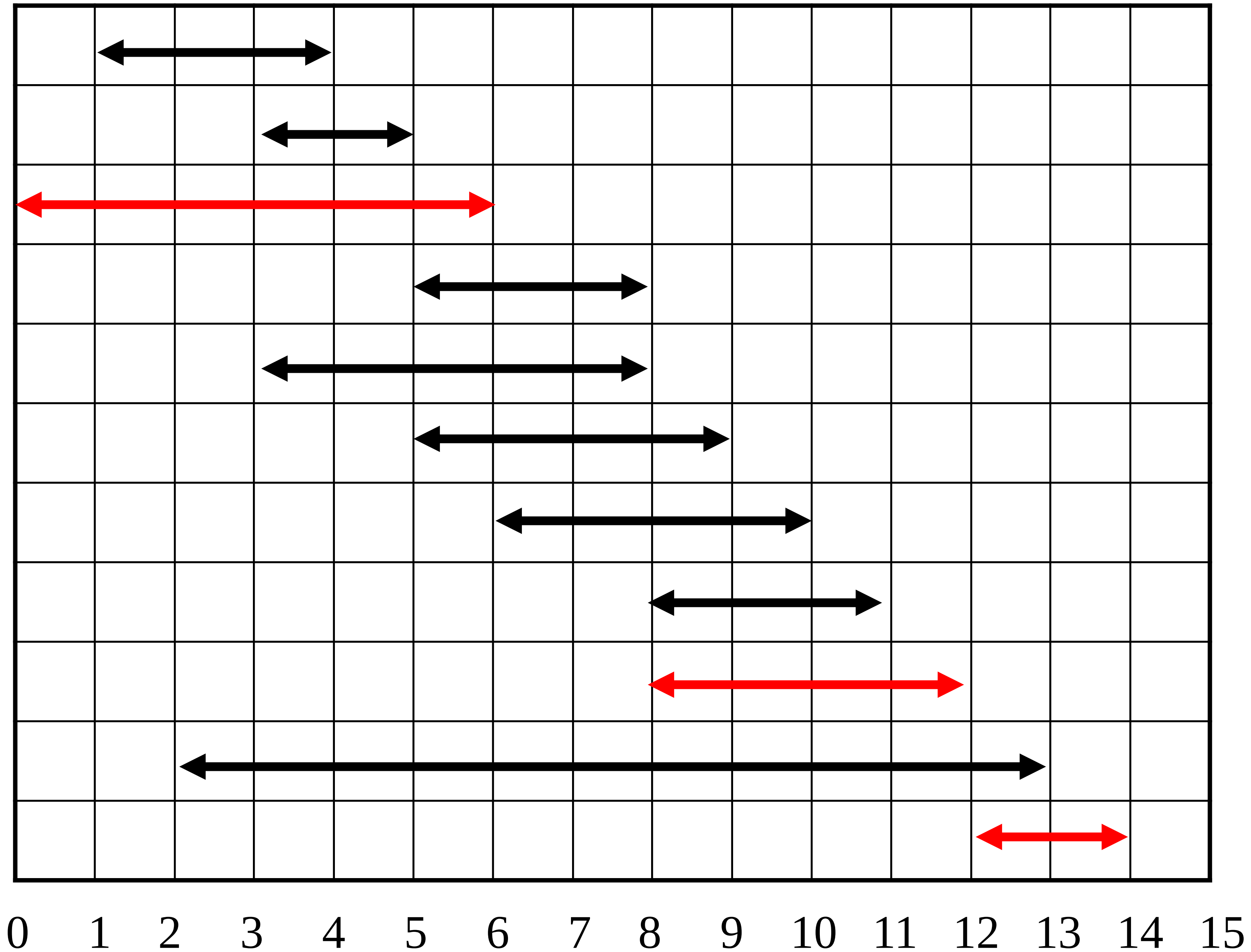
i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

