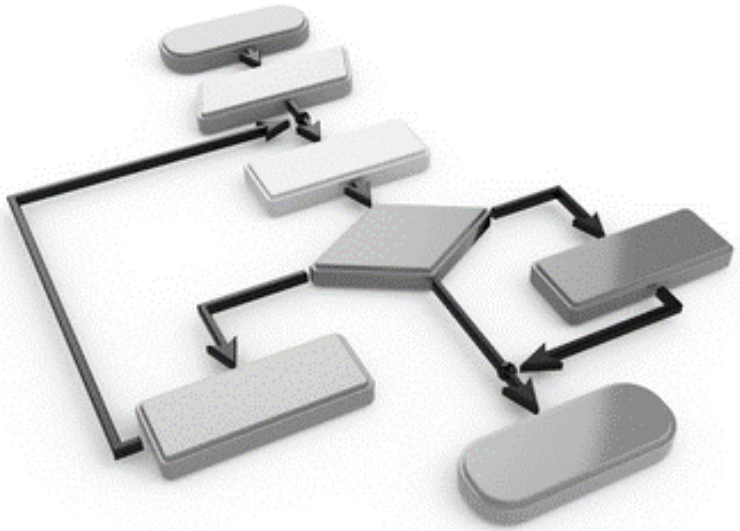


Lecture 04

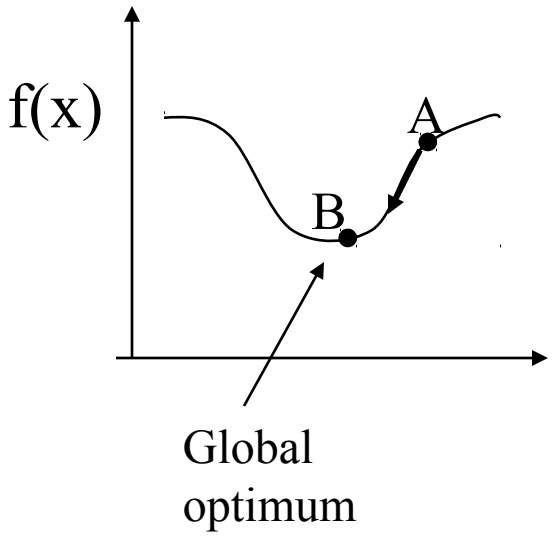
Greedy Algorithms

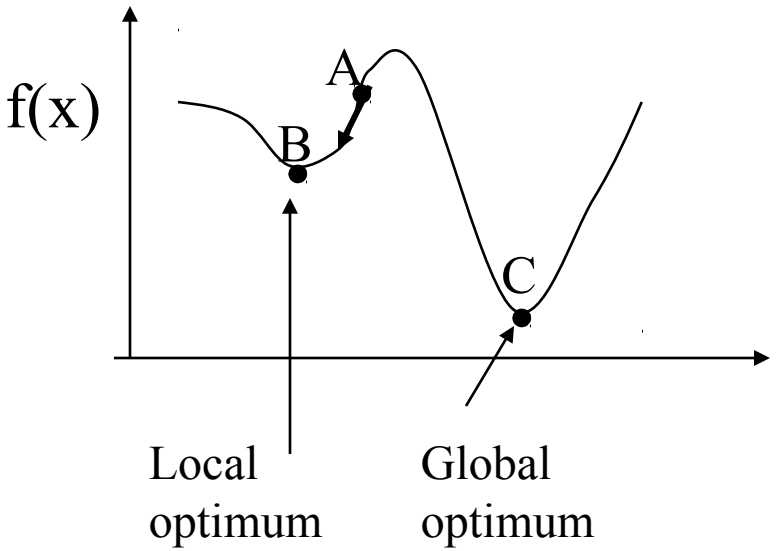
CSE373: Design and Analysis of Algorithms

Greedy Algorithm

- Solves an optimization problem
- Example:
  - Activity Selection Problem
  - Dijkstra’s Shortest Path Problem
  - Minimum Spanning Tree Problem
- For many optimization problems, greedy algorithm can be used. (not always)
- Algorithm for optimization problems typically go through a sequence of steps, with a set of choices at each step. Choice does not depend on evaluating potential future choices or presolving overlapping subproblems. With each step, the original problem is reduced to a smaller problem.
- Greedy algorithm always makes the choice that looks best at the moment.
- It makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution.

Finding the Optimal Solution





If we start at A and move in the direction of descent, we will end up at the local optimum, B. On the left graph, B is also at the global optimum. On the right graph, the global optimum is elsewhere, at C.

Elements of the Greedy Strategy

How can one tell if a greedy algorithm will solve a particular optimization problem??

There is no way in general. But there are 2 ingredients exhibited by most greedy problems:

1. Greedy Choice Property
2. Optimal Sub Structure

Greedy Choice Property

A globally optimal solution can be arrived at by making a locally optimal (**Greedy**) choice.

We make whatever choice seems best at the moment and then solve the sub problems 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.

Thus, a greedy strategy usually progresses in a ***top-down*** fashion, making one greedy choice after another, iteratively reducing each given problem instance to a smaller one.

Optimal Sub Structure

A problem exhibits ***optimal substructure*** if an optimal solution to the **problem** contains (within it) optimal solution to **sub-problems**

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
3. Prove that the GREEDY CHOICE property holds
   - that there is always an optimal solution to the original problem that makes the greedy choice
4. Prove that the OPTIMAL SUBSTRUCTURE property holds
   - the greedy choice + an optimal solution to the resulting subproblem leads to an optimal solution

The Activity Selection Problem

Definition: Scheduling a resource among several competing activities.

