Lecture 9: Greedy Algorithms

version of September 28b, 2016

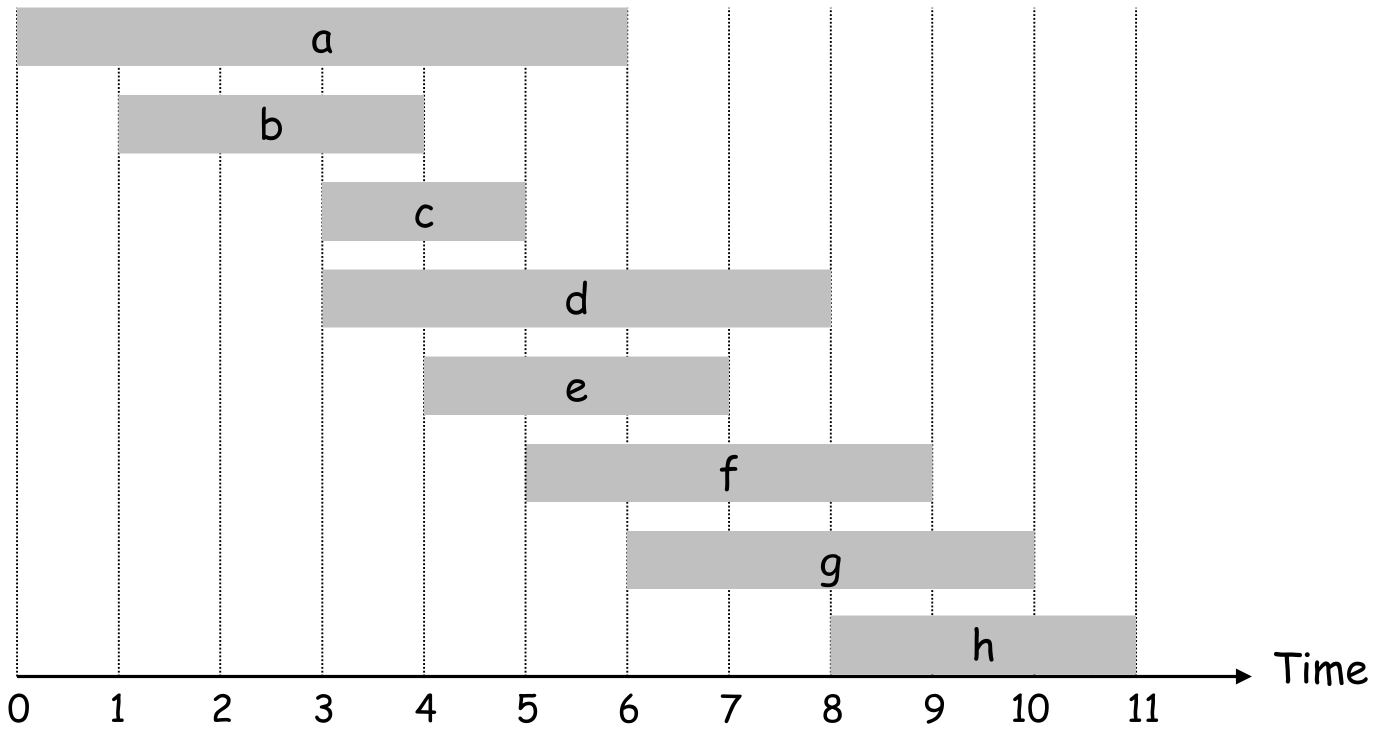
A greedy algorithm always makes the choice that looks best at the moment and adds it to the current partial solution.

Greedy algorithms don't always yield optimal solutions, but when they do, they're usually the simplest and most efficient algorithms available.

Interval Scheduling

Interval scheduling.

