Greedy Algorithms 1

Jie Wang

University of Massachusetts Lowell
Department of Computer Science

Optimization

DP is a powerful technique for solving optimization problems (min and max). Greedy algorithms are a simpler and faster technique.

- A greedy strategy makes the optimal choice at the moment.
  - DP for solving optimization problems also makes the optimal choice at the moment, but from a set of subproblems.
  - Greedy algorithms do not involve recurrence subproblems.
  - There are problems where a locally optimal greedy choice will lead to a globally optimal solution.
- Greedy algorithms are often easier to code.

Example: Activity Selection

Input: A set of activities $S = \{a_1, \ldots, a_n\}$.

- Each activity is represented by a start time and a finish time:

$$a_i = (s_i, f_i).$$

- Two activities are compatible if their start-finish time intervals do not overlap

Output: A maximum-size subset of mutually compatible activities.

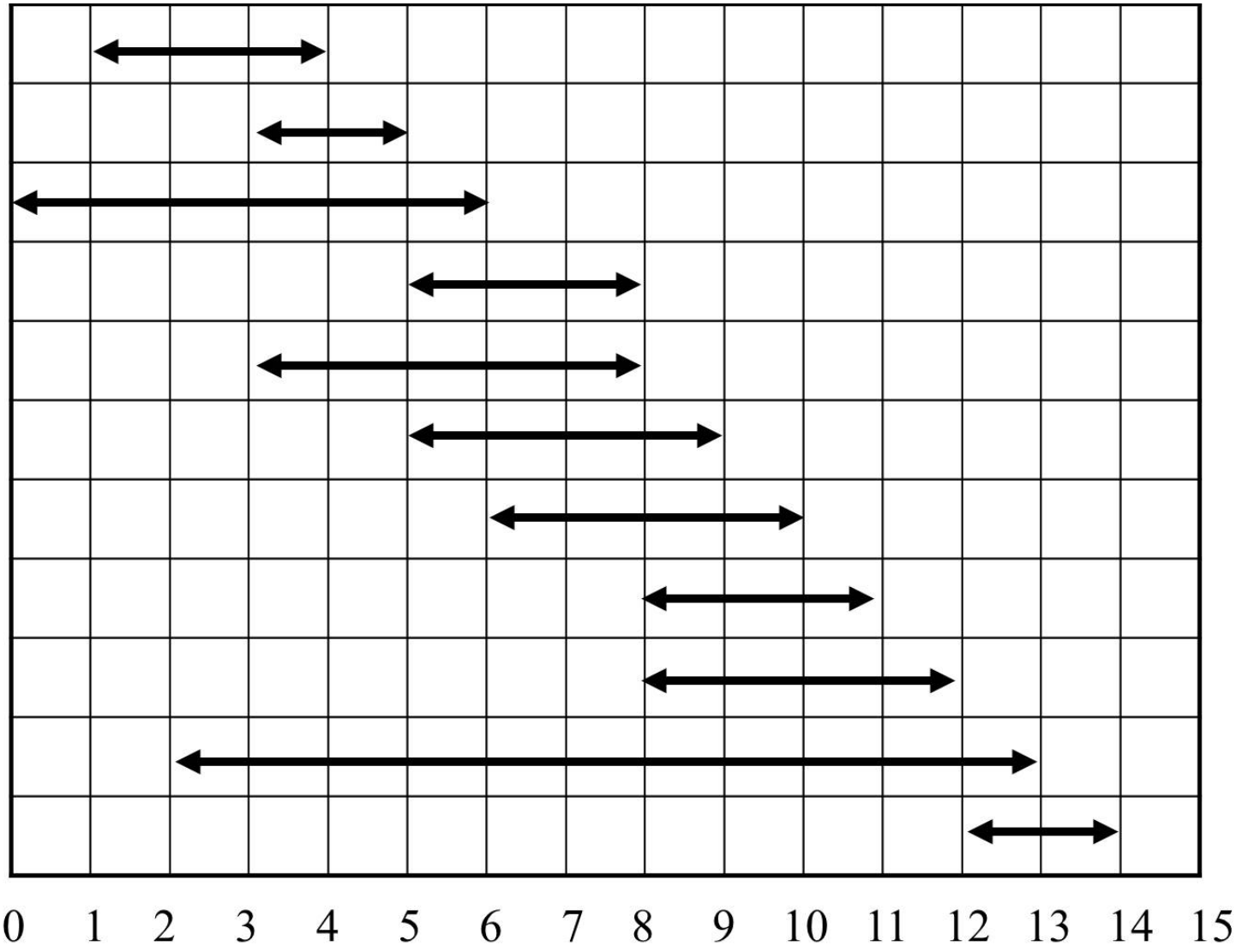
Example

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	13	14

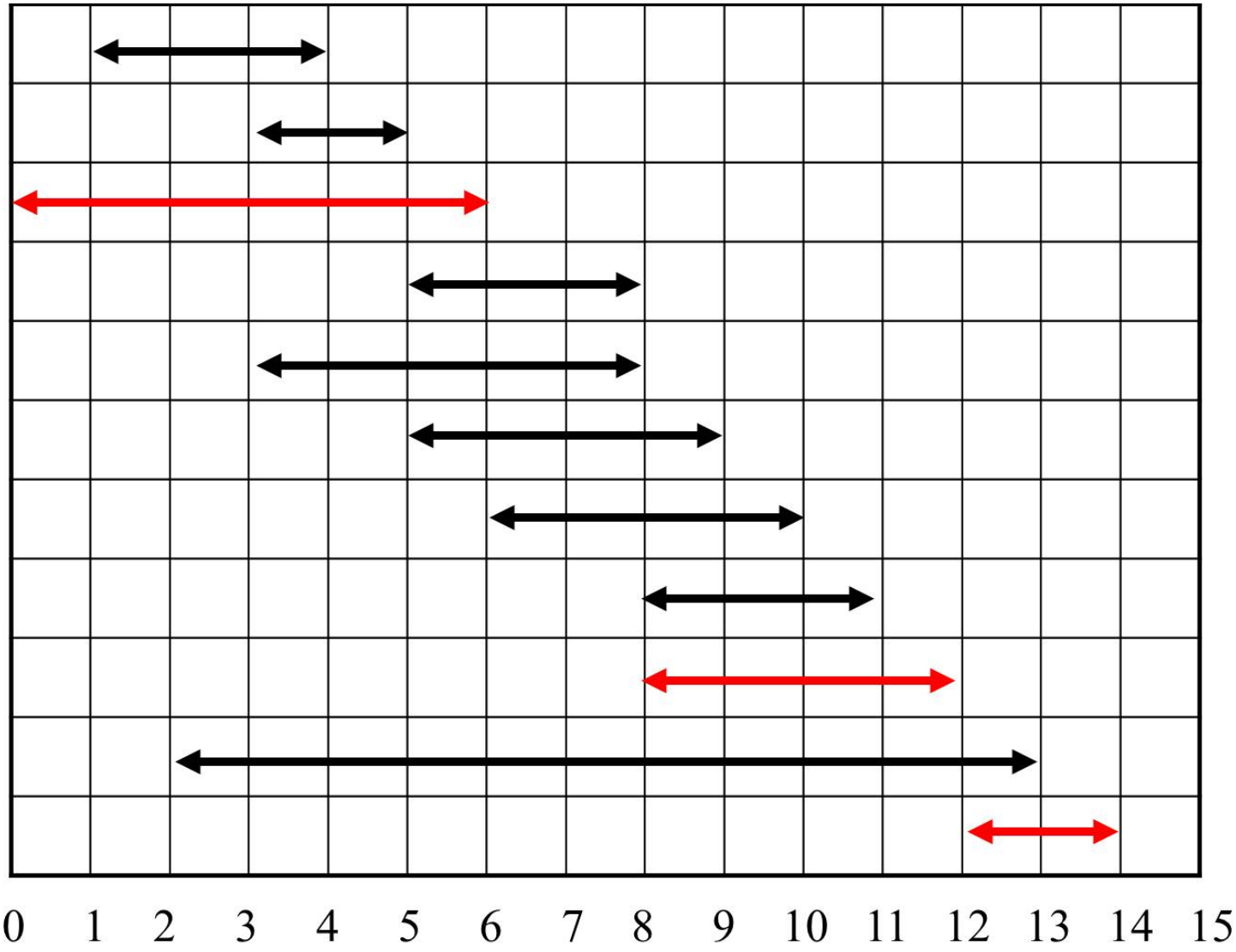
What is the maximum number of activities that can be completed?