Elaboration: Suppose we have a set $S = \{1, 2, \ldots, n\}$ of n proposed activities that wish to allocate a resource, such as a lecture hall, which can be used by only one activity at a time. Each activity i has a ***start time*** $s[i]$ and ***finish time*** $f[i]$ where $s[i] <= f[i]$.

Compatibility: Activities i and j are compatible if the interval $[s[i], f[i])$ and $[s[j], f[j])$ do not overlap (i.e. $s[i] >= f[j]$ or $s[j] >= f[i]$)

Goal: To select a maximum- size set 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
Sj	1	3	0	5	3	5	6	8	8	2	12
fj	4	5	6	7	9	9	10	11	12	14	16

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

- {a3, a9, a11} can be completed
- But so can {a1, a4, a8, a11} which is a larger set
- But it is not unique, consider {a2, a4, a9, a11}

The Activity Selection Problem

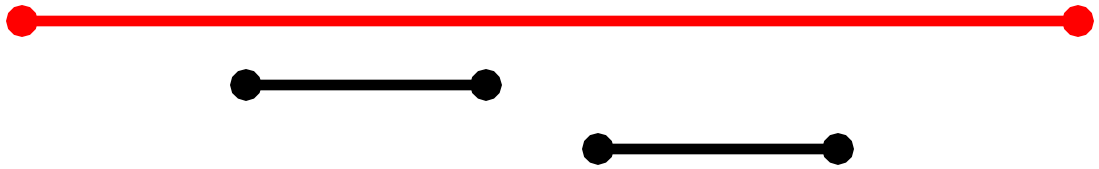
Algorithm 1:

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:

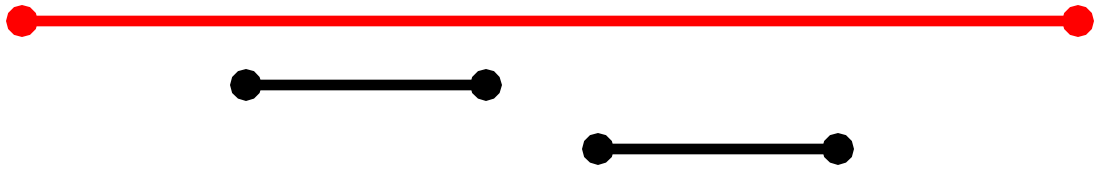
1. sort the activities by the starting time
2. pick the first activity ***a***
3. remove all activities conflicting with ***a***
4. repeat



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

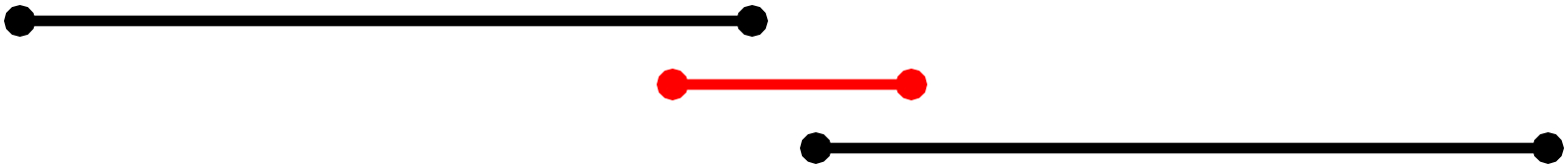
Algorithm 2:

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:

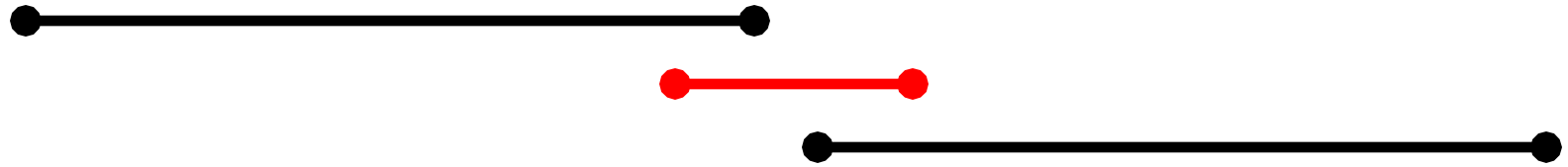
1. sort the activities by length
2. pick the shortest activity ***a***
3. remove all activities conflicting with ***a***
4. repeat



The Activity Selection Problem

Algorithm 2:

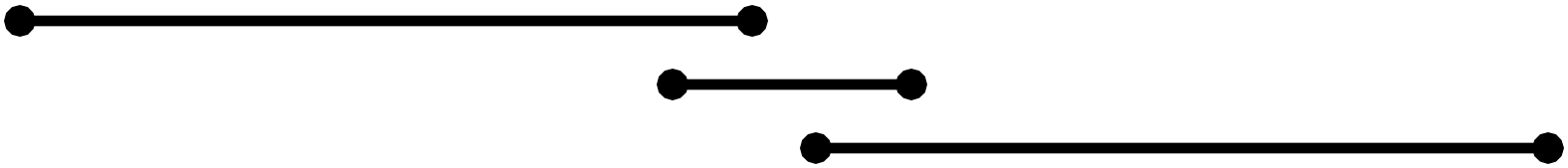
1. sort the activities by length
2. pick the shortest activity ***a***
3. remove all activities conflicting with ***a***
4. repeat



The Activity Selection Problem

Algorithm 3:

