Algorithms Design
Chap04-Greedy Algorithms

College of Computer Science

Nankai University

Tianjin, P.R.China

Chap04-Greedy Algorithms Outline

Greedy Algorithms

Greedy is a strategy that works well on optimization problems with the following characteristics:

- Greedy choice property
- Optimal substructure

Optimal substructure

A problem has the optimal substructure if an optimal solution to the problem contains optimal solutions to the subproblems.

That means we can solve subproblems and build up the solutions to solve larger problems.

Greedy choice property

The globally optimal solution can be obtained by making a greedy choice which can provide a locally optimal solution.

The choice made by a Greedy algorithm may depend on earlier choices but not on the future.

It iteratively makes one Greedy choice after another and reduces the given problem to a smaller one.

The general steps of a greedy algorithm

1) Identify the problem as an optimization problem where we need to find the best solution among a set of possible solutions.

2) Determine the set of feasible solutions for the problem.

The general steps of a greedy algorithm

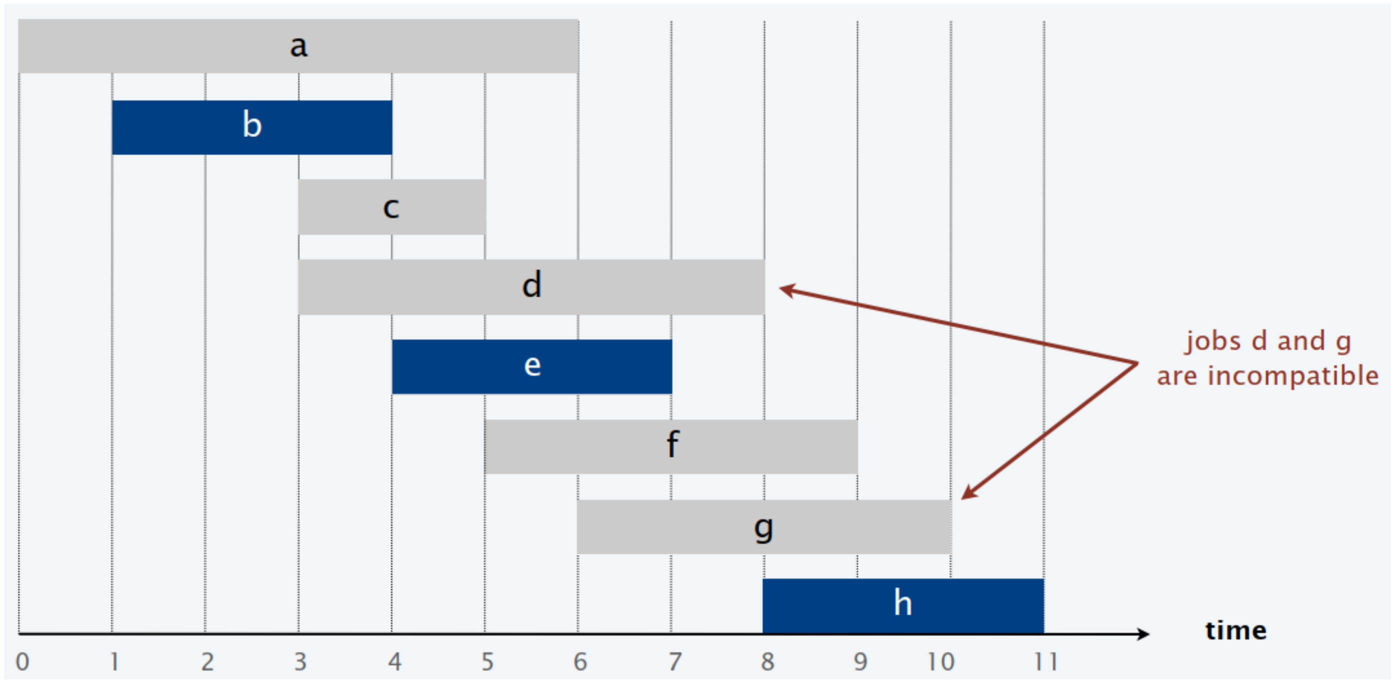
3) Identify the optimal substructure of the problem.

4) Develop a greedy strategy to construct a feasible solution step by step, making the locally optimal choice at each step.

5) Prove the correctness of the algorithm by showing that the locally optimal choices at each step lead to a globally optimal solution.

4.1 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.



Quiz 4-1-1

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.

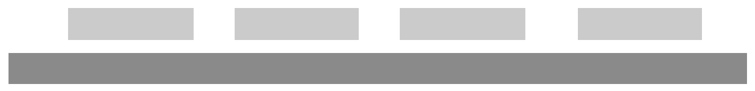
C. [Shortest interval] Consider jobs in ascending order of $f_j - s_j$.

D. [Fewest conflicts] For each job j, count the number of conflicting jobs c_j. Schedule in ascending order of c_j.

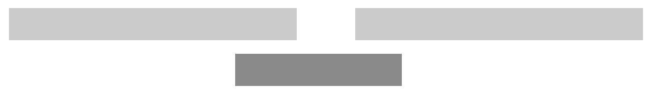
4.1 Interval Scheduling

Greedy Choice.

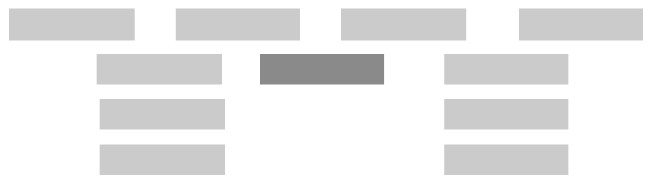
- Consider jobs in some natural order. Take each job provided it's compatible with the ones already taken.



