CS101 Algorithms and Data Structures

Greedy Algorithm
Algorithm Design Ch 4.1 4.2

Outline

- Coin changing
- Interval scheduling
- Interval partitioning
- Scheduling to minimize lateness
- Optimal caching

Greedy algorithms

Suppose it is possible to build a solution through a sequence of partial solutions

- At each step, we focus on one particular partial solution and we attempt to extend that solution
- 可行
- Ultimately, the partial solutions should lead to a feasible solution which is also optimal

partial

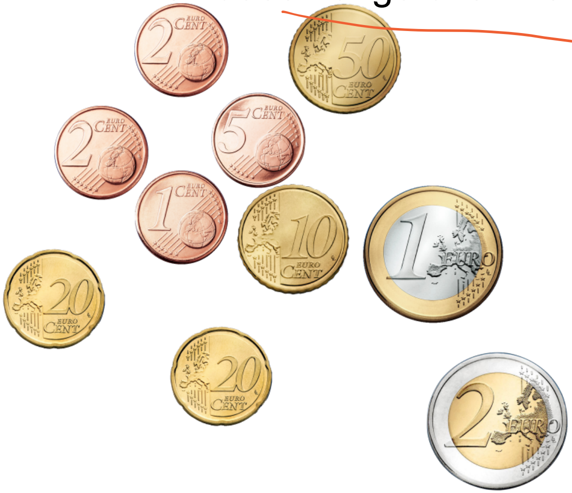
Making change

Consider this commonplace example:

- Making the exact change with the minimum number of coins
- Consider the Euro denominations of 1, 2, 5, 10, 20, 50 cents
- Stating with an empty set of coins, add the largest coin possible into the set which does not go over the required amount

strict

greedy



Making change

To make change for €0.72:

- Start with €0.50

Total €0.50

Making change

To make change for €0.72:

- Start with €0.50
- Add a €0.20

Total €0.70

Making change

To make change for €0.74:

- Start with €0.50
- Add a €0.20
- Skip the €0.10 and the €0. 05 but add a €0.02

Total €0.72

Making change

Notice that each digit can be worked with separately

- The maximum number of coins for any digit is three
- Thus, to make change for anything less than €1 requires at most six coins
- The solution is optimal

Making change

Does this strategy always work?

- What if our coin denominations grow quadraticly?

Consider 1, 4, 9, 16, 25, 36, and 49 dumbledores

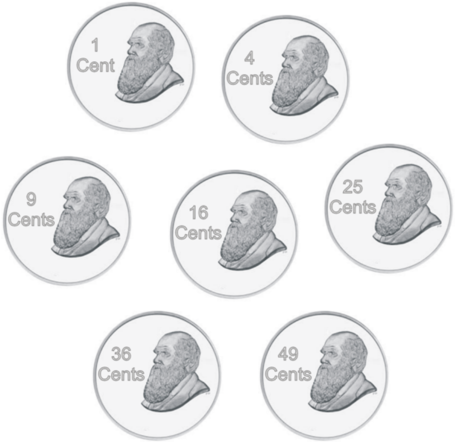
Reference: J.K. Rowlings, *Harry Potter*, Raincoast Books, 1997.

Making change

Using our algorithm, to make change for 72 dumbledores, we require six coins:

$$72 = 49 + 16 + 4 + 1 + 1 + 1$$

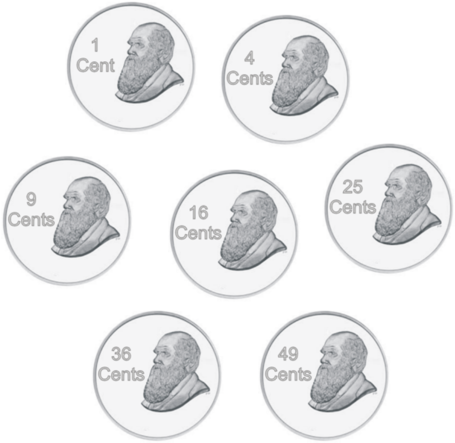
49 Cents
16 Cents
1 Cent
4 Cents
1 Cent
1 Cent



Making change

The optimal solution, however, is two 36 dumbledore coins

36 Cents
36 Cents



Definition

A greedy algorithm is an algorithm which has:

- A set of partial solutions from which a solution is built
- An *objective function* which assigns a value to any partial solution

Then given a partial solution, we

- Consider possible extensions of the partial solution
- Discard any extensions which are not feasible 可行
- Choose that extension which minimizes the object function

This continues until some criteria has been reached

Optimal example

Prim's algorithm is a greedy algorithm:

- Any connected sub-graph of *k* vertices and $k-1$ edges is a partial solution
- The value to any partial solution is the sum of the weights of the edges

Then given a partial solution, we

- Add that edge which does not create a cycle in the partial solution and which minimizes the increase in the total weight
- We continue building the partial solution until the partial solution has *n* vertices
- An optimal solution is found

Optimal and sub-optimal examples

Our coin change example is greedy:

- Any subset of *k* coins is a partial solution
- The value to any partial solution is the sum of the values

Then given a partial solution, we

- Add that coin which maximizes the increase in value without going over the target value

We continue building the set of coins until we have reached the target value

An optimal solution is found with euros and cents, but not with our *quadratic* dumbledore coins

Outline

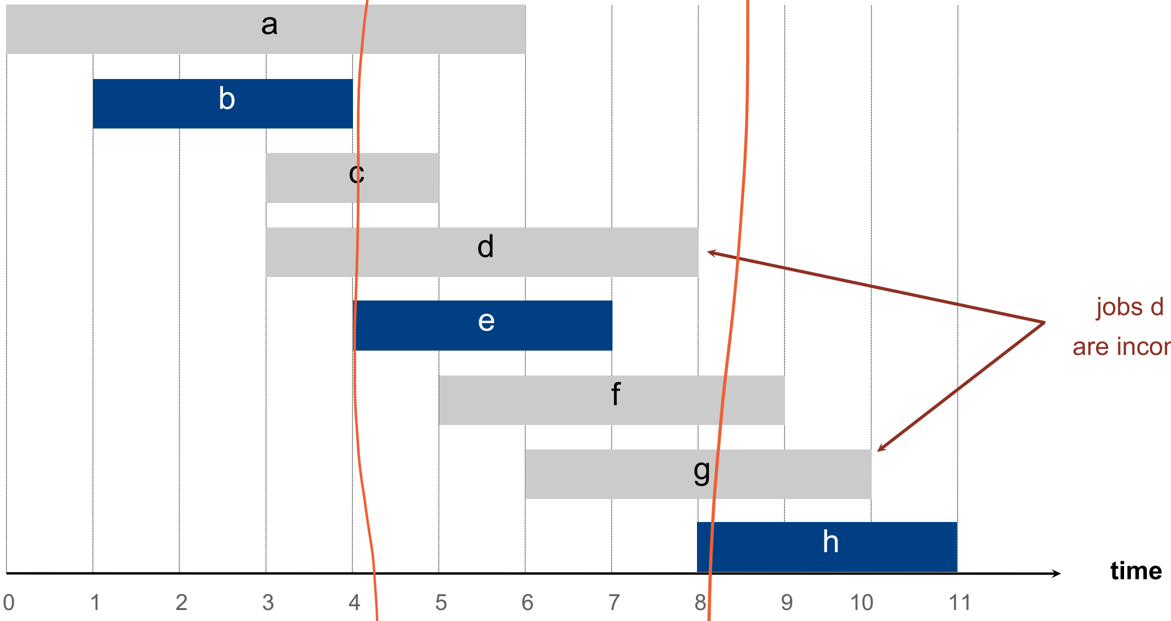
- Coin changing
- Interval scheduling
- Interval partitioning
- Scheduling to minimize lateness
- Optimal caching

[illegible]

Interval scheduling

兼容 -

- 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.



jobs d and g are incompatible

Interval scheduling: greedy algorithms

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

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

Time

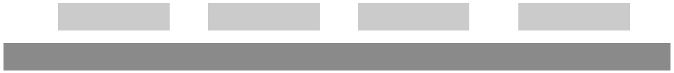
interval

Interval scheduling: greedy algorithms

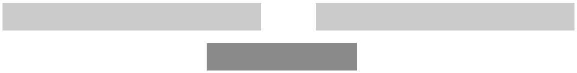
Greedy template. Consider jobs in some natural order.

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

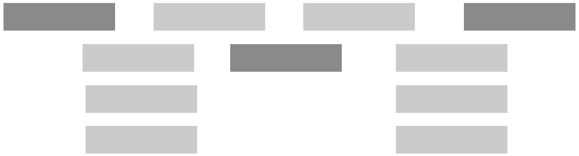
counterexample for earliest start time



counterexample for shortest interval



counterexample for fewest conflicts



Interval scheduling: earliest-finish-time-first algorithm

```
Earliest-Finish-Time-First ($n$, $s_1$, $s_2$, …, $s_n$, $f_1$, $f_2$, …, $f_n$)

Sort jobs by finish times and renumber so that $f_1 \leq f_2 \leq \ldots \leq f_n$.
$S \leftarrow \emptyset$.   ←—— set of jobs selected
For $j$ = 1 to $n$
    If (job $j$ is compatible with $S$)
        $S \leftarrow S \cup \{ j \}$.
Return $S$.
```

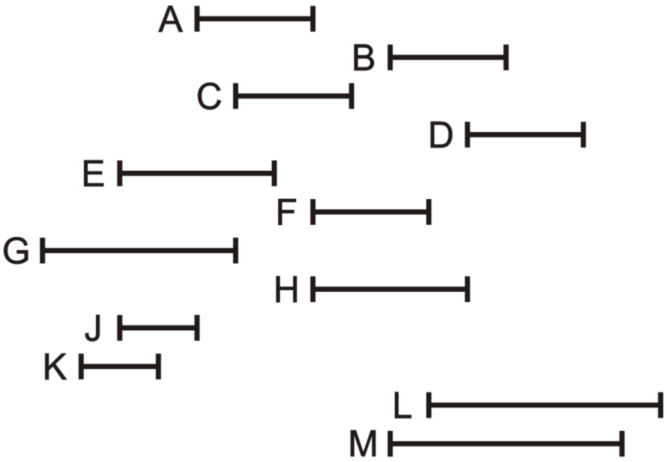
Proposition. Can implement earliest-finish-time first in $O(n \log n)$ time.

- 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: Example

Consider the following list of 12 processes together with the time interval during which they must be run

- Find the schedule with the earliest-deadline-first greedy algorithm

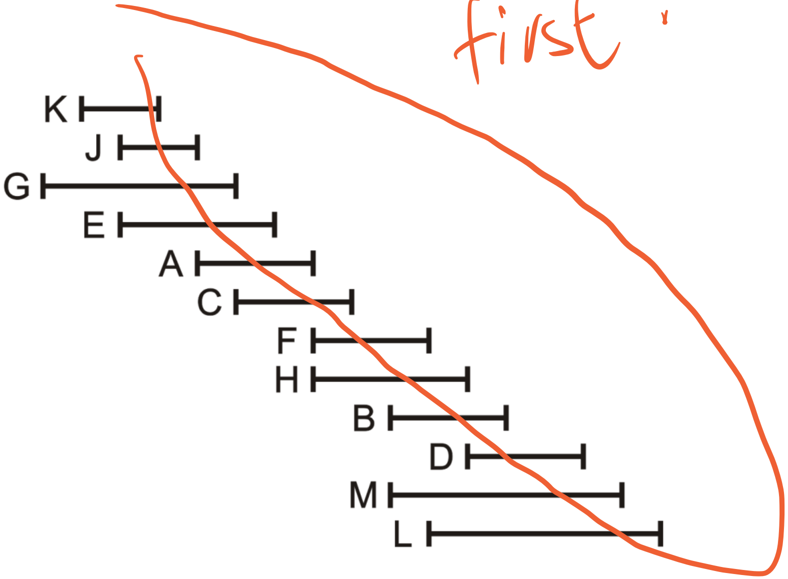


Process	Interval
A	5 – 8
B	10 – 13
C	6 – 9
D	12 – 15
E	3 – 7
F	8 – 11
G	1 – 6
H	8 – 12
J	3 – 5
K	2 – 4
L	11 – 16
M	10 – 15

Interval scheduling: Example

In order to simplify this, sort the processes on their end times

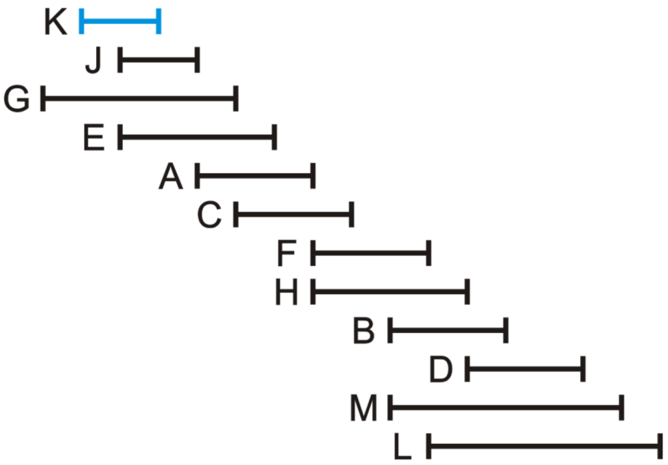
dead line first



Process	Interval
K	2 – 4
J	3 – 5
G	1 – 6
E	3 – 7
A	5 – 8
C	6 – 9
F	8 – 11
H	8 – 12
B	10 – 13
D	12 – 15
M	10 – 15
L	11 – 16

