Lecture 9: Greedy Algorithms

Version of February 27, 2019

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.

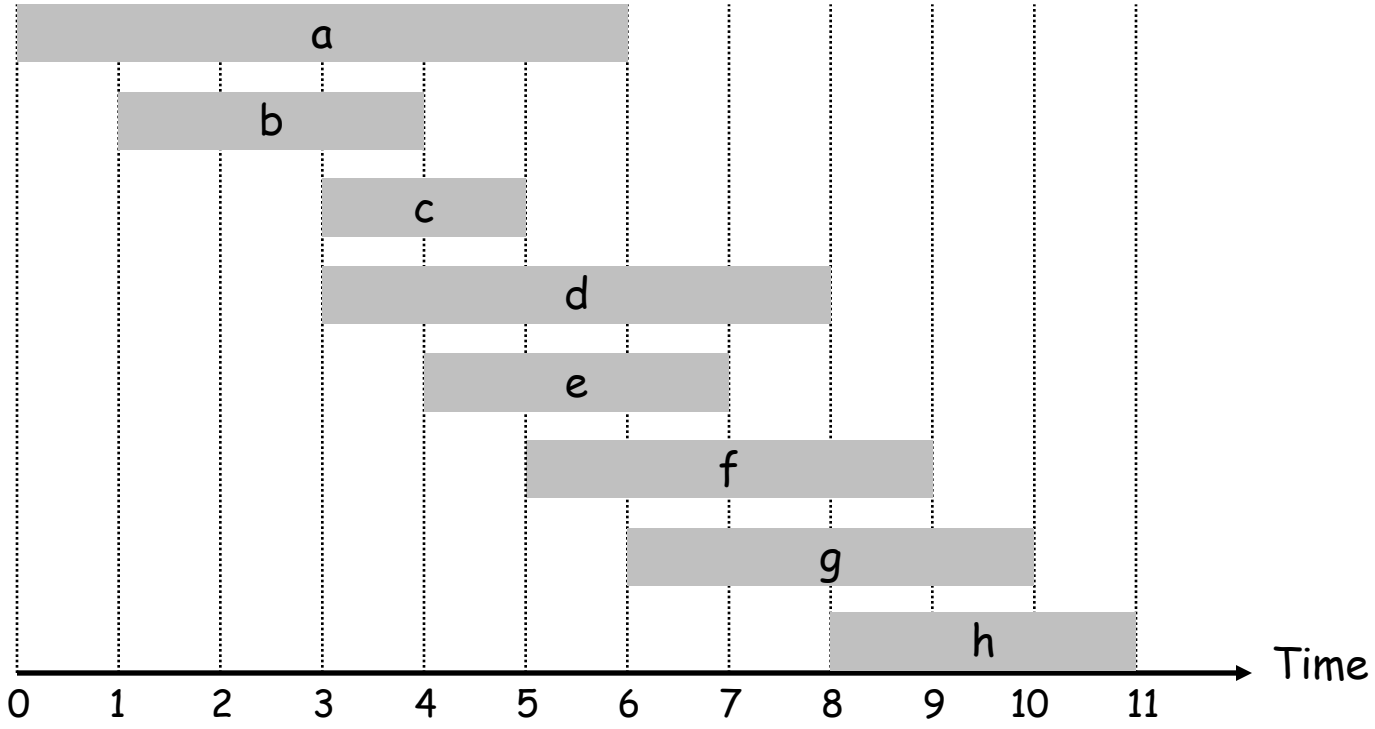
Outline

1. Interval Scheduling
2. Knapsack
3. Interval Partitioning

Interval Scheduling

Interval scheduling.