counterexample for earliest start time



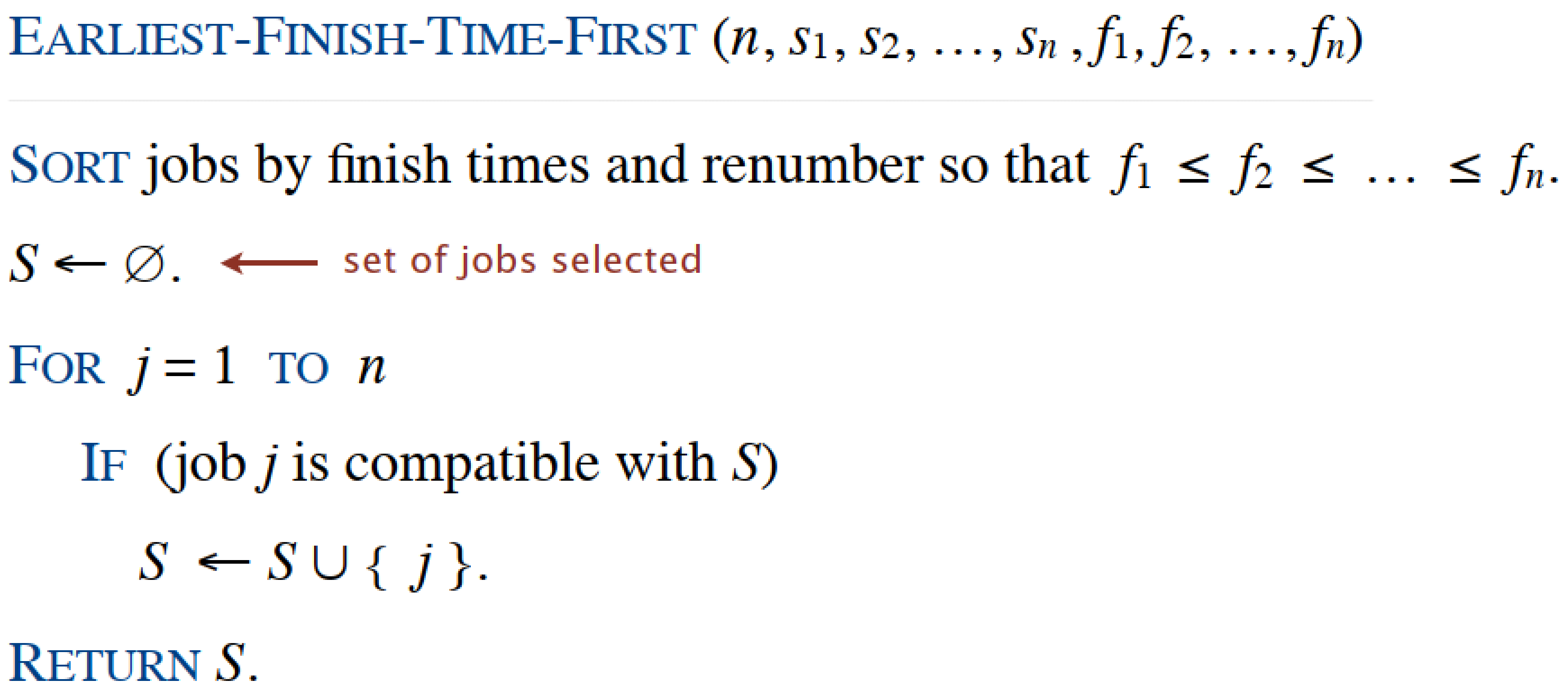
counterexample for shortest interval



counterexample for fewest conflicts

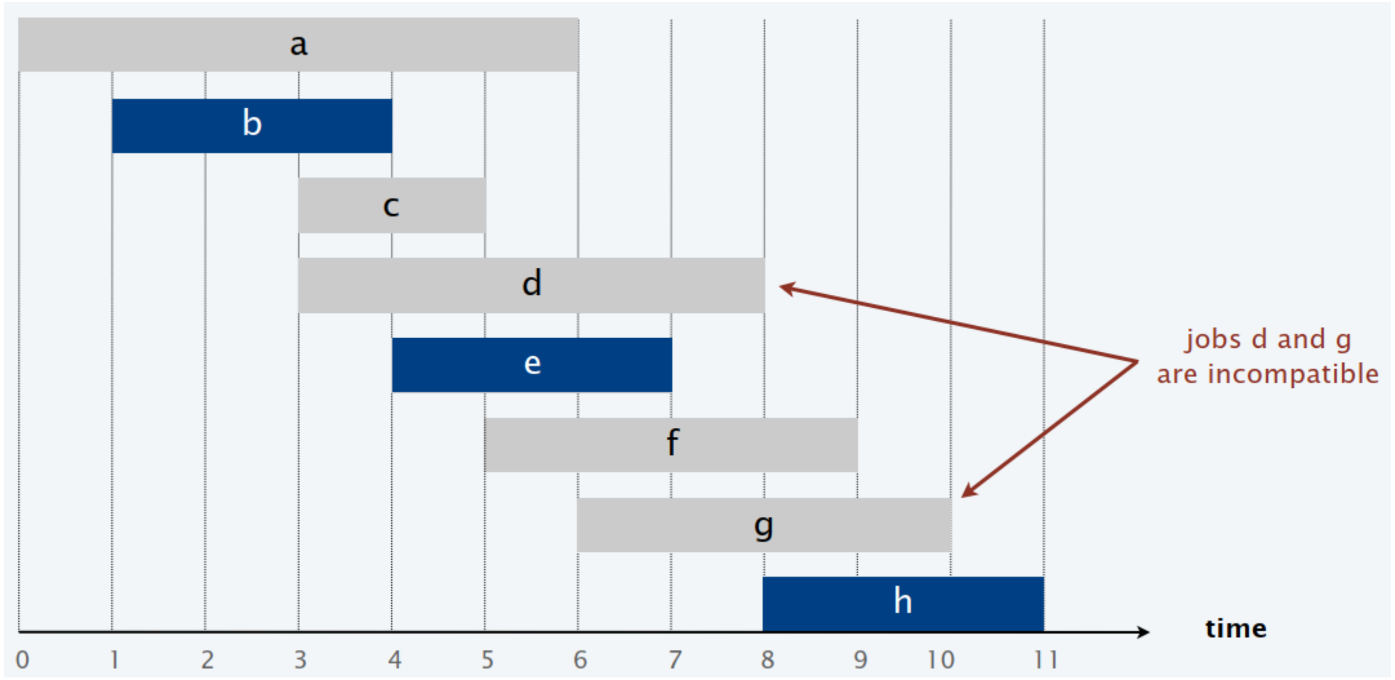
4.1 Interval Scheduling

Earliest-Finish-Time-First(EFTF) 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 $f_1 \le f_2 \le \ldots \le f_n$.
$S \leftarrow \varnothing$.   ← set of jobs selected
FOR $j = 1$ TO $n$
    IF (job $j$ is compatible with $S$)
        $S \leftarrow S \cup \{ j \}$.
RETURN $S$.
```

4.1 Interval Scheduling



$b < c < a < e < d < f < g < h$

$S = \{b\}, S = \{b, e\}, S = \{b, e, h\}$

4.1 Interval Scheduling

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.

4.1 Interval Scheduling

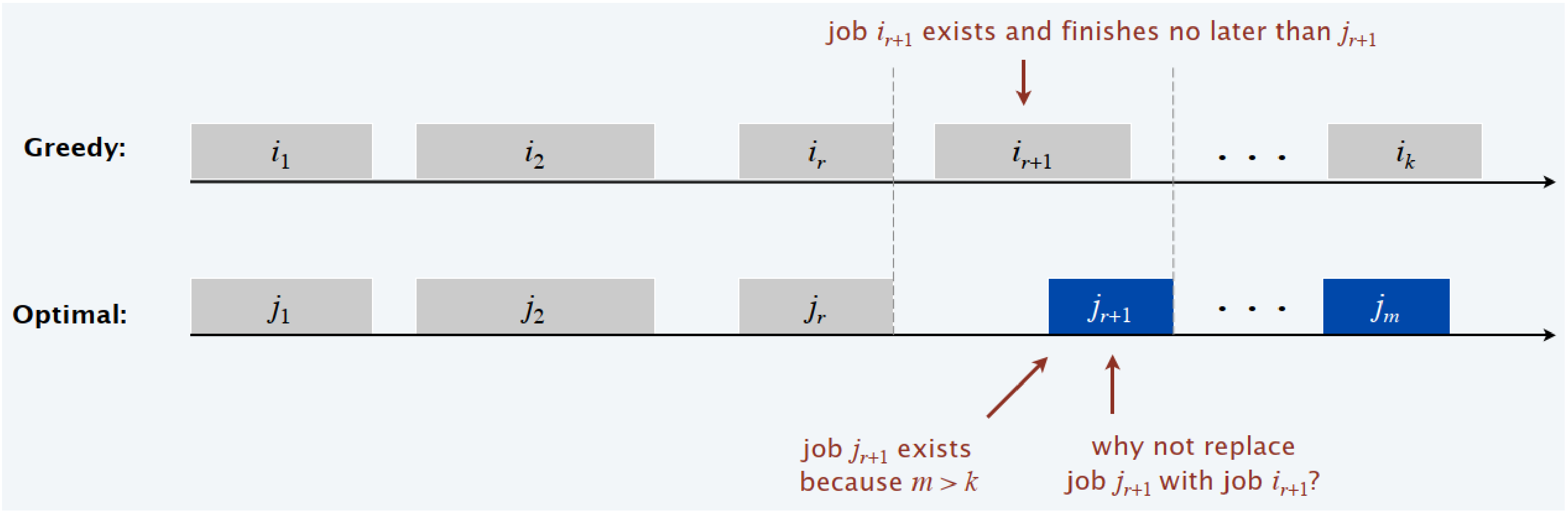
Theorem. The earliest-finish-time-first algorithm is optimal.

Pf. [by contradiction]

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

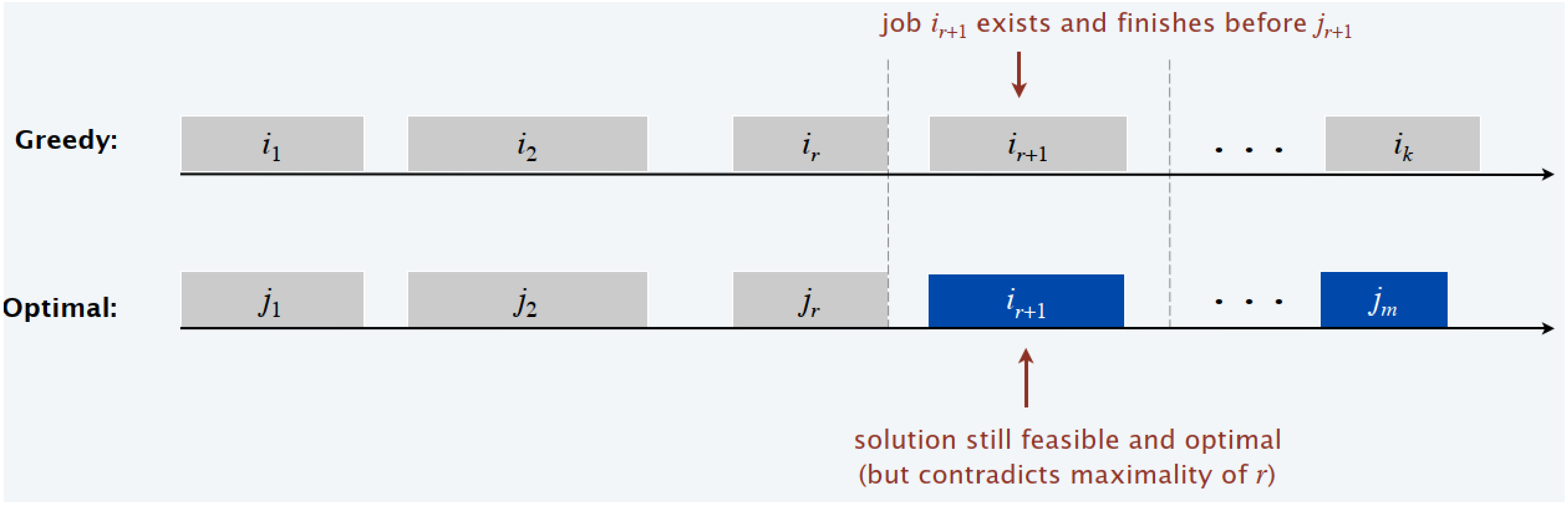
4.1 Interval Scheduling

E.g.


job i_{r+1} exists and finishes no later than j_{r+1}
Greedy:
i_1
i_2
i_r
i_{r+1}