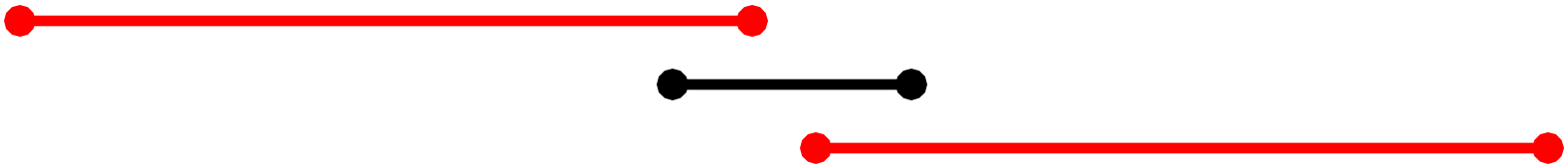
1. sort the activities by ending time
2. pick the activity ***a*** which ends first
3. remove all activities conflicting with ***a***
4. repeat



The Activity Selection Problem

Algorithm 3:

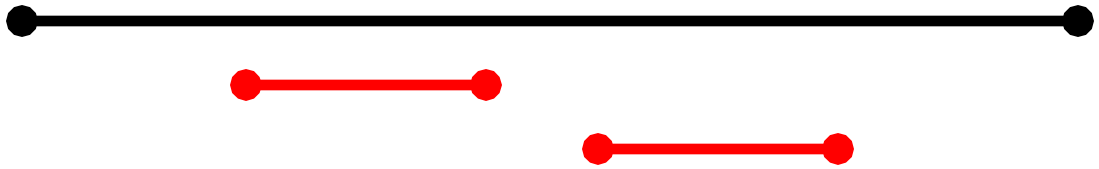
1. sort the activities by ending time
2. pick the activity ***a*** which ends first
3. remove all activities conflicting with ***a***
4. repeat



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



Greedy-Choice Property

Show that there is an optimal solution that begins with a greedy choice (with activity 1, which has the earliest finish time)

Suppose $A \subseteq S$ is an optimal solution

Order the activities in A by finish time. Let the first activity in A is k

If k = 1, the schedule A begins with a greedy choice

If $k \neq 1$, show that there is an optimal solution B to S that begins with the greedy choice, activity 1

Let $B = A - \{k\} \cup \{1\}$

$f[1] \leq f[k]$ => activities in B are disjoint (compatible)

Also, 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

Recursive Greedy Algorithm

s: array of start times

f: array of end times

i, j: indices of the subproblem $S_{i,j}$ to solve where

$S_{i,j} = \{k \in S: f[i] \leq s[k] < f[k] \leq s[j]\}$

RECURSIVE-ACTIVITY-SELECTOR(*s, f, i, j*)//initially: i=0, j=n+1

1. $m = i + 1$
2. **while** $m < j$ **and** $s[m] < f[i]$ //find earliest activity m which is compatible with i
3. $m = m + 1$
4. **if** $m < j$
5. Return $\{m\}$ U RECURSIVE-ACTIVITY-SELECTOR(*s, f, m, j*)
6. **else** return Ø

Running time is $\Theta(n)$

Assuming that the activities have already been sorted by finish times,

the running time of the call RECURSIVE-ACTIVITY-SELECTOR is $\Theta(n)$

Over all recursive calls, each activity is examined exactly once in the while loop test of line 2.

