Data Structures and Algorithms
CMPSC 465

LECTURES 17-19

Greedy Algorithms

- Interval Scheduling
- Interval Partitioning
- Scheduling for Maximum Lateness

Adam Smith

Design paradigms so far

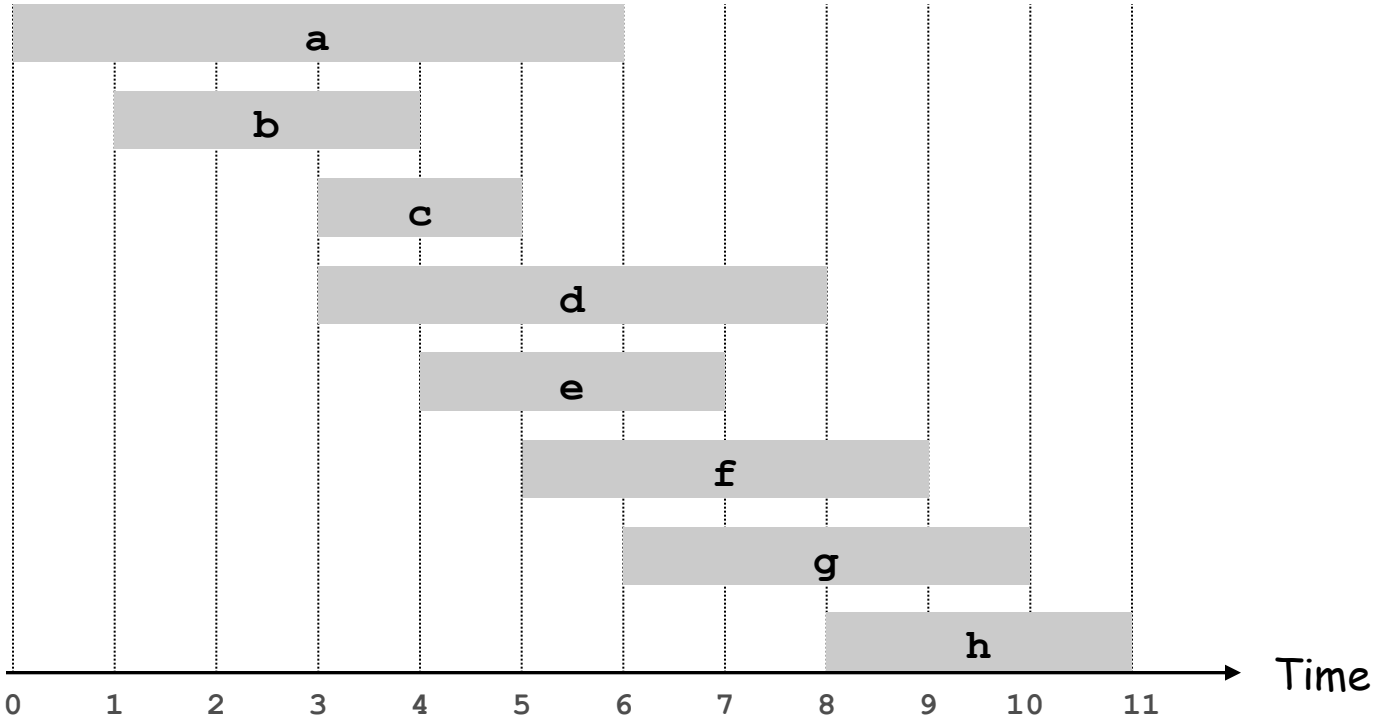
- Simple, iterative algorithms
  - Insertion sort
- Recursion-based algorithms
  - Divide and conquer
  - Backtracking
  - Dynamic programming
- Today: greedy algorithms
  - Simple, iterative structure: at each phase, "grab" make the best choice available, and never look back.
  - Typically very tricky to analyze
  - Easy to get wrong!!!

Weighted Interval Scheduling

- Weighted interval scheduling problem.
  - Job j starts at s_j, finishes at f_j, and has weight or value v_j .
  - Two jobs compatible if they don't overlap.
  - Goal: find maximum weight subset of mutually compatible jobs.
- Dynamic programming algorithm:
  - Either item j is in or out…
- Can we find a simpler algorithm for the "unweighted" case? (All weights equal)

Unweighted Interval Scheduling Problem

- Job j starts at s_j and finishes at f_j.
- Two jobs are **compatible** if they do not overlap.
- **Find:** maximum subset of mutually compatible jobs.



