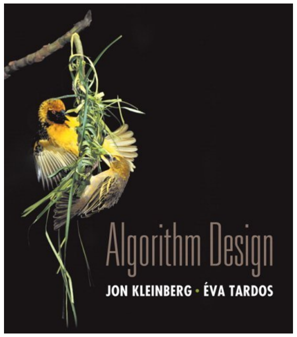
CSE 421

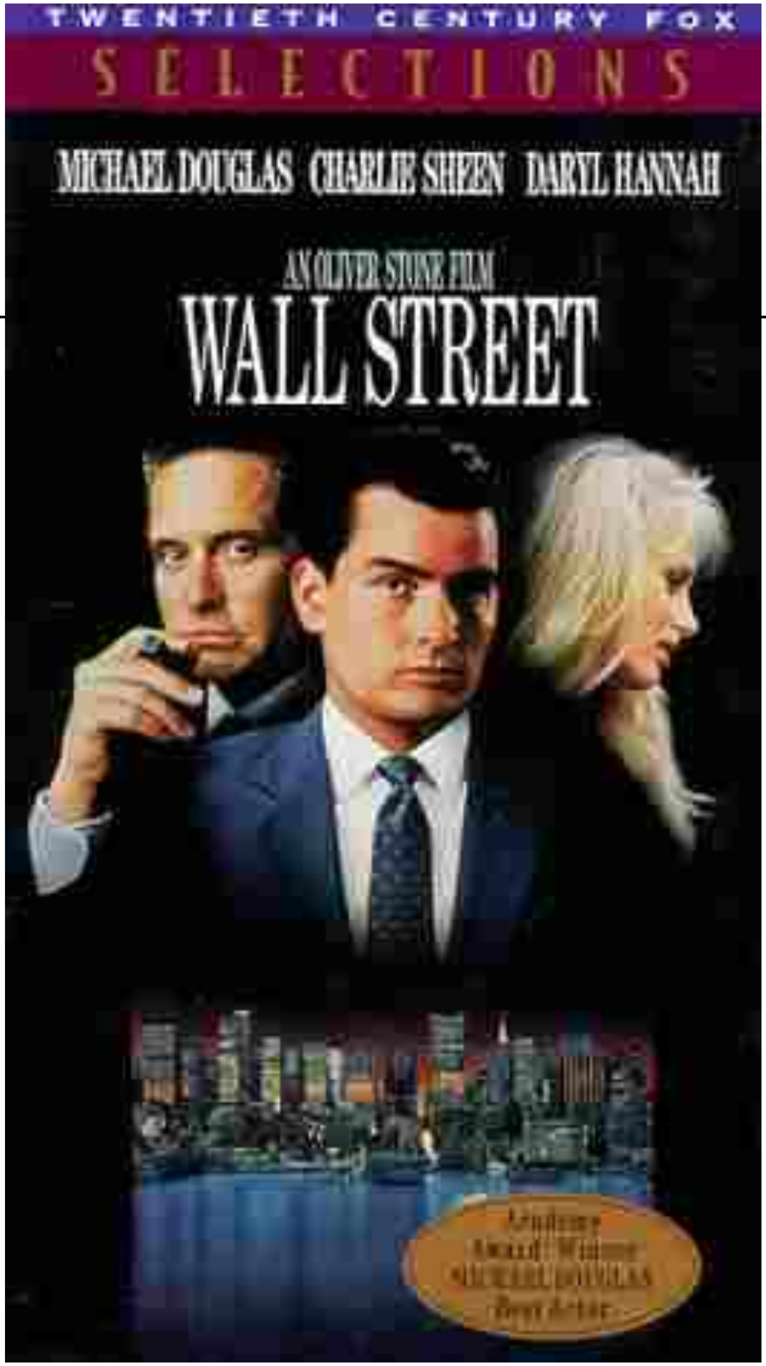
Chapter 4: Greedy Algorithms



Many Slides by Kevin Wayne.

Greed is good. Greed is right. Greed works. Greed clarifies, cuts through, and captures the essence of the evolutionary spirit.

- *Gordon Gecko (Michael Douglas)*



Intro: Coin Changing

Coin Changing

Goal. Given currency denominations: 1, 5, 10, 25, 100, give change to customer using *fewest* number of coins.

Ex: 34¢

Algorithm is "Greedy": One large coin better than two or more smaller ones

Cashier's algorithm. At each step, give the *largest* coin valued ≤ the amount to be paid.

Ex: $2.89

Coin-Changing: Does Greedy Always Work?

Observation. Greedy is sub-optimal for US *postal* denominations: 1, 10, 21, 34, 70, 100, 350, 1225, 1500.

Counterexample. 140¢.
- Greedy: 100, 34, 1, 1, 1, 1, 1, 1.
- Optimal: 70, 70.

Algorithm is "Greedy", but also short-sighted – attractive choice now may lead to dead ends later.

Correctness is key!

Outline & Goals

- “Greedy Algorithms”
  - what they are

- Pros
  - intuitive
  - often simple
  - often fast

- Cons
  - often incorrect!

- Proofs are crucial. 3 (of many) techniques:
  - stay ahead
  - structural
  - exchange arguments

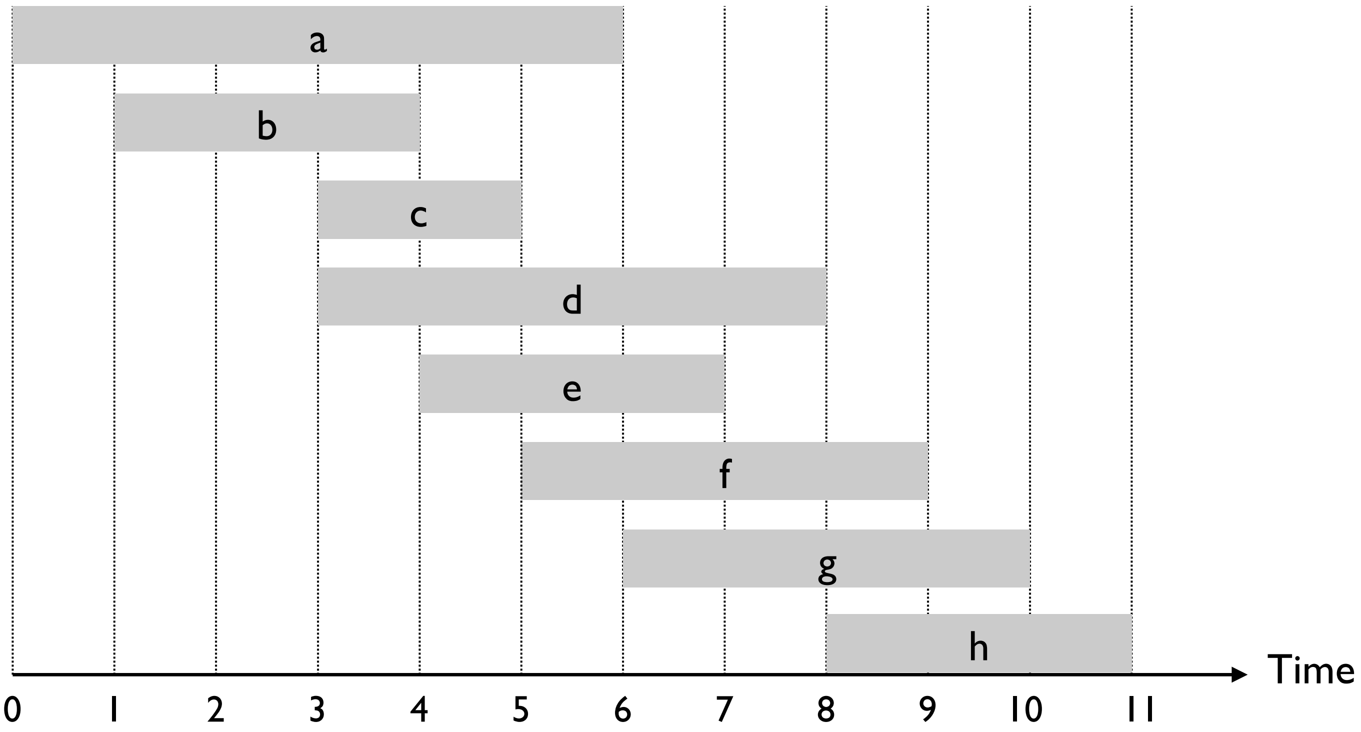
4.1 Interval Scheduling

Proof Technique 1: “greedy stays ahead”

Interval Scheduling

Interval scheduling.

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



Interval Scheduling: Greedy Algorithms

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

- What order?
- Does that give best answer?
- Why or why not?
- Does it help to be greedy about order?

Interval Scheduling: Greedy Algorithms

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

[Earliest start time] Order jobs by ascending start time s_j

[Earliest finish time] Order jobs by ascending finish time f_j

[Shortest interval] Order jobs by ascending interval length $f_j - s_j$

[Longest Interval] Reverse of the above

[Fewest conflicts] For each job j, let c_j be the count the number of jobs in conflict with j. Order jobs by ascending c_j

Can You Find Counterexamples?

E.g., Longest Interval:

Others?:

Interval Scheduling: Greedy Algorithms

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

breaks earliest start time

breaks shortest interval

breaks fewest conflicts

Interval Scheduling: *Earliest Finish First* Greedy Algorithm

Greedy algorithm. Consider jobs in *increasing order of finish time*. Take each job provided it's compatible with the ones already taken.

