CSE 246
Design & Analysis of Algorithms

Greedy Algorithms (Part 1)

Greedy Algorithm

- Greedy algorithms make the choice that looks best at the moment.
- This locally optimal choice may lead to a globally optimal solution (i.e. an optimal solution to the entire problem).

When can we use Greedy algorithms?

We can use a greedy algorithm when the following are true:

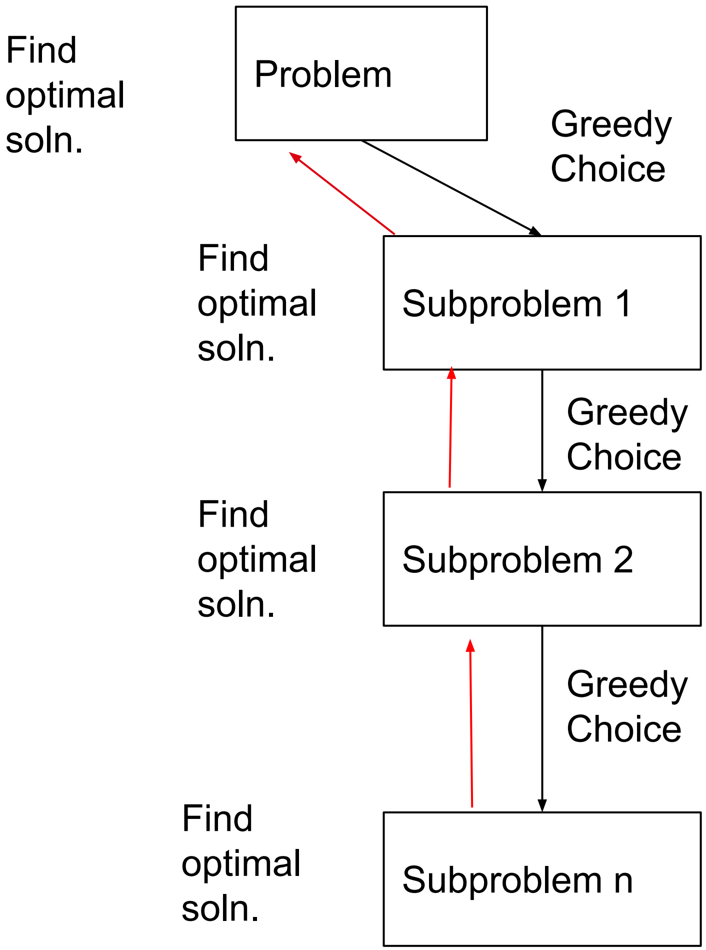
1) **The greedy choice property:** A globally optimal solution can be arrived at by making a locally optimal (greedy) choice.

1) **The optimal substructure property:** The optimal solution contains within its optimal solutions to subproblems.

Designing Greedy Algorithms

1. Cast the optimization problem as one for which:
   - we make a choice and are left with only one subproblem to solve
2. Prove the GREEDY CHOICE
   - that there is always an optimal solution to the original problem that makes the greedy choice
3. Prove the OPTIMAL SUBSTRUCTURE:
   - the greedy choice + an optimal solution to the resulting subproblem leads to an optimal solution

Optimal Substructure



Example: Making Change

- Instance: amount (in cents) to return to customer
- Problem: do this using fewest number of coins
- Example:
  - Assume that we have an unlimited number of coins of various denominations:
    - 1c (pennies), 5c (nickels), 10c (dimes), 25c (quarters), 1$ (loonies)
  - Objective: Pay out a given sum $5.64 with the smallest number of coins possible.

The Coin Changing Problem

- Assume that we have an unlimited number of coins of various denominations:
  - 1c (pennies), 5c (nickels), 10c (dimes), 25c (quarters), 1$ (loonies)
- Objective: Pay out a given sum S with the smallest number of coins possible.

- The greedy coin changing algorithm:
  - This is a $\Theta(m)$ algorithm where m = number of denominations.

```
while S > 0 do
   c := value of the largest coin no larger than S;
   num := S / c;
   pay out num coins of value c;
   S := S - num*c;
```

Example: Making Change

- E.g.:

$5.64 = $2 +$2 + $1 +
.25 + .25 + .10 +
.01 + .01 + .01 +.01

- Greedy Choice: In optimal Solution, we will always have k * P, P=maximum coin unit possible which can be taken, k = maximum can be taken. If it contradicts -> Solution Fails
- Optimal Substructure Property: Lets, optimal solution is S= C1 * k1 + C2 * K2 +C3 * k3 where C1 > C2 > C3 and each are coin unit, Now a subproblem is S` = S - (C1 * k1). S`'s best solution has to be gained from (C2 * K2 +C3 * k3), Otherwise optimal substructure property fails.

Making Change – A big problem

- Example 2: Coins are valued \$.30, \$.20, \$.05, \$.01
  - Does not have greedy-choice property, since \$.40 is best made with two \$.20's, but the greedy solution will pick three coins (which ones?)

More examples

- Euro Coins (in cents) : Supports Greedy prop.
  1 , 2, 5, 10, 20, 50, 100, 200
- {1, 3, 4} -> Not supports Greedy prop, ex: 6

The Fractional Knapsack Problem

- **Given:** A set S of n items, with each item i having
  - b_i - a positive benefit
  - w_i - a positive weight
- **Goal:** Choose items with maximum total benefit but with weight at most W.
- If we are allowed to take fractional amounts, then this is the **fractional knapsack problem**.
  - In this case, we let x_i denote the amount we take of item i
  - Objective: maximize

$$\sum_{i \in S} b_i (x_i / w_i)$$

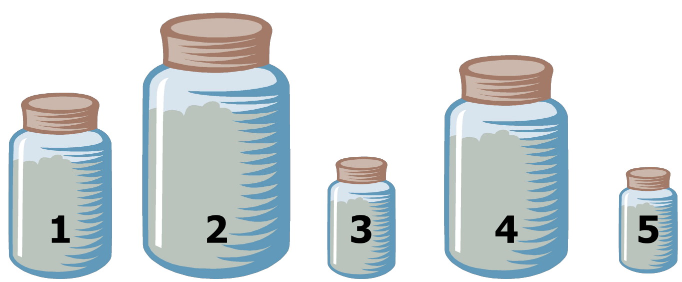
  - Constraint:

$$\sum_{i \in S} x_i \leq W, 0 \leq x_i \leq w_i$$

Example

- Given: A set S of n items, with each item i having
  - b_i - a positive benefit
  - w_i - a positive weight
- Goal: Choose items with maximum total benefit but with total weight at most W.

Items:



Weight:	4 ml	8 ml	2 ml	6 ml	1 ml
Benefit:	$12	$32	$40	$30	$50
Value: ($ per ml)	3	4	20	5	50

10 ml

"knapsack"

Solution: P

- 1 ml of 5 50$
- 2 ml of 3 40$
- 6 ml of 4 30$
- 1 ml of 2 4$

- Total Profit:124$

The Fractional Knapsack Algorithm

- Greedy choice: Keep taking item with highest **value** (benefit to weight ratio)
  - Since $$\sum_{i \in S} b_i (x_i / w_i) = \sum_{i \in S} (b_i / w_i) x_i$$

Algorithm ***fractionalKnapsack(S, W)***
Input: **set S of items w/ benefit b_i and weight w_i; max. weight W**
Output: **amount x_i of each item i to maximize benefit w/ weight at most W**

for ***each item i in S***
  $x_i \leftarrow 0$
  $v_i \leftarrow b_i / w_i$ {value}
$w \leftarrow 0$ {total weight}
while $w < W$
  remove item i with highest v_i
  $x_i \leftarrow \min\{w_i, W - w\}$
  $w \leftarrow w + \min\{w_i, W - w\}$