Greedy: Counterexamples

for earliest start time

for shortest interval

for fewest conflicts

Formulating Algorithm

- Arrays of start and finishing times
  - $s_1, s_2, \ldots, s_n$
  - $f_1, f_2, \ldots, f_n$

- Input length?
  - $2n = \Theta(n)$

Greedy Algorithm

- **Earliest finish time:** ascending order of f_j.

```
Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn.

A ← φ    M Set of selected jobs
for j = 1 to n {
   if (job j compatible with A)
      A ← A ∪ {j}
}
return A
```

- Implementation. **O(n log n) time; O(n) space.**
  - Remember job j^* that was added last to A.
  - Job j is compatible with A if $s_j \geq f_{j^*}$.

Running time: O(n log n)

O(n log n)

O(1)

n × O(1)

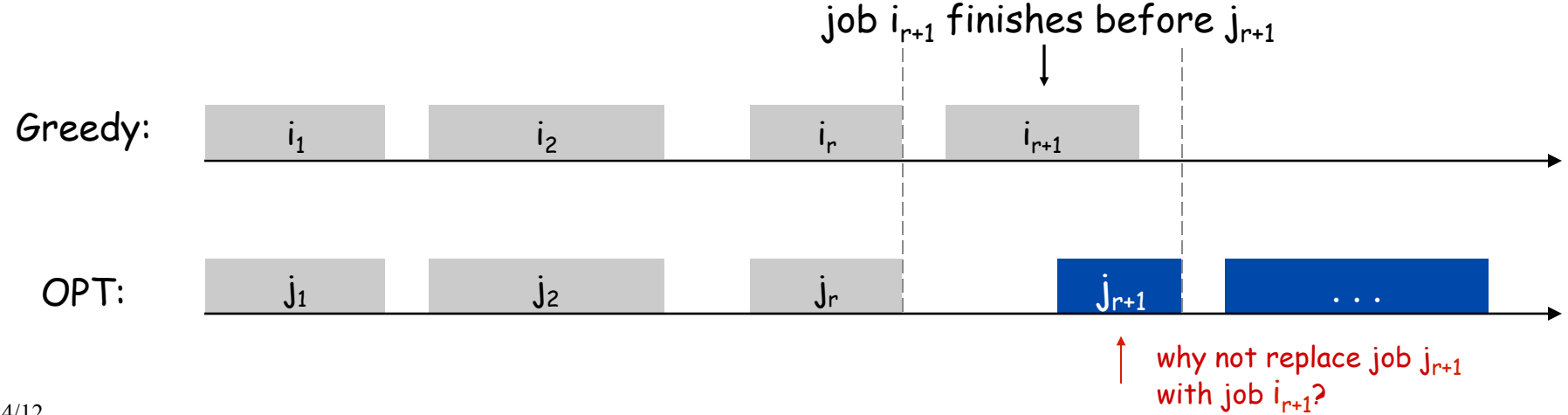
```
Sort jobs by finish times so that
      f_1 ≤ f_2 ≤ ... ≤ f_n.

A ← (empty)    M Queue of selected jobs
j* ← 0
for j = 1 to n {
   if (f_j* <= s_j)
      enqueue(j onto A)
}
return A
```

Analysis: Greedy Stays Ahead

- Theorem. Greedy algorithm is optimal.
- Proof strategy (by contradiction):
  - Suppose greedy solution G is not optimal.
  - Consider an optimal strategy… which one?
    - Consider the optimal strategy OPT that agrees with the greedy strategy for as many initial jobs as possible
    - Look at first place in list where OPT differs from greedy strategy
  - Show a new optimal strategy that agrees more with the greedy strategy