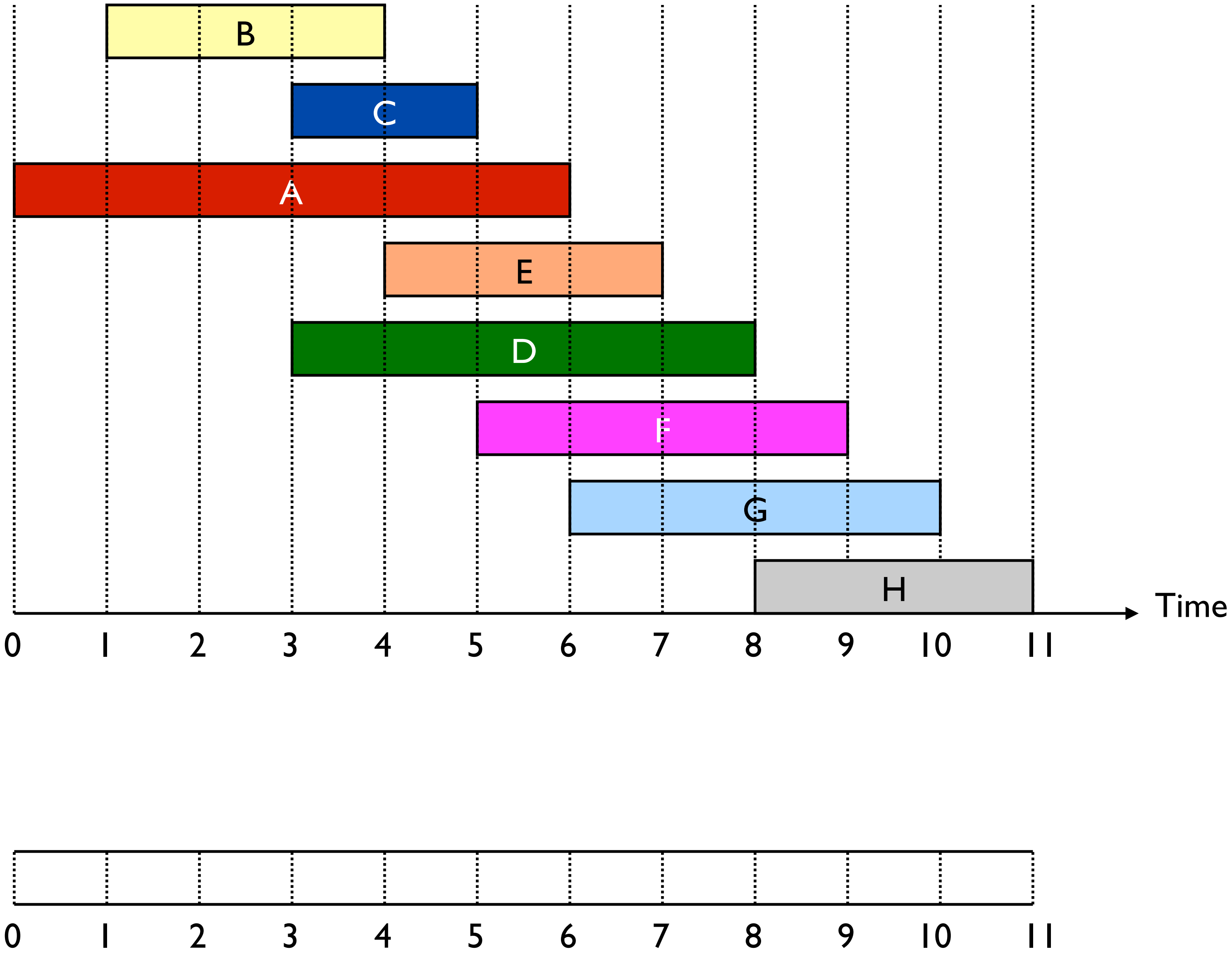
```
Sort jobs by finish times so that
$f_1 \le f_2 \le ... \le f_n$.

  jobs selected
A ← $\phi$
for j = 1 to n {
   if (job j compatible with A)
      A ← A $\cup$ {j}
}
return A
```

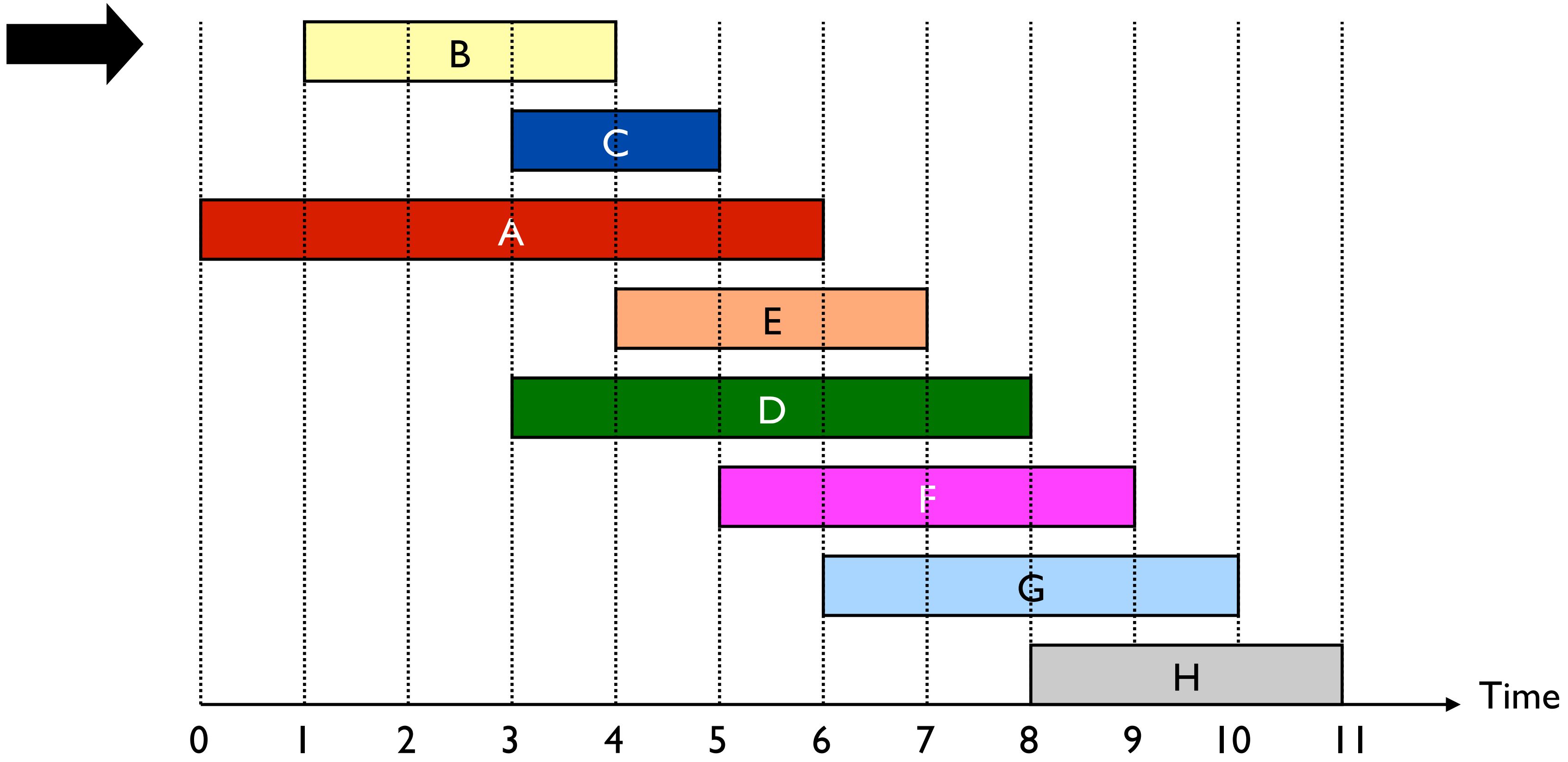
Implementation. O(n log n).

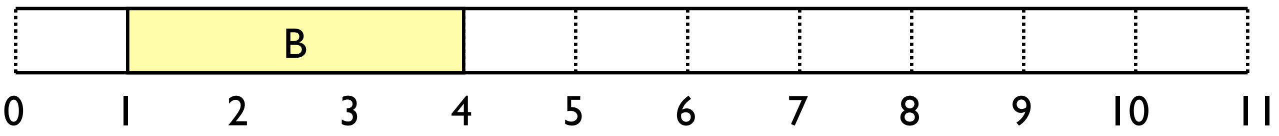
- Remember job j* that was added last to A.
- Job j is compatible with A if $s_j \ge f_{j*}$.