- $\{a_3, a_9, a_{11}\}$ can be completed
- So can $\{a_1, a_4, a_8, a_{11}\}$, which is a larger set
- Optimal selection may not be unique: $\{a_2, a_4, a_9, a_{11}\}$

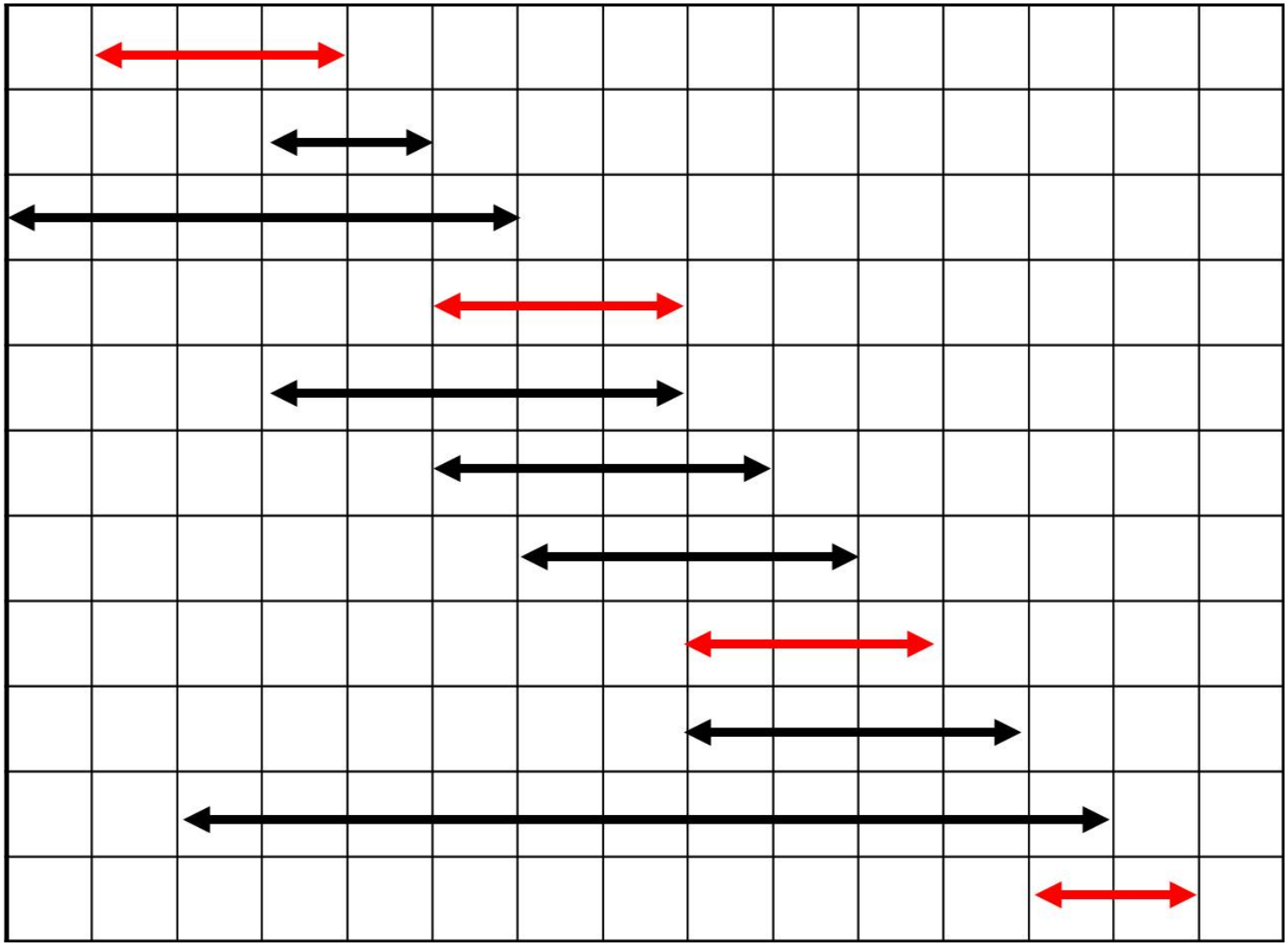
Graph Representation of the Given Activities



Graph Representation of the 1st Set of Activity Selection



Graph Representation of the 2nd Set of Activity Selection



Activity Selection Can Be Solved using DP

- Assume that the activities are sorted in monotonically increasing order of finish time:
$$f_1 \leq f_2 \leq \cdots \leq f_n.$$
- Let $S_{ij} = \{a_k \mid f_i \leq s_k \text{ and } f_k \leq s_j\}$.
  - Note: Any $a_k \in S_{ij}$ is compatible with a_i and a_j.
- **Formulation**: Let $c[i, j]$ denote the size of an optimal solution for S_{ij}. Want to compute $c[1, n]$.
- **Localization**:
$$c[i,j] = \begin{cases} 0, & \text{if } S_{ij} = \emptyset, \\ \max_{a_k \in S_{ij}} \{c[i,k] + c[k,j] + 1\}, & \text{if } S_{ij} \neq \emptyset. \end{cases}$$
- **Runtime**: $O(n^3)$.