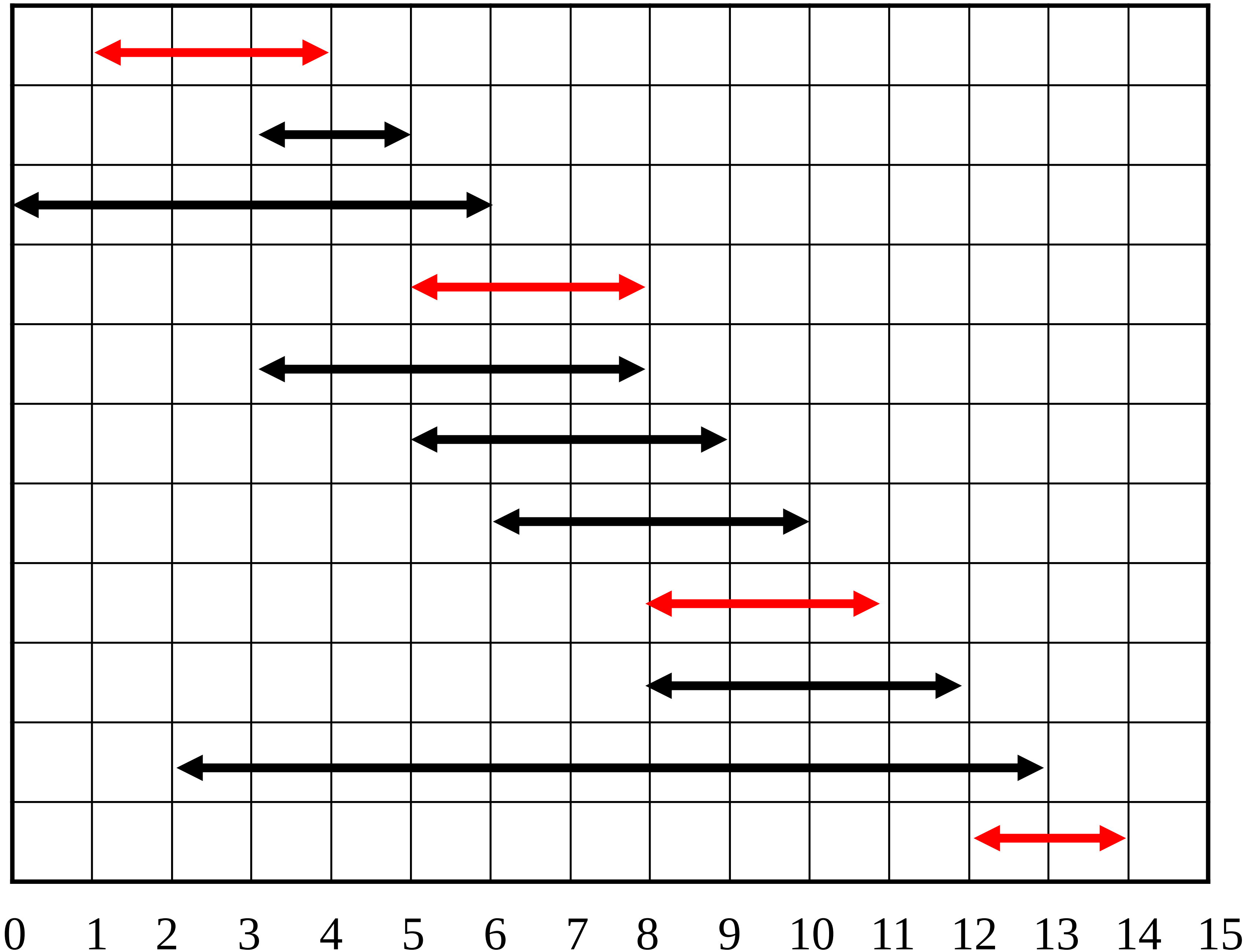


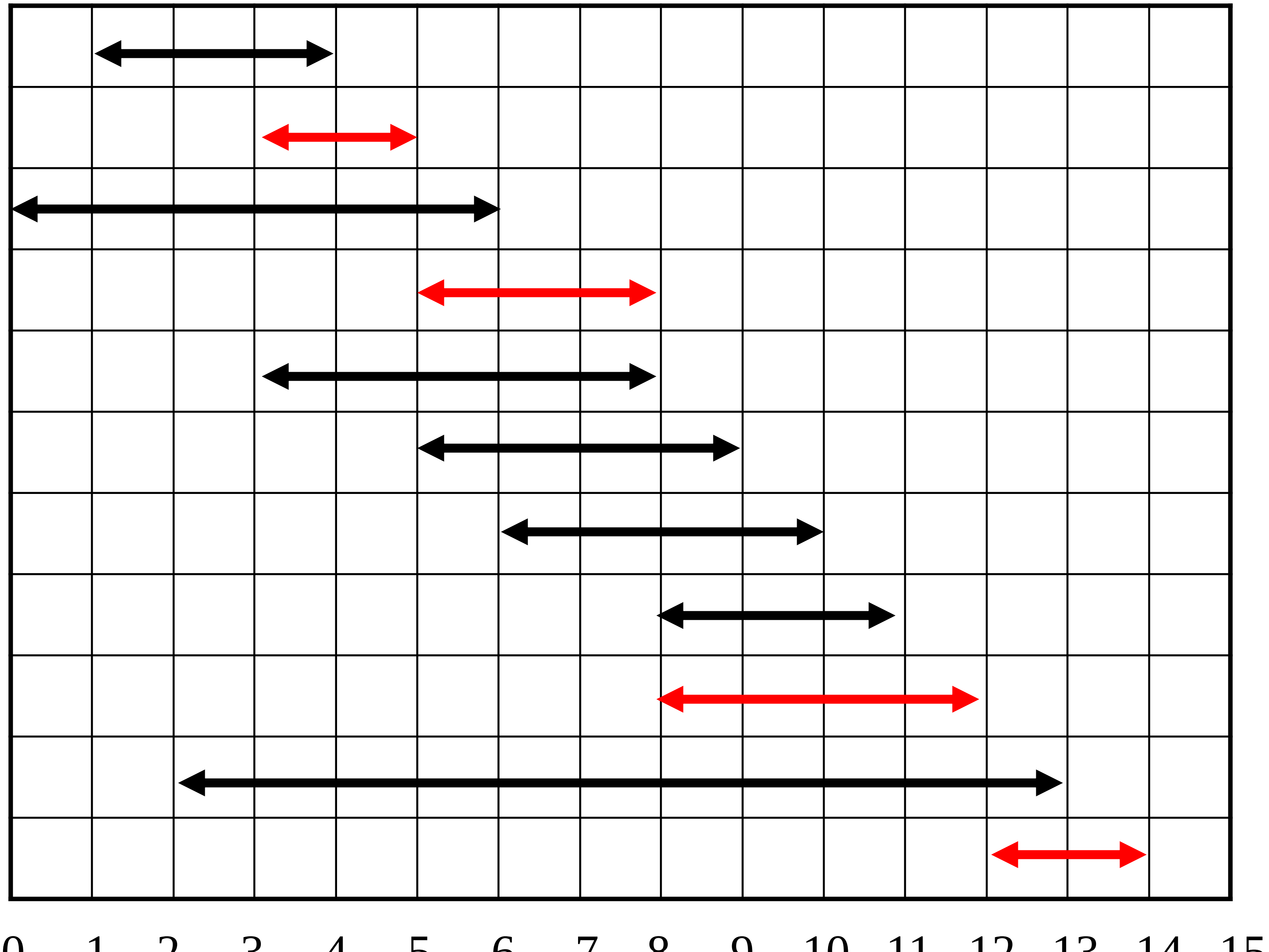
Activity Selection (AS) Problem (3/3)

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	8	9	10	11	12	13	14

- What is the maximum number of activities that can be completed?
  - $\{a_3, a_9, a_{11}\}$ can be completed
  - But so can $\{a_1, a_4, a_8, a_{11}\}$ which is a larger set
  - But it is not unique, consider $\{a_2, a_4, a_9, a_{11}\}$


0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

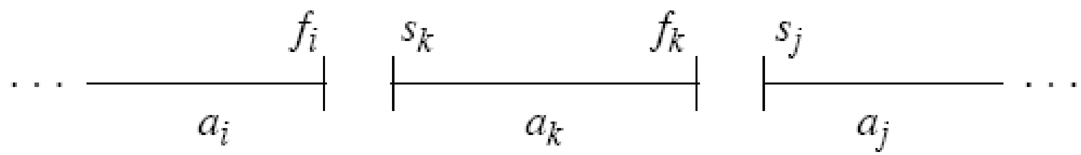

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15


0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Optimal Substructure of AS Problem (1/5)

- $S_{ij} = \{a_k \in S : f_i \le s_k < f_k \le s_j\}$

  = activities that start after a_i finishes and finish before a_j starts



- Activities in S_{ij} are compatible with
  - all activities that finish by f_i, and
  - all activities that start no earlier than s_j
- To represent the entire problem, add fictitious activities:
  - $a_0 = [-\infty, 0)$; $a_{n+1} = [\infty, \text{“}\infty + 1\text{”})$
  - We don’t care about $-\infty$ in a_0 or “$\infty + 1$” in a_{n+1}
- Then $S = S_{0,n+1}$
- Range for S_{ij} is $0 \le i, j \le n + 1$

Optimal Substructure of AS Problem (2/5)

- Assume that activities are sorted by monotonically increasing finish time
  - $f_0 \le f_1 \le f_2 \le \cdots \le f_n < f_n + 1$
- Then $i \ge j \Rightarrow S_{ij} = \varnothing$
  - If there exists $a_k \in S_{ij}$: $f_i \le S_k < f_k \le S_j < f_j \Rightarrow f_i < f_j$
  - But $i \ge j \Rightarrow f_i \ge f_j$ ➔ Contradiction
- So only need to worry about S_{ij} with $0 \le i < j \le n+1$
- All other S_{ij} are $\varnothing$

Optimal Substructure of AS Problem (3/5)

- Suppose that a solution to S_{ij} includes a_k.
  Have 2 subproblems:
  - S_{ik} (start after a_i finishes, finish before a_k starts)
  - S_{kj} (start after a_k finishes, finish before a_j starts)
- Solution to S_{ij}
  - { solution to S_{ik} } $\cup$ $\{a_k\}$ $\cup$ { solution to S_{kj} }
- Since a_k is in neither subproblem
  - the subproblems are disjoint
  - |solution to S| = |solution to S_{ik}| + 1 + |solution to S_{kj}|

Optimal Substructure of AS Problem (4/5)

- If an optimal solution to S_{ij} includes a_k, then
  the solutions to S_{ik} and S_{kj} used within this solution must be optimal as well

- Let A_{ij} = optimal solution to S_{ij}

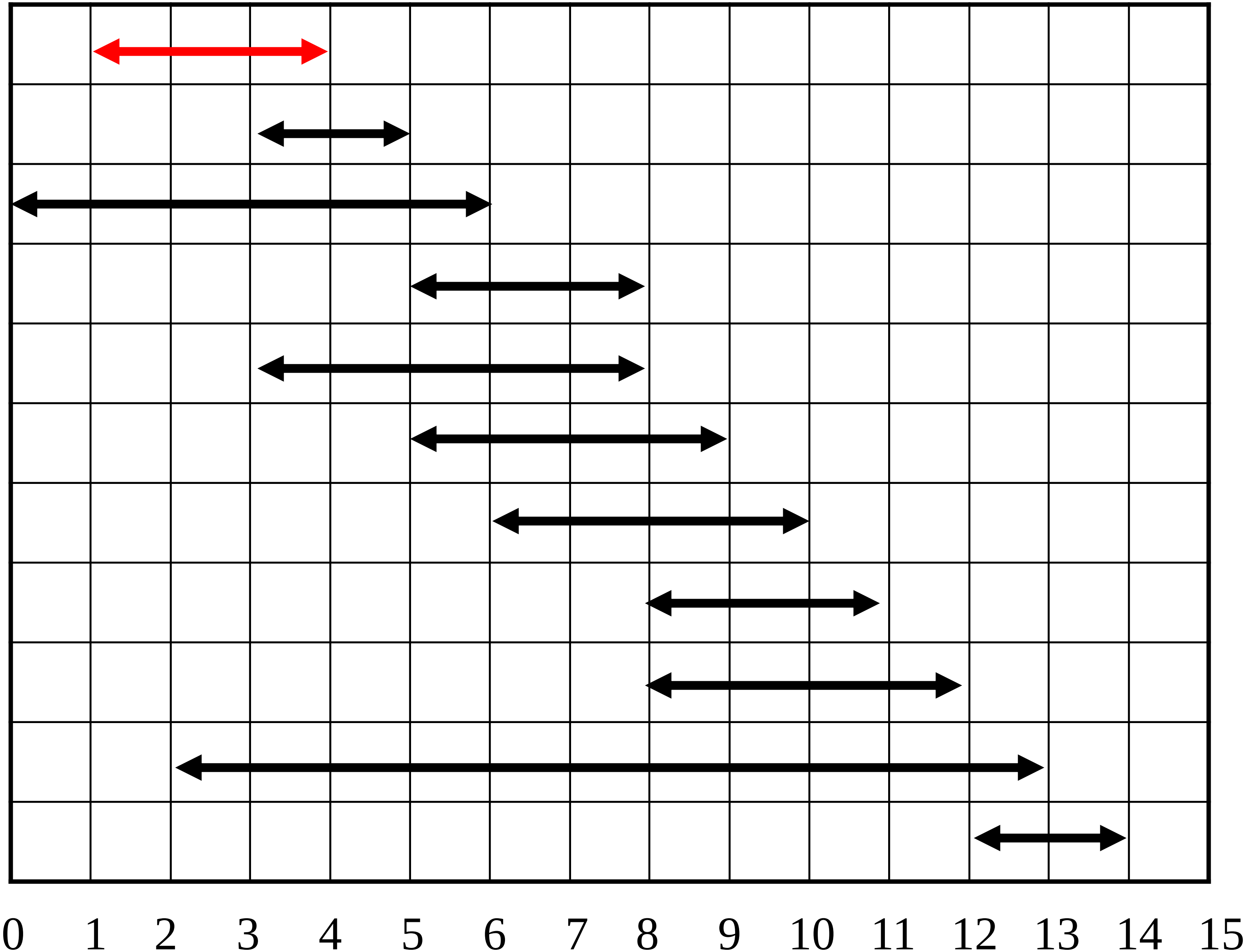
- So $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$, assuming:
  - S_{ij} is non-empty
  - we know a_k

