EL9343

Data Structure and Algorithm

Lecture 10: Dynamic Programming, Greedy Algorithm

Instructor: Yong Liu

Last Lecture

- Graphs Basics (cont.)
  - DAGs & Topological ordering
  - Strongly connected components
    - Kosaraju's algorithm
    - Tarjan's algorithm

- Introduction to Dynamic Programming

Dynamic Programming

- Dynamic programming: Break up a problem into a series of overlapping subproblems, and build up solutions to larger and larger subproblems.

- An algorithm design technique (like divide and conquer)

- Divide and conquer
  - Partition the problem into independent subproblems
  - Solve the subproblems recursively
  - Combine the solutions to solve the original problem

Dynamic Programming

- Applicable when subproblems are **not** independent
  - Subproblems share sub-subproblems
- A divide and conquer approach would repeatedly solve the common subproblems
- Dynamic programming solves every subproblem just once and stores the answer in a table

Today

- Dynamic Programming (cont.)
  - Rod cutting problem
  - Longest common subsequence
- Introduction to Greedy Algorithm

Rod Cutting

- Solve a simple problem in deciding where to cut steel rods. Serling Enterprises buys long steel rods and cuts them into shorter rods, which it then sells. Each cut is free. The management of Serling Enterprises wants to know the best way to cut up the rods.

Rod Cutting

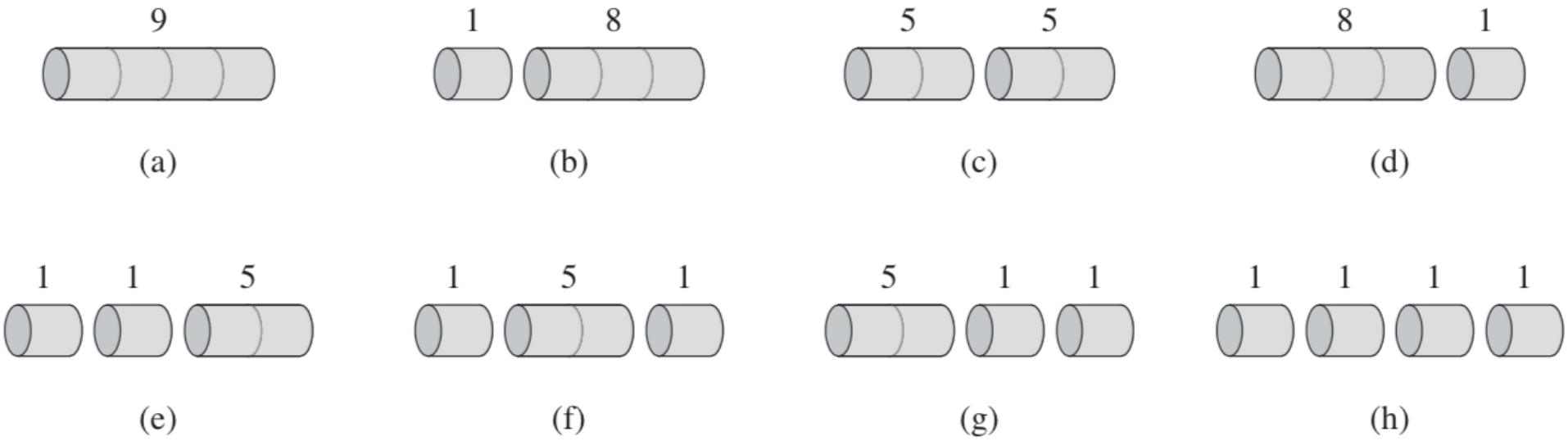
- We assume that we know, for i=1, 2, ..., the price p_i in dollars that Serling Enterprises charges for a rod of length i inches. Rod lengths are always an integral number of inches.

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

- The ***rod-cutting problem*** is the following. Given a rod of length n inches and a table of prices p_i for i=1, 2, ...,n, determine the maximum revenue r_n obtainable by cutting up the rod and selling the pieces. Note that if the price p_n for a rod of length n is large enough, an optimal solution may require no cutting at all.

Rod Cutting

- Consider the case when n=4.



The 8 possible ways of cutting up a rod of length 4. Above each piece is the value of that piece, according to the sample price chart. The optimal strategy is part (c)—cutting the rod into two pieces of length 2—which has total value 10.

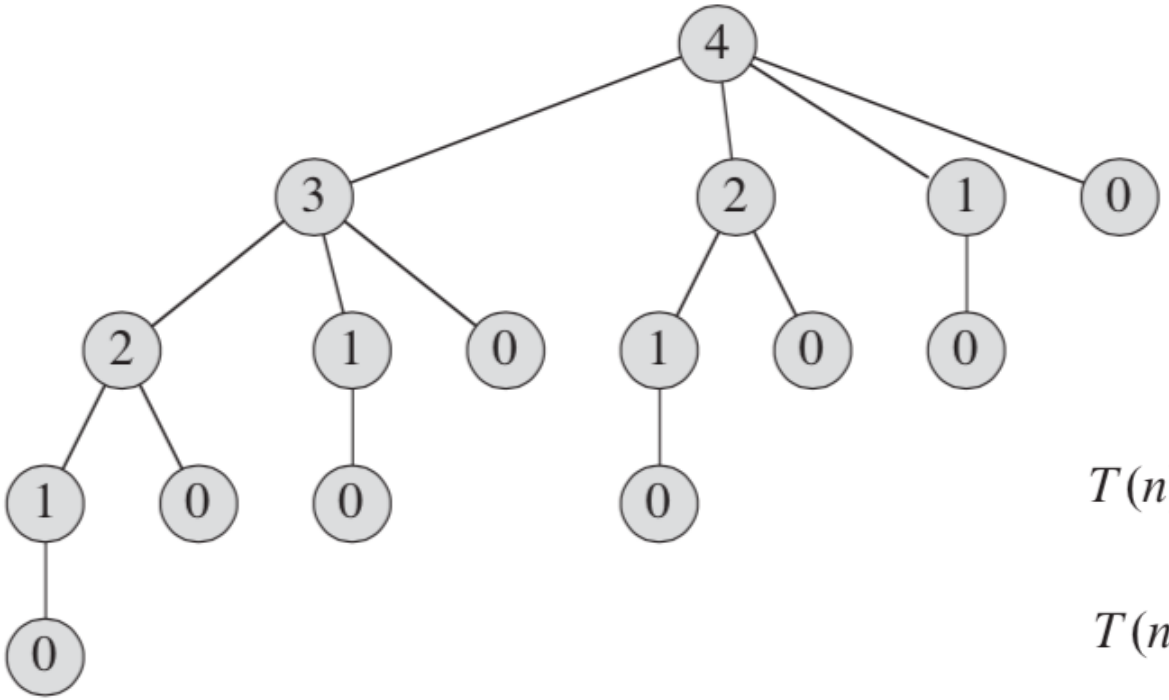
Rod Cutting

- How about more general case n?

Recursive top-down implementation

```
CUT-ROD(p, n)
1  if n == 0
2      return 0
3  q = -∞
4  for i = 1 to n
5      q = max(q, p[i] + CUT-ROD(p, n - i))
6  return q
```

Recursive top-down implementation



$$T(n) = 1 + \sum_{j=0}^{n-1} T(j) .$$

$$T(n) = 2^n$$

The recursion tree showing recursive calls resulting from a call CUT-ROD(p,n) for n=4. Each node label gives the size n of the corresponding subproblem, so that an edge from a parent with label s to a child with label t corresponds to cutting off an initial piece of size s-t and leaving a remaining subproblem of size t. A path from the root to a leaf corresponds to one of the 2^{n-1} ways of cutting up a rod of length n. In general, this recursion tree has 2^n nodes and 2^{n-1} leaves.

Using dynamic programming for optimal rod cutting

- Top-down with memoization

```
MEMOIZED-CUT-ROD(p, n)
1  let r[0..n] be a new array
2  for i = 0 to n
3      r[i] = -∞
4  return MEMOIZED-CUT-ROD-AUX(p, n, r)

MEMOIZED-CUT-ROD-AUX(p, n, r)
1  if r[n] ≥ 0
2      return r[n]
3  if n == 0
4      q = 0
5  else q = -∞
6      for i = 1 to n
7          q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n - i, r))
8  r[n] = q
9  return q
```

Running time $\Theta(n^2)$

Using dynamic programming for optimal rod cutting

- Bottom-up dynamic-programming approach

```
BOTTOM-UP-CUT-ROD(p, n)
1  let r[0..n] be a new array
2  r[0] = 0
3  for j = 1 to n
4      q = -∞
5      for i = 1 to j
6          q = max(q, p[i] + r[j - i])
7      r[j] = q
8  return r[n]
```

Running time $\Theta(n^2)$

Reconstructing a solution

- Above dynamic-programming solutions to the rod-cutting problem return the value of an optimal solution, but they do not return an actual solution: a list of piece sizes.

- We can extend the dynamic-programming approach to record not only the optimal *value* computed for each subproblem, but also a *choice* that led to the optimal value. With this information, we can readily print an optimal solution.