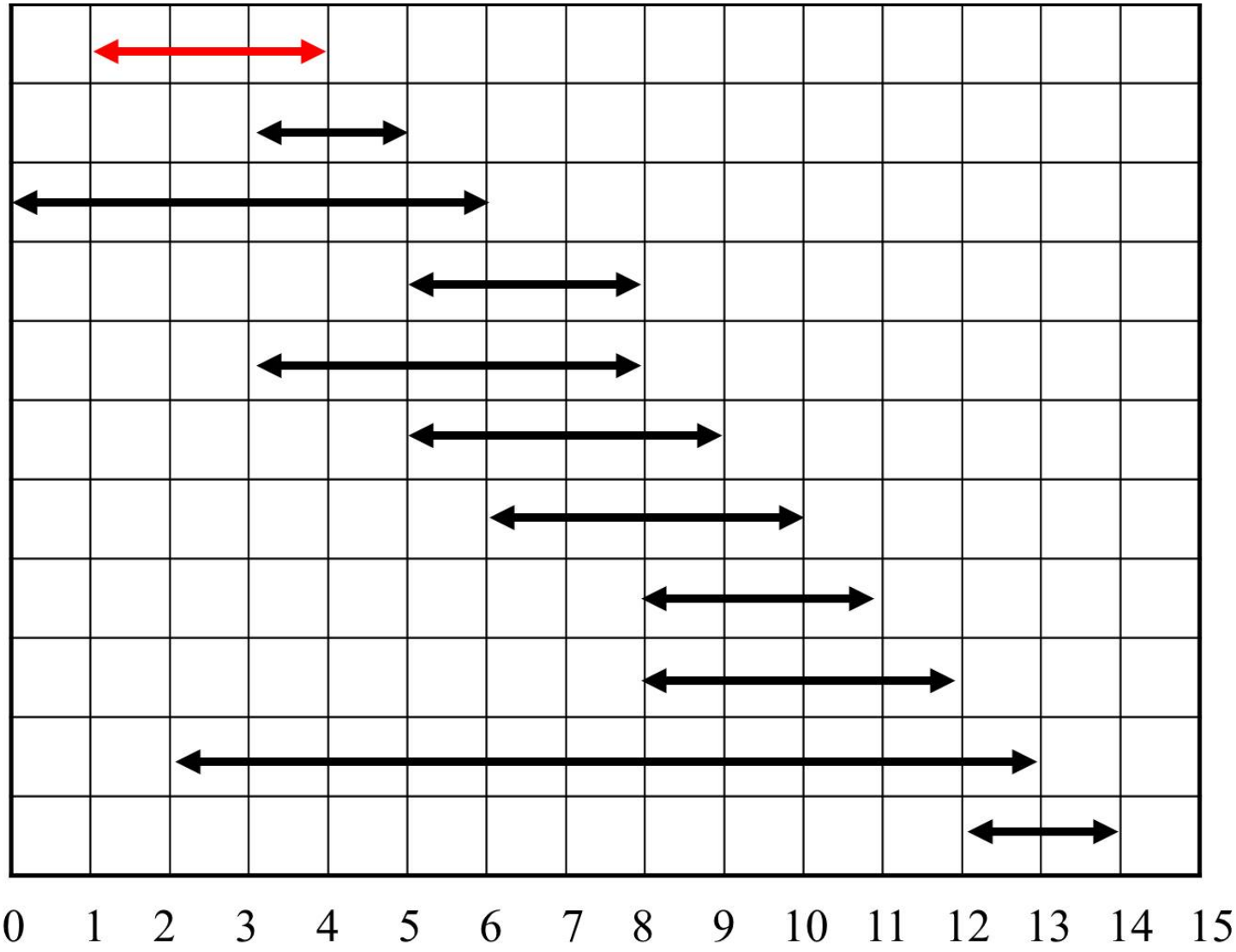
Early Finish Greedy

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

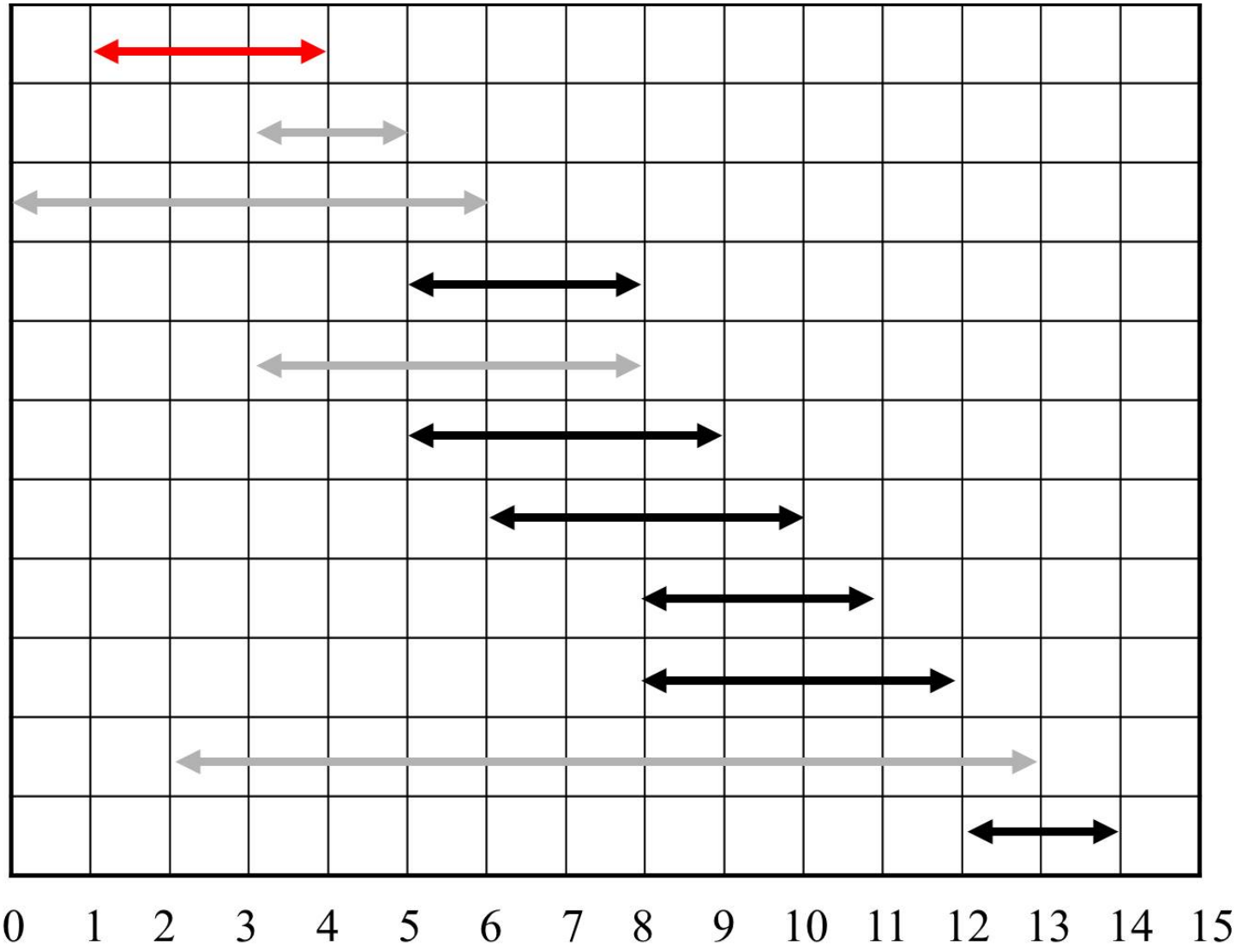
Let $S_k = \{a_i \mid s_i \geq f_k\}$.
Theorem. Let $a_m \in S_k$ with the earliest finish time. Then a_m is included in an maximum solution for S_k.
Proof.