Interval scheduling: Example

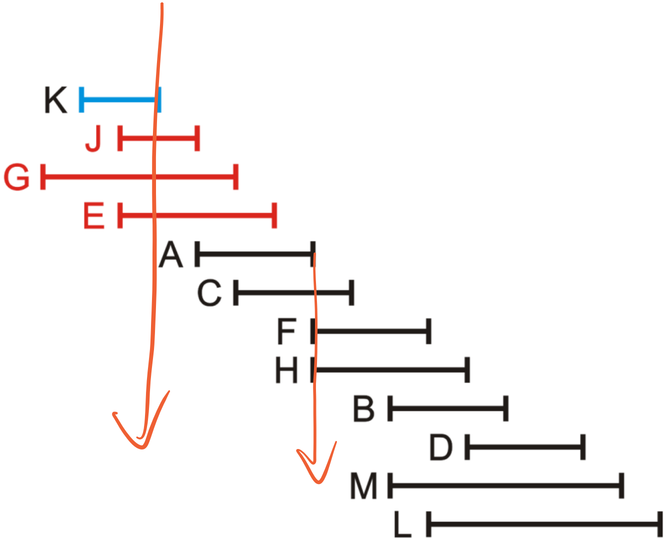
To begin, choose Process K



Process	Interval
K	2 – 4
J	3 – 5
G	1 – 6
E	3 – 7
A	5 – 8
C	6 – 9
F	8 – 11
H	8 – 12
B	10 – 13
D	12 – 15
M	10 – 15
L	11 – 16

Interval scheduling: Example

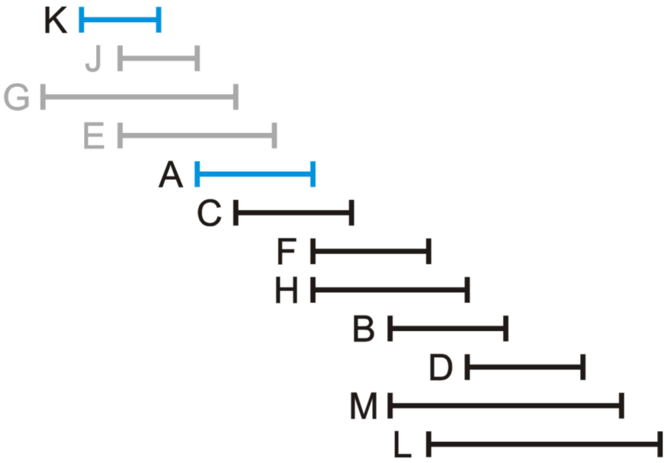
At this point, Process J, G and E can no longer be run



Process	Interval
K	2 – 4
J	3 – 5
G	1 – 6
E	3 – 7
A	5 – 8
C	6 – 9
F	8 – 11
H	8 – 12
B	10 – 13
D	12 – 15
M	10 – 15
L	11 – 16

Interval scheduling: Example

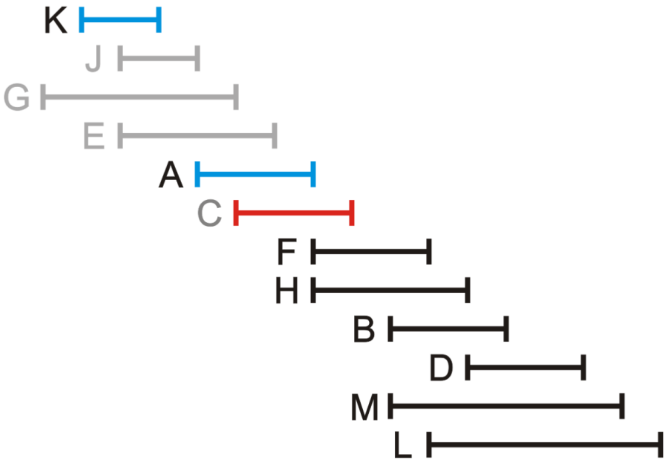
Next, run Process A



Process	Interval
K	2 – 4
J	3 – 5
G	1 – 6
E	3 – 7
A	5 – 8
C	6 – 9
F	8 – 11
H	8 – 12
B	10 – 13
D	12 – 15
M	10 – 15
L	11 – 16

Interval scheduling: Example

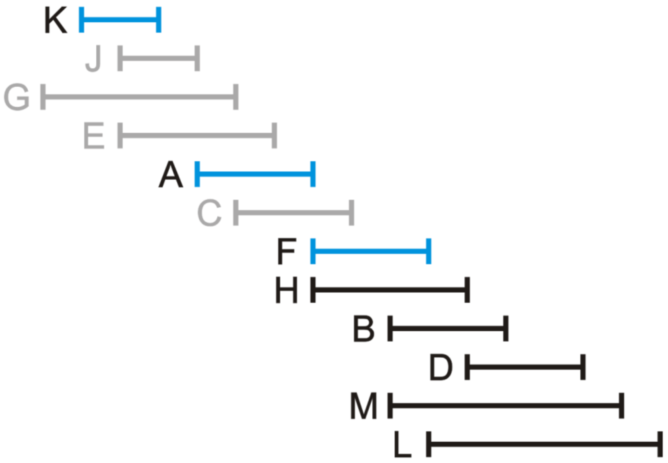
We can no longer run Process C



Process	Interval
K	2 – 4
J	3 – 5
G	1 – 6
E	3 – 7
A	5 – 8
C	6 – 9
F	8 – 11
H	8 – 12
B	10 – 13
D	12 – 15
M	10 – 15
L	11 – 16

Interval scheduling: Example

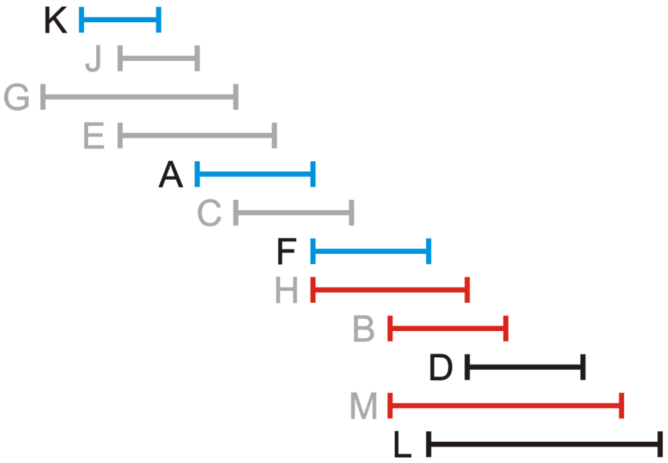
Next, we can run Process F



Process	Interval
K	2 – 4
J	3 – 5
G	1 – 6
E	3 – 7
A	5 – 8
C	6 – 9
F	8 – 11
H	8 – 12
B	10 – 13
D	12 – 15
M	10 – 15
L	11 – 16

Interval scheduling: Example

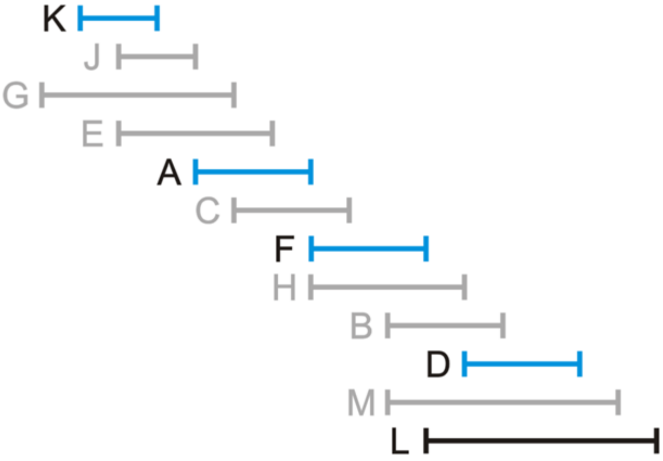
This restricts us from running Processes H, B and M



Process	Interval
K	2 – 4
J	3 – 5
G	1 – 6
E	3 – 7
A	5 – 8
C	6 – 9
F	8 – 11
H	8 – 12
B	10 – 13
D	12 – 15
M	10 – 15
L	11 – 16

Interval scheduling: Example

The next available process is D

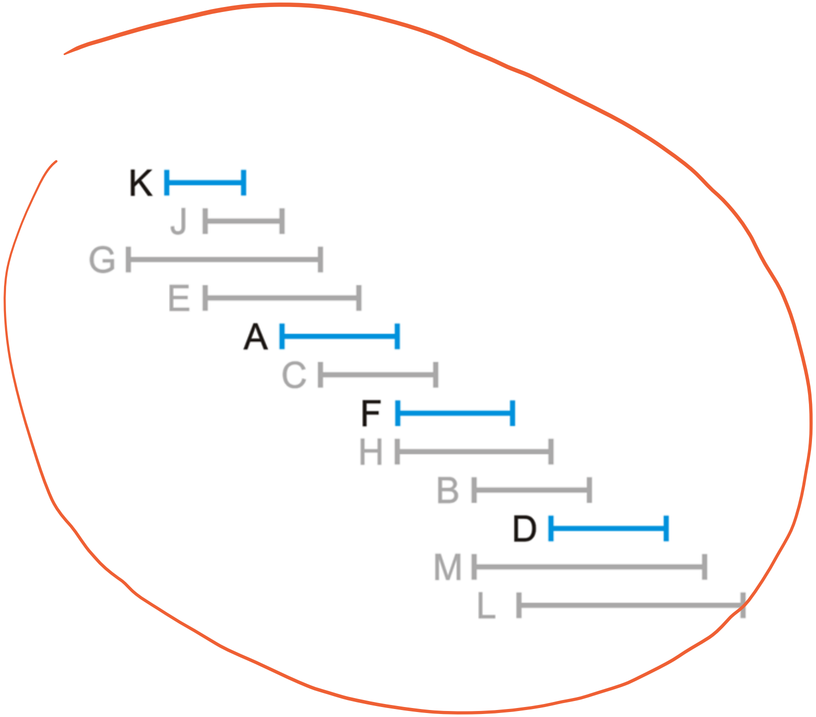


Process	Interval
K	2 – 4
J	3 – 5
G	1 – 6
E	3 – 7
A	5 – 8
C	6 – 9
F	8 – 11
H	8 – 12
B	10 – 13
D	12 – 15
M	10 – 15
L	11 – 16

Interval scheduling: Example

The prevents us from running Process L

- We are therefore finished



Process	Interval
K	**2 – 4**
J	3 – 5
G	1 – 6
E	3 – 7
A	**5 – 8**
C	6 – 9
F	**8 – 11**
H	8 – 12
B	10 – 13
D	**12 – 15**
M	10 – 15
L	11 – 16

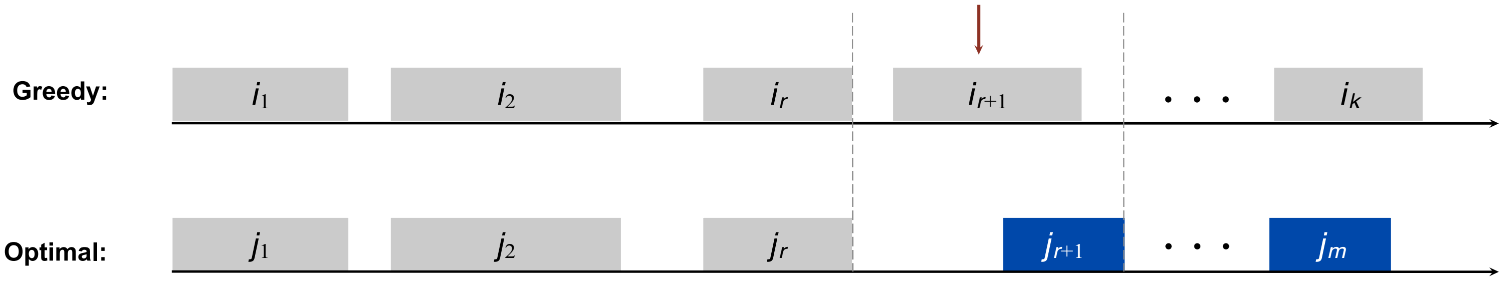
Interval scheduling: analysis of earliest-finish-time-first algorithm

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, ... i_k denote set of jobs selected by greedy.
- Let j_1, j_2, ... j_m denote set of jobs in an optimal solution with $i_1 = j_1$, $i_2 = j_2$, ..., $i_r = j_r$ for the largest possible value of r.

1. job i_{r+1} exists and finishes no later than j_{r+1}



job j_{r+1} exists because $m > k$

why not replace job j_{r+1} with job i_{r+1}?

