Design and Analysis of Algorithm

Lecture-8:
Greedy Algorithm

Contents

Recap

- Approach to design algorithm
    - Divide and conquer
    - Greedy
    - Dynamic
    - Backtracking

Greedy Algorithm

An algorithm that at every step selects the best choice available at that time without regard to possible future consequences.

The greedy method is applied to a wide variety of optimization problems where the problem have n inputs and require us to obtain a subset that satisfies some **constraints**.

Any subset that satisfies those constraints is called a feasible solution.

We need to find a feasible solution that either **maximizes** or **minimizes** a given **objective function**.

A **feasible** solution that does this is called an **optimal** solution.

Optimization Problem

An optimization problem is the one where the goal is to find the *best* solution from all feasible solution

These problems appear with an objective function such as:

1. Maximize profit
2. Minimize risk

is the objective function to be minimized

are called inequality

are called equality constraint

$$\begin{aligned} &\underset{x}{\text{minimize}} \\ &\text{subject to} \quad g_i(x) \le 0, \quad i = 1, \dots, m \\ &\qquad\qquad\quad h_j(x) = 0, \quad j = 1, \dots, p \end{aligned}$$

Solution

Solution

Any specifications of values of $x_1, x_2, \ldots \ldots . . x_n$ is called a solution

Feasible Solution

It is a solution for which all the constraints are satisfied.

Optimal Solution

It is a feasible solution that has most favorable value of the objective function (largest for maximize and smallest for minimize)

Solving optimization problem

For most optimization problems we want to find, ***not just a solution***, but the best solution.

A greedy algorithm sometimes works well for optimization problems. It works in phases. At each phase:

- You take the best you can get right now, without regard for future consequences.
- You hope that by choosing a local optimum at each step, you will end up at a global optimum

How Greedy Algorithm works

Suppose you have T time to do the assignment give to you . And assume that you know how much time each assignment will take to complete.

Objective Function

What you want ?

Complete maximum assignment to get better marks

Constraint

What is restricting you from completing all the assignment?

Time limit

This is a simple Greedy-algorithm problem. In each iteration, you have to greedily select the assignment which will take the minimum amount of time to complete.

How Greedy Algorithm works

Let A = {5, 2, 1, 3, 4} and T = 6

$1^{st}\ Choice$

Assignment 3

Assignment completed 1 Time remaining 5

$2^{nd}\ Choice$

Assignment 2

Assignment completed 2 Time remaining 3

$3^{rd}\ Choice$

Assignment 4

Assignment completed 3 Time remaining 0

Greedy Method

```
Algorithm Greedy(a, n)
// a[1 : n] contains the n inputs.
{
    solution := ∅; // Initialize the solution.
    for i := 1 to n do
    {
        x := Select(a);
        if Feasible(solution, x) then
            solution := Union(solution, x);
    }
    return solution;
}
```

Characteristics of greedy algorithm

Greedy algorithm makes best choice at each step of the algorithm

The choice made by a greedy algorithm depend on choices made so far

Greedy algorithm do not reconsider the decision take at previous step.

Greedy algorithms can be characterized as being 'short sighted', and also as 'non-recoverable'. They are ideal only for problems which have 'optimal substructure'

Applications of greedy

- Activity Selection Problem
- Job sequencing with deadline
- Knapsack Problem
- Minimum cost spanning tree
- Huffman coding
- Shortest path

Activity Selection Problem

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

- Act[ ]: array containing all the activities.
- S[ ]: array containing the starting time of all the activities.
- F[ ]: array containing the finishing time of all the activities.

Output: a maximum-size subset of mutually compatible activities

Example of activity selection problem with given start and end time

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 $\{a_2, a_4, a_{9'} a_{11}\}$

Activity Selection

To join the activity A_k,

- you must join at s_k;
- you must also stay until f_k

Since we want as many activities as possible, should we choose the one with

- Shortest duration
- Earliest start time
- Earliest end time

Shortest Duration

Shortest duration time may not be good:

A1 : [4:50, 5:10), A2 : [3:00, 5:00), A3 : [5:05, 7:00),

No. of activities in this solution R (shortest duration first) is A_1

One activity in R clashes with at most two activities.

Earliest Start

Earliest start time may even be worse:

A1 : [2:00, 10:00), A2 : [2:10, 3:20), A3 : [3:20, 3:30), A4 : [3:30, 3:40), A5 : [3:40, 3:50)