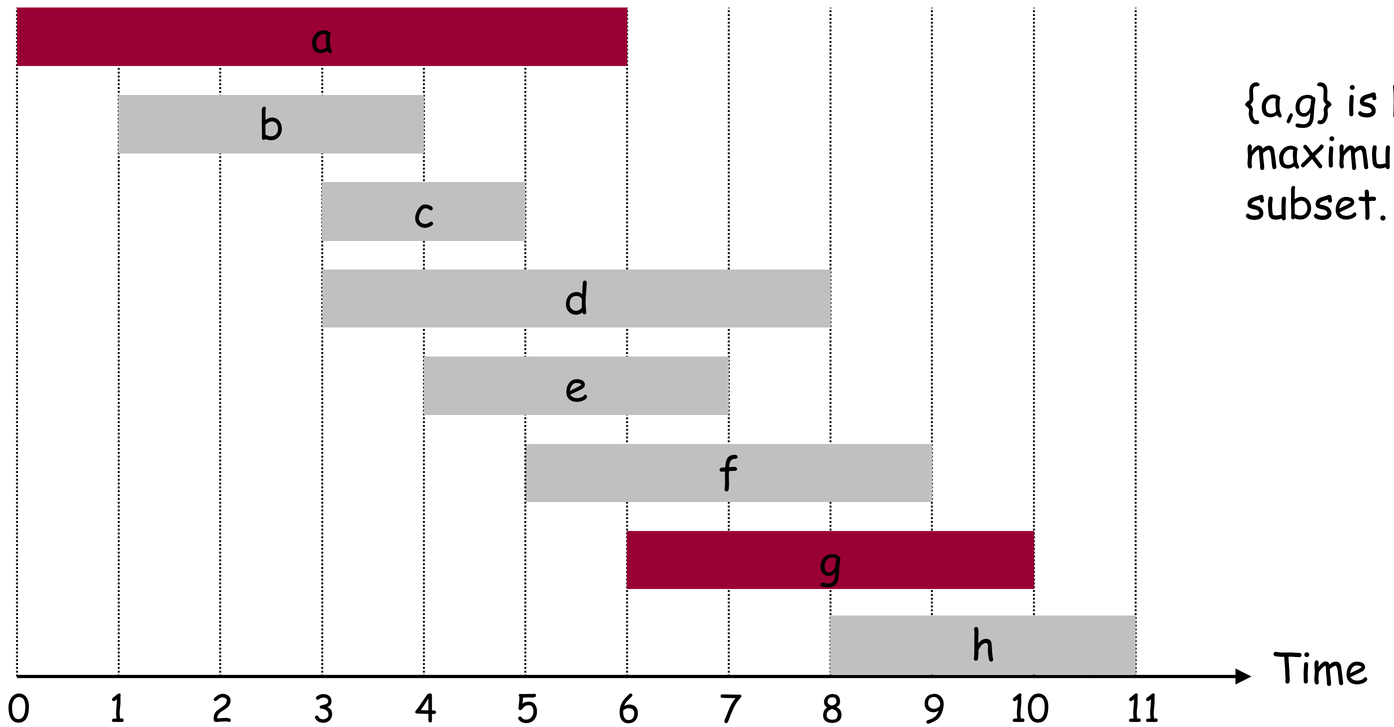
- Job j starts at s_j and finishes at f_j.
- Two jobs are compatible if they don't overlap.
- Goal: find maximum size subset of mutually compatible jobs.



Interval Scheduling

Interval scheduling.

- Job j starts at s_j and finishes at f_j.
- Two jobs are compatible if they don't overlap.
- Goal: find maximum size subset of mutually compatible jobs.

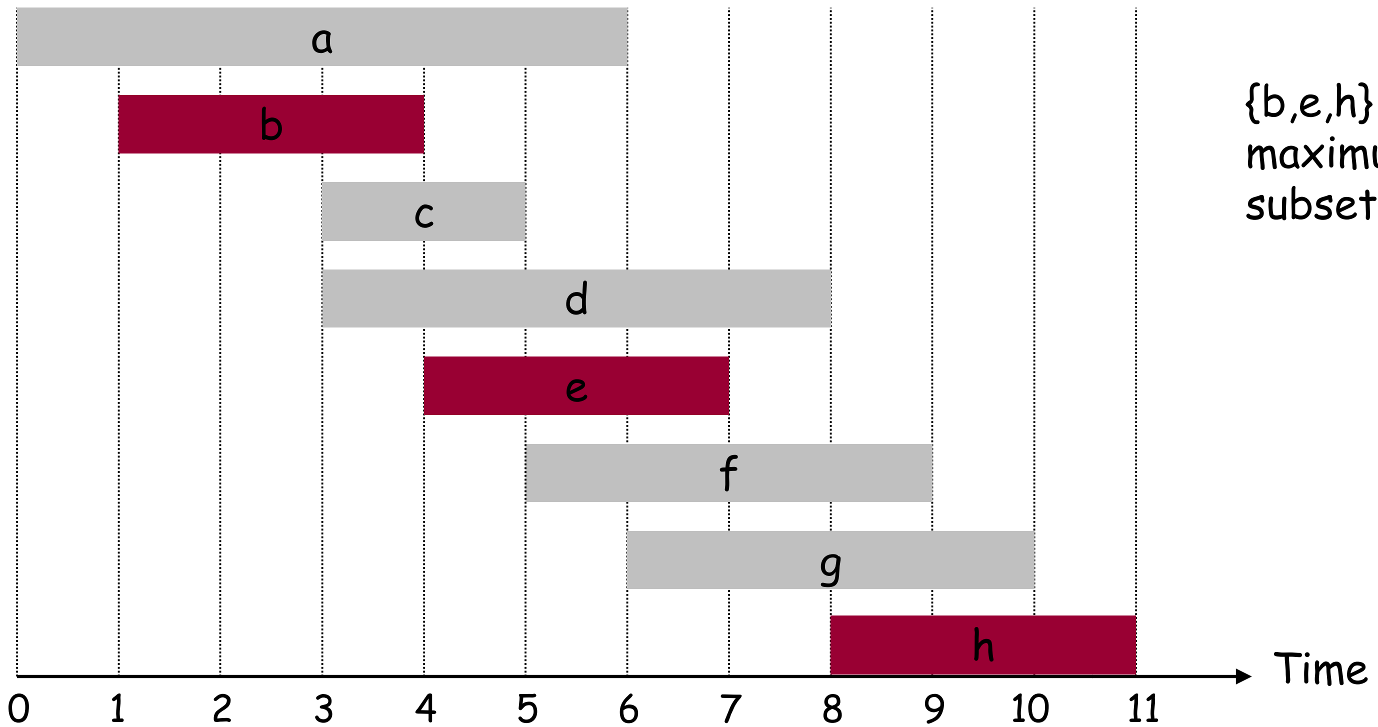


{a,g} is NOT a maximum-size subset.

Interval Scheduling

Interval scheduling.