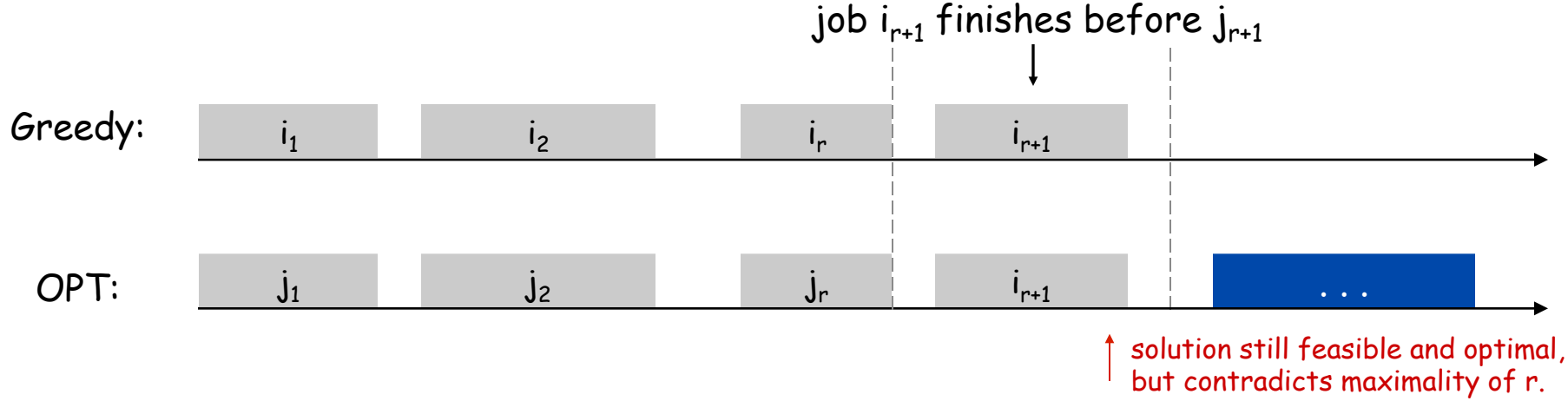
Analysis: Greedy Stays Ahead

- Theorem. Greedy algorithm is optimal.
- Pf (by contradiction): Suppose greedy is not optimal.
  - Let $i_1, i_2, \ldots i_k$ denote set of jobs selected by greedy.
  - Let $j_1, j_2, \ldots j_m$ be 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.



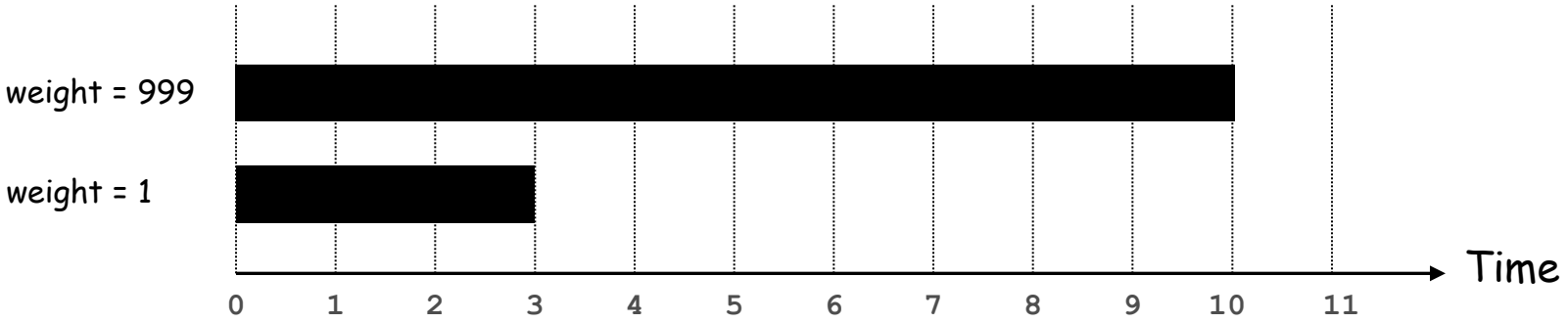
Analysis: Greedy Stays Ahead

- Theorem. Greedy algorithm is optimal.
- Pf (by contradiction): Suppose greedy is not optimal.
  - Let $i_1, i_2, \ldots i_k$ denote set of jobs selected by greedy.
  - Let $j_1, j_2, \ldots j_m$ be 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.



Weighted vs unweighted

- **Recall**: Greedy algorithm works if all weights are 1.
  - Consider jobs in ascending order of finish time.
  - Add job to subset if it is compatible with previously chosen jobs.