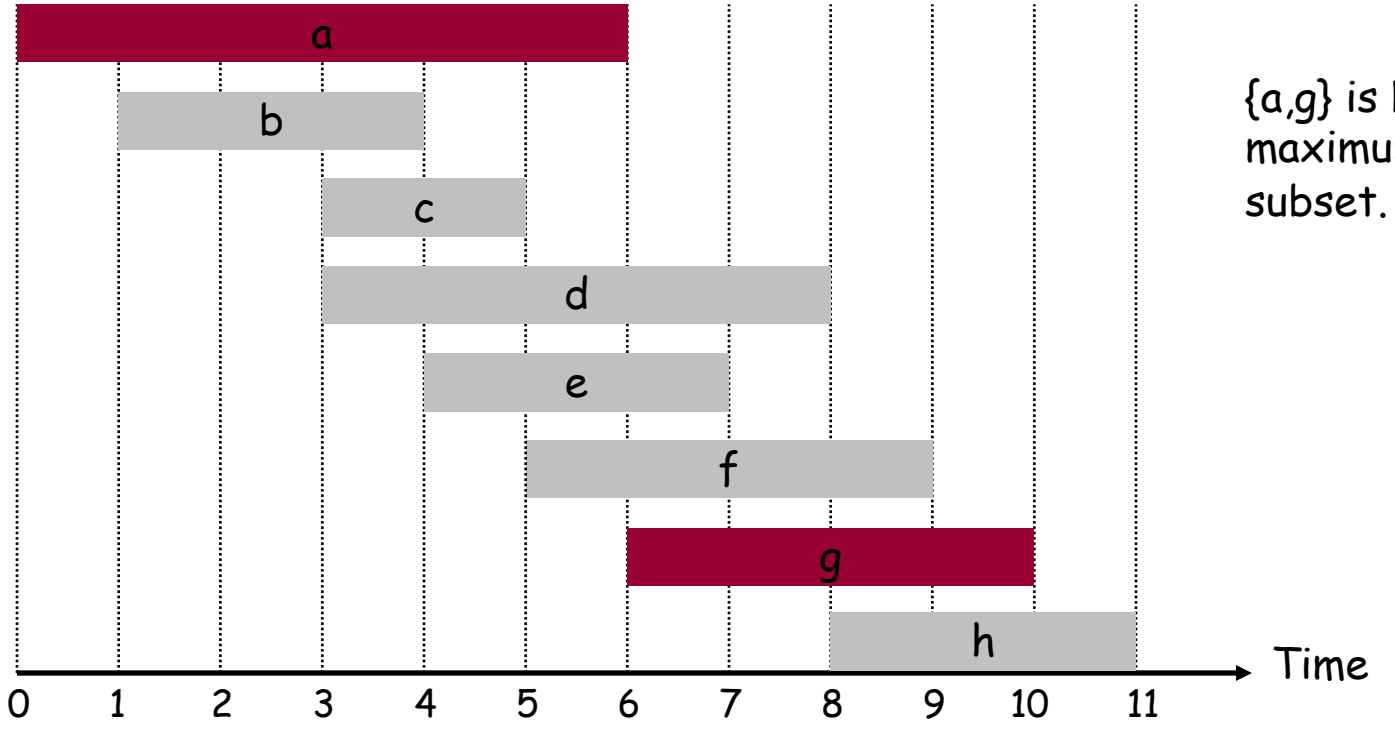
- Job j starts at s_j and finishes at f_j.
- Two jobs are compatible if they don't overlap.
- Goal: find maximum size subset of mutually compatible jobs.



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.