i_k
Optimal:
j_1
j_2
j_r
j_{r+1}
j_m
job j_{r+1} exists because $m > k$
why not replace job j_{r+1} with job i_{r+1}?

4.1 Interval Scheduling

E.g.


job i_{r+1} exists and finishes before j_{r+1}
Greedy:
i_1
i_2
i_r
i_{r+1}
. . .
i_k
Optimal:
j_1
j_2
j_r
i_{r+1}
. . .
j_m
solution still feasible and optimal
(but contradicts maximality of r)

Quiz 4-1

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.

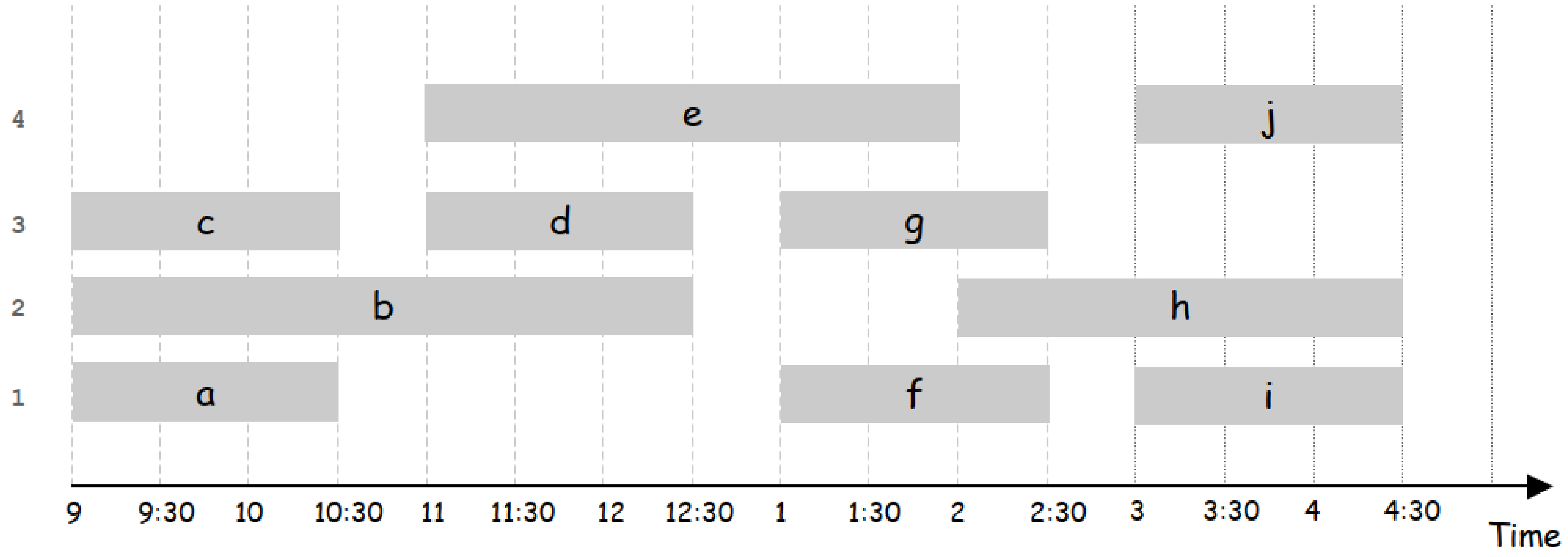
4.1 Interval Partitioning

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 occur at the same time in the same room.

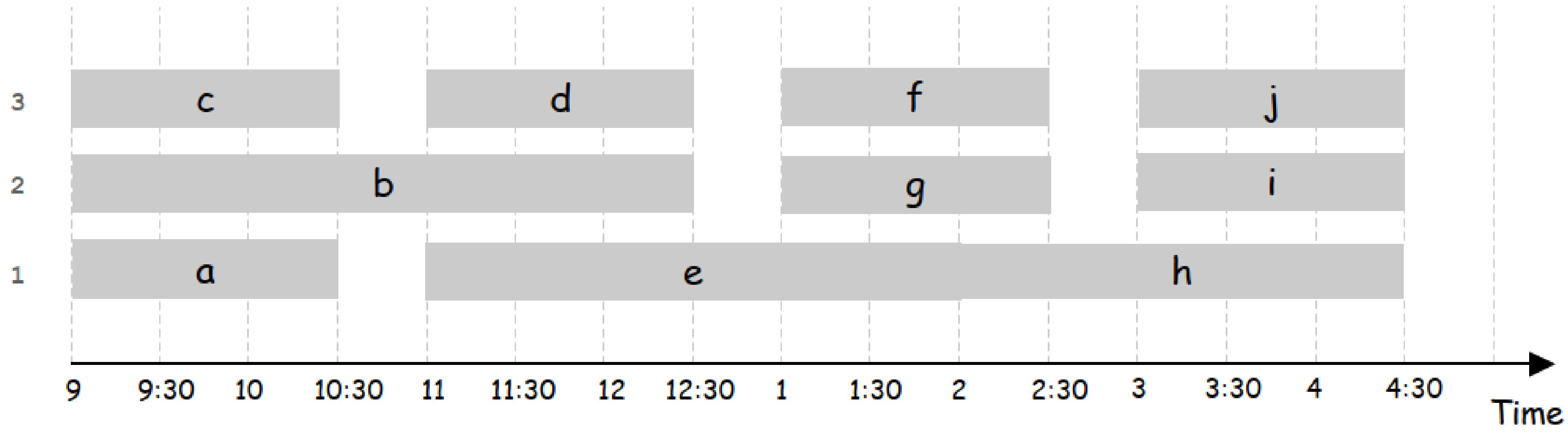
4.1 Interval Partitioning

E.g. This schedule uses 4 classrooms to schedule 10 lectures.



4.1 Interval Partitioning

E.g. This schedule uses 3 classrooms to schedule 10 lectures.



Quiz 4-2

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. [Fewest conflicts] For each lecture j, count the number of conflicting lectures c_j. Schedule in ascending order of c_j.

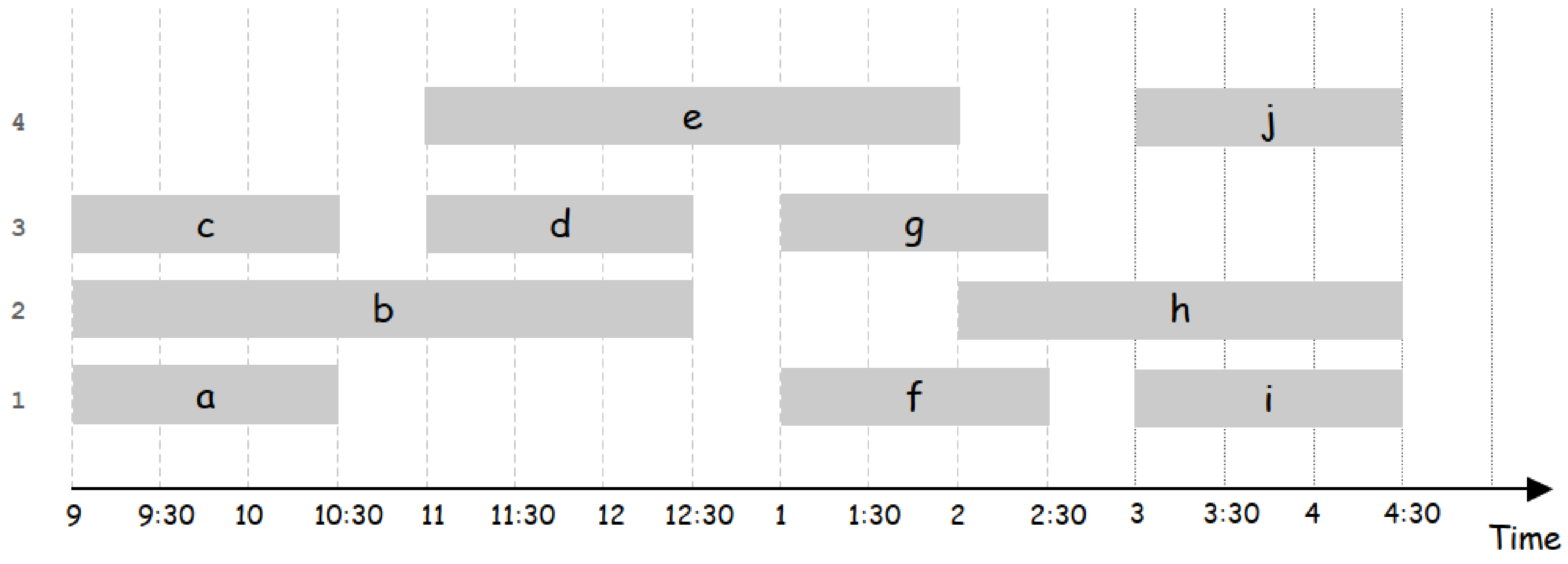
4.1 Interval Partitioning