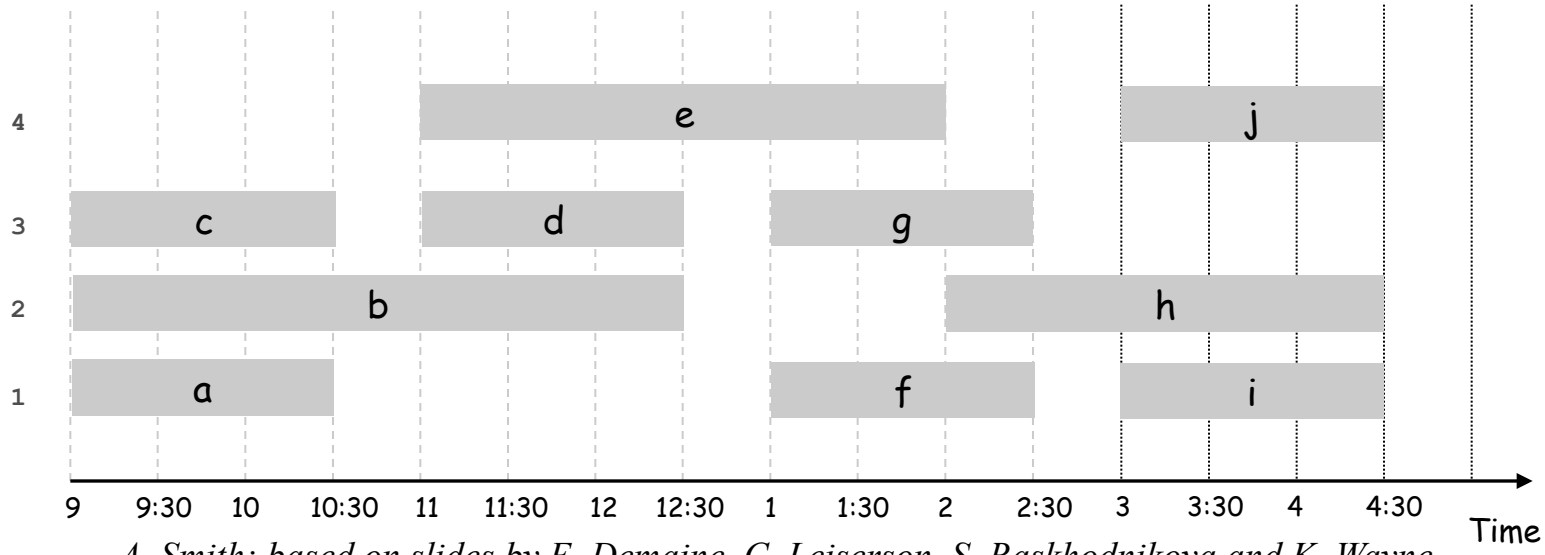


- **Observation**: Greedy algorithm can fail spectacularly if arbitrary weights are allowed.