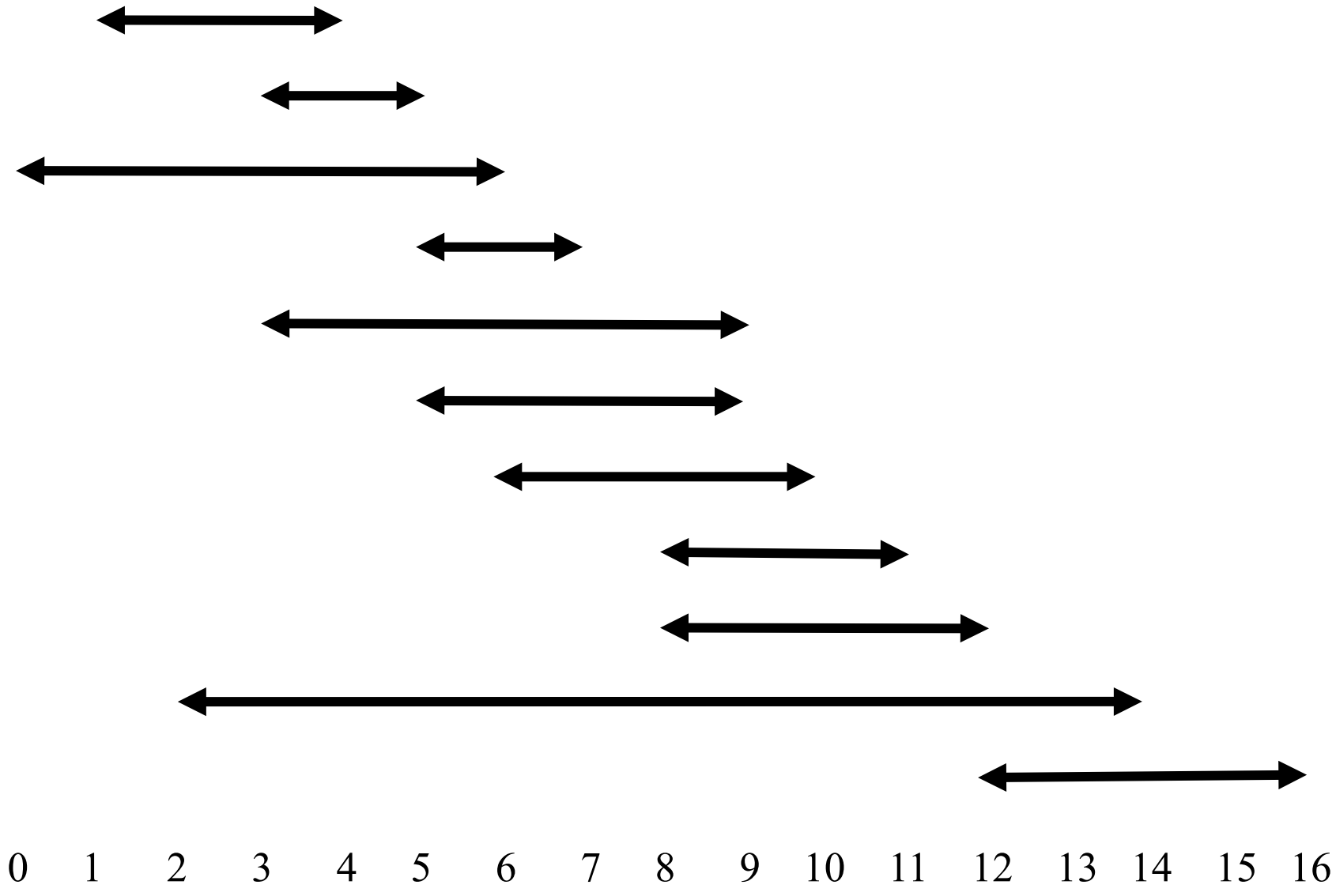
Iterative Greedy Algorithm

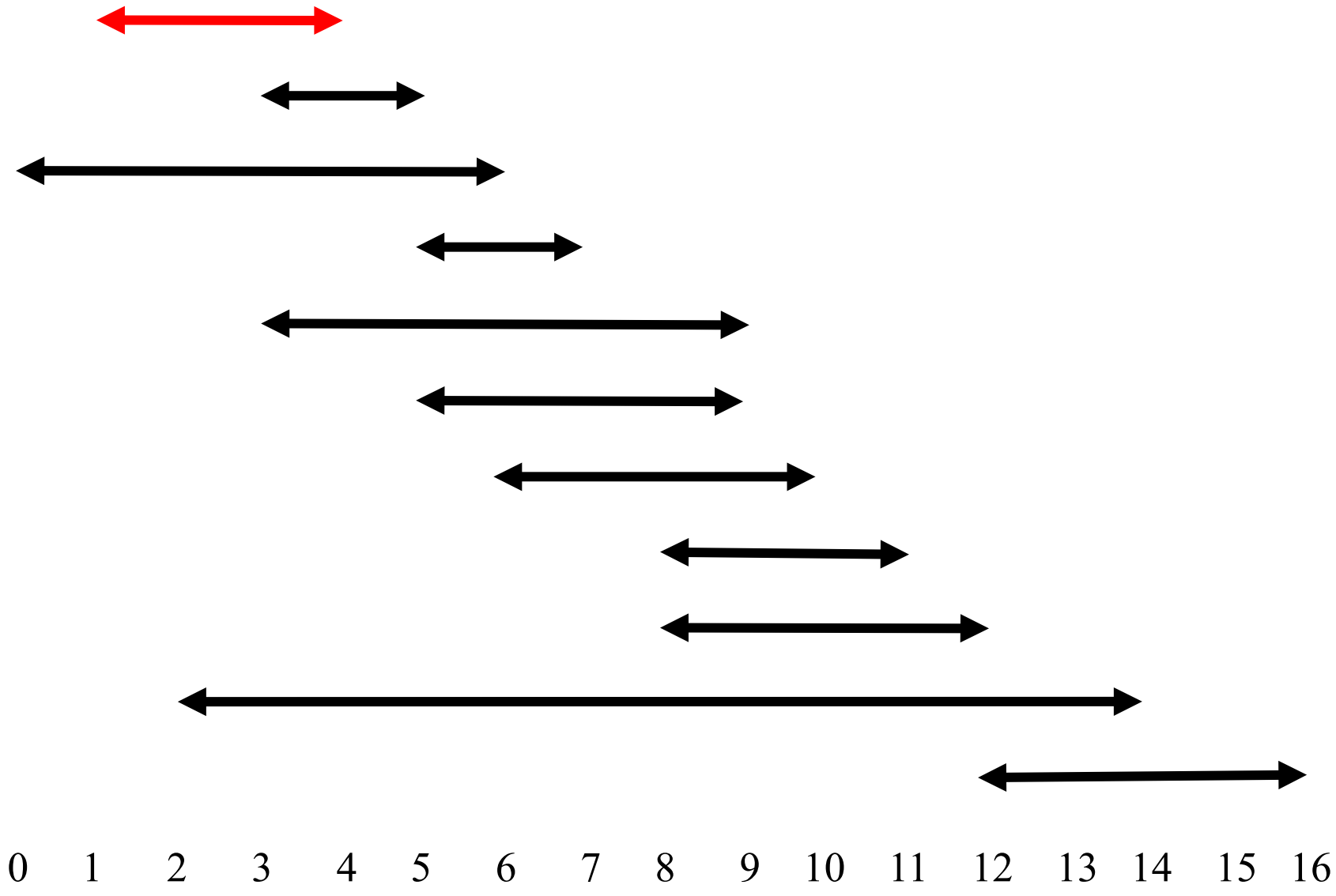
```
GREEDY-ACTIVITY-SELECTOR(s, f)
1.  n = s.length
2.  A = {1}
3.  i = 1
4.  for m = 2 to n
5.        if  s[m] ≥ f[i]  //find earliest activity m which is compatible with i
6.              A = A U {m}
7.              i = m       //f[i] = max{f[k]:k ∈ A}, since activities are sorted
8.  return A
```