Reconstructing a solution

```
EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
 1  let r[0..n] and s[0..n] be new arrays
 2  r[0] = 0
 3  for j = 1 to n
 4      q = -∞
 5      for i = 1 to j
 6          if q < p[i] + r[j - i]
 7              q = p[i] + r[j - i]
 8              s[j] = i
 9      r[j] = q
10  return r and s
```

```
PRINT-CUT-ROD-SOLUTION(p, n)
1  (r, s) = EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
2  while n > 0
3      print s[n]
4      n = n - s[n]
```

Reconstructing a solution

In this rod-cutting example, the call EXTENDED-BOTTOM-UP-CUT-ROD(p,10) would return the following arrays:

i	0	1	2	3	4	5	6	7	8	9	10
$r[i]$	0	1	5	8	10	13	17	18	22	25	30
$s[i]$	0	1	2	3	2	2	6	1	2	3	10

Problem: Longest Common Subsequence

A subsequence of a string S, is a set of characters that appear in left-to-right order, but not necessarily consecutively.

Example: ACTTGCG

- ACT , ATTC , T , ACTTGC are all subsequences.
- TTA is not a subsequence

A common subsequence of two strings is a subsequence that appears in both strings. A longest common subsequence is a common subsequence of maximal length.

Problem: Longest Common Subsequence

X = ACCG
Y = CCAGCA

Common Subsequence of length 1: = {A} {C} {G}
Common Subsequence of length 2: = {AC} {CC} {AG}
Common Subsequence of length 3: = {CCG}
Common Subsequence of length 4:

Longest Common Subsequence = {CCG}

Has applications in many areas including biology.

Naive Algorithm

Enumerate all subsequences of X, and check if they are subsequences of Y.

Questions:

- How do we implement this?
- How long does it take?

Naive Algorithm

X = ACCGGGTTACCGTTTAAAACCCGGGTAACCT (Size = N)

Y = CCAGGACCAGGGACCGTTTTCCAGCCTTAAACCA (Size = M)

```
for (int i=N; i>0; i--) {
        Find all subsequence of X with length of i;
        Find all subsequence of Y with length of i;
        if ( there is a common subsequence )
                break;
}
```

$$\frac{N!}{i!\,(N-i)!}$$

$$\sum_{i=N}^{1} \frac{N!}{i!(N-i)!} = O(2^N)$$

Dynamic Programming Approach

X = ACCGGGTTACCGTTT AAAACCCGGGTAACCT (Size = N)

Y = CCAGGACCAGGGACCGTTTTCCAGCCTTAAACCA (Size = M)

Define:

C[i][j] -- Length of LCS of sequence X[1..i] and Y[1..j]

Thus: C[i][0] == 0 for all i

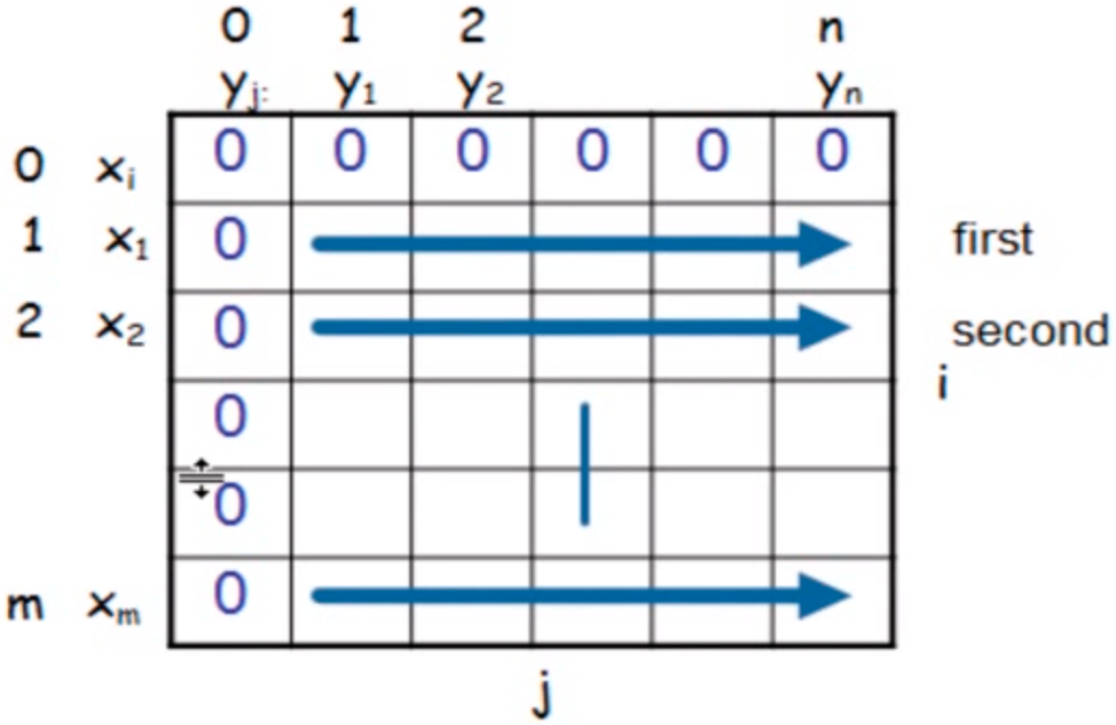
C[0][j] == 0 for all j

Goal: Find C[N][M]

$$c[i,j] = \begin{cases} 0 & \textbf{if } i = 0 \textbf{ or } j = 0 \ , \\ c[i-1, j-1] + 1 & \textbf{if } i, j > 0 \textbf{ and } x_i = y_j \ , \\ \max(c[i, j-1], c[i-1, j]) & \textbf{if } i, j > 0 \textbf{ and } x_i \neq y_j \ . \end{cases}$$

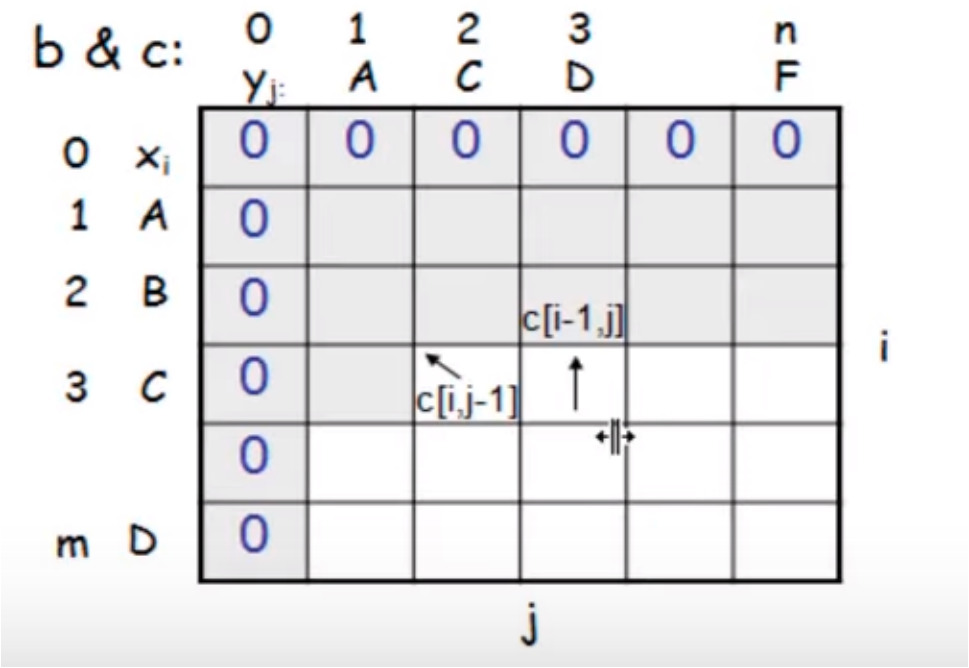
Computing the Length of LCS

$$c[i,j] = \begin{cases} 0 & \textbf{if } i = 0 \textbf{ or } j = 0 \ , \\ c[i-1,j-1]+1 & \textbf{if } i,j > 0 \textbf{ and } x_i = y_j \ , \\ \max(c[i,j-1], c[i-1,j]) & \textbf{if } i,j > 0 \textbf{ and } x_i \neq y_j \ . \end{cases}$$



Additional Information

$$c[i,j] = \begin{cases} 0 & \textbf{if } i = 0 \textbf{ or } j = 0 \ , \\ c[i-1,j-1]+1 & \textbf{if } i,j > 0 \textbf{ and } x_i = y_j \ , \\ \max(c[i,j-1], c[i-1,j]) & \textbf{if } i,j > 0 \textbf{ and } x_i \neq y_j \ . \end{cases}$$



A matrix b[i,j]
For a subproblem [i,j], tell us what choice was made to obtain the optimal value

if xi = yi
    b[i,j]="↖"
else, if c[i-1, j] >= c[i,j-1]
    b[i,j]="↑"
else
    b[i,j]="←"