No. of activities in the solution of shortest duration first are A_3 A_4 A_5

No. of activities in the solution of earliest start first are A_1

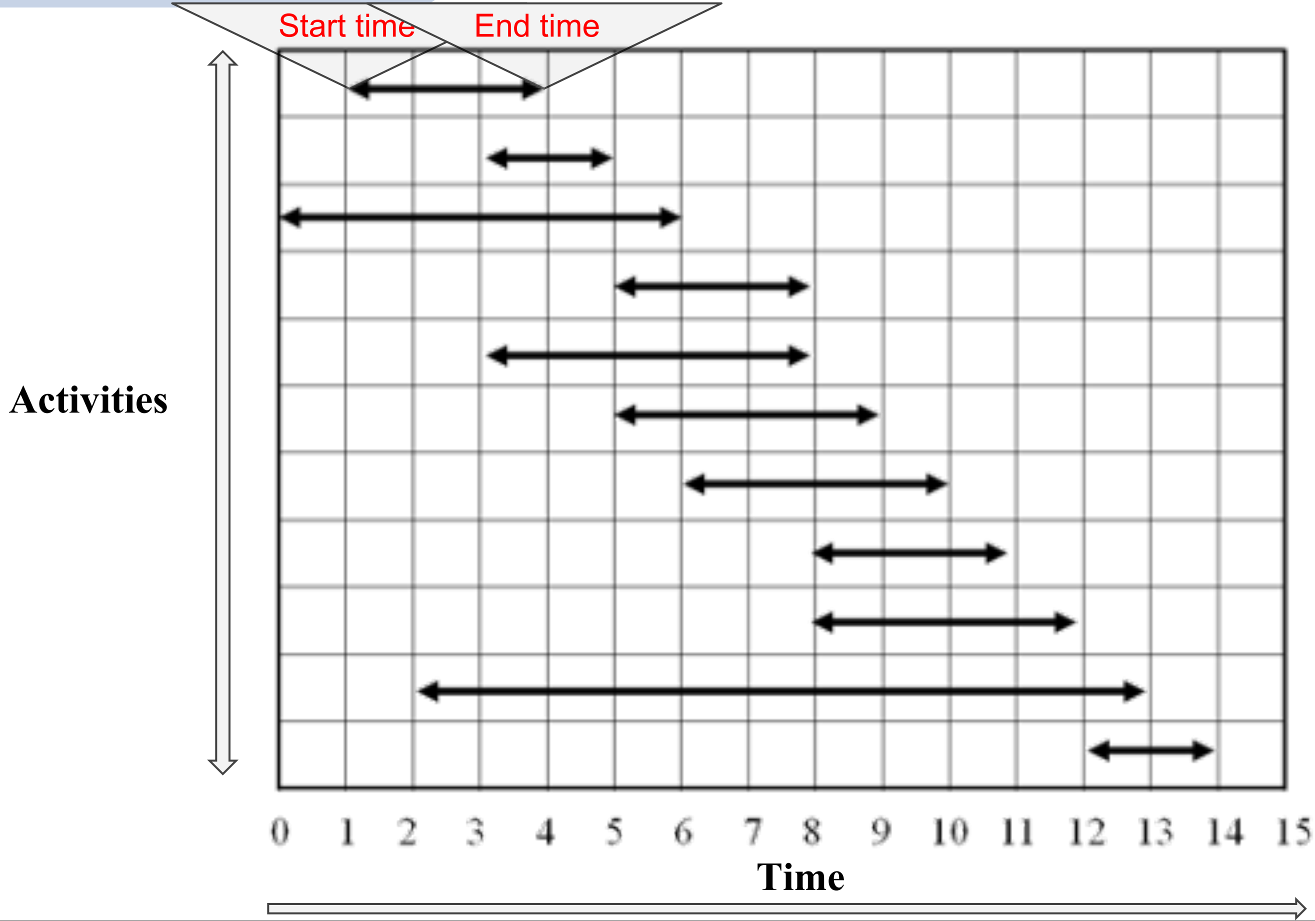
Early Finish Strategy

Select the activity with the earliest finish

Eliminate the activities that could not be scheduled

Repeat !

Example



Start time
End time
Activities
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
Time

Assume activities are sorted by finish time

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

Example

There are 6 activities with corresponding start and end time, the objective is to compute an execution schedule having maximum number of non-conflicting activities

Start Time (s)	Finish Time (f)	Activity Name
5	9	a1
1	2	a2
3	4	a3
0	6	a4
5	7	a5
8	9	a6

Example

Step 1: Sort the given activities in ascending order according to their finishing time.

Start Time (s)	Finish Time (f)	Activity Name
1	2	a2
3	4	a3
0	6	a4
5	7	a5
5	9	a1
8	9	a6