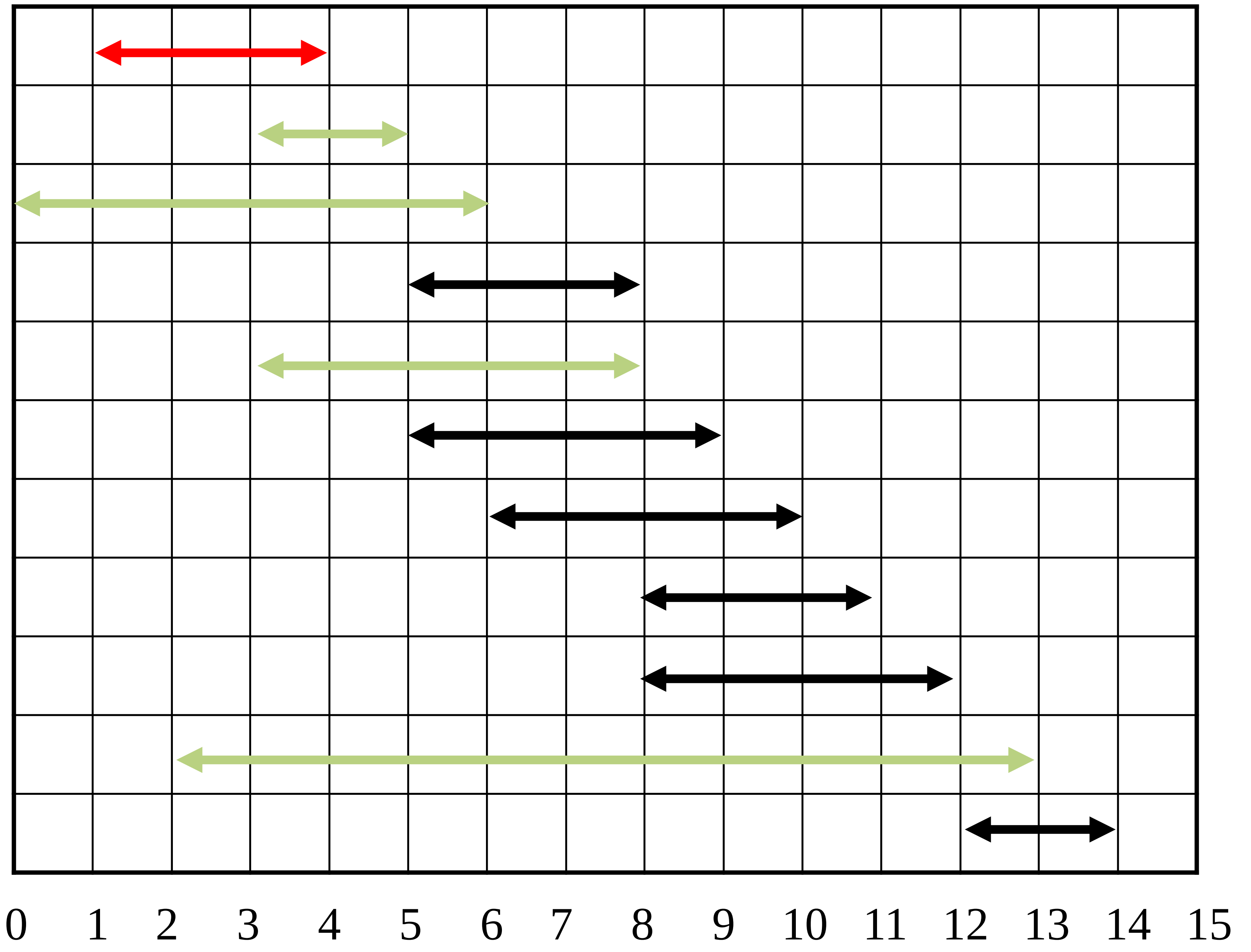
Optimal Substructure of AS Problem (5/5)

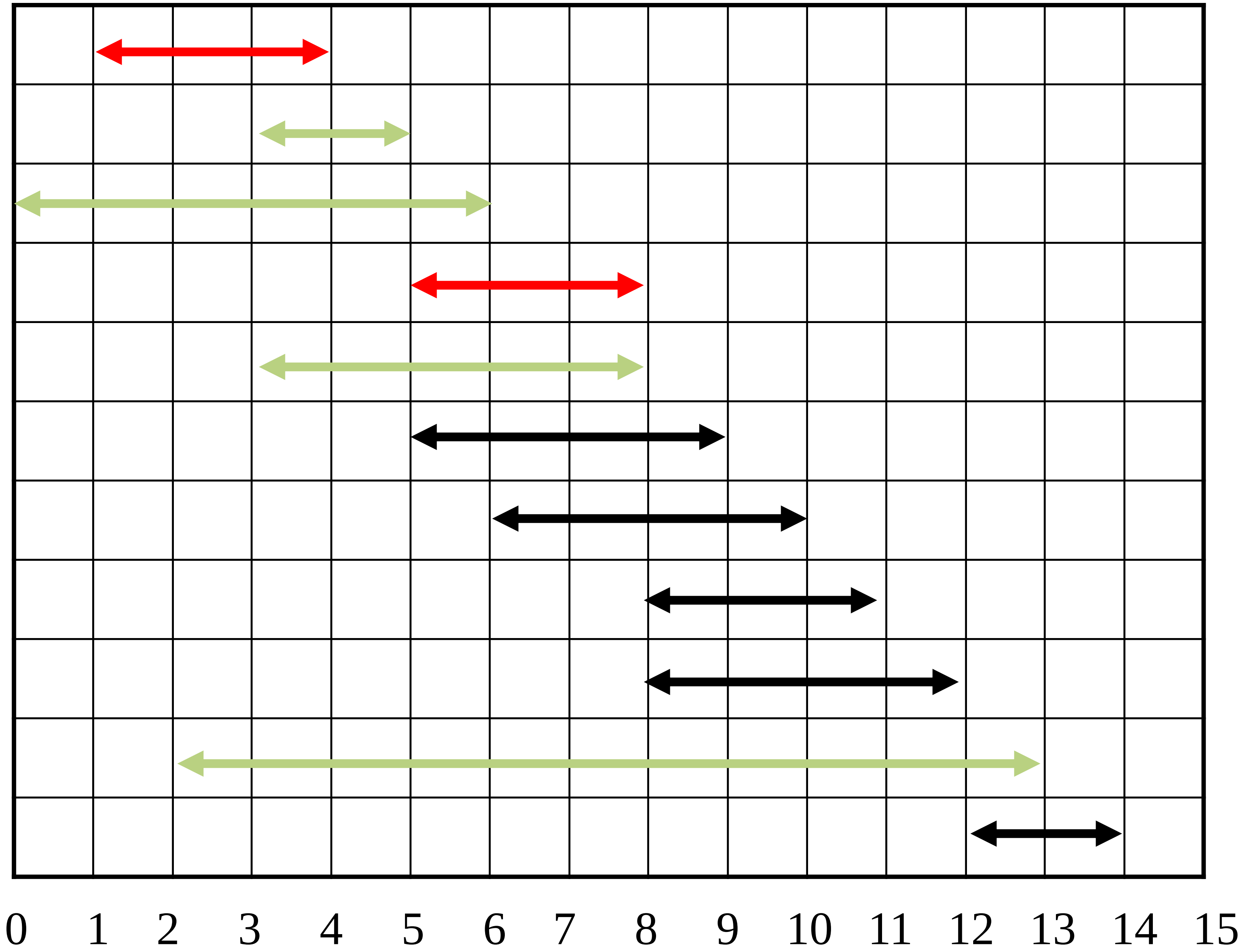
- $c[i,j]$: size of maximum-size subset of mutually compatible activities in S_{ij}
  - $i \geq j \;\Rightarrow\; S_{ij} = \emptyset \;\Rightarrow\; c[i,j] = 0$
- If $S_{ij} \neq \emptyset$, suppose we know that *a k* is in the subset
  - $c\,[i,j] = c\,[i,k] + 1 + c\,[k,j]$
- But, we don't know which *k* to use, and so
  - $$c[i,j] = \begin{cases} 0 & \text{if } S_{ij} = \emptyset\,, \\ \max\limits_{\substack{i<k<j \\ a_k \in S_{ij}}} \{c[i,k] + c[k,j] + 1\} & \text{if } S_{ij} \neq \emptyset\,. \end{cases}$$