Example

X=<A,B,C,B,D,A,B>

Y=<B,D,C,A,B,A>

$$c[i,j] = \begin{cases} 0 & \textbf{if } i = 0 \textbf{ or } j = 0 \ , \\ c[i-1,j-1]+1 & \textbf{if } i,j > 0 \textbf{ and } x_i = y_j \ , \\ \max(c[i,j-1], c[i-1,j]) & \textbf{if } i,j > 0 \textbf{ and } x_i \neq y_j \ . \end{cases}$$

A matrix b[i,j]

For a subproblem [i,j], tell us what choice was made to obtain the optimal value

if xi = yi
    b[i,j]="↖"
else, if c[i-1, j] >= c[i,j-1]
    b[i,j]="↑"
else
    b[i,j]="←"

		0	1	2	3	4	5	6
	y_j		B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

Code

```
LCS − Length(X, Y)
 1   m ← length[X]
 2   n ← length[Y]
 3   for i ← 1 to m
 4         do c[i, 0] ← 0
 5   for j ← 0 to n
 6         do c[0, j] ← 0
 7   for i ← 1 to m
 8         do for j ← 1 to n
 9                  do if x_i = y_j
10                        then c[i, j] ← c[i − 1, j − 1] + 1
11                             b[i, j] ← "↖"
12                        else  if c[i − 1, j] ≥ c[i, j − 1]
13                                 then c[i, j] ← c[i − 1, j]
14                                      b[i, j] ← "↑"
15                                 else  c[i, j] ← c[i, j − 1]
16                                      b[i, j] ← "←"
17   return c and b
```

Greedy Algorithm

- A greedy algorithm always makes the choice that looks best at the moment
  - You take the best you can get right now, without regard for future consequences
- Key point: Greed makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution
- Note: Greedy algorithms do not always yield optimal solutions, but for SOME problems they do

When do we use greedy algorithms?

- When we need a heuristic (e.g., hard problems)

- When the problem itself is “greedy”
  - Greedy Choice Property
  - Optimal Substructure Property (shared with DP)

Elements of the Greedy Algorithm

- Greedy-choice property: “A globally optimal solution can be arrived at by making a locally optimal (greedy) choice.”
  - Must prove that a greedy choice at each step yields a globally optimal solution

- Optimal substructure property:“A problem exhibits optimal substructure if an optimal solution to the problem contains within it optimal solutions to subproblems.”
  - This property is a key ingredient of assessing the applicability of greedy algorithm and dynamic programming.”

Elements of the Greedy Algorithm

- Greedy heuristics do not always produce optimal solutions.

- Greedy algorithms vs. dynamic programming (DP)
  - Common: optimal substructure
  - Difference: greedy-choice property
  - DP can be used to find optimal solution even if greedy-choice property does not hold and greedy solutions are not optimal.

An Activity-Selection Problem

Let $S = \{1, 2, \ldots, n\}$ be the set of activities that compete for a resource. Each activity i has its starting time s_i and finish time f_i with $s_i \leq f_i$, namely, if selected, i takes place during time $[s_i, f_i)$. No two activities can share the resource at any time point. We say that activities i and j are compatible if their time periods are disjoint.

- The activity-selection problem is the problem of selecting the largest set of mutually compatible activities.

An Activity-Selection Problem

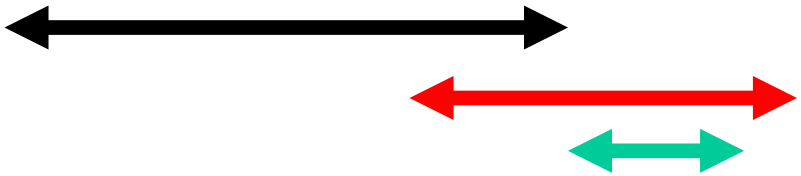
- Here are a set of start and finish times

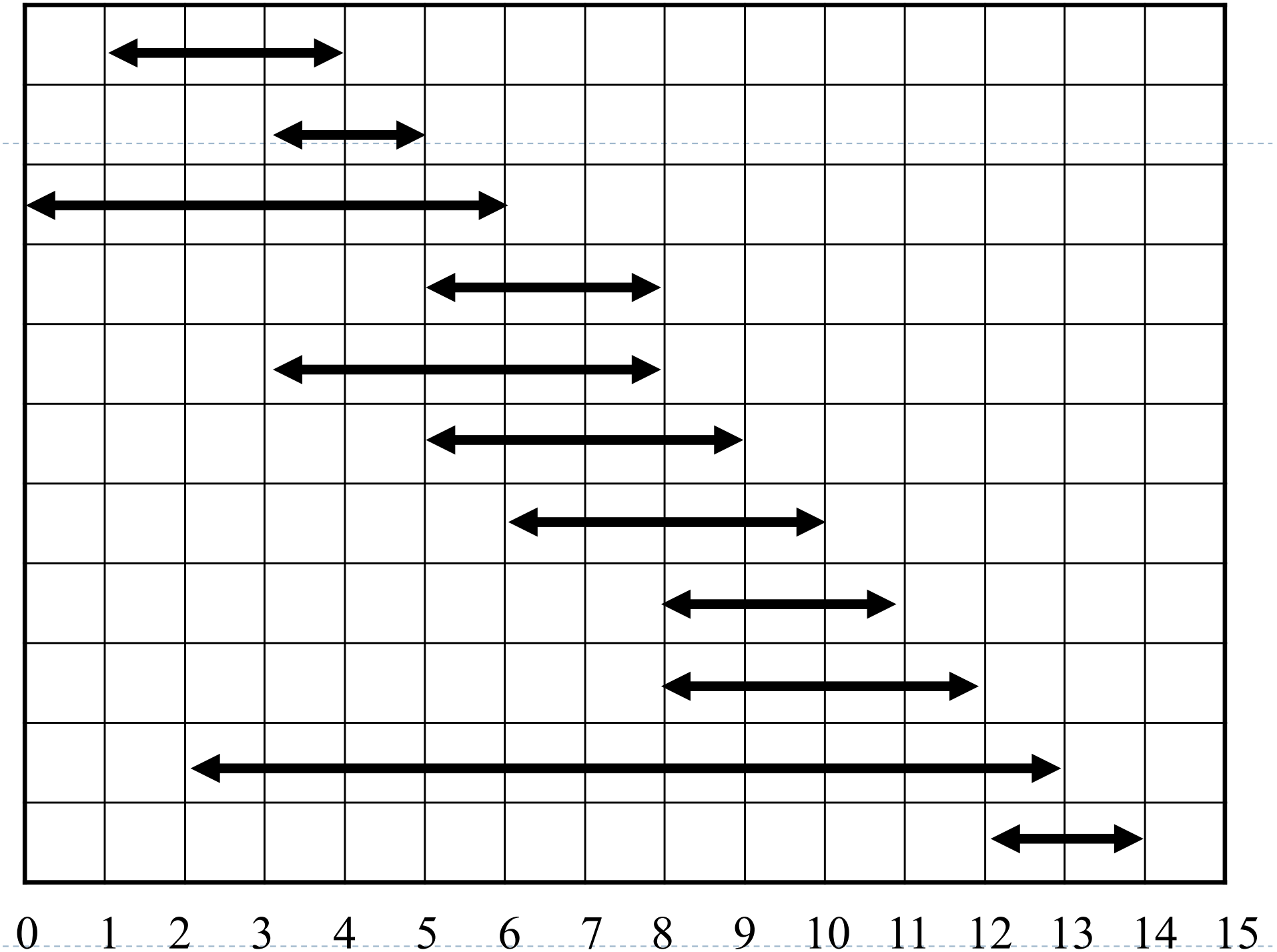
i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

