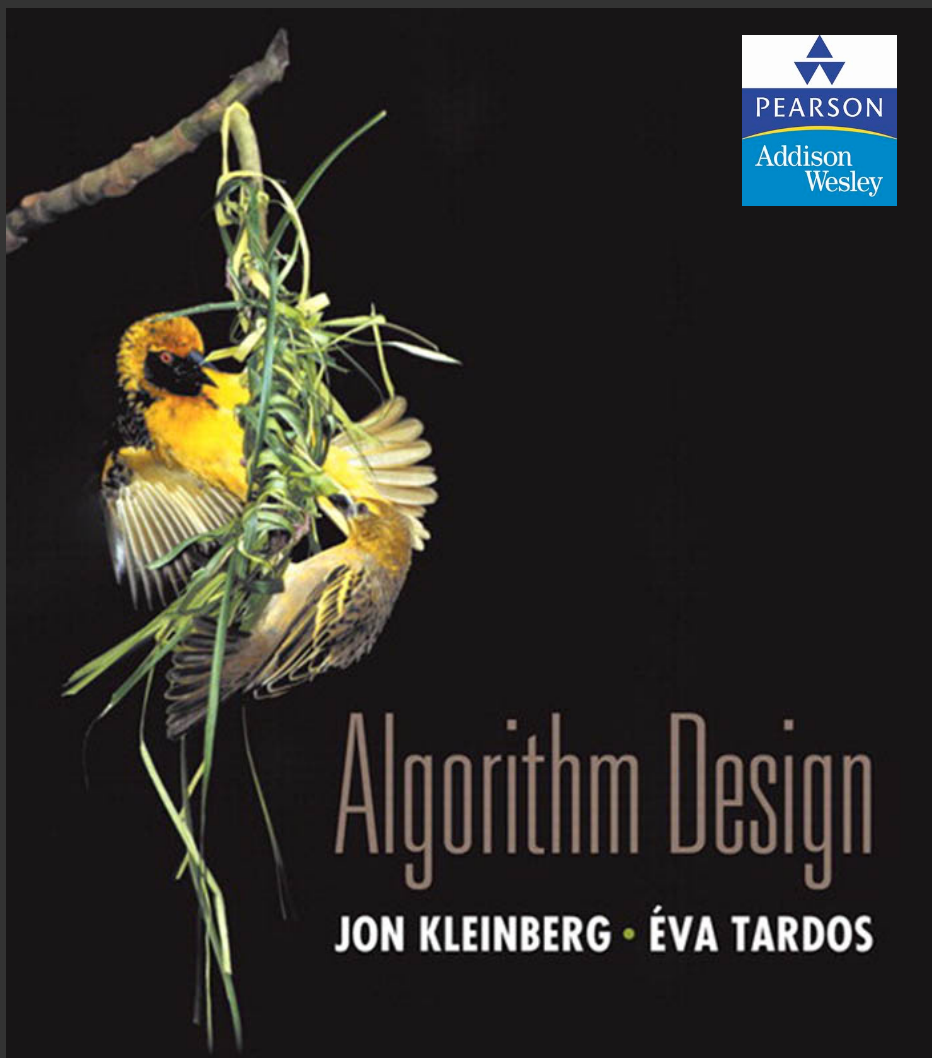


Greedy Algorithms I

- *coin changing*
- *interval scheduling*
- *interval partitioning*
- *scheduling to minimize lateness*
- *optimal caching*

Lecture slides by Kevin Wayne

http://www.cs.princeton.edu/~wayne/kleinberg-tardos

Last updated on 25/11/2019 16:23

Greedy Algorithms I

- *coin changing*
- *interval scheduling*
- *interval partitioning*
- *scheduling to minimize lateness*
- *optimal caching*

Coin changing

Goal. Given U. S. currency denominations { 1, 5, 10, 25, 100 }, devise a method to pay amount to customer using fewest coins.

Ex. 34¢.

Cashier's algorithm. At each iteration, add coin of the largest value that does not take us past the amount to be paid.

Ex. $2.89.

Cashier's algorithm

At each iteration, add coin of the largest value that does not take us past the amount to be paid.

CASHIERS-ALGORITHM $(x, c_1, c_2, \ldots, c_n)$

SORT n coin denominations so that $0 < c_1 < c_2 < \ldots < c_n$.

$S \leftarrow \varnothing$. ← multiset of coins selected

WHILE $(x > 0)$

  $k \leftarrow$ largest coin denomination c_k such that $c_k \leq x$.

  IF (no such k)

    RETURN *"no solution."*

  ELSE

    $x \leftarrow x - c_k$.

    $S \leftarrow S \cup \{ k \}$.

RETURN S.

Is the cashier's algorithm optimal?

A. Yes, greedy algorithms are always optimal.

B. Yes, for any set of coin denominations $c_1 < c_2 < \ldots < c_n$ provided $c_1 = 1$.

C. Yes, because of special properties of U.S. coin denominations.

D. No.

Cashier's algorithm (for arbitrary coin denominations)

Q. Is cashier's algorithm optimal for any set of denominations?

A. No. Consider U.S. postage: 1, 10, 21, 34, 70, 100, 350, 1225, 1500.

- Cashier's algorithm: 140¢ = 100 + 34 + 1 + 1 + 1 + 1 + 1 + 1.
- Optimal: 140¢ = 70 + 70.

A. No. It may not even lead to a feasible solution if $c_1 > 1$: 7, 8, 9.

- Cashier's algorithm: 15¢ = 9 + ?.
- Optimal: 15¢ = 7 + 8.

Properties of any optimal solution (for U.S. coin denominations)

Property. Number of pennies ≤ 4.

Pf. Replace 5 pennies with 1 nickel.

Property. Number of nickels ≤ 1.

Property. Number of quarters ≤ 3.

Property. Number of nickels + number of dimes ≤ 2.

Pf.

- Recall: ≤ 1nickel.
- Replace 3 dimes and 0 nickels with 1 quarter and 1 nickel;
- Replace 2 dimes and 1 nickel with 1 quarter.

dollars (100¢) **quarters** (25¢) **dimes** (10¢) **nickels** (5¢) **pennies** (1¢)

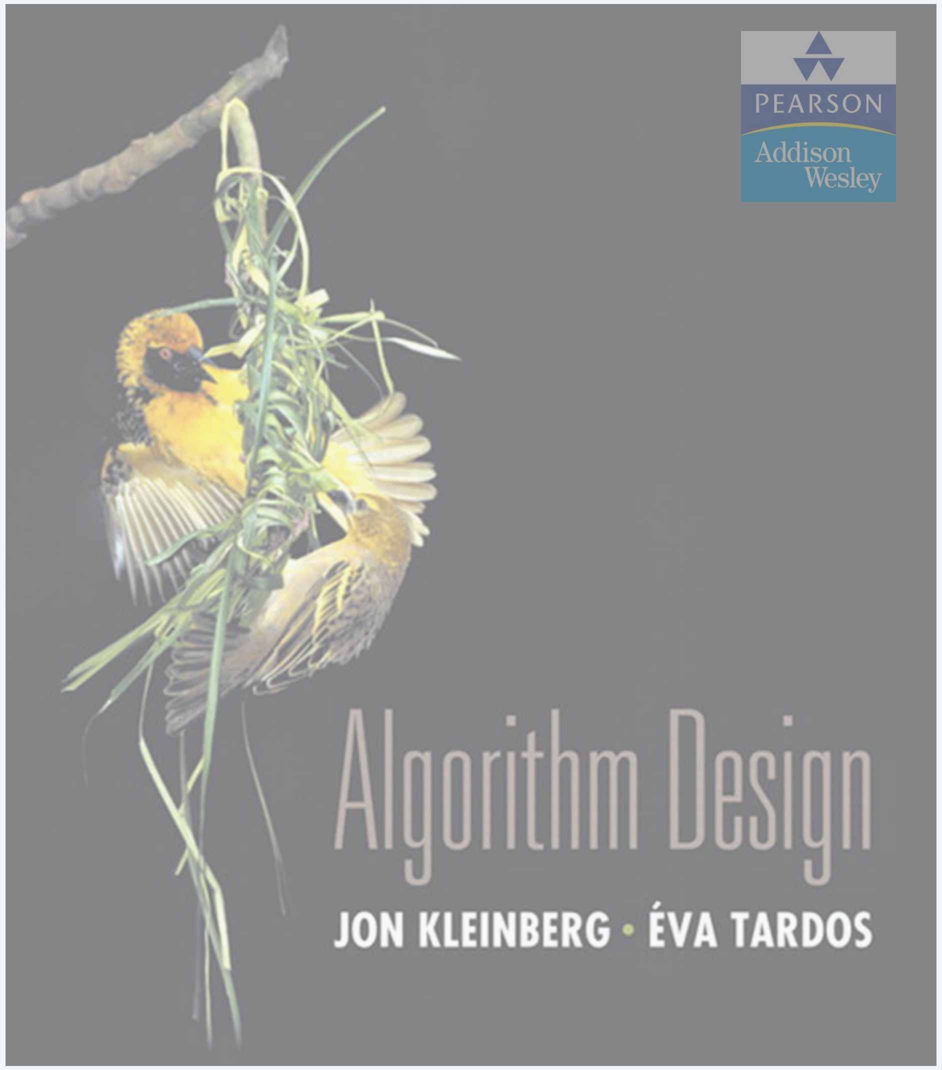
Optimality of cashier's algorithm (for U.S. coin denominations)

Theorem. Cashier's algorithm is optimal for U.S. coins { 1, 5, 10, 25, 100 }.