Interval Scheduling

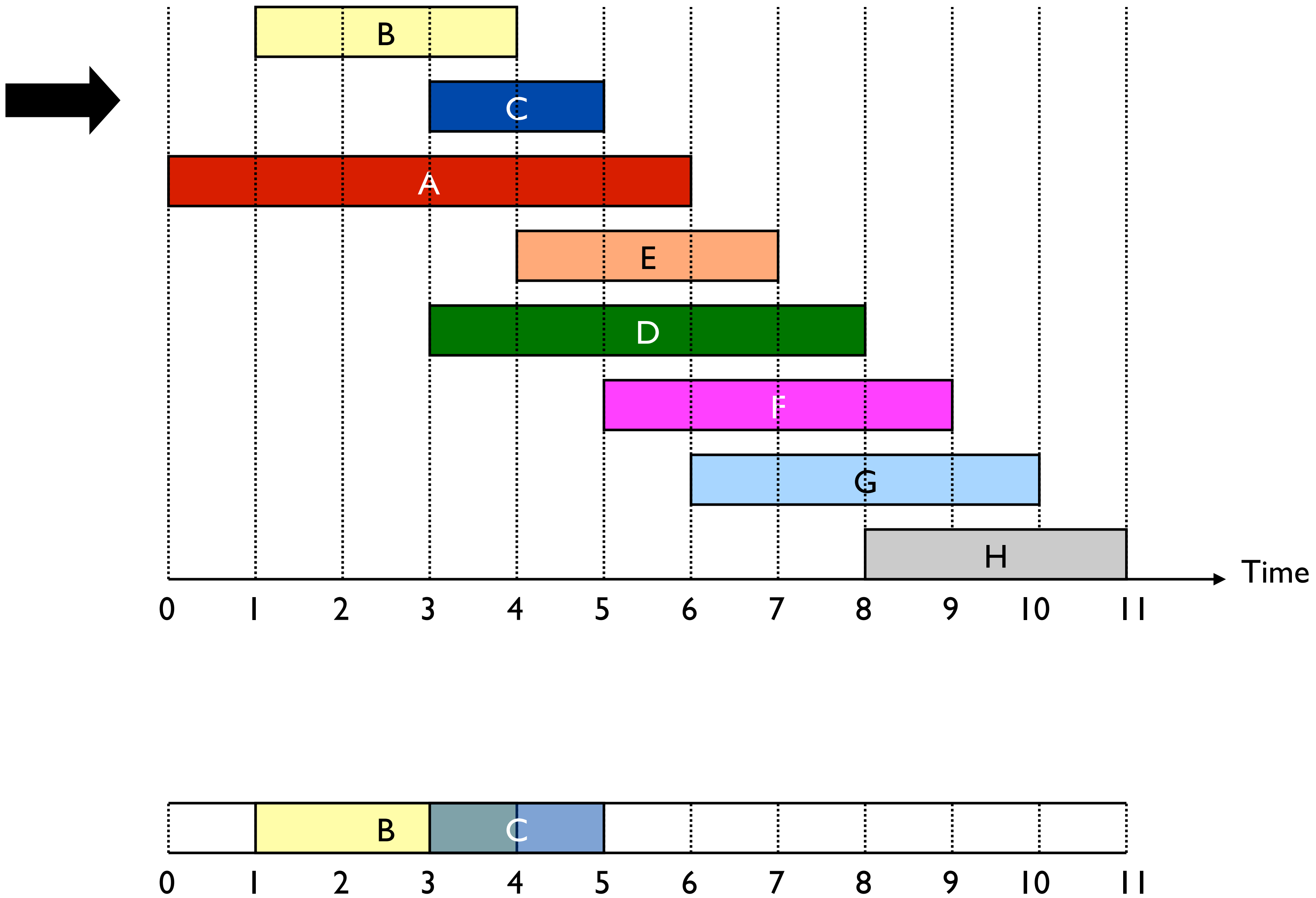


Interval Scheduling

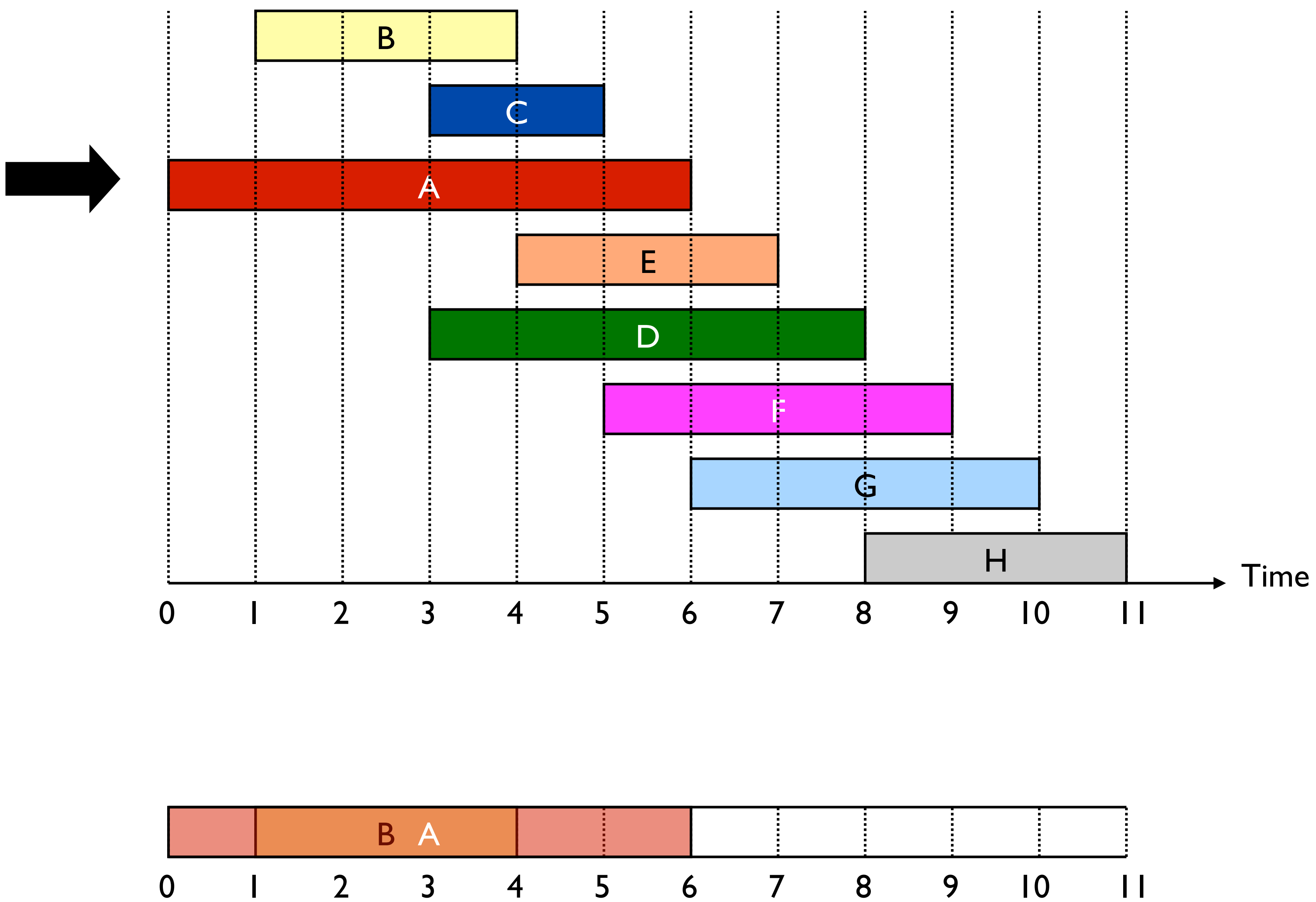




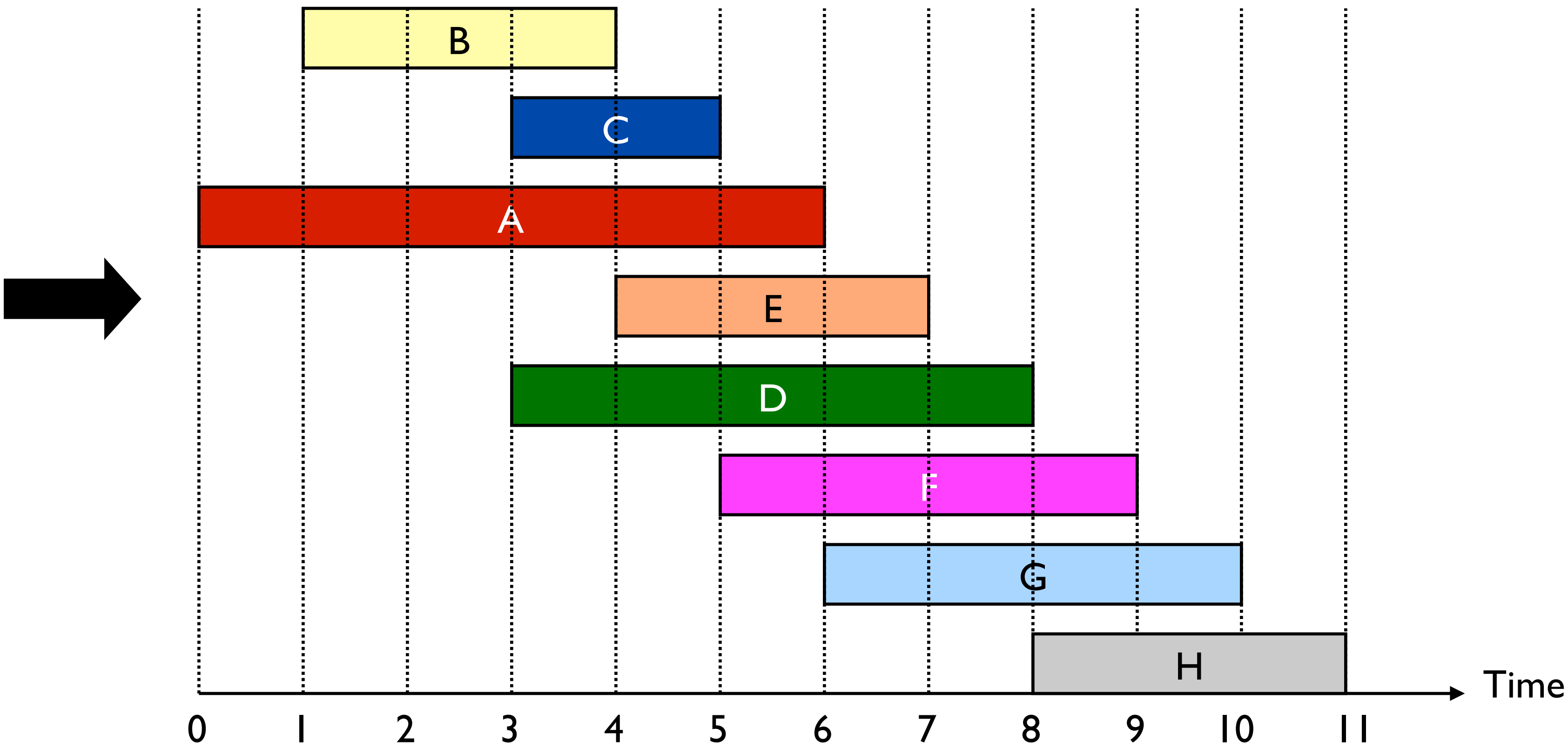
Interval Scheduling

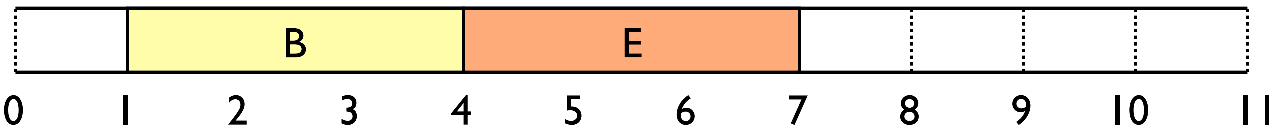


Interval Scheduling

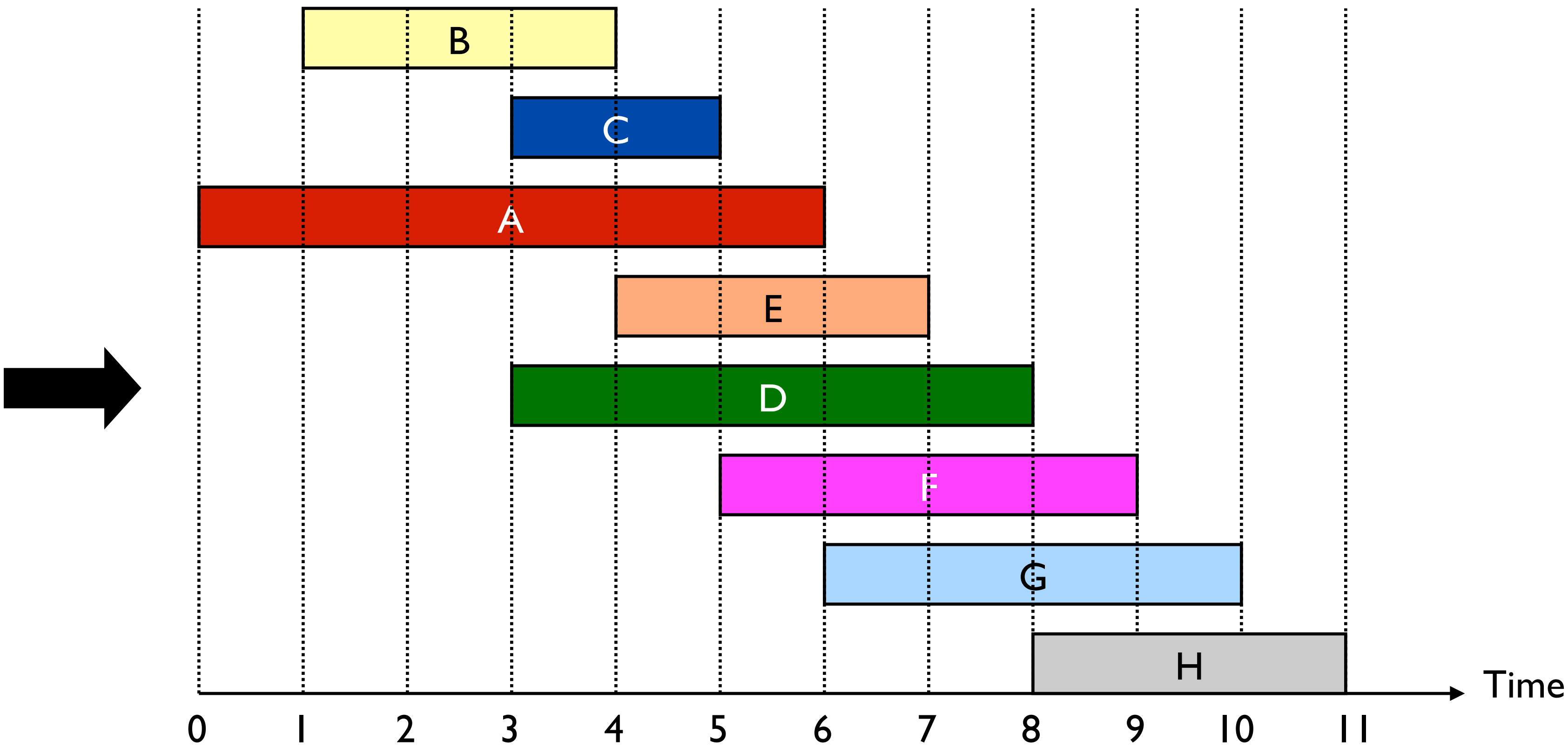


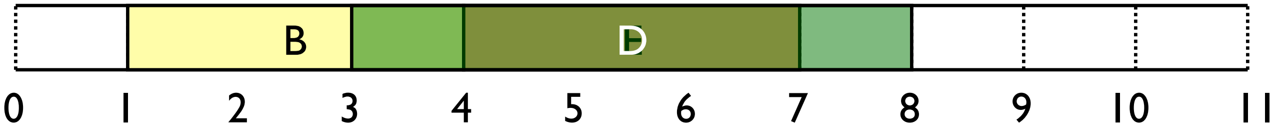
Interval Scheduling



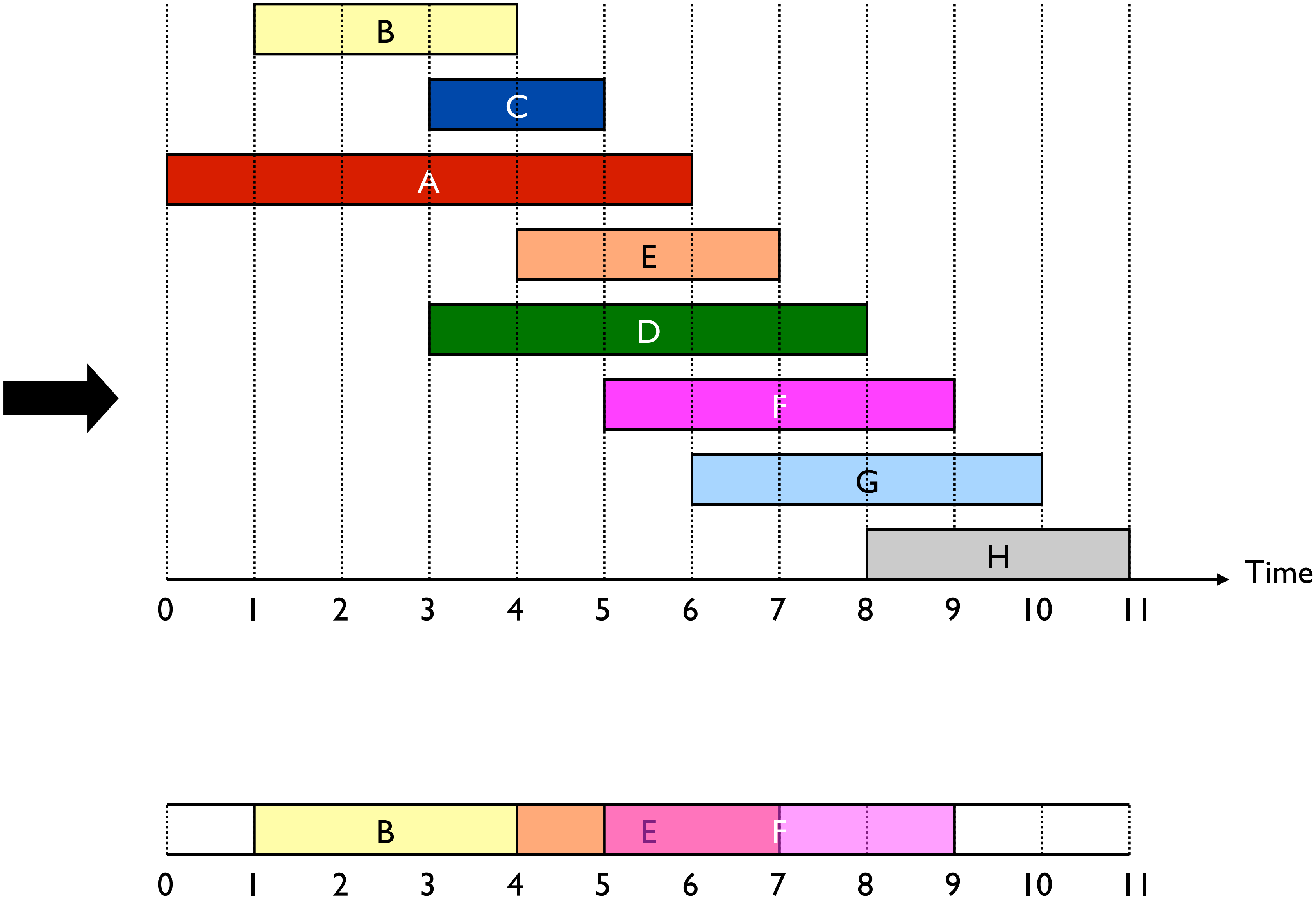


Interval Scheduling

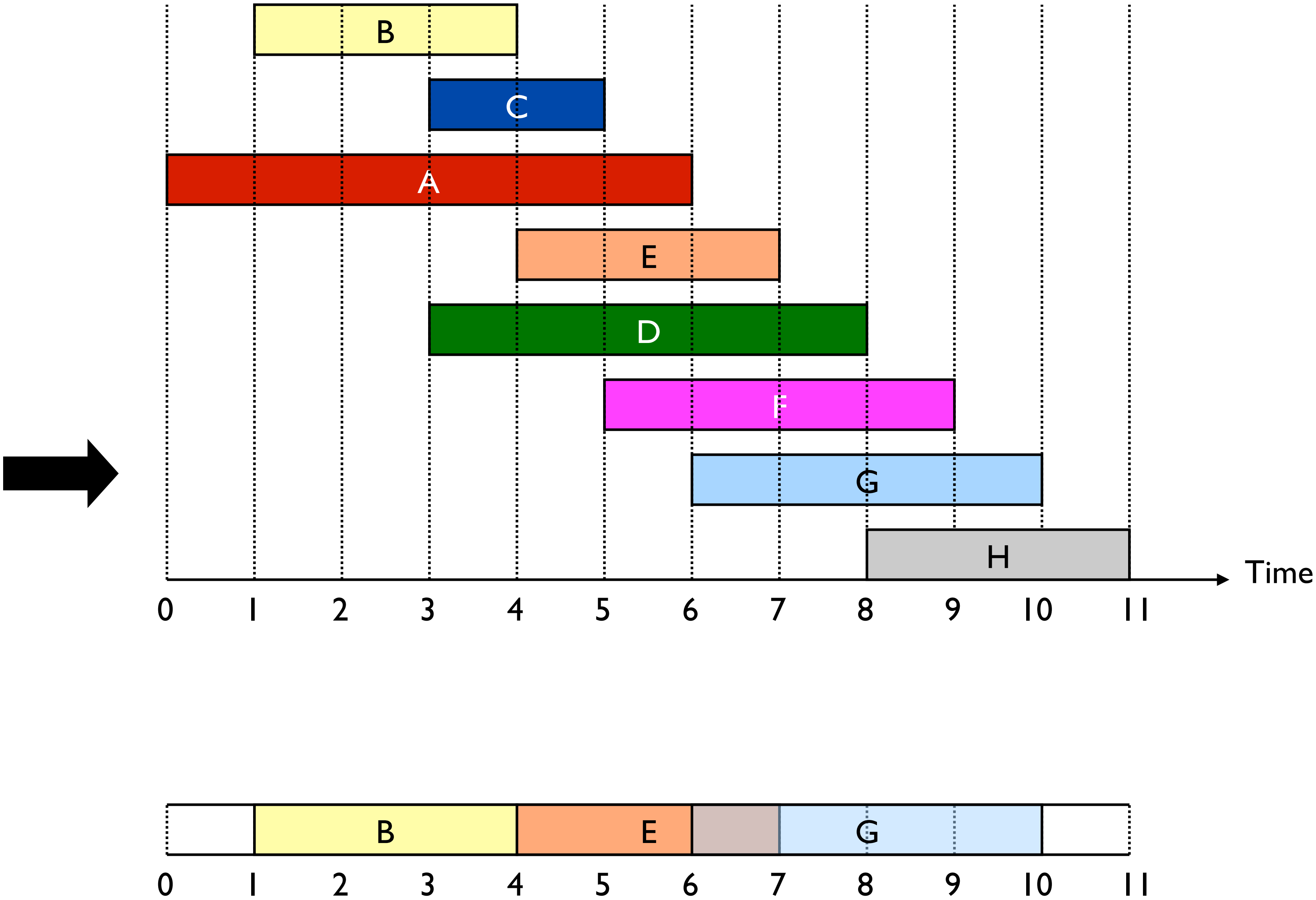