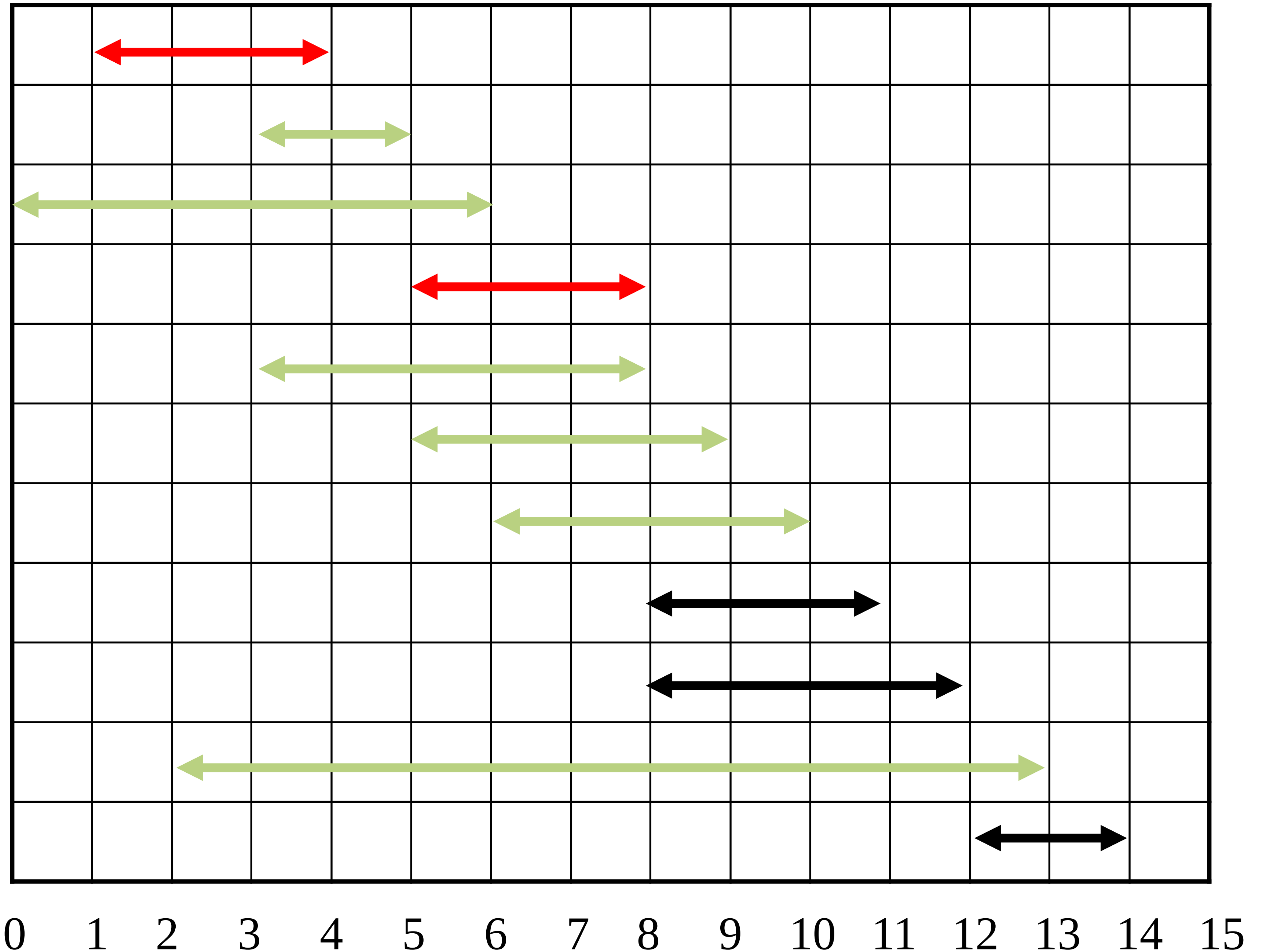
Early Finish Greedy

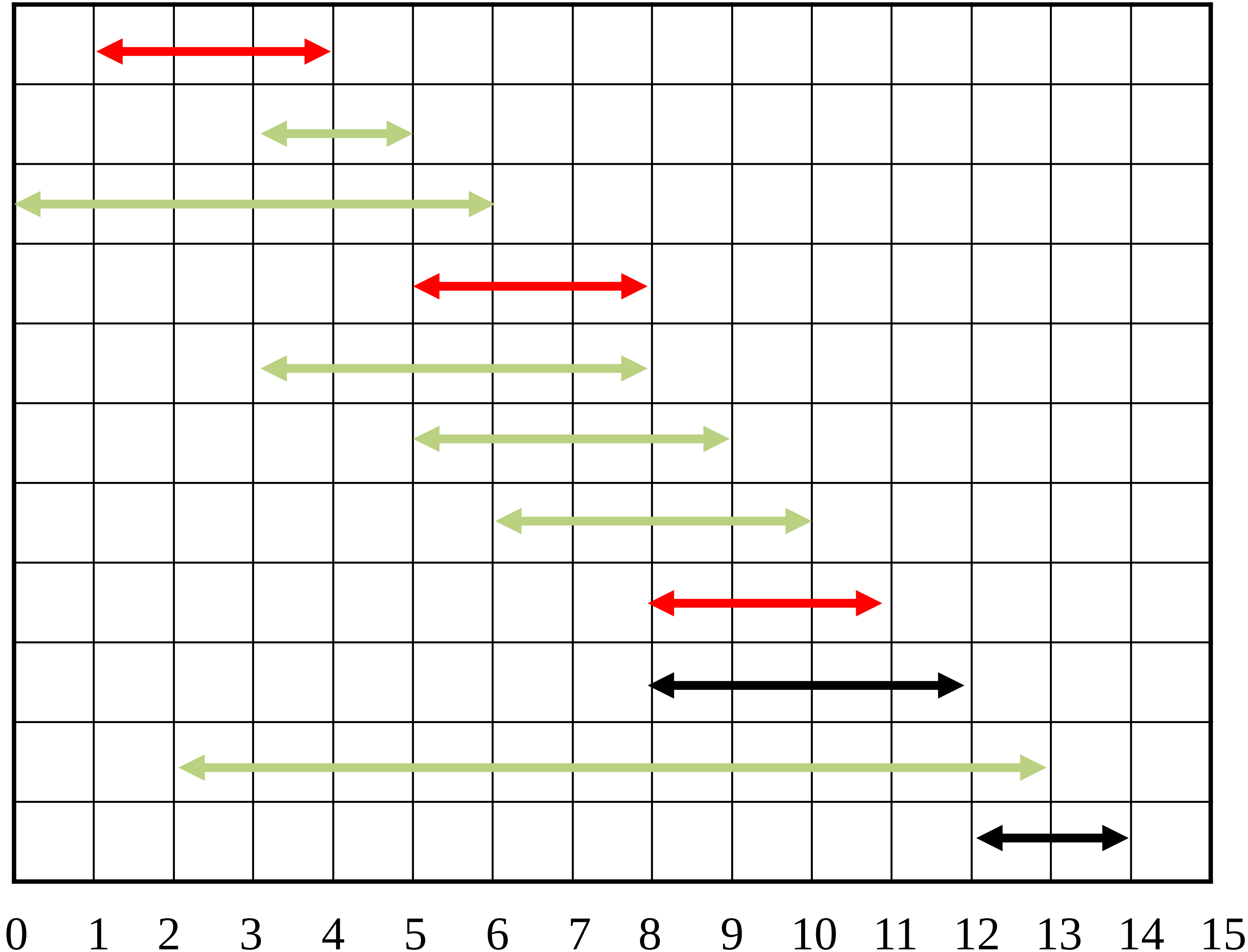
- Select the activity with the earliest finish
- Eliminate the activities that could not be scheduled
- Repeat!