Interval Partitioning

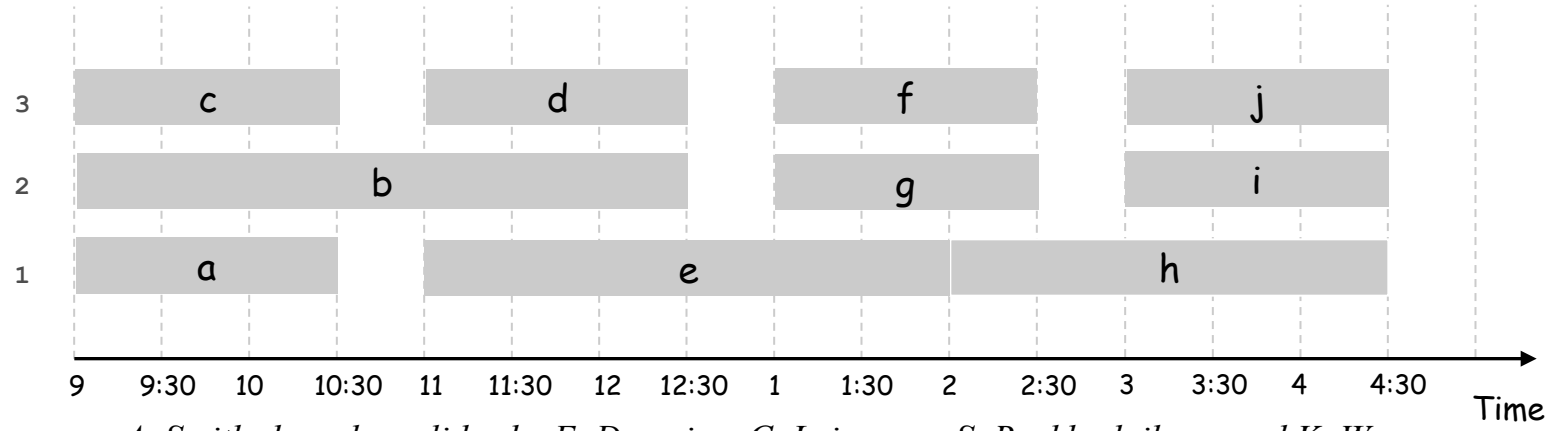
Interval Partitioning Problem

- Lecture j starts at s_j and finishes at f_j.
- **Find**: minimum number of classrooms to schedule all lectures so that no two occur at the same time in the same room.
- **E.g.**: 10 lectures are scheduled in **4** classrooms.



Interval Partitioning

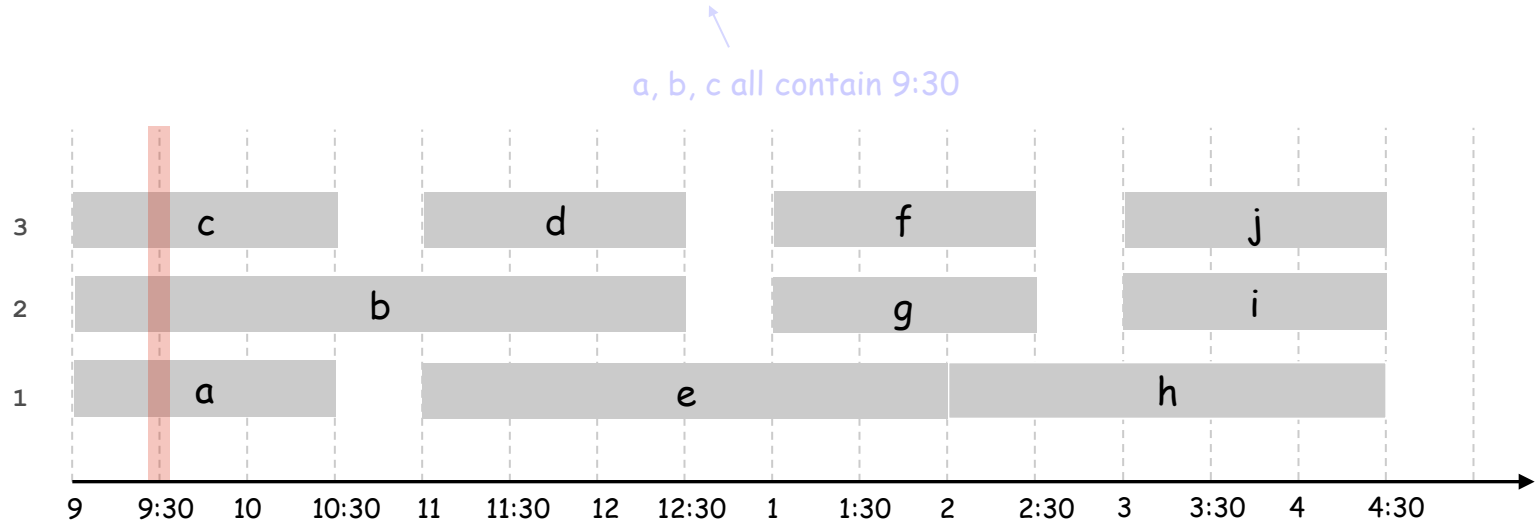
- Lecture j starts at s_j and finishes at f_j.
- **Find**: minimum number of classrooms to schedule all lectures so that no two occur at the same time in the same room.
- **E.g.**: Same lectures are scheduled in **3** classrooms.



A. Smith; based on slides by E. Demaine, C. Leiserson, S. Raskhodnikova and K. Wayne

Lower Bound

- **Definition**. The **depth** of a set of open intervals is the maximum number that contain any given time.
- **Key observation**. Number of classrooms needed ≥ depth.
- **E.g.:** Depth of this schedule = 3 ⇒ this schedule is optimal.