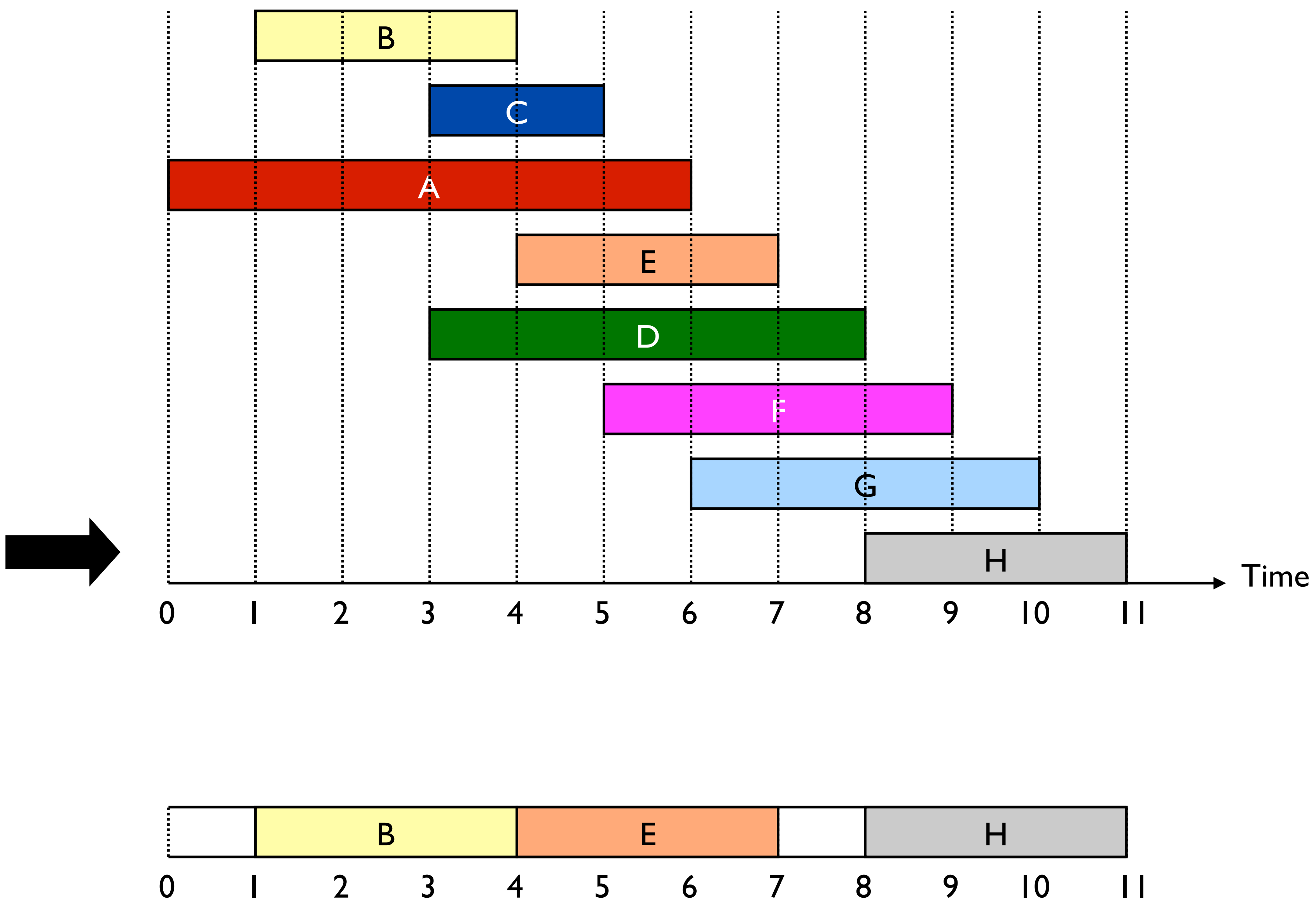
Interval Scheduling



Interval Scheduling



Interval Scheduling



Interval Scheduling: Correctness

Theorem. *Earliest Finish First* Greedy algorithm is optimal.

Pf. ("greedy stays ahead")

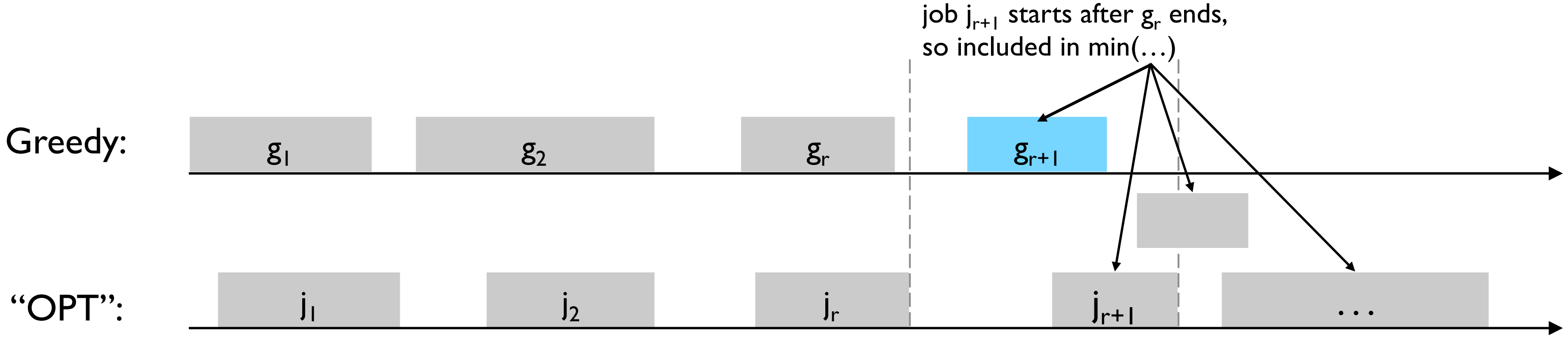
Let g_1, ... g_k be greedy's job picks, j_1, ... j_m those in some optimal solution

Show $f(g_r) \le f(j_r)$ by induction on r.