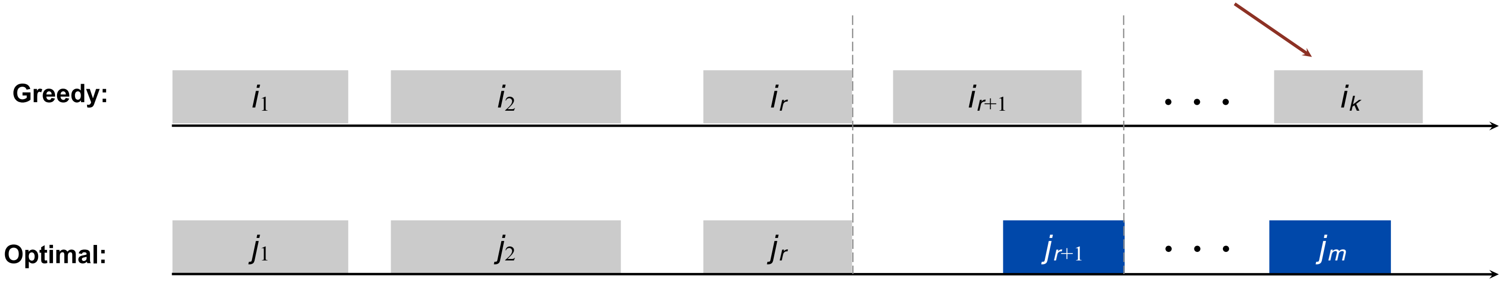
Interval scheduling: analysis of earliest-finish-time-first algorithm

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.

2. job i_k finishes no later than j_k, but m > k, the greedy algorithm should not stop, as job j_{k+1} can be chosen



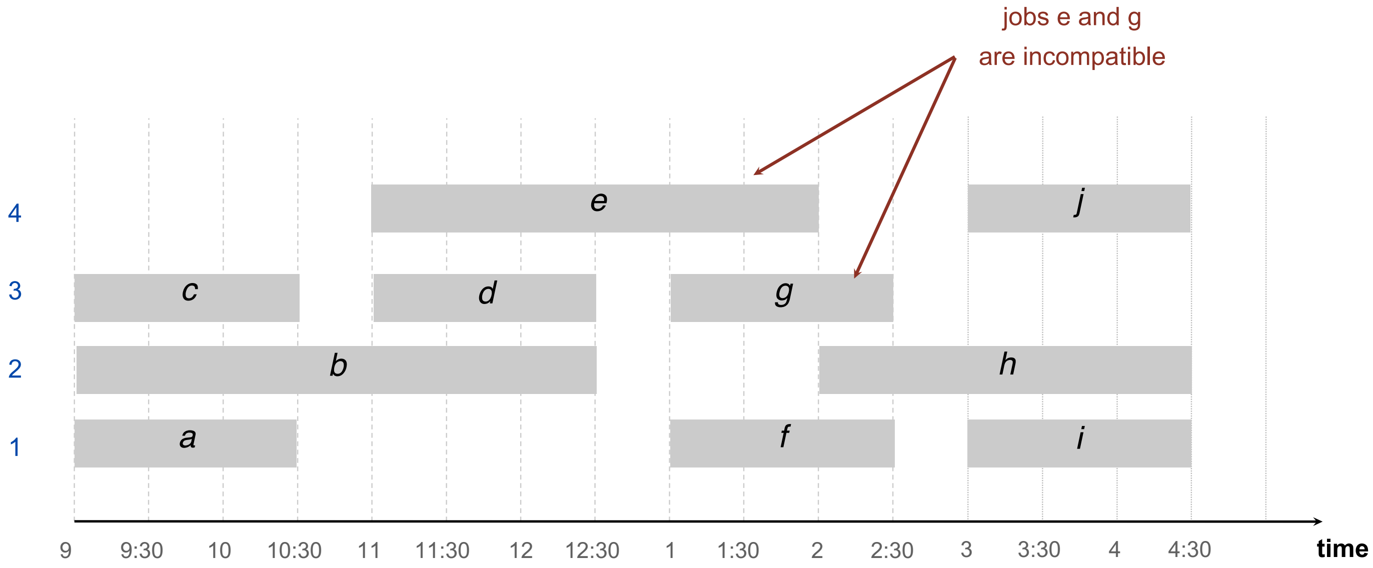
Outline

- 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.

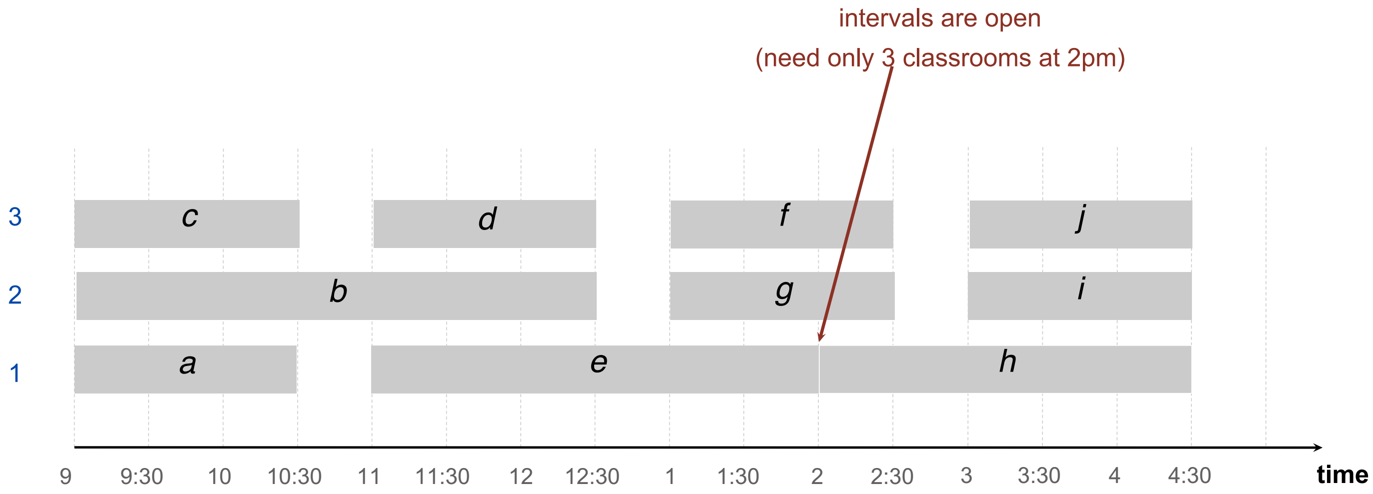
Ex. This schedule uses 4 classrooms to schedule 10 lectures.



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.



Interval partitioning: greedy algorithms ✓

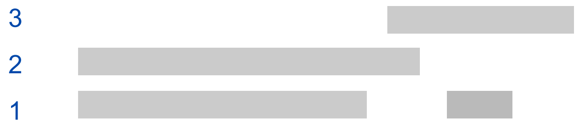
Greedy template. Consider lectures in some natural order. 1 Assign each lecture to an available classroom (which one?); 2 allocate a new classroom if none are available.

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

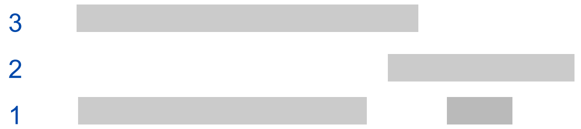
Interval partitioning: greedy algorithms

Greedy template. Consider lectures in some natural order. Assign each lecture to an available classroom (which one?); allocate a new classroom if none are available.