The Fractional Knapsack Algorithm

- Running time: Given a collection S of n items, such that each item i has a benefit b_i and weight w_i, we can construct a maximum-benefit subset of S, allowing for fractional amounts, that has a total weight W in O(nlogn) time.
  - Use heap-based priority queue to store S
  - Removing the item with the highest value takes O(logn) time
  - In the worst case, need to remove all items

An Activity Selection Problem (Conference Scheduling Problem)

- **Input: A set of activities S = {a_1,…, a_n}**
- Each activity has start time and a finish time
  - $a_i=(s_i, f_i)$
- Two activities are compatible if and only if their interval does not overlap
- **Output: a maximum-size subset of mutually compatible activities**

The Activity Selection Problem

- Here are a set of start and finish times

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

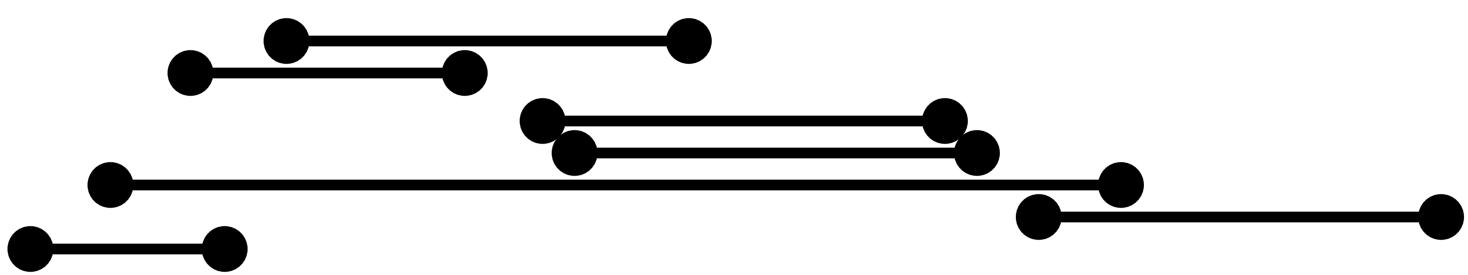
- What is the maximum number of activities that can be completed?
  - $\{a_3, a_9, a_{11}\}$ can be completed
  - But so can $\{a_1, a_4, a_8, a_{11}\}$ which is a larger set
  - But it is not unique, consider $\{a_2, a_4, a_9, a_{11}\}$

The Activity Selection Problem

Input: list of time-intervals L

Output: a non-overlapping subset S of the intervals

Objective: maximize |S|



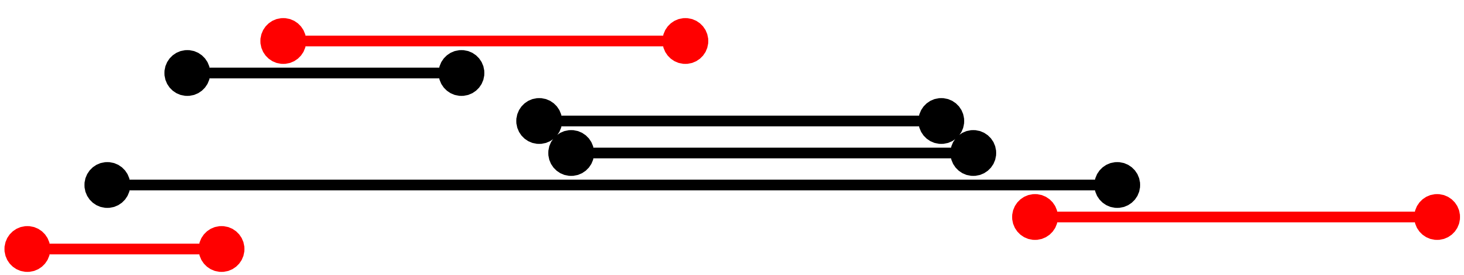
3,7
2,4
5,8
6,9
1,11
10,12
0,3

The Activity Selection Problem

Input: list of time-intervals L

Output: a non-overlapping subset S of the intervals

Objective: maximize |S|



3,7
2,4
5,8
6,9
1,11
10,12
0,3

Answer = 3

The Activity Selection Problem

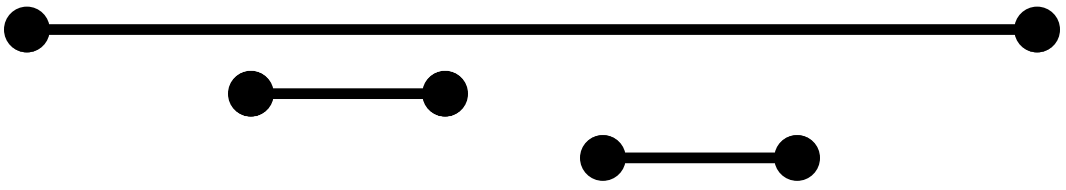
Algorithm 1:

1. sort the activities by the starting time
2. pick the first activity a
3. remove all activities conflicting with a
4. repeat

The Activity Selection Problem

Algorithm 1:

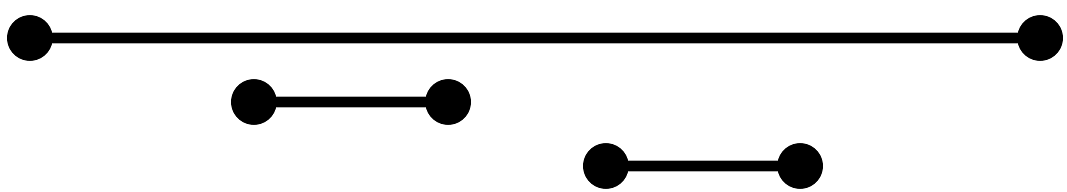
1. sort the activities by the starting time
2. pick the first activity ***"a"***
3. remove all activities conflicting with ***"a"***
4. repeat



The Activity Selection Problem

Algorithm 1:

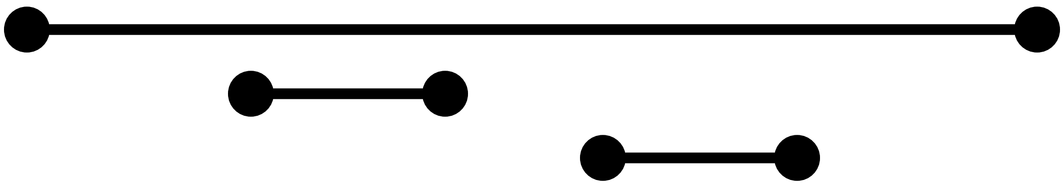
1. sort the activities by the starting time
2. pick the first activity ***“a”***
3. remove all activities conflicting with ***“a”***
4. repeat



The Activity Selection Problem

Algorithm 2:

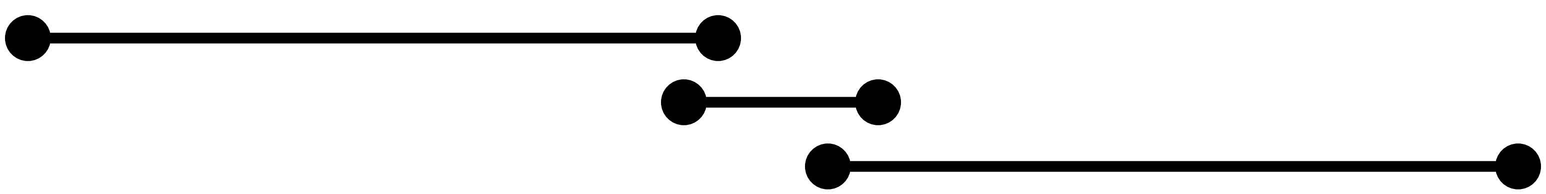
1. sort the activities by length
2. pick the shortest activity ***"a"***
3. remove all activities conflicting with ***"a"***
4. repeat