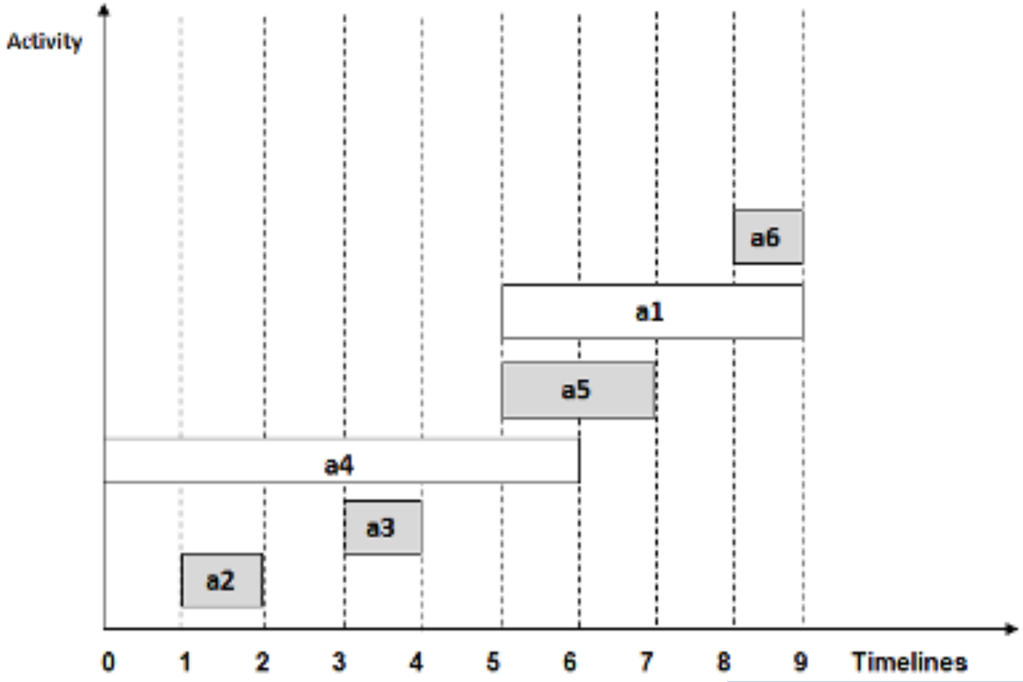
Example

Step 2: Select the first activity from sorted array act[] and add it to the sol[] array, thus **sol = {a2}**.

Step 3: Repeat the steps 4 and 5 for the remaining activities in act[].

Step4: If the start time of the currently selected activity is greater than or equal to the finish time of the previously selected activity, then add it to sol[].

Step 5: Select the next activity in act[]



Example

For the data given in the above table,

1. Select activity **a3**. Since the start time of **a3** is greater than the finish time of **a2** (i.e. s(a3) > f(a2)), we add **a3** to the solution set.

   Thus sol = {a2, a3}.

2. Select **a4**. Since s(a4) < f(a3), it is not added to the solution set.
3. Select **a5**. Since s(a5) > f(a3), **a5** gets added to solution set.

   Thus sol = {a2, a3, a5}

4. Select **a1**. Since s(a1) < f(a5), **a1** is not added to the solution set.
5. Select **a6**. **a6** is added to the solution set since s(a6) > f(a5).

   Thus sol = {a2, a3, a5, a6}.

Step 6: At last, print the array sol[]

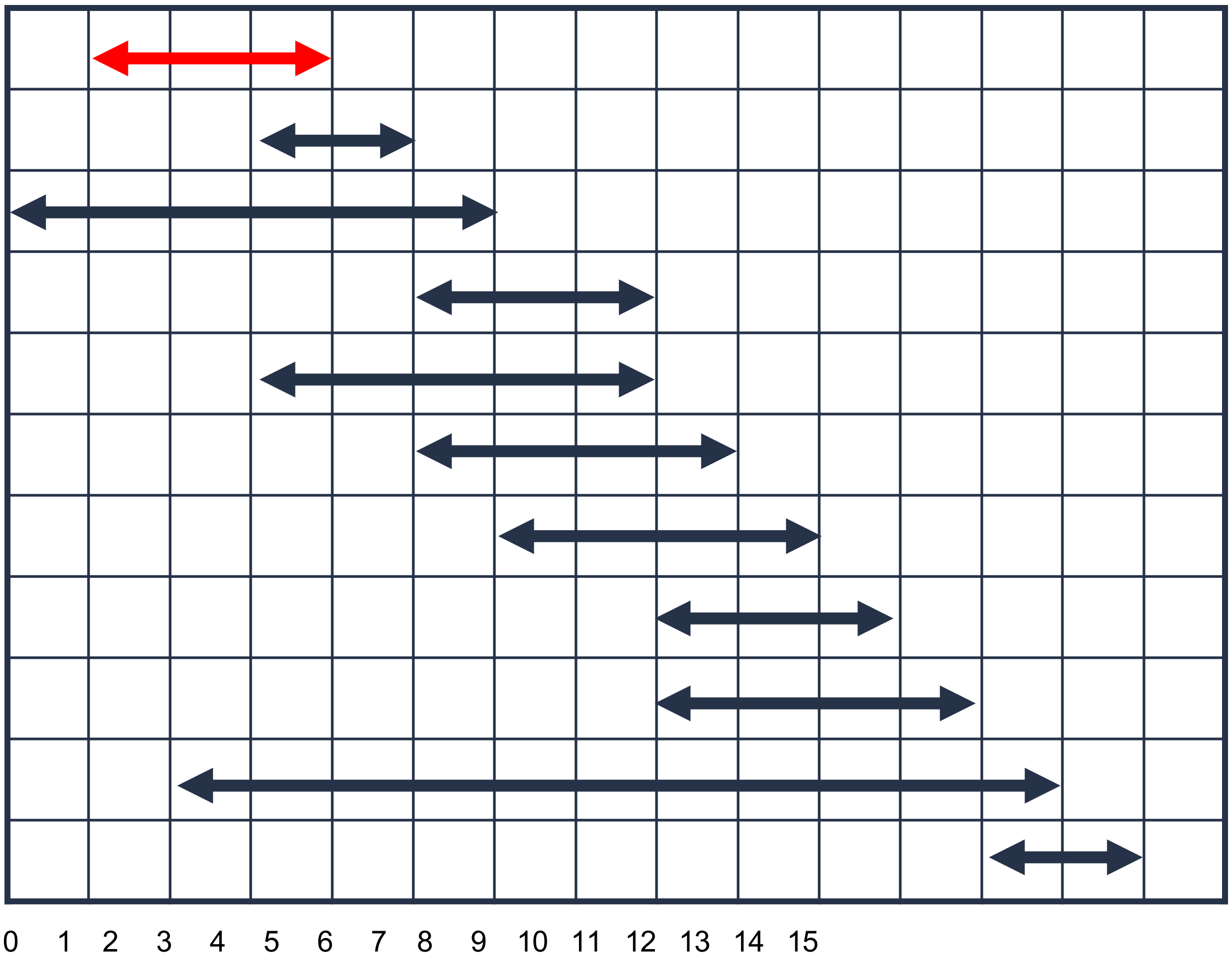
Hence, the execution schedule of maximum number of non-conflicting activities will be:

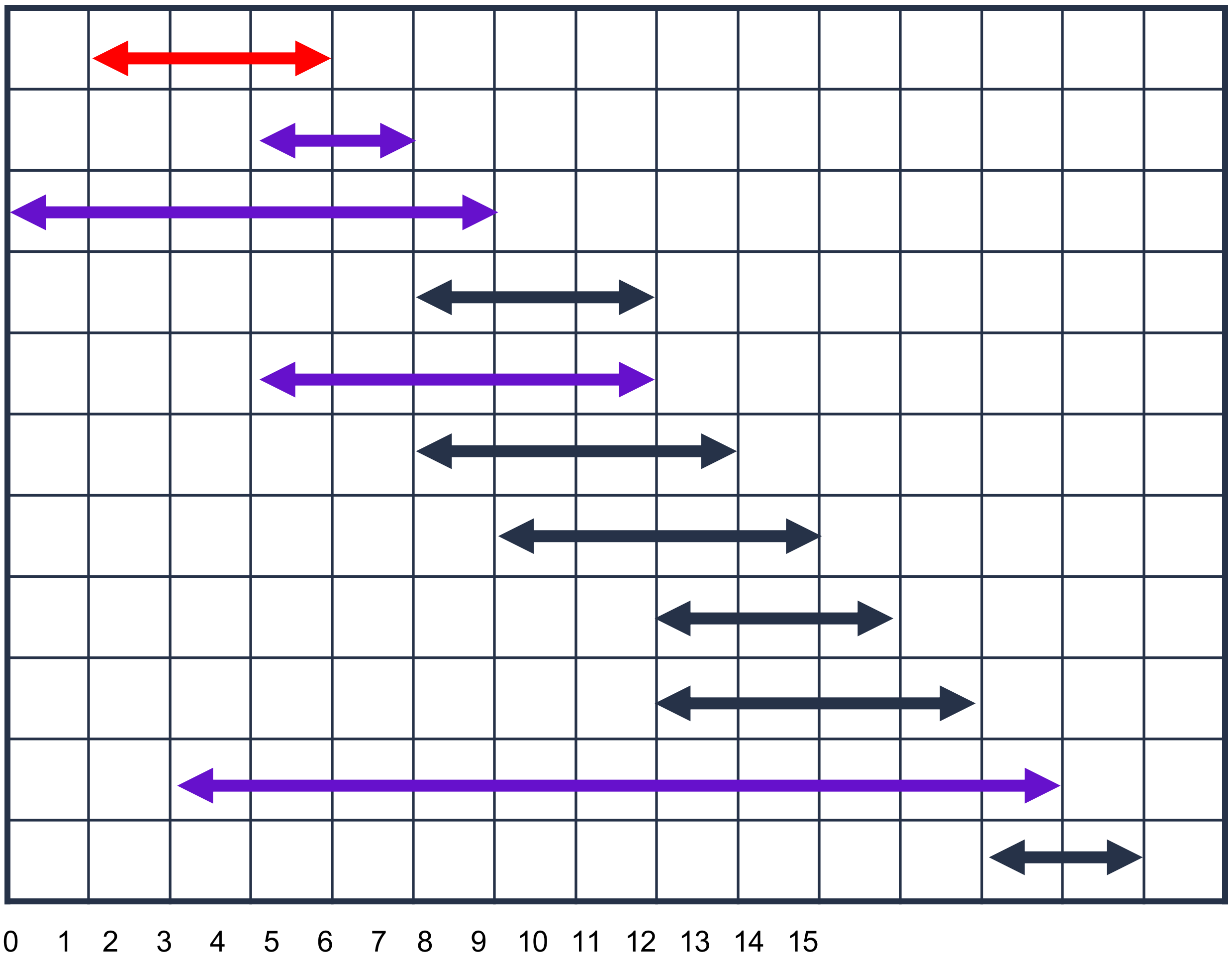
(1,2)