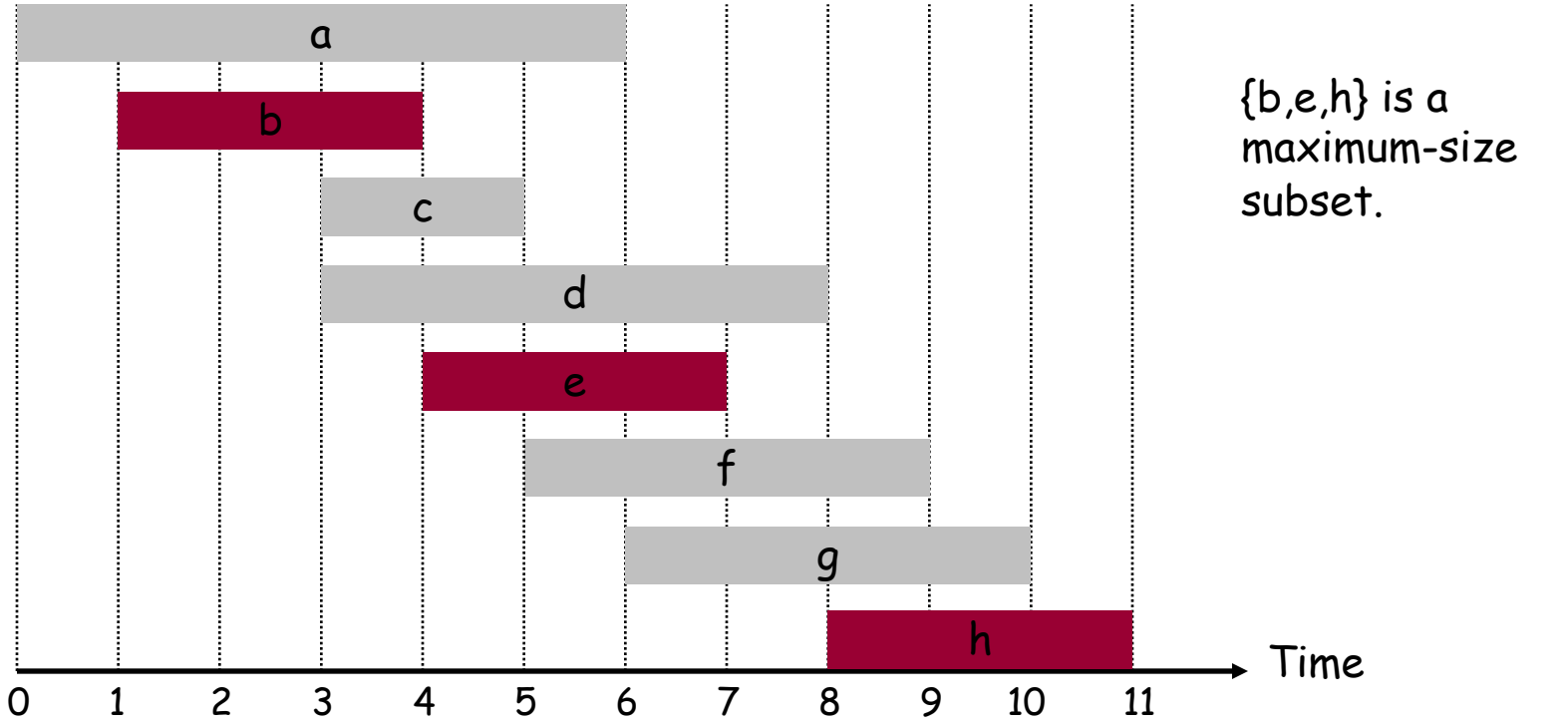


{a,g} is NOT a maximum-size subset.

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.



Interval Scheduling: Greedy Algorithms

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

Usually, many compatible jobs can co-exist.
Need to choose a rule specifying which job to choose next.
Three *possible* rules are:

- [Earliest start time]
  Consider jobs in increasing order of start time s_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.

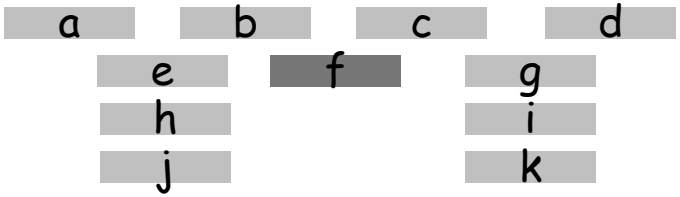
Take a job provided it's compatible with the ones already taken.

Order on earliest start time

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

Order on shortest interval

Chooses {c} instead of {a,b}



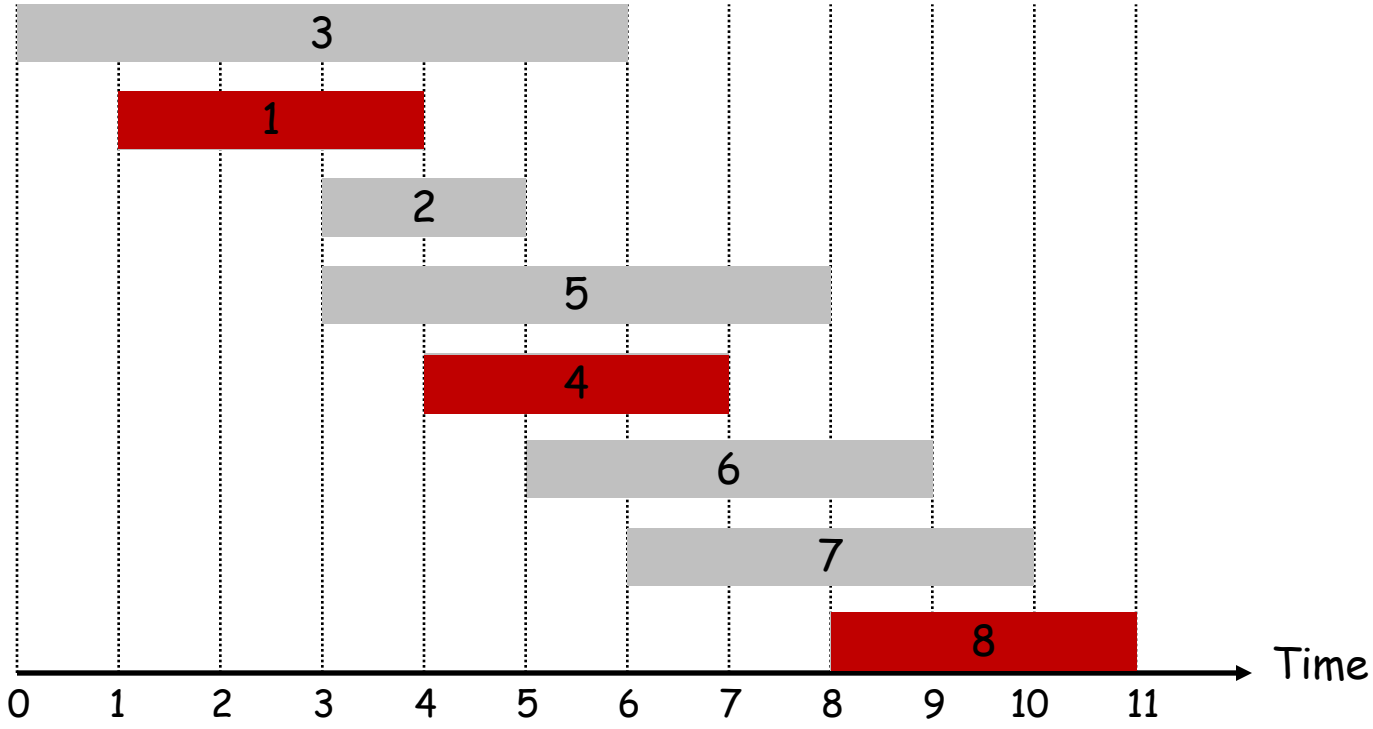
Order on fewest conflicts

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

Examples above provide ***counterexamples*** for the three rules proposed. For each of them, provides an example input for which following that rule yields a non-optimal (i.e., non max-size) schedule.