The Activity Selection Problem

Algorithm 2:

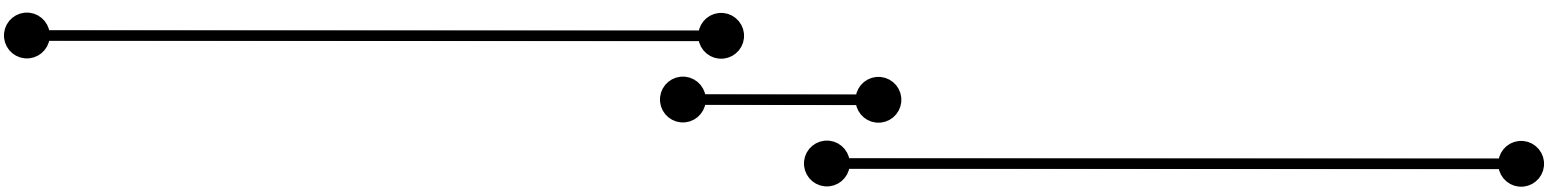
1. sort the activities by length
2. pick the shortest activity ***"a"***
3. remove all activities conflicting with ***"a"***
4. repeat



The Activity Selection Problem

Algorithm 2:

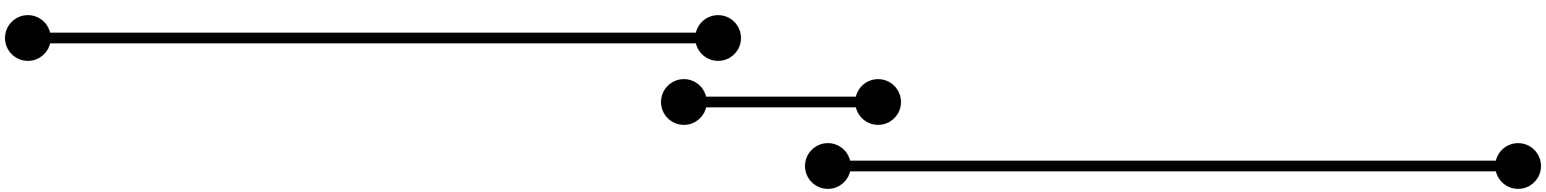
1. sort the activities by length
2. pick the shortest activity ***"a"***
3. <u>remove</u> all activities conflicting with ***"a"***
4. repeat



The Activity Selection Problem

Algorithm 3:

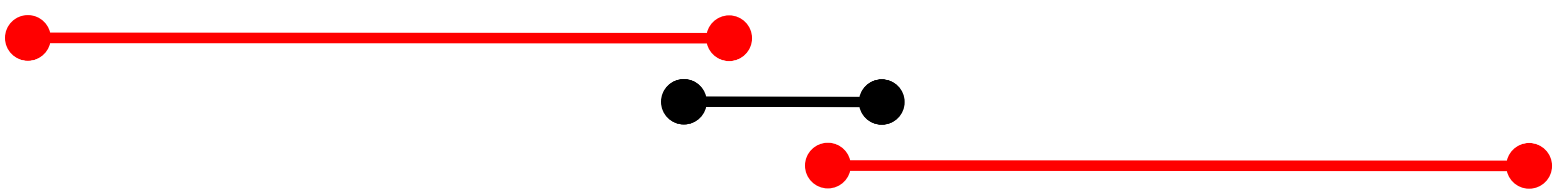
1. sort the activities by ending time
2. pick the activity which ends first
3. remove all activities conflicting with a
4. repeat



The Activity Selection Problem

Algorithm 3:

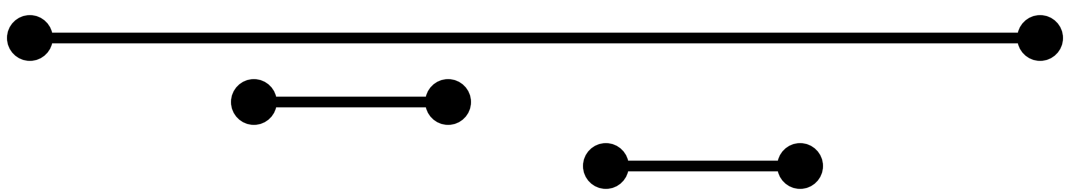
1. sort the activities by ending time
2. pick the activity which ends first
3. remove all activities conflicting with a
4. repeat



The Activity Selection Problem

Algorithm 3:

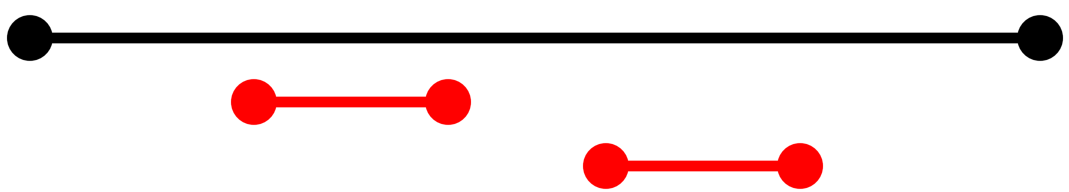
1. sort the activities by ending time
2. pick the activity which ends first
3. remove all activities conflicting with a
4. repeat



The Activity Selection Problem

Algorithm 3:

1. sort the activities by ending time
2. pick the activity which ends first
3. remove all activities conflicting with a
4. repeat



The Activity Selection Problem

Algorithm 3:

1. sort the activities by ending time
2. pick the activity a which ends first
3. remove all activities conflicting with a
4. repeat

Theorem:

Algorithm 3 gives an optimal solution to the activity selection problem.

Activity Selection Algorithm

Idea: At each step, select the activity with the smallest finish time that is compatible with the activities already chosen.

```
Greedy-Activity-Selector(s, f)
    n <- length[s]
    A <- {1}              {Automatically select first activity}
    j <- 1                {Last activity selected so far}
    for i <- 2 to n do
        if si >= fj then
            A <- A U {i}  {Add activity i to the set}
            j <- i        {record last activity added}
    return A
```

The Activity Selection Problem

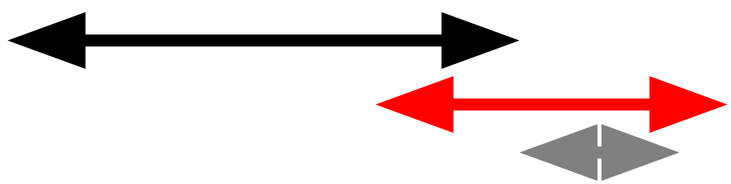
- Here are a set of start and finish times