Basis: g_1 chosen to have min finish time, so $f(g_1) \le f(j_1)$

Ind: $f(g_r) \le f(j_r) \le s(j_{r+1})$, so j_{r+1} is among the candidates considered by greedy when it picked g_{r+1}, & it picks min finish, so $f(g_{r+1}) \le f(j_{r+1})$

Similarly, $k \ge m$, else j_{k+1} is among (nonempty) set of candidates for g_{k+1}



4.1 Interval Partitioning

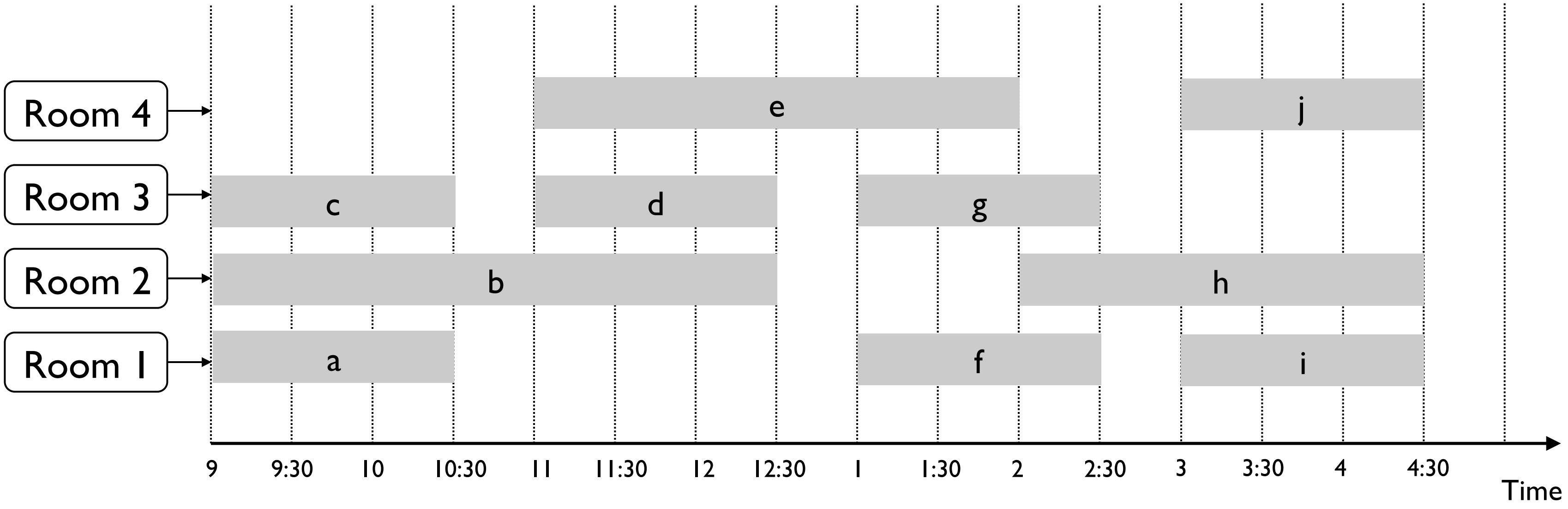
Proof Technique 2: “Structural”

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.