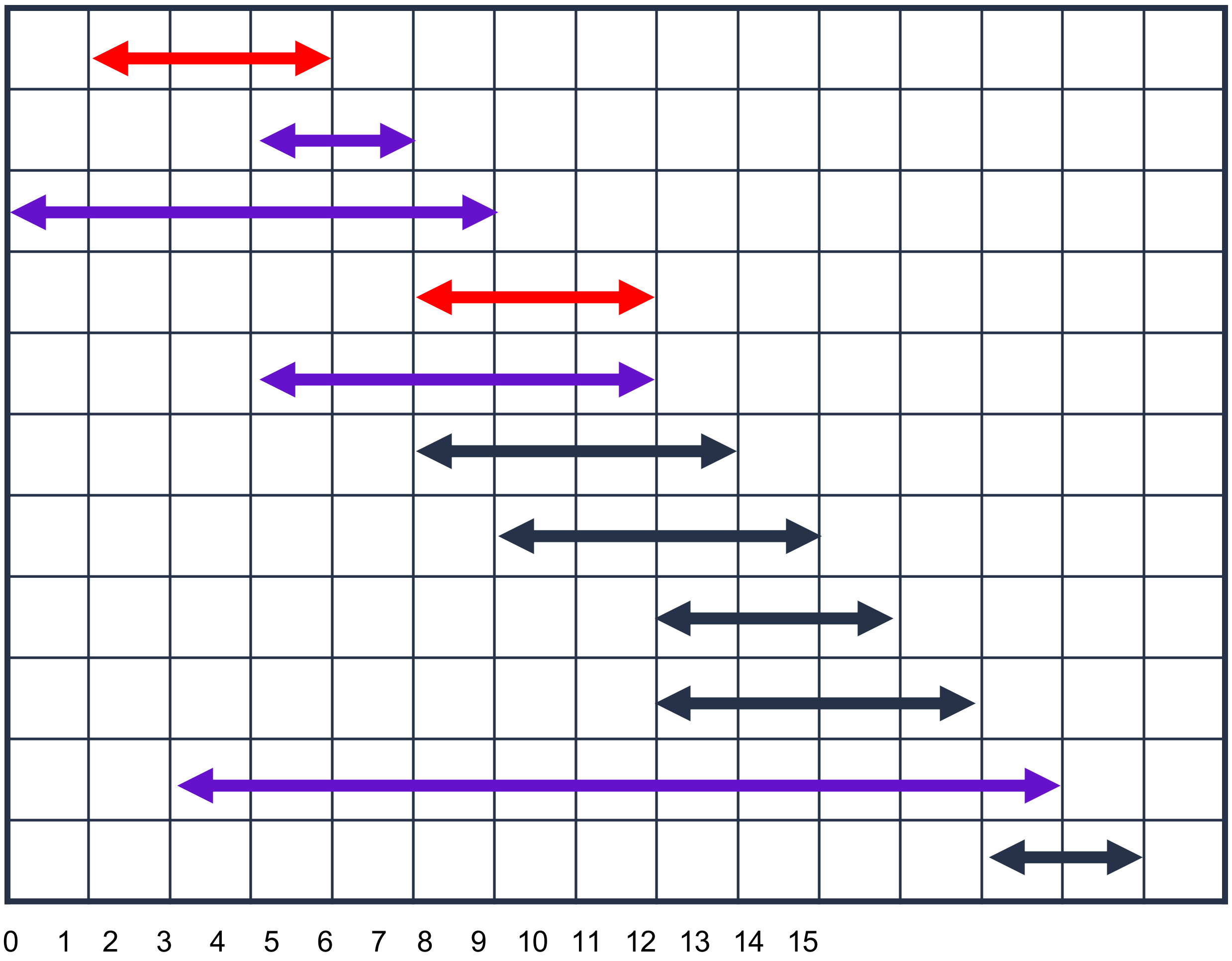
(3,4)