- Job j starts at s_j and finishes at f_j.
- Two jobs are compatible if they don't overlap.
- Goal: find maximum size subset of mutually compatible jobs.



{b,e,h} is a maximum-size subset.

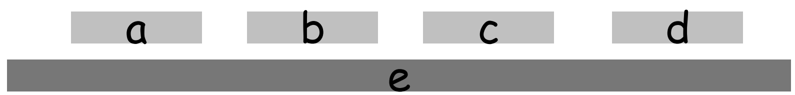
Interval Scheduling: Greedy Algorithms

Greedy template. Consider jobs in some order. Take a job provided it's *compatible* with the ones already taken.

- [Earliest start time] Consider jobs in increasing order of start time s_j.
- [Earliest finish time] Consider jobs in increasing order of finish time f_j.
- [Shortest interval] Consider jobs in increasing order of interval length $f_j - s_j$.
- [Fewest conflicts] For each job, count the number of conflicting jobs c_j. Schedule in ascending order of conflicts c_j.

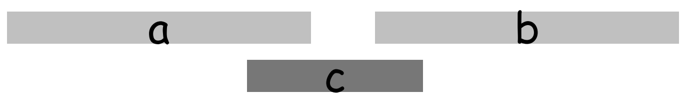
Interval Scheduling: Greedy Algorithms

Greedy template. Consider jobs in some order. Take a job provided it's compatible with the ones already taken.



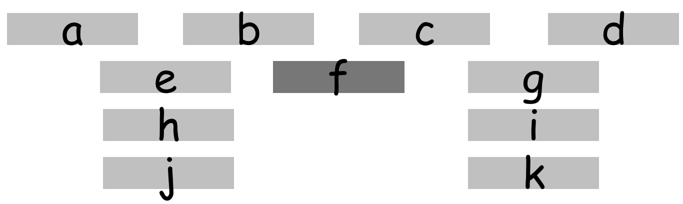
breaks earliest start time

Chooses {e} instead of {a,b,c,d}



breaks shortest interval

Chooses {c} instead of {a,b}

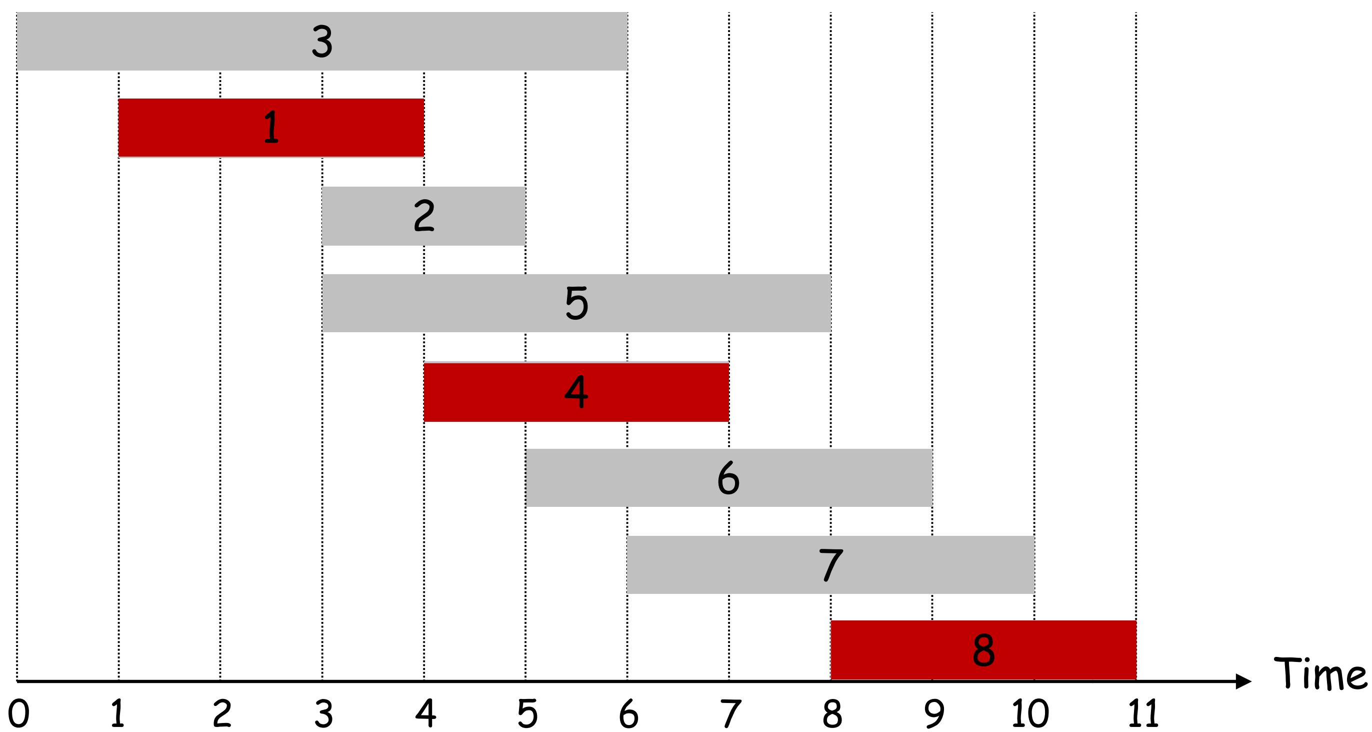


breaks fewest conflicts

Chooses {f} which forces choosing {a,f,d} instead of {a,b,c,d}

Interval Scheduling

Greedy algorithm. Consider jobs in increasing order of finish time.
Take each job provided it's compatible with the ones already taken



Interval Scheduling: Greedy Algorithm

Greedy algorithm. Consider jobs in increasing order of finish time. Take each job provided it's compatible with the ones already taken.

```
Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn
A ← ∅,  last ← 0
for j ← 1 to n
    if sj ≥ last then A ← A ∪ {j},  last ← fj
return A
```

Running time: $\Theta(n \log n)$.