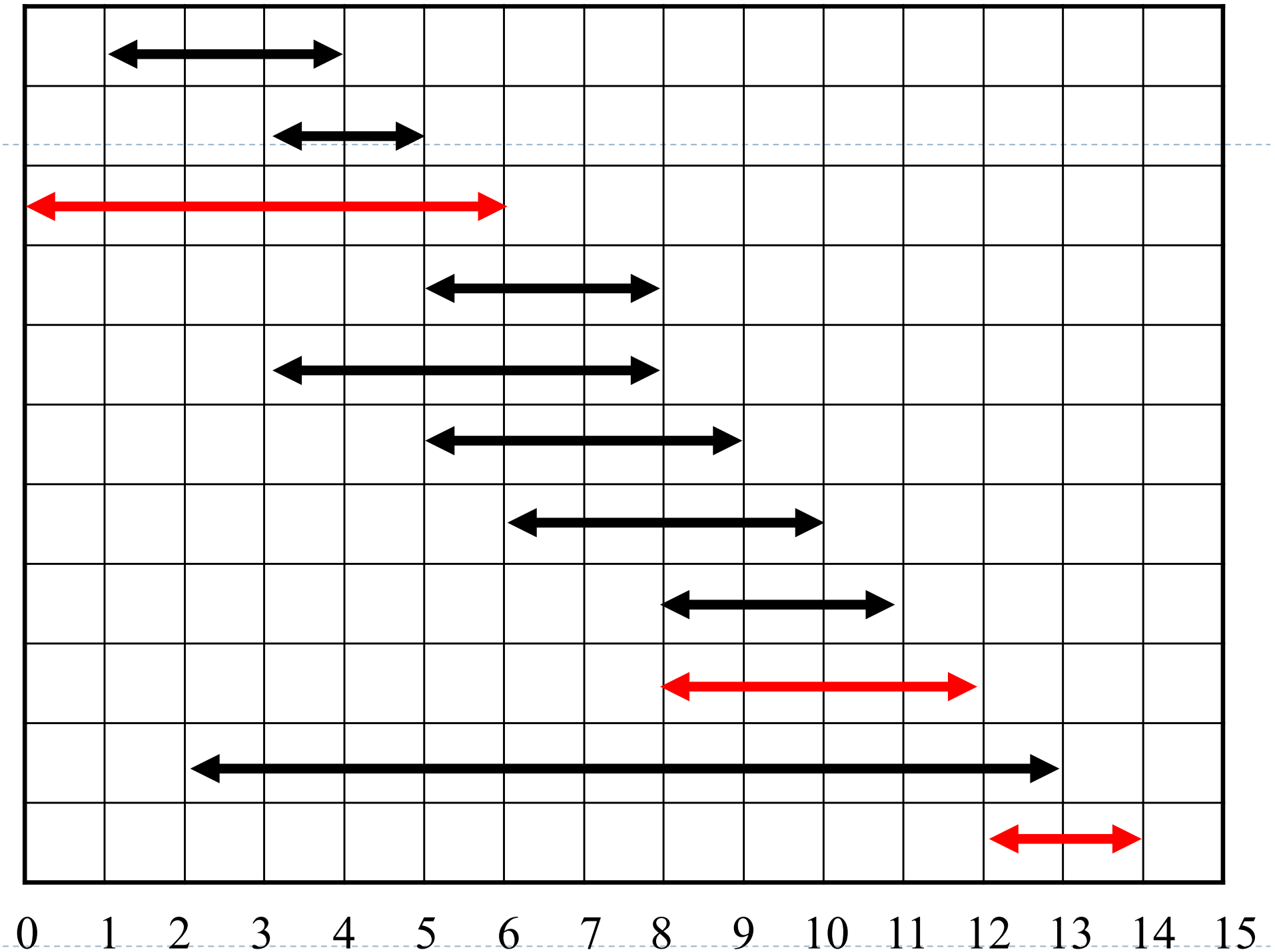
- What is the maximum number of activities that can be completed?
  - $\{a_3, a_9, a_{11}\}$ can be completed
  - But so can $\{a_1, a_4, a_8, a_{11}\}$ which is a larger set
  - But it is not unique, consider $\{a_2, a_4, a_9, a_{11}\}$

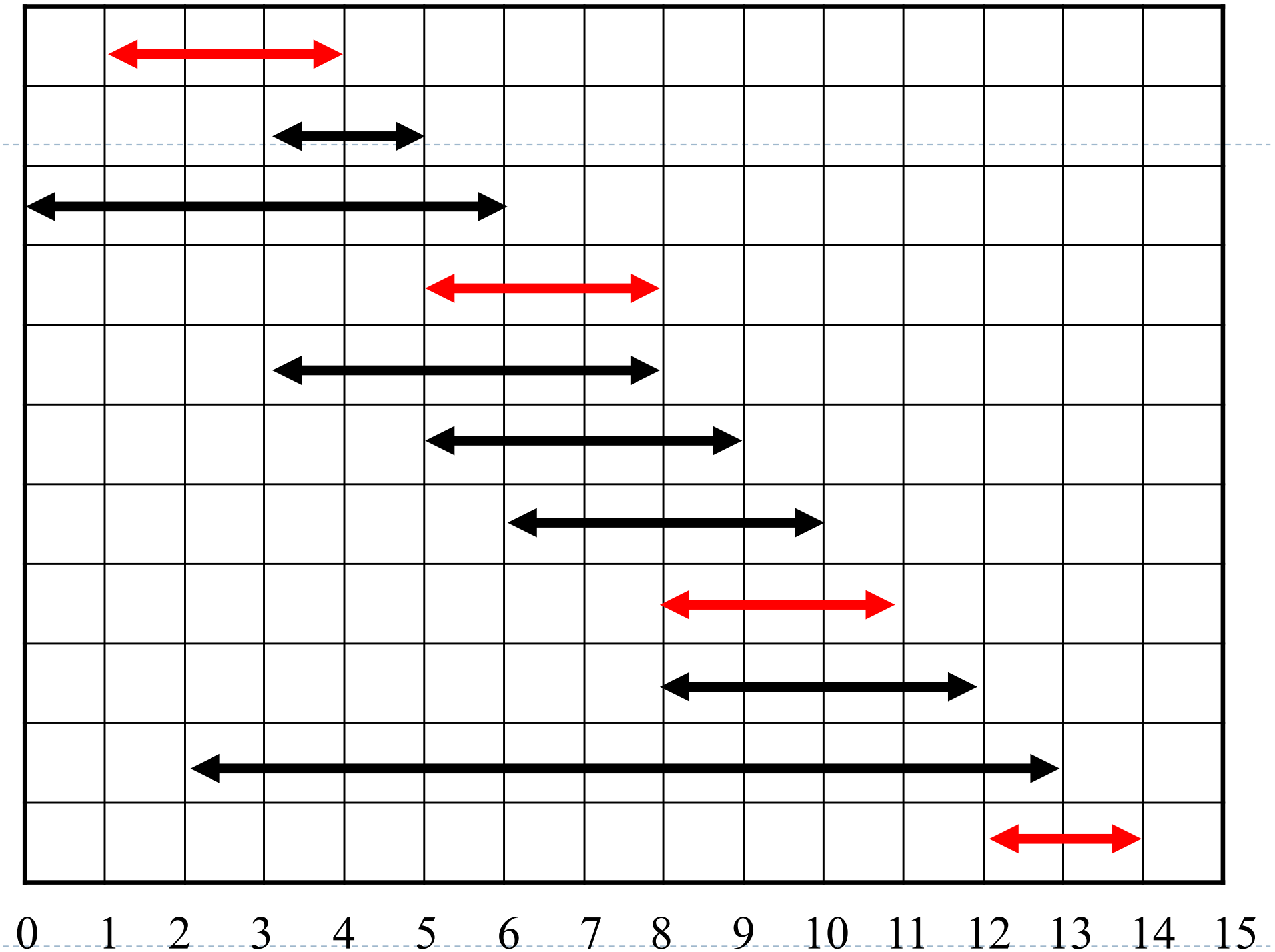
Interval Representation

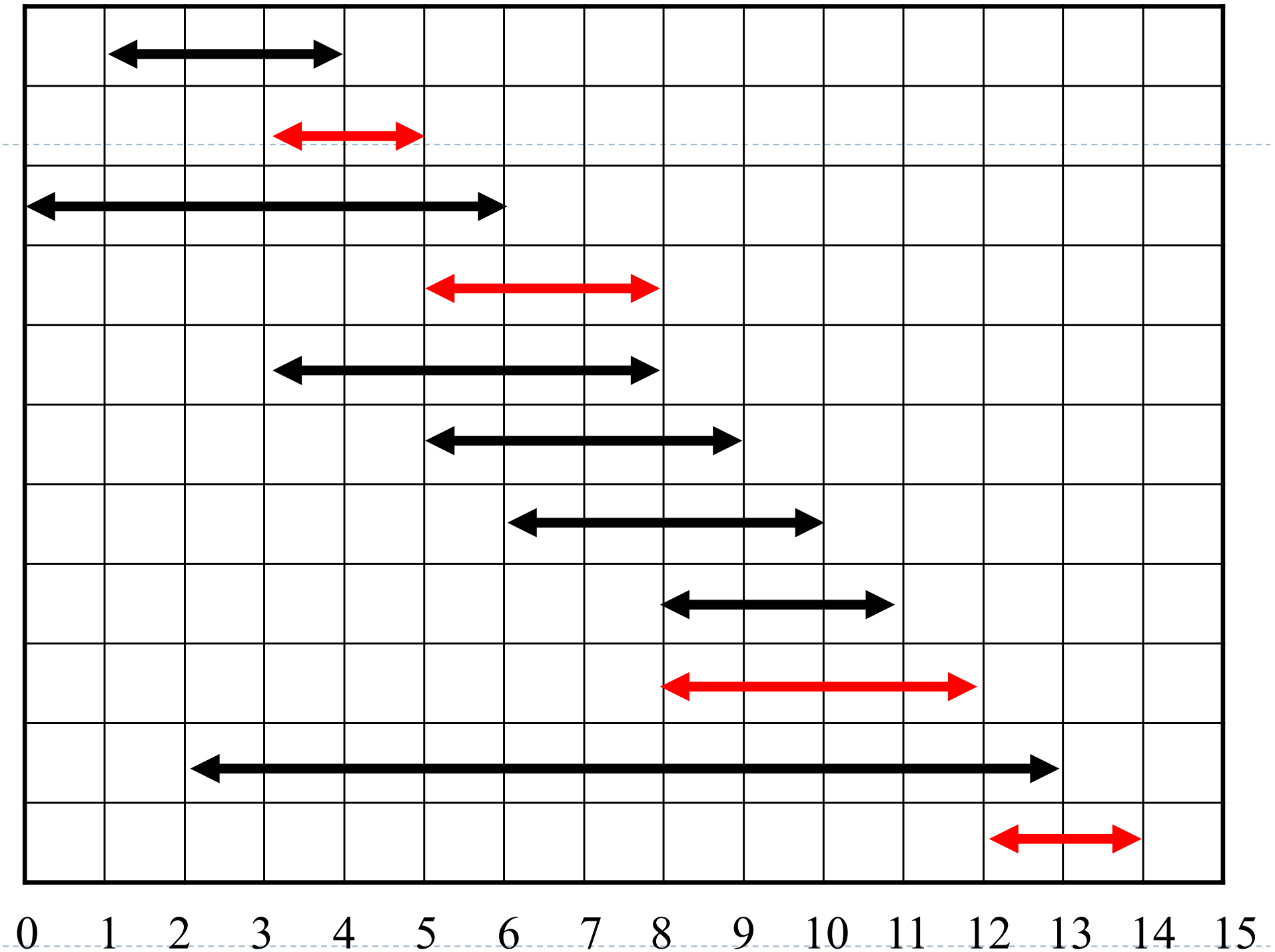
i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14