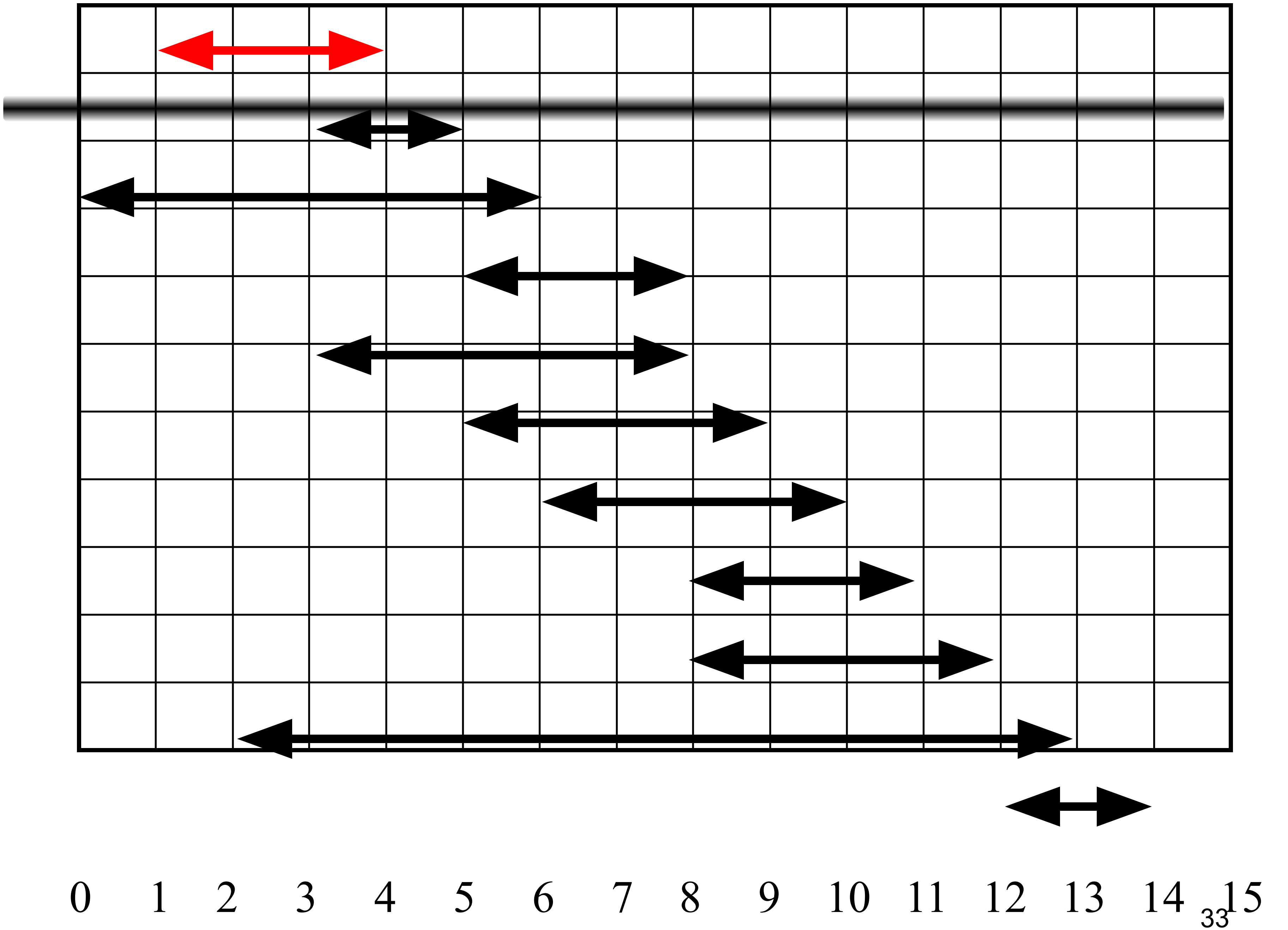
i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

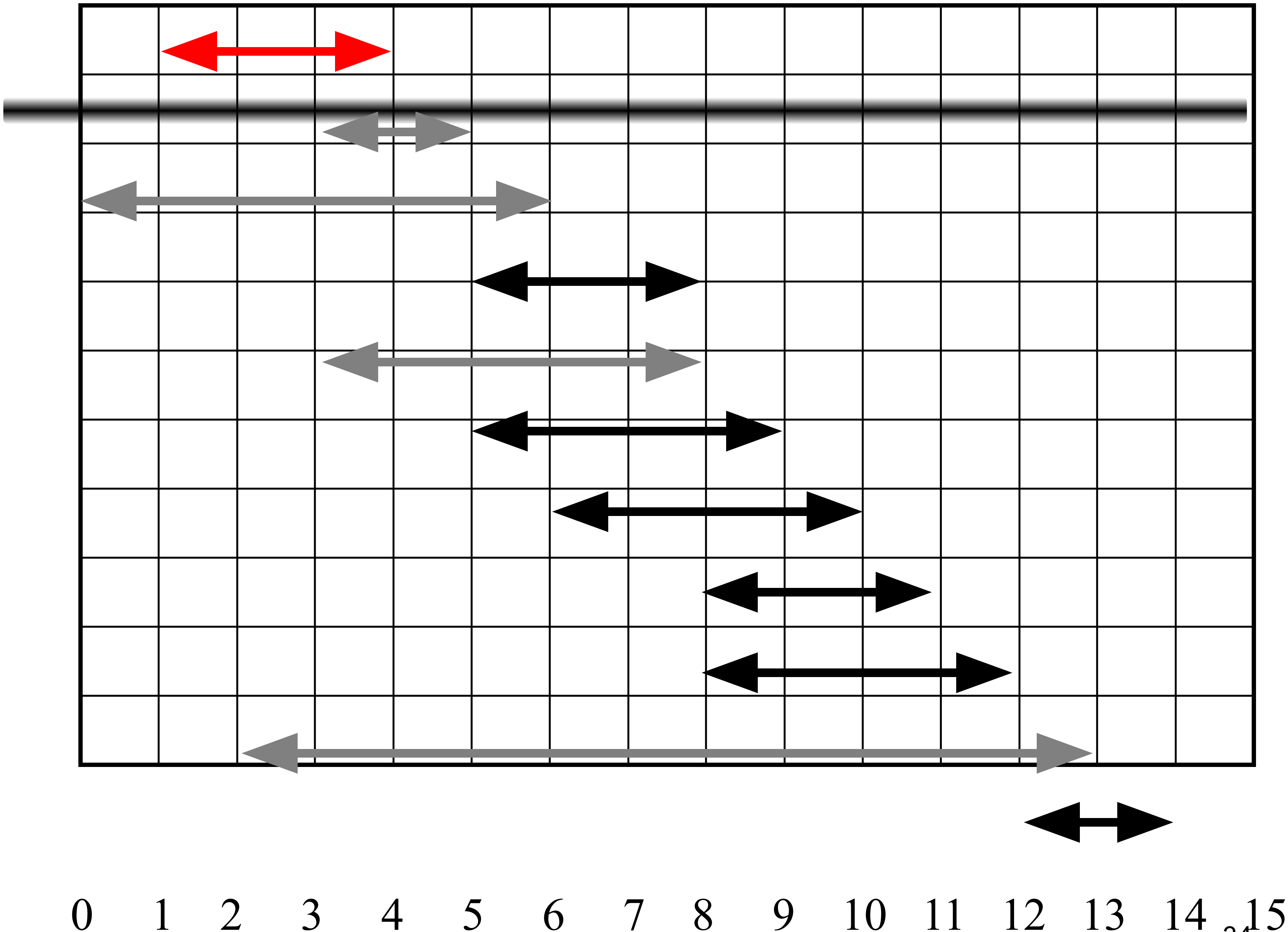
- What is the maximum number of activities that can be completed?
  - $\{a_3, a_9, a_{11}\}$ can be completed
  - But so can $\{a_1, a_4, a_8, a_{11}\}$ which is a larger set
  - But it is not unique, consider $\{a_2, a_4, a_9, a_{11}\}$