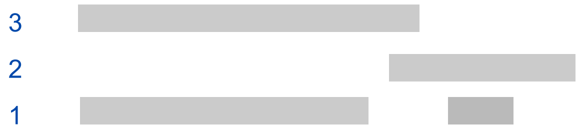
counterexample for earliest finish time



counterexample for shortest interval



counterexample for fewest conflicts

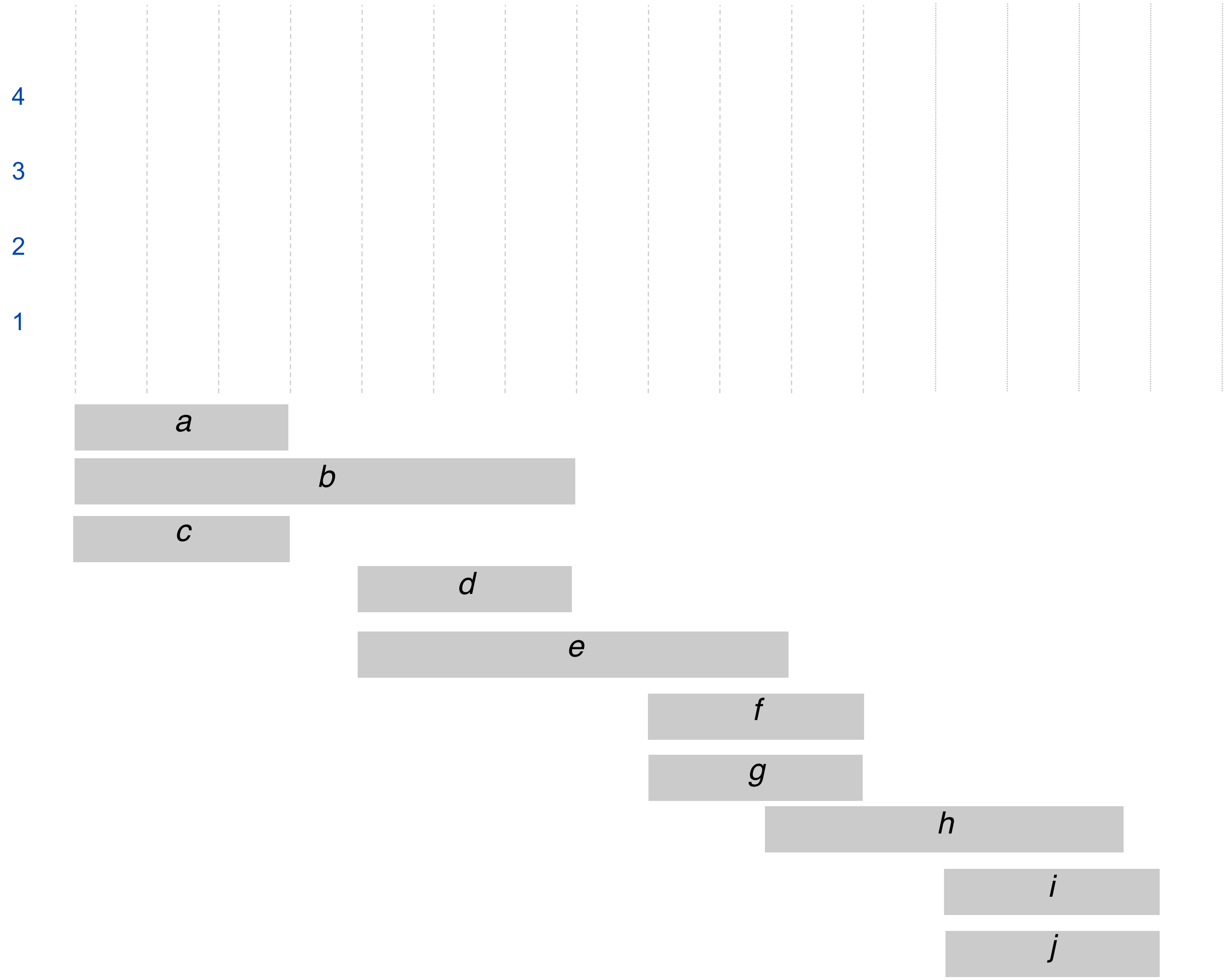


Interval partitioning: earliest-start-time-first algorithm

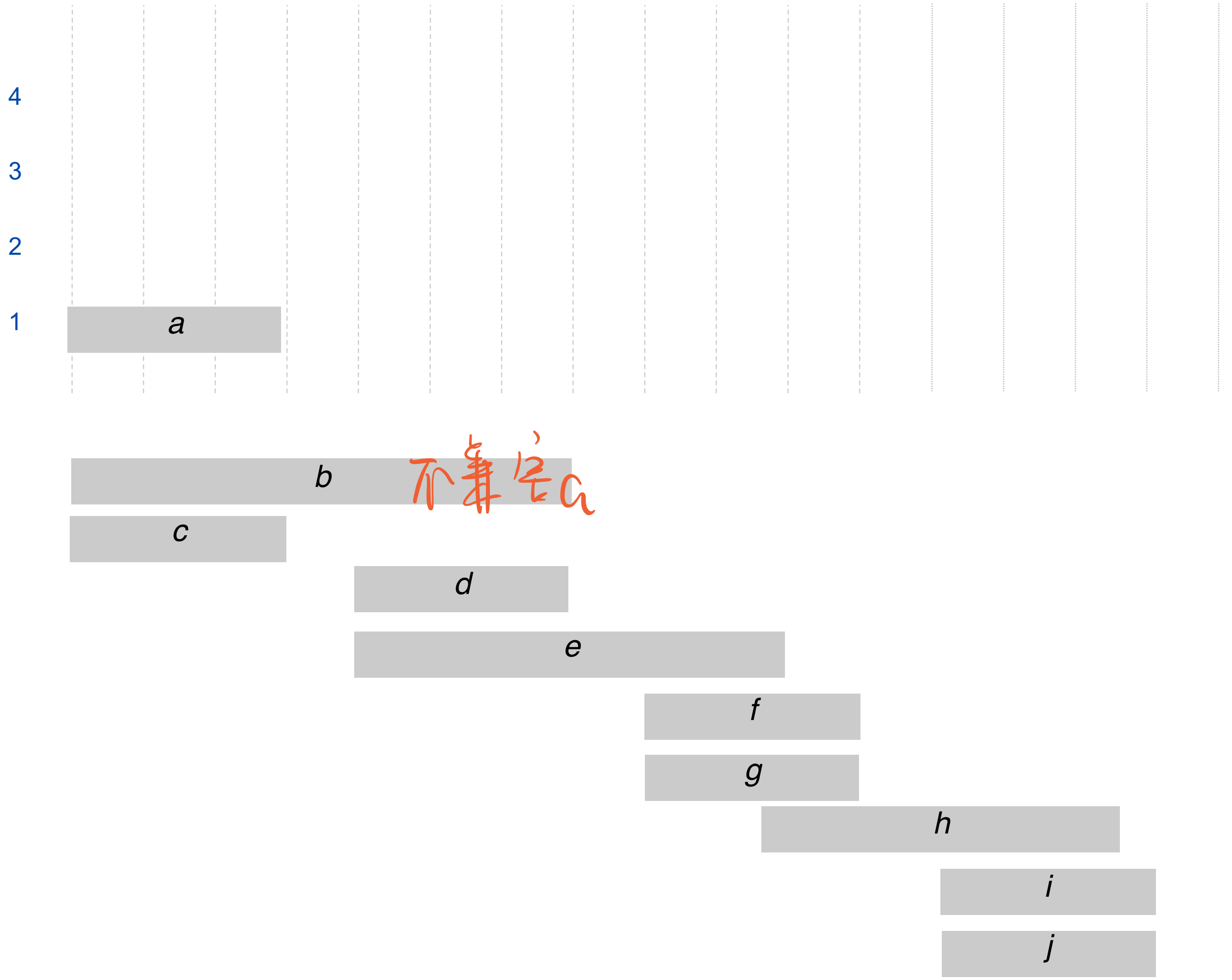
```
Earliest-Start-Time-First ($n$, $s_1$, $s_2$, …, $s_n$, $f_1$, $f_2$, …, $f_n$)

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 $n$                可兼容
    If (lecture $j$ is compatible with some classroom)
        Schedule lecture $j$ in any such classroom $k$.
    Else                          不可兼容
        Allocate a new classroom $d + 1$.
        Schedule lecture $j$ in classroom $d + 1$.
        $d \leftarrow d +$
Return schedule.
```