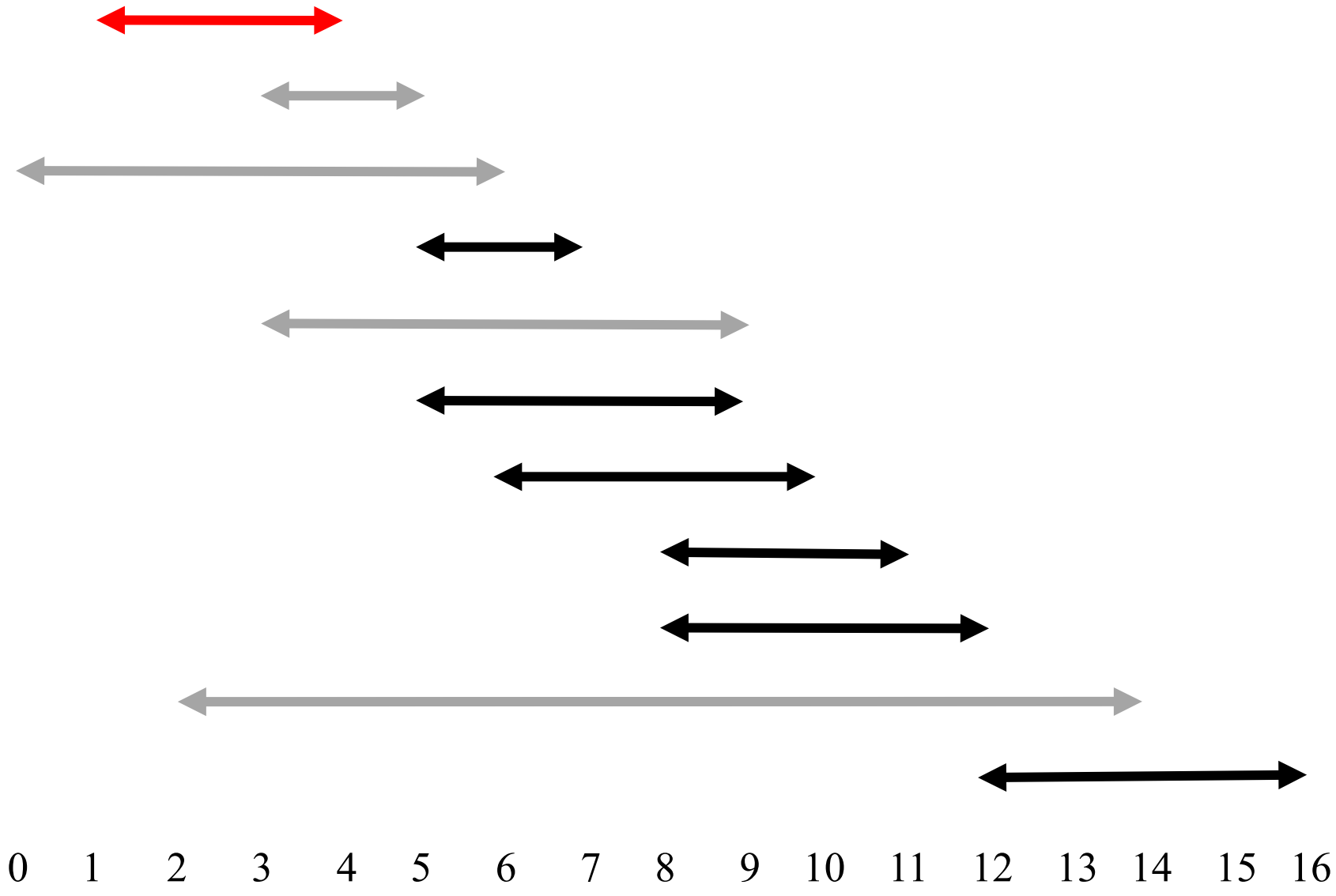
Running time is $\Theta(n)$

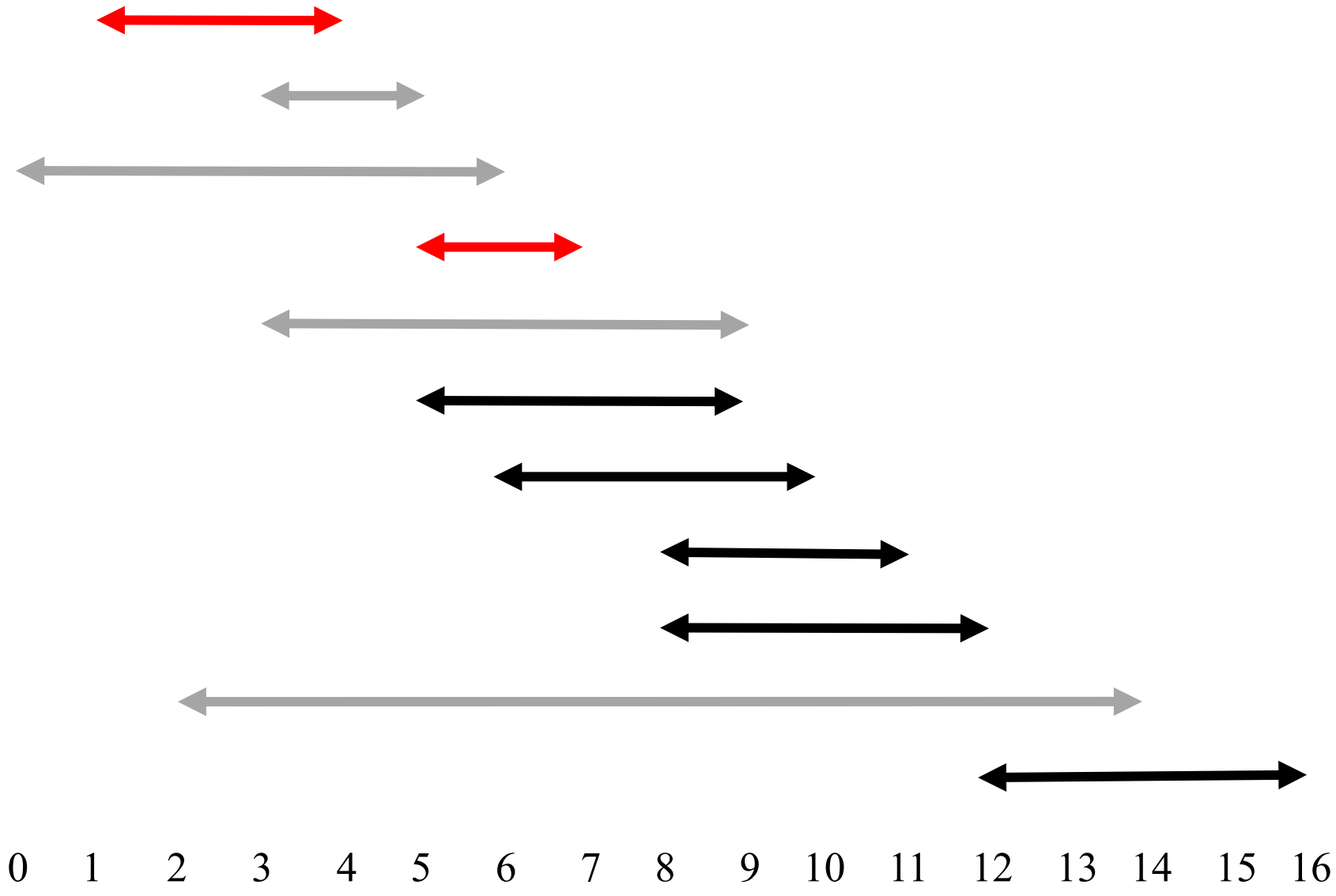
Like the recursive version, GREEDY-ACTIVITY-SELECTOR schedules a set of n activities in $\Theta(n)$ time,