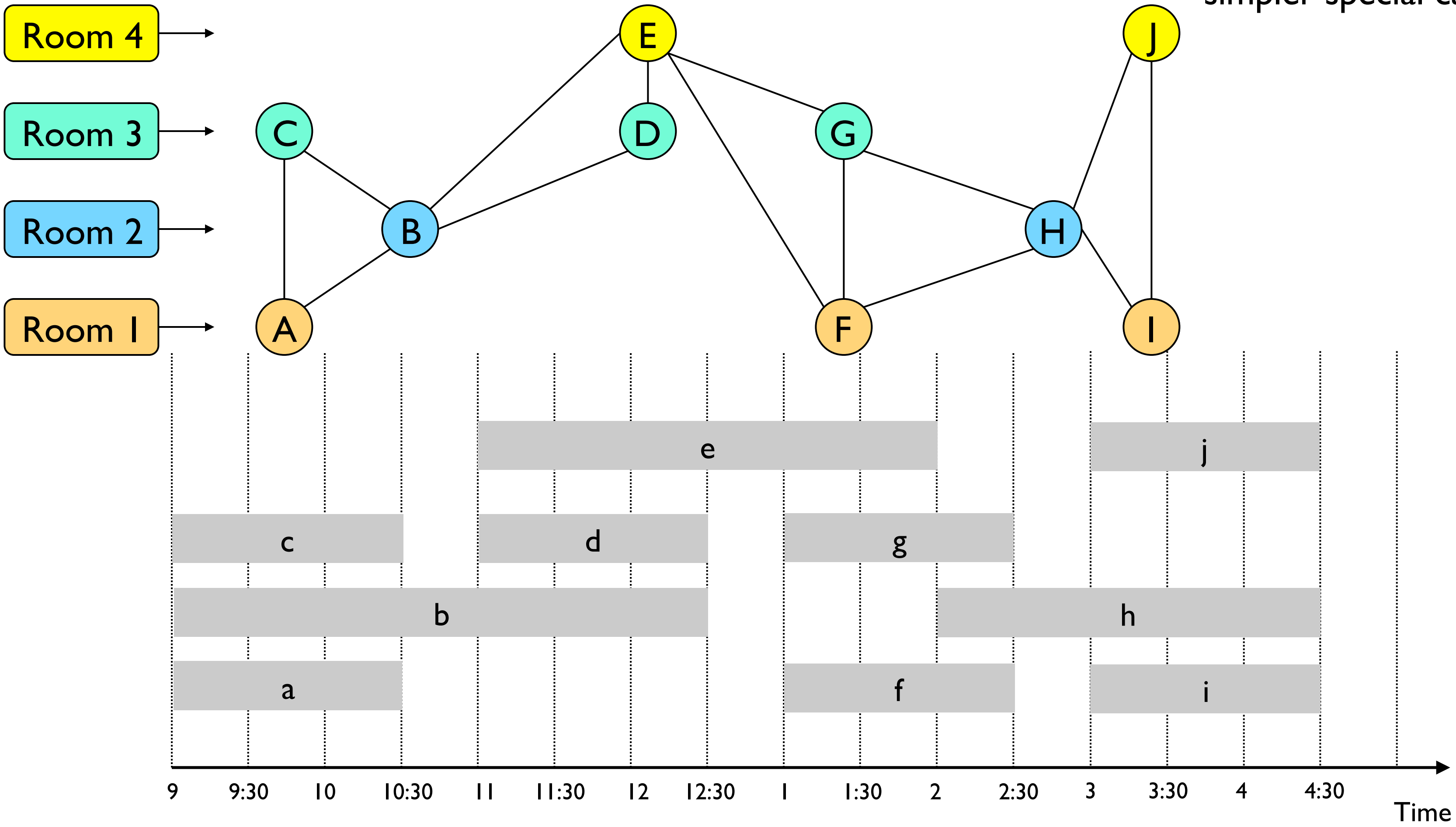
Ex: This schedule uses 4 classrooms to schedule 10 lectures.



Interval Partitioning as Interval Graph Coloring

Vertices = classes;
edges = conflicting class pairs;
different colors = different assigned rooms

Note: graph coloring is very hard in general, but graphs corresponding to interval intersections are a much simpler special case.

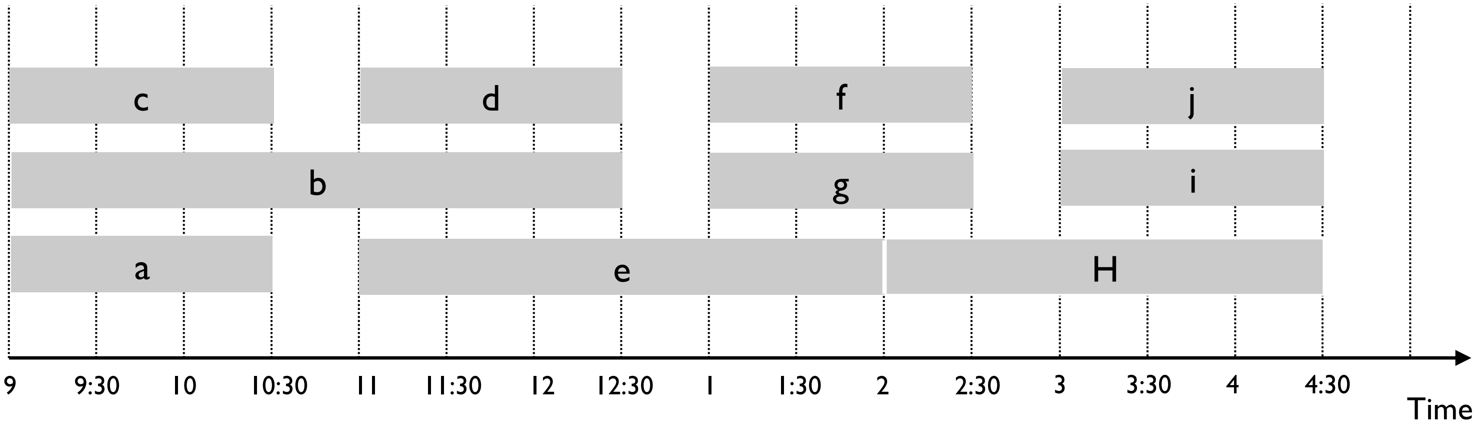


Interval Partitioning

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.

Ex: Same classes, but this schedule uses only 3 rooms.



Interval Partitioning: A "Structural" Lower Bound on Optimal Solution

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

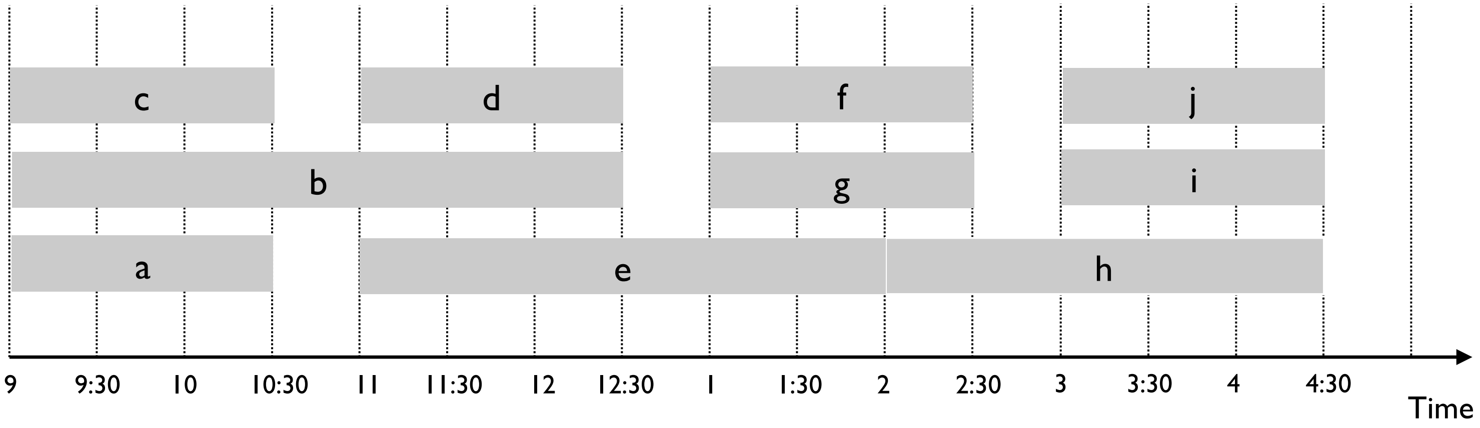
no collisions at ends

Key observation. Number of classrooms needed ≥ depth.

Ex: Depth of schedule below = 3 ⇒ schedule is optimal.

e.g., a, b, c all contain 9:30

Q. Does a schedule equal to depth of intervals always exist?



Interval Partitioning: Earliest Start First Greedy Algorithm

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

```
Sort intervals by start time so $s_1 \le s_2 \le \ldots \le s_n$.
d ← 0   ← number of allocated classrooms

for j = 1 to n {
   if (lect j is compatible with some room k, 1≤k≤d)
      schedule lecture j in classroom k
   else
      allocate a new classroom d + 1
      schedule lecture j in classroom d + 1
      d ← d + 1
}
```

Implementation? Run-time?
Exercises

Interval Partitioning: Greedy Analysis

Observation. Earliest Start First Greedy algorithm never schedules two incompatible lectures in the same classroom.

Theorem. Earliest Start First Greedy algorithm is optimal.
Pf (exploit structural property).

- Let d = number of rooms the greedy algorithm allocates.
- Classroom d is opened because we needed to schedule a job, say j, that is incompatible with all d-1 previously used classrooms.
- Since we sorted by start time, all these incompatibilities are caused by lectures that start no later than s_j.
- Thus, d lectures overlap at time $s_j + \varepsilon$, i.e. depth $\geq$ d
- "Key observation" on earlier slide $\Rightarrow$ all schedules use $\geq$ depth rooms, so d = depth and greedy is optimal

4.2 Scheduling to Minimize Lateness

Proof Technique 3: “Exchange” Arguments

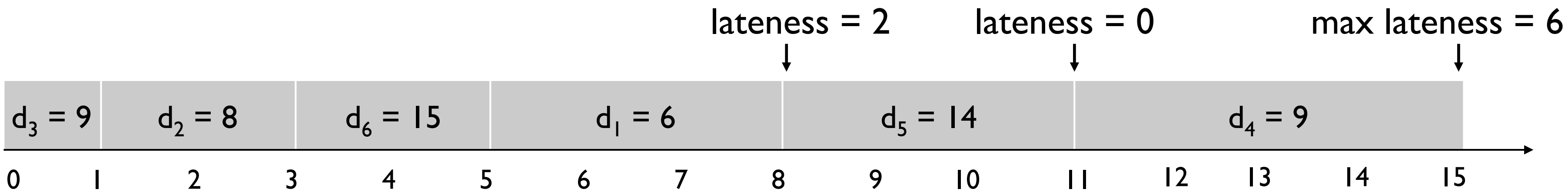
Scheduling to Minimize Lateness

Minimizing lateness problem.

- Single resource processes one job at a time.
- Job j requires t_j units of processing time & 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 to minimize max lateness $L = \max \ell_j$.

Ex:

j	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 in ascending order of processing time t_j.