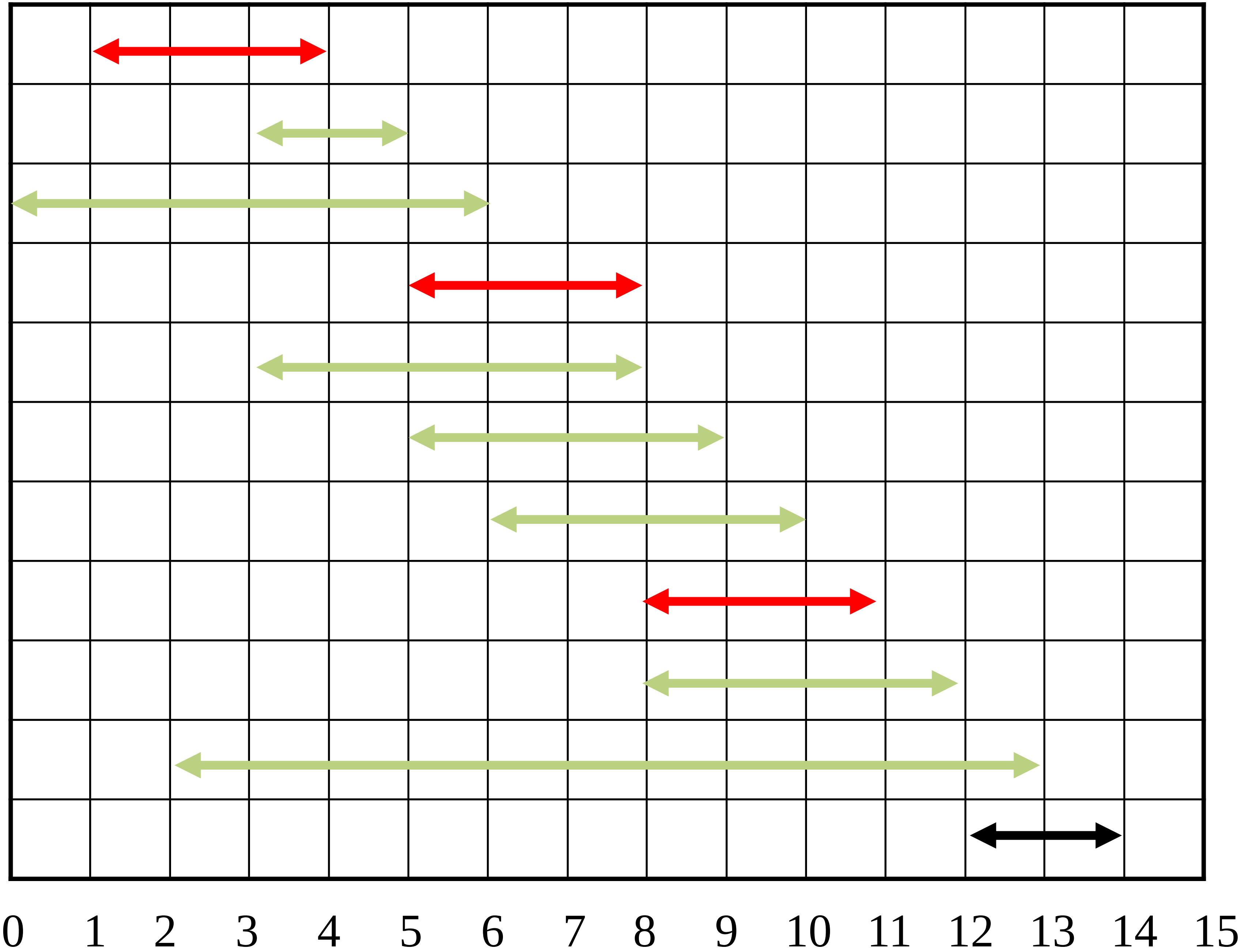

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

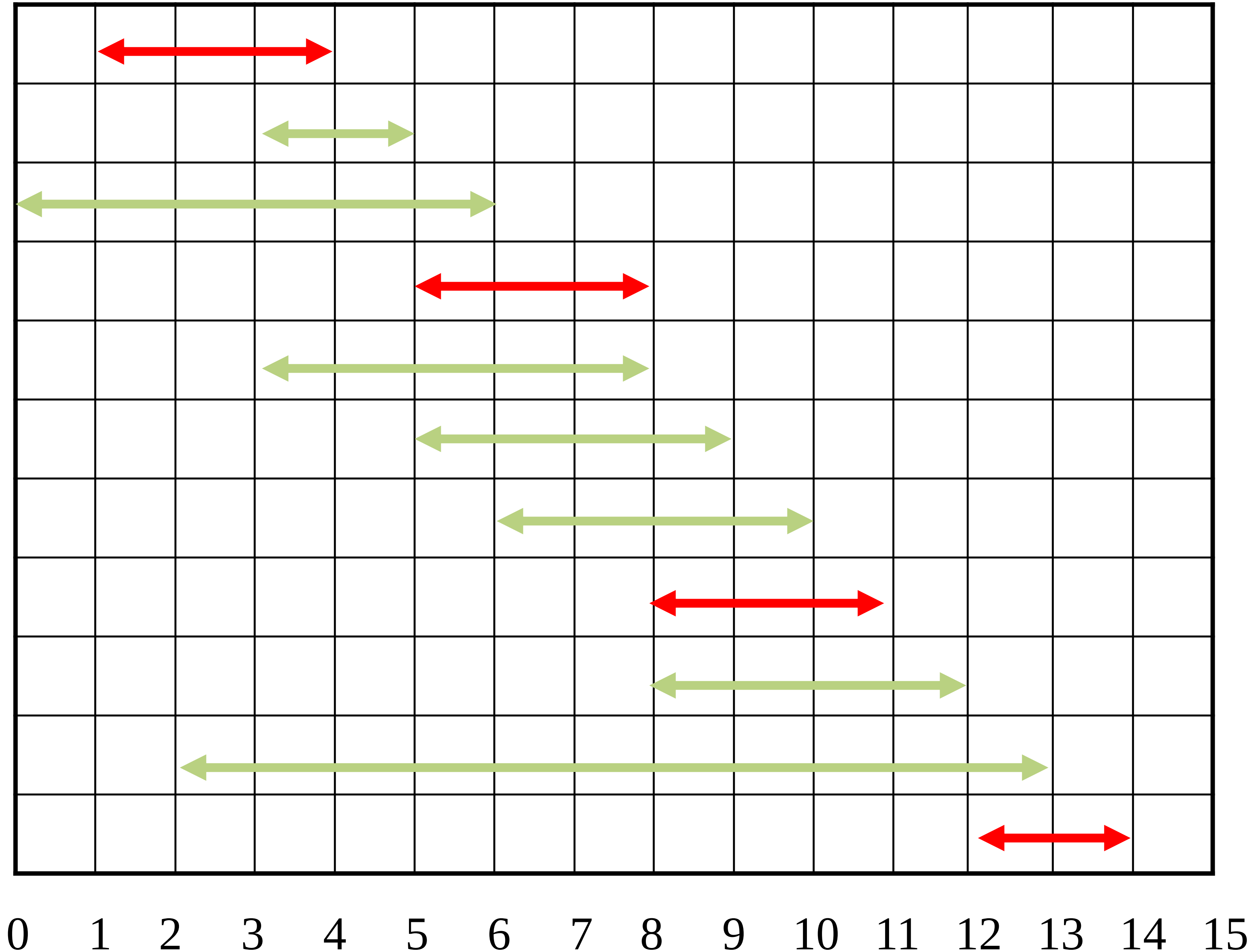

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15


0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15


0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15


0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15


0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15


0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

A Recursive Greedy Algorithm

- Assumes activities already sorted by monotonically increasing finish time
  - If not, then sort in $O(n \lg n)$ time
  - Return an optimal solution for $S_{i,n+1}$
  - REC-ACTIVITY-SELECTOR(s, f, i, n)

$m \leftarrow i + 1$
while $m \leq n$ and $s_m < f_i$ $\qquad \triangleright$ Find first activity in $S_{i,n+1}$
  do $m \leftarrow m + 1$
if $m \leq n$
  then return $\{a_m\} \cup$ REC-ACTIVITY-SELECTOR(s, f, m, n)
  else return $\emptyset$

- *Initial call: REC-ACTIVITY-SELECTOR(s, f, 0, n)*
- Time: $\Theta(n)$ - each activity examined exactly once

An Iterative Greedy Algorithm

GREEDY-ACTIVITY-SELECTOR(s, f)

$n \leftarrow length[s]$
$A \leftarrow \{a_1\}$
$i \leftarrow 1$
for $m \leftarrow 2$ **to** n
    do if $s_m \geq f_i$
        then $A \leftarrow A \cup \{a_m\}$
            $i \leftarrow m$
return A

Time $\Theta(\boldsymbol{n})$

Greedy-Choice Property

- A globally optimal solution can be arrived at by making a *locally optimal (greedy) choice*
- *Dynamic programming*
  - Make a choice at each step
  - Choice depends on knowing optimal solutions to subproblems
  - Solve subproblems first
  - Bottom-up manner
- *Greedy algorithm*
  - Make a choice at each step
  - Make the choice before solving the subproblems
  - Top-down fashion

The Knapsack Problem

- The famous *knapsack problem*:
  - A thief breaks into a museum.
    Fabulous paintings, sculptures, and jewels are everywhere.
    The thief has a good eye for the value of these objects, and knows that each will fetch hundreds or thousands of dollars on the clandestine art collector's market.
    But, the thief has only brought a single knapsack to the scene of the robbery, and can take away only what he can carry.
    What items should the thief take to maximize the haul?

