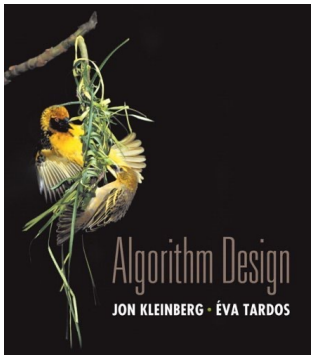
CSE 417

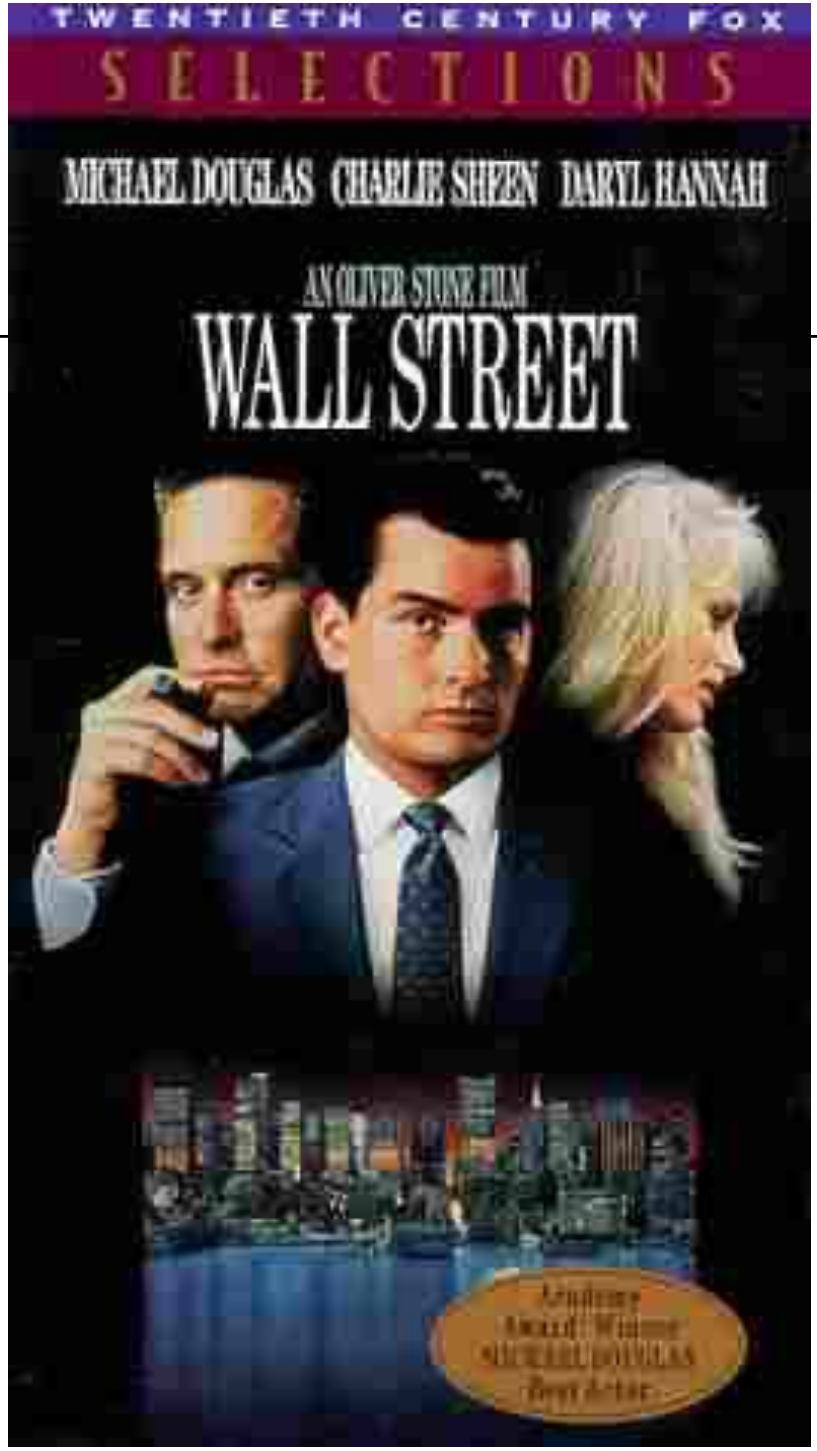
Chapter 4: Greedy Algorithms



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?

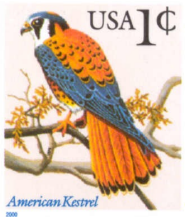
Observation. Greedy is sub-optimal for US *postal* denominations: 1, 10, 21, 34, 70, 100, 350, 1225, 1500.

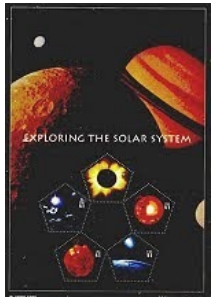
Counterexample. 140¢.

- 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

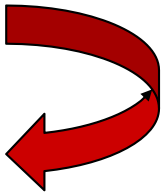
Proofs by Induction

"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



Proofs by Induction

Given some statement P(k), like

"For any $x \geq -1$, $(1+x)^k \geq 1+kx$",

I want to prove that P(k) is true for all (integer) $k \geq 1$

One way:

(base case) Prove that P(1) is true

(induction step) Prove, for all $k \geq 1$, that "P(k) implies P(k+1)"

(Note: without both parts, you're toast…)

E.g.

(base case): P(1) is "$(1+x)^1 \geq 1+1*x$" which is obviously true since LHS = RHS

(induction step):

$$(1+x)^{k+1} = (1+x) * (1+x)^k \geq (1+x) * (1+kx) = 1+ (k+1)x+ kx^2 \geq 1 + (k+1)x$$

$(1+x)^k \geq (1+kx)$ By Induction Hypothesis

≥ 0

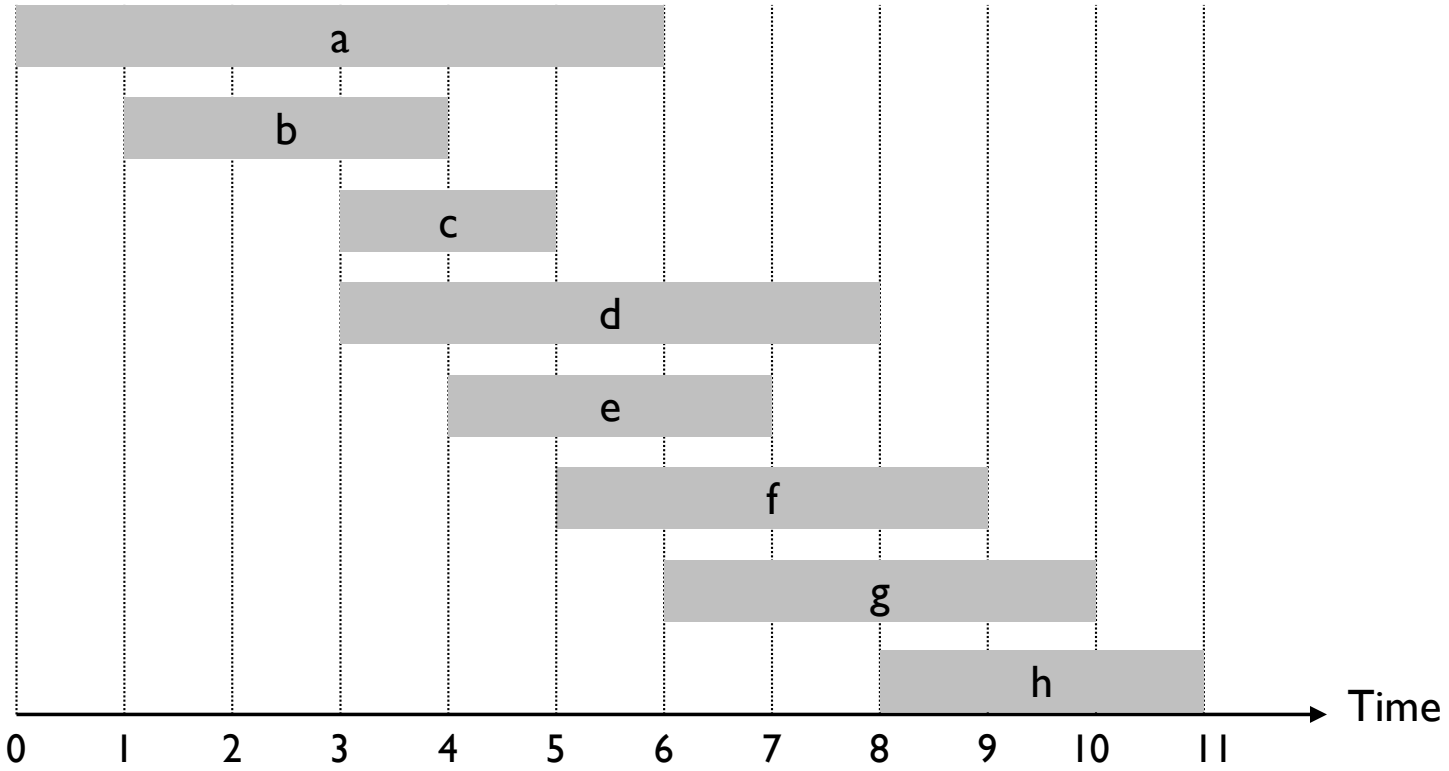
Thus $(1+x)^{k+1} \geq 1 + (k+1)x$, i.e. P(k) implies P(k+1). QED

4.1 Interval Scheduling

Proof Technique 1: "greedy stays ahead"

Interval scheduling.

