MA 515: Introduction to Algorithms & MA353 : Design and Analysis of Algorithms [3-0-0-6]

Lecture 34

http://www.iitg.ernet.in/psm/indexing_ma353/y09/index.html

Partha Sarathi Mandal

psm@iitg.ernet.in

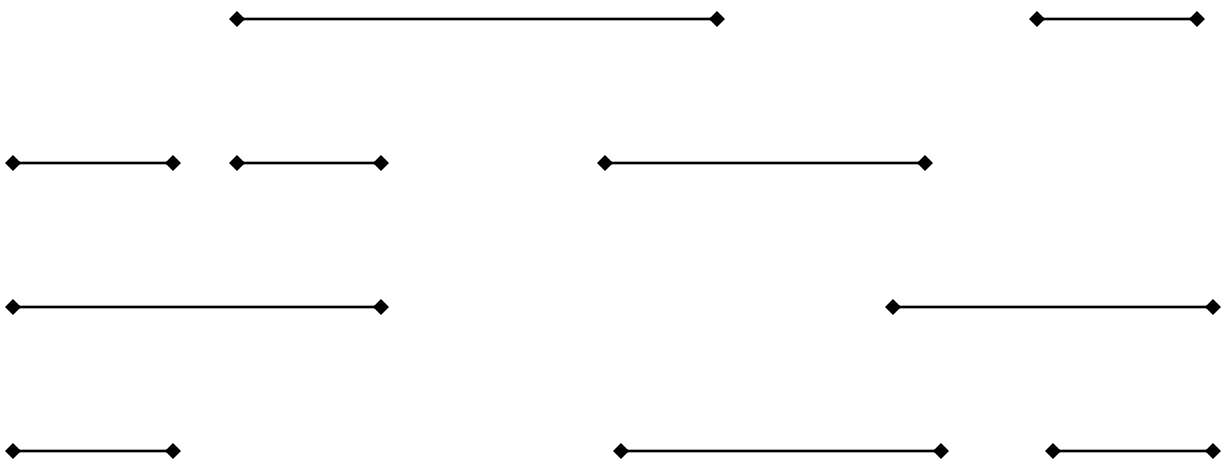
Dept. of Mathematics, IIT Guwahati

Mon 10:00-10:55 Tue 11:00-11:55 Fri 9:00-9:55

Class Room : 2101

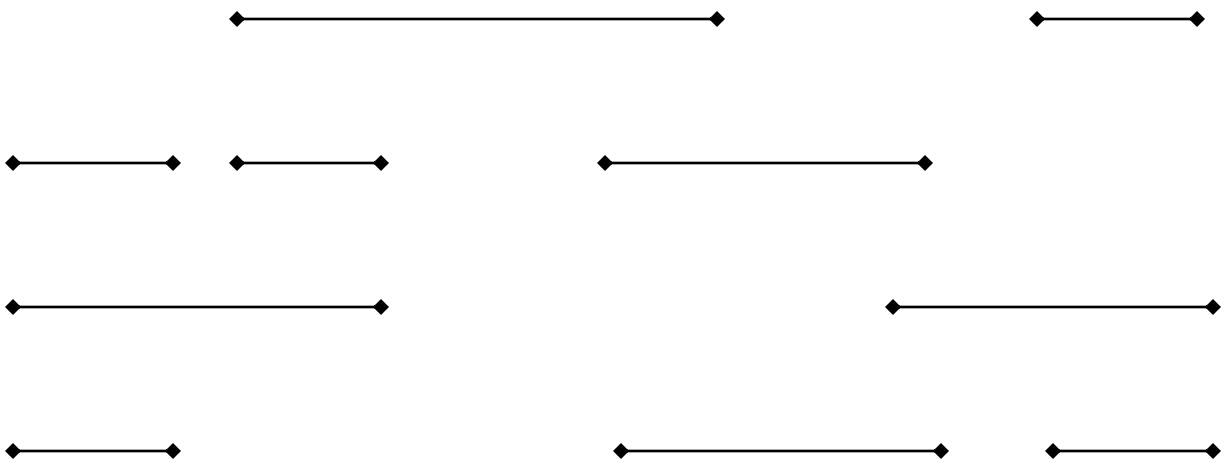
Scheduling *all* intervals

- Given *n* activities, we need to schedule **all** activities. Minimize the number of resources required.