Greedy Algorithm vs Dynamic Programming

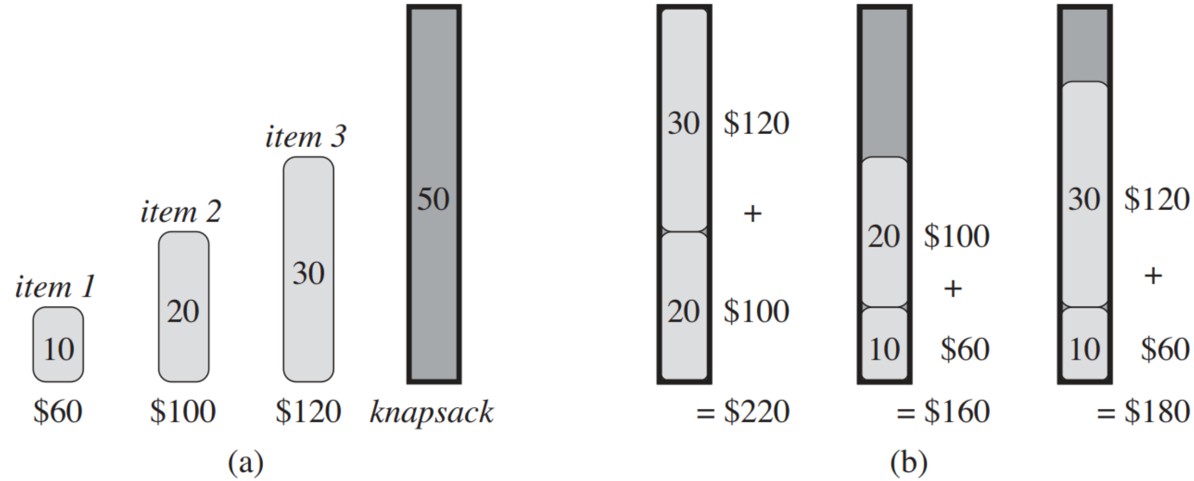
- The knapsack problem
  - Good example of the difference
- 0-1 knapsack problem: not solvable by greedy algorithm
  - n items
  - Item i is worth \$$v_i$, weighs w_i pounds
  - Find a most valuable subset of items with total weight $\leq W$
  - Have to either take an item or not take it
    - Can't take part of it

Greedy Algorithm vs Dynamic Programming

- Fractional knapsack problem:
  - Solvable by greedy
  - Like the 0-1 knapsack problem, but can take fraction of an item
  - Both have optimal substructure
  - But the fractional knapsack problem has the greedy-choice property, and the 0-1 knapsack problem does not
  - To solve the fractional problem, rank items by value/weight
    - v_i/w_i
  - Let $v_i/w_i \geq v_{i+1}/w_{i+1}$ for all i

0-1 Knapsack Problem

i	1	2	3
v_i	60	100	120
w_i	10	20	30
v_i / w_i	6	5	4



- W = 50
- Greedy solution
  - Take items 1 and 2
  - value = 160, weight = 30
  - Have 20 pounds of capacity left over
- Optimal solution
  - Take items 2 and 3
  - value = 220, weight = 50
  - No leftover capacity

0-1 Knapsack Problem

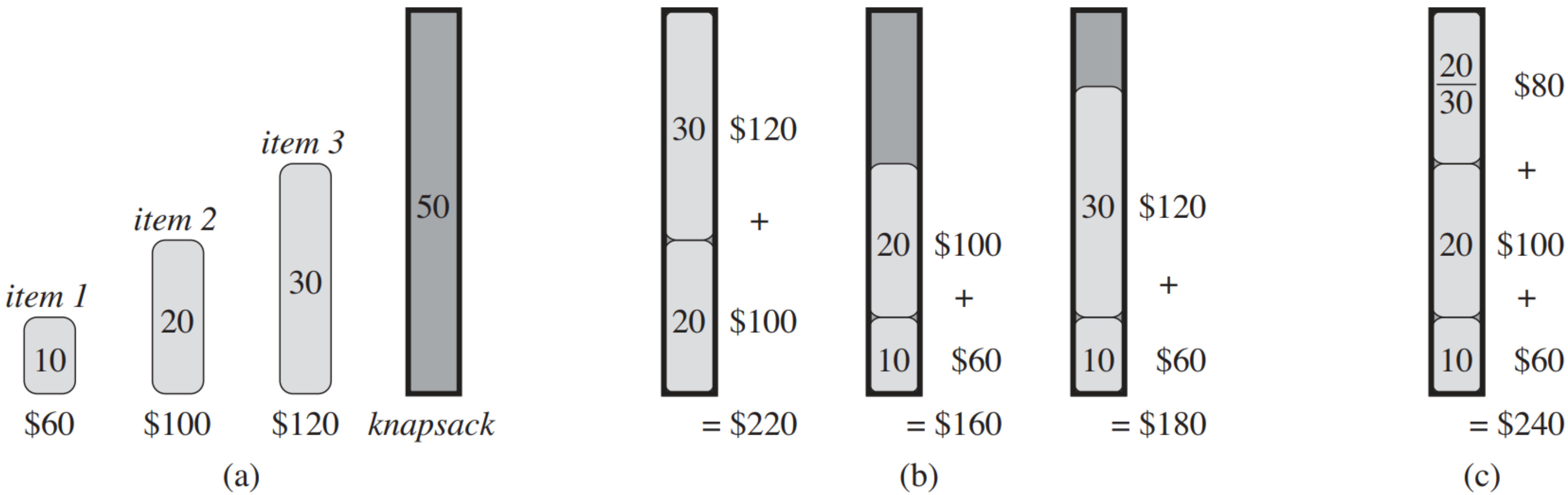


Figure 16.2 An example showing that the greedy strategy does not work for the 0-1 knapsack problem. **(a)** The thief must select a subset of the three items shown whose weight must not exceed 50 pounds. **(b)** The optimal subset includes items 2 and 3. Any solution with item 1 is suboptimal, even though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

0-1 Knapsack Dynamic Programming

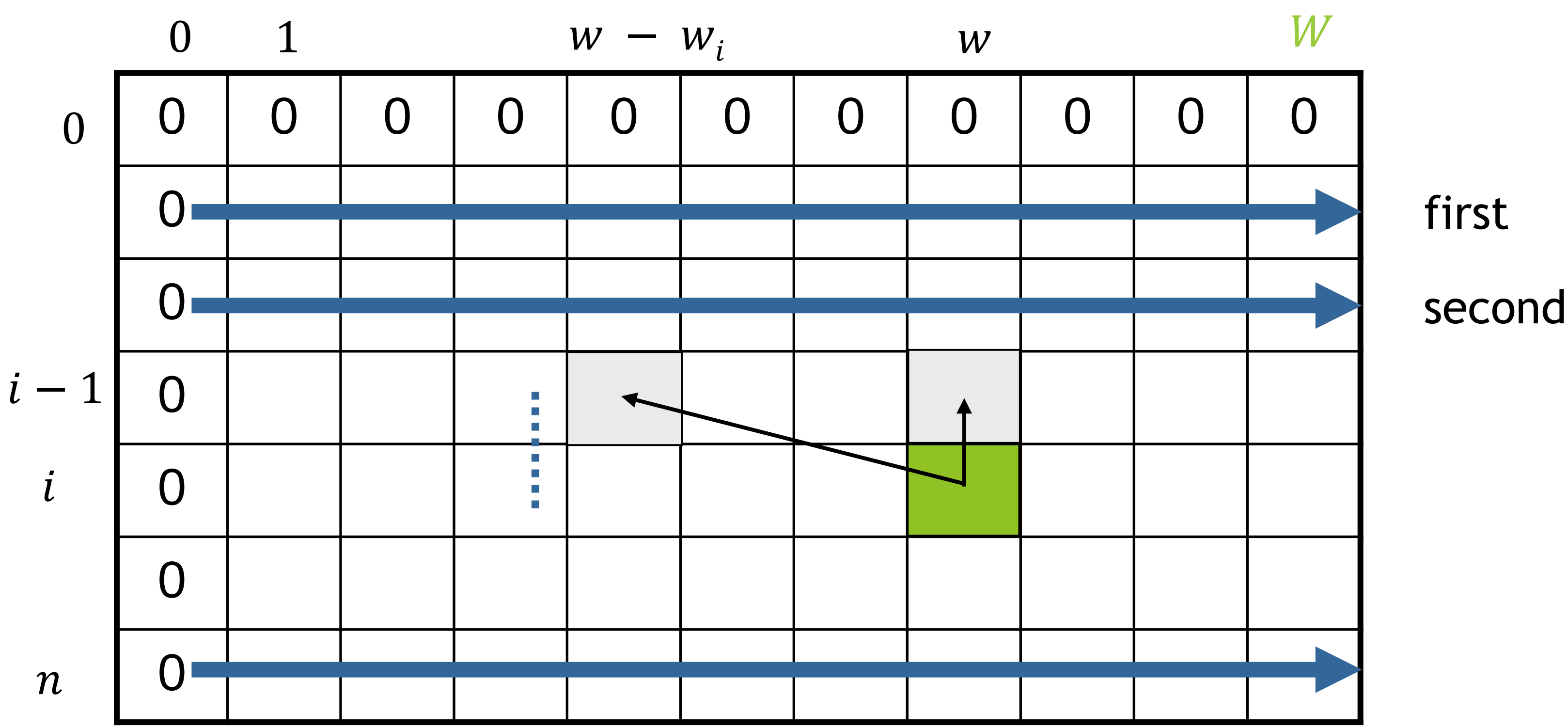
- $P(i, w)$ - the maximum profit that can be obtained from items 1 to i, if the knapsack has size w