```
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.
```

4.1 Interval Partitioning

E.g.



4.1 Interval Partitioning

Allocate new classroom 1

Schedule lecture a(9) in classroom 1

Allocate new classroom 2

Schedule lecture b(9) in classroom 2

Allocate new classroom 3

Schedule lecture c(9) in classroom 3

4.1 Interval Partitioning

Schedule lecture d(11) in classroom 3/1

Schedule lecture e(11) in classroom 1/3

Schedule lecture f(13) in classroom 3/2

Schedule lecture g(13) in classroom 2/3

Schedule lecture h(14) in classroom 1

Schedule lecture i(15) in classroom 2/3

Schedule lecture j(15) in classroom 3/2

4.1 Interval Partitioning

Greedy Choice.

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

Implementation. $O(n \log n)$.

- For each classroom k, maintain the finish time of the last job added.
- Keep the classrooms in a priority queue.

4.1 Interval Partitioning

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

Pf.

- Sorting by start times takes $O(n \log n)$ time.
- Store classrooms in a priority queue (key = finish time of its last lecture).
    - to allocate a new classroom, INSERT classroom onto priority queue.
    - to schedule lecture j in classroom k, INCREASE-KEY of classroom k to f_j.
    - to determine whether lecture j is compatible with any classroom, compare s_j to FIND-MIN
- Total # of priority queue operations is $O(n)$; each takes $O(\log n)$ time.

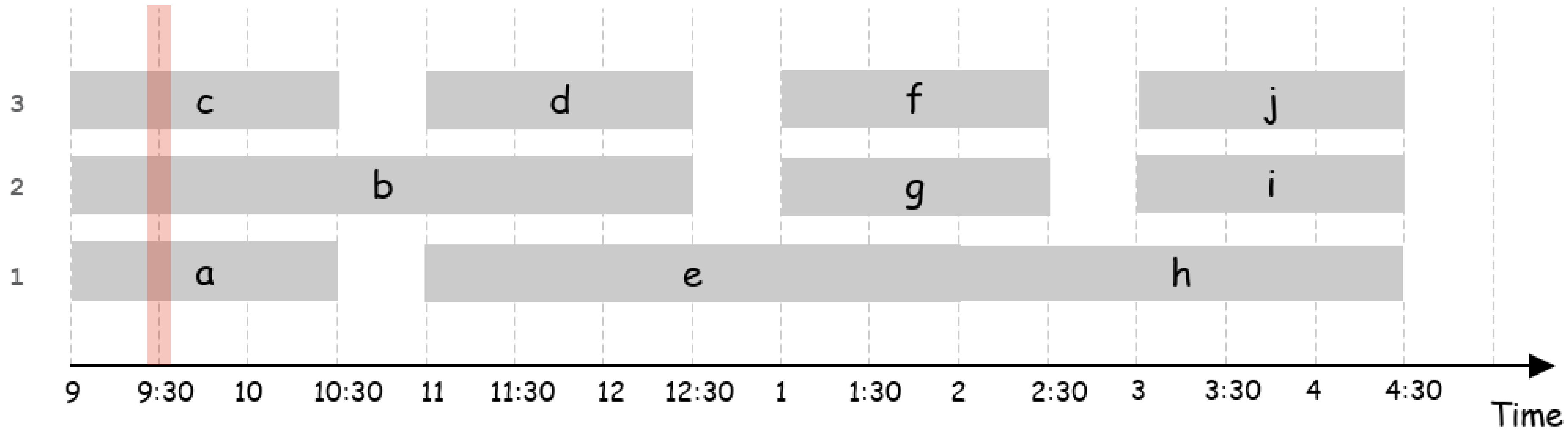
4.1 Interval Partitioning

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

Observation. Number of classrooms needed ≥ depth.

4.1 Interval Partitioning

E.g. Depth of schedule below = 3 $\Rightarrow$ schedule below is optimal.



4.1 Interval Partitioning

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.

4.1 Interval Partitioning

Observation. The Earliest-Start-Time First(ESTF) algorithm never schedules two incompatible lectures in the same classroom.

4.1 Interval Partitioning

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. $s_j < f_{i \le d}$
- Since we sorted by start time, each of these incompatible lectures start no later than s_j. $s_{i \le d} < s_j$
- Thus, we have d lectures overlapping at time $s_j + \varepsilon$.
- Key observation $\Rightarrow$ all schedules use $\geq d$ classrooms.

Chap04-Greedy Algorithms Outline

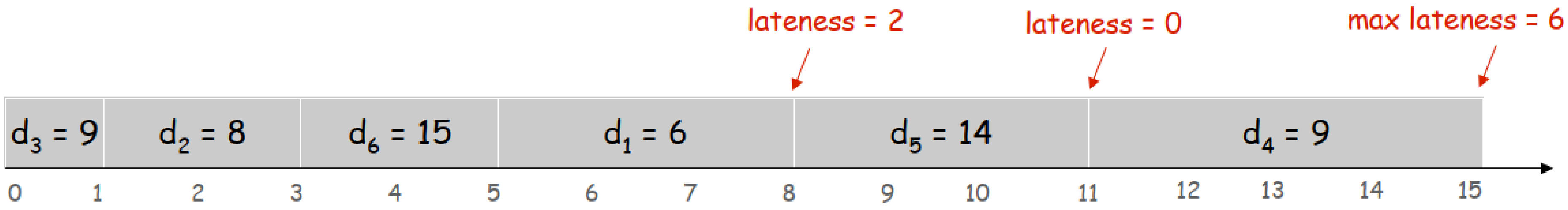
4.2 Scheduling to Minimize Lateness

- 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 $L = max_j \ \ell_j$.

4.2 Scheduling to Minimize Lateness

E.g.

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



Quiz 4-3

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.

4.2 Scheduling to Minimize Lateness

Greedy Choice. Consider jobs in some order.

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

	1	2
t_j	1	10
d_j	100	10

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

	1	2
t_j	1	10
d_j	2	10

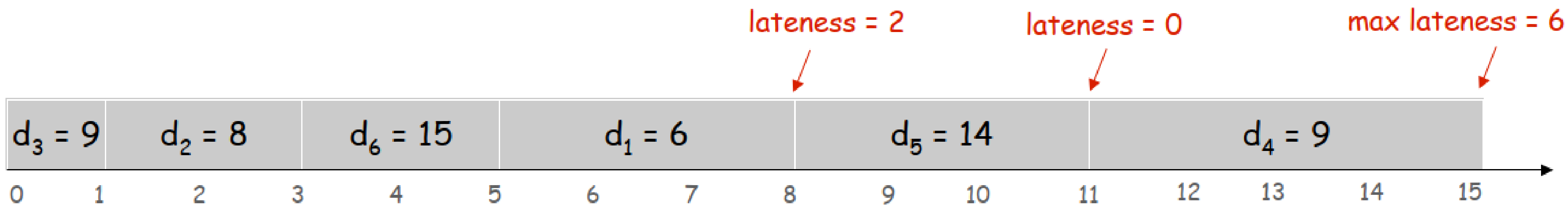
4.2 Scheduling to Minimize Lateness

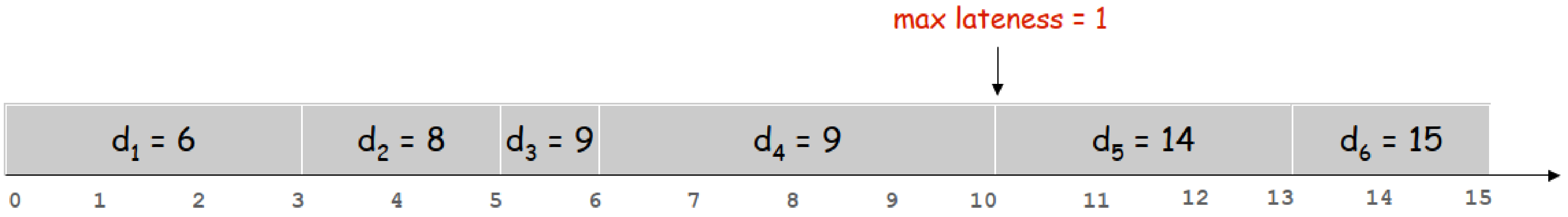
```
EARLIEST-DEADLINE-FIRST ($n, t_1, t_2, \ldots, t_n, d_1, d_2, \ldots, d_n$)