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

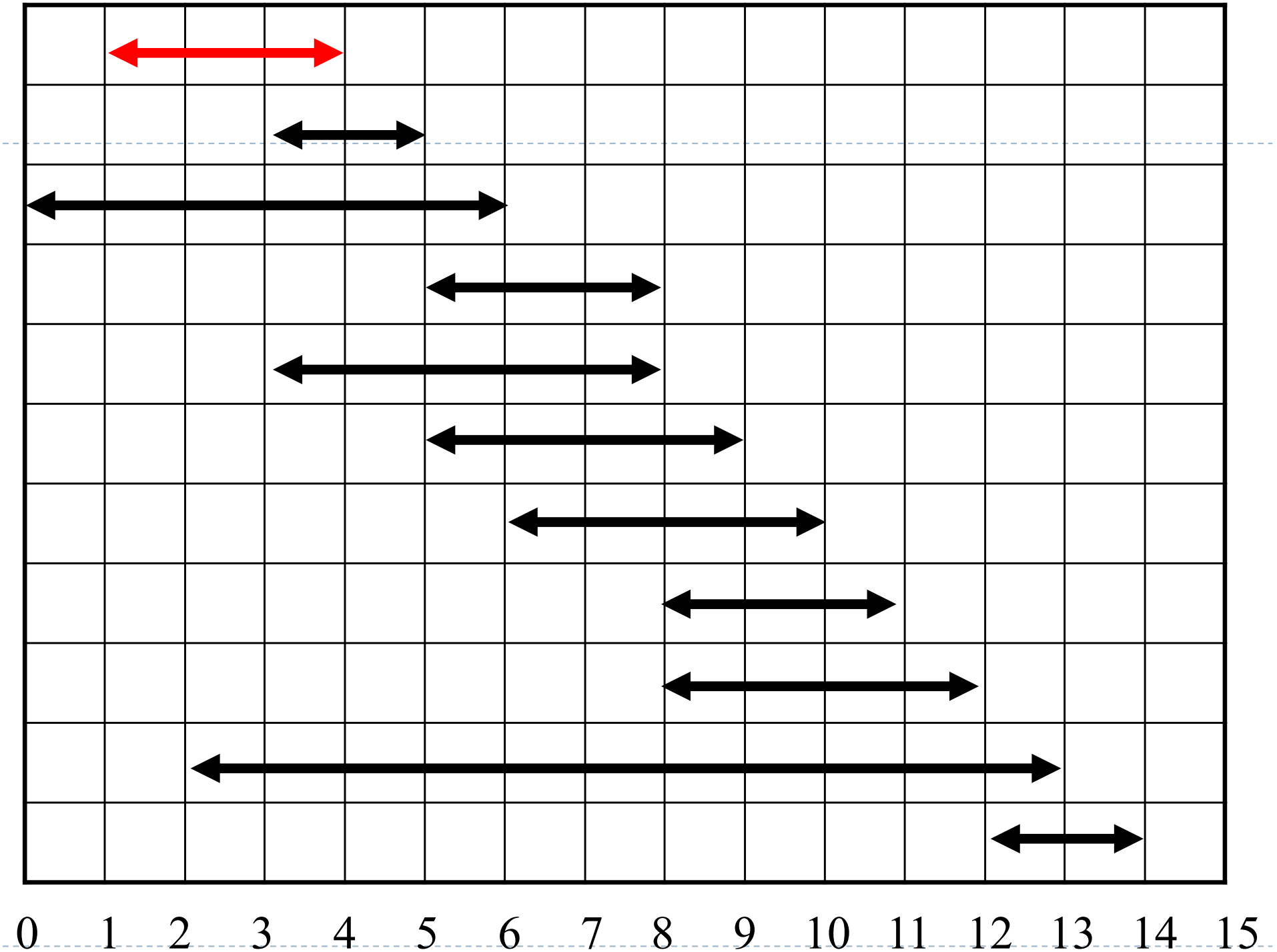

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

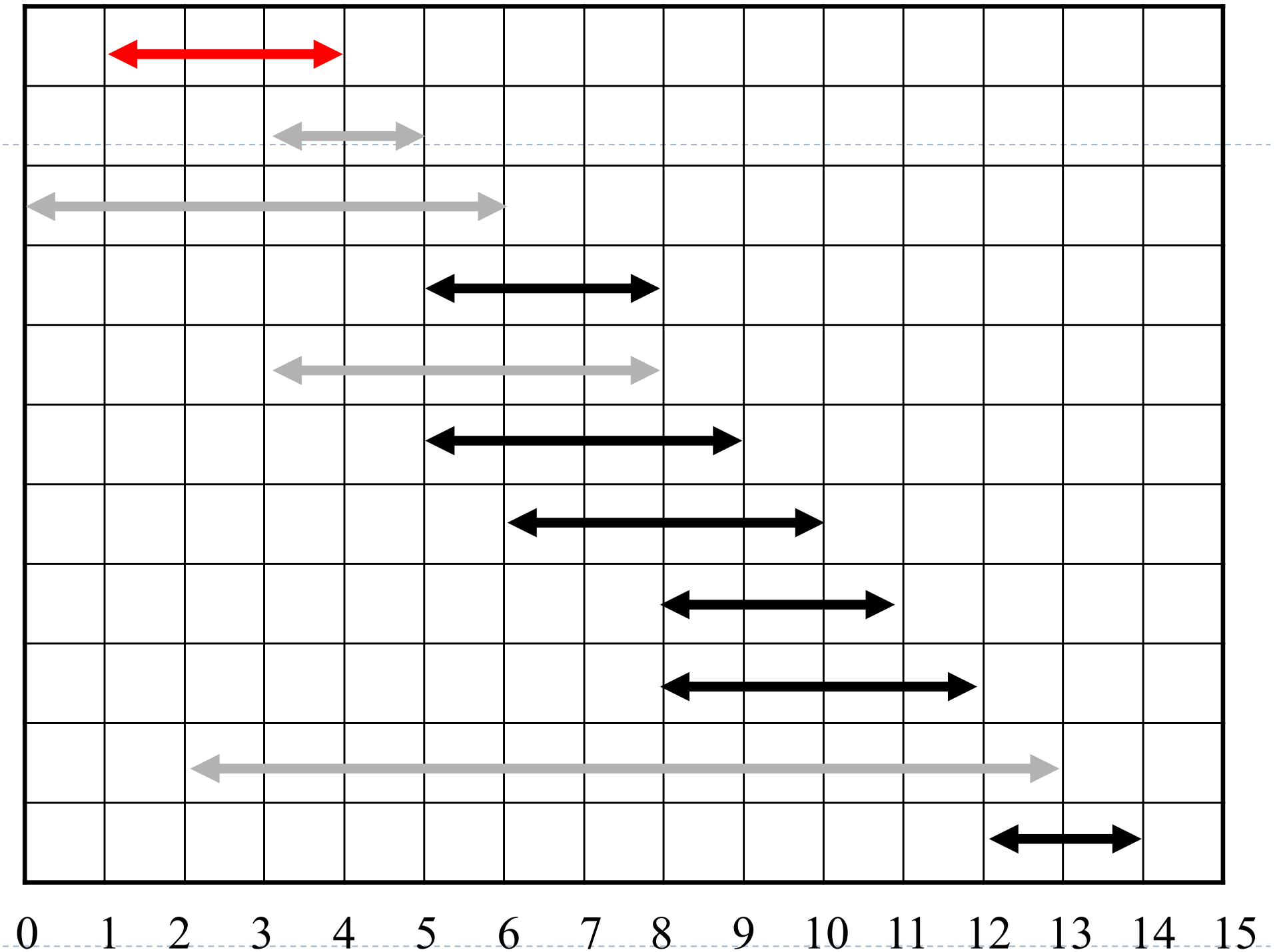

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

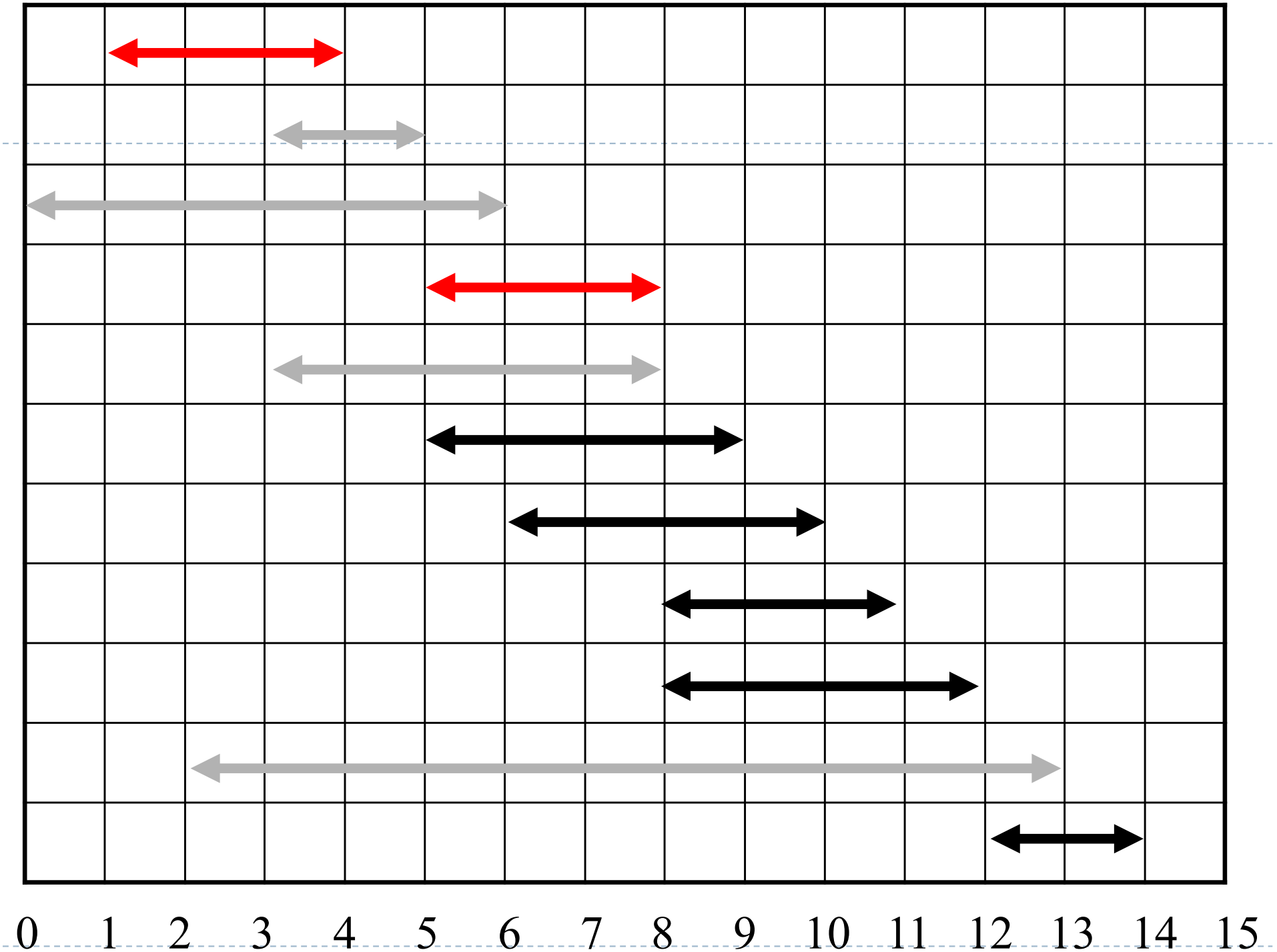