- Case 1: thief takes item i
  - $P(i,\ w)\ = v_i + P(i-1,\ w - w_i)$

- Case 2: thief does not take item i
  - $P(i,\ w)\ = P(i-1,\ w)$

0-1 Knapsack Dynamic Programming

$$P(i,w) = \max\{\overbrace{v_i + P(i-1, w-w_i)}^{\text{Item } i \text{ was taken}}, \overbrace{P(i-1,w)}^{\text{Item i was not taken}}\}$$



Example

W = 5

Item	Weight	Value
1	2	12
2	1	10
3	3	20
4	2	15

$$P(i, w) = \max\{vi + P(i-1, w-w_i), P(i-1, w)\}$$

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0	10	15	25	30	37

P(1, 1) = P(0, 1) = 0

P(1, 2) = max{12+0, 0} = 12

P(1, 3) = max{12+0, 0} = 12

P(1, 4) = max{12+0, 0} = 12

P(1, 5) = max{12+0, 0} = 12

P(2, 1)= max{10+0, 0} = 10

P(2, 2)= max{10+0, 12} = 12

P(2, 3)= max{10+12, 12} = 22

P(2, 4)= max{10+12, 12} = 22

P(2, 5)= max{10+12, 12} = 22

P(3, 1)= P(2,1) = 10

P(3, 2)= P(2,2) = 12

P(3, 3)= max{20+0, 22}=22

P(3, 4)= max{20+10,22}=30

P(3, 5)= max{20+12,22}=32

P(4, 1)= P(3,1) = 10

P(4, 2)= max{15+0, 12} = 15

P(4, 3)= max{15+10, 22}=25

P(4, 4)= max{15+12, 30}=30

P(4, 5)= max{15+22, 32}=37

Reconstructing the Optimal Solution

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0	10	15	25	30	37

- Item 4
- Item 2
- Item 1

- Start at $P(n, W)$
- When you go left-up
  - item i has been taken
- When you go straight up
  - item i has not been taken

0-1 Knapsack Dynamic Programming

$$P[k,w] = \begin{cases} P[k-1,w] & \text{if } w_k > w \\ \max\{P[k-1,w-w_k]+v_k, P[k-1,w]\} & \text{else} \end{cases}$$

- Define $P[k,w]$ to be the best selection from S_k with weight at most w
- Since $P[k,w]$ is defined in terms of $P[k-1,*]$, we can use two arrays of instead of a matrix
- Running time: $O(n*W)$
- Not a polynomial-time algorithm since W may be large
- This is a pseudo-polynomial time algorithm

Algorithm *0-1 Knapsack*(*S*, *W*):

Input: set S of n items with benefit v_i and weight w_i; maximum weight W

Output: benefit of best subset of S with weight at most W

```
let A and B be arrays of length W + 1
for w ← 0 to W do
    B[w] ← 0
for k ← 1 to n do
    copy array B into array A
    for w ← w_k to W do
        if A[w−w_k] + v_k > A[w] then
            B[w] ← A[w−w_k] + v_k
return B[W]
```

Self-Study

- Section 16.3 Huffman codes

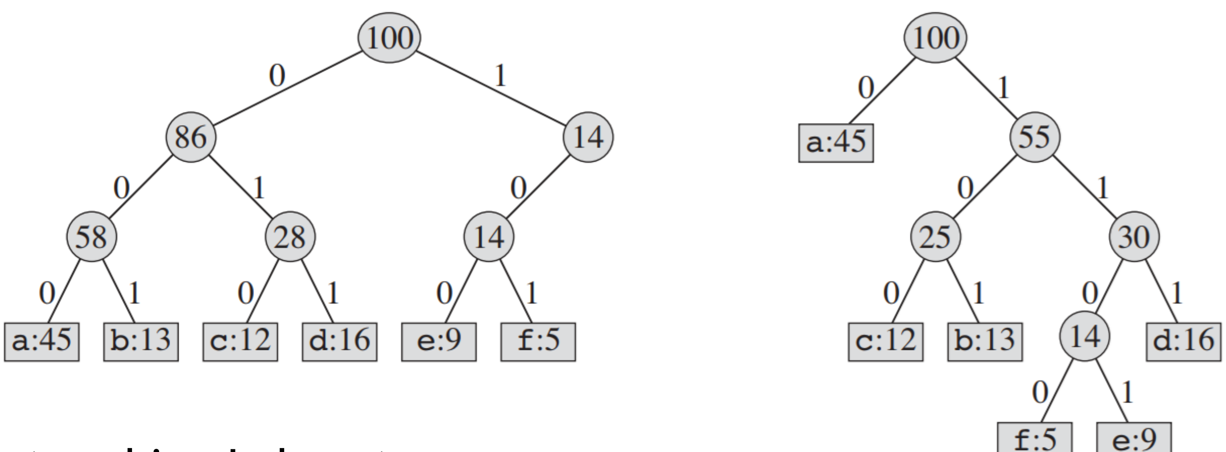
Huffman Codes

- Huffman codes compress data very effectively
  - savings of 20% to 90% are typical
- Data compression by assigning binary codes to characters
  - Huffman's greedy algorithm uses a table giving how often each character occurs (i.e., its frequency)
  - It builds up an optimal way of representing each character as a binary string.
- Example: Storing a data file of 100,000 characters contains only six different characters: a, b, c, d, e ,f

	a	b	c	d	e	f	
Frequency (in thousands)	45	13	12	16	9	5	
Fixed-length codeword	000	001	010	011	100	101	→ 300,000 bits
Variable-length codeword	0	101	100	111	1101	1100	→ 224,000 bits

Prefix Codes (1/2)

- In a prefix code, no code is also a prefix of some other code
  - Any optimal character code has an equally optimal prefix code. Example: “abc” → 0.101.100 = 0101100
- Easy encoding and decoding
  - Use a binary tree for decoding
  - Once a string of bits matches a character code, output that character with no ambiguity (no need to look ahead)



Prefix Code (2/2)

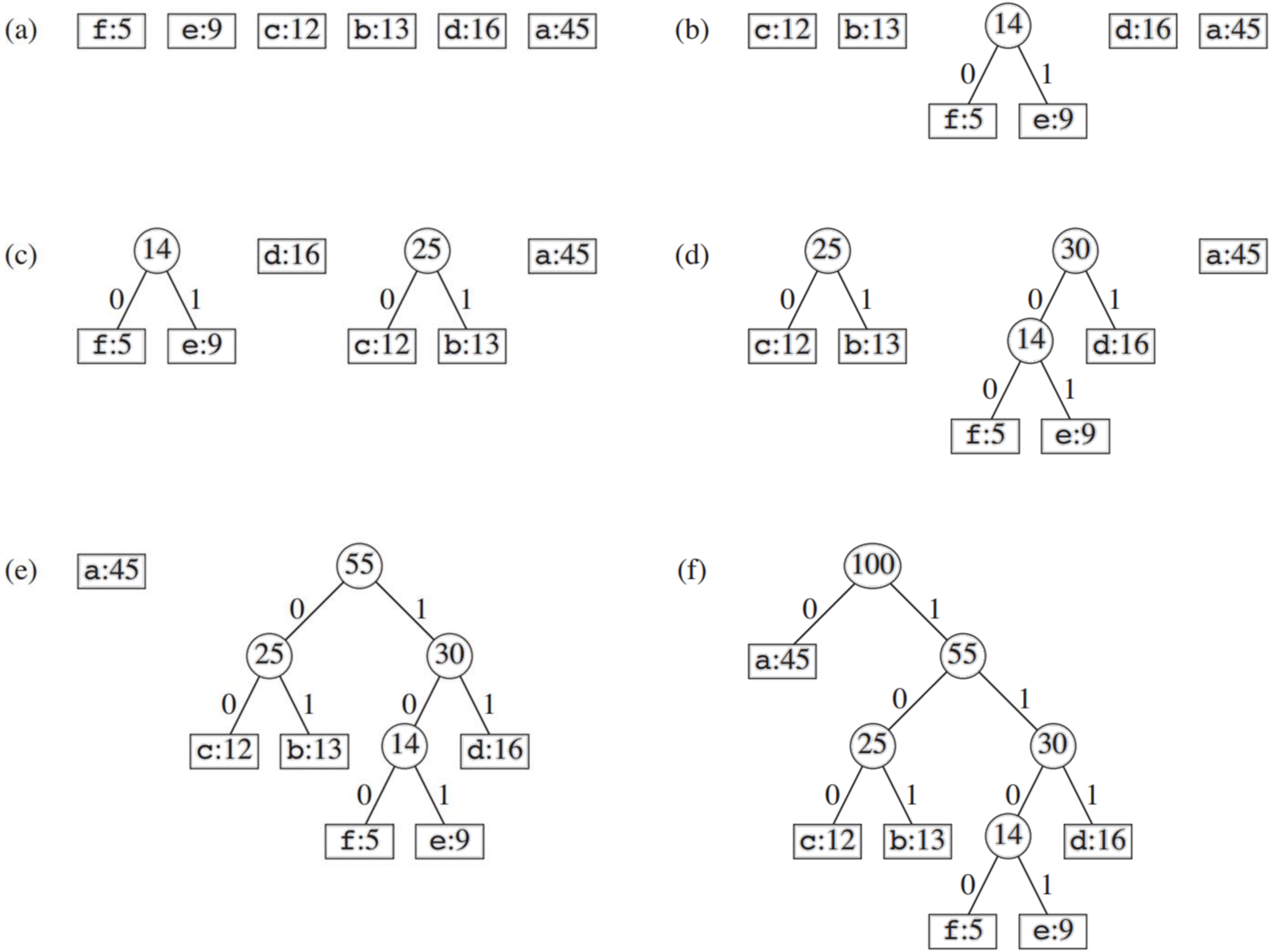
- An optimal code for a file is always represented by a *full* binary tree in which every non-leaf node has two children
- Given a tree T corresponding to a prefix code, we can easily compute the number of bits required to encode a file
  - Let C = set of unique characters in file
  - Let $f(c)$ = frequency of character c in file
  - Let $d_T(c)$ = depth of c's leaf node in T
  - → The number of bits required to encode a file (*cost* of the tree T)