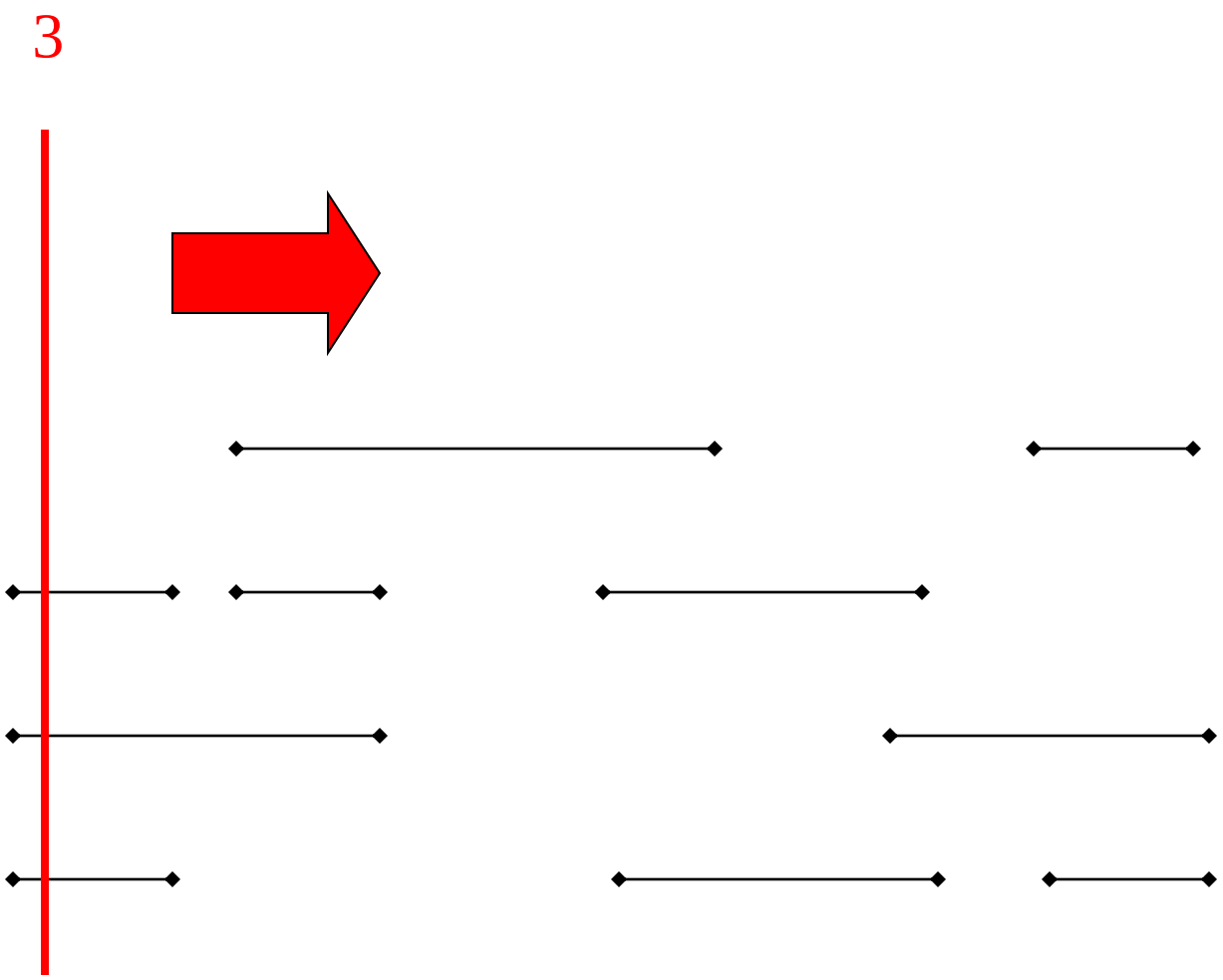


Greedy approach?