Pf. [by induction on amount to be paid x]

- Consider optimal way to change $c_k \le x < c_{k+1}$: greedy takes coin k.
- We claim that any optimal solution must take coin k.
  - if not, it needs enough coins of type $c_1, \ldots, c_{k-1}$ to add up to x
  - table below indicates no optimal solution can do this
- Problem reduces to coin-changing $x - c_k$ cents, which, by induction, is optimally solved by cashier's algorithm. ▪

k	c_k	all optimal solutions must satisfy	max value of coin denominations $c_1, c_2, \ldots, c_{k-1}$ in any optimal solution
1	1	$P \le 4$	–
2	5	$N \le 1$	4
3	10	$N + D \le 2$	4 + 5 = 9
4	25	$Q \le 3$	20 + 4 = 24
5	100	*no limit*	75 + 24 = 99



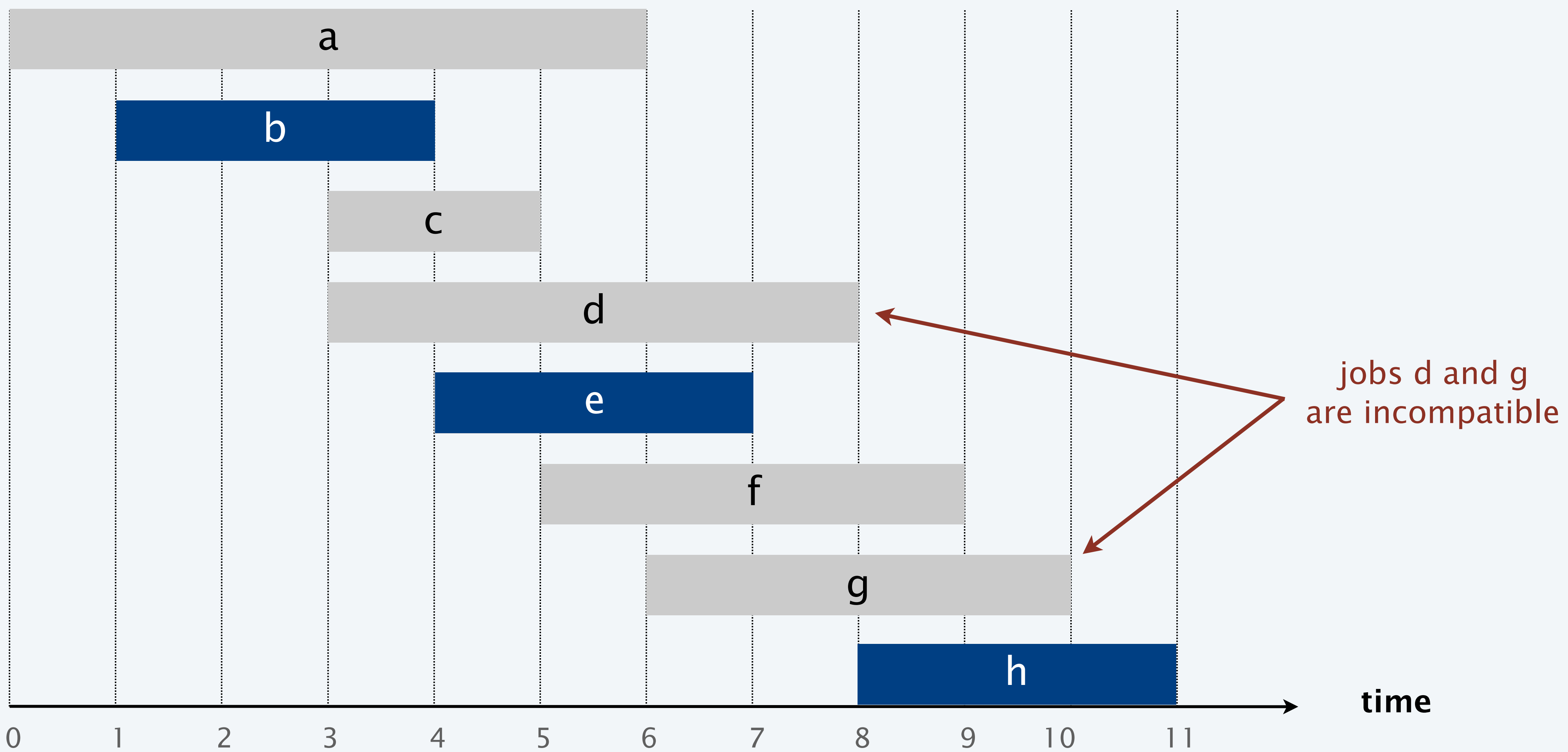
SECTION 4.1

GREEDY ALGORITHMS I

- coin changing
- *interval scheduling*
- interval partitioning
- scheduling to minimize lateness
- optimal caching

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 subset of mutually compatible jobs.



Greedy algorithms I: quiz 2

Consider jobs in some order, taking each job provided it's compatible with the ones already taken. Which rule is optimal?

A. [Earliest start time] Consider jobs in ascending order of s_j.

B. [Earliest finish time] Consider jobs in ascending order of f_j.

C. [Shortest interval] Consider jobs in ascending order of $f_j - s_j$.

D. None of the above.

Interval scheduling: earliest-finish-time-first algorithm

EARLIEST-FINISH-TIME-FIRST $(n, s_1, s_2, \ldots, s_n, f_1, f_2, \ldots, f_n)$

SORT jobs by finish times and renumber so that $f_1 \leq f_2 \leq \ldots \leq f_n$.

$S \leftarrow \varnothing$. ← set of jobs selected

FOR $j = 1$ TO n

  IF (job j is compatible with S)

    $S \leftarrow S \cup \{ j \}$.

RETURN S.

Proposition. Can implement earliest-finish-time first in $O(n \log n)$ time.

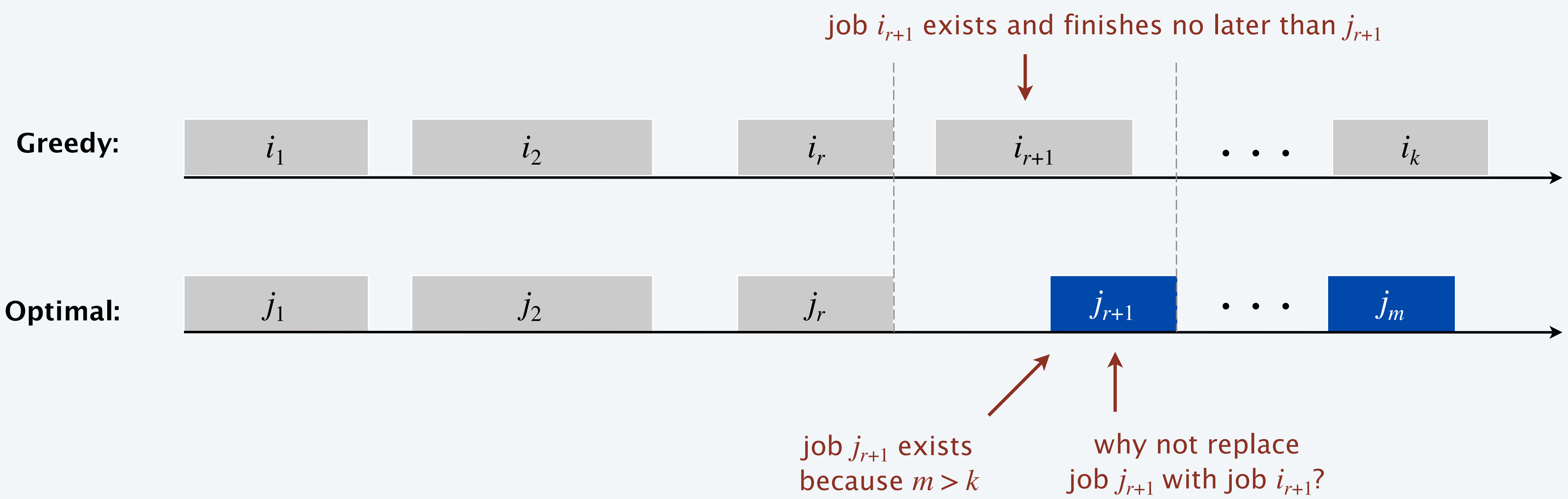
- Keep track of job j^* that was added last to S.
- Job j is compatible with S iff $s_j \geq f_{j^*}$.
- Sorting by finish times takes $O(n \log n)$ time.

Interval scheduling: analysis of earliest-finish-time-first algorithm

Theorem. The earliest-finish-time-first algorithm is optimal.

Pf. [by contradiction]

- Assume greedy is not optimal, and let's see what happens.
- Let $i_1, i_2, \ldots\ i_k$ denote set of jobs selected by greedy.
- Let $j_1, j_2, \ldots\ j_m$ denote set of jobs in an optimal solution with $i_1 = j_1, i_2 = j_2, \ldots, i_r = j_r$ for the largest possible value of r.