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



Interval Scheduling: Greedy Algorithms

Greedy template. Consider jobs in some order. Take next 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

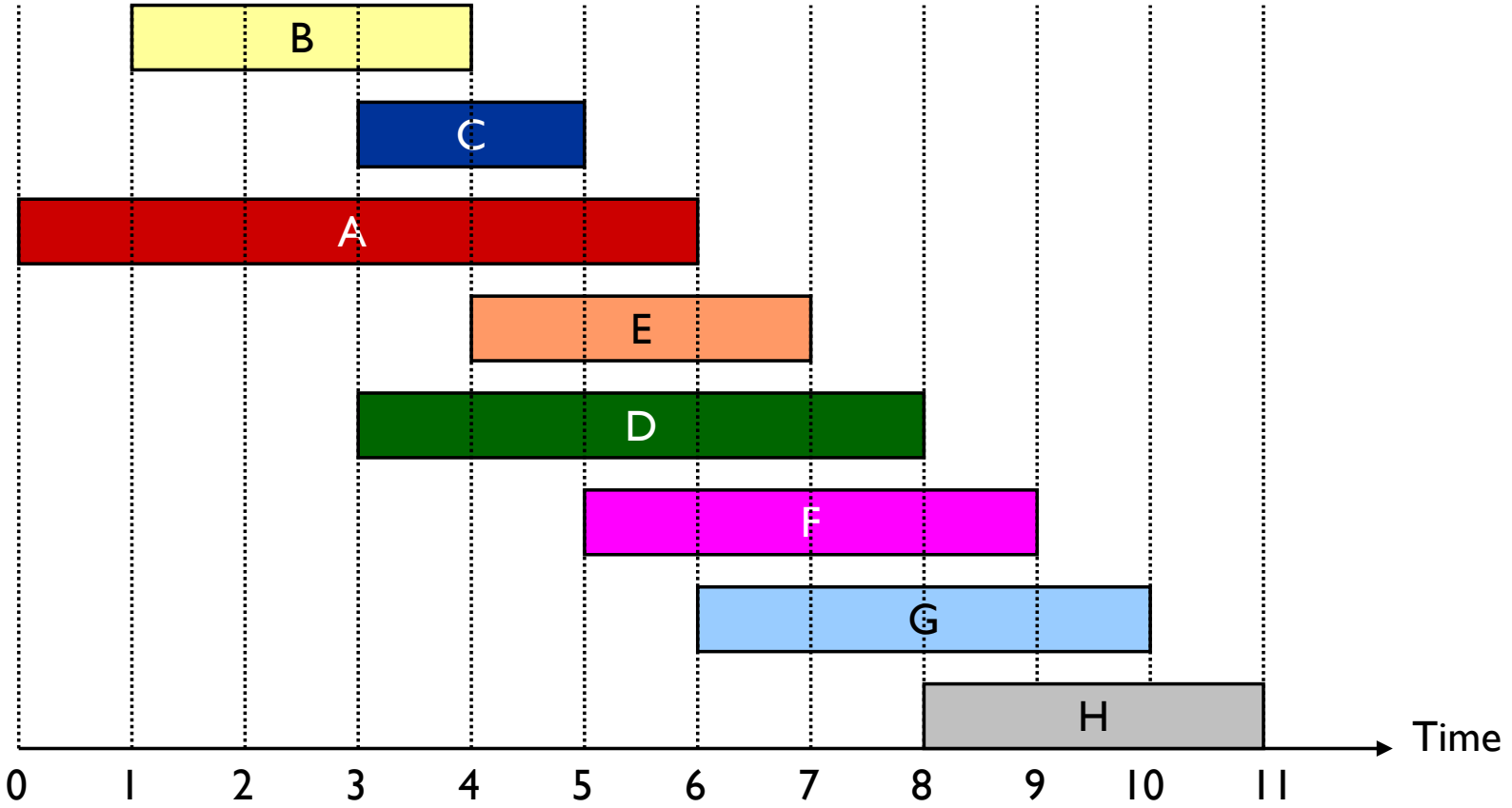
Greedy algorithm. Consider jobs in *increasing order of finish time*. Take each job provided it's compatible with the ones already taken.