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

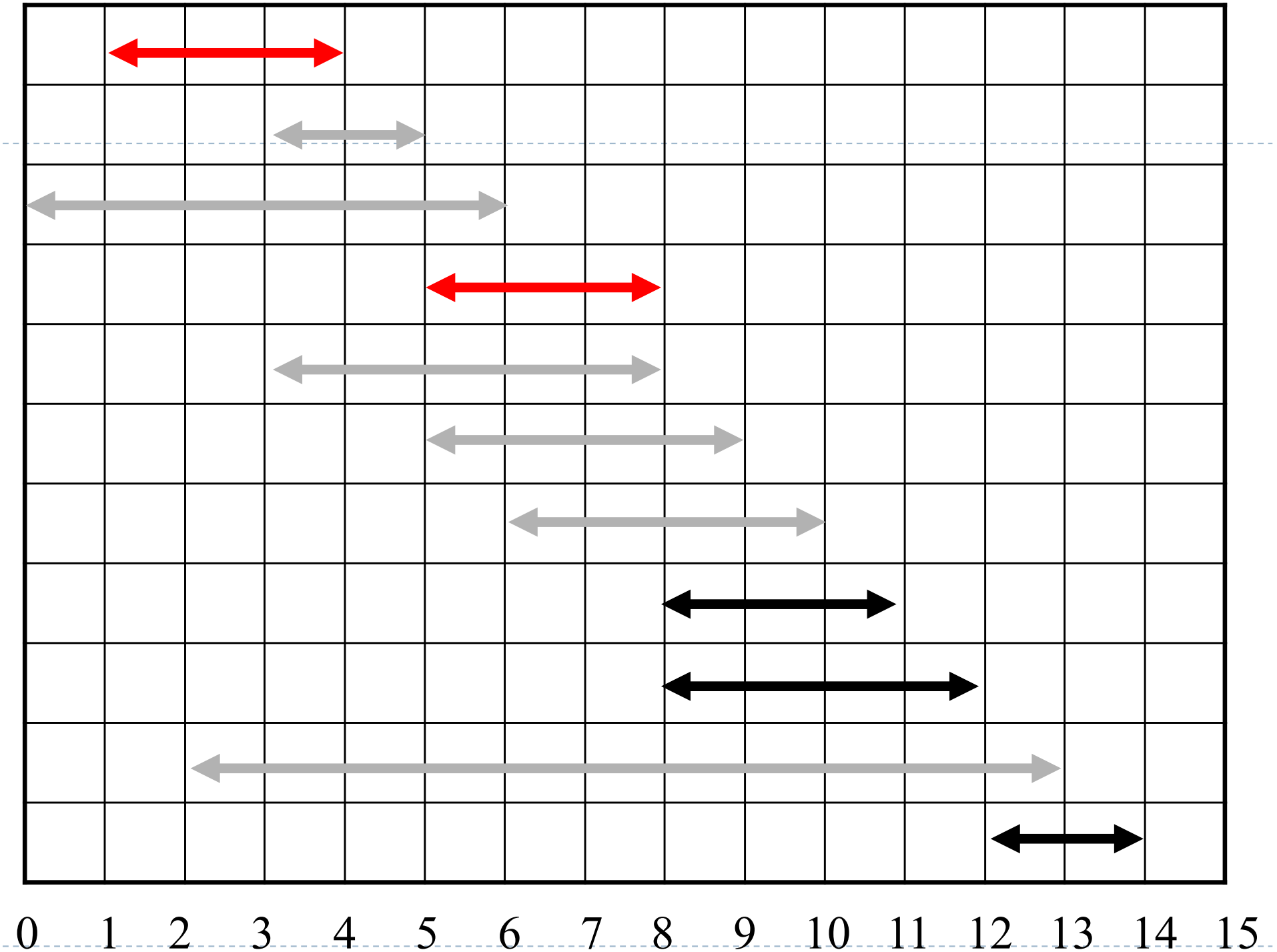
Early Finish Greedy

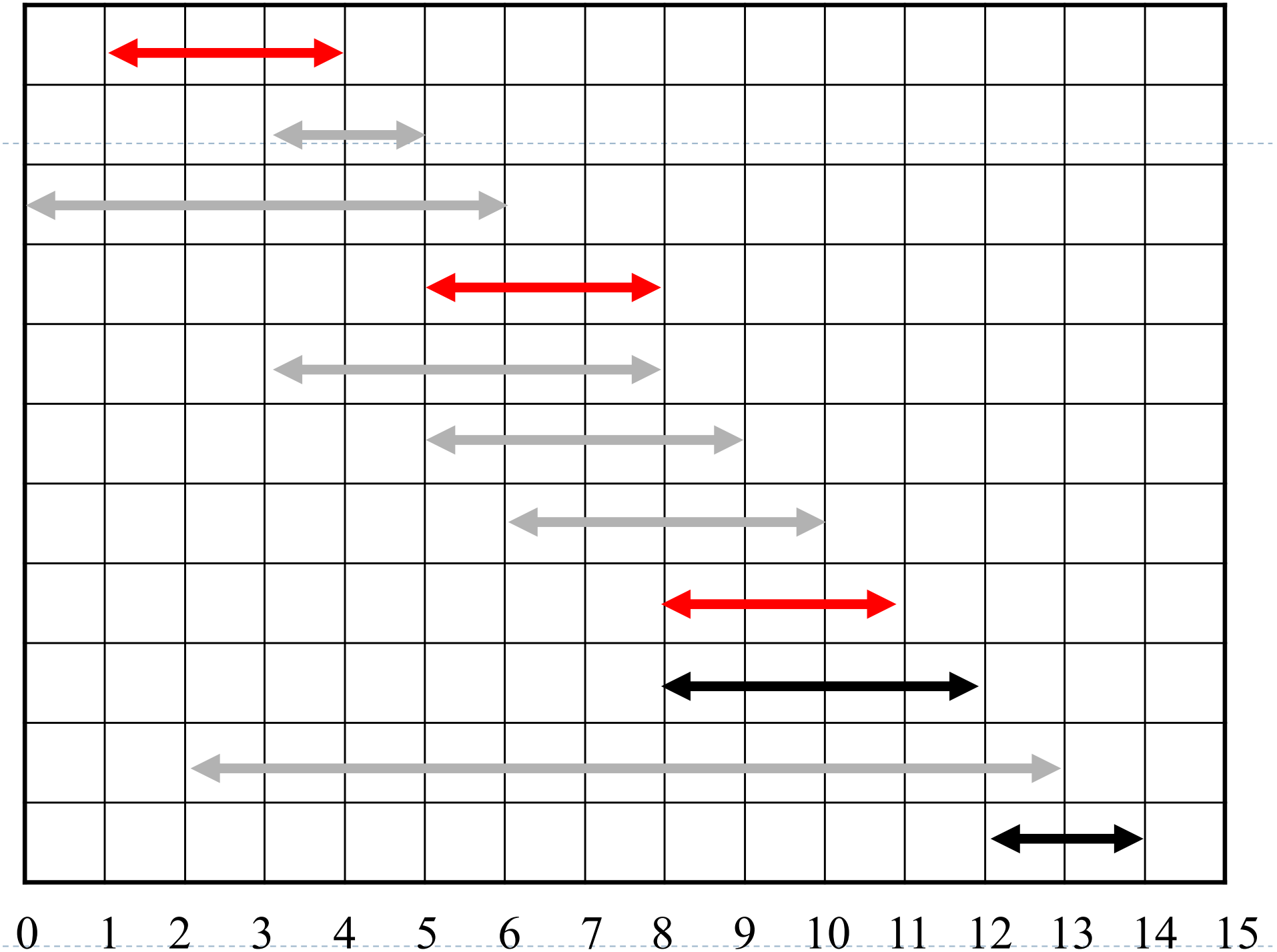
- Select the activity with the earliest finish time
- Eliminate the activities that could not be scheduled
- Repeat!