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



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.

Intuition: leaves maximum interval for scheduling the rest of the jobs.

```
Sort jobs by finish times so that f_1 ≤ f_2 ≤ ... ≤ f_n
A ← ∅,  last ← 0
for j ← 1 to n
    if s_j ≥ last then A ← A ∪ {j},  last ← f_j
return A
```

Running time dominated by cost of sorting: $\Theta(n \log n)$.

- 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.

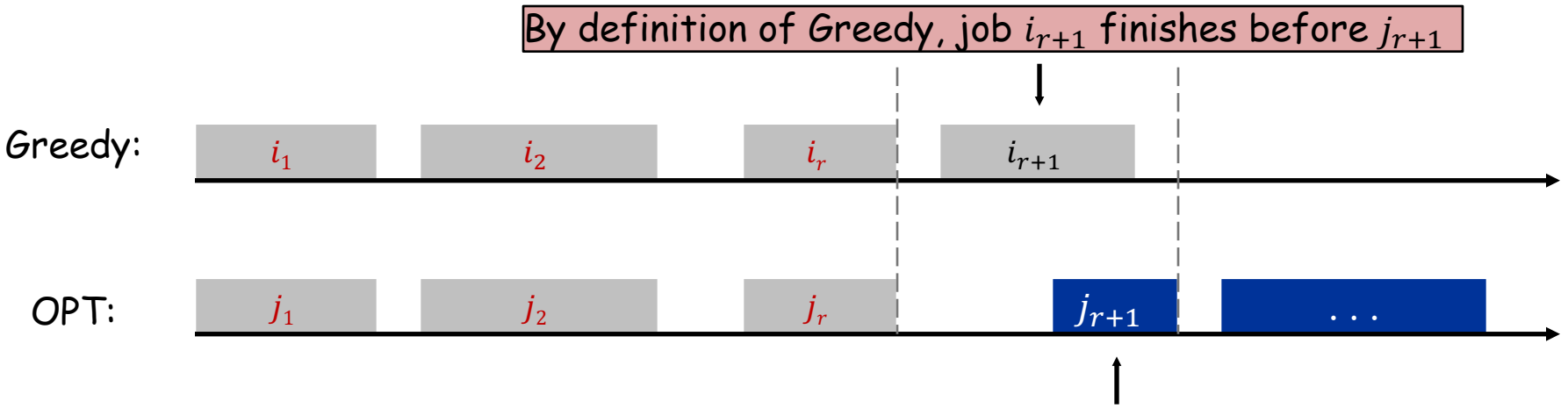
Need to prove!

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 $i_1 = j_1,\ i_2 = j_2,\ \ldots,\ i_r = j_r$



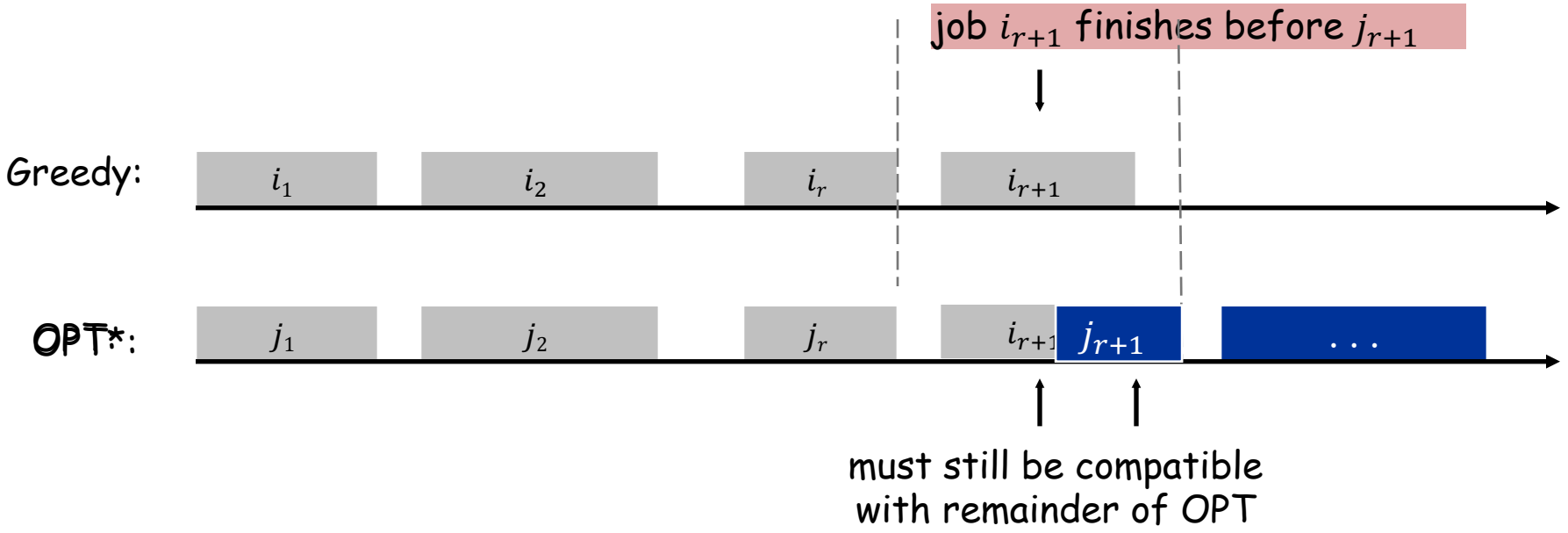
Interval Scheduling: Correctness

Theorem. Greedy algorithm is optimal.