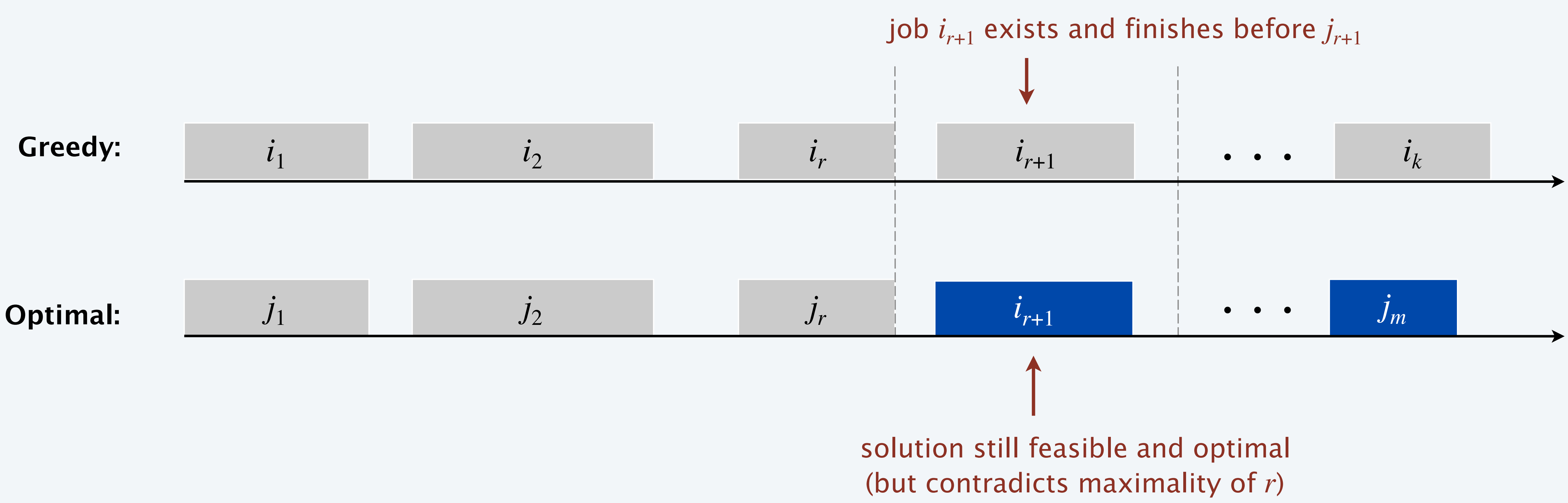


Interval scheduling: analysis of earliest-finish-time-first algorithm

Theorem. The earliest-finish-time-first algorithm is optimal.

Pf. [by contradiction]

- Assume greedy is not optimal, and let's see what happens.
- Let $i_1, i_2, \ldots i_k$ denote set of jobs selected by greedy.
- Let $j_1, j_2, \ldots j_m$ denote set of jobs in an optimal solution with $i_1 = j_1, i_2 = j_2, \ldots, i_r = j_r$ for the largest possible value of r.



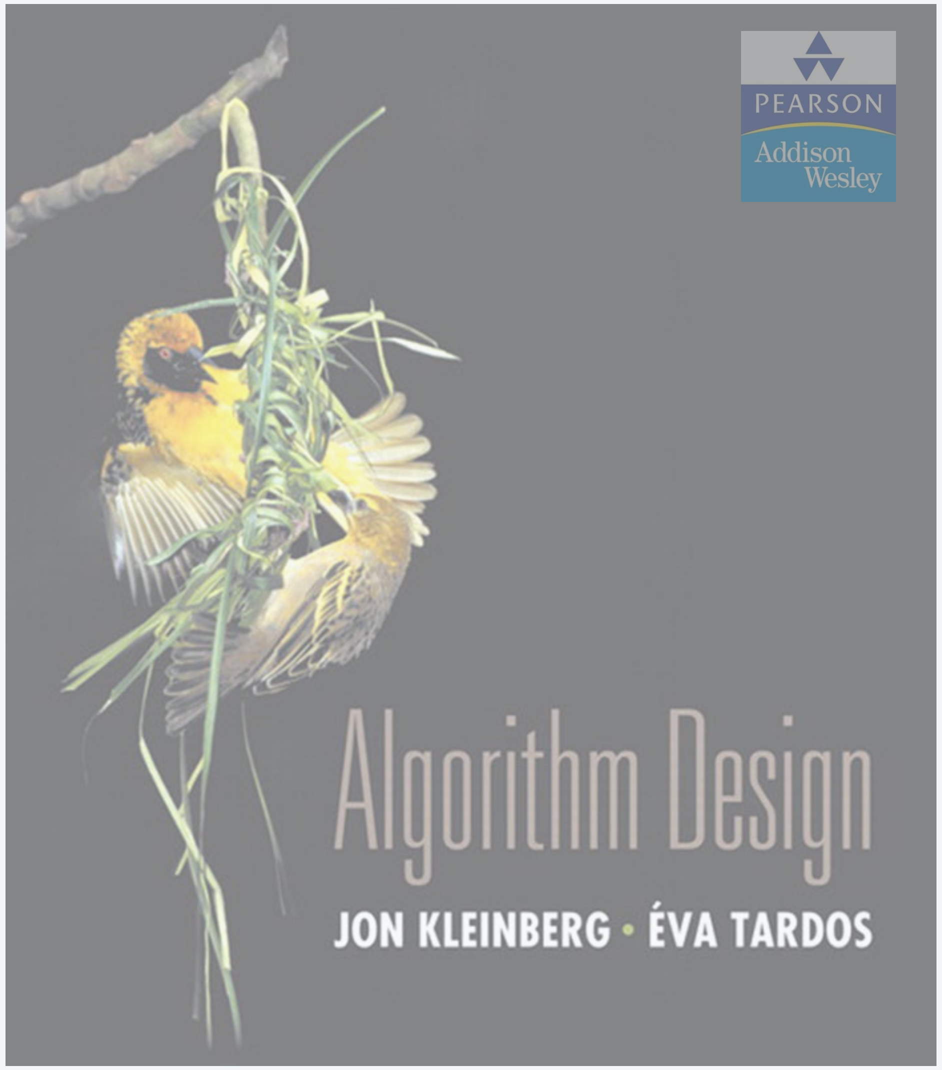
Suppose that each job also has a positive weight and the goal is to find a maximum weight subset of mutually compatible intervals. Is the earliest-finish-time-first algorithm still optimal?

A. Yes, because greedy algorithms are always optimal.

B. Yes, because the same proof of correctness is valid.

C. No, because the same proof of correctness is no longer valid.

D. No, because you could assign a huge weight to a job that overlaps the job with the earliest finish time.



SECTION 4.1

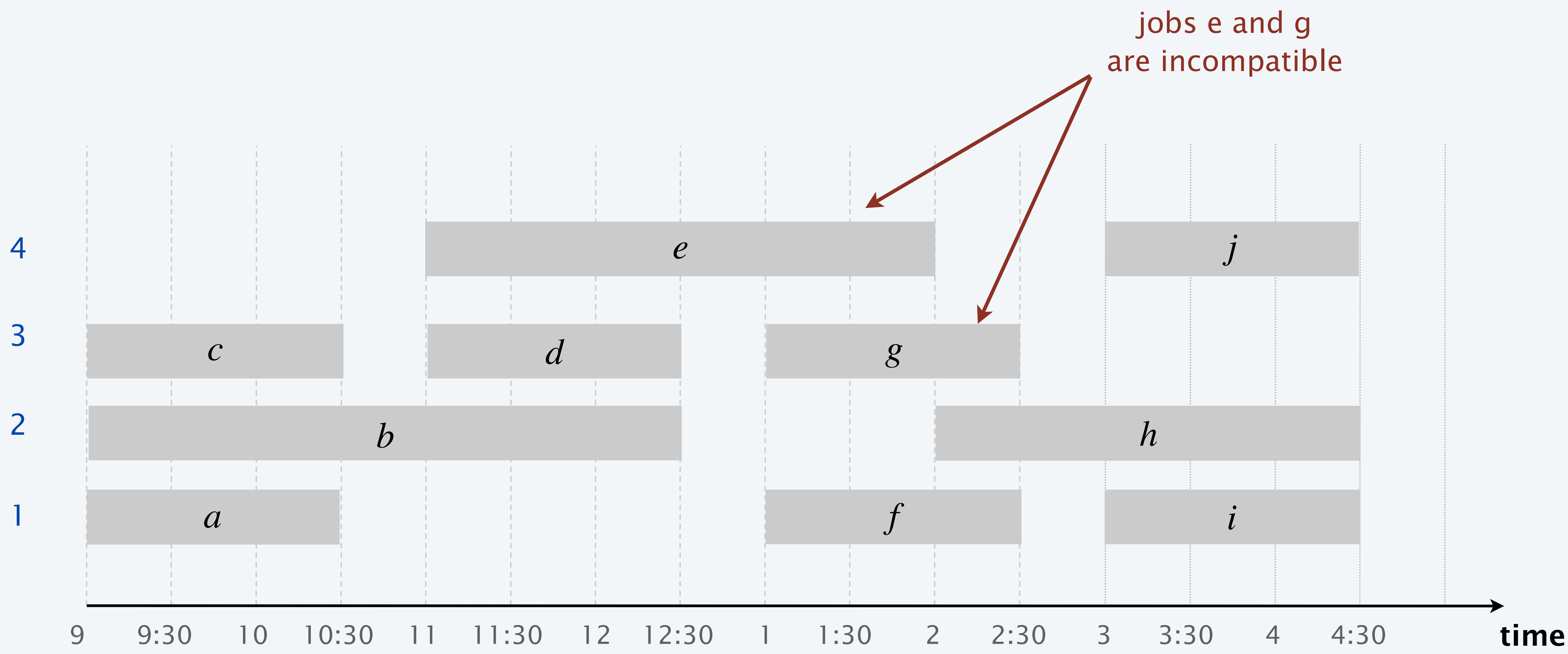
GREEDY ALGORITHMS I

- *coin changing*
- *interval scheduling*
- *interval partitioning*
- *scheduling to minimize lateness*
- *optimal caching*