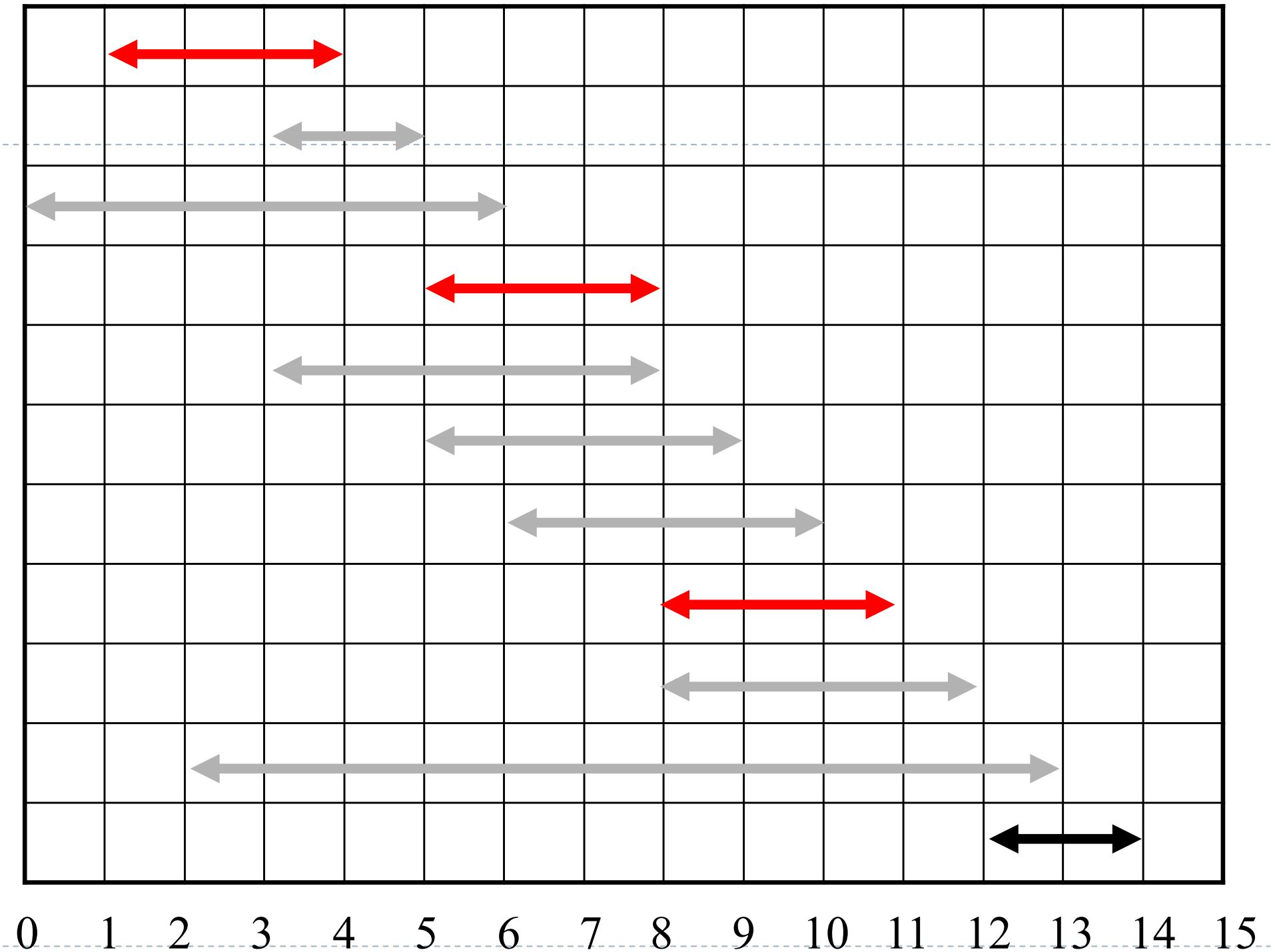

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

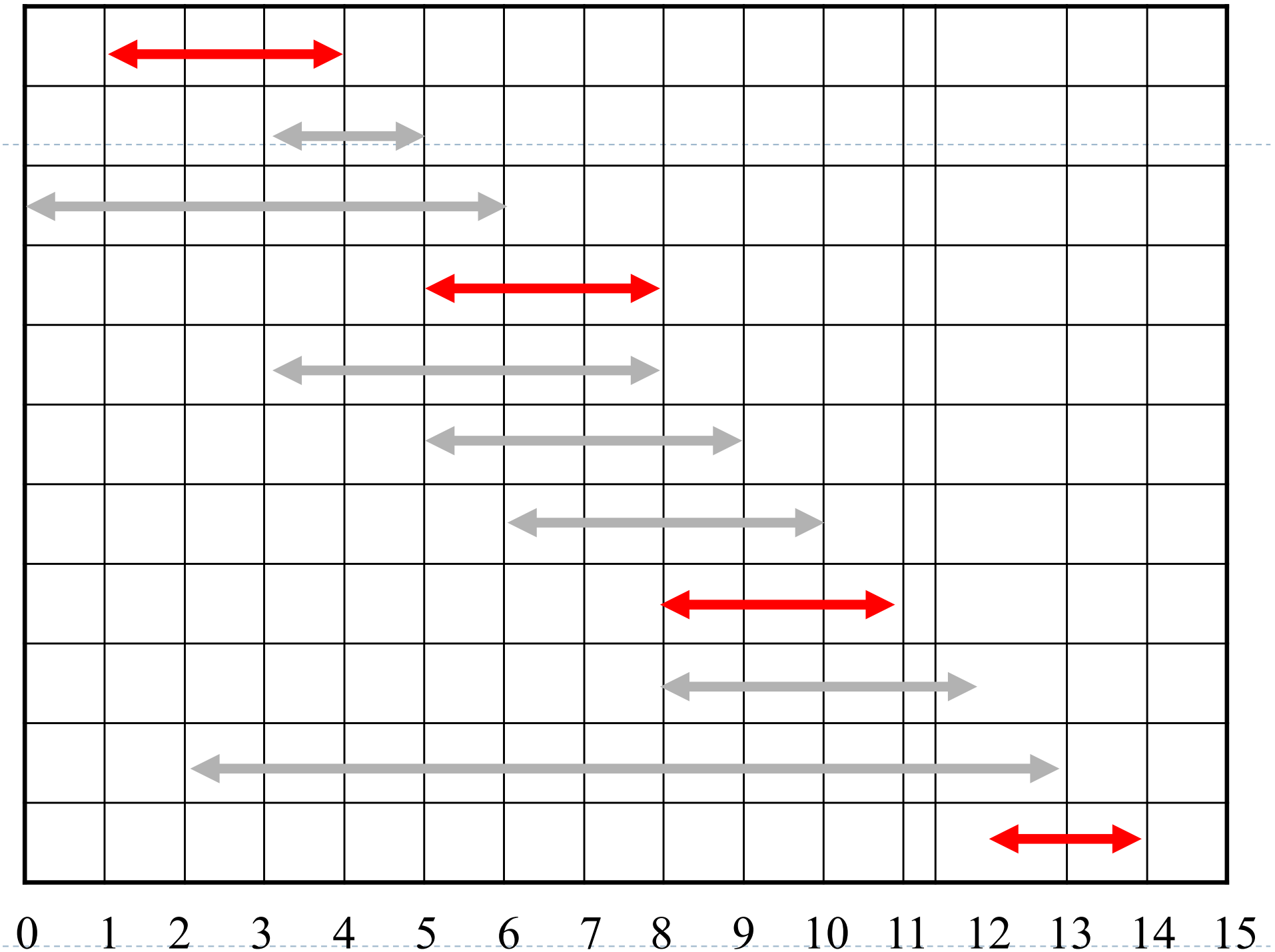

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15


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

Greedy Activity Selection Algorithm

- Specifically
  - Sort the activities by finish time
  - Schedule the first activity
  - Then schedule the next activity in sorted list which starts after previous activity finishes
  - Repeat until no more activities
- Intuition is very simple:
  - Always pick the shortest ride available at the time

Greedy Activity Selection Algorithm

- Assuming activities are sorted by finish time

```
GREEDY-ACTIVITY-SELECTOR(s, f)
1  n ← length[s]
2  A ← {a_1}
3  i ← 1
4  for m ← 2 to n
5        do if s_m ≥ f_i
6              then A ← A ∪ {a_m}
7                   i ← m
8  return A
```

Why it is Greedy?

- Greedy in the sense that it leaves as much opportunity as possible for the remaining activities to be scheduled
- The greedy choice is the one that maximizes the amount of unscheduled time remaining

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 $A \subseteq S$ 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 $k \neq 1$, show that there is an optimal solution B to S that begins with the greedy choice, activity 1
  - Let $B = A - \{k\} \cup \{1\}$
    - $f_1 \leq f_k \rightarrow$ activities in B are disjoint (compatible)
    - B has the same number of activities as A
    - Thus, B is optimal

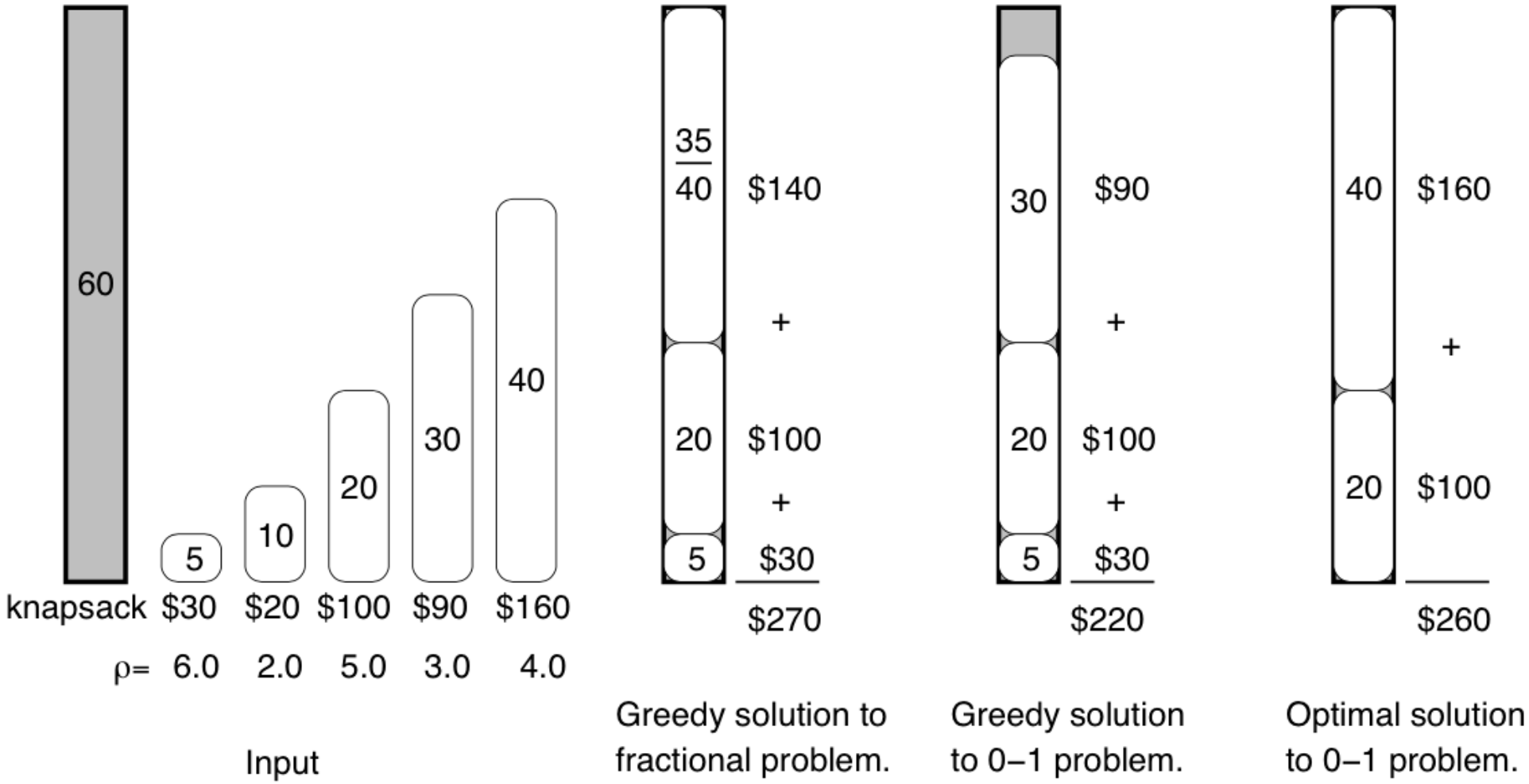
Optimal Substructures

- 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

Knapsack Problem

- **Knapsack Problem:** Given n items, with ith item worth v_i dollars and weighing w_i pounds, a thief wants to take as valuable a load as possible, but can carry at most W pounds in his knapsack.
  - (Fill limited knapsack with most profitable items)
- **The 0-1 knapsack problem**: Each item is either taken or not taken (0-1 decision).
- **The fractional knapsack problem**: Allow to take fraction of items (infinitely divisible).

Example of Knapsack Problem



Fractional vs. 0-1 Knapsack

- Both fractional and 0-1 knapsack have optimal substructure.