Proof (Continued)

- Assume Greedy is different from OPT.
- Choose largest r such that $i_t = j_t$ for $t \leq r$ and $i_{r+1} \neq j_{r+1}$.
- Create OPT* from OPT by replacing j_{r+1} with i_{r+1}.

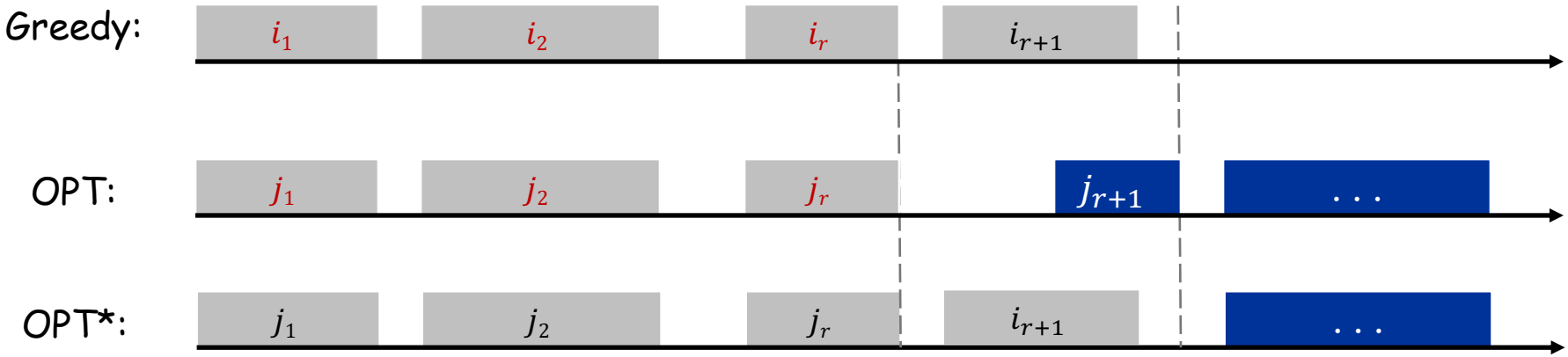
- OPT* is still a legal solution and has same size as OPT.
- $\Longrightarrow$ OPT* is also Optimal



Interval Scheduling: Correctness

Proof. So far

- Assumed Greedy≠OPT.
- Let r be such that OPT shares first r items with Greedy.
- Process creates new Optimal solution OPT*, which shares first r+1 items with Greedy



Can repeat this process starting with Greedy and (optimal) OPT*
Continue repeating this process until OPT is the same as greedy.

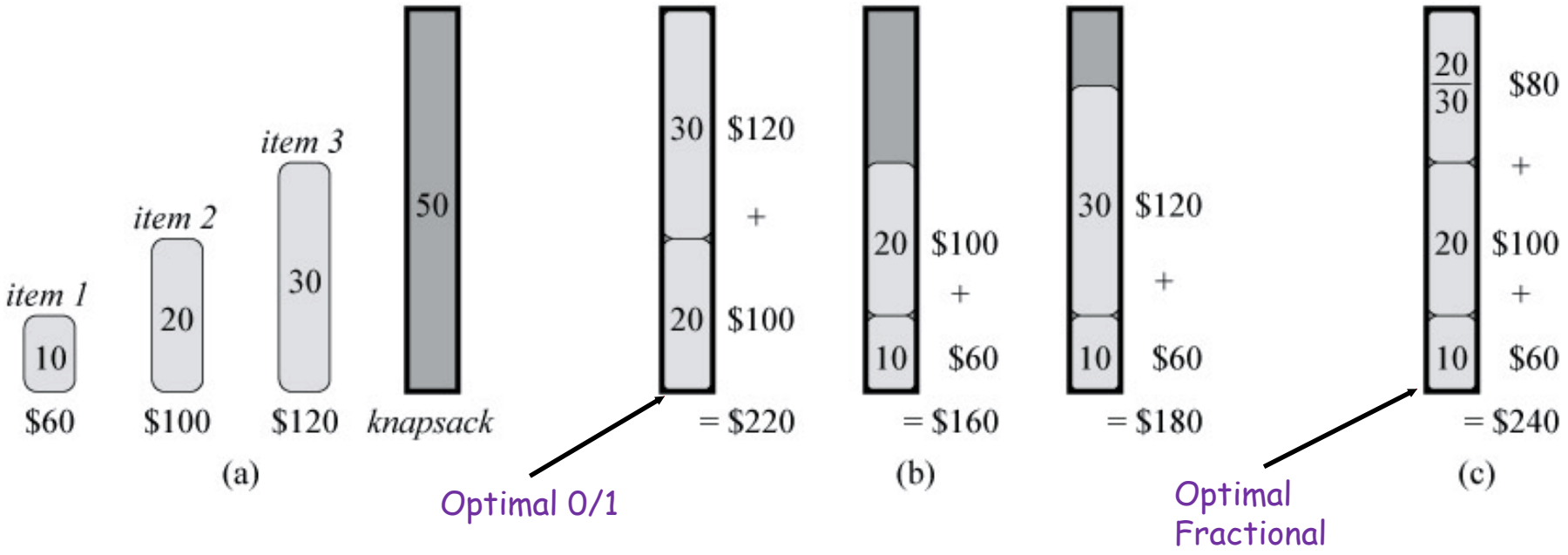
Important: Since cost remains the same, final solution we've created, which is Greedy, is optimal!