Interval partitioning: Example

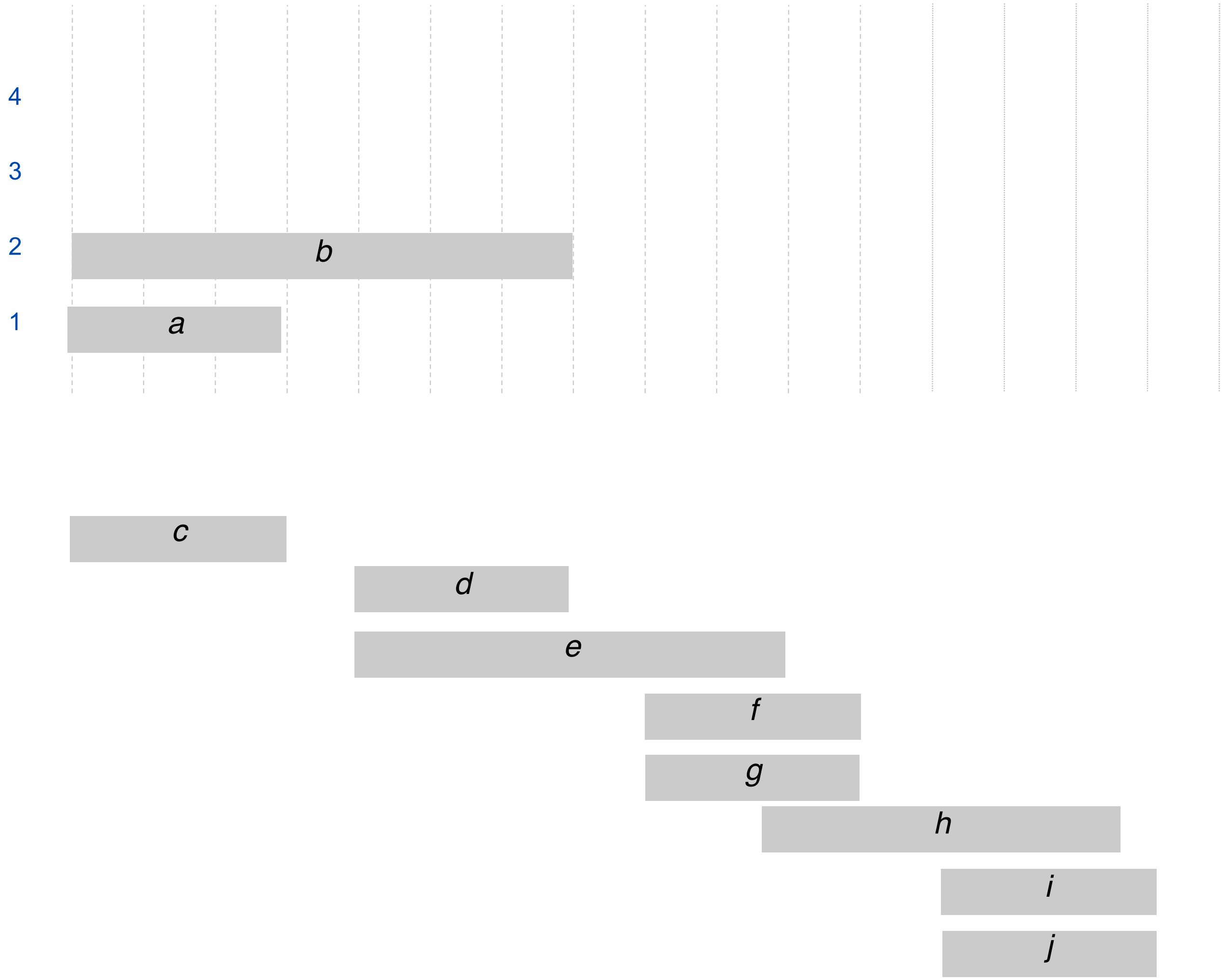


Interval partitioning: Example

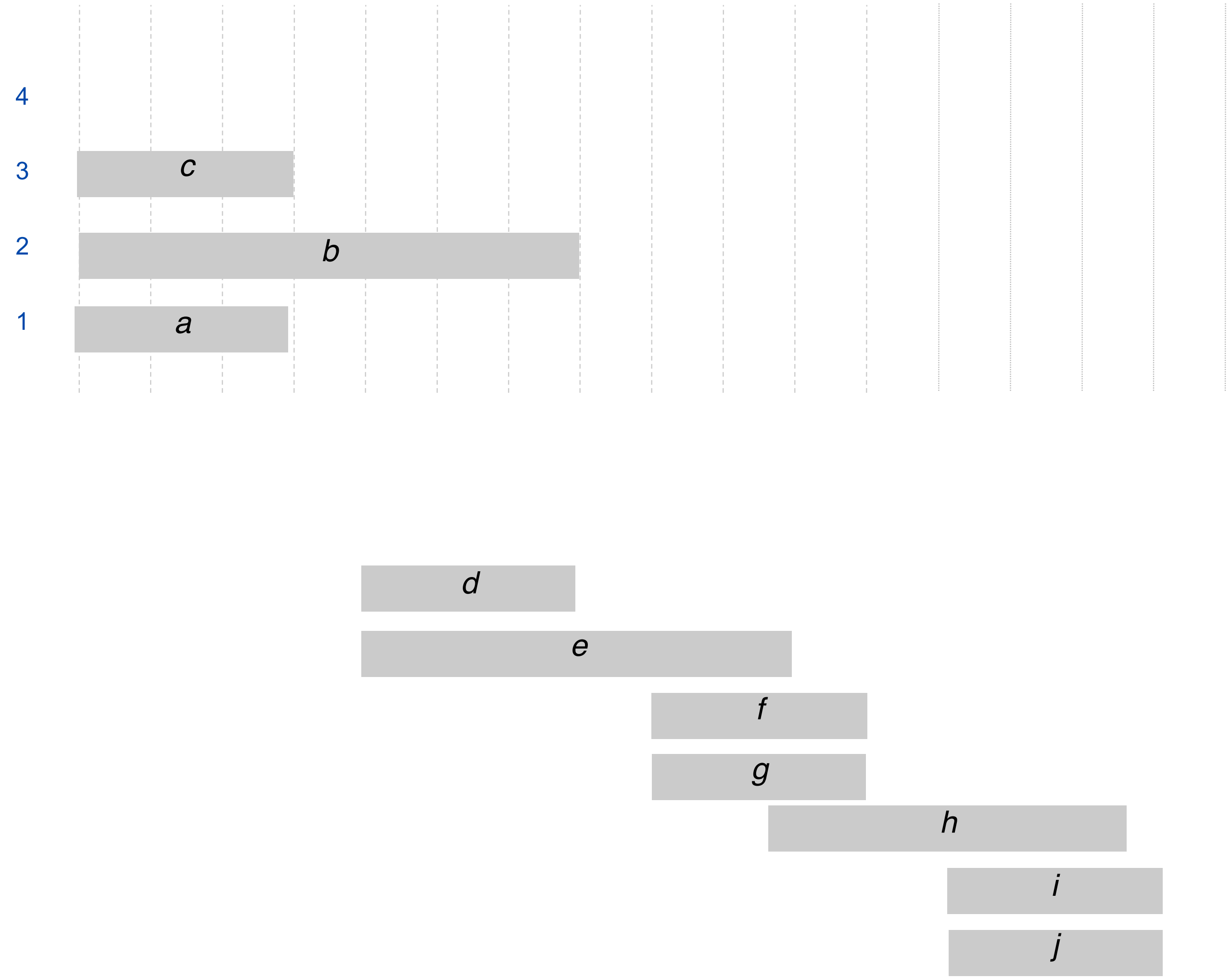


不兼容a

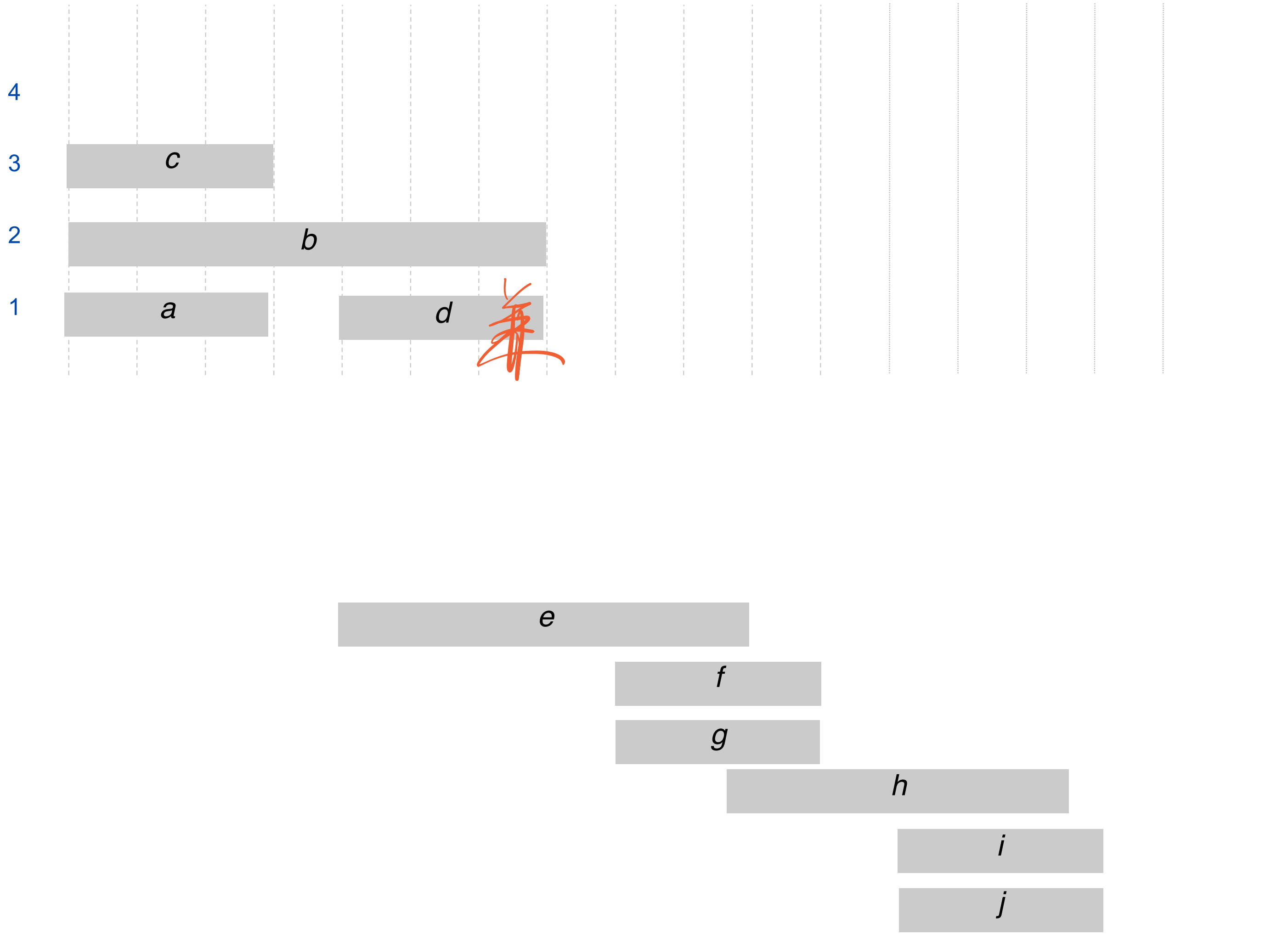
Interval partitioning: Example



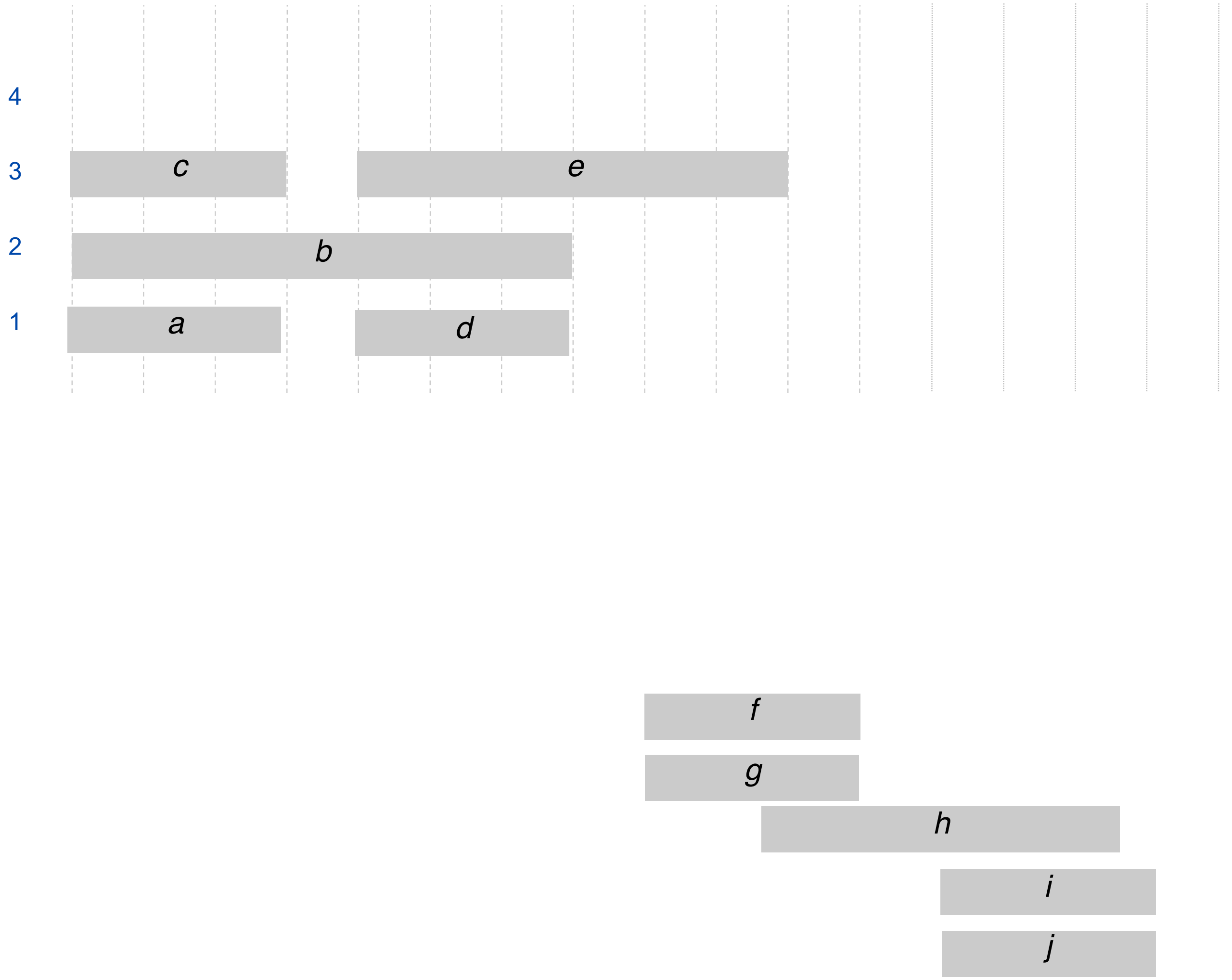
Interval partitioning: Example



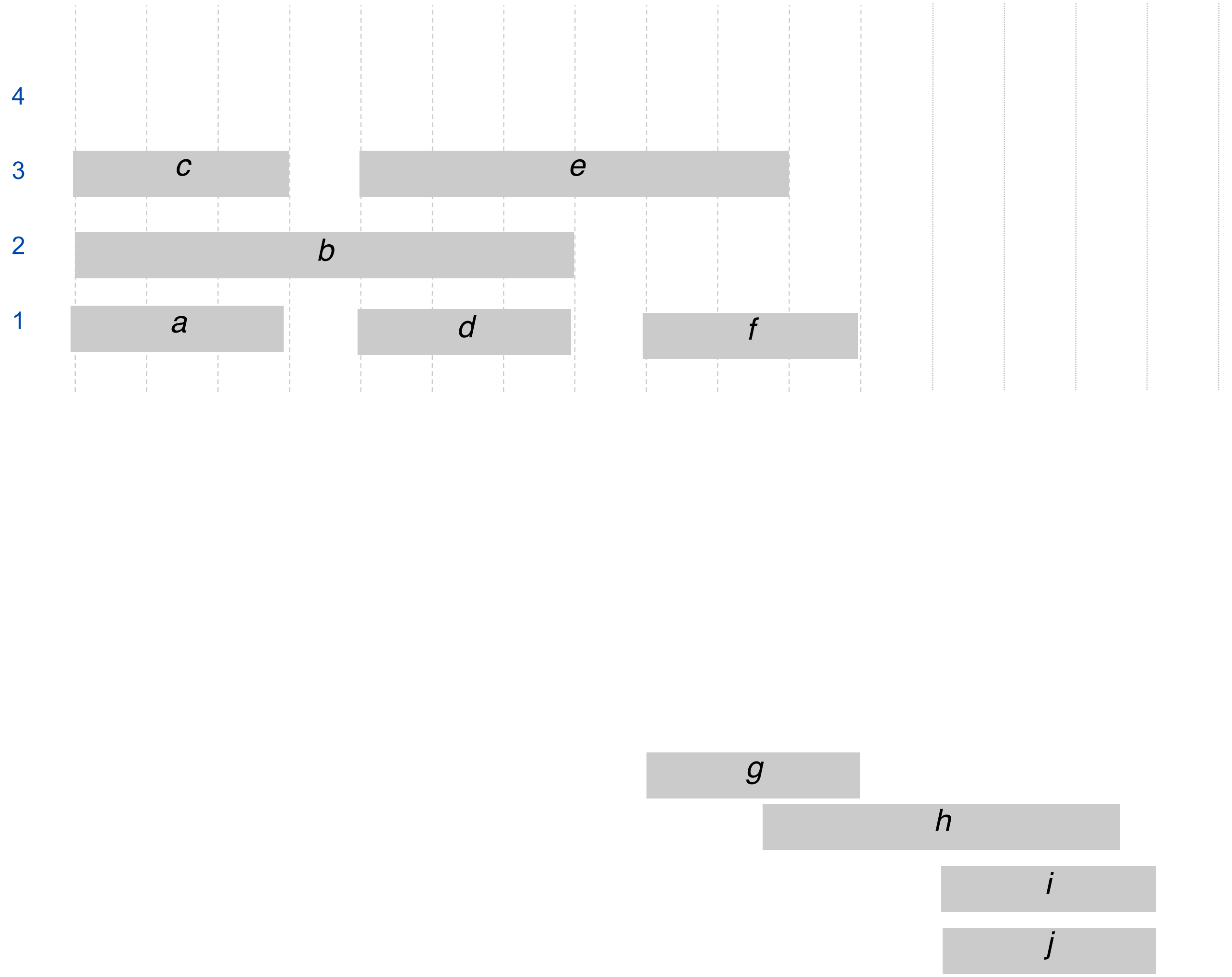
Interval partitioning: Example



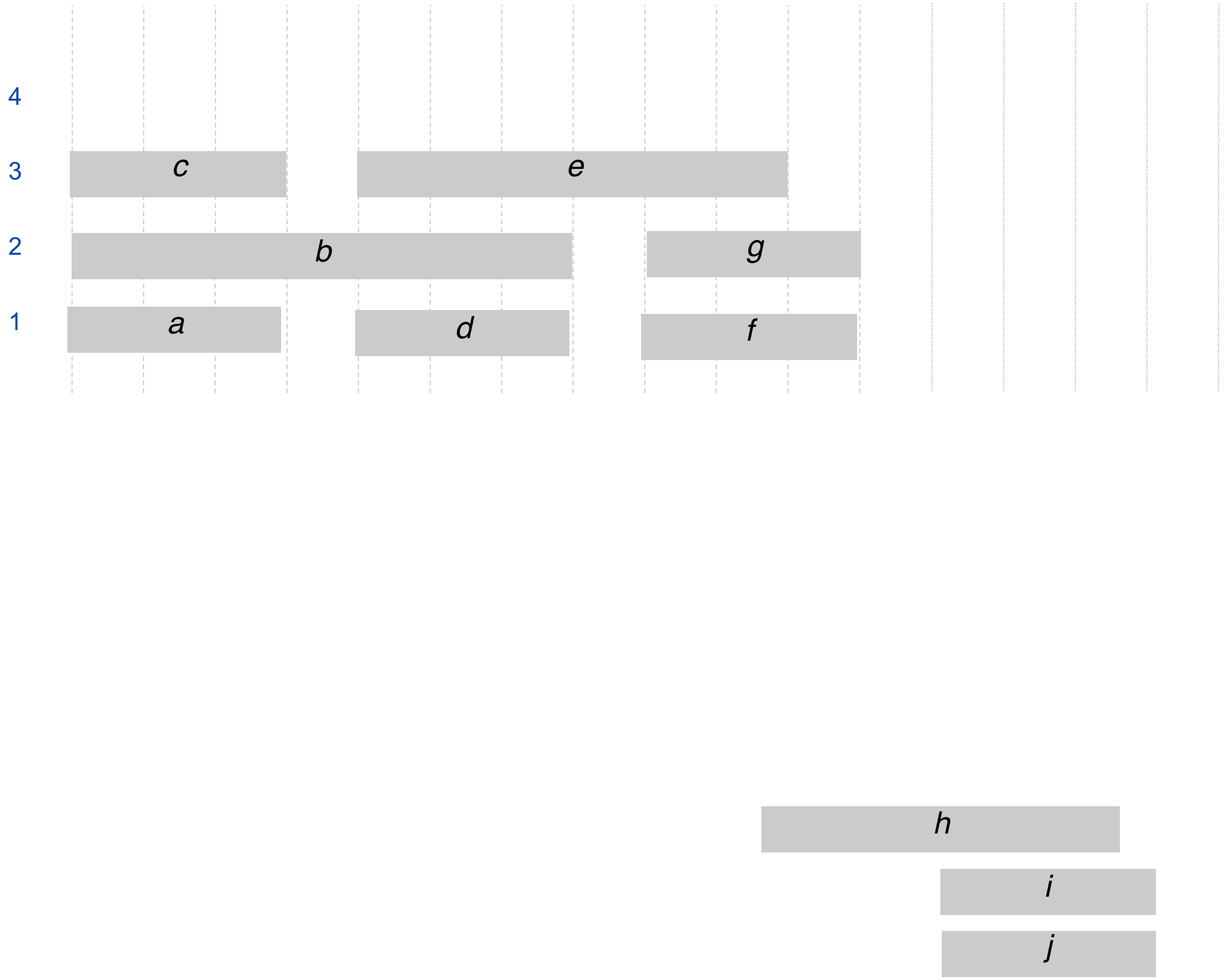
Interval partitioning: Example



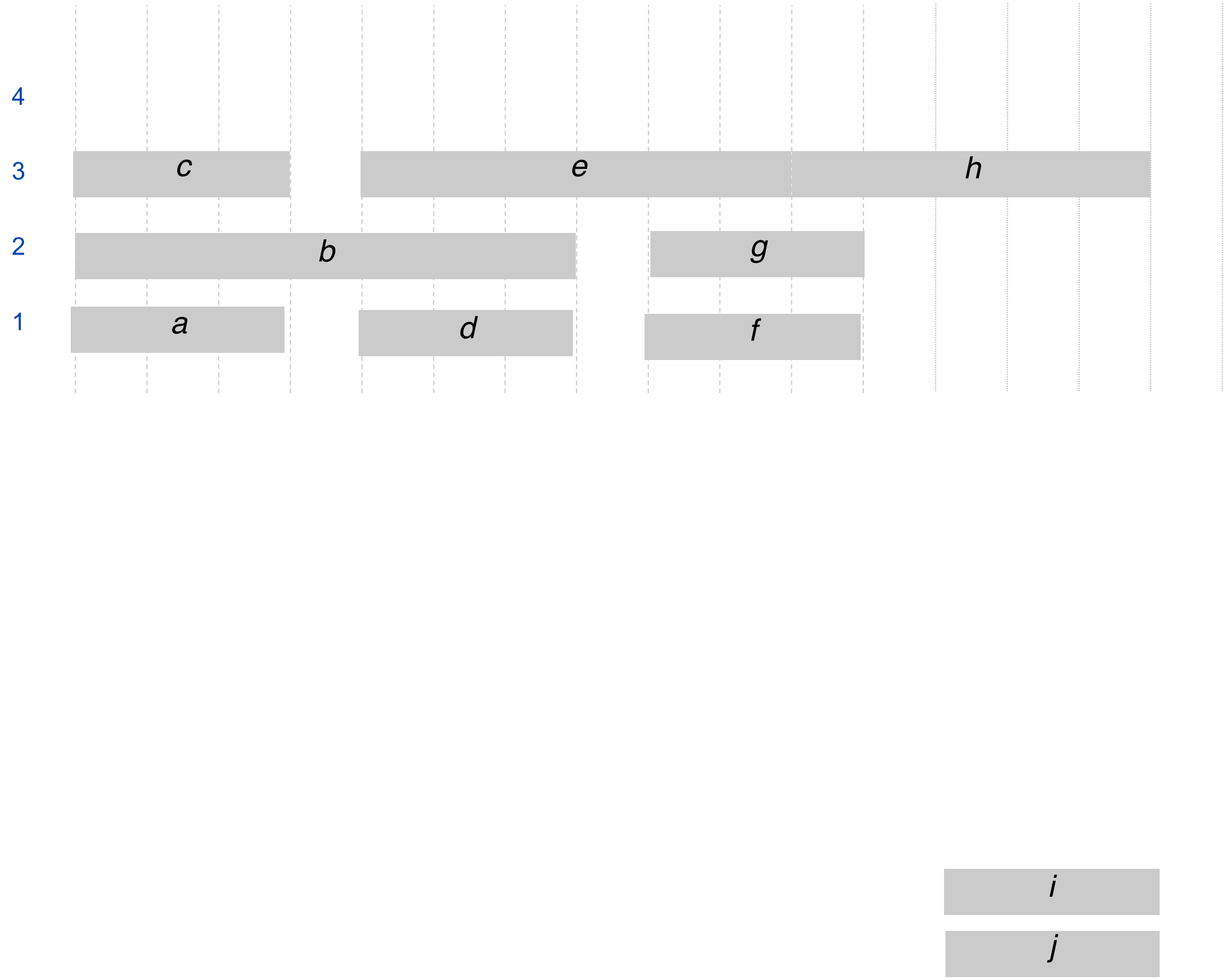
Interval partitioning: Example



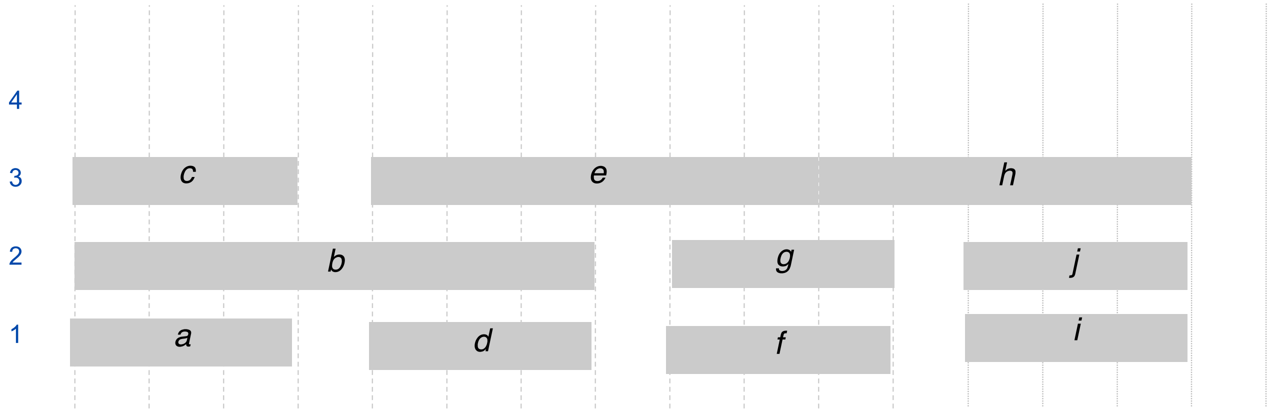
Interval partitioning: Example



Interval partitioning: Example



Interval partitioning: Example



Time

time

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).

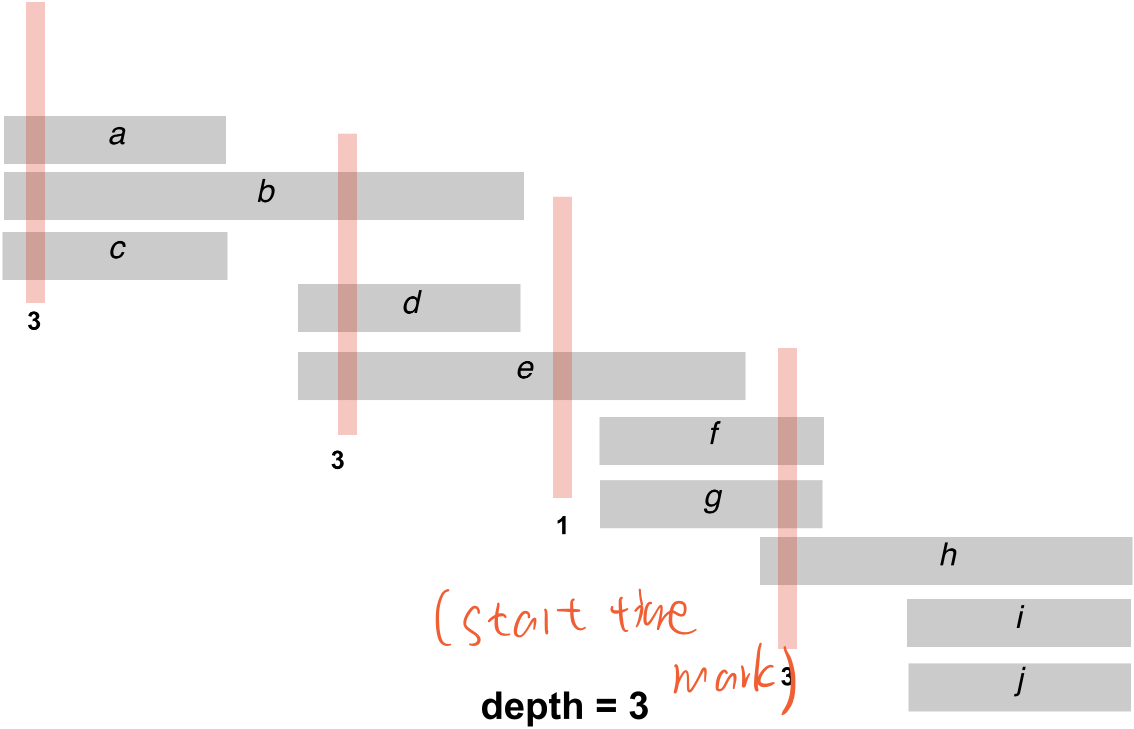
- To determine whether lecture j is compatible with some classroom, compare s_j to key of min classroom k in priority queue.
- To add lecture j to classroom k, increase key of classroom k to f_j.
- Total number of priority queue operations is $O(n)$.