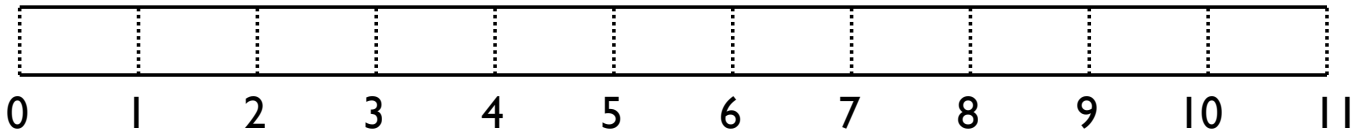
```
Sort jobs by finish times so that
f1 ≤ f2 ≤ ... ≤ fn.

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

Implementation. O(n log n) to sort + O(n) for the rest.

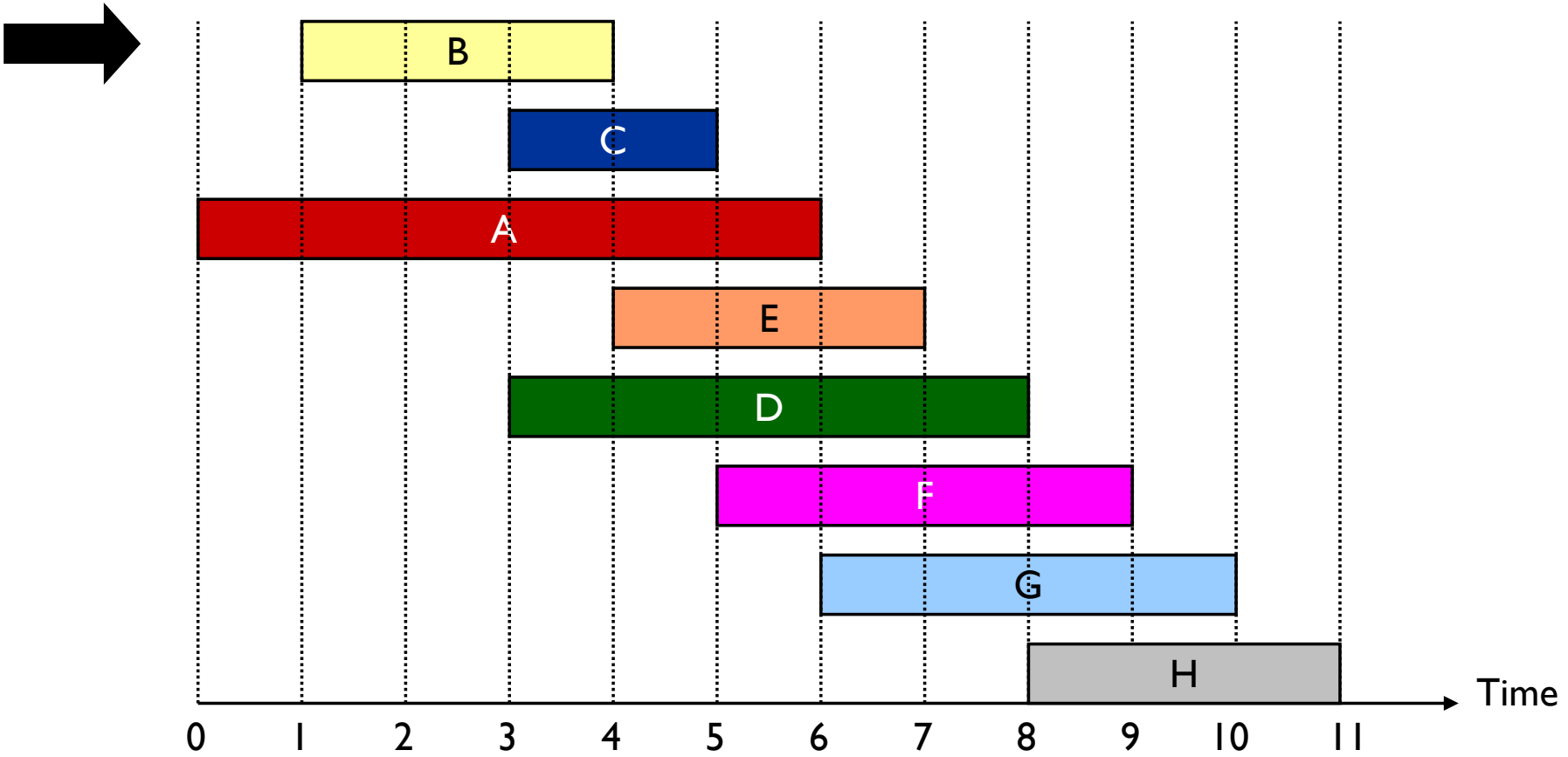
- Remember job j^* that was added last to A.
- Job j is compatible with A if $s_j \geq f_{j^*}$.

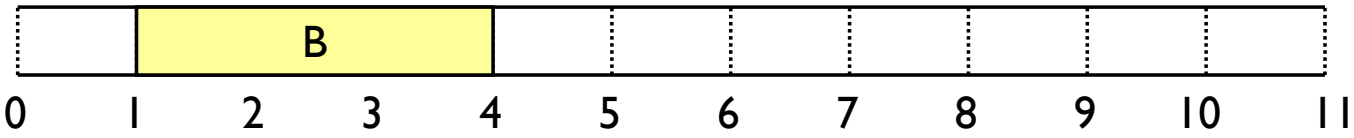
Interval Scheduling



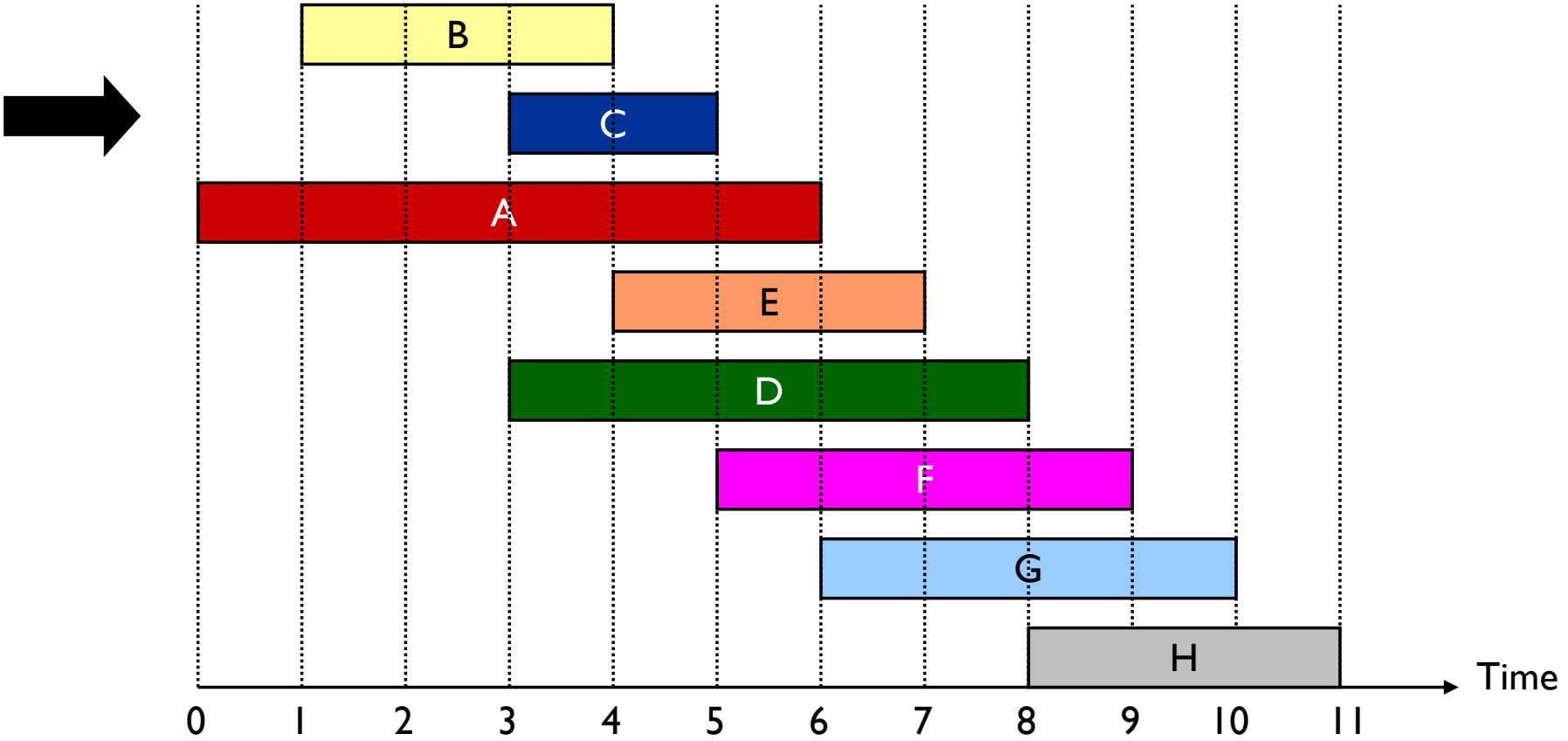


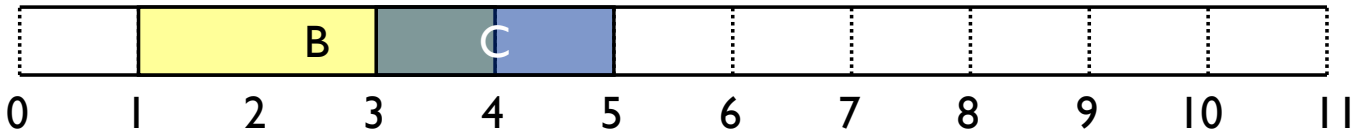
Interval Scheduling



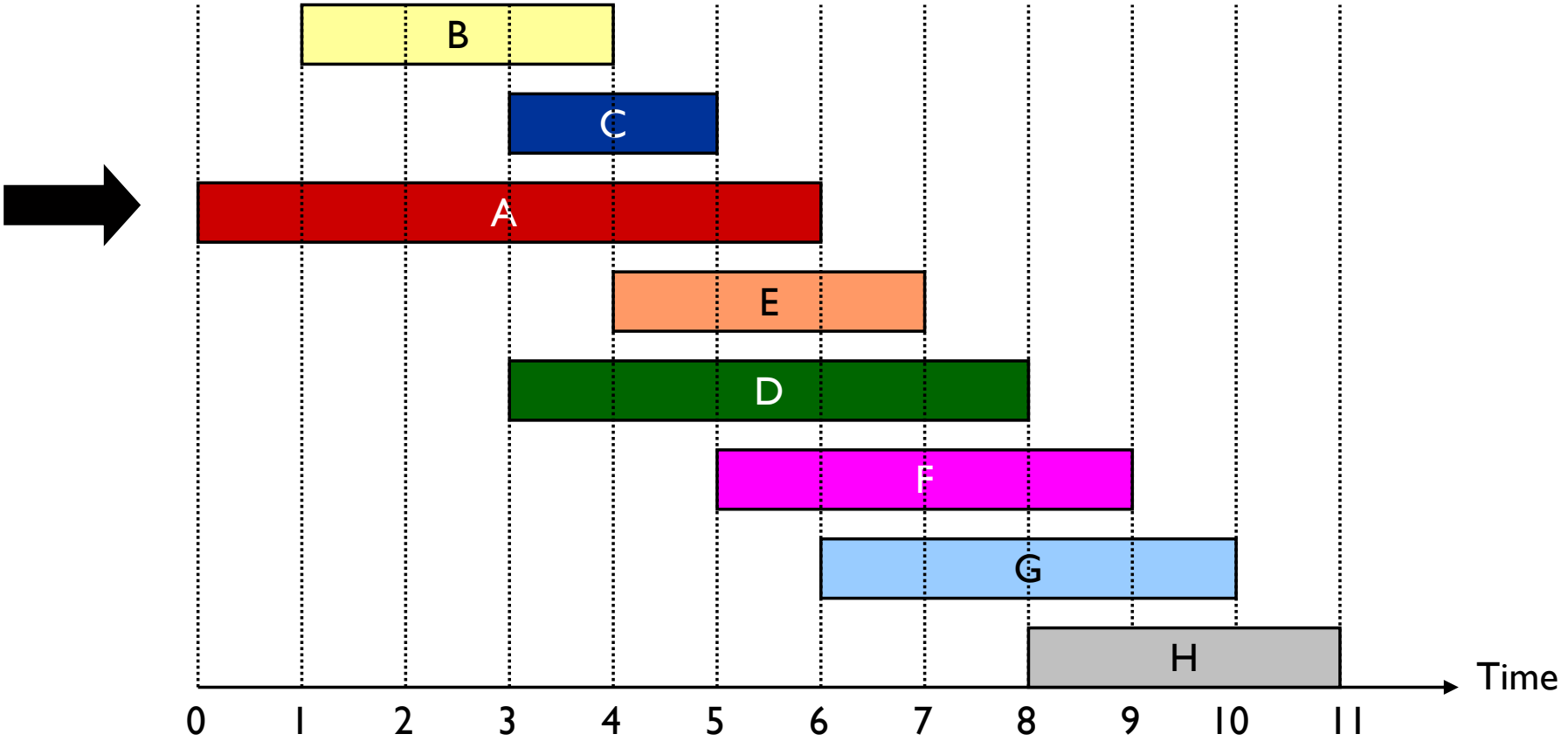


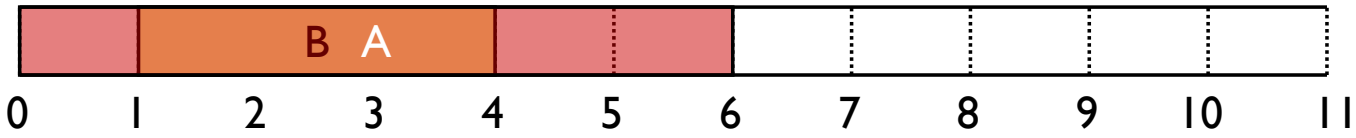