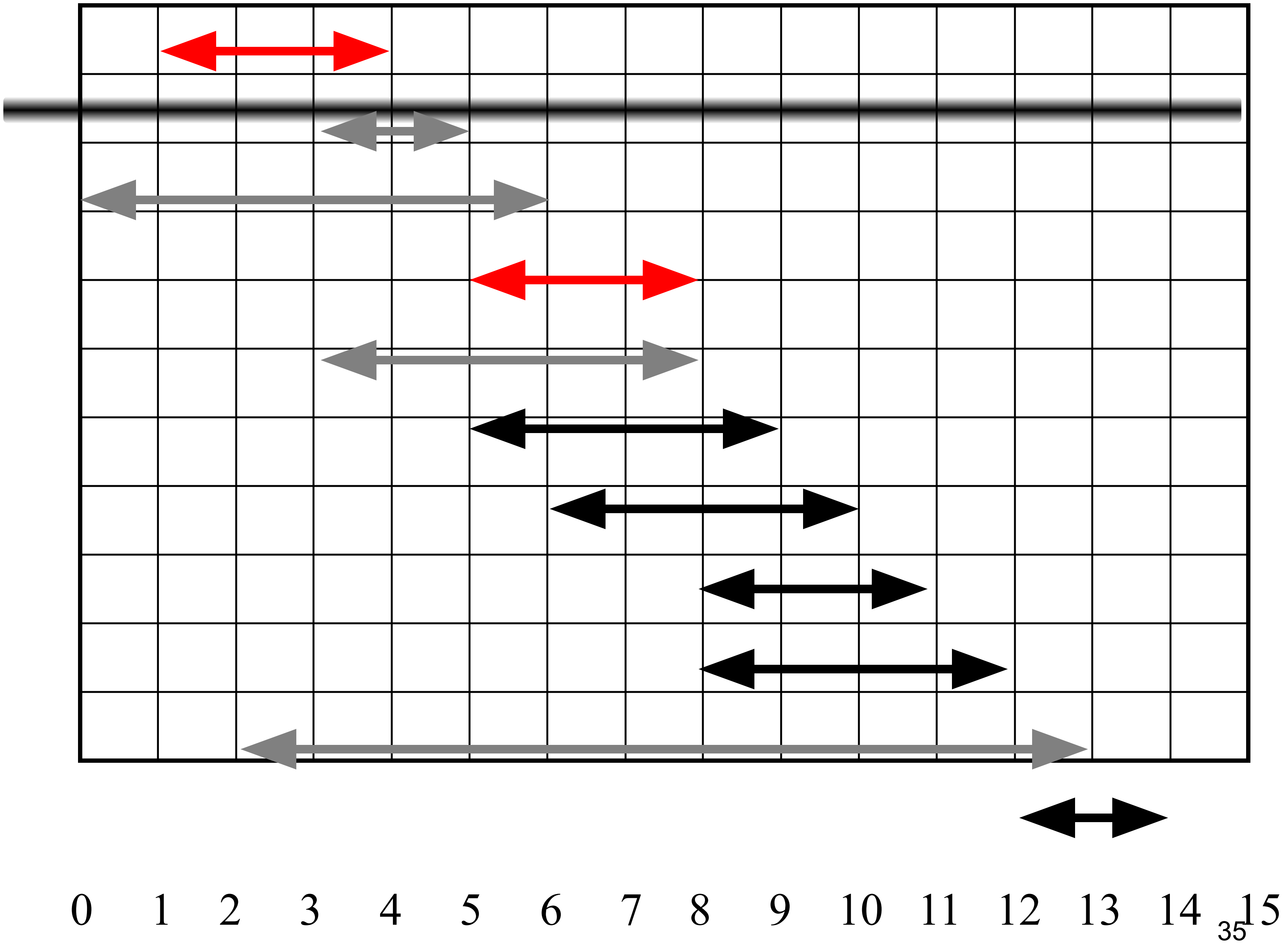
Interval Representation

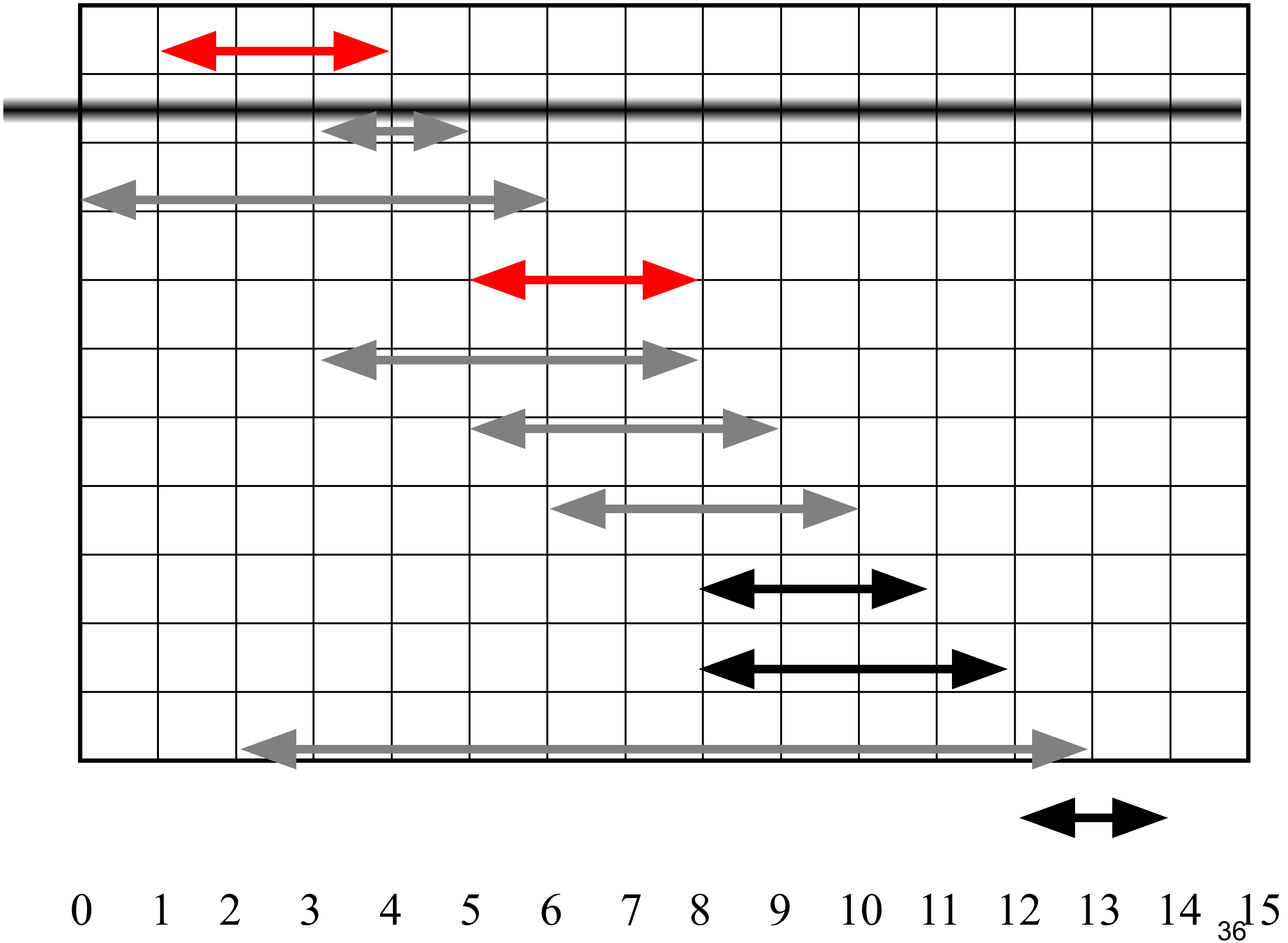
i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