Interval partitioning

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

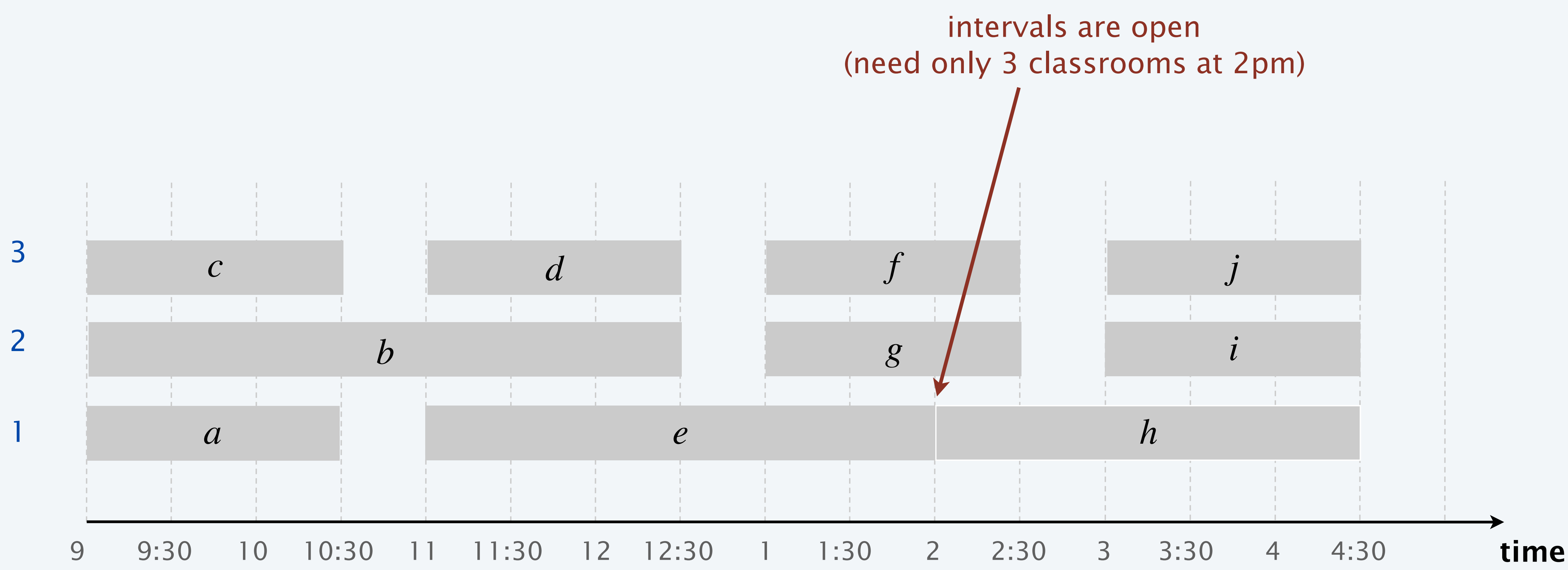
Ex. This schedule uses 4 classrooms to schedule 10 lectures.



Interval partitioning

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

Ex. This schedule uses 3 classrooms to schedule 10 lectures.



Consider lectures in some order, assigning each lecture to first available classroom (opening a new classroom if none is available). Which rule is optimal?

A. [Earliest start time] Consider lectures in ascending order of s_j.

B. [Earliest finish time] Consider lectures in ascending order of f_j.

C. [Shortest interval] Consider lectures in ascending order of $f_j - s_j$.

D. None of the above.

Interval partitioning: earliest-start-time-first algorithm

EARLIEST-START-TIME-FIRST $(n, s_1, s_2, \ldots, s_n, f_1, f_2, \ldots, f_n)$

SORT lectures by start times and renumber so that $s_1 \leq s_2 \leq \ldots \leq s_n$.

$d \leftarrow 0$. ⟵ number of allocated classrooms

FOR $j = 1$ TO n

  IF (lecture j is compatible with some classroom)

    Schedule lecture j in any such classroom k.

  ELSE

    Allocate a new classroom $d + 1$.

    Schedule lecture j in classroom $d + 1$.

    $d \leftarrow d + 1$.

RETURN schedule.

Interval partitioning: earliest-start-time-first algorithm

Proposition. The earliest-start-time-first algorithm can be implemented in $O(n \log n)$ time.

Pf. Store classrooms in a priority queue (key = finish time of its last lecture).

- To determine whether lecture j is compatible with some classroom, compare s_j to key of min classroom k in priority queue.
- To add lecture j to classroom k, increase key of classroom k to f_j.
- Total number of priority queue operations is $O(n)$.
- Sorting by start times takes $O(n \log n)$ time. ▪

Remark. This implementation chooses a classroom k whose finish time of its last lecture is the earliest.

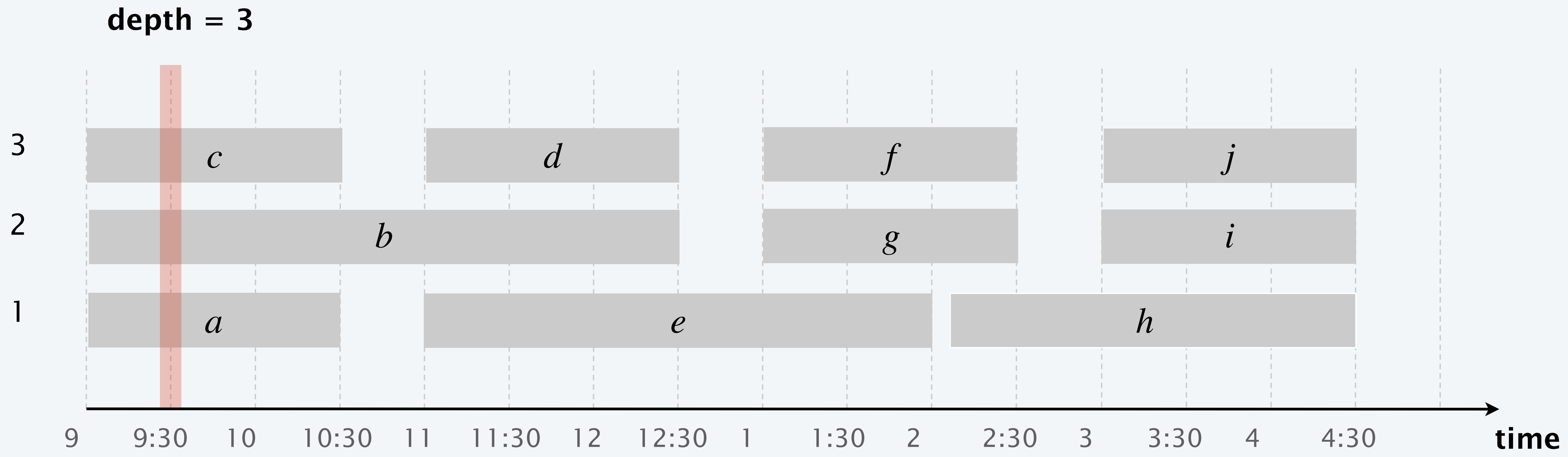
Interval partitioning: lower bound on optimal solution

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

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

Q. Does minimum number of classrooms needed always equal depth?

A. Yes! Moreover, earliest-start-time-first algorithm finds a schedule whose number of classrooms equals the depth.



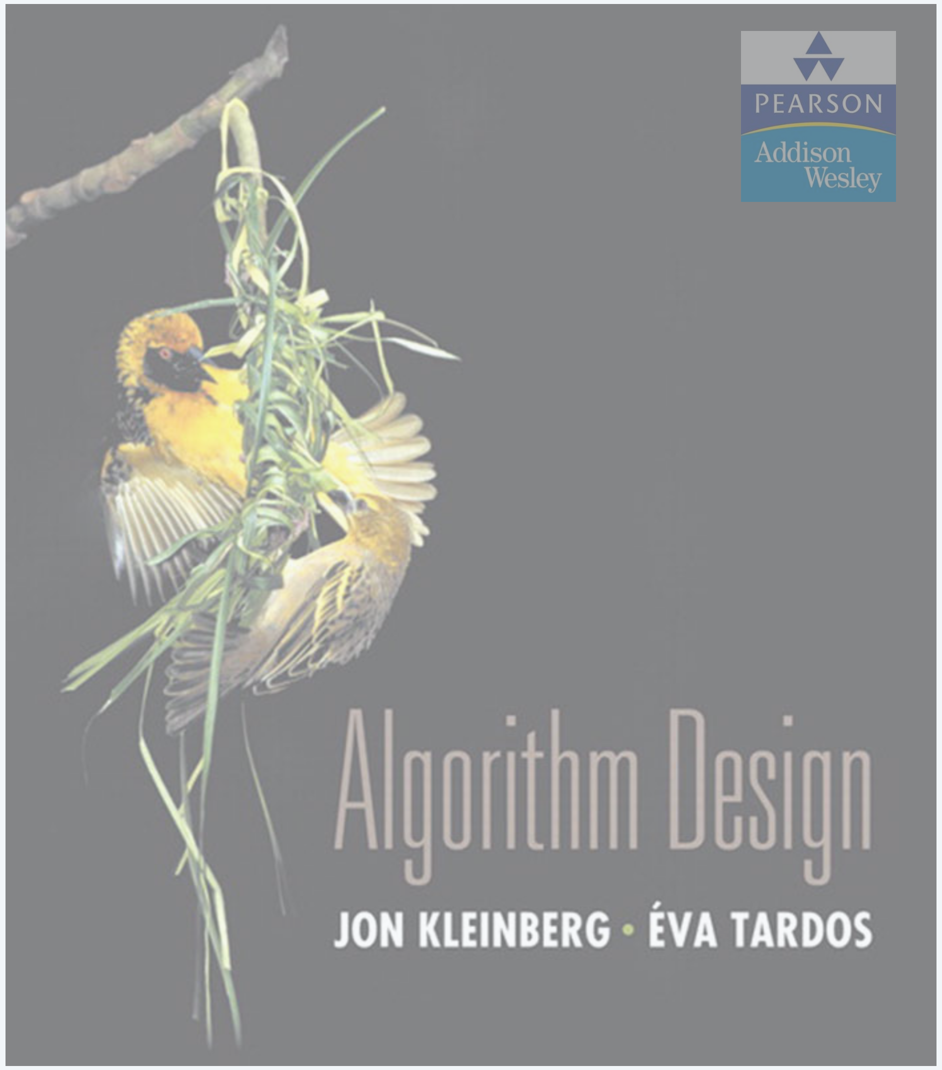
Interval partitioning: analysis of earliest-start-time-first algorithm

Observation. The earliest-start-time first algorithm never schedules two incompatible lectures in the same classroom.