Finished!

Outline

1. Interval Scheduling
2. Knapsack
3. Interval Partitioning

The Fractional Knapsack Problem



Input: Set of n items: 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.

Greedy Algorithm: Correctness

Theorem: The greedy algorithm is optimal.

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

Let the greedy solution be $\boldsymbol{G = (x_1, x_2, \ldots, x_k, 0, \ldots, 0)}$

- Note: for $i < k,\ x_i = 1$; for $i > k,\ x_i = 0;\ 0 \leq x_k \leq 1$.

Consider any optimal solution $\boldsymbol{O = (y_1, y_2, \ldots, y_n)}$

- Note: Since both $\boldsymbol{G}$ and $\boldsymbol{O}$ must fully pack the knapsack,

$$\sum_{i=1}^{k} x_i\, w_i = W = \sum_{i=1}^{n} y_i\, w_i$$

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

$\boldsymbol{G} =$	x_1	x_2	x_3	…	x_{i-1}	x_i	…	x_k	…	0	0
$\boldsymbol{O} =$	x_1	x_2	x_3	…	x_{i-1}	y_i	…	y_k	…	y_{n-1}	y_n

By definition of greedy, $x_i > y_i$

- Let $x = x_i - y_i > 0$

Greedy Algorithm: Correctness (continued)

We now modify O as follows:

- Set $y_i \leftarrow x_i$ and remove amount of some items in $i+1$ to n of total weight xw_i
- This is always doable because in both O and G, the used total weight of items i to n is the same. $(\sum_{i=1}^{k} x_i\, w_i = W = \sum_{i=1}^{n} y_i\, w_i)$

After the modification:

- The total value of this new O has not decreased, since all the items $i+1$ to n have lesser or equal value-per-pound than item i
- This new O's value can not be greater than before, since O was already an optimal solution,
- => O's value stays the same => O is still an optimal solution
- O's first index that differs from G is now at least $i+1$

Repeating this process will eventually convert O into G
(1st index that differs always increases so the process must stop).
Note that process keeps solution value of O invariant (and optimal).

$\Longrightarrow$ at end $G = O$ $\Longrightarrow$ Greedy is optimal

Outline

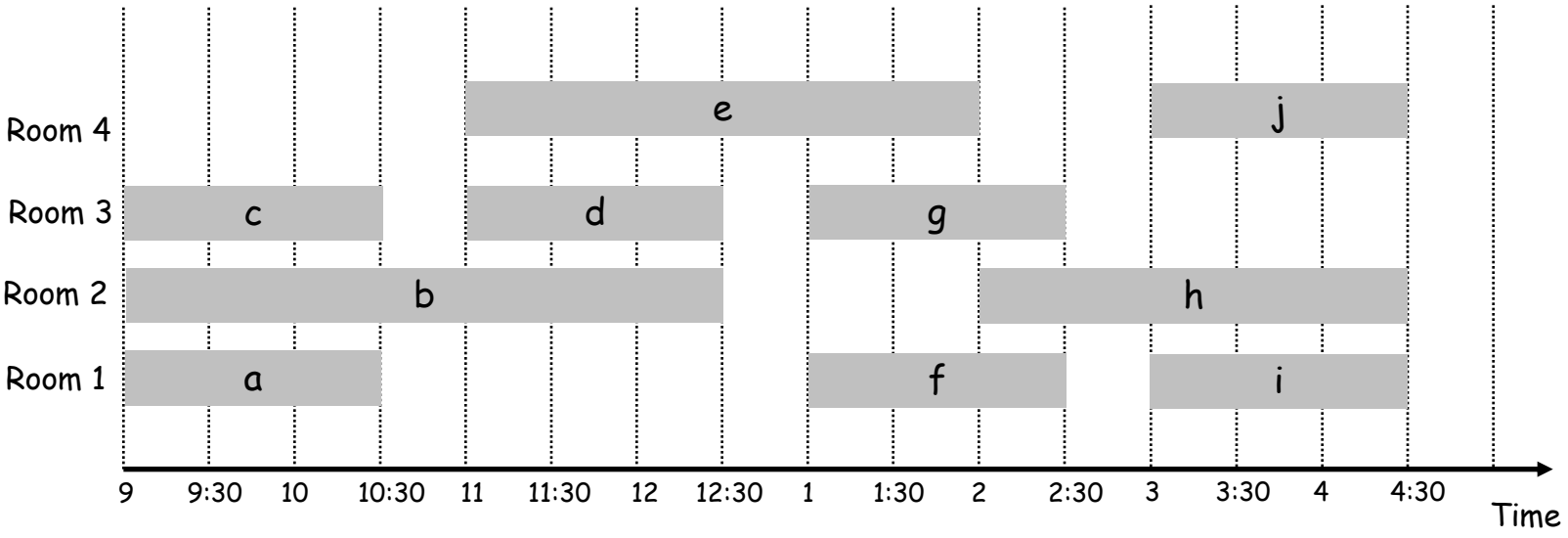
1. Interval Scheduling
2. Knapsack
3. Interval Partitioning