Interval Scheduling



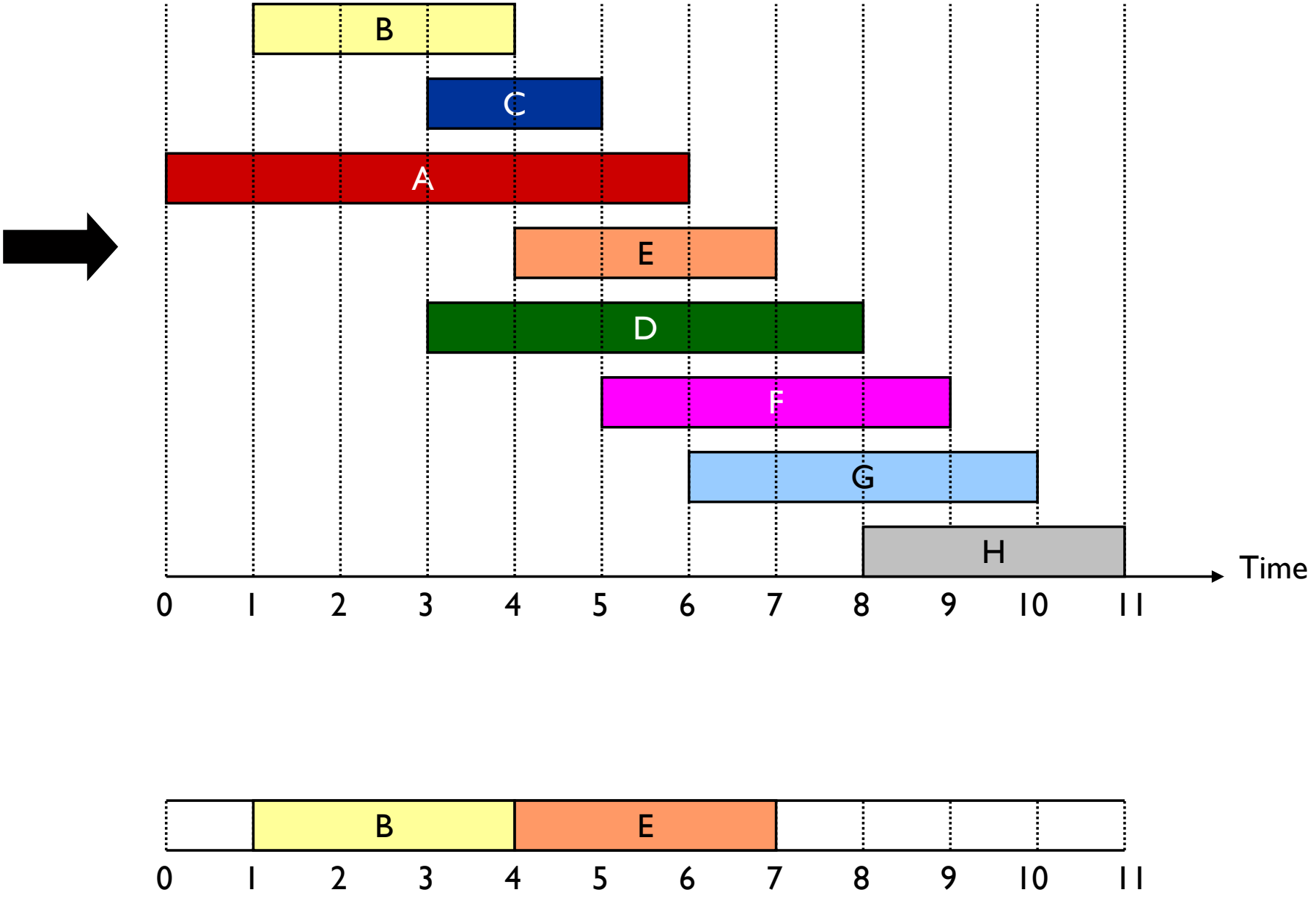


Interval Scheduling

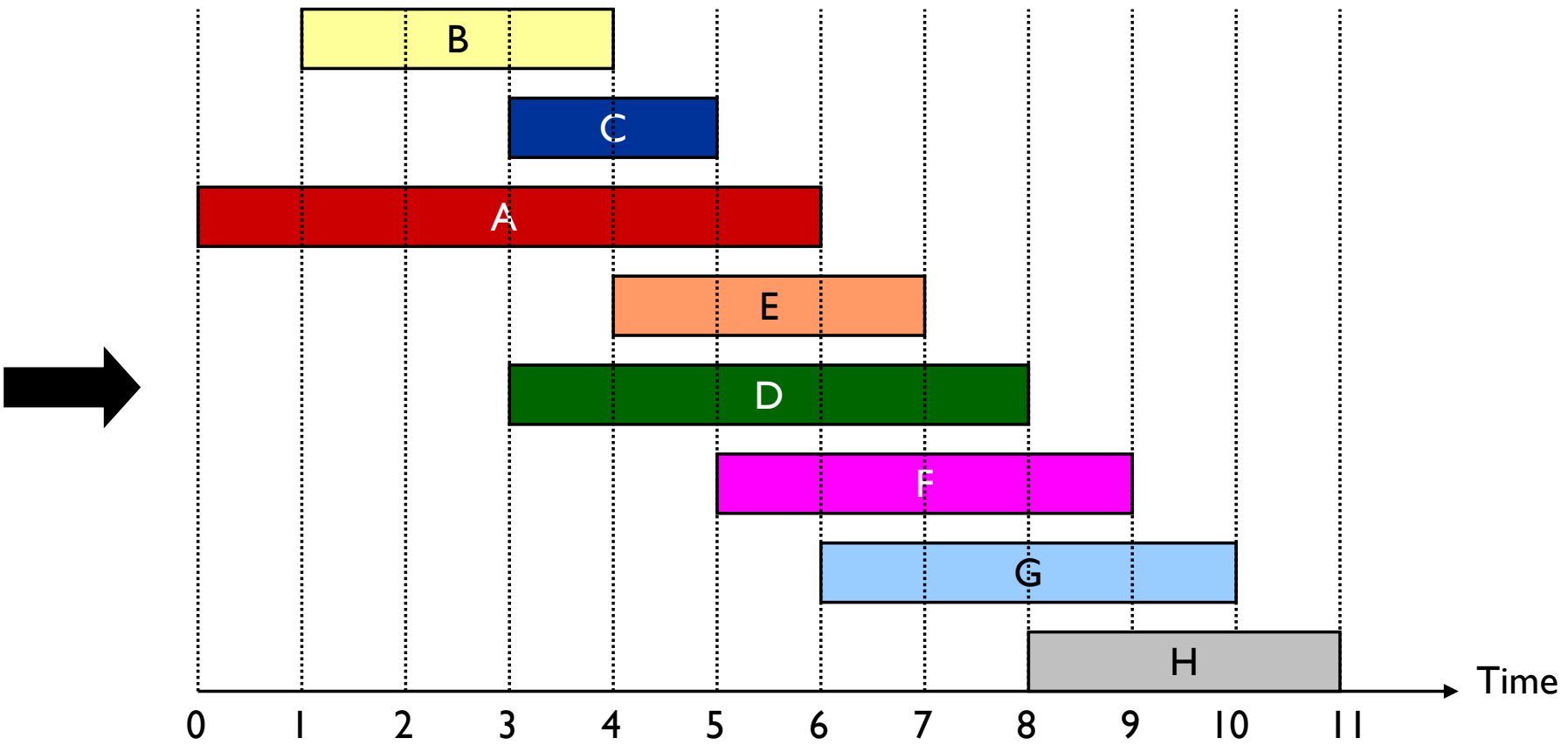


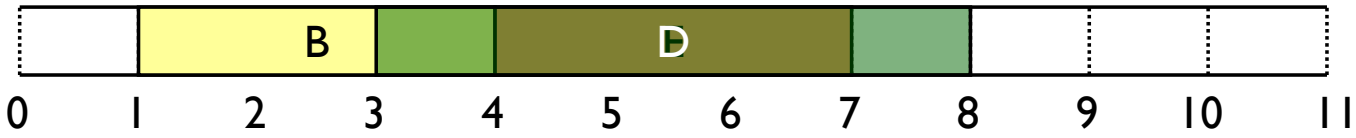


Interval Scheduling

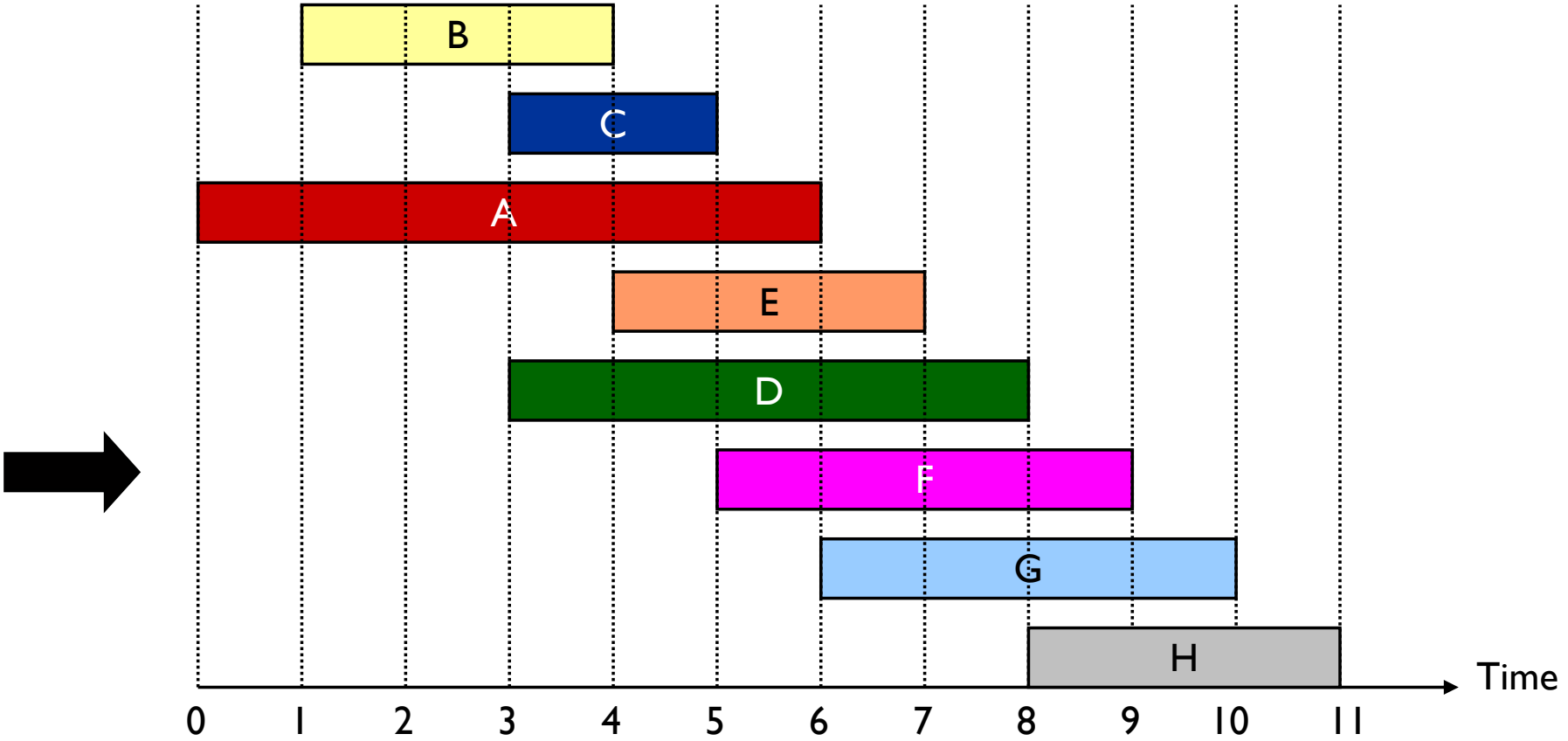


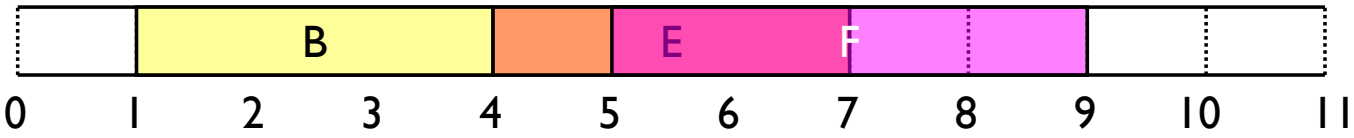
Interval Scheduling



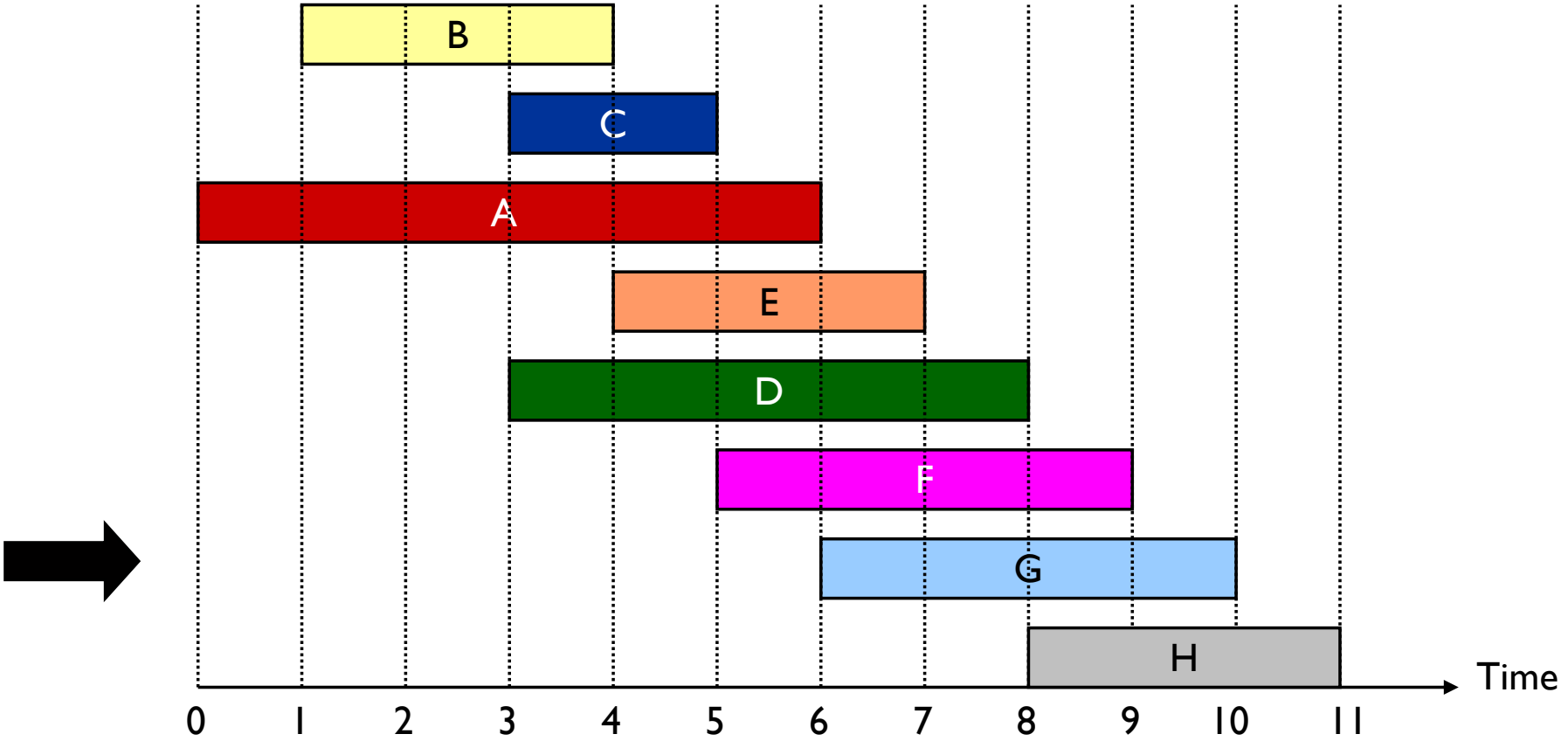


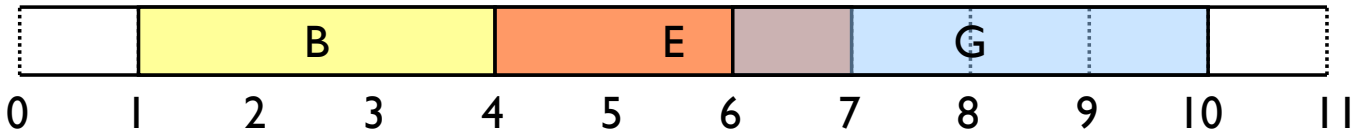
Interval Scheduling



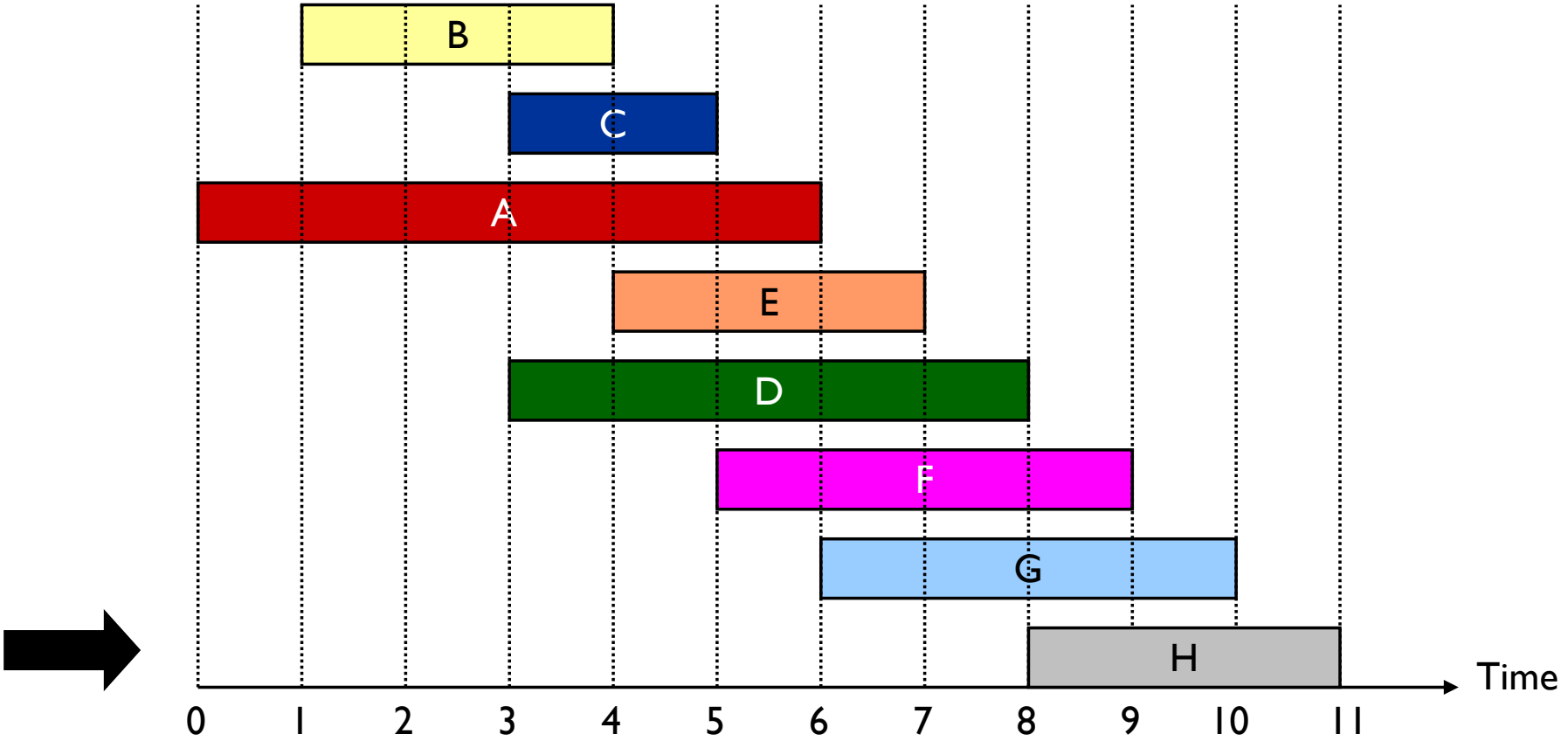


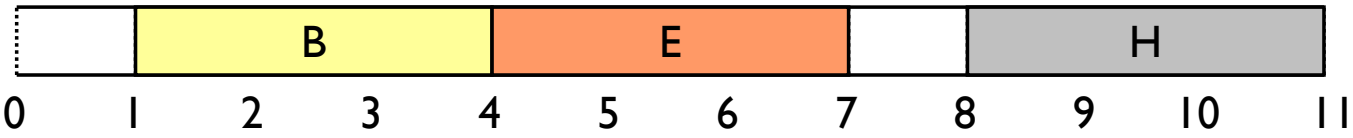
Interval Scheduling





Interval Scheduling





Interval Scheduling: Correctness

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