Theorem. Earliest-start-time-first algorithm is optimal.

Pf.

- Let d = number of classrooms that the algorithm allocates.
- Classroom d is opened because we needed to schedule a lecture, say j, that is incompatible with a lecture in each of $d - 1$ other classrooms.
- Thus, these d lectures each end after s_j.
- Since we sorted by start time, each of these incompatible lectures 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. ▪



SECTION 4.2

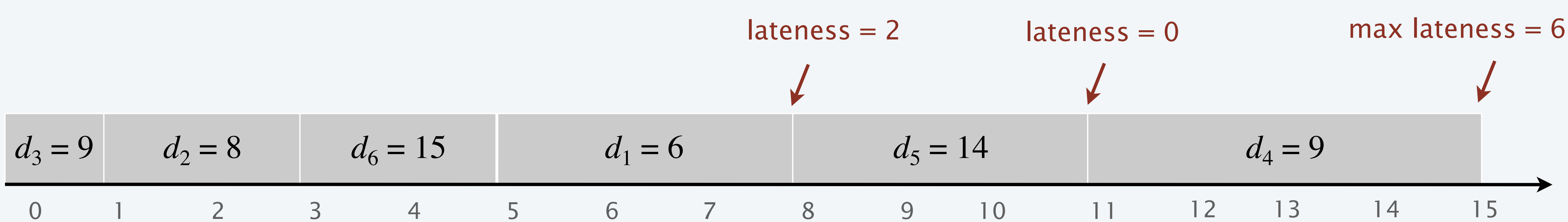
GREEDY ALGORITHMS I

- *coin changing*
- *interval scheduling*
- *interval partitioning*
- *scheduling to minimize lateness*
- *optimal caching*

Scheduling to minimizing lateness

- 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 $L = \max_j \ell_j$.

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



Schedule jobs according to some natural order. Which order minimizes the maximum lateness?

A. [shortest processing time] Ascending order of processing time t_j.

B. [earliest deadline first] Ascending order of deadline d_j.

C. [smallest slack] Ascending order of slack: $d_j - t_j$.

D. None of the above.

Minimizing lateness: earliest deadline first

EARLIEST-DEADLINE-FIRST $(n, t_1, t_2, \ldots, t_n, d_1, d_2, \ldots, d_n)$

SORT jobs by due times and renumber so that $d_1 \leq d_2 \leq \ldots \leq d_n$.

$t \leftarrow 0$.

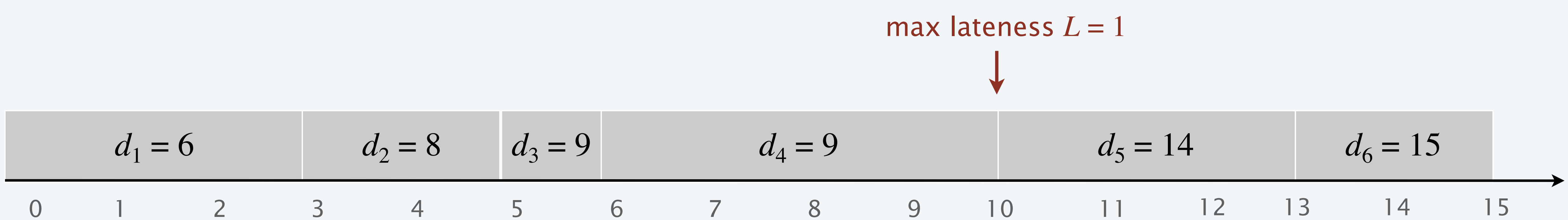
FOR $j = 1$ TO n

  Assign job j to interval $[t, t + t_j]$.

  $s_j \leftarrow t$; $f_j \leftarrow t + t_j$.

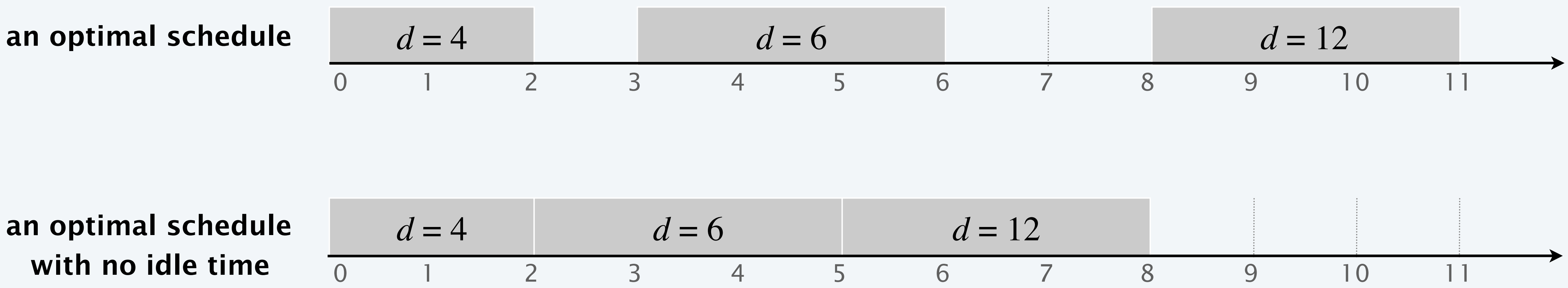
  $t \leftarrow t + t_j$.

RETURN intervals $[s_1, f_1], [s_2, f_2], \ldots, [s_n, f_n]$.



Minimizing lateness: no idle time

Observation 1. There exists an optimal schedule with no idle time.

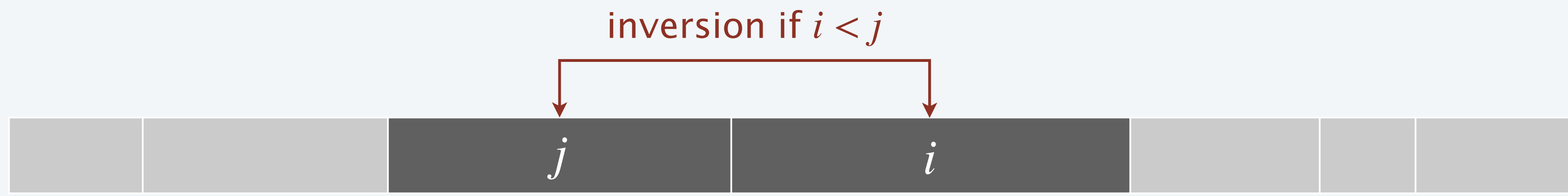


Observation 2. The earliest-deadline-first schedule has no idle time.

Minimizing lateness: inversions

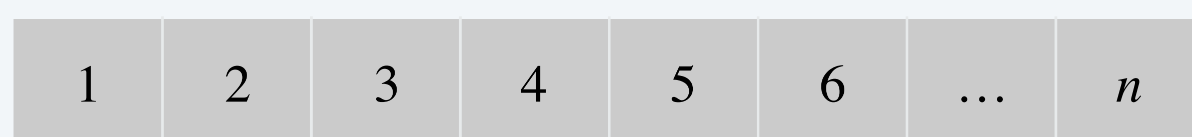
Def. Given a schedule S, an inversion is a pair of jobs i and j such that: $i < j$ but j is scheduled before i.

a schedule with an inversion



recall: we assume the jobs are numbered so that $d_1 \le d_2 \le \ldots \le d_n$

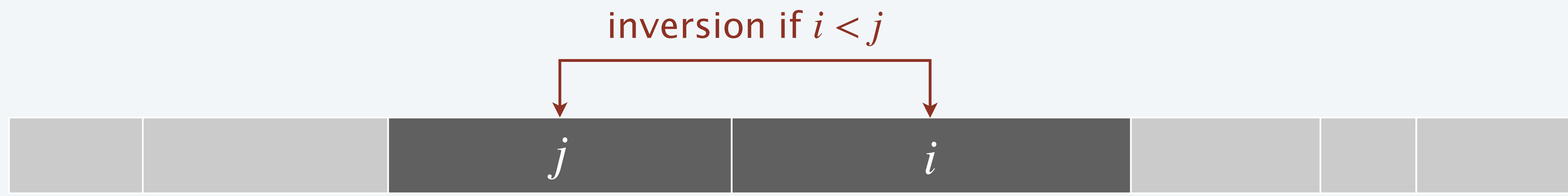
Observation 3. The earliest-deadline-first schedule is the unique idle-free schedule with no inversions.