- Sorting by start times takes $O(n \log n)$ time.

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.



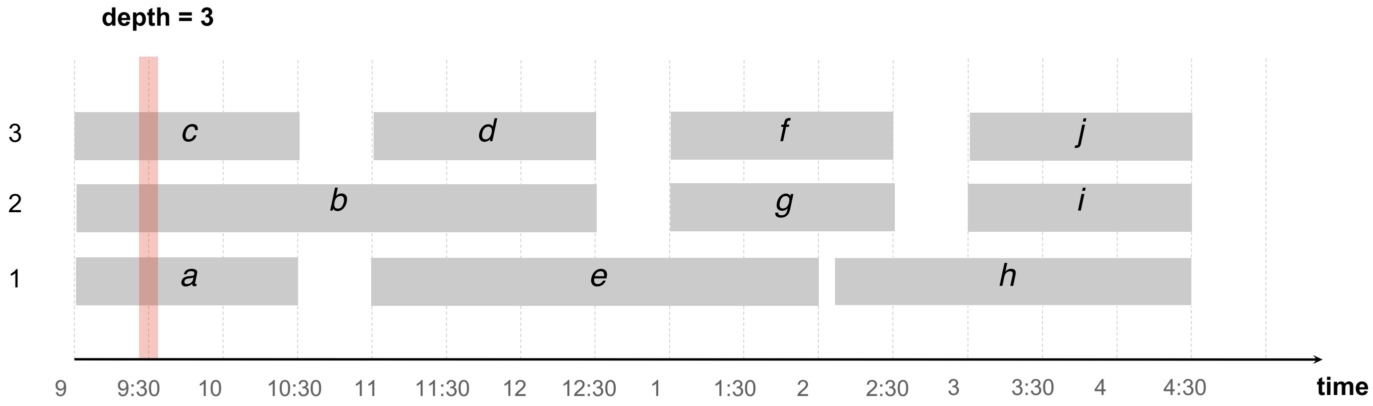
Interval partitioning: lower bound on optimal solution

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 $s_j + \boxed{?}$ ε
- Key observation $\boxed{?}$ all schedules use $\boxed{?}$ d classrooms.

$\Rightarrow$ $\geq$

optimal 证

Outline

- 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 $L = \max_j \ell_j$.

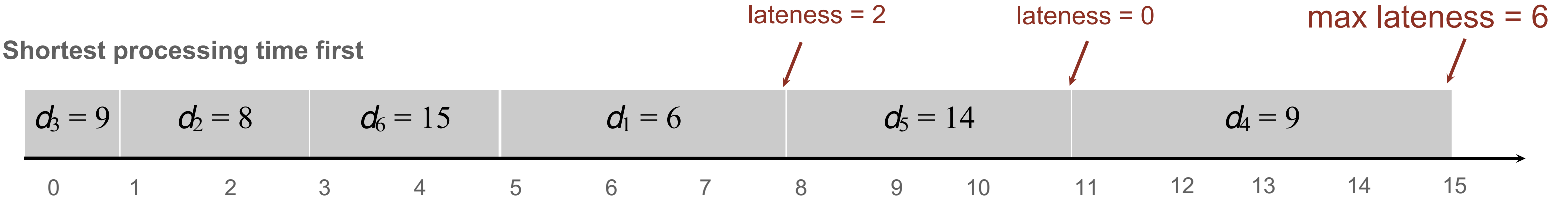
j

$s_j + t_j \rightarrow f_j$

min → max

d_j

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


Shortest processing time first

lateness = 2

lateness = 0

max lateness = 6

Minimizing lateness: greedy algorithms

Greedy template. Schedule jobs according to some natural order.

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

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

富余

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

Minimizing lateness: greedy algorithms

Greedy template. Schedule jobs according to some natural order.