- **Q:** Is it always sufficient to have number of classrooms = depth?

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   M Number of allocated classrooms
for j = 1 to n {
   if (lecture j is compatible with some classroom k)
      schedule lecture j in classroom k
   else
      allocate a new classroom d + 1
      schedule lecture j in classroom d + 1
      d ← d + 1
}
```

- Implementation. O(n log n) time; O(n) space.
  - For each classroom, maintain the finish time of the last job added.
  - Keep the classrooms in a priority queue (main loop n log(d) time)

Analysis: Structural Argument

- **Observation**. Greedy algorithm never schedules two incompatible lectures in the same classroom.
- **Theorem**. Greedy algorithm is optimal.
- **Proof:** Let d = number of classrooms allocated by greedy.
  - Classroom d is opened because we needed to schedule a lecture, say j, that is incompatible with all d-1 last lectures in other classrooms.
  - These d lectures each end after s_j.
  - Since we sorted by start time, they start no later than s_j.
  - Thus, we have d lectures overlapping at time $s_j + \varepsilon$.
  - Key observation $\Rightarrow$ all schedules use $\geq$ d classrooms. ▪

Scheduling to minimize lateness

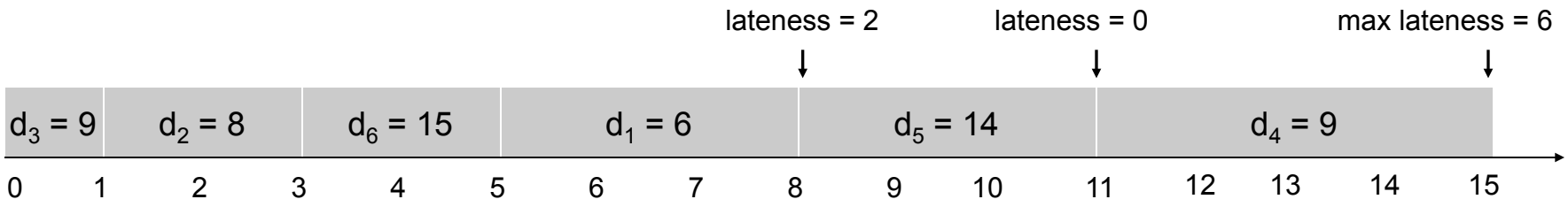
Scheduling to Minimizing Lateness

Minimizing lateness problem.

- Single resource processes one job at a time.
- Job j requires tj units of processing time and is due at time dj.
- If j starts at time sj, it finishes at time fj = sj + tj.
- Lateness: ℓj = max { 0, fj - dj }.
- Goal: schedule all jobs to minimize maximum lateness L = max ℓj.

Ex:

	1	2	3	4	5	6
t_j	3	2	1	4	3	2
d_j	6	8	9	9	14	15



Minimizing Lateness: Greedy Algorithms

Greedy template. Consider jobs in some order.

- [Shortest processing time first] Consider jobs in ascending order of processing time tj.