- Proof by contradiction: Suppose that a_m is not in a given maximum solution for S_k.
- Let a_j be in the solution with the smallest finish time. Since $f_m \leq f_j$, we can replace a_j with a_m to achieve a new maximum solution.

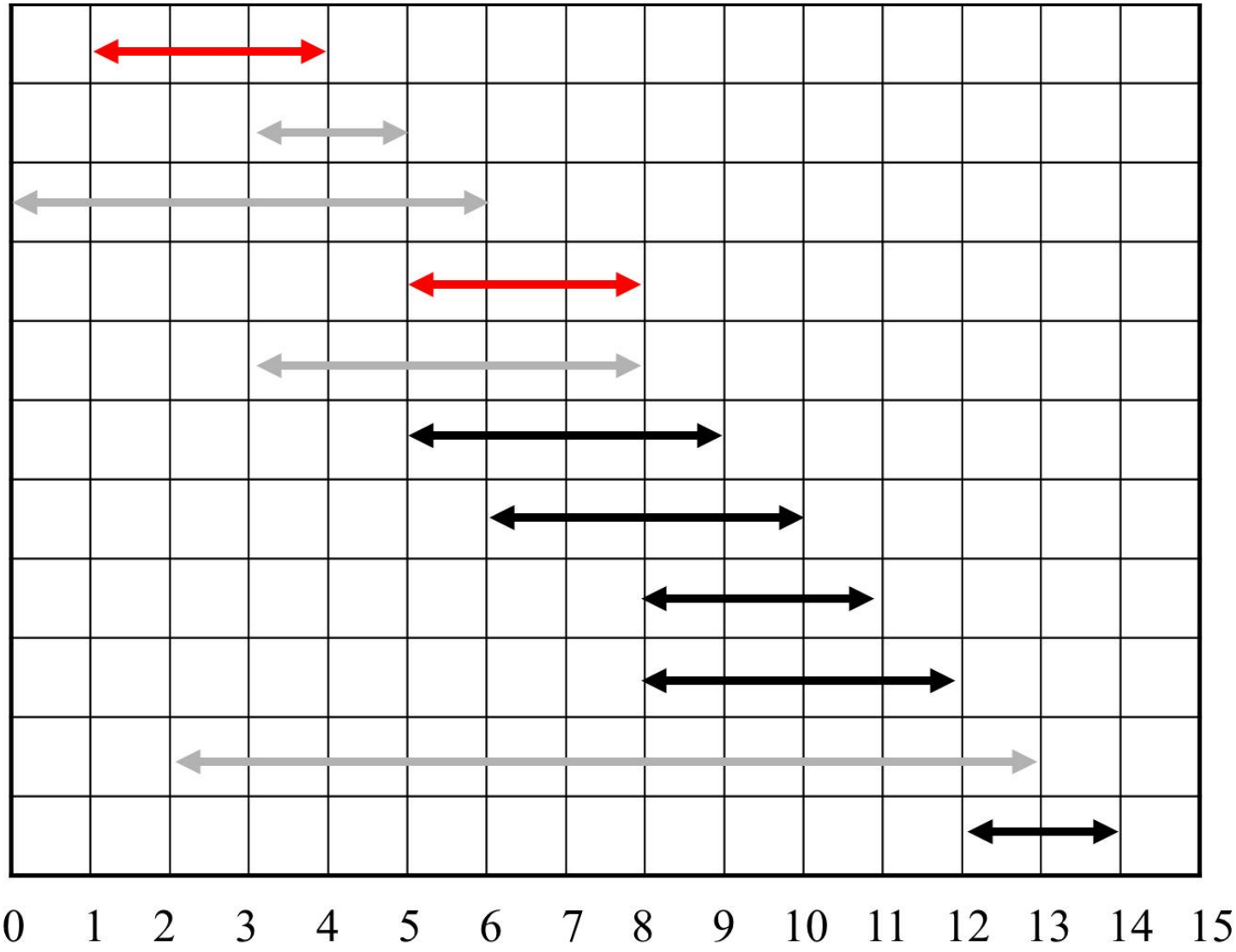
Early Finish Greedy 1



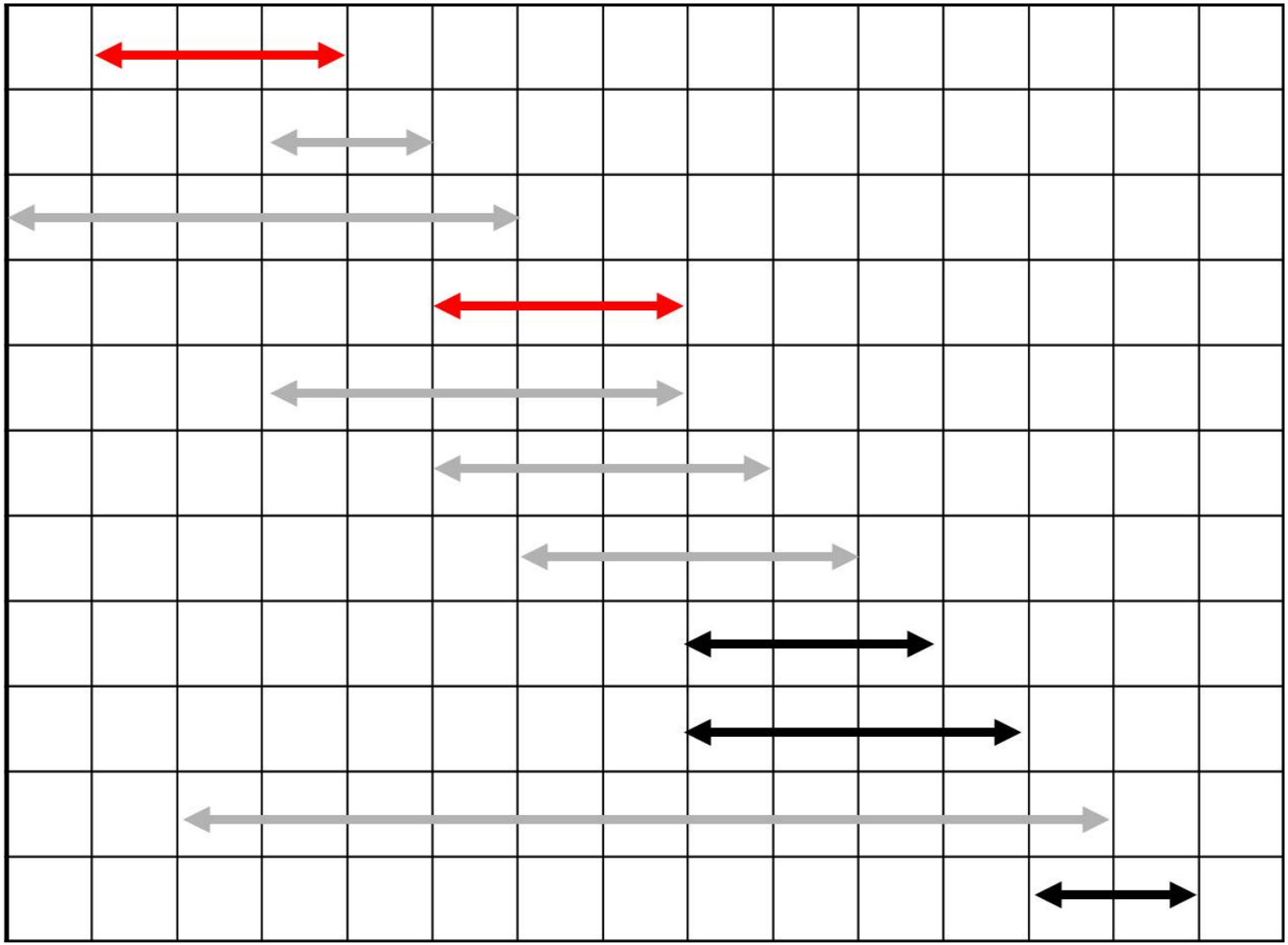
Early Finish Greedy 2



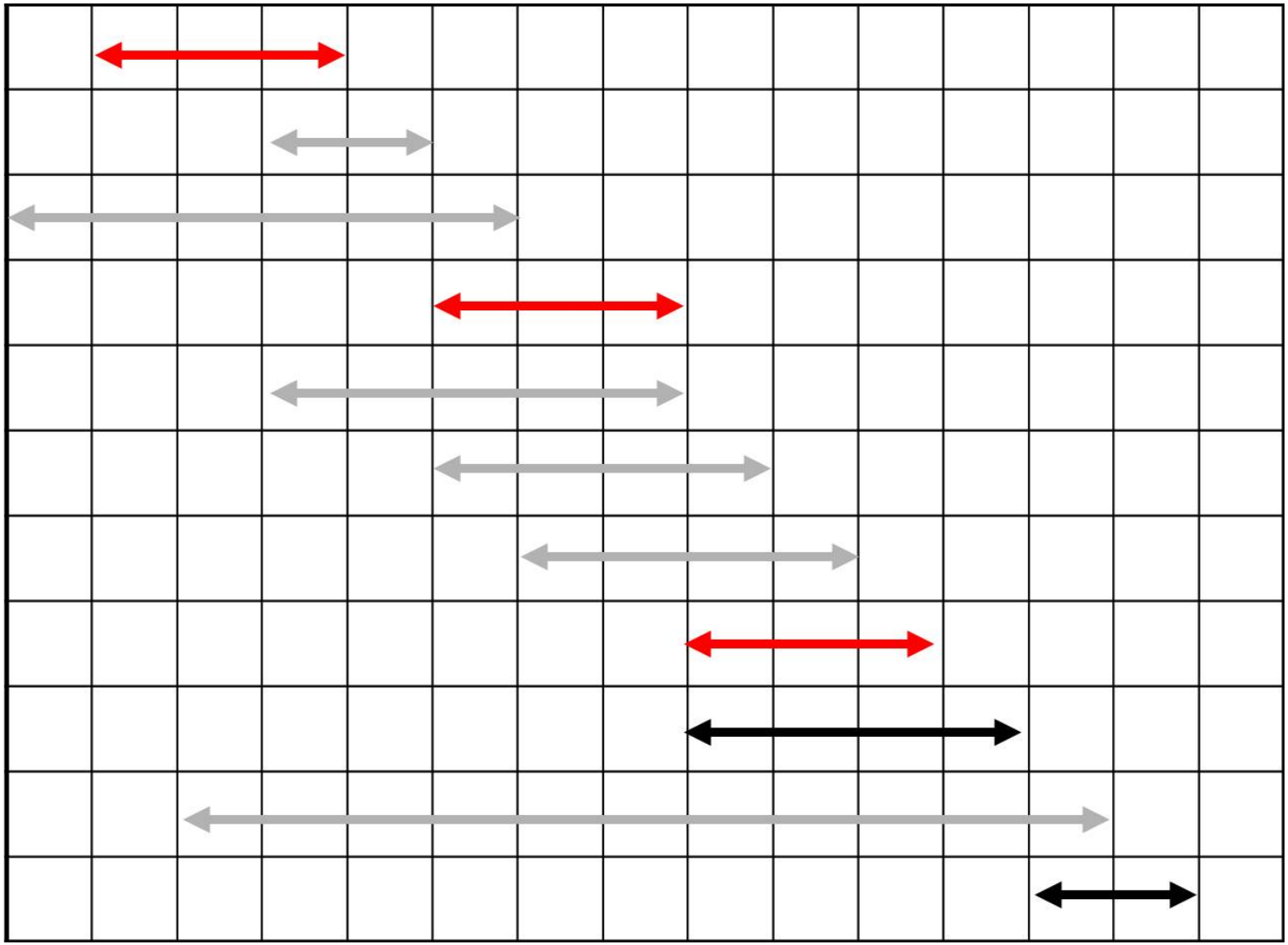
Early Finish Greedy 3



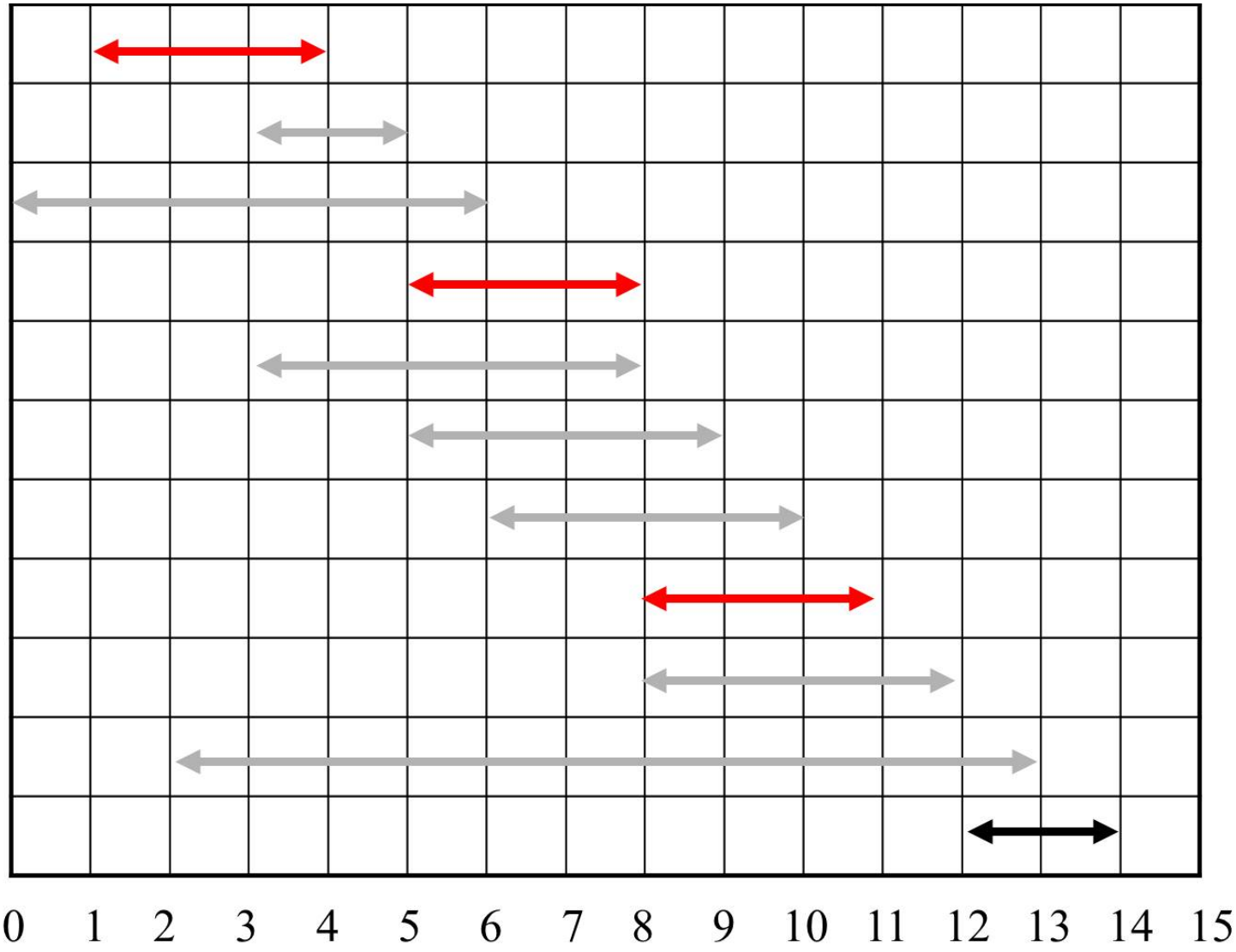
Early Finish Greedy 4



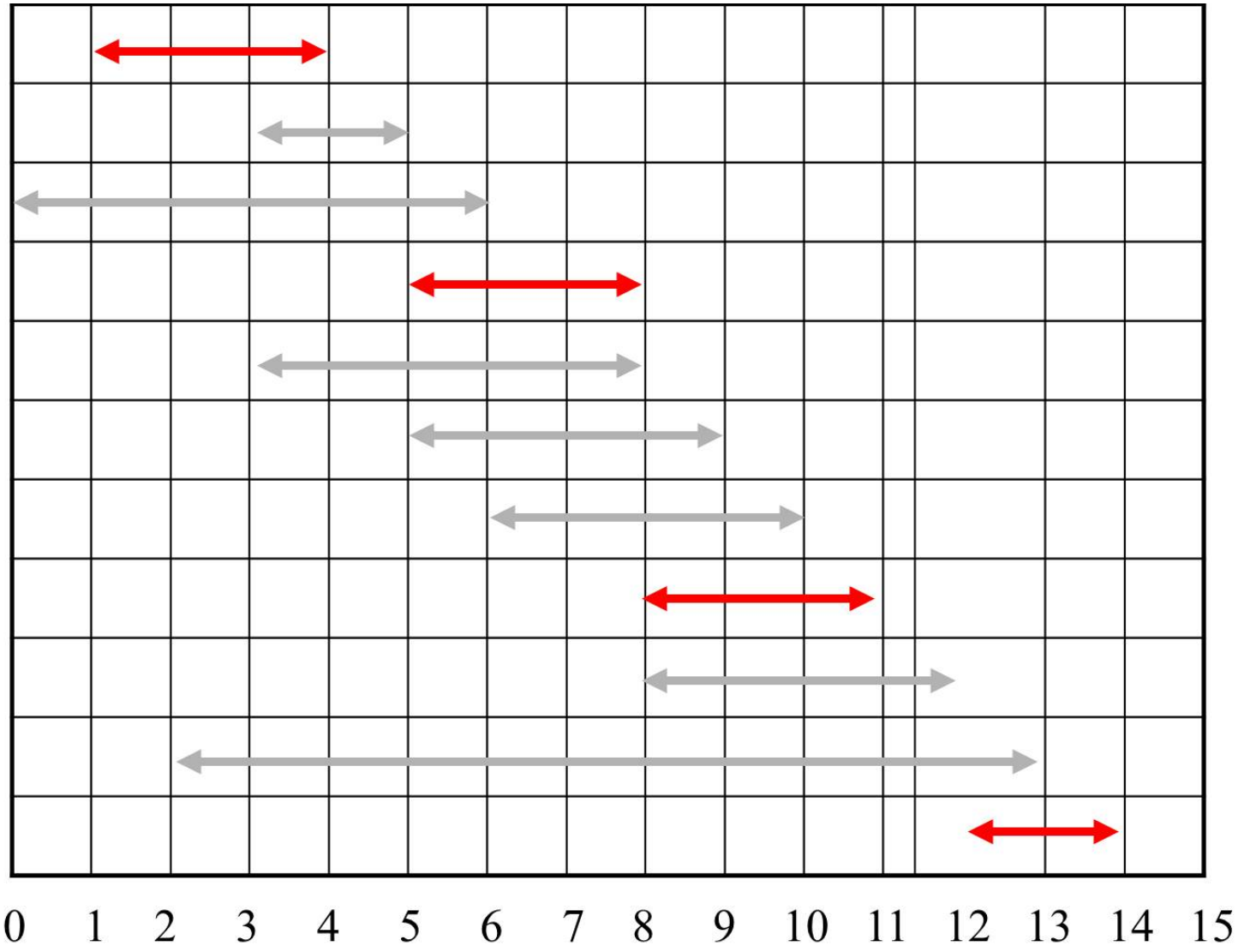
Early Finish Greedy 5



Early Finish Greedy 6



Early Finish Greedy 7



Early Finish Greedy Algorithm