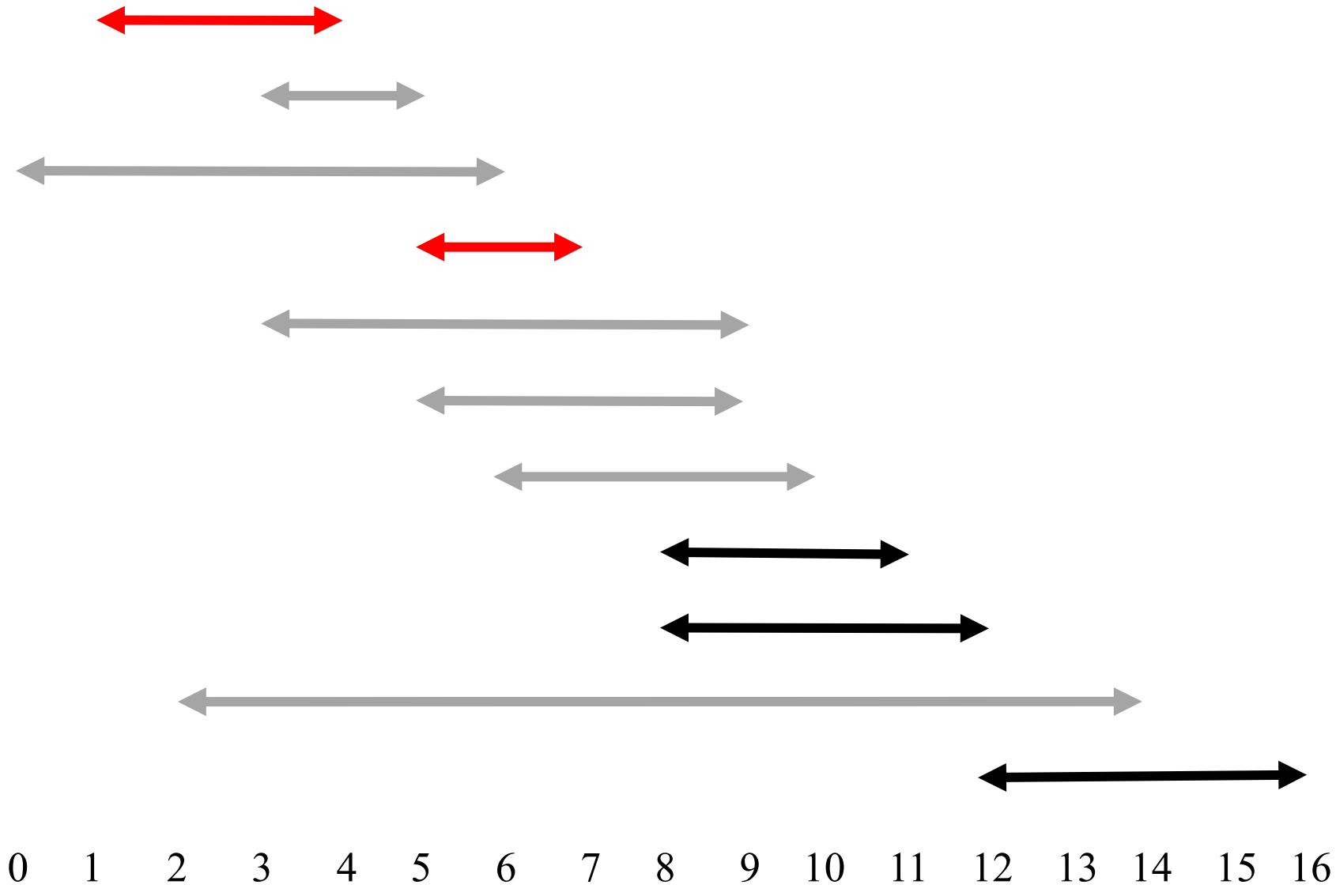
assuming that the activities were already sorted initially by their finish times.