Minimizing lateness: inversions

Def. Given a schedule S, an **inversion** is a pair of jobs i and j such that: $i < j$ but j is scheduled before i.

a schedule with an inversion



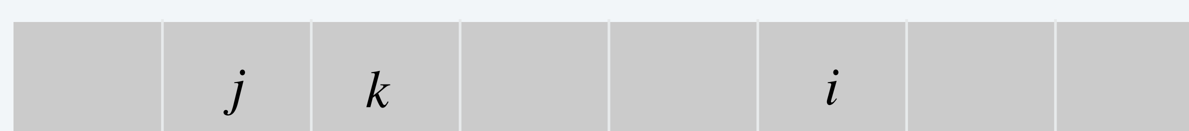
recall: we assume the jobs are numbered so that $d_1 \leq d_2 \leq \ldots \leq d_n$

Observation 4. If an idle-free schedule has an inversion, then it has an adjacent inversion.

two inverted jobs scheduled consecutively

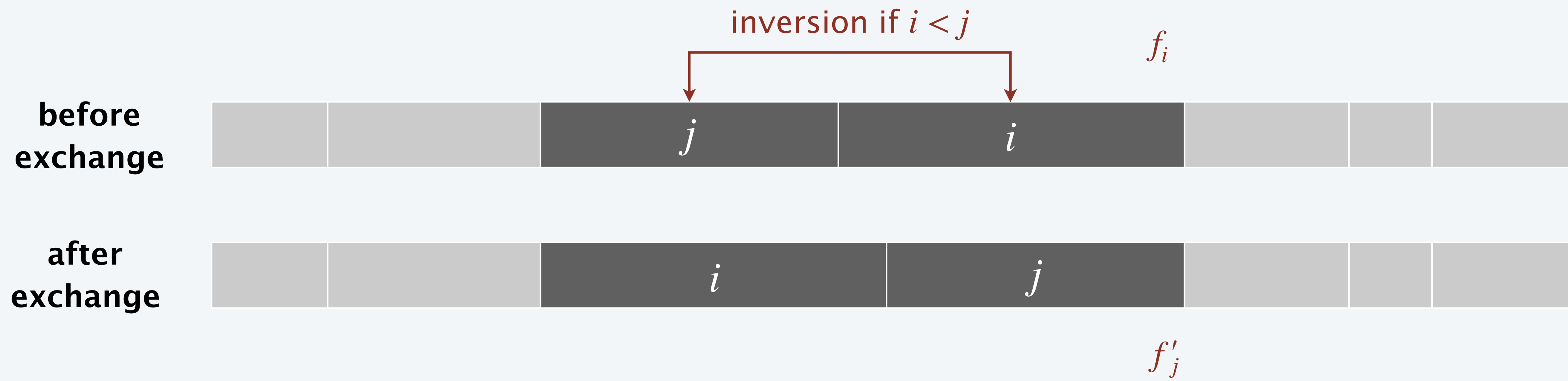
Pf.

- Let i– j be a closest inversion.
- Let k be element immediately to the right of j.
- Case 1. $[\, j > k \,]$ Then j–k is an adjacent inversion.
- Case 2. $[\, j < k \,]$ Then i–k is a closer inversion since $i < j < k$. ※



Minimizing lateness: inversions

Def. Given a schedule S, an inversion is a pair of jobs i and j such that: $i < j$ but j is scheduled before i.



Key claim. Exchanging two adjacent, inverted jobs i and j reduces the number of inversions by 1 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, ℓ'_j

$$\begin{aligned} \ell'_j &= f'_j - d_j && \longleftarrow \text{definition} \\ &= f_i - d_j && \longleftarrow j \text{ now finishes at time } f_i \\ &\leq f_i - d_i && \longleftarrow i < j \;\Rightarrow\; d_i \leq d_j \\ &\leq \ell_i\,. && \longleftarrow \text{definition} \end{aligned}$$

Minimizing lateness: analysis of earliest-deadline-first algorithm

Theorem. The earliest-deadline-first schedule S is optimal.

optimal schedule can have inversions

Pf. [by contradiction]

Define S^* to be an optimal schedule with the fewest inversions.

- Can assume S^* has no idle time. ← Observation 1
- Case 1. [S^* has no inversions] Then $S = S^*$. ← Observation 3
- Case 2. [S^* has an inversion]
  - let $i–j$ be an adjacent inversion ← Observation 4
  - exchanging jobs i and j decreases the number of inversions by 1 without increasing the max lateness ← key claim
  - contradicts "fewest inversions" part of the definition of S^* ※

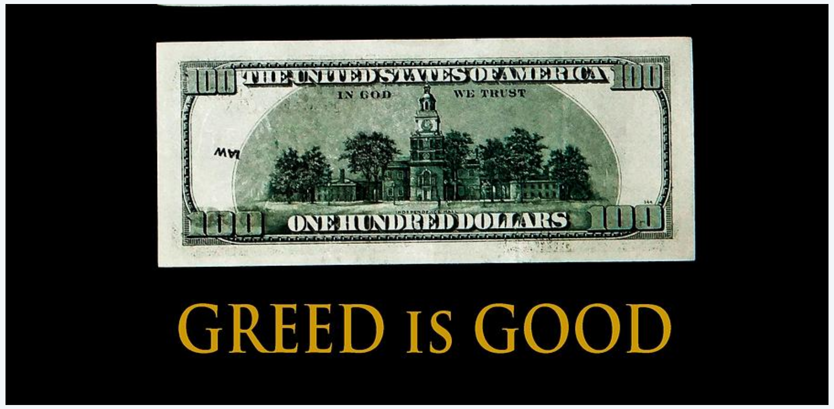
Greedy analysis strategies

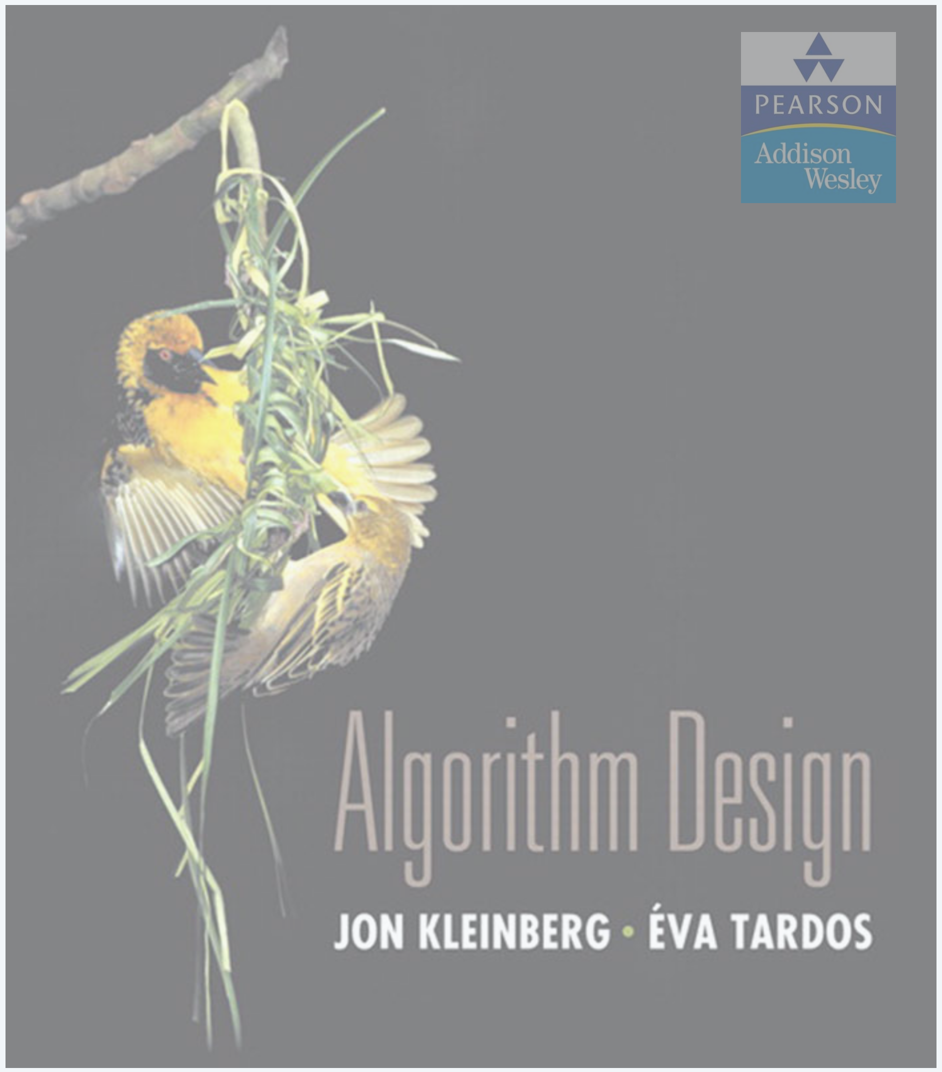
Greedy algorithm stays ahead. Show that after each step of the greedy algorithm, its solution is at least as good as any other algorithm's.

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.

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

Other greedy algorithms. Gale–Shapley, Kruskal, Prim, Dijkstra, Huffman, ...