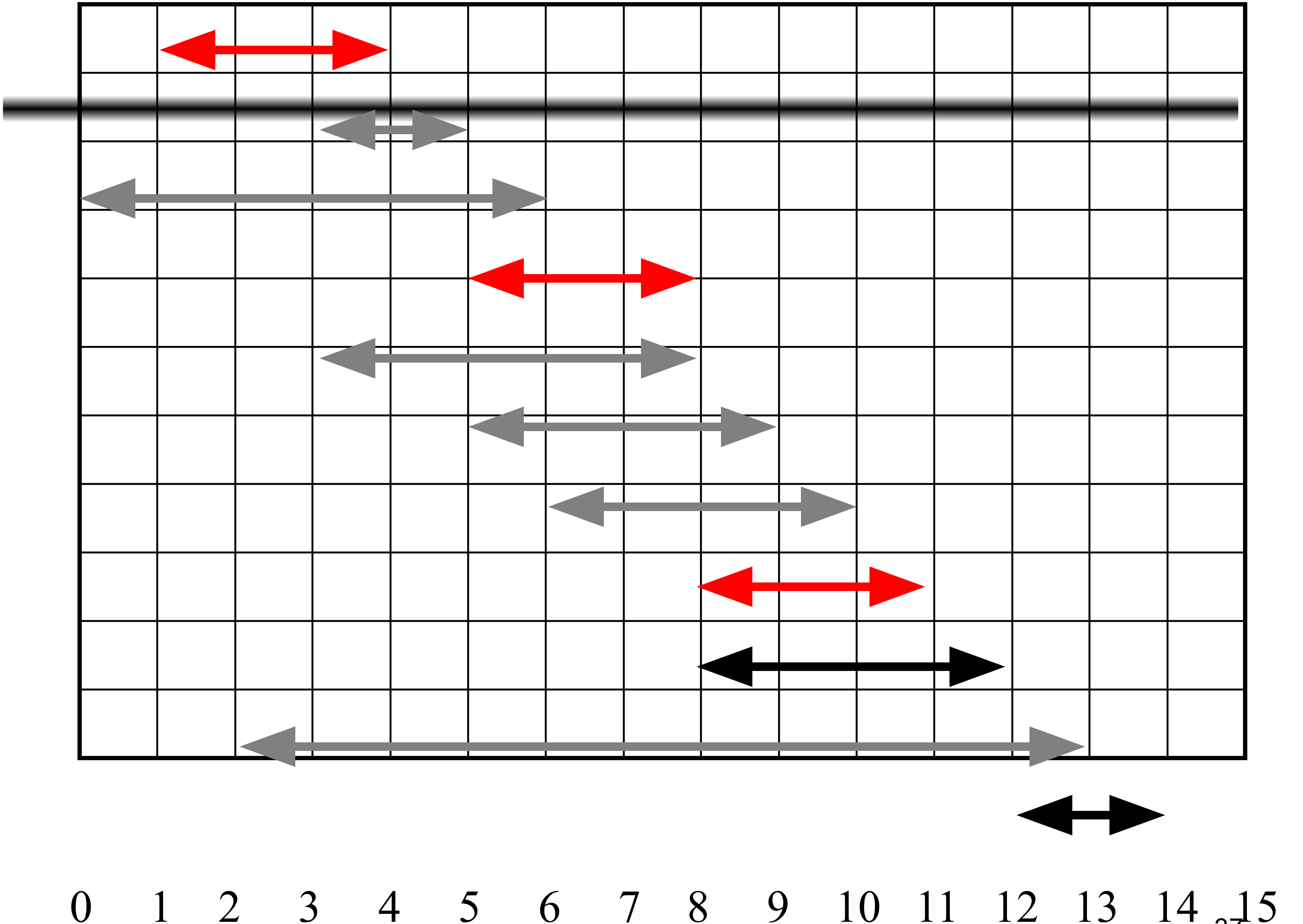


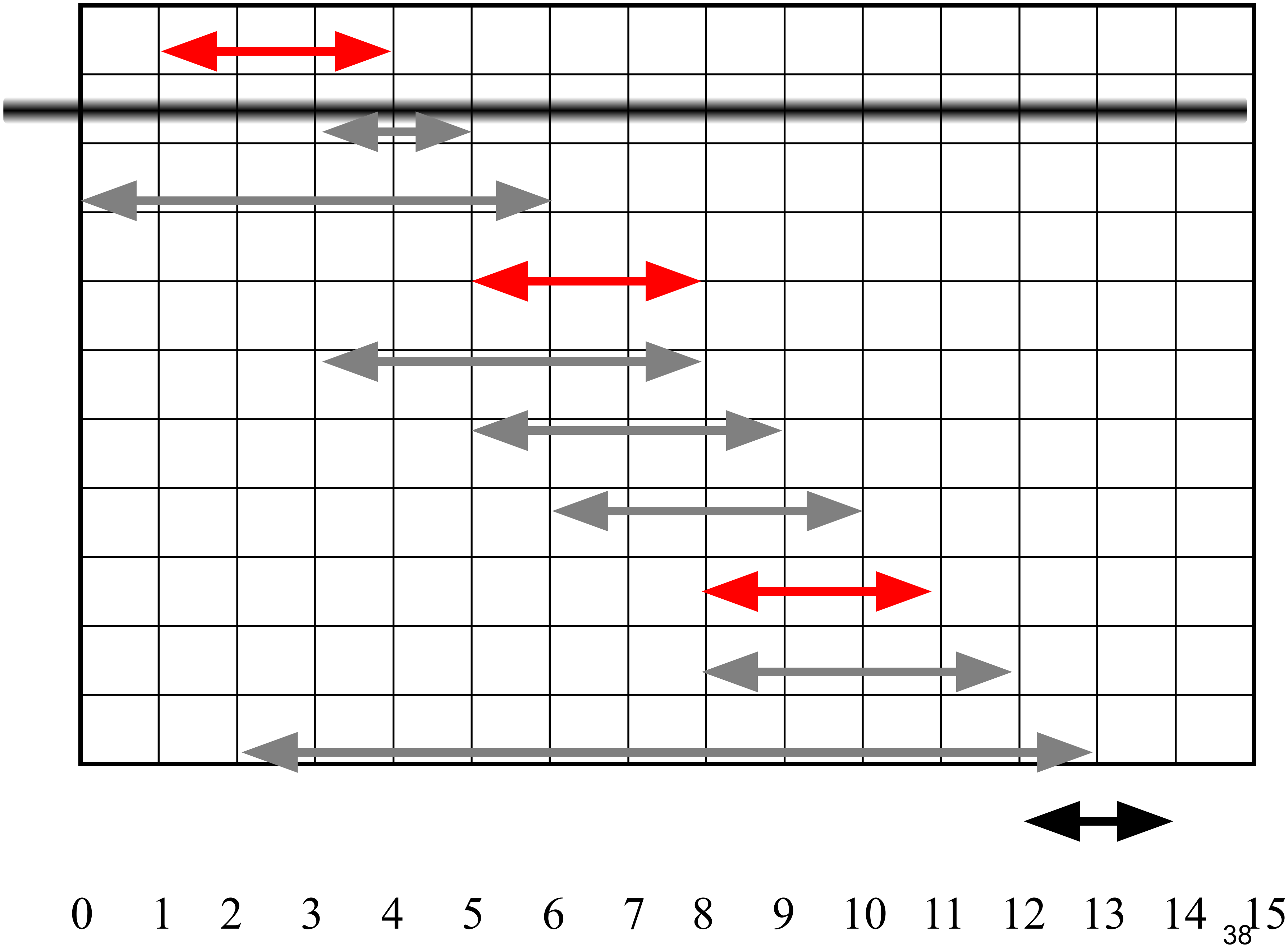

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

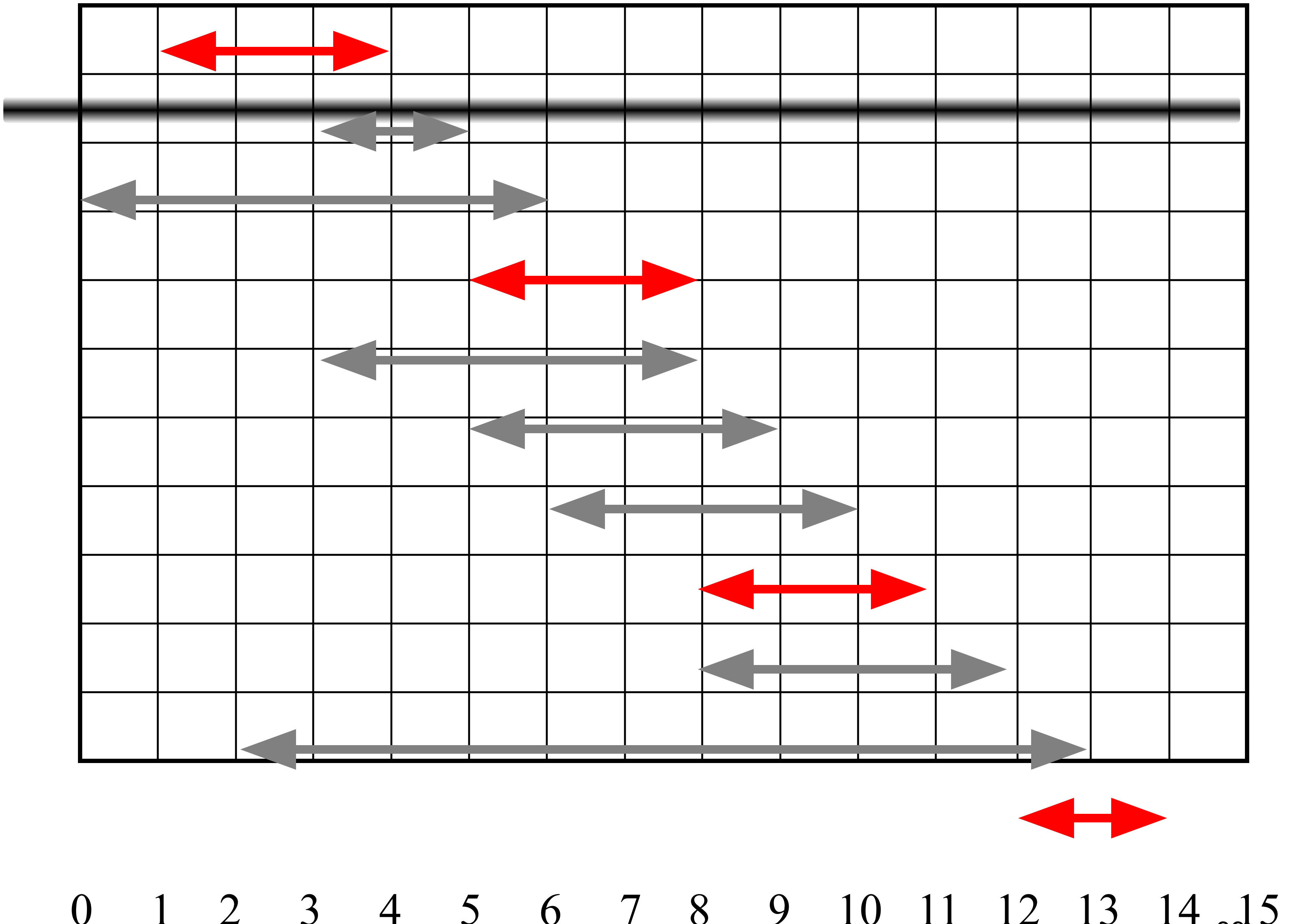

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15


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

Why this Algorithm is Optimal?

- We will show that this algorithm uses the following properties
  - The problem has the optimal substructure property
  - The algorithm satisfies the greedy-choice property
- Thus, it is Optimal

Greedy-Choice Property

- Show there is an optimal solution that begins with a greedy choice (with activity 1, which as the earliest finish time)
- Suppose A ⊆ S in an optimal solution
  - Order the activities in A by finish time. The first activity in A is k
    - If k = 1, the schedule A begins with a greedy choice
    - If k ≠ 1, show that there is an optimal solution B to S that begins with the greedy choice, activity 1
  - Let B = A – {k} ∪ {1}
    - $f_1 \leq f_k$ □ activities in B are disjoint (compatible)
    - B has the same number of activities as A
    - Thus, B is optimal

Optimal Substructures

- Once the greedy choice of activity 1 is made, the problem reduces to finding an optimal solution for the activity-selection problem over those activities in S that are compatible with activity 1
  - Optimal Substructure
  - If A is optimal to S, then $A' = A - \{1\}$ is optimal to $S'=\{i \in S: s_i \geq f_1\}$
  - Why?
    - If we could find a solution B' to S' with more activities than A', adding activity 1 to B' would yield a solution B to S with more activities than A contradicting the optimality of A
- After each greedy choice is made, we are left with an optimization problem of the same form as the original problem