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

4.2 Scheduling to Minimize Lateness

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





4.2 Scheduling to Minimize Lateness

Earliest-Deadline-First(EDF) Algorithms

Assign job 1(6) to $[0,0+3]$. $(t_1 = 3)$

Assign job 2(8) to $[3,3+2]$. $(t_2 = 2)$

Assign job 3(9) to $[5,5+1]$. $(t_3 = 1)$

Assign job 4(9) to $[6,6+4]$. $(t_4 = 4)$

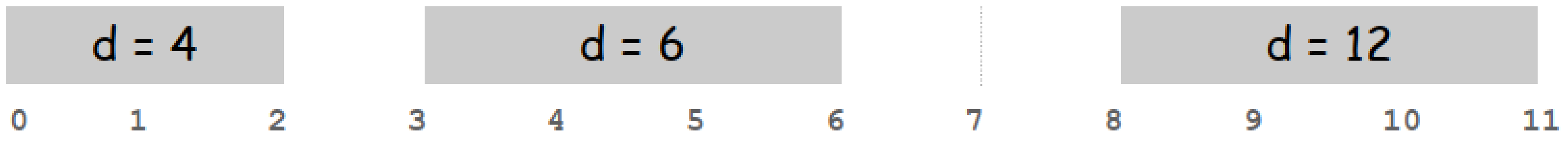
Assign job 5(14) to $[10,10+3]$. $(t_4 = 3)$