- Remember the finish time of the last job added to A.
- Job j is compatible with A if $s_j \geq last$.

Remember: Correctness (optimality) of greedy algorithms is usually not obvious. Need to prove!

Interval Scheduling: Correctness

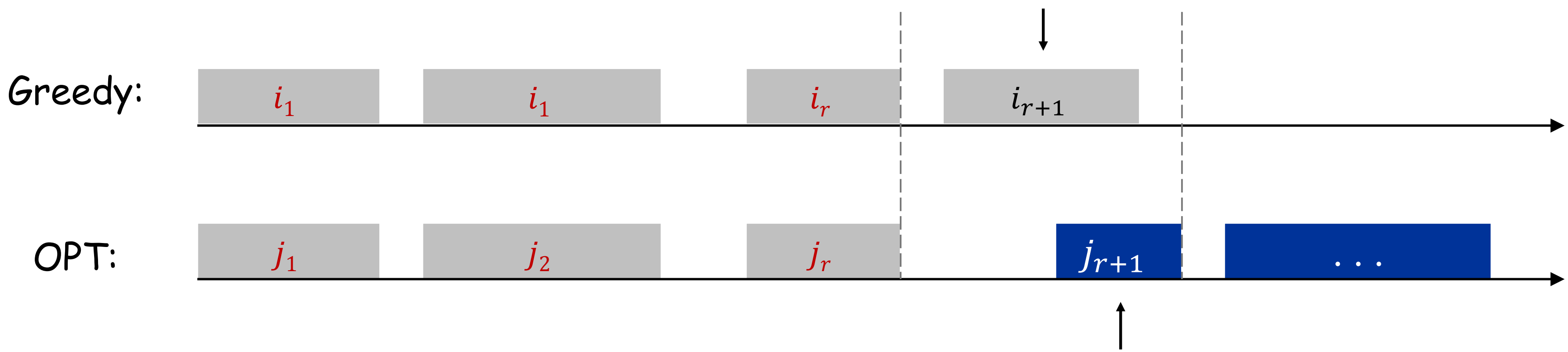
Theorem. Greedy algorithm is optimal.

Proof.

- Assume greedy is different from OPT. Let's see what's different.
- Let $i_1, i_2, \ldots\ i_k$ denote the set of jobs selected by greedy.
- Let $j_1, j_2, \ldots\ j_m$ denote set of jobs in the optimal solution.

  Find largest possible value of r such that $i_1 = j_1,\ \ i_2 = j_2,\ \ \ldots\ ,\ \ i_r = j_r$

By definition of Greedy, job i_{r+1} finishes before j_{r+1}



In OPT, we replace job j_{r+1} with job i_{r+1} , keeping the remainder of OPT the same.
=> OPT still has the same number of jobs, **so it remains optimal**.

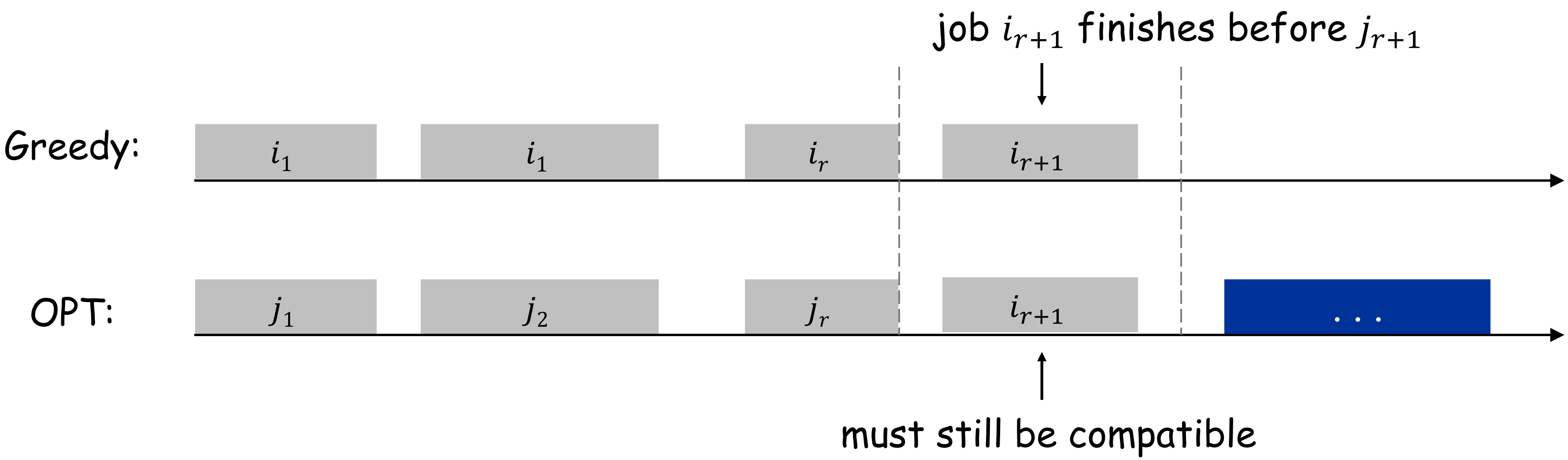
This has created a new optimal solution **that shares its first r+1 jobs** with Greedy.