Pf. ("greedy stays ahead")

(both lists time-ordered)

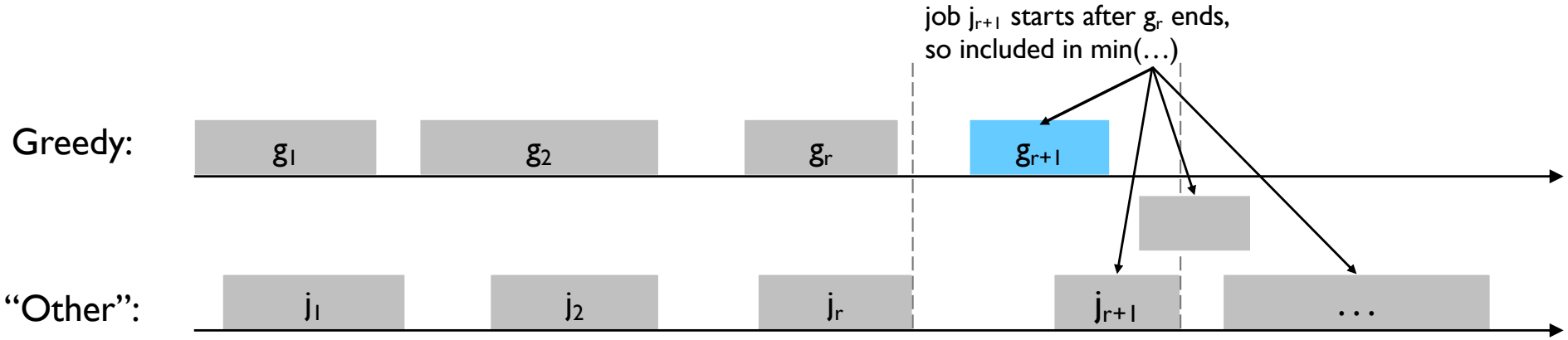
Let g_1, ... g_k be greedy's job picks, j_1, ... j_m those in some other solution

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

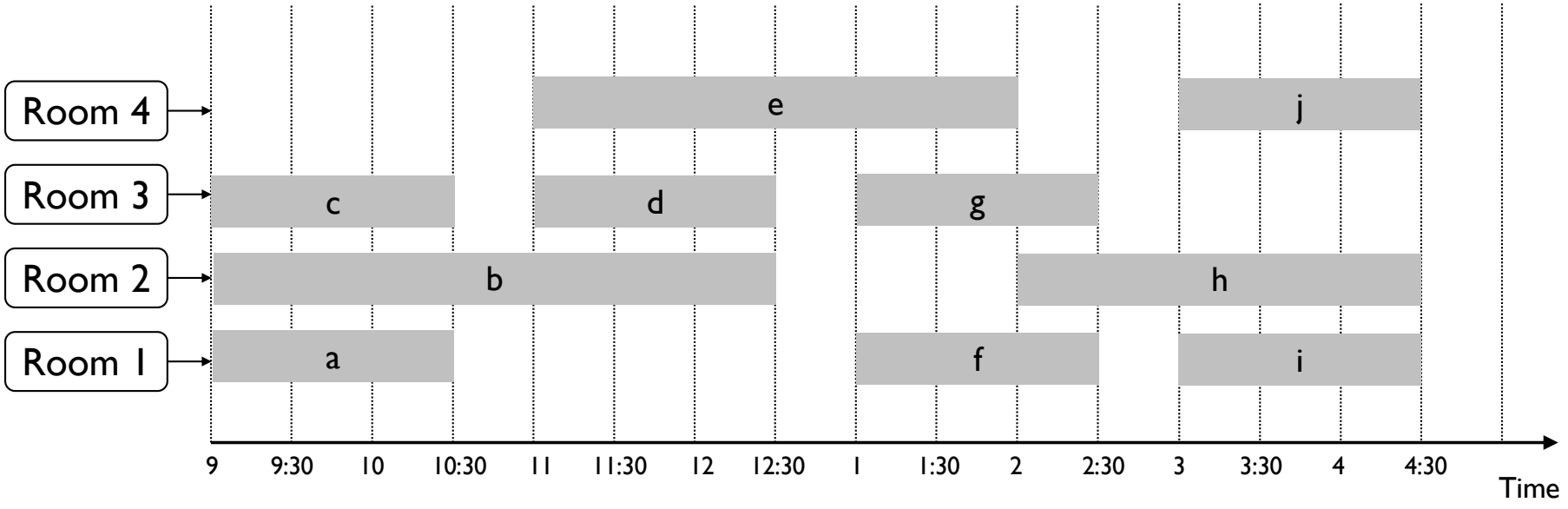
Proof Technique 2: "Structural"

Interval Partitioning
(AKA classroom scheduling)

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.

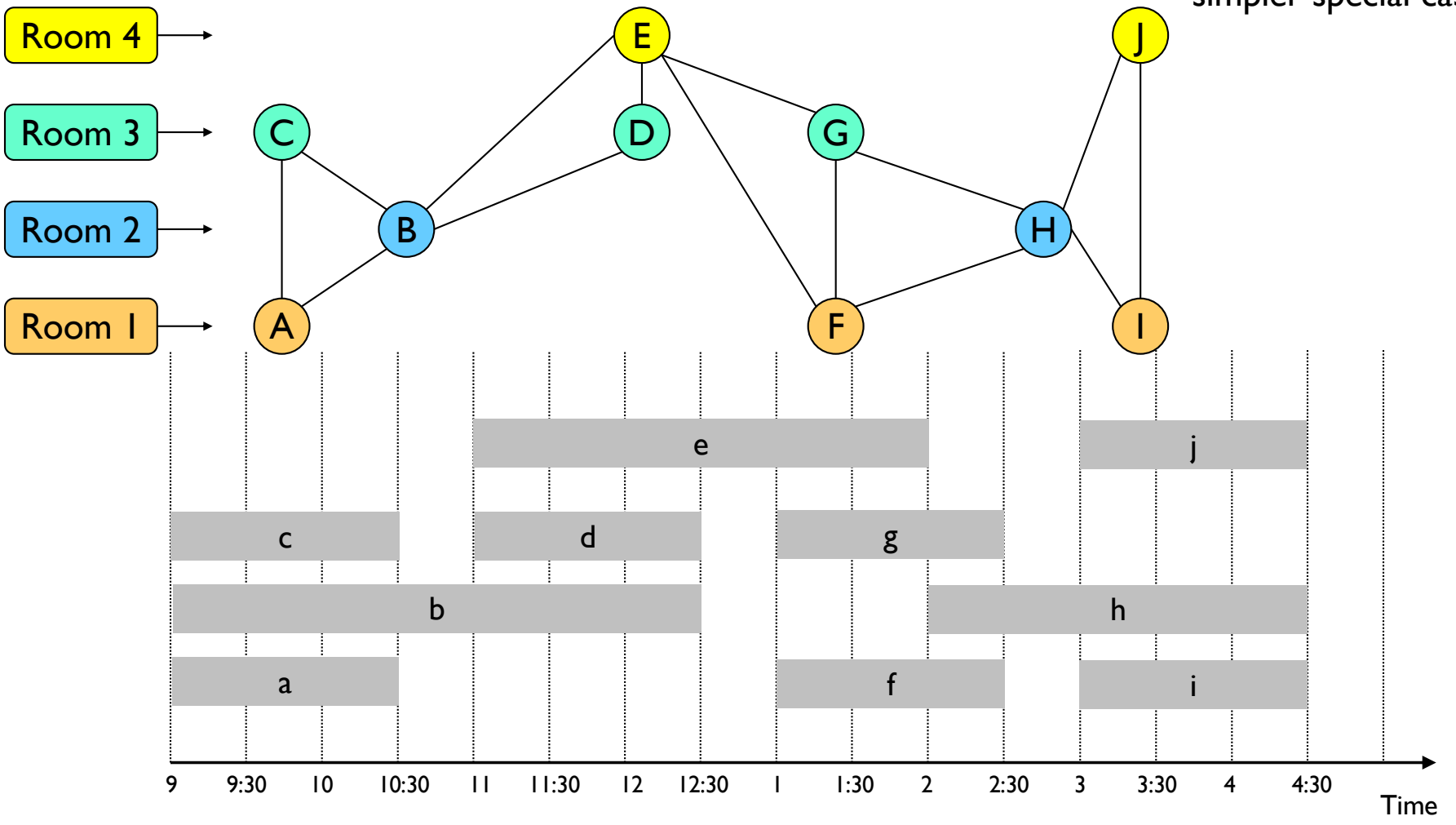
Ex: This schedule uses 4 classrooms to schedule 10 lectures.



Interval Partitioning as Interval Graph Coloring

Vertices = classes;