  - By induction on the number of choices made, making the greedy choice at every step produces an optimal solution

Huffman Codes

- Widely used technique for data compression
- Assume the data to be a sequence of characters
- Looking for an effective way of storing the data
- ***Binary character code***
  - Uniquely represents a character by a binary string

Fixed-Length Codes

E.g.: Data file containing 100,000 characters

	a	b	c	d	e	f
Frequency (thousands)	45	13	12	16	9	5

- 3 bits needed
- a = 000, b = 001, c = 010, d = 011, e = 100, f = 101
- Requires: $100{,}000 \cdot 3 = 300{,}000$ bits

Huffman Codes

- Idea:
  - Use the frequencies of occurrence of characters to build a optimal way of representing each character

	a	b	c	d	e	f
Frequency (thousands)	45	13	12	16	9	5

Variable-Length Codes

E.g.: Data file containing 100,000 characters

	a	b	c	d	e	f
Frequency (thousands)	45	13	12	16	9	5

- Assign short codewords to frequent characters and long codewords to infrequent characters
- a = 0, b = 101, c = 100, d = 111, e = 1101, f = 1100
- $(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 1{,}000$

  $= 224{,}000$ bits

Prefix Codes

- Prefix codes:
  - Codes for which no codeword is also a prefix of some other codeword
  - Better name would be “prefix-free codes”
- We can achieve optimal data compression using prefix codes
  - We will restrict our attention to prefix codes

Encoding with Binary Character Codes

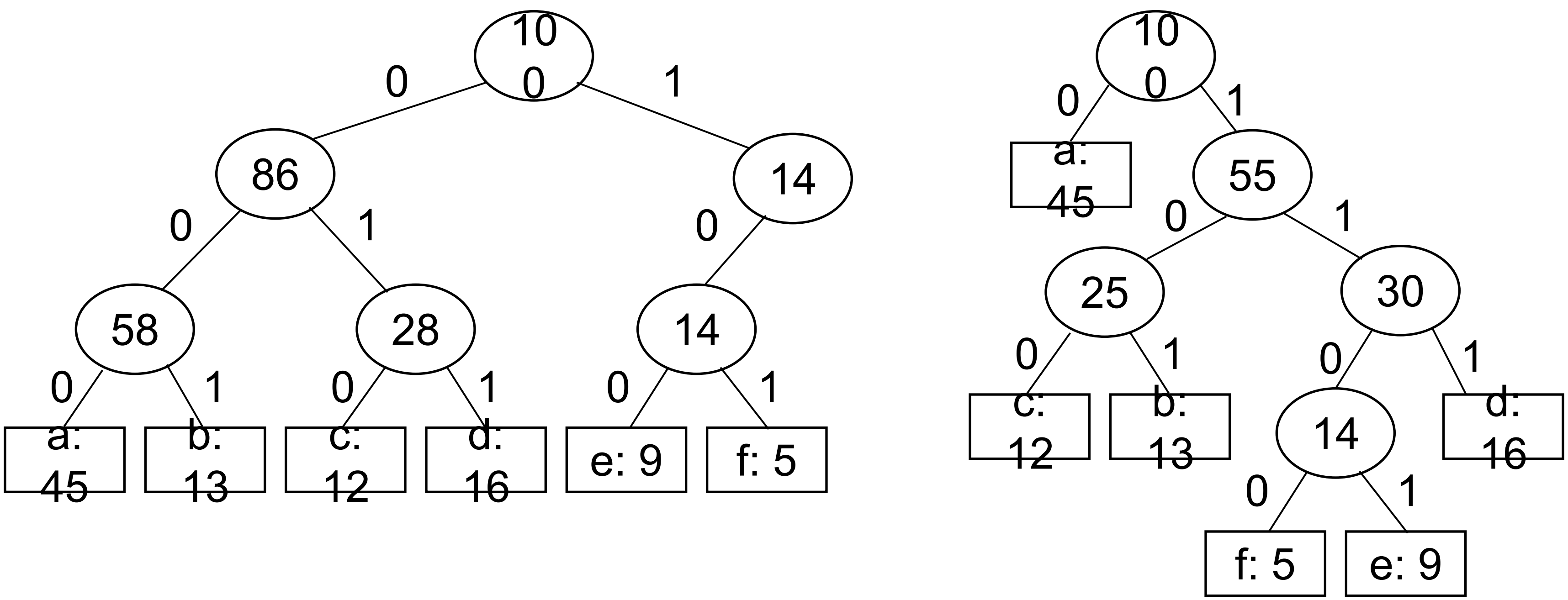
- Encoding
  - Concatenate the codewords representing each character in the file
- *E.g.*:
  - a = 0, b = 101, c = 100, d = 111, e = 1101, f = 1100
  - abc = 0 $\cdot$ 101 $\cdot$ 100 = 0101100