Interval Scheduling: Correctness

Theorem. Greedy algorithm is optimal.

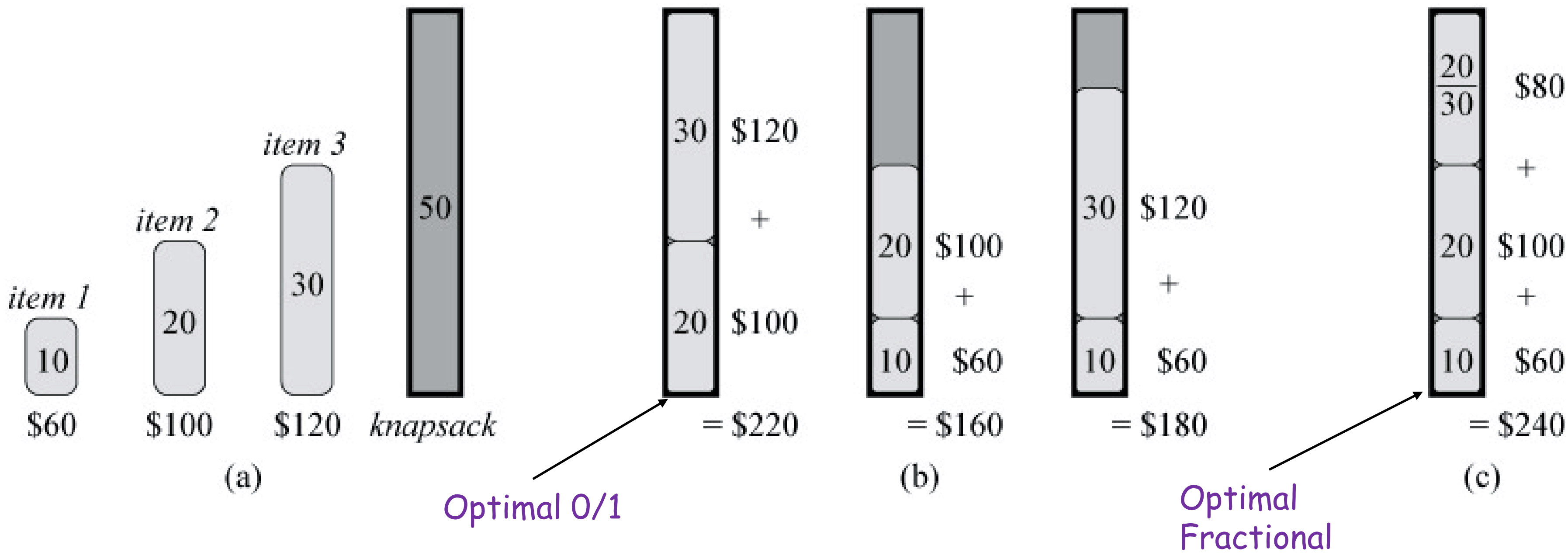
Proof.

- Assume greedy is different from OPT. Let's see what's different.
- Let $i_1, i_2, \ldots i_k$ denote the set of jobs selected by greedy.
- Let $j_1, j_2, \ldots j_m$ denote set of jobs in the optimal solution with $i_1 = j_1, i_2 = j_2, \ldots, i_r = j_r$ for the largest possible value of r.



- Do this replacement repeatedly until OPT is the same as greedy.
  - Important: Since cost remains the same, final solution we've created, which **is** Greedy, is optimal!

The Fractional Knapsack Problem



Input: A set of n items, where item i has weight w_i and value v_i, and a knapsack with capacity W.

Goal: Find $0 \leq x_1, \ldots, x_n \leq 1$ such that $\sum_{i=1}^{n} x_i w_i \leq W$ and $\sum_{i=1}^{n} x_i v_i$ is maximized.

There are two different versions of this problem:

- The x_i's must be 0 or 1: The 0/1 knapsack problem.
- The x_i's can take fractional values: The fractional knapsack problem

The Greedy Algorithm for Fractional Knapsack

```
Sort items so that v_1/w_1 > v_2/w_2 > ... > v_n/w_n
w ← W
for i ← 1 to n
    if w_i ≤ w then
        x_i ← 1
        w ← w − w_i
    else
        x_i ← w/w_i
        return
return
```

Idea:

- Sort all items by value-per-pound
- For each item, take as much as possible

Running time: $\Theta(n \log n)$

Note: This algorithm cannot solve the 0/1 version optimally (why).

Greedy Algorithm: Correctness

Theorem: The greedy algorithm is optimal.

Proof: We will assume that $\sum_{i=1}^{n} w_i \geq W$. Otherwise the algorithm is trivially optimal.

Let the greedy solution be $G = (x_1, x_2, \dots, x_k, 0, \dots, 0)$

- Note: All x_i's must be equal to 1, except possibly for $i = k$.

Consider any optimal solution $O = (y_1, y_2, \dots, y_n)$

- Note: Both G and O must fully pack the knapsack.

Look at the first item i where the two solutions differ.