Section 4.3

Greedy Algorithms I

- *coin changing*
- *interval scheduling*
- *interval partitioning*
- *scheduling to minimize lateness*
- *optimal caching*

Optimal offline caching

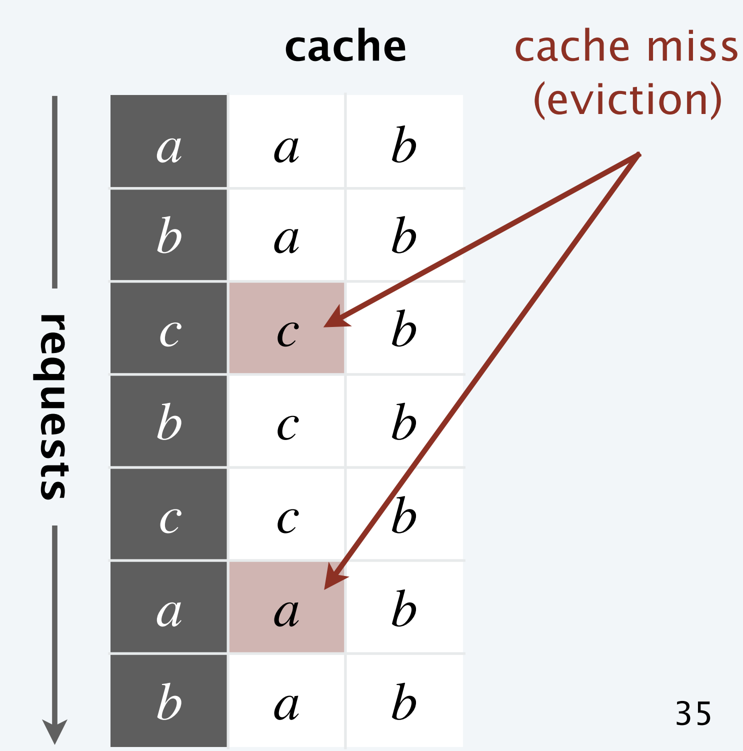
Caching.

- Cache with capacity to store k items.
- Sequence of m item requests $d_1, d_2, \ldots, d_m$.
- Cache hit: item in cache when requested.
- Cache miss: item not in cache when requested.
  (must evict some item from cache and bring requested item into cache)

Applications. CPU, RAM, hard drive, web, browser, ...

Goal. Eviction schedule that minimizes the number of evictions.

Ex. $k = 2$, initial cache $= ab$, requests: a, b, c, b, c, a, b.
Optimal eviction schedule. 2 evictions.



Optimal offline caching: greedy algorithms

LIFO/FIFO. Evict item brought in least (most) recently.

LRU. Evict item whose most recent access was earliest.

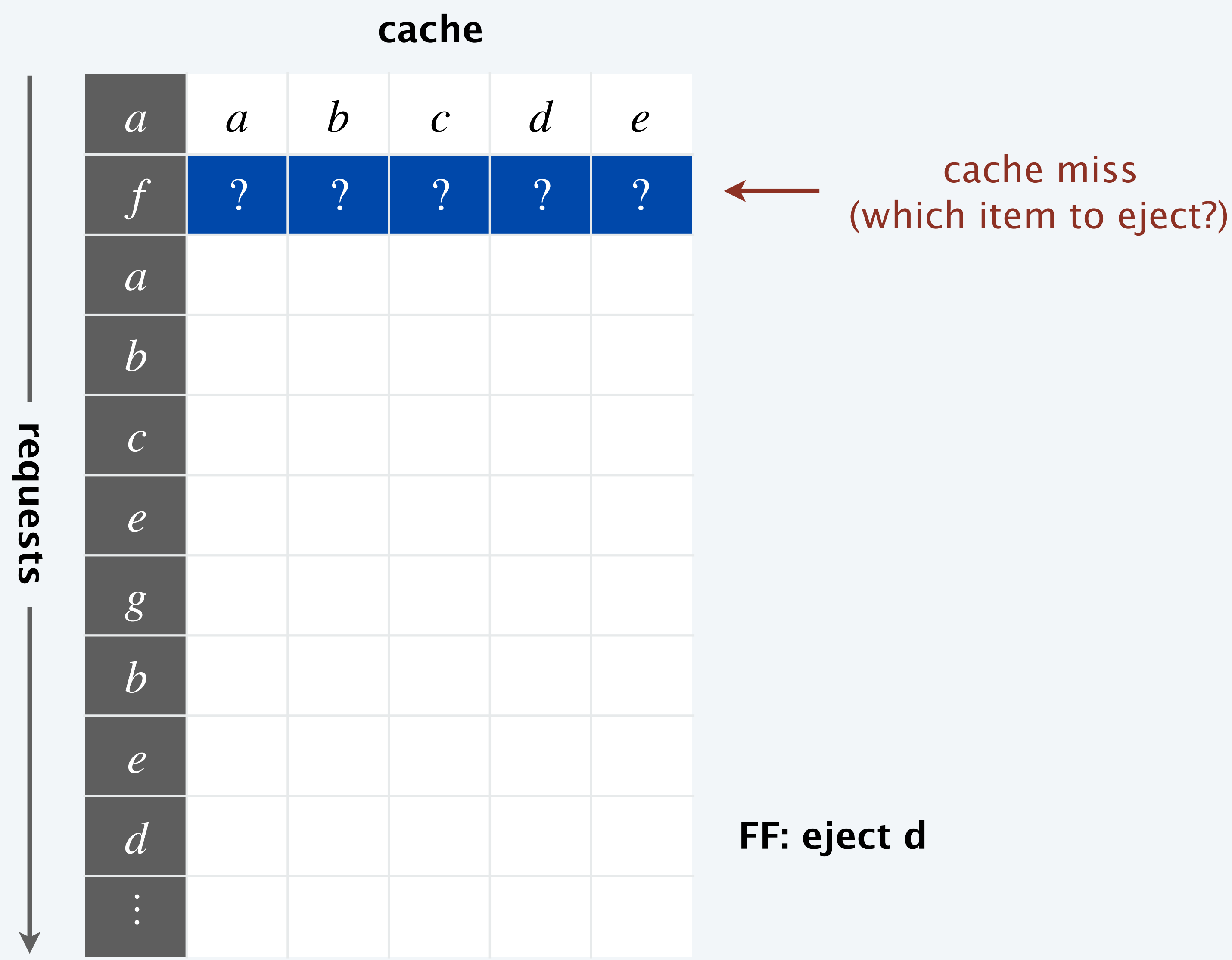
LFU. Evict item that was least frequently requested.

cache

requests						
⋮	.	.	.	.	.	
a	*a*	*w*	*x*	*y*	*z*	**FIFO: eject a**
d	*a*	*w*	*x*	*d*	*z*	**LRU: eject d**
a	*a*	*w*	*x*	*d*	*z*	
b	*a*	*b*	*x*	*d*	*z*	
c	*a*	*b*	*c*	*d*	*z*	
e	*a*	*b*	*c*	*d*	*e*	**LIFO: eject e**
g	?	?	?	?	?	cache miss (which item to eject?)
b						
e						
d						
⋮						

Optimal offline caching: farthest-in-future (clairvoyant algorithm)

Farthest-in-future. Evict item in the cache that is not requested until farthest in the future.



Theorem. [Bélády 1966] FF is optimal eviction schedule.

Pf. Algorithm and theorem are intuitive; proof is subtle.

Greedy algorithms I: quiz 6

Which item will be evicted next using farthest-in-future schedule?

A.

B.

C.

D.

E.

cache

requests				
⋮	.	.	.	.
B	D	B	Y	A
C	D	B	C	A
E	D	E	C	A
F	?	?	?	?
C				
D				
A				
E				
A				
C				
⋮				

cache miss
(which item to eject?)

Reduced eviction schedules

Def. A reduced schedule is a schedule that brings an item d into the cache in step j only if there is a request for d in step j and d is not already in the cache.

a	a	b	c
a	a	b	c
c	a	d	c
d	a	d	c
a	a	c	b
b	a	c	b
c	a	c	b
d	d	c	b
d	d	c	d

d enters cache without a request

d enters cache even though already in cache

an unreduced schedule

a	a	b	c
a	a	b	c
c	a	b	c
d	a	d	c
a	a	d	c
b	a	d	b
c	a	c	b
d	d	c	b
d	d	c	b

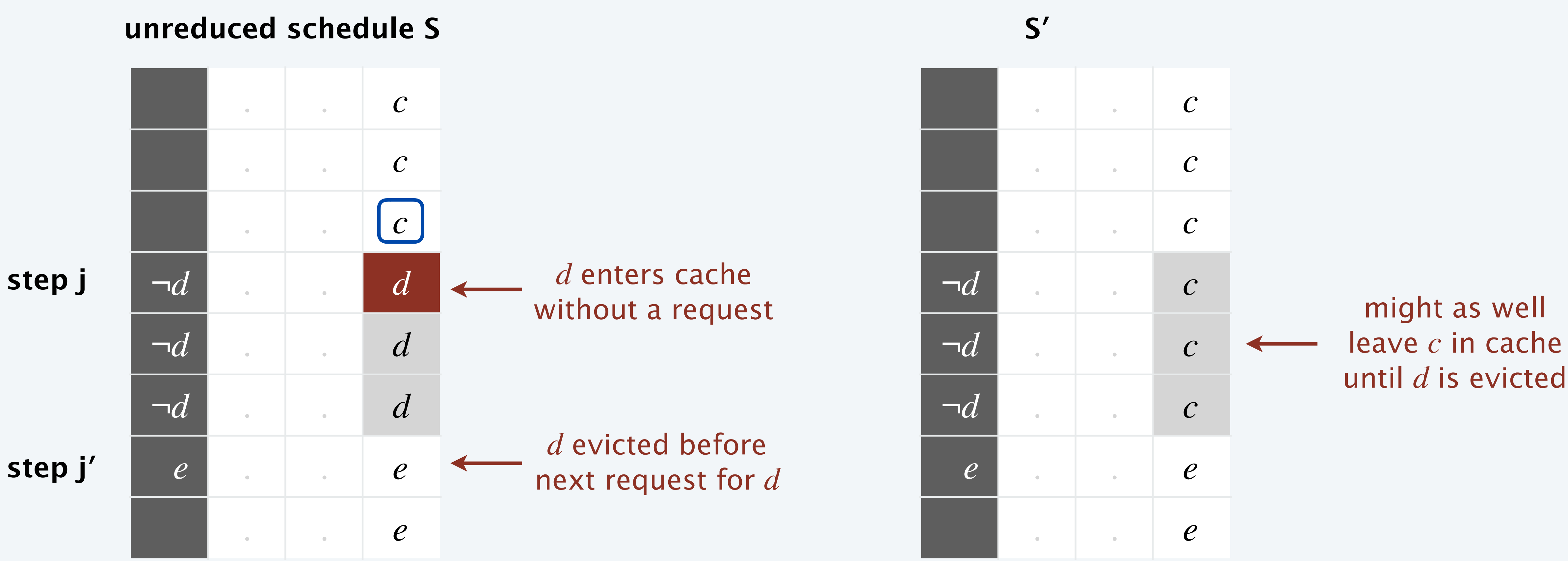
a reduced schedule

Reduced eviction schedules

Claim. Given any unreduced schedule S, can transform it into a reduced schedule S' with no more evictions.