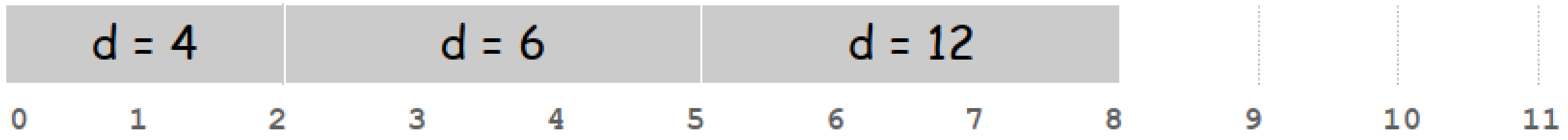
Assign job 6(15) to $[13,13+2]$. $(t_4 = 2)$

4.2 Scheduling to Minimize Lateness

Minimizing lateness: no idle time

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

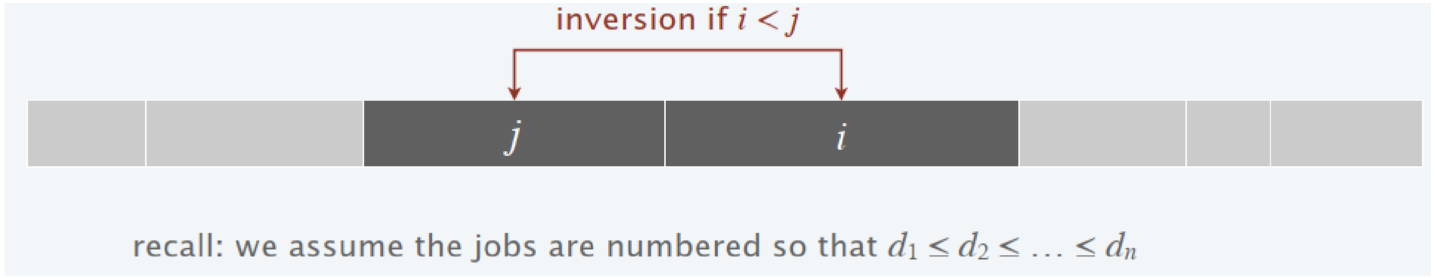




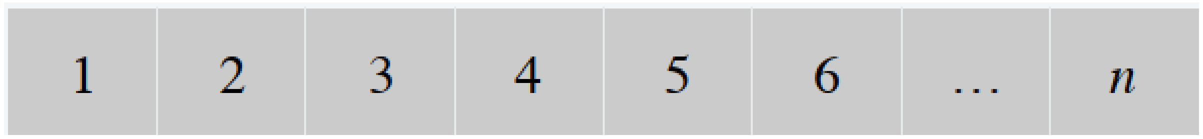
Observation 2. The earliest-deadline-first schedule has no idle time.

4.2 Scheduling to Minimize Lateness

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.


inversion if $i < j$
j
i
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.



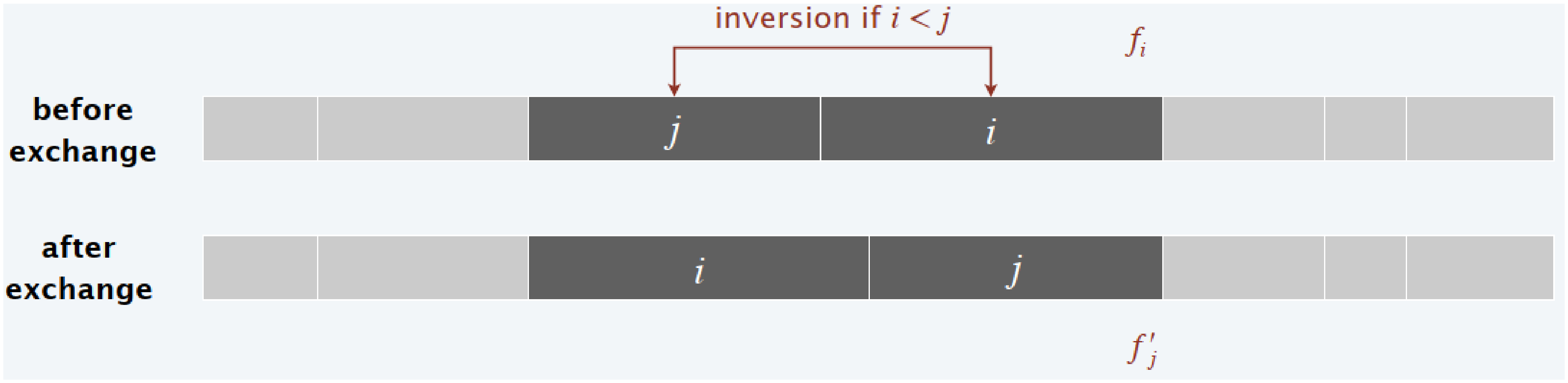
4.2 Scheduling to Minimize Lateness

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

Pf.

- Let i–j be a closest inversion. $\boxed{i < j}$
- 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 $i < j < k$.

4.2 Scheduling to Minimize Lateness



4.2 Scheduling to Minimize Lateness

Claim 4-1. 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.

- $l'_k = l_k$ for all $k \neq i, j$
- $l'_i \leq l_i$
- If job *j* is late:

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

4.2 Scheduling to Minimize Lateness

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

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 Claim(4-1)
  - contradicts "fewest inversions" part of the definition of S^*

4.2 Scheduling to Minimize Lateness

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, …

Chap04-Greedy Algorithms Outline

4.3 Optimal Caching

Caching.

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

Applications.

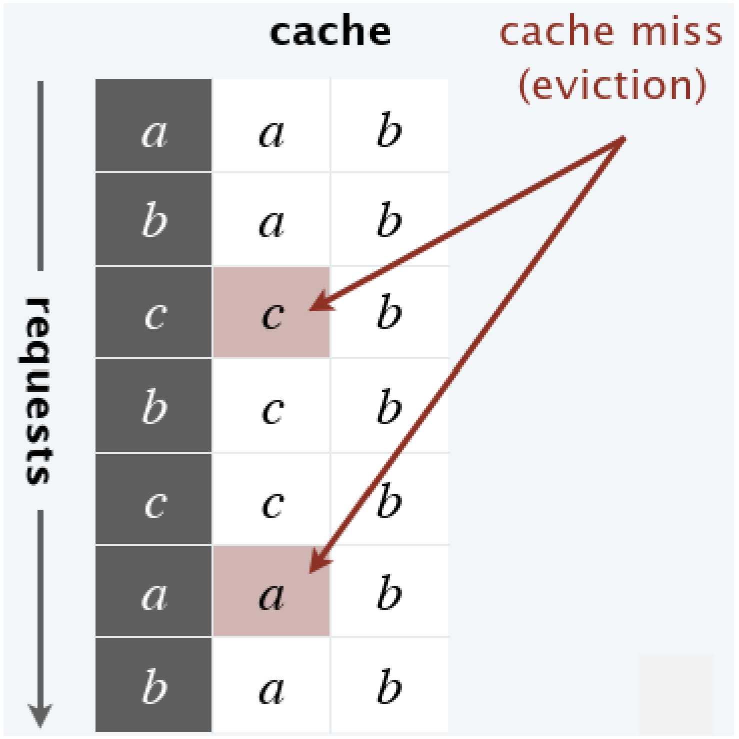
- CPU, RAM, hard drive, web, browser, ….