- That is

$$G = (x_1, x_2, \dots, x_{i-1}, x_i, \dots, x_k, 0, \dots, 0)$$
$$O = (x_1, x_2, \dots, x_{i-1}, y_i, \dots \dots \dots \dots \dots, y_n)$$

- By definition of greedy , $x_i > y_i$
- Let $x = x_i - y_i$

Greedy Algorithm: Correctness (continued)

Recall $x = x_i - y_i$

$G = (x_1, x_2, \dots, x_{i-1}, x_i, \dots, x_k, 0, \dots, 0)$ $\quad O = (x_1, x_2, \dots, x_{i-1}, y_i, \dots \dots \dots \dots \dots, y_n)$

We will modify O as follows:

- Set $y_i \leftarrow x_i$ and
  remove xw_i units of total weight from items $i+1$ to item n
- This is always doable because, in O, the total weight of items i to n is the same as that in G

After the modification:

- The total value cannot decrease, since all the subsequent items have lesser or equal value-per-pound.
- Since O is already an optimal solution, its value cannot increase.
- So O's value must stay the (optimal) same.

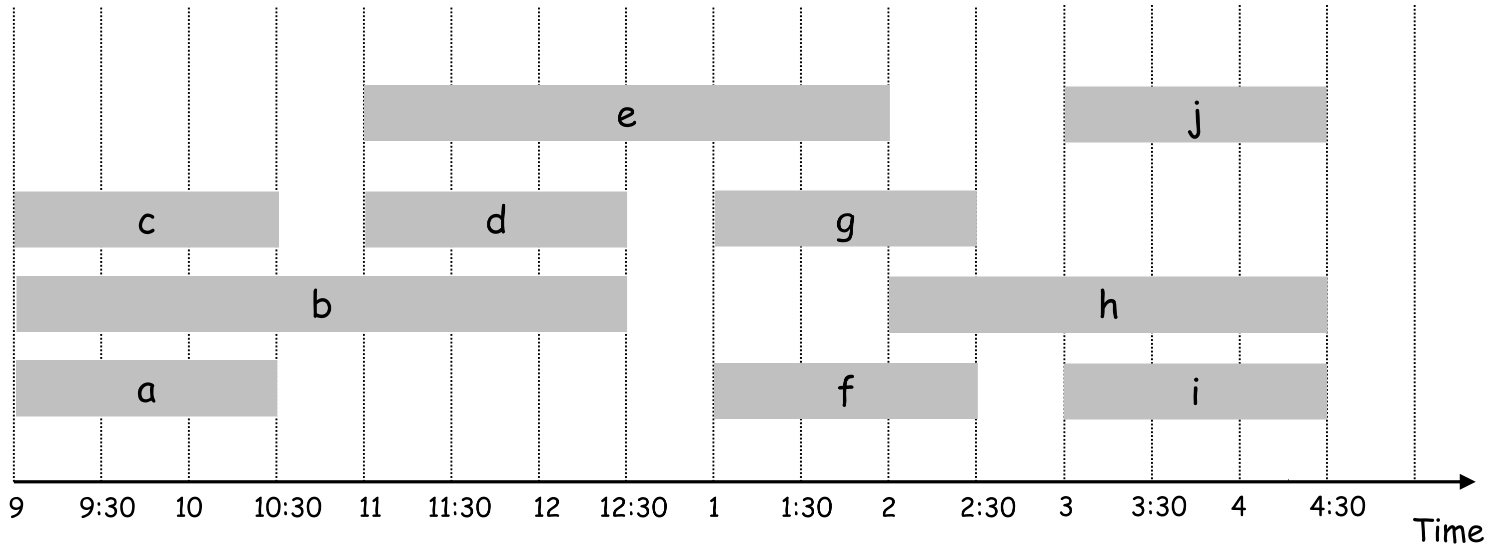
By repeating this process, we will eventually convert O into G, without changing the total value of the selection. Therefore G is also optimal.

Interval Partitioning

Interval partitioning.

- Lecture j starts at s_j and finishes at f_j.
- Goal: find the **minimum number of classrooms** to schedule all lectures so that no two occur at the same time in the same room.

Ex: This schedule uses 4 classrooms to schedule 10 lectures.