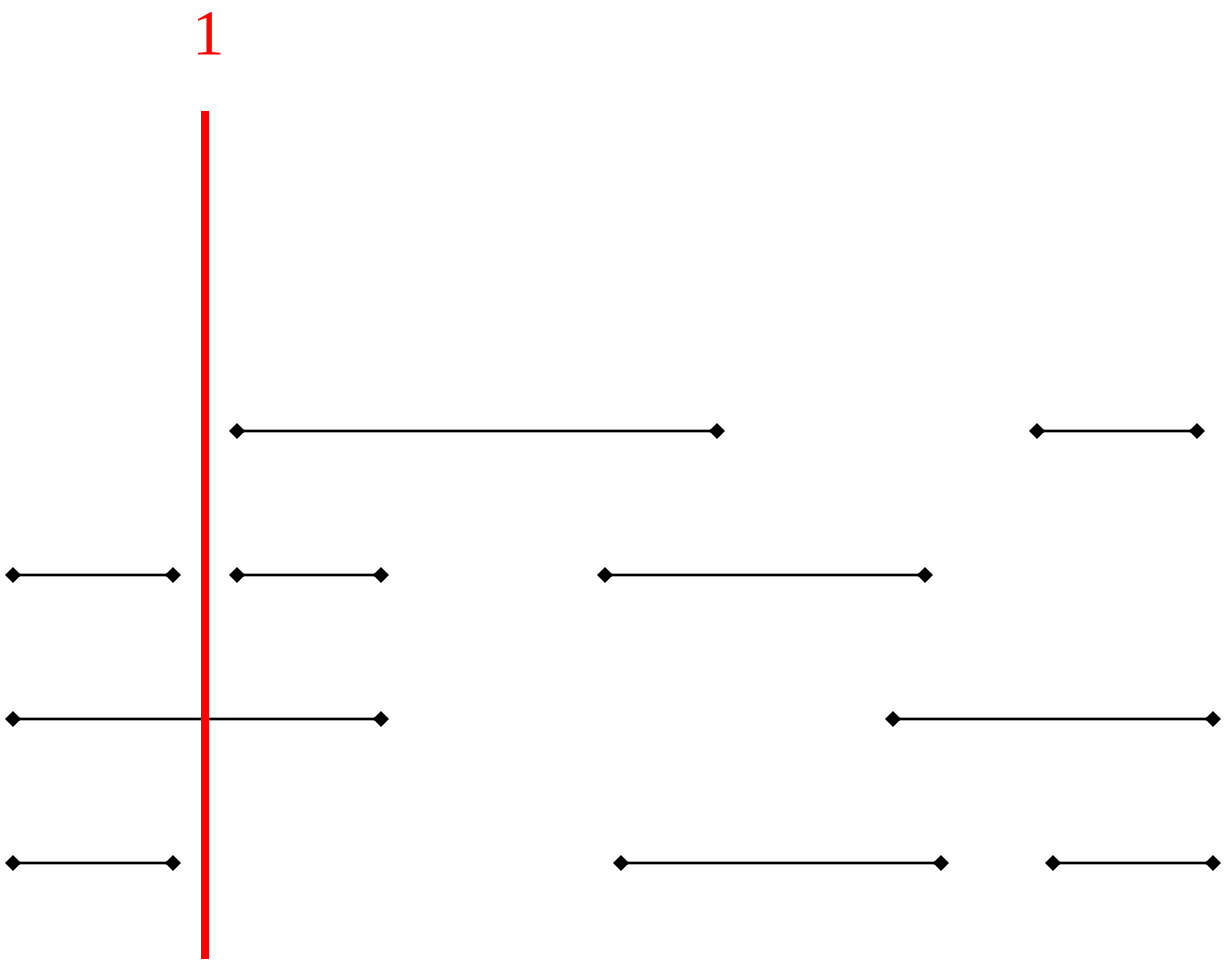
- The best we could ever do is the maximum number of conflicts for any time period