Edges = conflicting class pairs;
Different colors = different assigned rooms

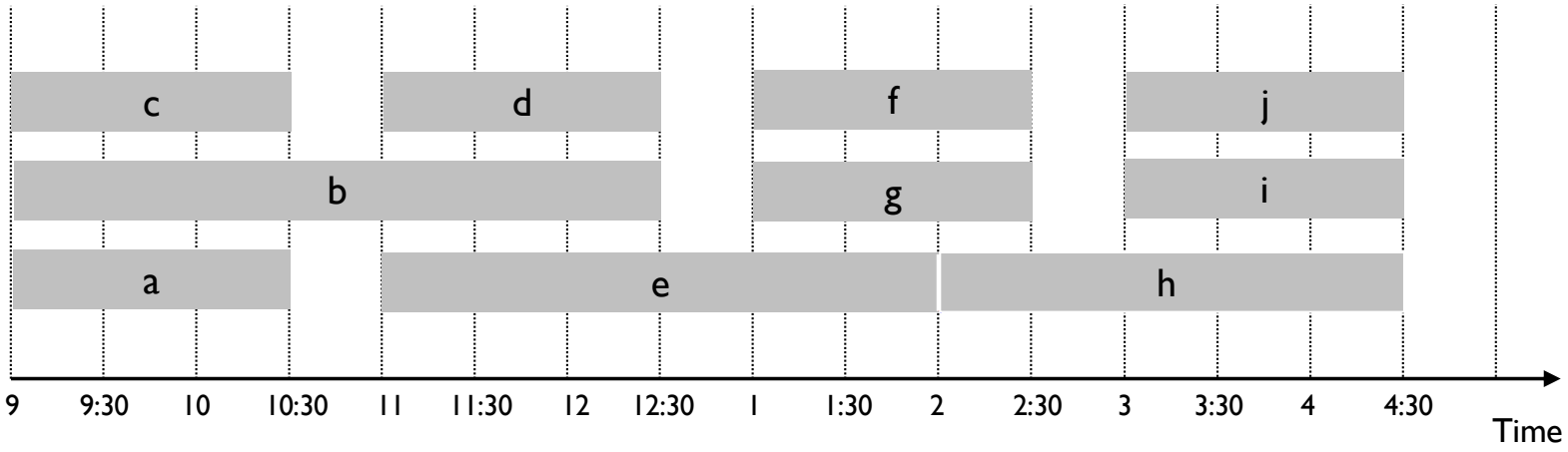
Note: graph coloring is very hard in general, but graphs corresponding to interval intersections are a much simpler special case.



Interval partitioning.

- 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.

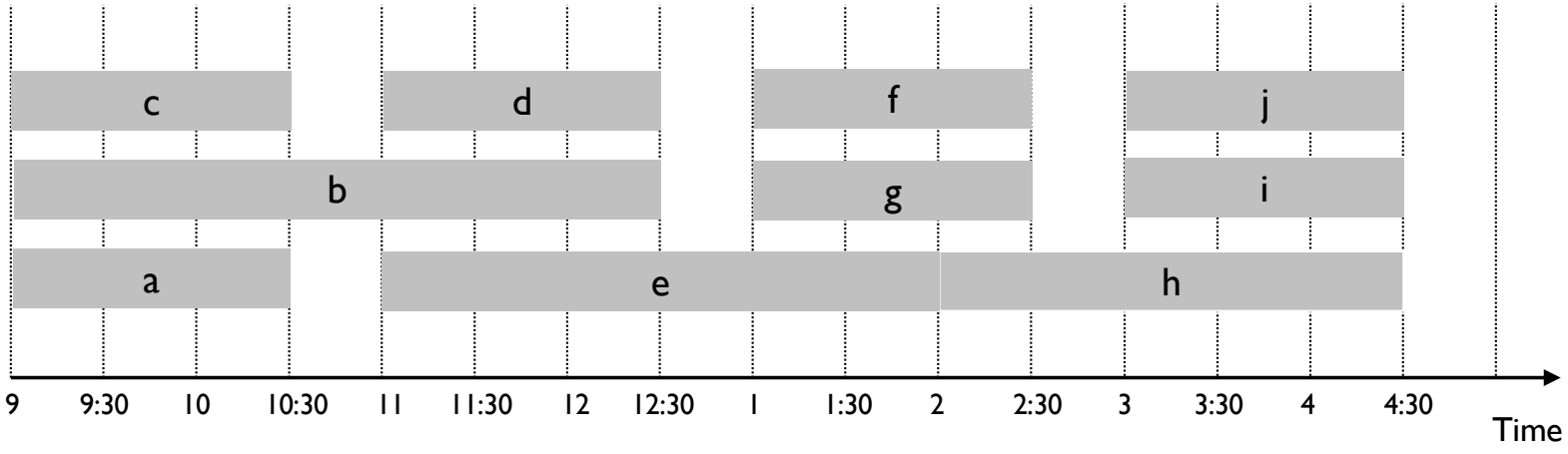
no collisions at ends

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

Ex: Depth of schedule below = 3 $\Rightarrow$ 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 ≤ s_2 ≤ ... ≤ 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
}
```