- [Earliest deadline first] Consider jobs in ascending order of deadline dj.

- [Smallest slack] Consider jobs in ascending order of slack dj - tj.

Minimizing Lateness: Greedy Algorithms

Greedy template. Consider jobs in some order.

- [Shortest processing time first] Consider jobs in ascending order of processing time tj.

	1	2
t_j	1	10
d_j	100	10

counterexample

- [Smallest slack] Consider jobs in ascending order of slack dj - tj.

	1	2
t_j	1	10
d_j	2	10

counterexample

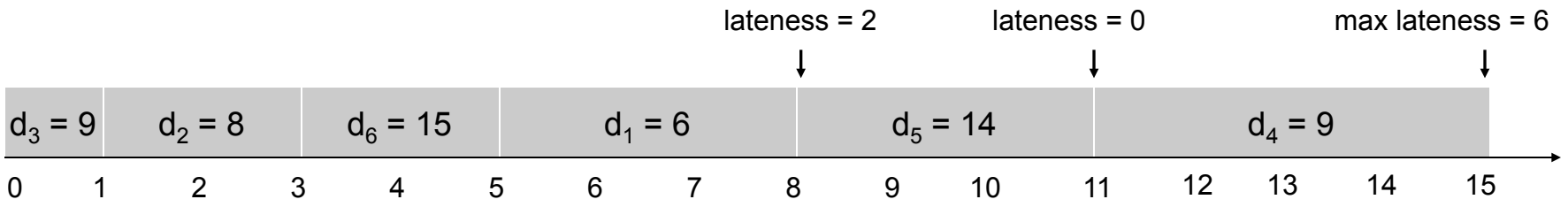
Scheduling to Minimizing Lateness

Minimizing lateness problem.

- Single resource processes one job at a time.
- Job j requires t_j units of processing time and is due at time d_j.
- If j starts at time s_j, it finishes at time $f_j = s_j + t_j$.
- Lateness: $\ell_j = \max \{ 0, f_j - d_j \}$.
- Goal: schedule all jobs to minimize maximum lateness $L = \max \ell_j$.

Ex:

	1	2	3	4	5	6
t_j	3	2	1	4	3	2
d_j	6	8	9	9	14	15



Minimizing Lateness: Greedy Algorithms

Greedy template. Consider jobs in some order.

- [Shortest processing time first] Consider jobs in ascending order of processing time t_j.

- [Earliest deadline first] Consider jobs in ascending order of deadline d_j.

- [Smallest slack] Consider jobs in ascending order of slack $d_j - t_j$.

Minimizing Lateness: Greedy Algorithms

Greedy template. Consider jobs in some order.

- [Shortest processing time first] Consider jobs in ascending order of processing time t_j.

	1	2
t_j	1	10
d_j	100	10

counterexample

- [Smallest slack] Consider jobs in ascending order of slack $d_j - t_j$.

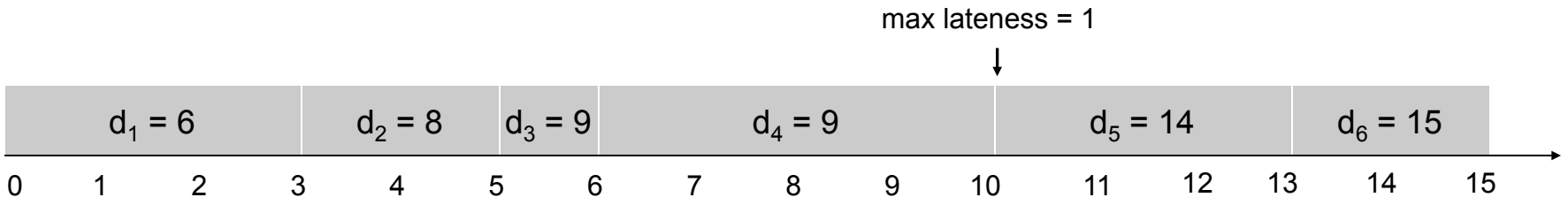
	1	2
t_j	1	10
d_j	2	10

counterexample

Minimizing Lateness: Greedy Algorithm

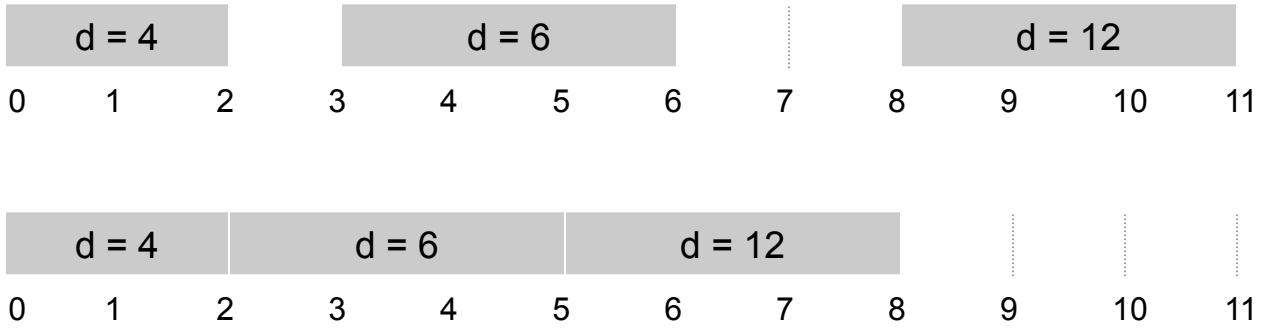
Greedy algorithm. Earliest deadline first.