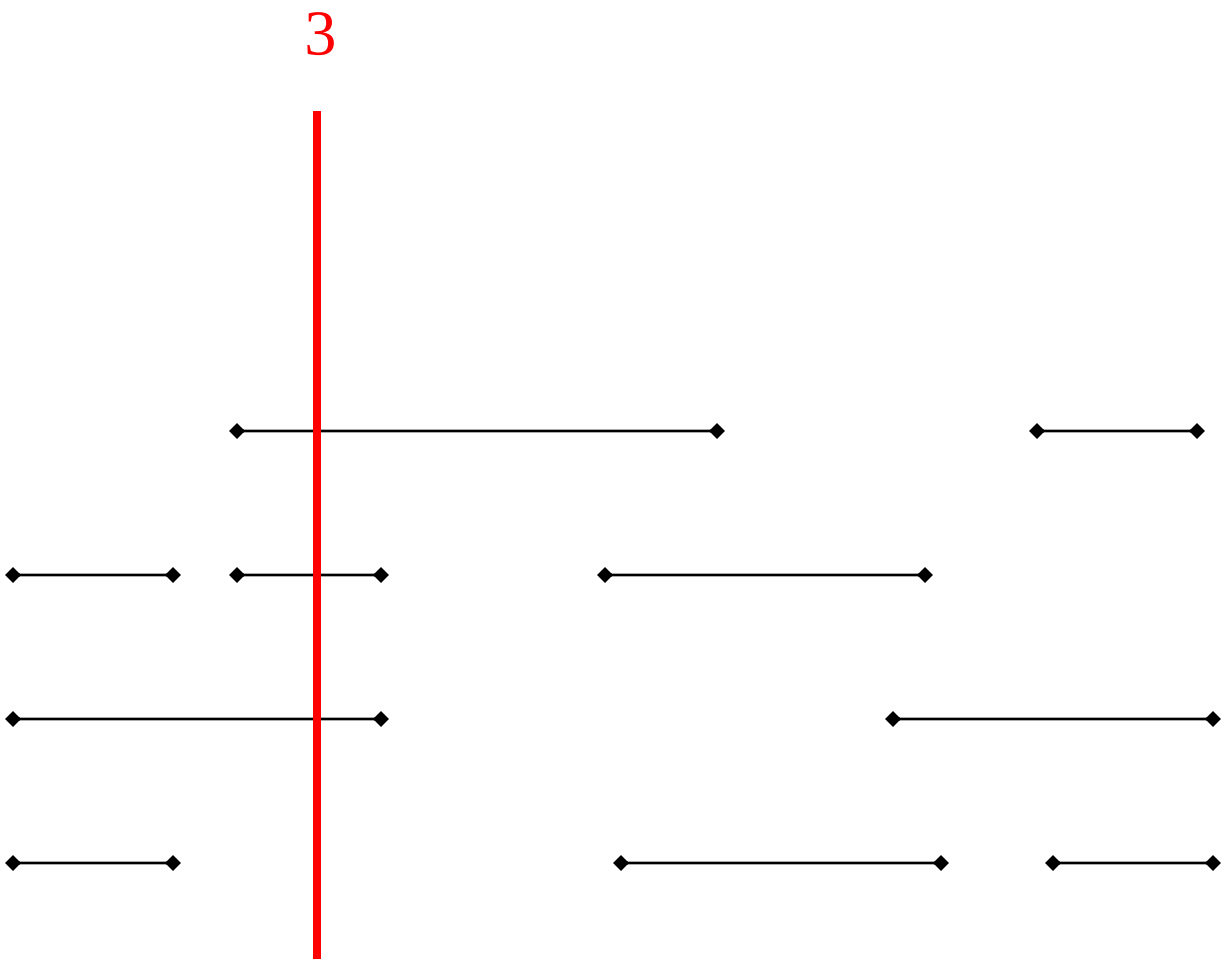
Calculating max conflicts efficiently



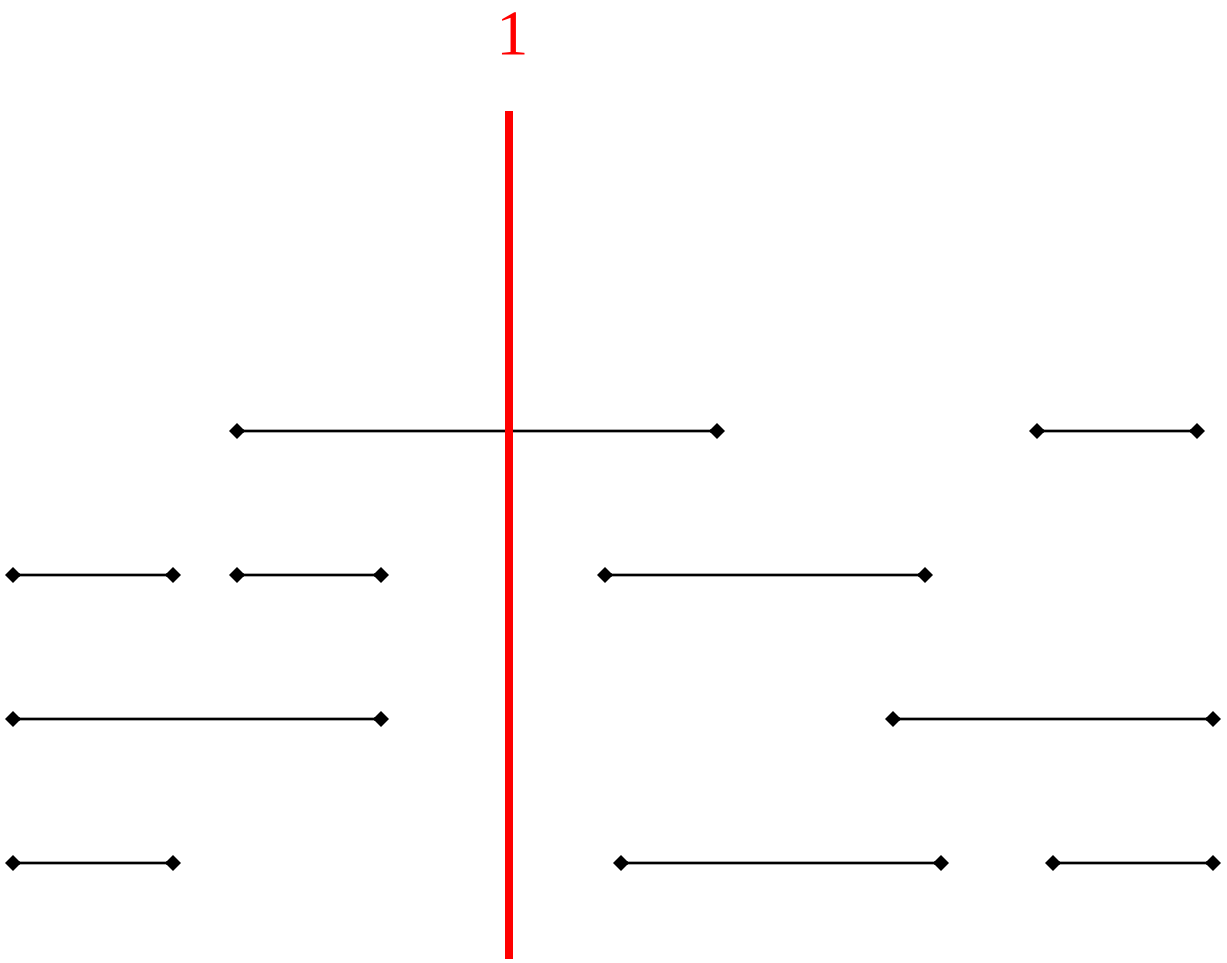
Calculating max conflicts efficiently



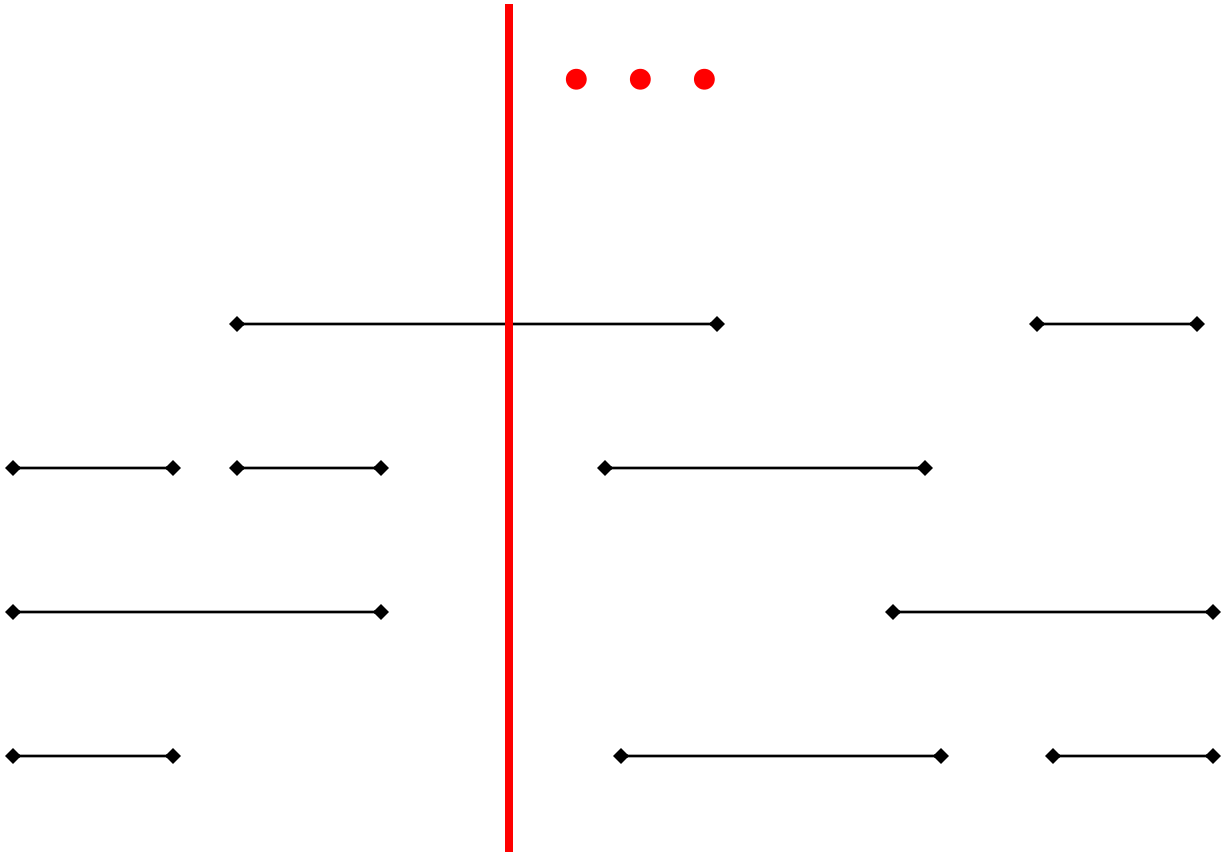
Calculating max conflicts efficiently



Calculating max conflicts efficiently



Calculating max conflicts efficiently



Calculating max conflicts

```
AllIntervalScheduleCount(A)
 1  Sort the start and end times, call this X
 2  current ← 0
 3  max ← 0
 4  for i ← 1 to length[X]
 5          if x_i is a start node
 6                  current++
 7          else
 8                  current--
 9          if current > max
10                  max ← current
11  return max
```

Correctness?