GREEDY-ACTIVITY-SELECTOR(s, f)

```
1  n = s.lenth
2  A = {a_1}
3  k = 1
4  for m = 2 to n
5        if s[m] ≥ f[k]
6              A = A ∪ {a_m}
7              k = m
8  return A
```

Run time: $\Theta(n \log n)$.

Elements of Greedy Algorithms

- A greedy algorithm makes a sequence of choices, each of the choices that seems best at the moment is chosen
- NOT always produce an optimal solution
- Two ingredients that are exhibited by most problems that lend themselves to a greedy strategy
  - Greedy-choice property
  - Optimal substructure

The Greedy-Choice Property

- A globally optimal solution can be arrived at by making a locally optimal (greedy) choice
- Make whatever choice seems best at the moment and then solve the subproblem arising after the choice is made
- The choice made by a greedy algorithm may depend on choices so far, but it cannot depend on any future choices or on the solutions to subproblems
    - Of course, we must prove that a greedy choice at each step yields a globally optimal solution

Coin Changing

Let's look at the following example:

- A hot dog and a drink at Costco are \$1.50
- Plus tax it is 1.5*1.06 = \$1.59
- Often, we give the cashier 2 \$1 notes
- She need to give back, 41 cents as change
- Generally, you never see she gives you 41 pennies.