  - 0-1: If item *j* is removed from an optimal packing, the remaining packing is an optimal packing with weight at most $W-w_j$ that can be taken from the *n-1* items other than *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 $w_j - w$ of item *j*
- Only fractional knapsack has the greedy choice property
  - Let j be the item with maximum v_i/w_i. Then there exists an optimal solution in which you take as much of item j as possible.

Fractional knapsack: Greedy Choice Property

Proof

- Suppose fpoc, that there exists an optimal solution in you didn't take as much of item j as possible.
- If the knapsack is not full, add some more of item j, and you have a higher value solution. Contradiction
- We thus assume the knapsack is full.
- There must exist some item $k \neq j$ with $\frac{v_k}{w_k} < \frac{v_j}{w_j}$ that is in the knapsack.
- We also must have that not all of j is in the knapsack.
- We can therefore take a piece of k, with ϵ weight, out of the knapsack, and put a piece of j with ϵ weight in.
- This increases the knapsacks value by

$$\epsilon\frac{v_j}{w_j} - \epsilon\frac{v_k}{w_k} = \epsilon\left(\frac{v_j}{w_j} - \epsilon\frac{v_k}{w_k}\right) > 0$$

Contradition to the original solution being optimal.

Fractional knapsack: Greedy Algorithm

Fractional knapsack can be solved by the greedy strategy

- Compute the value per pound v_i/w_i 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

Fractional knapsack: Greedy Algorithm

1. Sort items by vj/wj, renumber.
2. For i = 1 to n
   - Add as much of item i as possible

Complexity of the algorithm

- $O(n \lg n)$ (we need to sort the items by value per pound)

0-1 Knapsack

0-1 knapsack is harder! In fact it is an NP-complete problem (meaning that there probably doesn't exist an efficient solution). It cannot be solved by the greedy strategy.

- Unable to fill the knapsack to capacity, and the empty space lowers the effective value per pound of the packing.
- 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!**

Dynamic Programming for 0-1 Knapsack

Set of n Items: $\mathcal{S}_n = \{\langle v_1, w_1\rangle, \cdots \langle v_i, w_i\rangle \cdots\cdots \langle v_n, w_n\rangle\}$

Let $\mathcal{B}[k, w]$ be the solution of 0-1 knapsack problem over items from 1 to k in $\mathcal{S}_n$ under weight budget of w

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