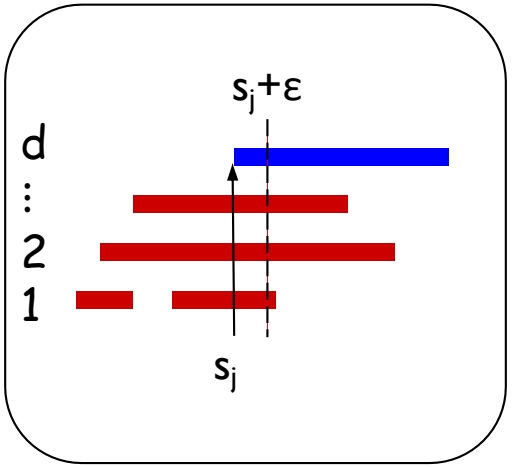
Implementation? Run-time?
Exercises

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 opened when we needed to schedule a job, say j, incompatible with all d-1 previously used classrooms.
- We sorted by start time, so all incompatibilities are with lectures starting no later than s_j.
- So, 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



Exercises: (1) show that the alg fails if not sorted by start time, (2) where is "sortedness" used in the proof above?

4.2 Scheduling to Minimize Lateness

Proof Technique 3: "Exchange" Arguments

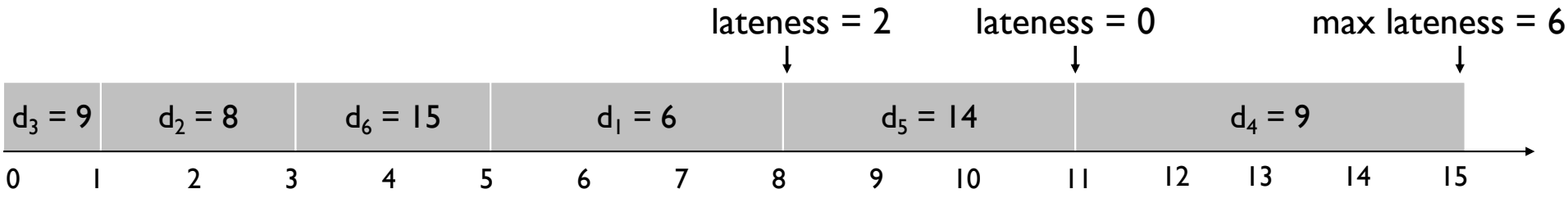
Scheduling to Minimize Lateness

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 job 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 first]

Consider jobs in ascending order of *slack* $d_j - t_j$.

Minimizing Lateness: Greedy Algorithms

Greedy template. Consider jobs in some order.

[Shortest job 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

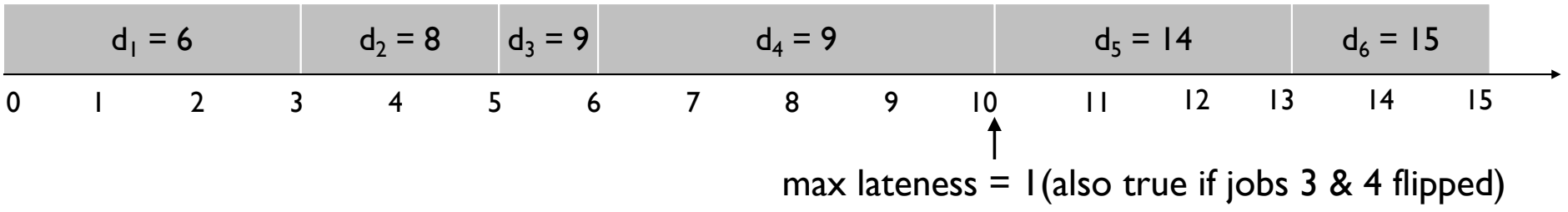
counterexample

Minimizing Lateness: Greedy Algorithm

Greedy algorithm. Earliest deadline first.

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

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

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



Minimizing Lateness

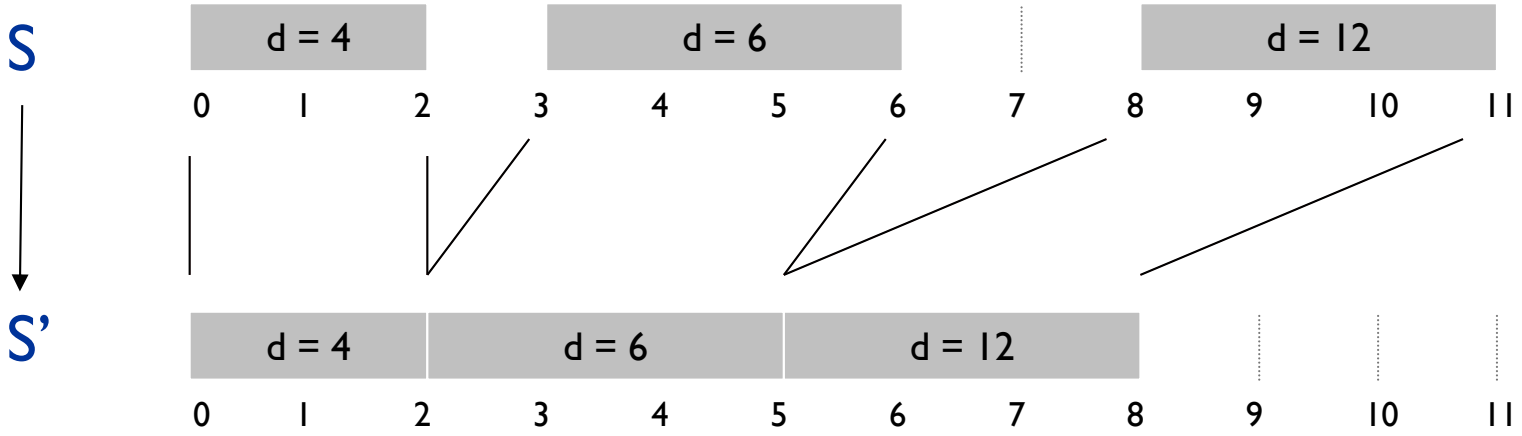
An Alternative Proof

The 6 slides below outline an alternative proof that I think is a little simpler than the one in the text. It uses the same core "exchange argument" idea, while avoiding the correct but slightly tangential discussion of multiple jobs with the same deadline. It also feels a little more algorithm-oriented in that it shows how to turn an arbitrary schedule into exactly the greedy schedule.

I think you will find it instructive to compare this to the text's version.

Minimizing Lateness: No Idle Time

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



No job ends later in S' than S, so max lateness in not increased

Henceforth, assume all schedules are idle-free

Proof Strategy

A *schedule* is an ordered list of jobs. (No idle; only order matters)

Suppose S_1 is any schedule & let G be the the greedy algorithm's schedule

To show: Lateness(S_1) ≥ Lateness(G)

Idea: find simple changes that successively transform S_1 into other schedules increasingly like G, each better (or at least no worse) than the last, until we reach G. I.e.

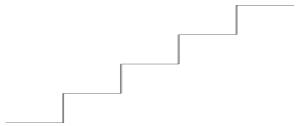
Lateness(S_1) ≥ Lateness(S_2) ≥ Lateness(S_3) ≥ … ≥ Lateness(G)

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

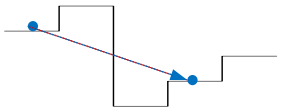
HOW?: *exchange* pairs of jobs

Minimizing Lateness: Inversions

(Re-)number the jobs in the order that Greedy schedules them. Then a "Schedule" is just permutation of 1..n. E.g.:



G: 1 2 3 4 5



S: 4 5 1 2 3

Def. An *inversion* in schedule S is a pair of jobs i and j s.t. greedy did i before j (i.e., i < j), but S does j before i.

E.g., (4,2) in S above; also (4,1), (5,3), …

Claim 2: If schedule S has an inversion, it has an *adjacent* inversion, i.e., a pair of inverted jobs scheduled consecutively.

Ex: (4,2) are not adjacent, but (5,1) is an adjacent inversion

Pf: If j,i is an inversion, the sublist of S from j to i must have an adjacent inversion since i is smaller than j. "A walk from high to low must have a 1st step down."

Ex: 4 5 1 2

Minimizing Lateness: Inversions

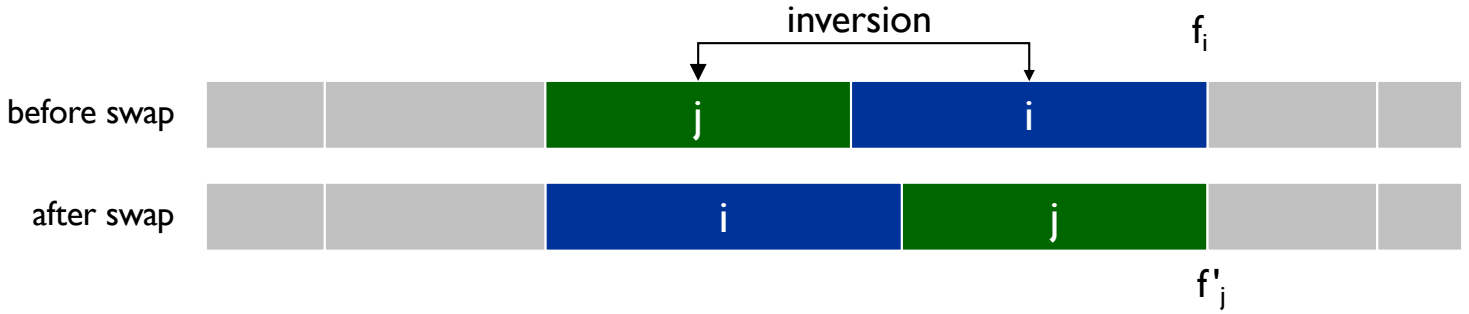
Claim 3: Swapping an *adjacent* inversion reduces the total number of inversions by 1 (exactly)

Pf: To be clear about the defn, since S is just a list of the numbers between 1 and n, in some order, for any p≠q in 1..n, p,q is an inversion ⇔ the larger precedes the smaller in list S. Let i, j be an adjacent inversion. Inversion status of any pair p,q is unchanged by i⟷j swap unless {p,q} = {i,j}, and the i,j inversion is removed by that swap. In more detail, if neither p nor q is either i or j, then neither p nor q moves, so status is unchanged. If one of p,q is i or j, say, p≠i, and q=j, then since j is moved *only one position* in the list (i & j are adjacent), it can't move to the other side of p, and again status is unchanged.