```
Sort n jobs by deadline so that d_1 ≤ d_2 ≤ …
≤ d_n

t ← 0
for j = 1 to n
   Assign job j to interval [t, t + t_j]
   s_j ← t, f_j ← t + t_j
   t ← t + t_j
output intervals [s_j, f_j]
```



Minimizing Lateness: No Idle Time

Observation. There exists an optimal schedule with no idle time.



Observation. The greedy schedule has no idle time.

Minimizing Lateness: Inversions

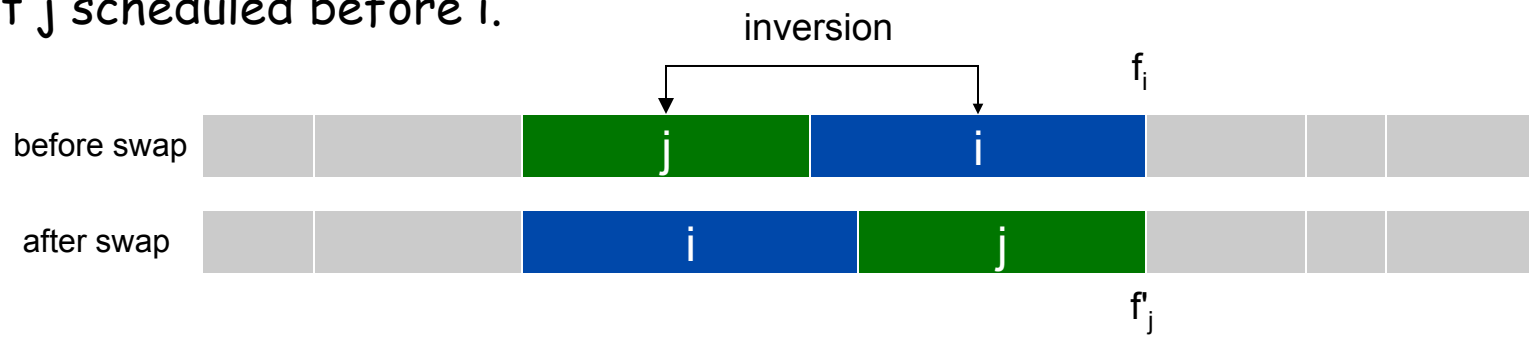
Def. An inversion in schedule S is a pair of jobs i and j such that: i < j but j scheduled before i.

Observation. Greedy schedule has no inversions.

Observation. If a schedule (with no idle time) has an inversion, it has one with a pair of inverted jobs scheduled consecutively.

Minimizing Lateness: Inversions

Def. An inversion in schedule S is a pair of jobs i and j such that: i < j but j scheduled before i.



Claim. Swapping two adjacent, inverted jobs reduces the number of inversions by one and does not increase the max lateness.

Pf. Let ℓ be the lateness before the swap, and let ℓ' be it afterwards.

- $\ell'_k = \ell_k$ for all $k \neq i, j$
- $\ell'_i \leq \ell_i$
- If job j is late:

$$
\begin{aligned}
\ell'_j &= f'_j - d_j && \text{(definition)} \\
&= f_i - d_j && (j \text{ finishes at time } f_i) \\
&\leq f_i - d_i && (i < j) \\
&\leq \ell_i && \text{(definition)}
\end{aligned}
$$

Minimizing Lateness: Analysis of Greedy Algorithm

Theorem. Greedy schedule S is optimal.

Pf. Define S* to be an optimal schedule that has the fewest number of inversions, and let's see what happens.

- Can assume S* has no idle time.
- If S* has no inversions, then S = S*.
- If S* has an inversion, let i-j be an adjacent inversion.
  - swapping i and j does not increase the maximum lateness and strictly decreases the number of inversions
  - this contradicts definition of S* ▪

Greedy Analysis Strategies

Greedy algorithm stays ahead. Show that after each step of the greedy algorithm, its solution is at least as good as any other algorithm's.

Exchange argument. Gradually transform any solution to the one found by the greedy algorithm without hurting its quality.

Structural. Discover a simple "structural" bound asserting that every possible solution must have a certain value. Then show that your algorithm always achieves this bound.