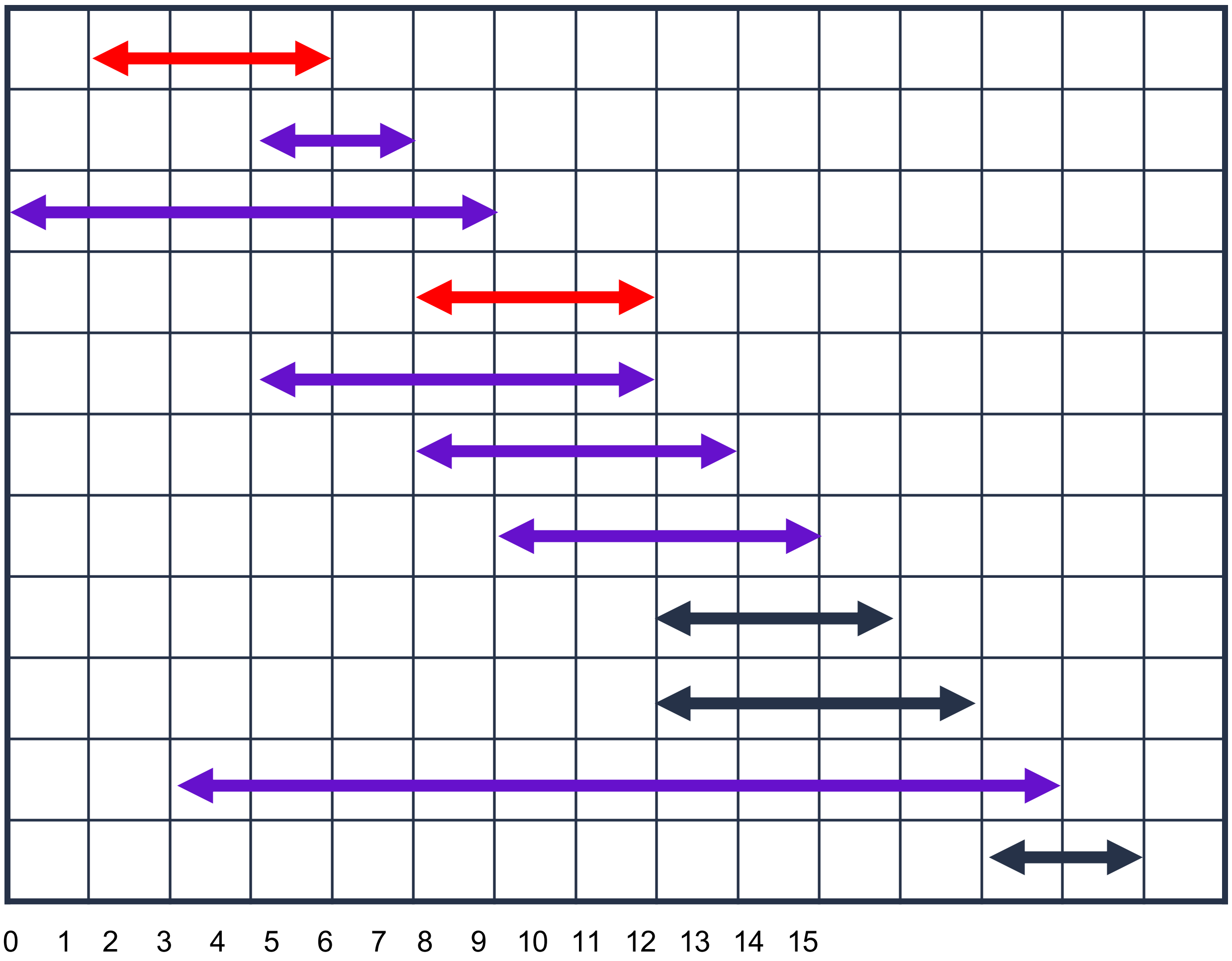
(5,7)