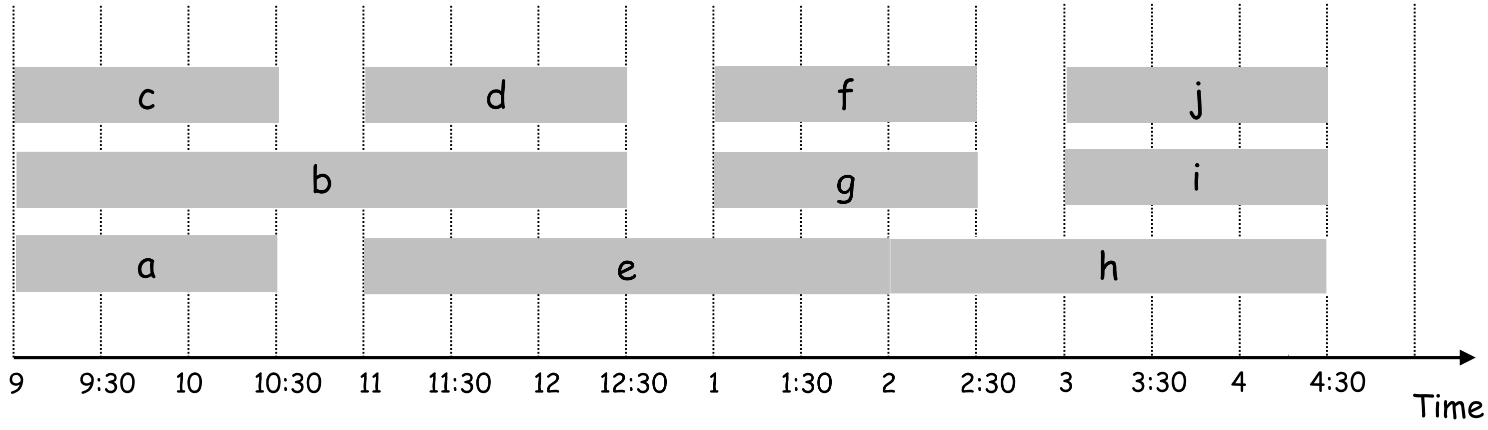


Interval Partitioning

Interval partitioning.

- Lecture j starts at s_j and finishes at f_j.
- Goal: find the minimum number of classrooms to schedule all lectures so that no two occur at the same time in the same room.

Ex: This schedule uses only 3.



Interval Partitioning: Greedy Algorithm

Greedy algorithm. Consider lectures in increasing order of start time: assign lecture to any compatible classroom.

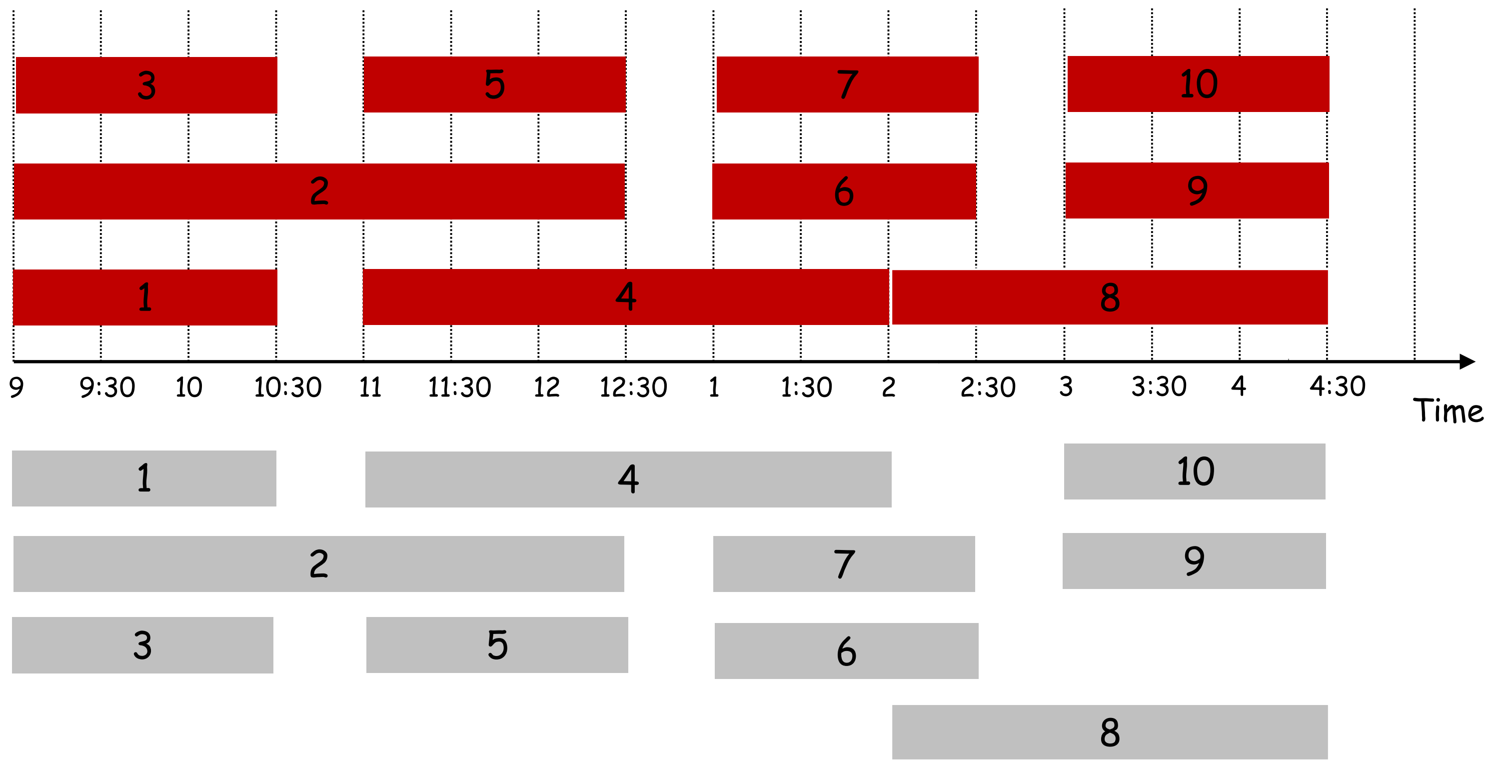
```
Sort intervals by starting time so that s1 ≤ s2 ≤ ... ≤ sn.
d ← 0  // # classrooms used so far
for j ← 1 to n
    if lecture j is compatible with some classroom k then
        schedule lecture j in classroom k
    else
        allocate a new classroom d + 1
        schedule lecture j in classroom d + 1
        d ← d + 1
```

Interval Partitioning

Interval partitioning.