What is algorithm here?

Coin Changing Problem

- Given certain amount of change: n cents
- The denominations of coins are 25, 10, 5, 1
- How to use the fewest coins to make this change? i.e. Let $n = 25a + 10b + 5c + d$, what are the values of a, b, c, and d, minimizing $(a + b + c + d)$.
- Can you design an algorithm to solve this problem?

DP Solution

Formulation: Let $CoinDP(i)$ denote the smallest number of coins whose values add up to i. Want to compute $CoinDP(n)$.

Localization:

$$coinDP(i) = \begin{cases} +\infty, & \text{if } i < 0, \\ 0, & \text{if } i == 0, \\ i, & \text{if } i < 5, \\ 1, & \text{if } i == 5, \\ 1, & \text{if } i == 10, \\ 1, & \text{if } i == 25, \\ 1 + \min \left\{ \begin{array}{l} coinDP(i-25), \\ coinDP(i-10), \\ coinDP(i-5), \\ coinDP(i-1) \end{array} \right\}, & \text{otherwise.} \end{cases}$$

Greedy Solution

Greedy strategy: Always select coins with the largest values as many as possible.

coinGreedy(n)

```
1   if n ≥ 25
2         s = coinGreedy(n − 25)
3         s.a++;
4   elseif n ≥ 10
5         s = coinGreedy(n − 10)
6         s.b++
7   elseif n ≥ 5
8         s = coinGreedy(n − 5)
9         s.c++
10  else s = (a = 0, b = 0, c = 0, d = n, sum = n)
11  s.sum++
12  return s
```

Reasoning

- Optimal substructure
  - After the greedy choice, assuming the greedy choice is correct, can we get the optimal solution from suboptimal result?
    - Example: 38 cents
    - According to the greedy, we first choose 25.
    - It is evident that a quarter + optimal coin(38-25) is the optimal solution of 38 cents.
- Greedy choice property
  - If we do not choose the largest coin, is there a better solution?
- Discussion
  - For coin denominations of 25, 10, 5, 1,
    - The greedy choice property is not violated.
    - Will prove its correctness.
  - For other coin denominations
    - May violate it.
    - E.g. 10, 7, 1
    - 15 cents: The greedy choice generates a choice of 10,1,1,1,1
    - But 7,7,1 is optimal

Correctness Proof of Greedy Choice for 25, 10, 5, 1

Suppose that S is an optimal solution.

- Case 1: Only coin denomination of "1" is applicable. Trivial.
- Case 2: Only coin denominations of "5,1" are applicable.
  - If S contains a nickel, done.
  - Otherwise, S must contain n pennies.
  - Since $n = 5 + d$, we have $1 + d < 5 + d = n$; S cannot be optimal.
  - The greedy choice works.
- Case 3: Only coin denominations of "10,5,1" are applicable.
  - If S contains a dime, done.
  - Otherwise, it means that $n = 5c + d$.
  - Since $n \geq 10$, Case 2 implies $c \geq 2$.
  - Thus $n = 5c + d = 10 + 5(c - 2) + d$.
  - Note that $c + d > 1 + (c - 2) + d$, and so S cannot be optimal.
  - The greedy choice works.

Correctness Proof Continued

- Case 4: Coin denominations of "25,10,5,1" are applicable.
  - If S contains a quarter, done.
  - Otherwise, it means that $n = 10b + 5c + d$.
  - Since $n \geq 25$, Case 3 implies $b \geq 2$.
  - Thus, $n = 10b + 5c + d = 25 + 10(b-2) + 5(c-1) + d$.
  - Note that $b + c + d > 1 + (b-2) + (c-1) + d$, and so S cannot be optimal.
  - The greedy choice works.

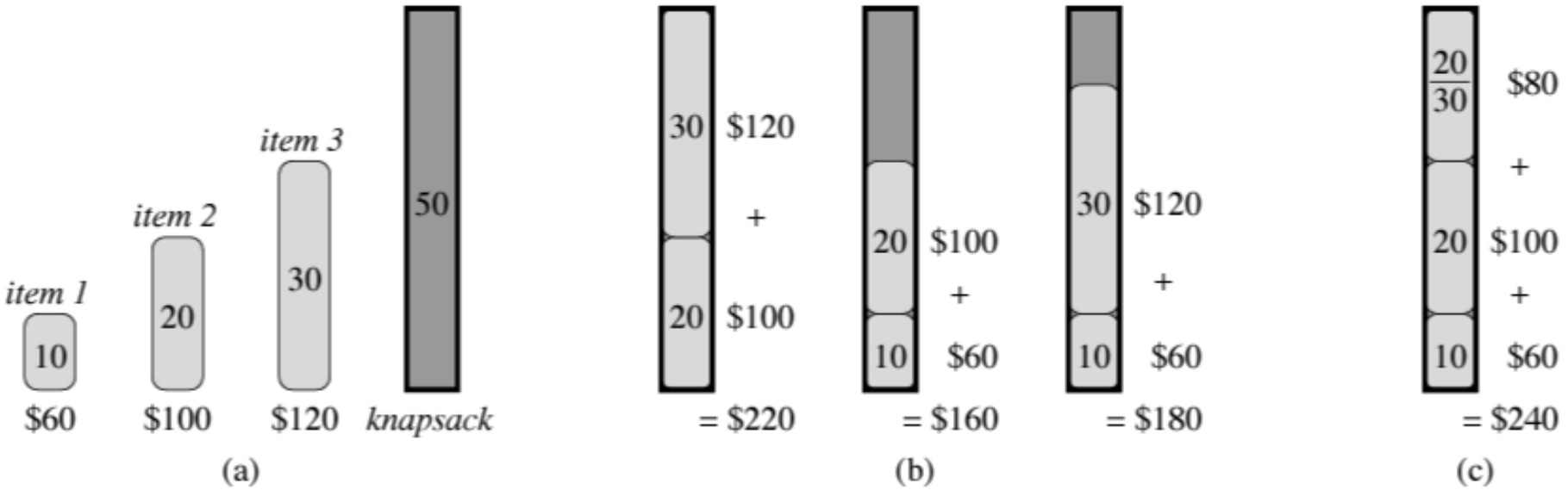
Greedy vs. DP

- Knapsack Problem.
  - One wants to pack items in a knapsack from n given items.
  - The ith item is worth v_i dollars and weighs w_i pounds.
  - Maximize the value but cannot exceed W pounds.
  - v_i, w_i, W are integers.
- 0-1 knapsack: Each item is either taken or not taken.
- Fractional knapsack: Fractions of items can be taken.
- 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 $W - w_j$.
  - Fractional: If w pounds of item j is removed from an optimal packing, the remaining packing is an optimal packing with weight at most $W - w$ that can be taken from other $n - 1$ items plus $w_j Cw$ of item j.