Minimizing Lateness: Inversions

Def. An *inversion* in schedule S is a pair of jobs i and j s.t. greedy did i before j (i.e., i < j), but S does j before i.

Claim 4. *Swapping* two adjacent, inverted jobs does not increase the max lateness.



(j had later or equal deadline, so is not tardier after swap than i was before swap)

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

Theorem. Greedy schedule G is optimal

Pf. Let S_1 be an optimal schedule. If S_1 has idle time, by claim 1, we can remove it to form S_2 without increasing lateness. If S_2 has any inversions, by claim 2 it has an adjacent inversion, and by claims 3 & 4, we can swap to form S_3 which has fewer inversions and no greater maximum lateness. Repeating this produces an idle-free, inversion-free schedule, which is exactly the greedy schedule G, without ever having increased lateness. Hence Lateness(G) ≤ Lateness(S_1), and so is optimal.

A slightly tidier way to say this:

Among all optimal schedules, let S* be one with the fewest inversions, and, by claim 1, no idle time. If S* has inversions, it has adjacent inversions (claim 2); swapping one decreases the number of inversions (claim 3) without increasing maximum lateness (claim 4), contradicting choice of S*. So, S* has no inversions nor idle time. But that's exactly schedule G, hence G is optimal.

Optional Exercise

Here's an outline for a third proof, that is, in my opinion, even simpler. You might enjoy fleshing this out as an exercise.

Defn: two vectors $(u_1, u_2, \ldots u_n)$ and $(v_1, v_2, \ldots, v_n)$ are *lexicographically ordered,* $u \prec v$, if for some i, $u_1=v_1$, $u_2=v_2$, …, $u_{i-1}=v_{i-1}$, and $u_i < v_i$
I.e., they're identical in first i-1 positions, and u is smaller in the i^{th} , the first position where they differ.

Ex: the 6 permutations of 1,2,3 in lex order: $123 \prec 132 \prec 213 \prec 231 \prec 312 \prec 321$

Proof Outline: Let S* be the lexicographically first idle-free optimal schedule. Argue by contradiction that S* = G, since otherwise, letting i be the 1^{st} position where they differ, S* looks like

(1, 2, 3, …, i-2, i-1, x, y, …, z, i, …) where $x \neq i$.

But z must be larger than i (why?), so z,i is an adjacent inversion; flipping it gives a lexicographically *smaller* sequence of no larger max lateness, contradicting choice of S*. (This uses claims 1 & 4 above; claims 2 & 3 are no longer needed.)

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 in finding a useful structural characteristic.)

Exchange argument. Gradually transform any solution into 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, too.)

4.4 Shortest Paths in a Graph

You've seen this in prerequisite courses, so this section of the text 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, hint, hint: *very* frequent fodder for job interview questions…)

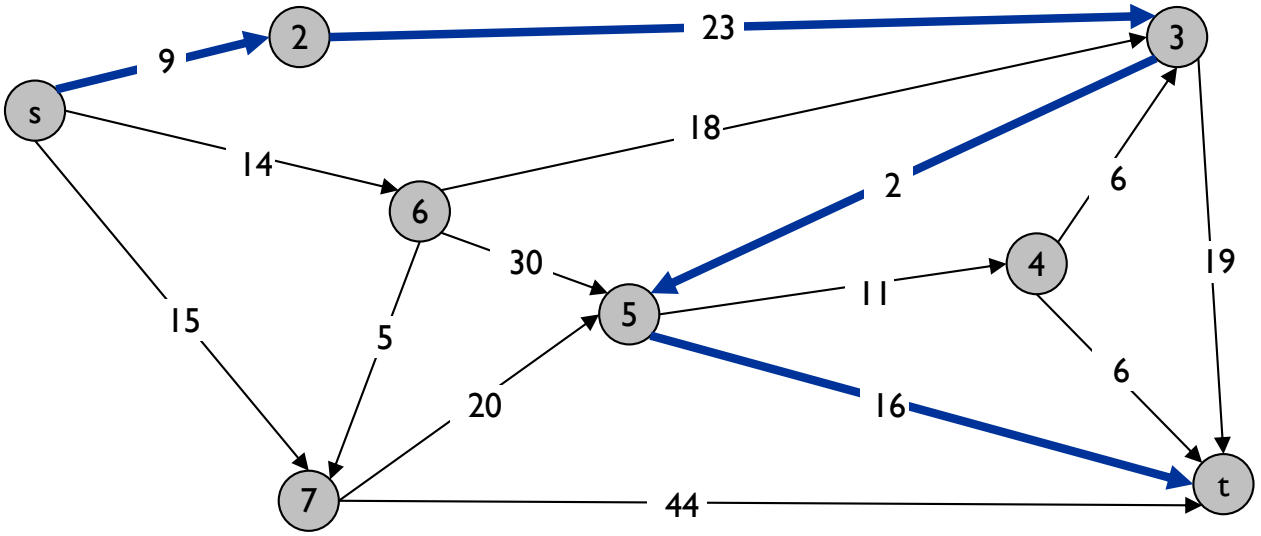
Shortest Path Problem

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

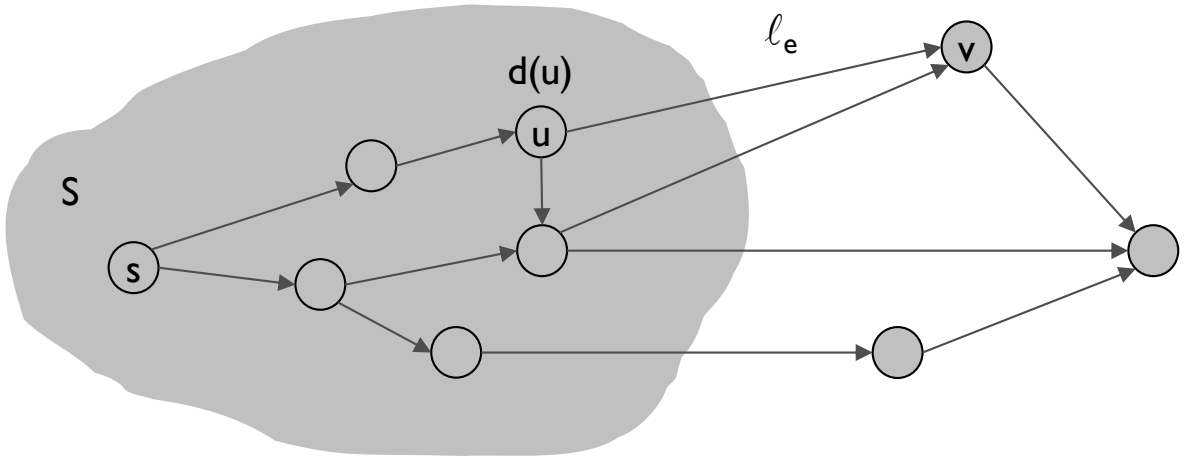
Dijkstra's algorithm.

- 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

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

add v to S, and set d(v) = π(v).

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



Dijkstra's Algorithm

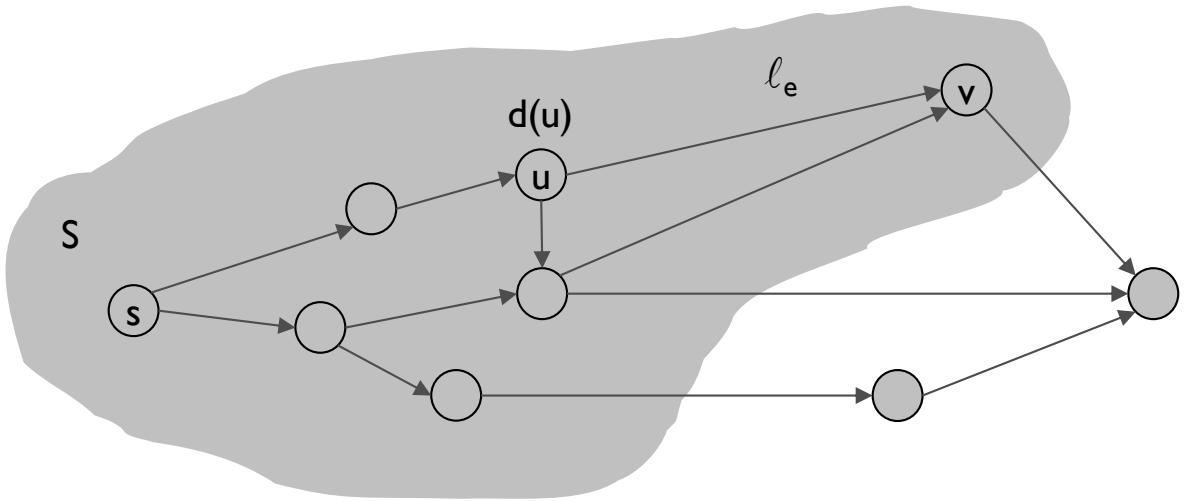
Dijkstra's algorithm.

- 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

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

add v to S, and set d(v) = π(v).

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



Summary

"Greedy" algorithms: often natural, intuitive, simple, efficient
But seductive – often incorrect!
    E.g., "Change making," depends on available denominations
So, we looked at a few examples, each useful in its own right,
    but emphasized *correctness*, and various approaches to
    reasoning about these algorithms

Interval Scheduling – greedy stays ahead
Interval Partitioning – greedy matches structural lower bound
Minimizing Lateness – exchange arguments

Next: Huffman codes and another exchange argument

Also: This is a good time to review shortest paths and min
    spanning trees (is there a job interview in your future?)