	1	2
t_j	1	10
d_j	100	10

counterexample

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

	1	2
t_j	1	10
d_j	2	10

counterexample

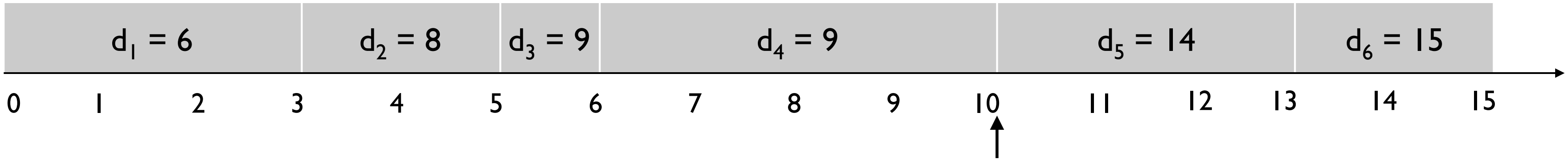
Minimizing Lateness: Greedy Algorithm

Greedy algorithm. Earliest deadline first.

```
Sort n jobs by deadline so that $d_1 \le d_2 \le \ldots \le d_n$

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 $[s_j, f_j]$
```

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



max lateness = 1

Proof Strategy

A *schedule* is an ordered list of jobs

Suppose S_1 is any schedule

Let G be the/a schedule produced by the greedy algorithm

To show: Lateness(S_1) ≥ Lateness(G)

Idea: find a series of simple changes that successively transform S_1 into other schedules that are more and more like G, each better than the last, until we reach G. I.e.

$$\text{Lateness}(S_1) \geq \text{Lateness}(S_2) \geq \text{Lateness}(S_3) \geq \ldots \geq \text{Lateness}(G)$$

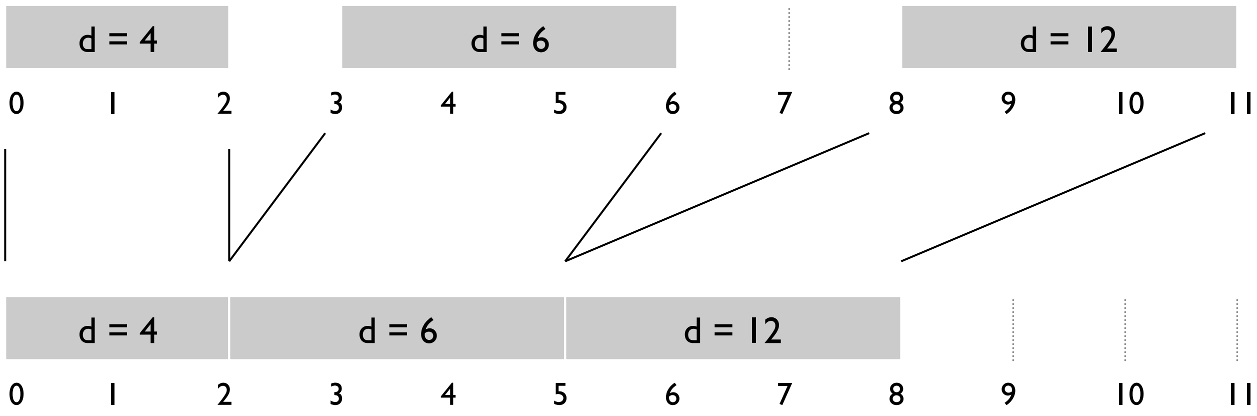
If it works for *any* starting S_1, it will work for an *optimal* S_1, so G is optimal

HOW?: *exchange* pairs of jobs

Minimizing Lateness: No Idle Time

Notes:

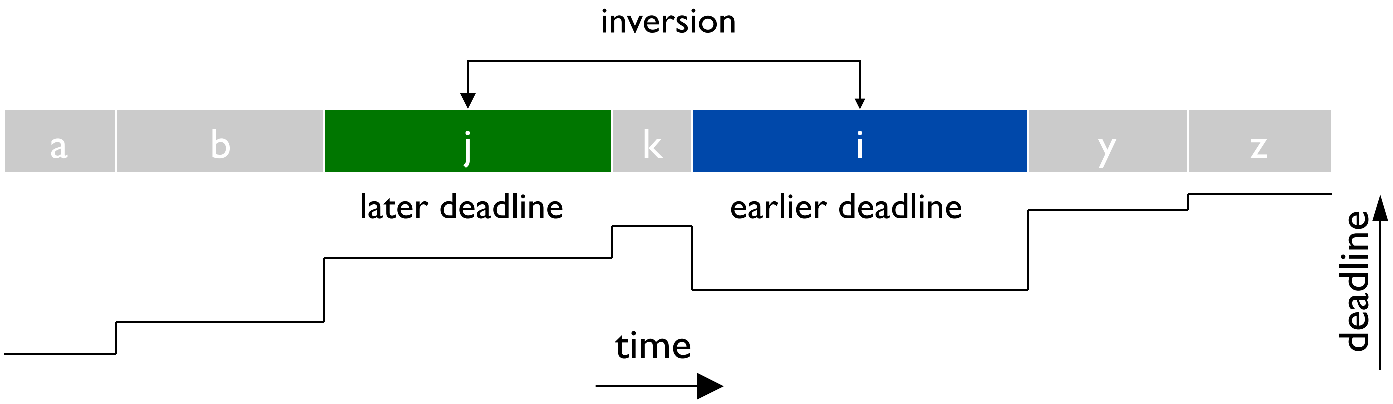
1. There is an optimal schedule with no idle time.



2. The greedy schedule has no idle time.

Minimizing Lateness: Inversions

Def. An *inversion* in schedule S is a pair of jobs i and j s.t.: deadline i < deadline j but j scheduled before i.

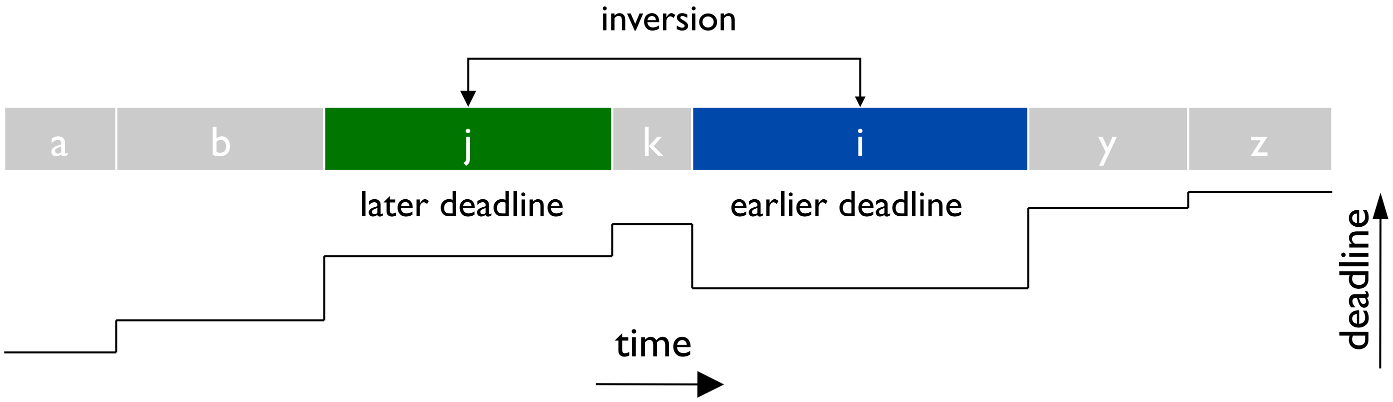


- Greedy schedule has no inversions.
- Claim: If a schedule has an inversion, it has an adjacent inversion, i.e., a pair of inverted jobs scheduled consecutively.

  (Pf: If j & i aren't consecutive, then look at the job k scheduled right after j. If $d_k < d_j$, then (j,k) is a consecutive inversion; if not, then (k,i) is an inversion, & nearer to each other - repeat.)

Minimizing Lateness: Inversions

Def. An *inversion* in schedule S is a pair of jobs i and j s.t.: deadline i < deadline j but j scheduled before i.

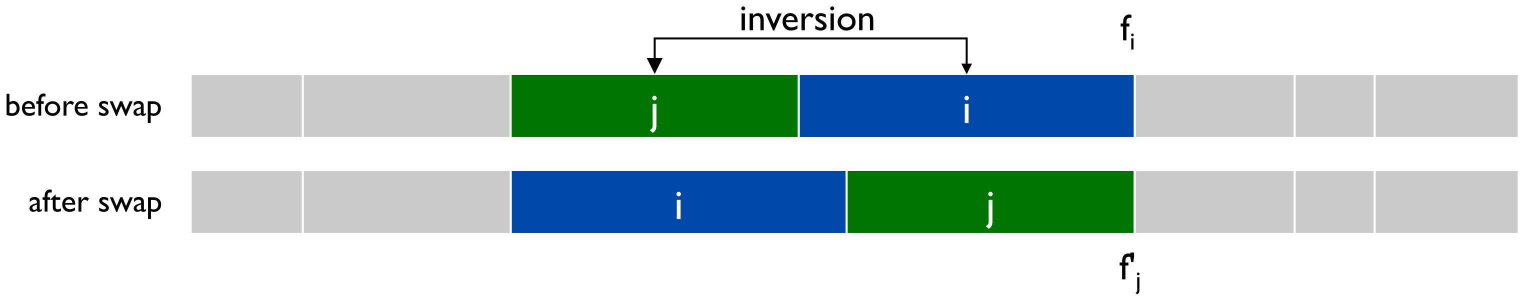


- Claim: Swapping an *adjacent* inversion reduces # inversions by 1 (exactly)

  Pf: Let i,j be an adjacent inversion. For any pair (p,q), inversion status of (p,q) is unchanged by i↔j swap unless {p, q} = {i, j}, and the i,j inversion is removed by that swap.

Minimizing Lateness: Inversions

Def. An *inversion* in schedule S is a pair of jobs i and j s.t.: deadline i < j but j scheduled before i.



(j had later deadline, so is less tardy than i was)

Claim. *Swapping* two adjacent, inverted jobs does not increase the max lateness.

Pf. Let ℓ / ℓ' be the lateness before / after swap, resp.

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

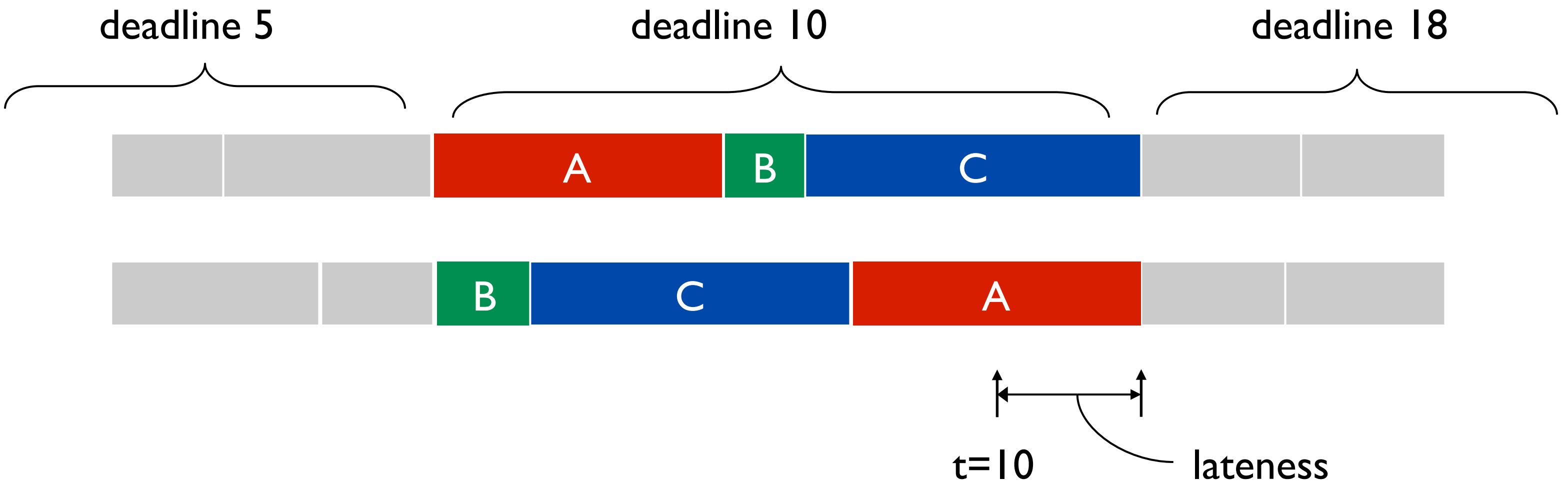
$$\begin{aligned} \ell'_j &= f'_j - d_j && \text{(definition)} \\ &= f_i - d_j && (j \text{ finishes at time } f_i) \\ &\leq f_i - d_i && (d_i \leq d_j) \\ &= \ell_i && \text{(definition)} \end{aligned}$$

only j moves later, but it's no later than i was, so max not increased

Minimizing Lateness: No Inversions

Claim. All idle-free, inversion-free schedules S have the same max lateness.