Pf. [by induction on number of steps j]

- Suppose S brings d into the cache in step j without a request.
- Let c be the item S evicts when it brings d into the cache.
- Case 1a: d evicted before next request for d.

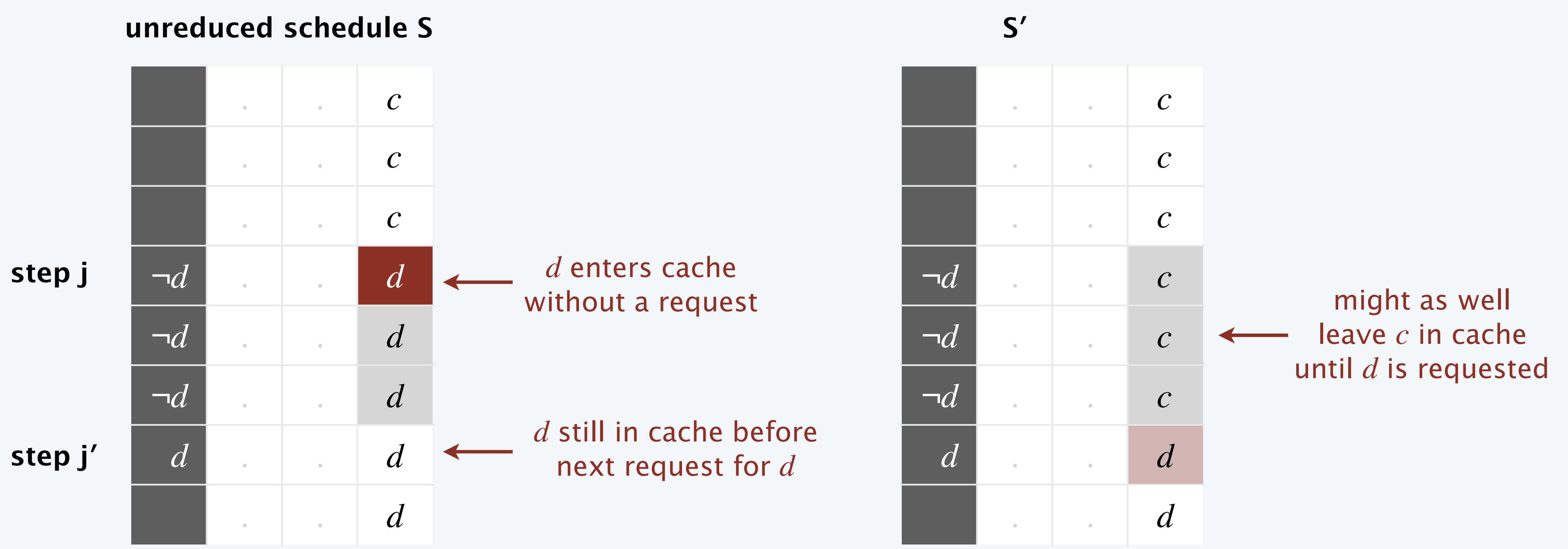


Reduced eviction schedules

Claim. Given any unreduced schedule S, can transform it into a reduced schedule S' with no more evictions.

Pf. [by induction on number of steps j]

- Suppose S brings d into the cache in step j without a request.
- Let c be the item S evicts when it brings d into the cache.
- Case 1a: d evicted before next request for d.
- Case 1b: next request for d occurs before d is evicted.



Reduced eviction schedules

Claim. Given any unreduced schedule S, can transform it into a reduced schedule S' with no more evictions.

Pf. [by induction on number of steps j]

- Suppose S brings d into the cache in step j even though d is in cache.
- Let c be the item S evicts when it brings d into the cache.
- Case 2a: d evicted before it is needed.

unreduced schedule S

		d_1	a	c	
		d_1	a	c	
		d_1	a	$\boxed{c}$	
step j	d	d_1	a	d_3	d_3 enters cache even though d_1 is already in cache
	d	d_1	a	d_3	d_3 not needed
	c	c	a	d_3	
step j’	b	c	a	b	d_3 evicted
	d	c	a	d_3	d_3 needed

S’

	d_1	a	c	
	d_1	a	c	
	d_1	a	c	
d	d_1	a	c	might as well leave c in cache until d_3 in evicted
d	d_1	a	c	
c	c	a	c	
b	c	a	b	
d	c	a	d_3	

Reduced eviction schedules

Claim. Given any unreduced schedule S, can transform it into a reduced schedule S' with no more evictions.

Pf. [by induction on number of steps j]

- Suppose S brings d into the cache in step j even though d is in cache.
- Let c be the item S evicts when it brings d into the cache.
- Case 2a: d evicted before it is needed.
- Case 2b: d needed before it is evicted.

unreduced schedule S

		d_1	a	c	
		d_1	a	c	
		d_1	a	$\boxed{c}$	
step j	d	d_1	a	d_3	← d_3 enters cache even though d_1 is already in cache
	d	d_1	a	d_3	← d_3 not needed
	c	c	a	d_3	
	a	c	a	d_3	
step j′	d	c	a	d_3	← d_3 needed

S′

	d_1	a	c	
	d_1	a	c	
	d_1	a	c	
d	d_1	a	c	← might as well leave c in cache until d_3 in needed
d	d_1	a	c	
c	c	a	c	
a	c	a	c	
d	c	a	d_3	

Reduced eviction schedules

Claim. Given any unreduced schedule S, can transform it into a reduced schedule S' with no more evictions.

Pf. [by induction on number of steps j]

- Case 1: S brings d into the cache in step j without a request. ✔
- Case 2: S brings d into the cache in step j even though d is in cache. ✔
- If multiple unreduced items in step j, apply each one in turn, dealing with Case 1 before Case 2. ▪

resolving Case 1 might trigger Case 2

Farthest-in-future: analysis

Theorem. FF is optimal eviction algorithm.
Pf. Follows directly from the following invariant.

Invariant. There exists an optimal reduced schedule S that has the same eviction schedule as S_{FF} through the first j steps.
Pf. [by induction on number of steps j]
Base case: $j = 0$.
Let S be reduced schedule that satisfies invariant through j steps.
We produce S' that satisfies invariant after $j+1$ steps.

- Let d denote the item requested in step $j+1$.
- Since S and S_{FF} have agreed up until now, they have the same cache contents before step $j+1$.
- Case 1: d is already in the cache.
  $S' = S$ satisfies invariant.
- Case 2: d is not in the cache and S and S_{FF} evict the same item.
  $S' = S$ satisfies invariant.

Farthest-in-future: analysis

Pf. [continued]

- Case 3: d is not in the cache; S_{FF} evicts e; S evicts $f \neq e$.
  - begin construction of S' from S by evicting e instead of f

  - now S' agrees with S_{FF} for first $j+1$ steps; we show that having item f in cache is no worse than having item e in cache
  - let S' behave the same as S until S' is forced to take a different action (because either S evicts e; or because either e or f is requested)

Farthest-in-future: analysis

Let j' be the first step after $j+1$ that S' must take a different action from S; let g denote the item requested in step j'.

involves either e or f (or both)

same	e	**step j′**	*same*	f

S **S′**

- Case 3a: $g = e$.
  Can't happen with FF since there must be a request for f before e.
  S' agrees with S_{FF} through first $j+1$ steps

- Case 3b: $g = f$.
  Element f can't be in cache of S; let e' be the item that S evicts.
  - if $e' = e$, S' accesses f from cache; now S and S' have same cache
  - if $e' \neq e$, we make S' evict e' and bring e into the cache;
    now S and S' have the same cache

  We let S' behave exactly like S for remaining requests.

S' is no longer reduced, but can be transformed into a reduced schedule that agrees with FF through first $j+1$ steps

Farthest-in-future: analysis

Let j' be the first step after $j+1$ that S' must take a different action from S; let g denote the item requested in step j'.

involves wither e or f (or both)

same	e	**step j′**	*same*	f

S **S′**

otherwise S' could have taken the same action

- Case 3c: $g \neq e, f$. S evicts e.
  - make S' evict f.

same	g	**step j′**	*same*	g

S **S′**

  - now S and S' have the same cache
  - let S' behave exactly like S for the remaining requests ▪

Caching perspective

Online vs. offline algorithms.

- Offline: full sequence of requests is known a priori.
- Online (reality): requests are not known in advance.
- Caching is among most fundamental online problems in CS.

LIFO. Evict item brought in most recently.

LRU. Evict item whose most recent access was earliest.

FF with direction of time reversed!

Theorem. FF is optimal offline eviction algorithm.

- Provides basis for understanding and analyzing online algorithms.
- LIFO can be arbitrarily bad.
- LRU is k-competitive: for any sequence of requests σ, $LRU(\sigma) \leq k\, FF(\sigma) + k$.
- Randomized marking is $O(\log k)$-competitive.

see SECTION 13.8