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

	1	2
t_j	1	10
d_j	100	10

counterexample

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

	1	2
t_j	1	10
d_j	2	10

counterexample

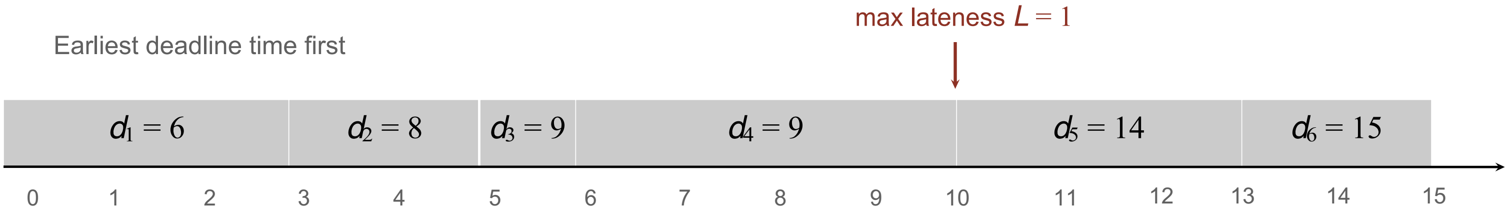
Minimizing lateness: earliest deadline first

```
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 $d_1 \le d_2 \le \ldots \le d_n$.
$t \leftarrow 0$.
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 $[s_1, f_1], [s_2, f_2], \ldots, [s_n, f_n]$.
```

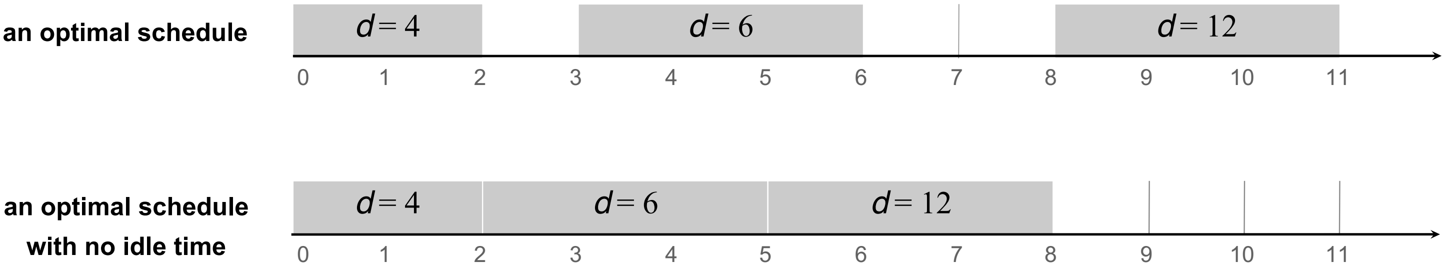
	1	2	3	4	5	6
t_j	3	2	1	4	3	2
d_j	6	8	9	9	14	15

Earliest deadline time first



Minimizing lateness: no idle time

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

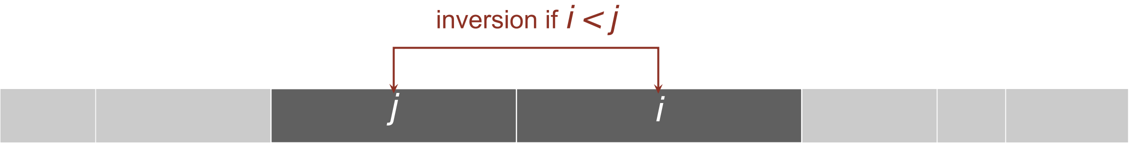


Observation 2. The earliest-deadline-first schedule has no idle time.

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 $d_1 \leq d_2 \leq \ldots \leq d_n$

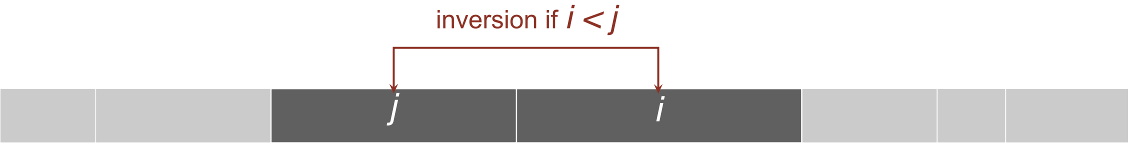
Observation 3. The earliest-deadline-first schedule is the unique idle-free schedule with no inversions.

1	2	3	4	5	6	…	n

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



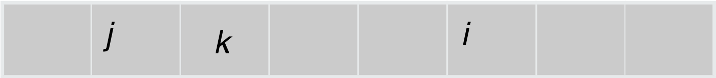
recall: we assume the jobs are numbered so that $d_1 \leq d_2 \leq \ldots \leq d_n$

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

two inverted jobs scheduled consecutively

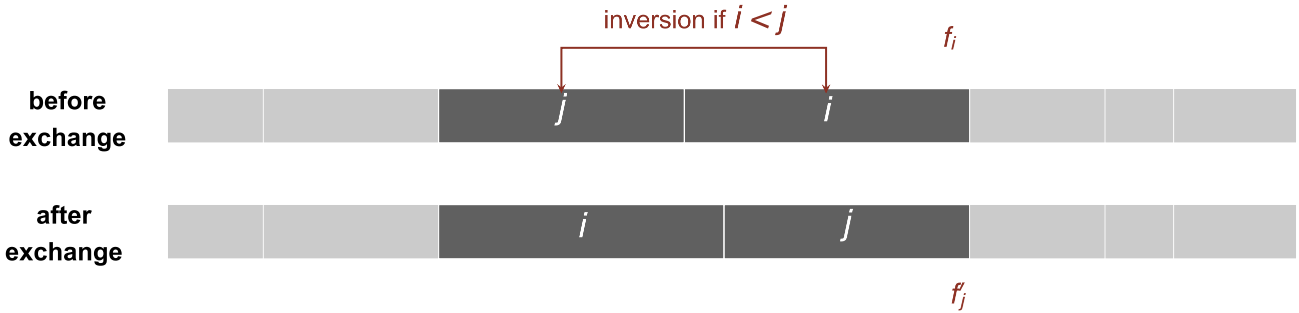
Pf.

- 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 $k \neq i, j$.
- $\ell'_i \boxed{?} \ell_i$.
- 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. Kruskal, Dijkstra, Huffman, …

Outline

- Coin changing
- Interval scheduling
- Interval partitioning
- Scheduling to minimize lateness
- Optimal caching

Optimal offline caching

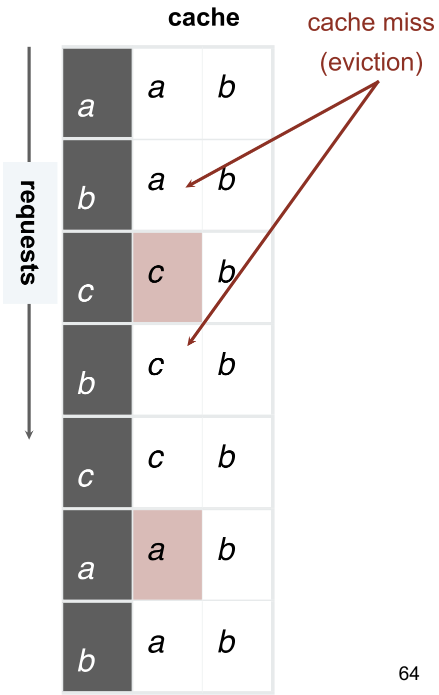
Caching.

- 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) [illegible]

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

Goal. Eviction schedule that minimizes the number of evictions.

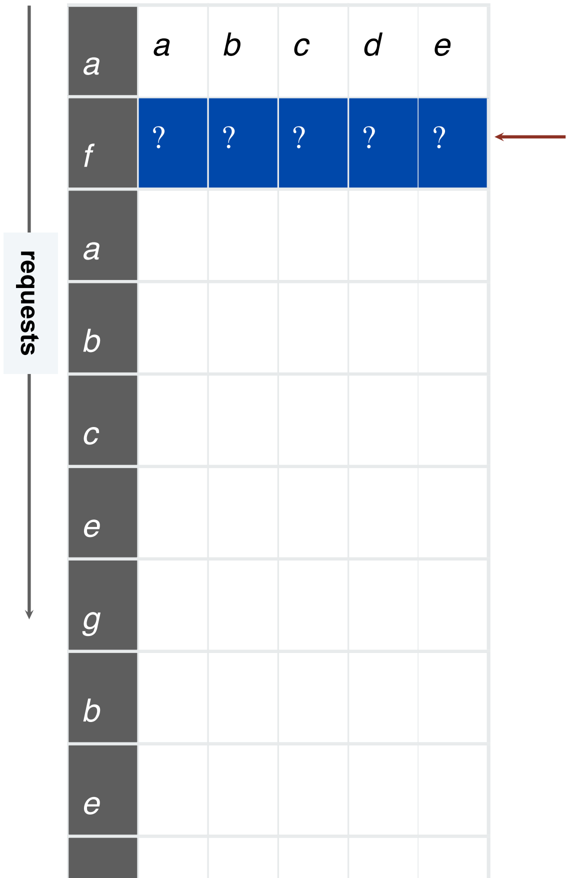
Ex. $k = 2$, initial cache = ab, requests: a, b, c, b, c, a, b.
Optimal eviction schedule. 2 evictions.



requests	cache	
a	a	b
b	a	b
c	c	b
b	c	b
c	c	b
a	a	b
b	a	b

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.



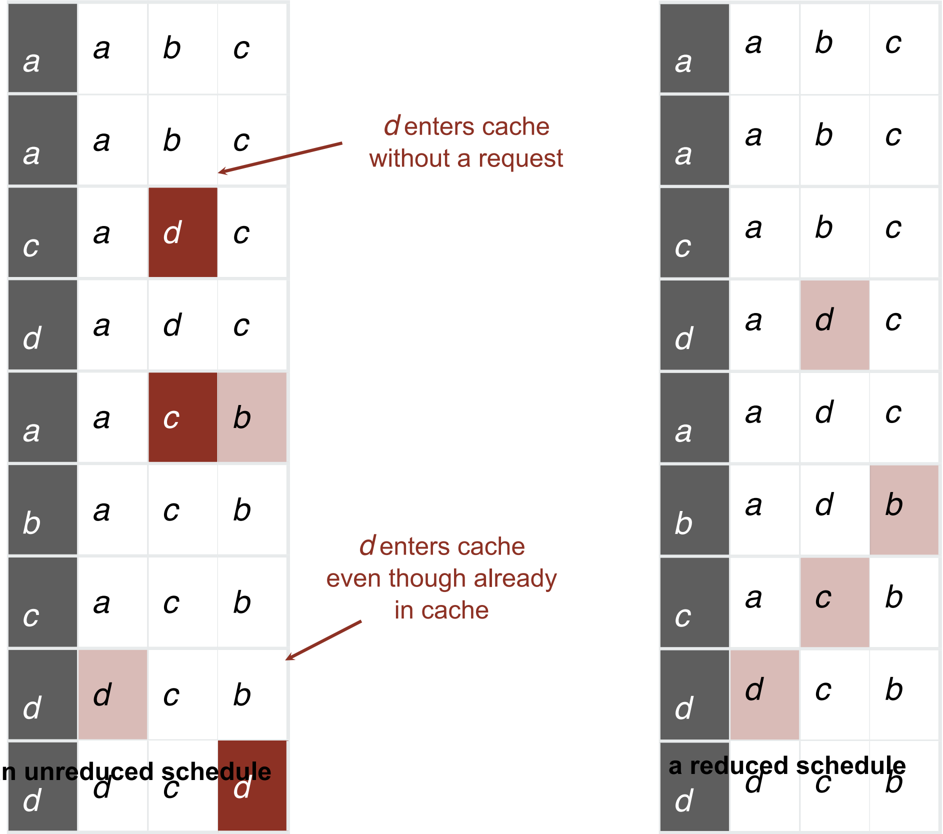
requests	cache				
a	*a*	*b*	*c*	*d*	*e*
f	?	?	?	?	?
a					
b					
c					
e					
g					
b					
e					

cache miss
(which item to eject?)

FF: eject d

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

an unreduced schedule

d enters cache without a request

d enters cache even though already in cache

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

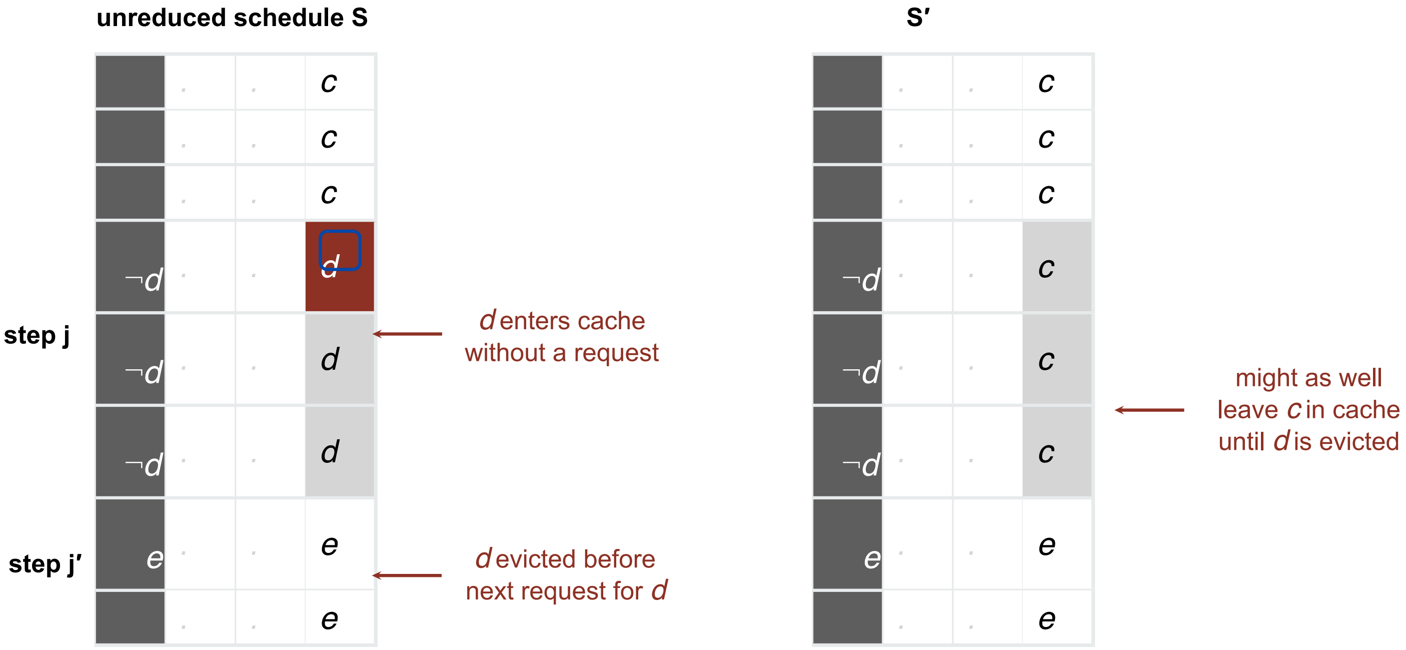
a reduced schedule

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*.