Pf. If S has no inversions, then deadlines of scheduled jobs are monotonically nondecreasing (i.e., increase or stay the same) as we walk through the schedule from left to right. Two such schedules can differ only in the order of jobs with the same deadlines. Within a group of jobs with the same deadline, the max lateness is the lateness of the last job in the group - order within the group doesn't matter.



Minimizing Lateness: Correctness of Greedy Algorithm

Theorem. Greedy schedule S is optimal

Pf. Let S* be an optimal schedule with the fewest number of inversions among all optimal schedules

- Can assume S* has no idle time.
- 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*
- So, S* has no inversions. Hence Lateness(S) = Lateness(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. (Part of the cleverness is deciding what’s “good.”)

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. (Cleverness here is usually in finding a useful structural characteristic.)

Exchange argument. Gradually transform any solution to the one found by the greedy algorithm without hurting its quality. (Cleverness usually in choosing which pair to swap.)

(In all 3 cases, proving these claims may require cleverness.)

4.3 Optimal Caching

^{1}cache

Pronunciation: 'kash
Function: *noun*
Etymology: French, from *cacher* to press, hide

a hiding place especially for concealing and preserving provisions or implements

^{2}cache

Function: *transitive verb*

to place, hide, or store in a cache

-Webster's Dictionary

Optimal Offline Caching

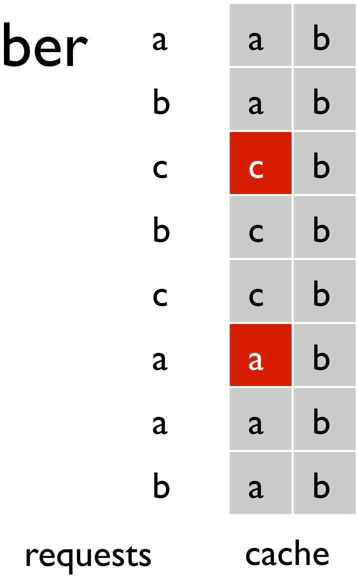
Caching.

- Cache with capacity to store k items.
- Sequence of m item requests $d_1, d_2, \ldots, d_m$.
- Cache hit: item 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.

Goal. Eviction schedule that minimizes number of cache misses.

Ex: k = 2, initial cache = ab,
requests: a, b, c, b, c, a, a, b.

Optimal eviction schedule: 2 cache misses.

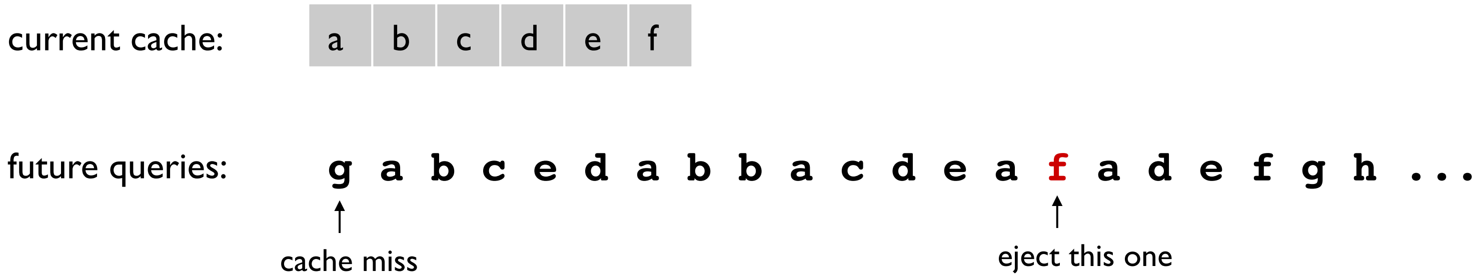


a	a	b
b	a	b
c	c	b
b	c	b
c	c	b
a	a	b
a	a	b
b	a	b

requests cache

Optimal Offline Caching: Farthest-In-Future

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



current cache: a b c d e f

future queries: `g a b c e d a b b a c d e a f a d e f g h ...`

cache miss

eject this one

Theorem. [Bellady, 1960s] FF is optimal eviction schedule.
Pf. Algorithm and theorem are intuitive; proof is subtle.

Motivation: "Online" problem is typically what's needed in practice - decide what to evict *without* seeing the future. How to evaluate such an alg? Fewer misses is obviously better, but how few? FF is a useful benchmark - best online alg is unknown, but it's no better than FF, so online performance close to FF's is the best you can hope for.

4.4 Shortest Paths in a Graph

You've seen this in prerequisite courses, so this section and next two on min spanning tree are review. I won't lecture on them, but you should review the material. Both, but especially shortest paths, are common problems, having many applications. (And frequent fodder for job interview questions…)

Shortest Path Problem

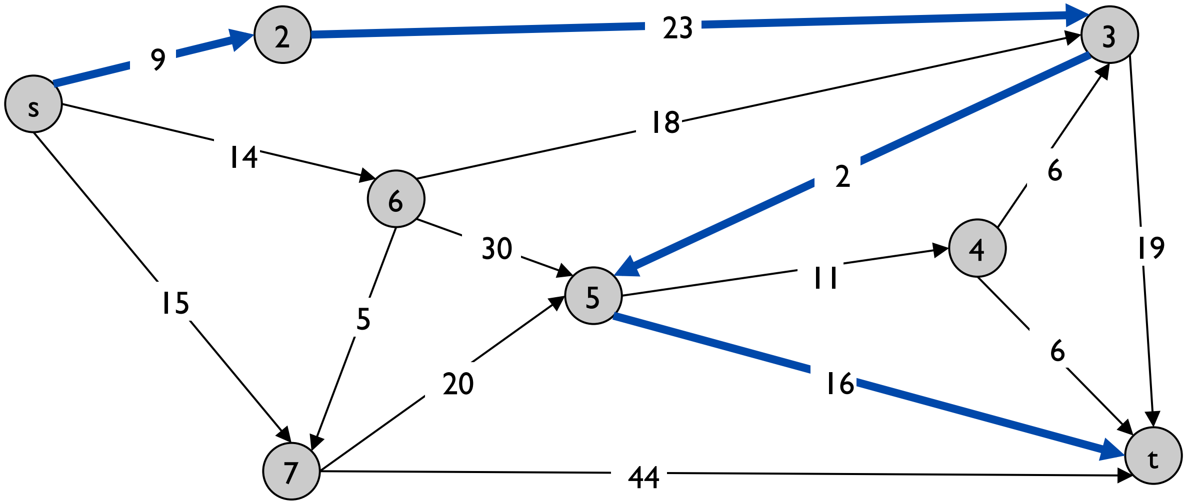
Shortest path network.

- Directed graph G = (V, E).
- Source s, destination t.
- Length ℓ_e = length of edge e.

Shortest path problem: find shortest directed path from s to t.

↑

cost of path = sum of edge costs in path



Cost of path s-2-3-5-t
= 9 + 23 + 2 + 16
= 48.

Dijkstra's Algorithm

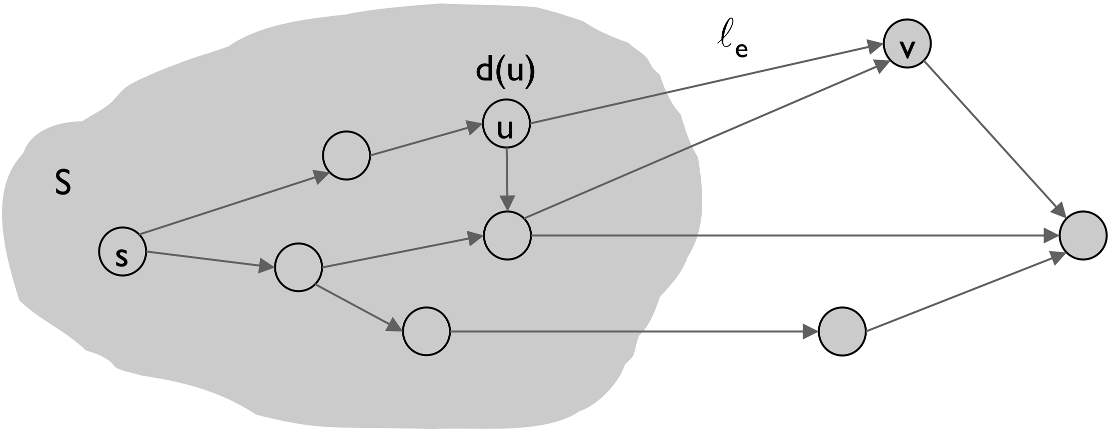
Dijkstra's algorithm.

- Maintain a set of explored nodes S for which we have determined the shortest path distance d(u) from s to u.
- Initialize S = { s }, d(s) = 0.
- Repeatedly choose unexplored node v which minimizes

$$\pi(v) = \min_{e = (u,v) : u \in S} d(u) + \ell_e ,$$

shortest path to some u in explored part, followed by a single edge (u, v)

add v to S, and set d(v) = π(v).



Dijkstra's Algorithm

Dijkstra's algorithm.

- Maintain a set of explored nodes S for which we have determined the shortest path distance d(u) from s to u.
- Initialize S = { s }, d(s) = 0.
- Repeatedly choose unexplored node v which minimizes

$$\pi(v) = \min_{e = (u,v) : u \in S} d(u) + \ell_e ,$$

shortest path to some u in explored part, followed by a single edge (u, v)

add v to S, and set d(v) = π(v).