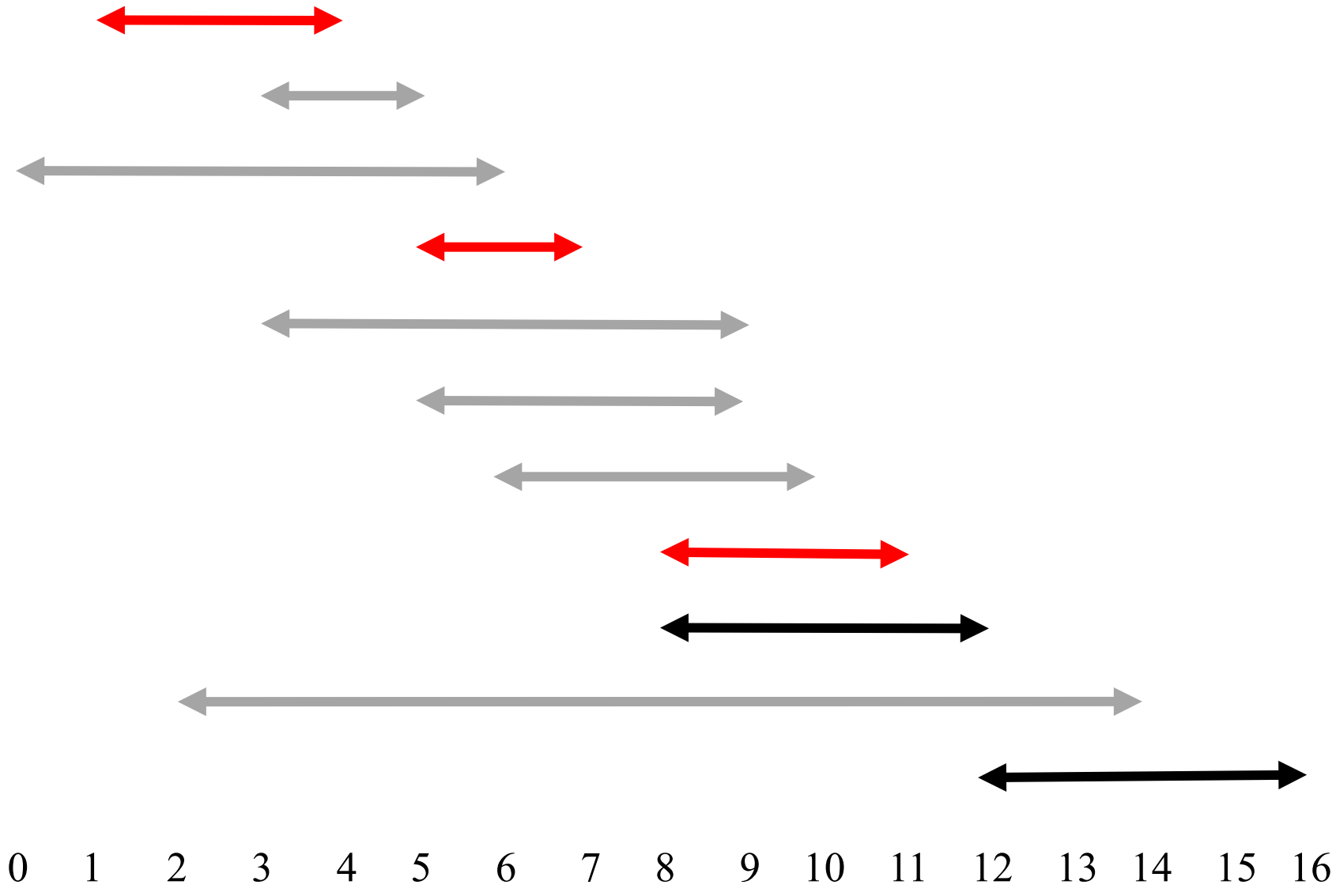

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

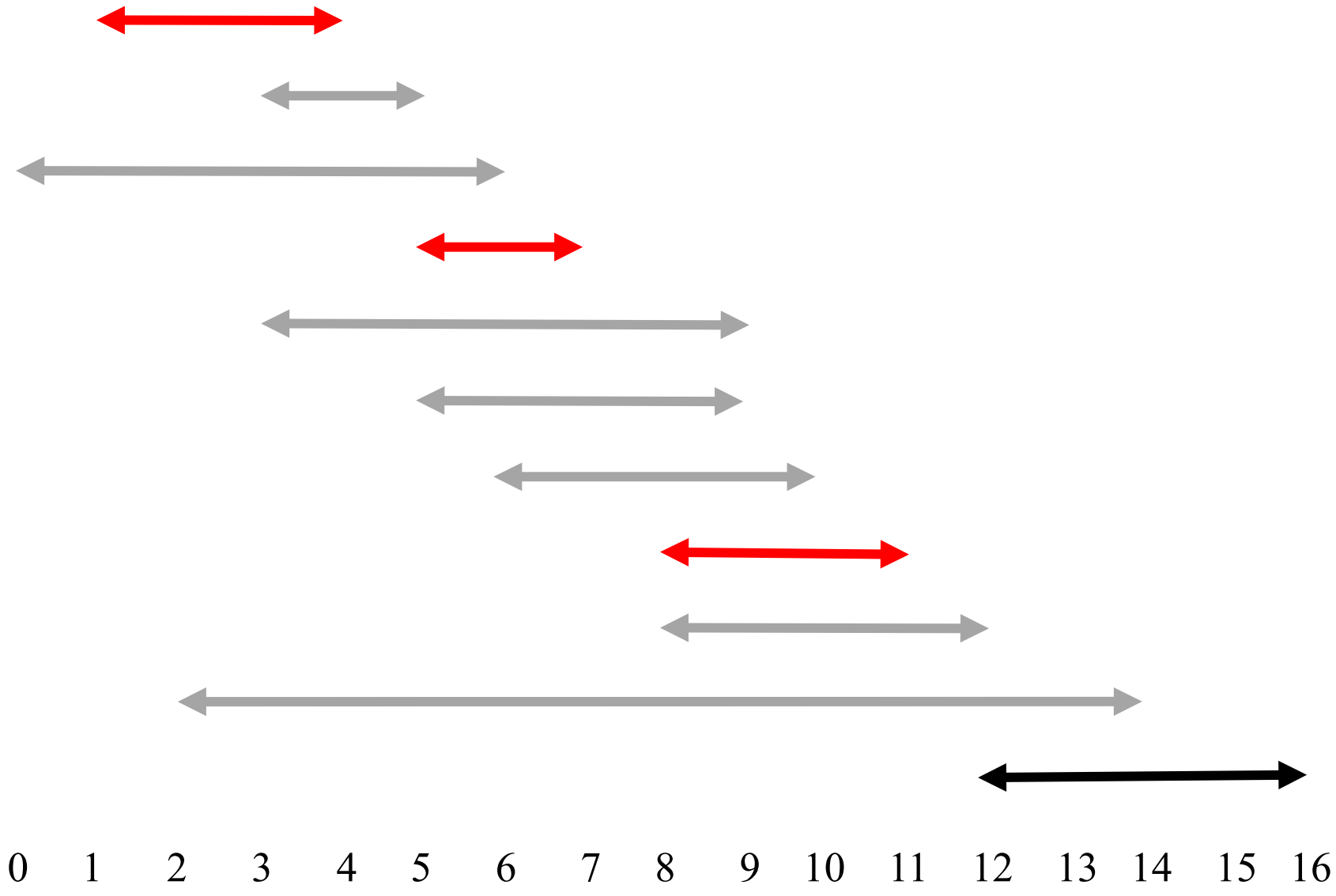

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

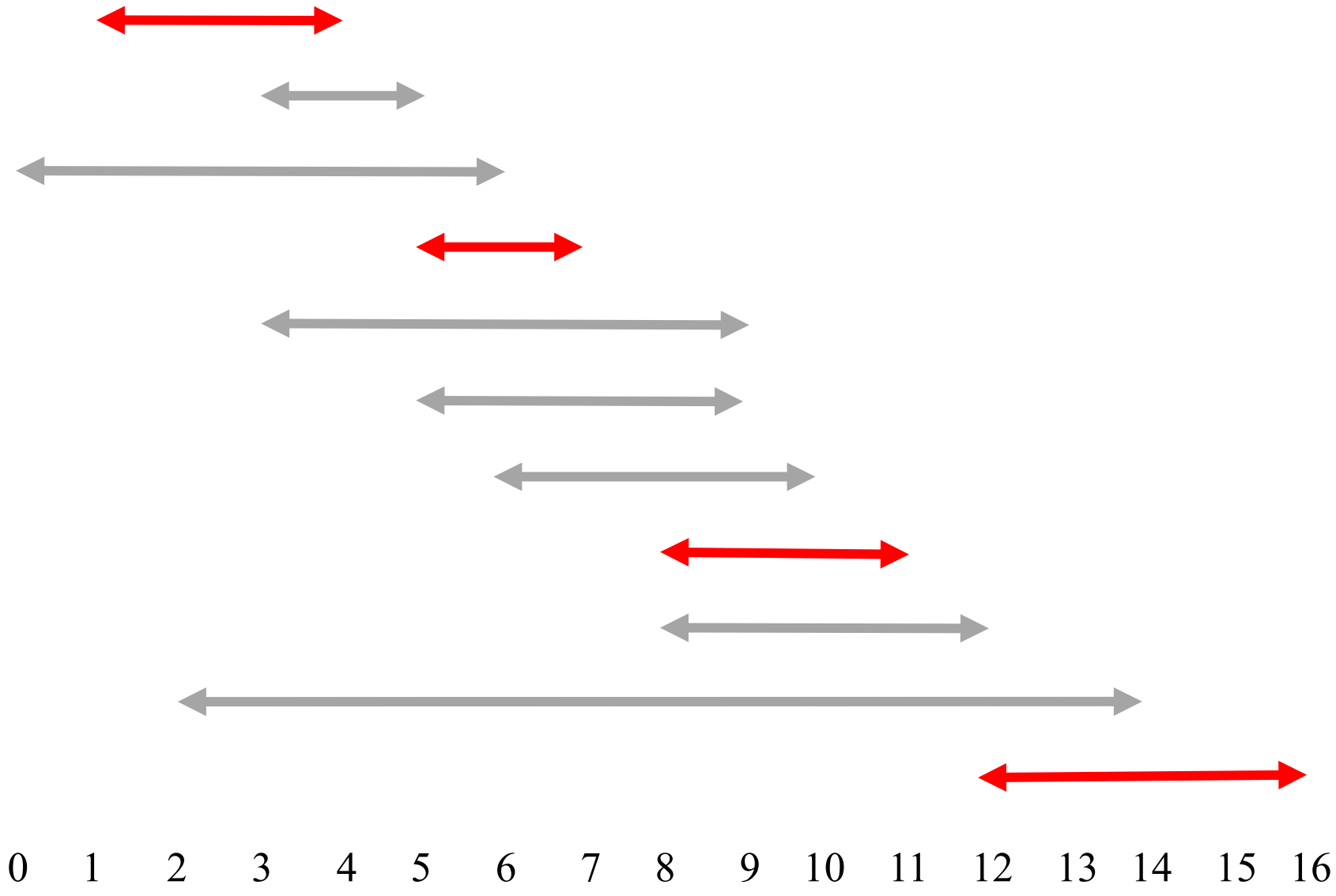

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16


0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16


0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16


0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16


0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16


0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16