Decoding with Binary Character Codes

- Prefix codes simplify decoding
  - No codeword is a prefix of another ⇒ the codeword that begins an encoded file is unambiguous
- Approach
  - Identify the initial codeword
  - Translate it back to the original character
  - Repeat the process on the remainder of the file
- *E.g.*:
  - a = 0, b = 101, c = 100, d = 111, e = 1101, f = 1100
  - 001011101 = 0 · 0 · 101 · 1101 = aabe

Prefix Code Representation

- Binary tree whose leaves are the given characters
- Binary codeword
  - the path from the root to the character, where 0 means “go to the left child” and 1 means “go to the right child”
- Length of the codeword
  - Length of the path from root to the character leaf (depth of node)



Optimal Codes

- An optimal code is always represented by a **full binary tree**
  - Every non-leaf has two children
  - Fixed-length code is not optimal, variable-length is
- How many bits are required to encode a file?
  - Let C be the alphabet of characters
  - Let $f(c)$ be the frequency of character c
  - Let $d_T(c)$ be the depth of c's leaf in the tree T corresponding to a prefix code

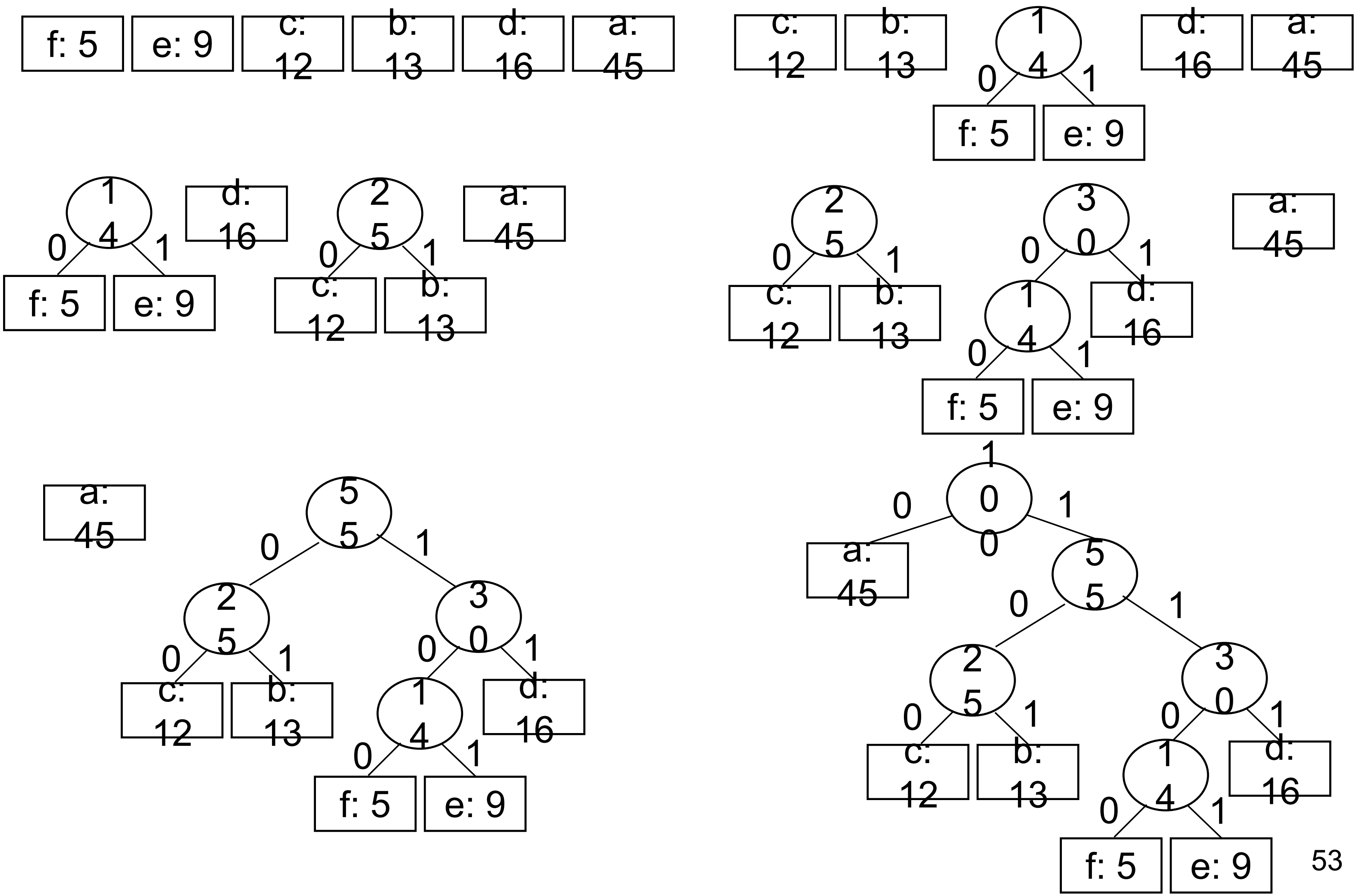
$$B(T) = \sum_{c \in C} f(c) d_T(c)$$ the cost of tree T

Constructing a Huffman Code

- A greedy algorithm that constructs an optimal prefix code called a **Huffman code**
- Assume that:
  - C is a set of n characters
  - Each character has a frequency $f(c)$
  - The tree T is built in a bottom up manner
- Idea:
  - Start with a set of $|C|$ leaves
  - At each step, merge the two least frequent objects: the frequency of the new node = sum of two frequencies
  - Use a min-priority queue Q, keyed on f to identify the two least frequent objects

f: 5	e: 9	c: 12	b: 13	d: 16	a: 45

Example



Building a Huffman Code

Alg.: HUFFMAN(C) Running time: $O(n\lg n)$

```
n ← |C|
Q ← C                                        O(n)
for i ← 1 to n - 1
    do allocate a new node z
        left[z] ← x ← EXTRACT-MIN(Q)
        right[z] ← y ← EXTRACT-MIN(Q)         O(nlgn)
        f[z] ← f[x] + f[y]
        INSERT (Q, z)
return EXTRACT-MIN(Q)
```