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.

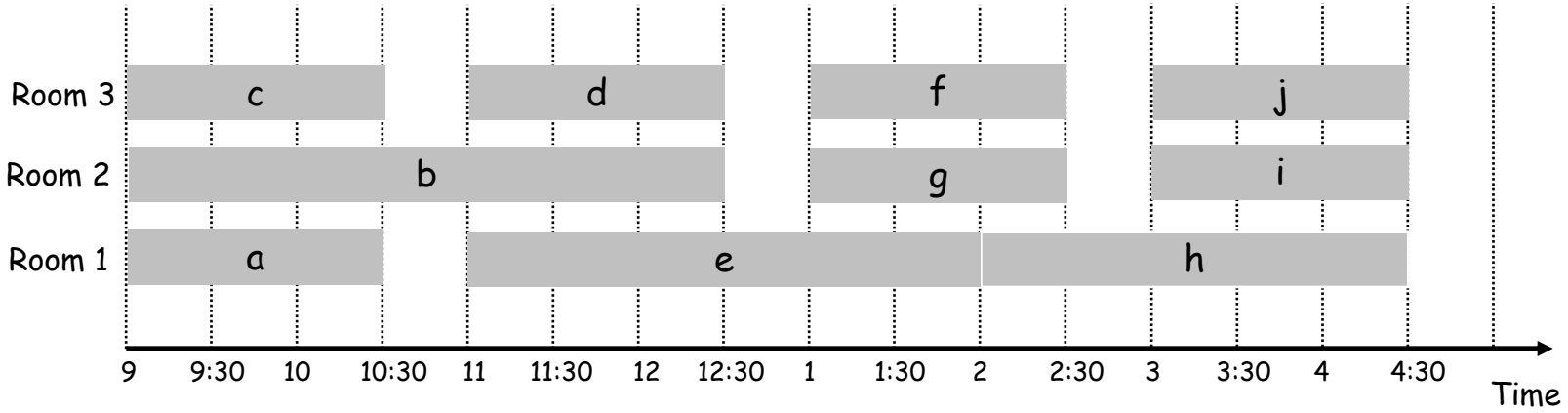


Interval Partitioning

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 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
```

Greedy! It only opens a new classroom if it is needed.

Need to prove optimality.
More specifically, need to show that processing the lectures **ordered by starting time** implies optimality.

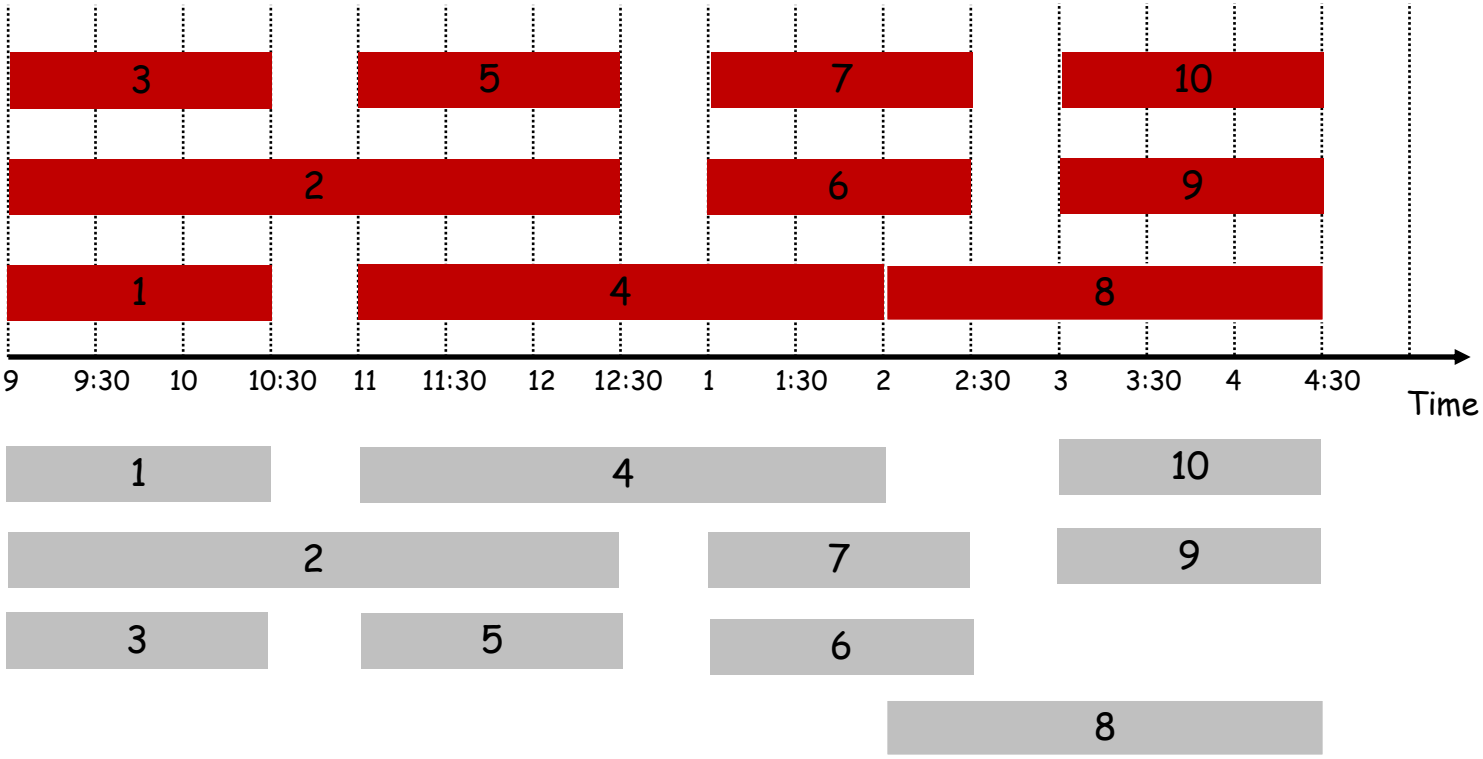
Convince yourself that processing by finishing time does NOT yield optimal solution!

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

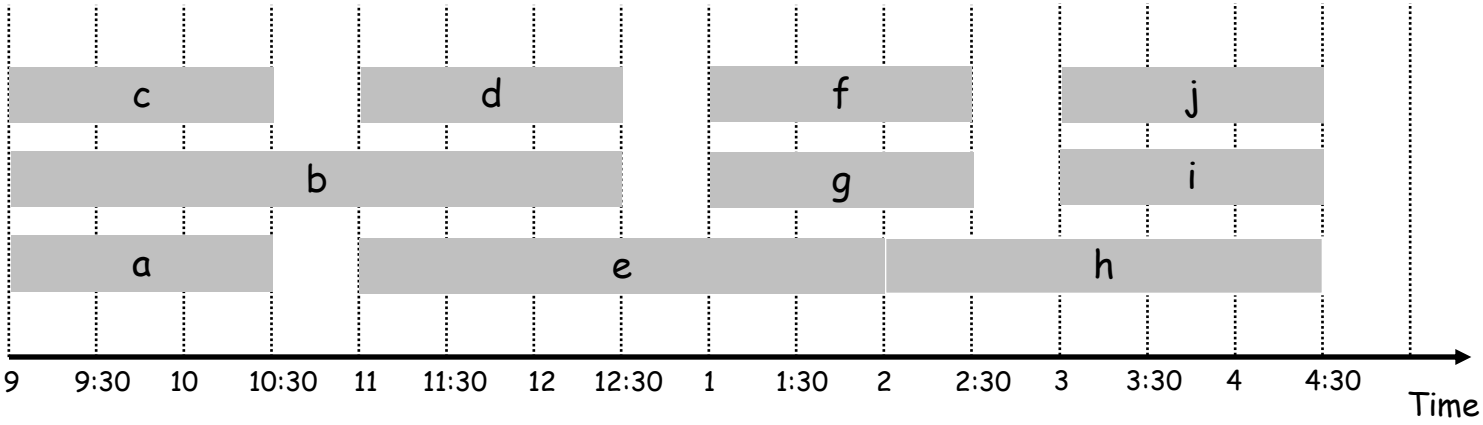
Def. The depth of a set of open intervals is the maximum number that exist at any instant of time.

Intuition: At each second of the day keep track of how many simultaneous classes are being taught at that time. The depth is the maximum number you have seen.

Key observation. Minimum 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 before (or at same time as) s_j and finish later 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 $\geq d$.
- Minimum number of classrooms needed $\geq$ depth $\geq d$, Since Greedy uses only d, Greedy is optimal.

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
```

- 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 (*) uses $O(n)$ 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 min finishing time f_{min} in the priority queue and check whether $f_{min} \leq s_j$

$O(\log n)$ - To implement (**) just add the new finishing time f_j to p. queue

$O(\log n)$ - To implement (***) re-insert f_{min} back into p. queue AND insert the new finishing time f_j into p. queue