$$B[T] = \sum_{c \in C} f(c)\, d_T(c)$$

Constructing a Huffman Code (1/2)

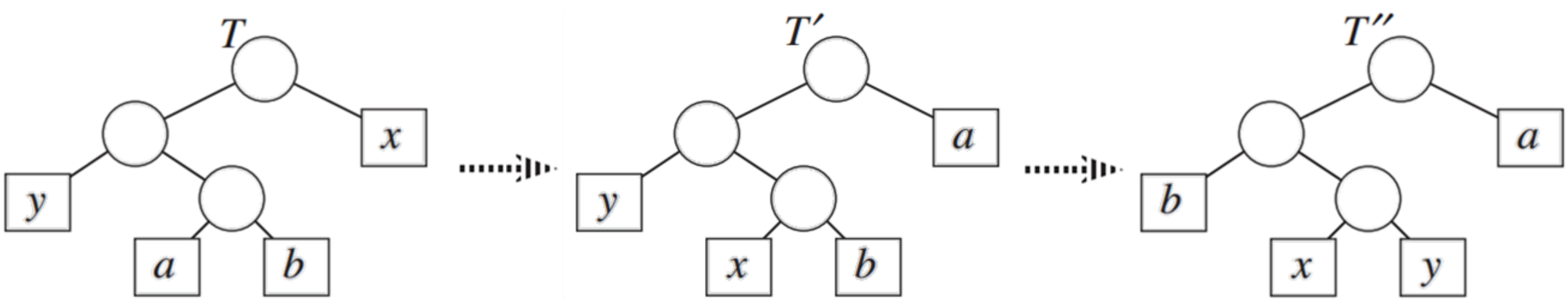
- Build a tree T by "merging" $|C|$ nodes in a bottom-up manner
- Use a min-priority queue Q to keep nodes ordered by frequency

```
Huffman(c)                          ; Analysis
    n = |c|                         ; Q is a binary heap
    Q = c                           ; O(n) BuildHeap
    for i = 1 to n-1                ; O(n)
        z = Allocate-Node()
        x = Extract-Min(Q)          ; O(lgn) O(n) times
        y = Extract-Min(Q)          ; O(lgn) O(n) times
        left(z) = x
        right(z) = y
        f(z) = f(x) + f(y)
        Insert(Q,z)                 ; O(lgn) O(n) times
    return Extract-Min(Q)           ; -----------------
                                    ; O(nlgn)
```


(a)
f:5
e:9
c:12
b:13
d:16
a:45
(b)
c:12
b:13
14
0
1
f:5
e:9
d:16
a:45
(c)
14
0
1
f:5
e:9
d:16
25
0
1
c:12
b:13
a:45
(d)
25
0
1
c:12
b:13
30
0
1
14
d:16
0
1
f:5
e:9
a:45
(e)
a:45
55
0
1
25
30
0
1
c:12
b:13
0
1
14
d:16
0
1
f:5
e:9
(f)
100
0
1
a:45
55
0
1
25
30
0
1
c:12
b:13
0
1
14
d:16
0
1
f:5
e:9

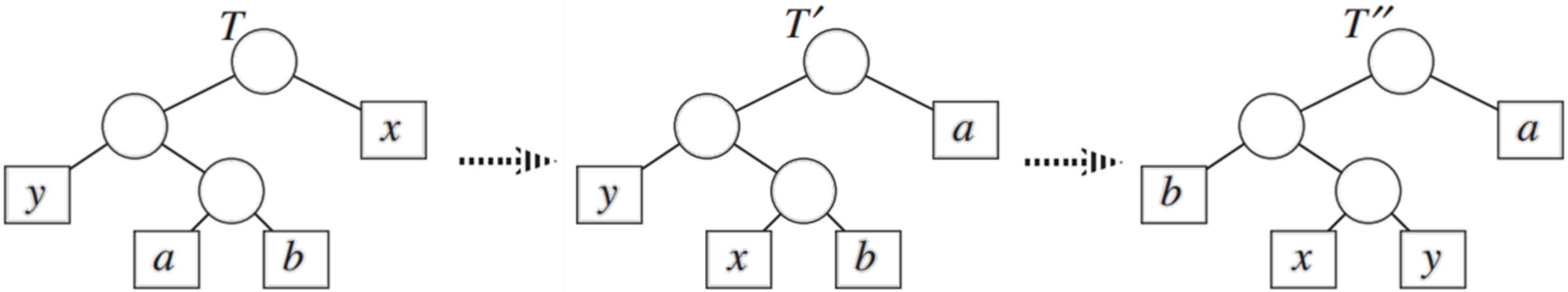
Correctness of Huffman's Algorithm (1/4)

- Huffman exhibits the greedy choice property
  - Given a tree T representing an arbitrary optimal prefix code
  - Let a and b be two characters that are sibling leaves of maximum depth in a tree T
  - Assume x and y have lowest frequencies
    - There exists an optimal code in which x and y are at the maximum depth (greedy choice)
    - Prove that moving x to the bottom (similarly, y to the bottom) yields a better (optimal) solution



Correctness of Huffman's Algorithm (2/4)

- Assume $f(x) \le f(y)$ *and* $f(a) \le f(b)$. We know $f(x) \le f(a)$ *and* $f(y) \le f(b)$



$$
\begin{aligned}
B[T] - B[T'] &= \sum_{c \in C} f(c)\, d_T(c) - \sum_{c \in C} f(c)\, d_{T'}(c) \\
&= f(x)d_T(x) + f(a)d_T(a) - f(x)d_{T'}(x) + f(a)d_{T'}(a) \\
&= f(x)d_T(x) + f(a)d_T(a) - f(x)d_T(a) + f(a)d_T(x) \\
&= (f(a) - f(x))(d_T(a) - d_T(x)) \ge 0
\end{aligned}
$$

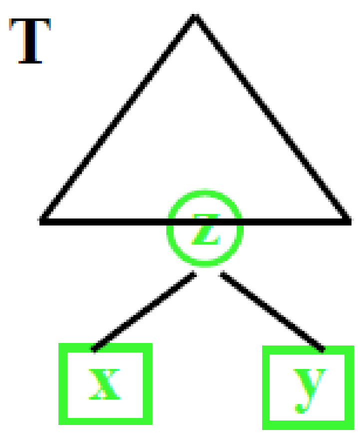
→ $\boldsymbol{B[T] \ge B[T']}$ (similarly, $\boldsymbol{B[T] \ge B[T'']}$)

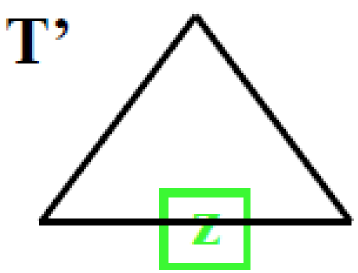
Correctness of Huffman's Algorithm (3/4)

- Huffman exhibits optimal substructure
  - Consider the optimal tree T for characters C
  - Let x and y be lowest frequency characters in C
  - Consider the optimal tree T' for $C' = C - \{x, y\} U \{z\}$, where $f(z) = f(x) + f(y)$
  - If there is a better tree for C', call it T'', then we could use T'' to build a better original tree by adding in x and y under z
  - The optimal tree T is optimal, so this is a contradiction
  - Thus T' is the optimal tree for C'

Correctness of Huffman's Algorithm (4/4)

- Huffman produces optimal prefix code
  - Immediate from claims in the previous slide





$B(T) = B(T') + f(x) + f(y)$
if $B(T'') < B(T')$
then $B(T'') + f(x) + f(y) < B(T)$
Contradiction!

Thanks to contributors

Mr. Phuoc-Nguyen Bui (2022)

Dr. Thien-Binh Dang (2017 – 2022)

Prof. Hyunseung Choo (2001 – 2022)