4.3 Optimal Caching

Goal. Eviction schedule that minimizes the number of evictions.

E.g. $k = 2$, 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

4.3 Optimal Caching

Optimal offline caching: greedy algorithms

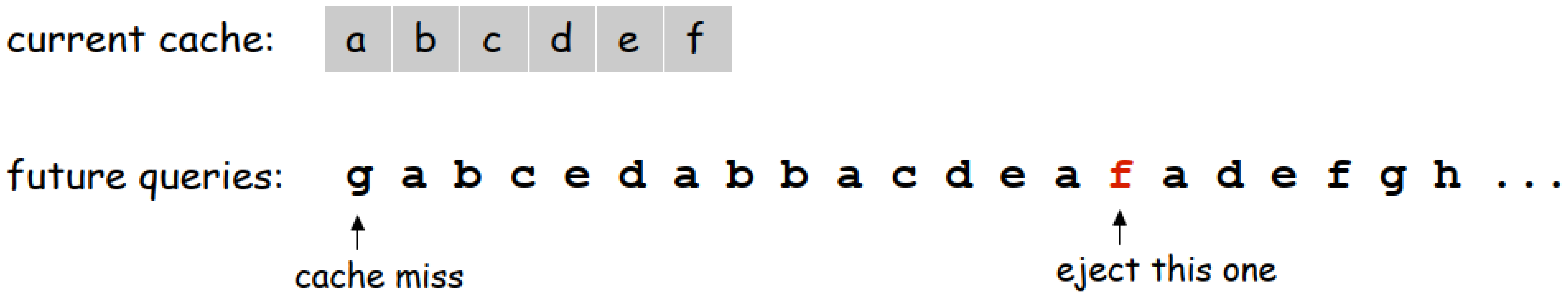
- LIFO/FIFO. Evict item brought in least (most) recently.
- LRU. Evict item whose most recent access was earliest.
- LFU. Evict item that was least frequently requested.

4.3 Optimal Caching

requests	cache					
⋮	.	.	.	.	.	
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						
⋮						

4.3 Optimal Caching

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.

Quiz 4-4

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

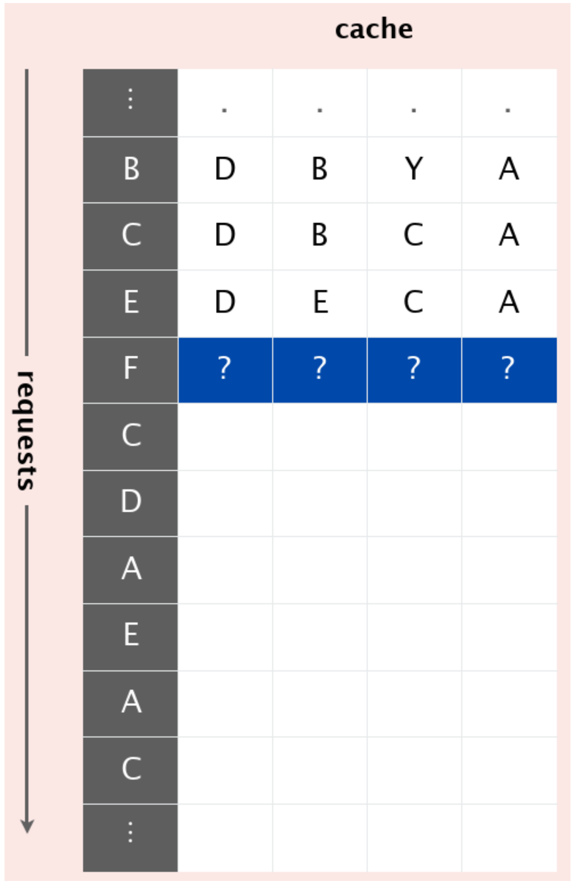
A

B

C

D

E



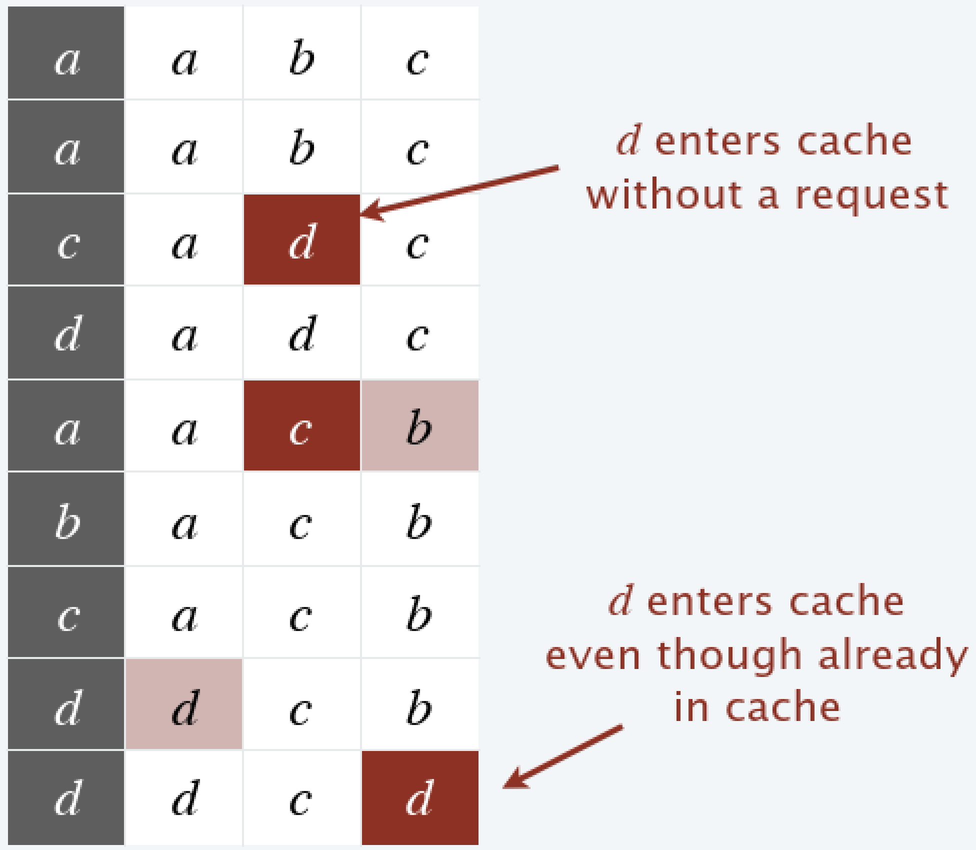
requests	cache			
⋮	.	.	.	.
B	D	B	Y	A
C	D	B	C	A
E	D	E	C	A
F	?	?	?	?
C				
D				
A				
E				
A				
C				
⋮				

4.3 Optimal Caching

Def. A reduced schedule is a schedule that only inserts an item into the cache in a step in which that item is requested.