- We can do no better then the max number of conflicts. This exactly counts the max number of conflicts.

```
AllIntervalScheduleCount(A)
 1  Sort the start and end times, call this X
 2  current ← 0
 3  max ← 0
 4  for i ← 1 to length[X]
 5          if x_i is a start node
 6                  current++
 7          else
 8                  current--
 9          if current > max
10                  max ← current
11  return max
```

Runtime?

- O(2n log 2n + n) = O(n log n)

```
AllIntervalScheduleCount(A)
 1  Sort the start and end times, call this X
 2  current ← 0
 3  max ← 0
 4  for i ← 1 to length[X]
 5          if x_i is a start node
 6                  current++
 7          else
 8                  current--
 9          if current > max
10                  max ← current
11  return max
```

Huffman Codes

- Huffman codes: compressing data (savings of 20% to 90%)
- Huffman's greedy algorithm uses a table of the frequencies of occurrence of each character to build up an optimal way of representing each character as a binary string

Huffman Codes

- Idea: use short codes for more frequent characters and long codes for less frequent

char	a	b	c	d	e	f	total bits
#	45	13	12	16	9	5	
fixed	000	001	010	011	100	101	300
variable	0	101	100	111	1101	1100	224

a-f from 97 to 102 in Dec (ASCII)

Finding Optimal Code

- Input:
  - data file of characters and
  - number of occurrences of each character
- Output:
  - a binary encoding of each character so that the data file can be represented as efficiently as possible
  - "optimal code"