- Lecture j starts at s_j and finishes at f_j.
- Goal: find the minimum number of classrooms to schedule all lectures so that no two occur at the same time in the same room.

ALG: Sort by start time. Insert in order, opening new classroom when needed



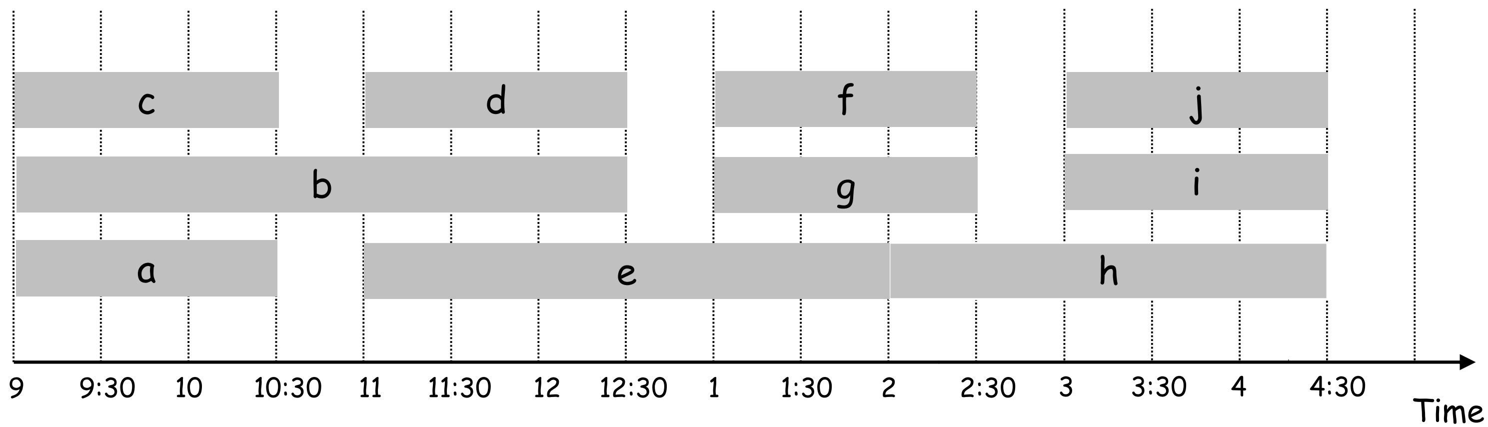
Interval Partitioning: Lower Bound on Optimal Solution

Def. The depth of a set of open intervals is the maximum number that contain any time instance.

Key observation. Number of classrooms needed $\geq$ depth.

Ex: Depth of schedule below = 3 $\Rightarrow$ this schedule is optimal.

We will show: The # classrooms used by the greedy algorithm = depth.



Interval Partitioning: Correctness

Theorem. Greedy algorithm is optimal.

Pf.

- Let d = number of classrooms opened by greedy algorithm .
- Classroom d is opened because we needed to schedule a lecture, say j, that is incompatible with all $d-1$ other classrooms.
- Since we sorted by start time, all these incompatibilities are caused by lectures that all start no later than s_j and finish later than than s_j.
- Thus, we have d lectures all overlapping at time $s_j + \varepsilon$ for some $\varepsilon > 0$; the d-1 incompatible ones and lecture j.
- $\Rightarrow$ depth $\geq d$.
- Thus, Since every algorithm uses at least depth classrooms Greedy is optimal.

Interval Partitioning: Running Time

```
Sort intervals by starting time so that s1 ≤ s2 ≤ ... ≤ sn.
d ← 0   // # classrooms used so far
for j ← 1 to n
    if lecture j is compatible with some classroom k then (*)
        schedule lecture j in classroom k   (**)
    else
        allocate a new classroom d + 1
        schedule lecture j in classroom d + 1   (***)
        d ← d + 1
```

Running time: $\Theta(n \log n)$

- To implement line (*) the algorithm maintains, for each classroom, the finishing time of the last item placed in the classroom. It then compares s_j to those finishing times. If $s_j \geq$ one of those finishing times, it places lecture j in the associated classroom

- A Brute-force implementation of line (*) takes $O(n)$ time
  $\Rightarrow O(n^2)$ in total

- Observation: If j is not compatible with the classroom with **the earliest finish time**, then j is not compatible with any other classroom

Interval Partitioning: Running Time

```
Sort intervals by starting time so that s_1 ≤ s_2 ≤ ... ≤ s_n.
d ← 0   // # classrooms used so far
for j ← 1 to n
    if lecture j is compatible with some classroom k then (*)
        schedule lecture j in classroom k   (**)
    else
        allocate a new classroom d + 1
        schedule lecture j in classroom d + 1   (***)
        d ← d + 1
```

Running time: $\Theta(n \log n)$

- To implement line (*) we can keep the classrooms in a min priority queue using the finishing times of the last class in the room as key
- $O(\log n)$ To check whether there is a compatible classroom we do an extract-min to find the minimum finishing time in the priority queue.
- $O(\log n)$ If (**) is implemented then just add the new finishing time s_j to p. queue
- $O(\log n)$ If (***) is implemented then re-insert that minimum finishing time back into the p. queue AND insert the new finishing time s_j into the p. queue