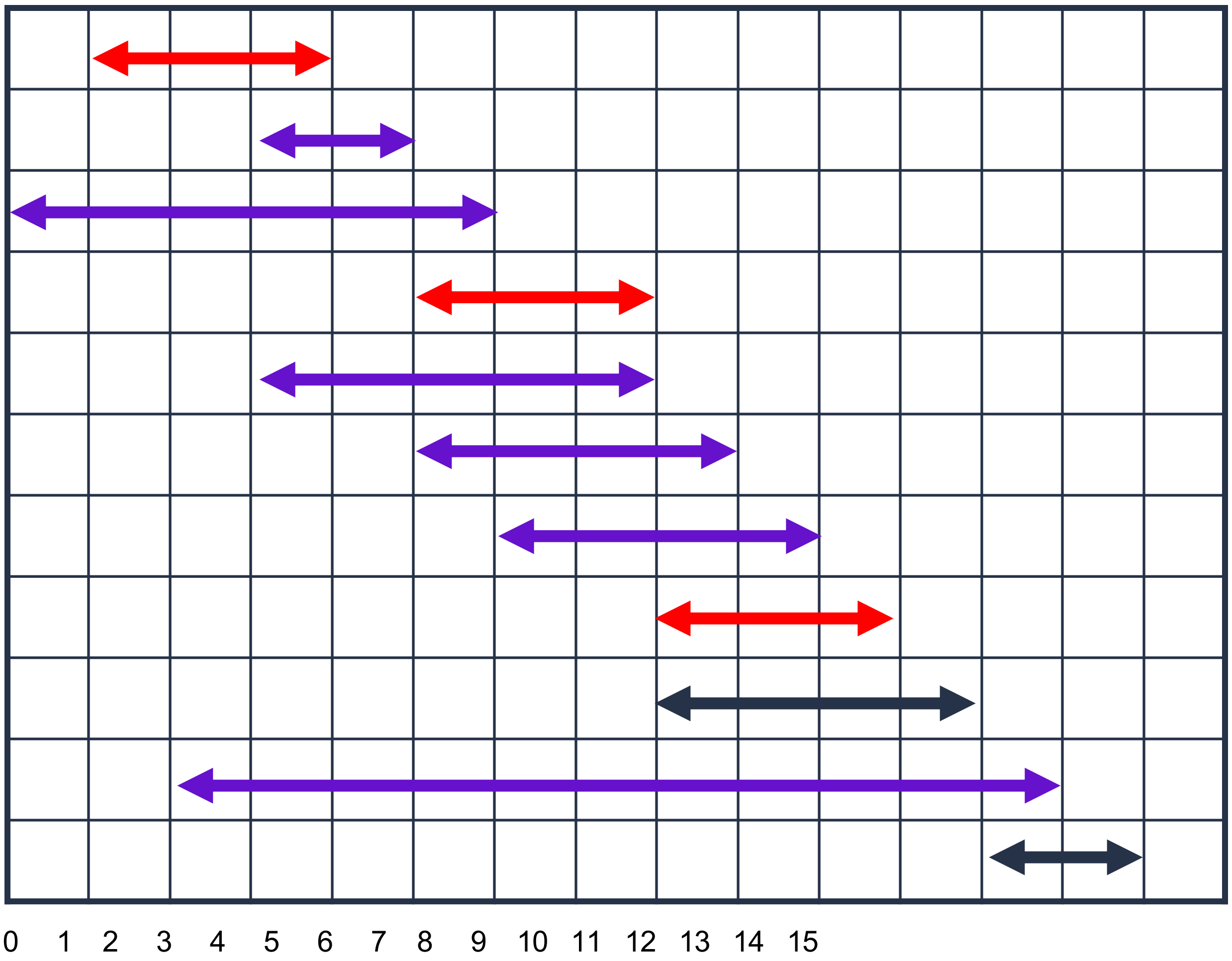
(8,9)