Greedy Solves Fractional Knapsack

- Compute the value per pound v_i/w_i for each item.
- Greedy choice: take the item as many as possible with the largest value per pound.
- If the supply of that item is exhausted and there is still room, take as many as possible of the item with the next largest value per pound, and so forth until there is no more room.
- Runtime: $O(n \log n)$ (we need to sort the items by value per pound)

Greedy Fails 0-1 Knapsack



DP for 0-1 Knapsack

Formulation: Let $K(i)$ denote the maximum values of items with the total weights at most i pounds. Want to compute $K(W)$.
Localization:

$$K(i) = \begin{cases} 0, & \text{if } i = 0, \\ \max_{1 \le w_j \le i}\{v_j + K[i - w_j]\}, & \text{otherwise.} \end{cases}$$

Runtime: $\Theta(nW)$.

Huffman Codes

- Huffman codes are prefix codes, representing a symbol (or an object) using a unique binary string, such that a prefix of any code is not a valid code.
- Huffman algorithm assigns a shorter code to a symbol with higher frequency.

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

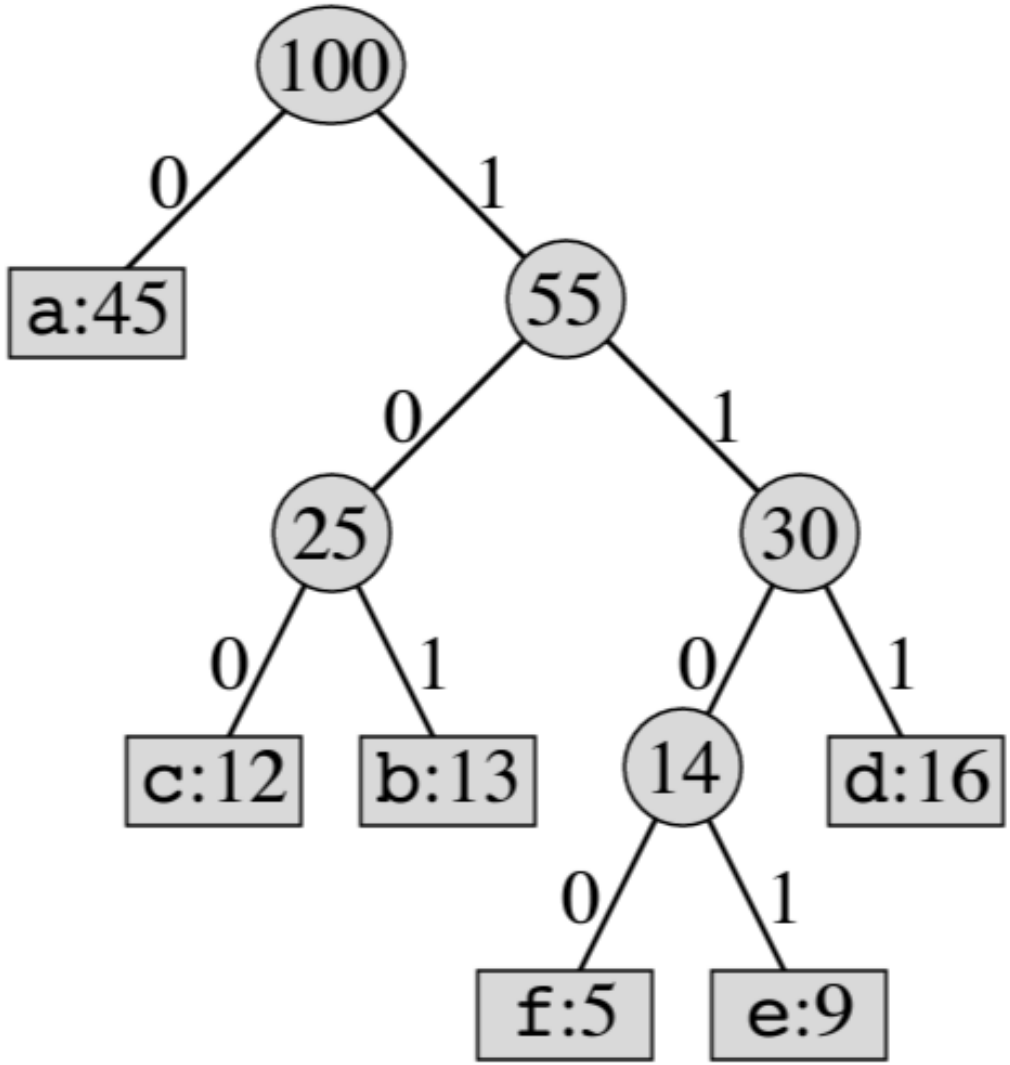
Greedy Construction of Huffman Code

- Constructing Huffman Code is constructing a summation tree, where leaves are symbols sorted according to frequencies (weights).
- These symbols are placed in a queue Q.
- In each step select two nodes x and y from Q with the smallest weights, generate a parent node z such that

$$z.weight = x.weight + y.weight.$$

- Remove x and y from Q and place z to Q.
- Repeat until $Q = \emptyset$.

Huffman Tree Example



Correctness Proof of Huffman's Algorithm

Lemma. Let x and y have the smallest frequencies. Then there exists an optimal code such that $|c(x)| = |c(y)|$ and $c(x)$ and $c(y)$ differ only in the last bit.

Proof. Let T be an optimal code tree. Convert T to a new tree such that x and y are summed first.

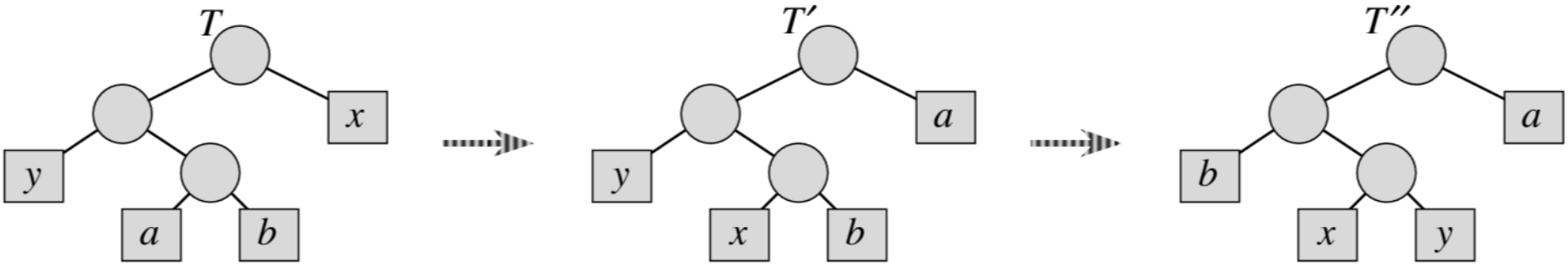


Figure 16.6 An illustration of the key step in the proof of Lemma 16.2. In the optimal tree T, leaves a and b are two siblings of maximum depth. Leaves x and y are the two characters with the lowest frequencies; they appear in arbitrary positions in T. Assuming that $x \neq b$, swapping leaves a and x produces tree T', and then swapping leaves b and y produces tree T''. Since each swap does not increase the cost, the resulting tree T'' is also an optimal tree.