Intuition. Can transform an unreduced schedule into a reduced one with no more cache misses.

4.3 Optimal Caching

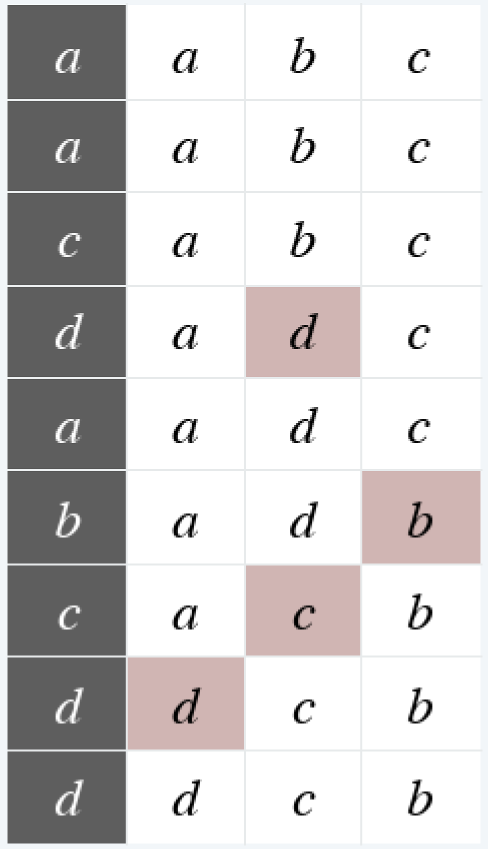


a	*a*	*b*	*c*
a	*a*	*b*	*c*
c	*a*	*d*	*c*
d	*a*	*d*	*c*
a	*a*	*c*	*b*
b	*a*	*c*	*b*
c	*a*	*c*	*b*
d	*d*	*c*	*b*
d	*d*	*c*	*d*

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

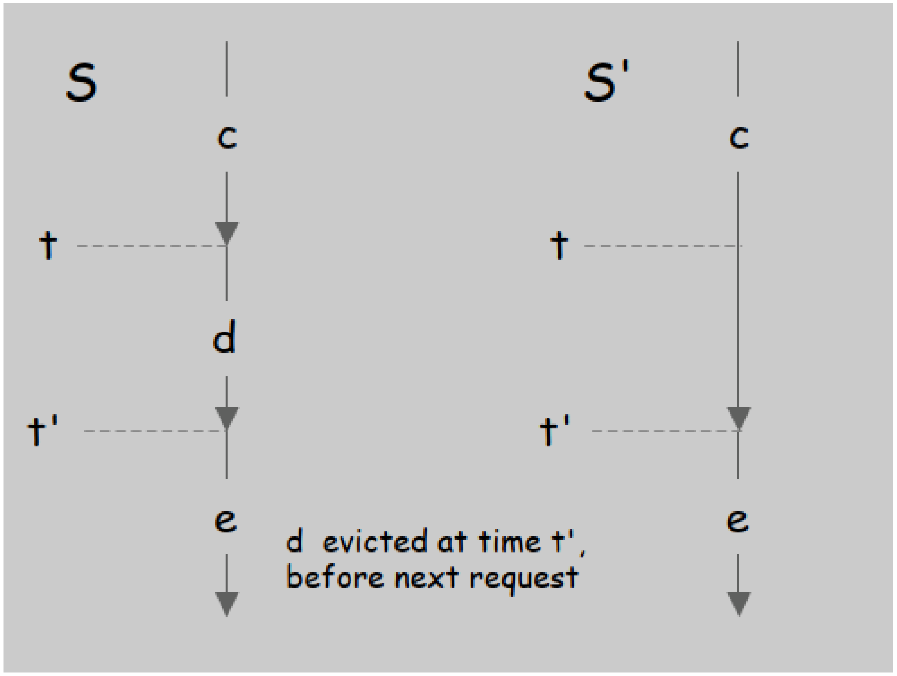
4.3 Optimal Caching

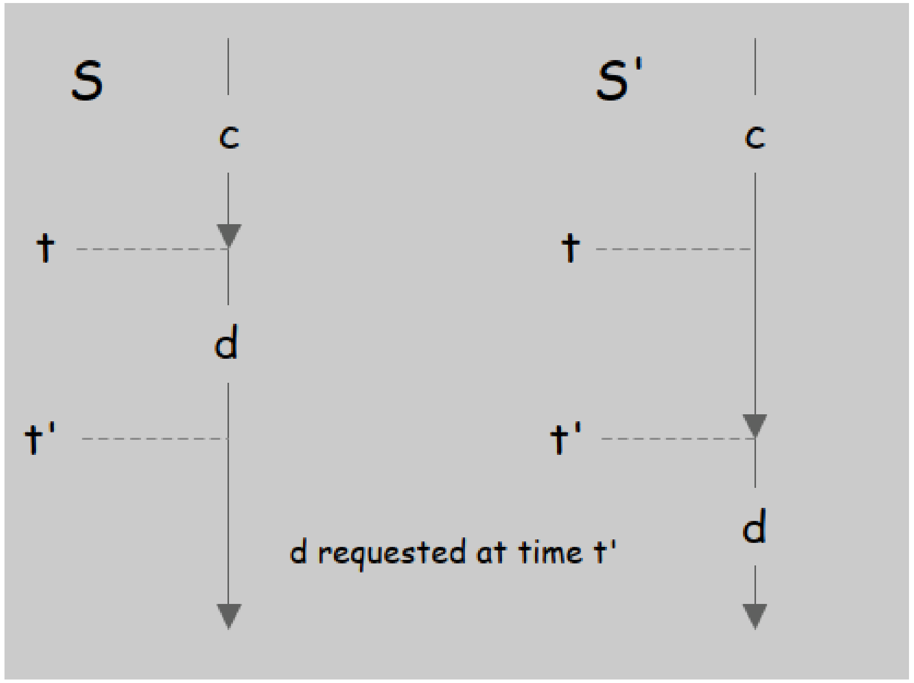
Claim. Given any unreduced schedule S, can transform it into a reduced schedule S' with no more evictions.

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 1: d evicted at time t', before next request for d.
- Case 2: d requested at time t' before d is evicted.

4.3 Optimal Caching





4.3 Optimal Caching

Theorem. FF is optimal eviction algorithm.

Pf. Follows directly from the following invariant.

4.3 Optimal Caching

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.

4.3 Optimal Caching

Pf. [continued]

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.

4.3 Optimal Caching

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)

4.3 Optimal Caching

Pf. [continued]

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'.

- Case 3a: $g = e$. Can't happen with FF since there must be a request for f before e.
- 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.

4.3 Optimal Caching

Pf. [continued]

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'.

- 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

4.3 Optimal Caching

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.

4.3 Optimal Caching

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, $LRU(\sigma) \le k\, FF(\sigma) + k$.
- Randomized marking is $O(\log k)$-competitive.

Thanks for Listening

College of Computer Science

Nankai University

Tianjin, P.R.China