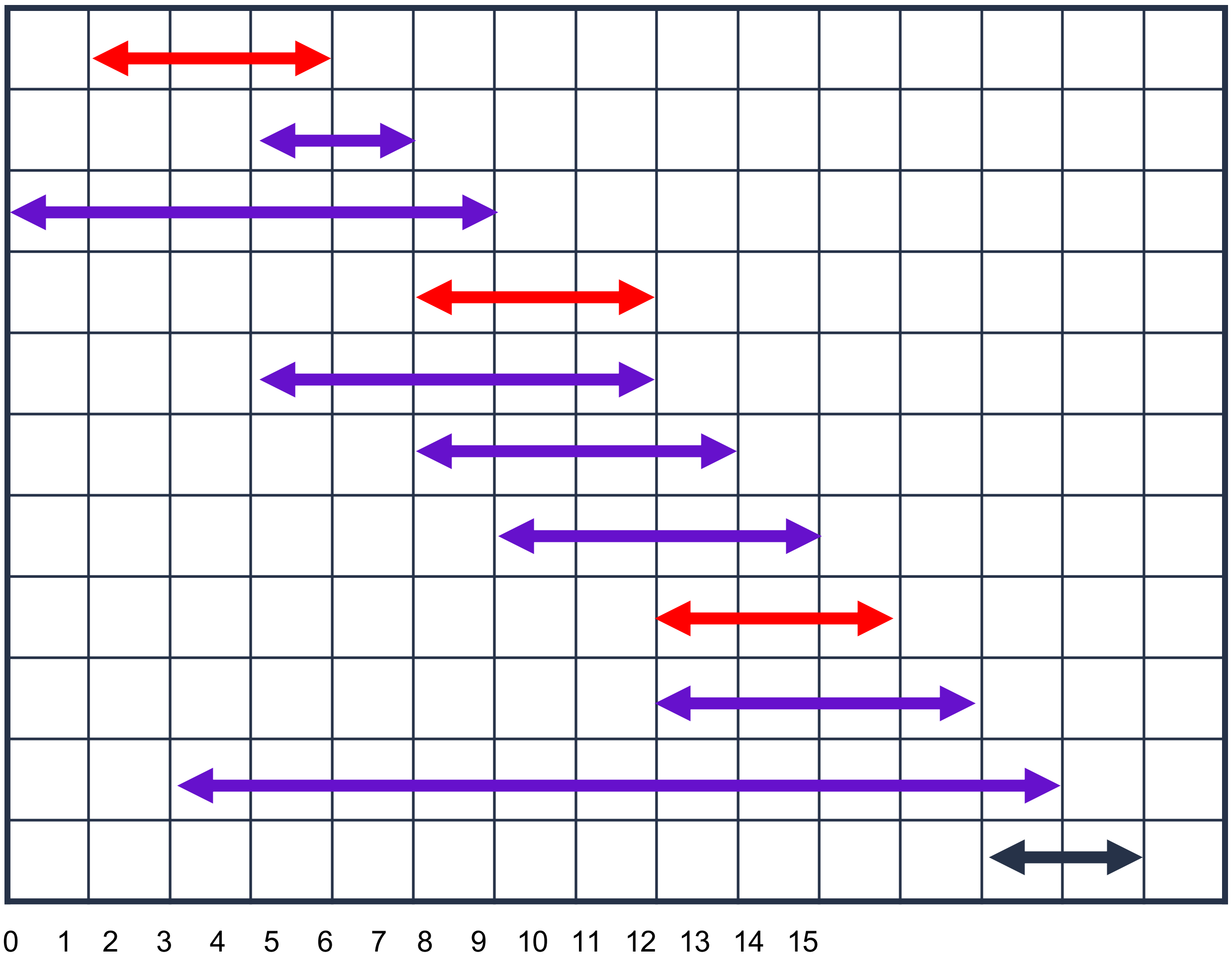

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

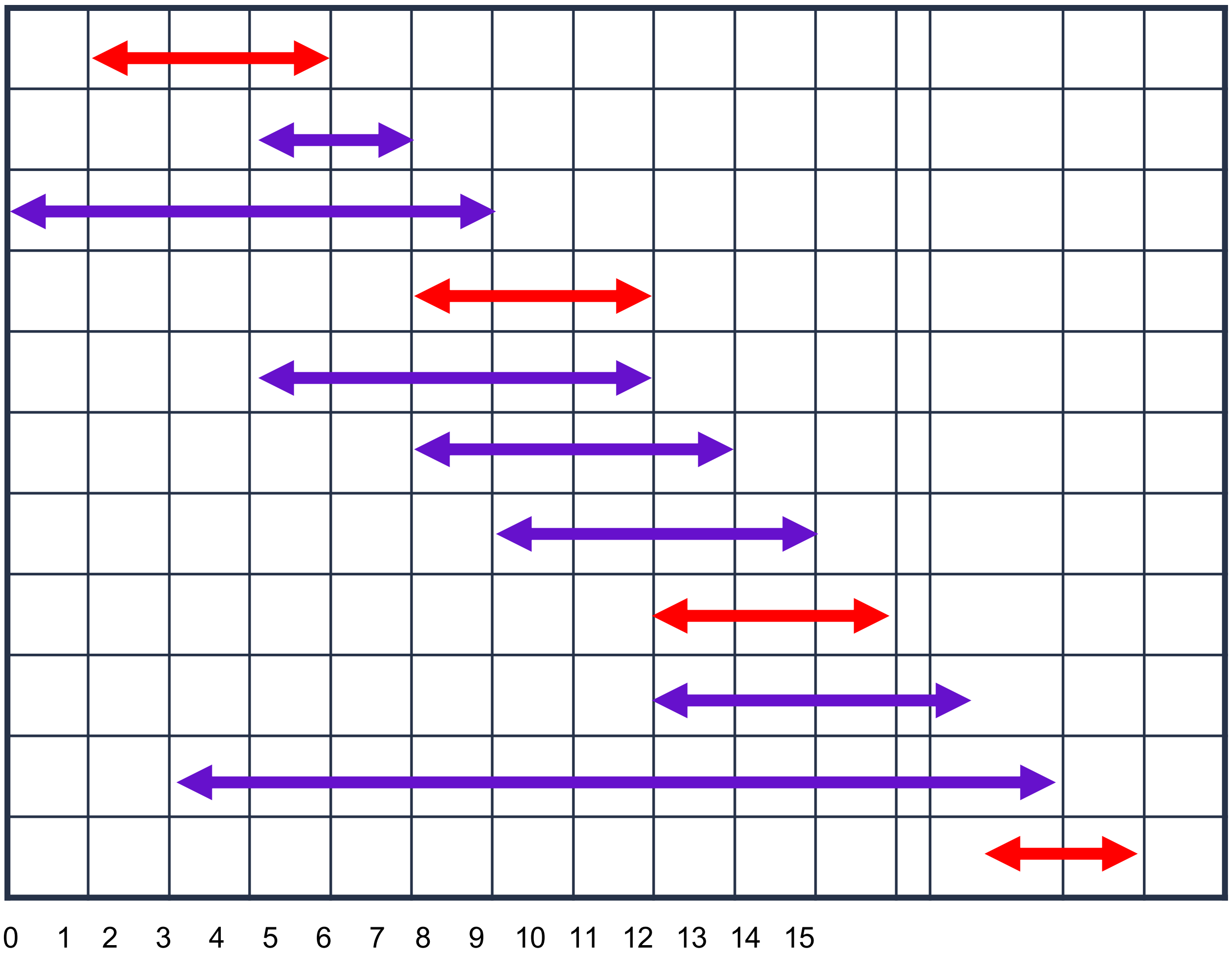

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15


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