unreduced schedule S

S′

d enters cache without a request

d evicted before next request for *d*

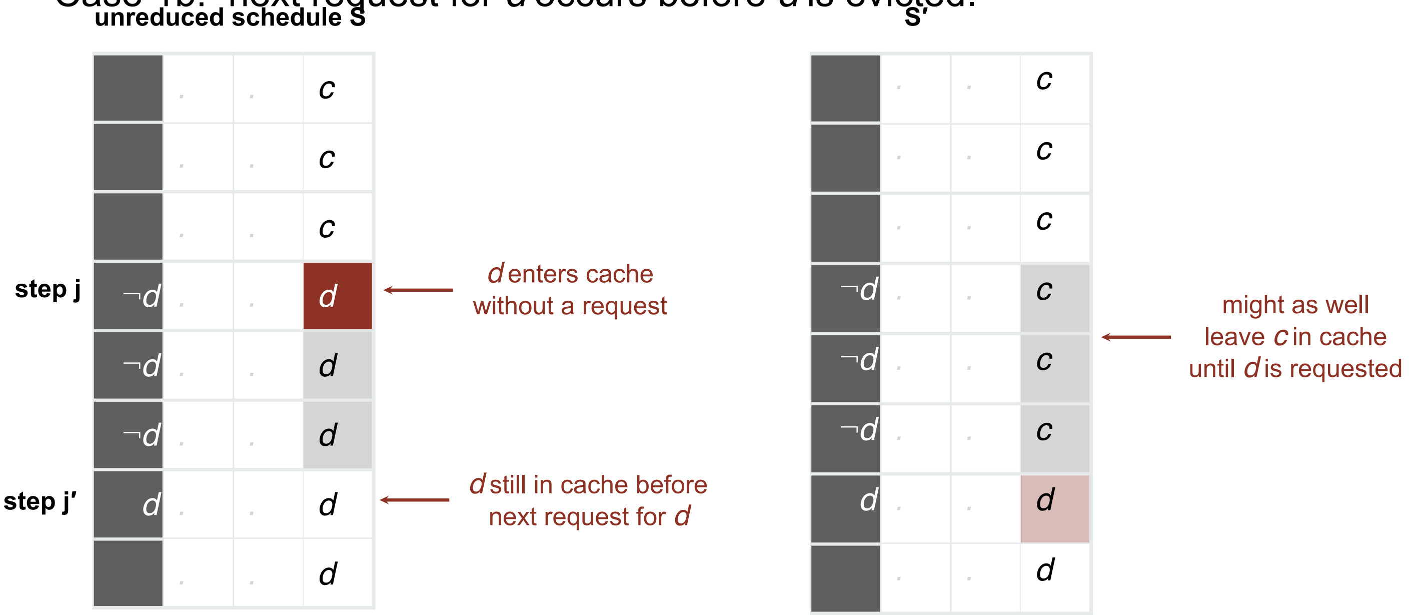
might as well leave *c* in cache until *d* is evicted

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* 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.



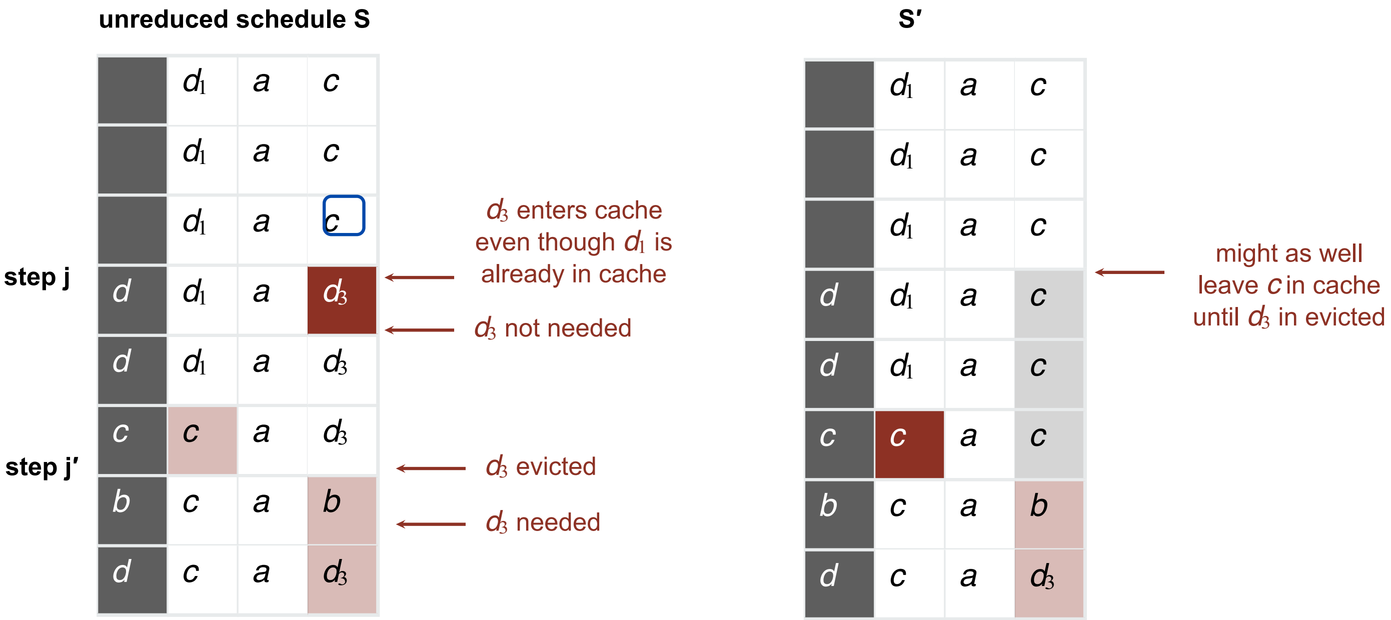
unreduced schedule S
S′
step j
step j′
c
c
c
¬d
d
¬d
d
¬d
d
d
d
d
d
c
c
c
¬d
c
¬d
c
¬d
c
d
d
d
d enters cache without a request
d still in cache before next request for *d*
might as well leave *c* in cache until *d* is requested

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	c
step j	d	d_1	a	d_3
	d	d_1	a	d_3
	c	c	a	d_3
step j′	b	c	a	b
	d	c	a	d_3

d_3 enters cache even though d_1 is already in cache

d_3 not needed

d_3 evicted

d_3 needed

S′

	d_1	a	c
	d_1	a	c
	d_1	a	c
d	d_1	a	c
d	d_1	a	c
c	c	a	c
b	c	a	b
d	c	a	d_3

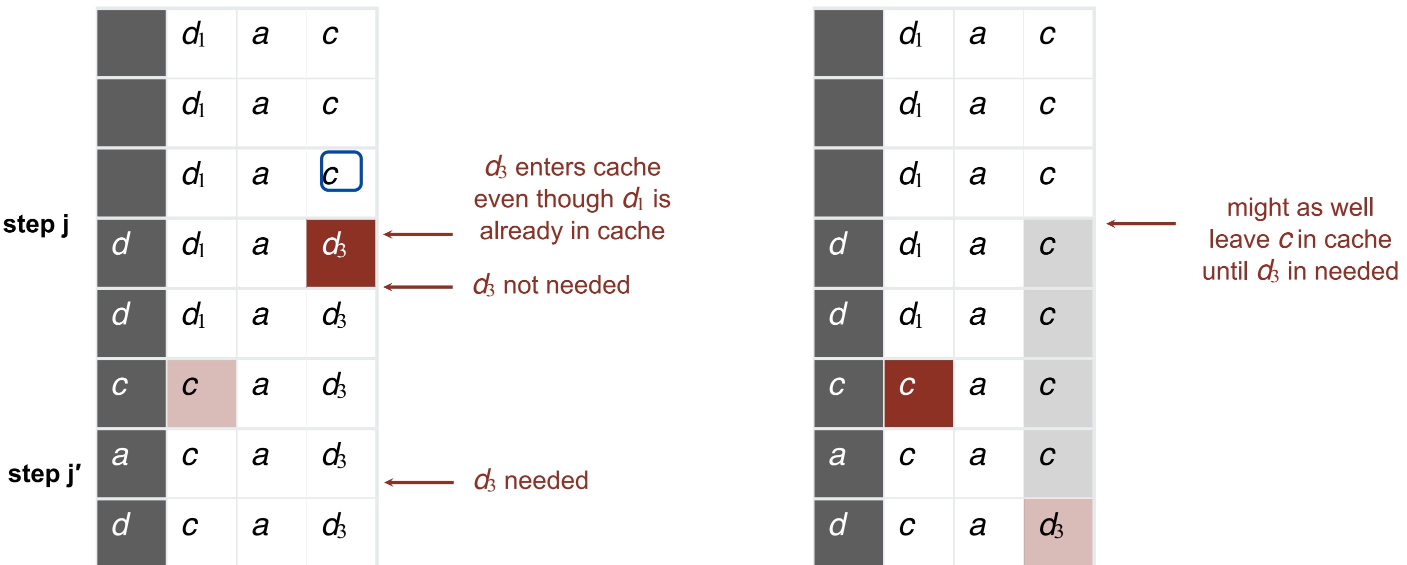
might as well leave c in cache until d_3 in evicted

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.
- Case 2b: *d* needed before it is evicted.



unreduced schedule S

		d_1	a	c
		d_1	a	c
		d_1	a	c
step j	d	d_1	a	d_3
	d	d_1	a	d_3
	c	c	a	d_3
step j′	a	c	a	d_3
	d	c	a	d_3

d_3 enters cache even though d_1 is already in cache

d_3 not needed

d_3 needed

S′

	d_1	a	c
	d_1	a	c
	d_1	a	c
d	d_1	a	c
d	d_1	a	c
c	c	a	c
a	c	a	c
d	c	a	d_3

might as well leave *c* in cache until d_3 in needed

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.

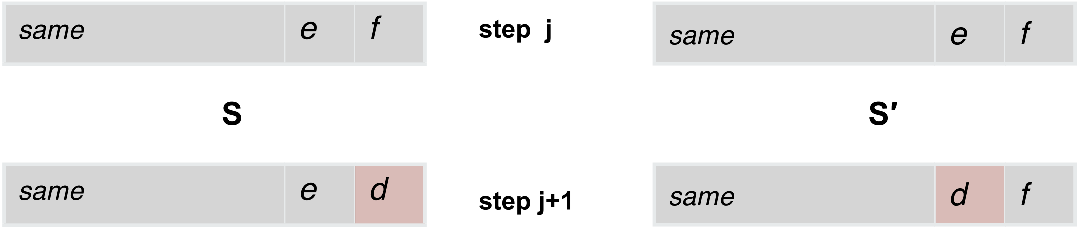
We produce S' that satisfies invariant after $j + 1$ steps.

- 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



same | *e* | *f* — step j — *same* | *e* | *f*

S — S'

same | *e* | *d* — step j+1 — *same* | *d* | *f*

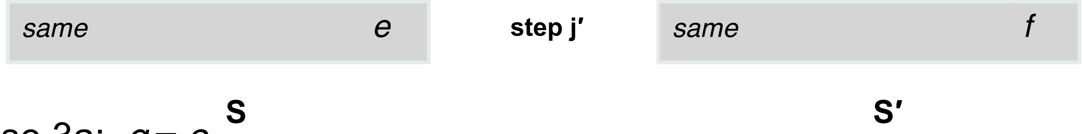
  - now S' agrees with S_{FF} for first $j+1$ steps; we show that having item f in cache is no worse than having item e in cache

  - 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

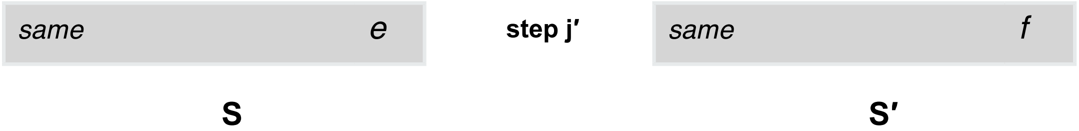
  We let S' behave exactly like S for remaining requests.

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

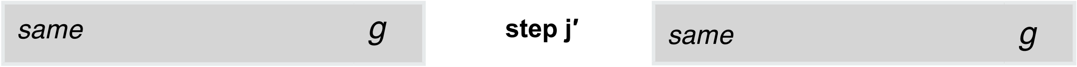
Let j' be the first step after $j+1$ that S' must take a different action from S; let g denote the item requested in step j'.

involves wither e or f (or both)



otherwise S' could have taken the same action

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



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