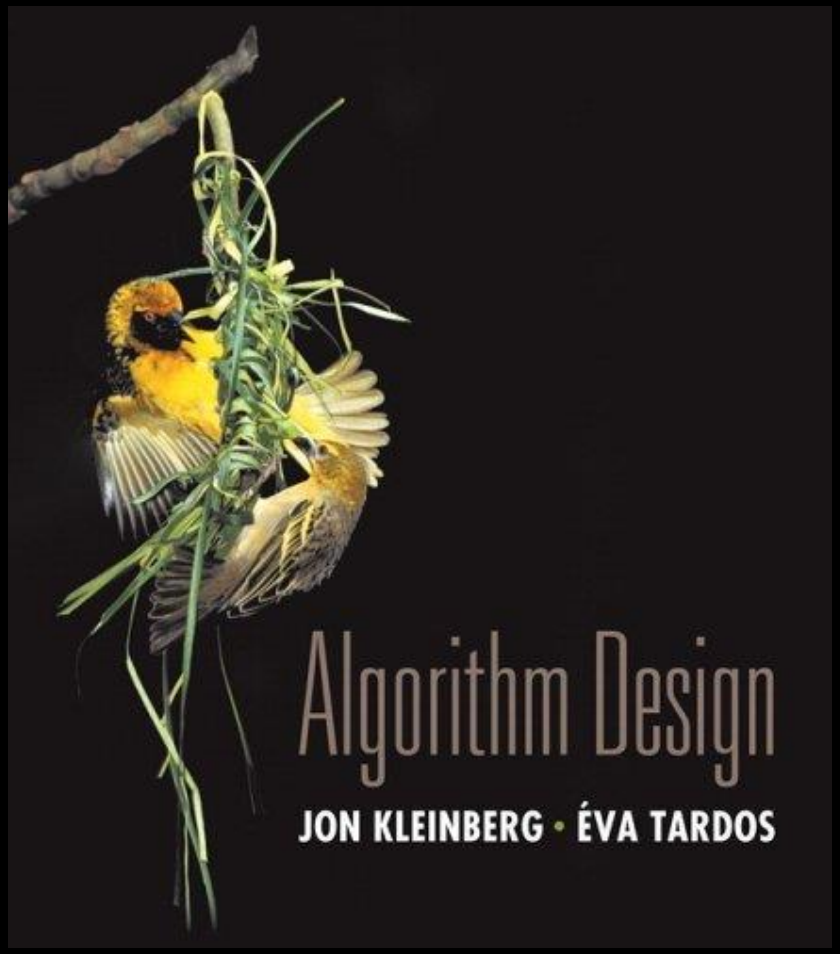


Chapter 4

Greedy Algorithms

Slides by Kevin Wayne.
Copyright © 2005 Pearson-Addison Wesley.
All rights reserved.

Greedy Algorithms

A greedy algorithm is an algorithm that makes the locally optimal choice at each step.

4.1 Interval Scheduling

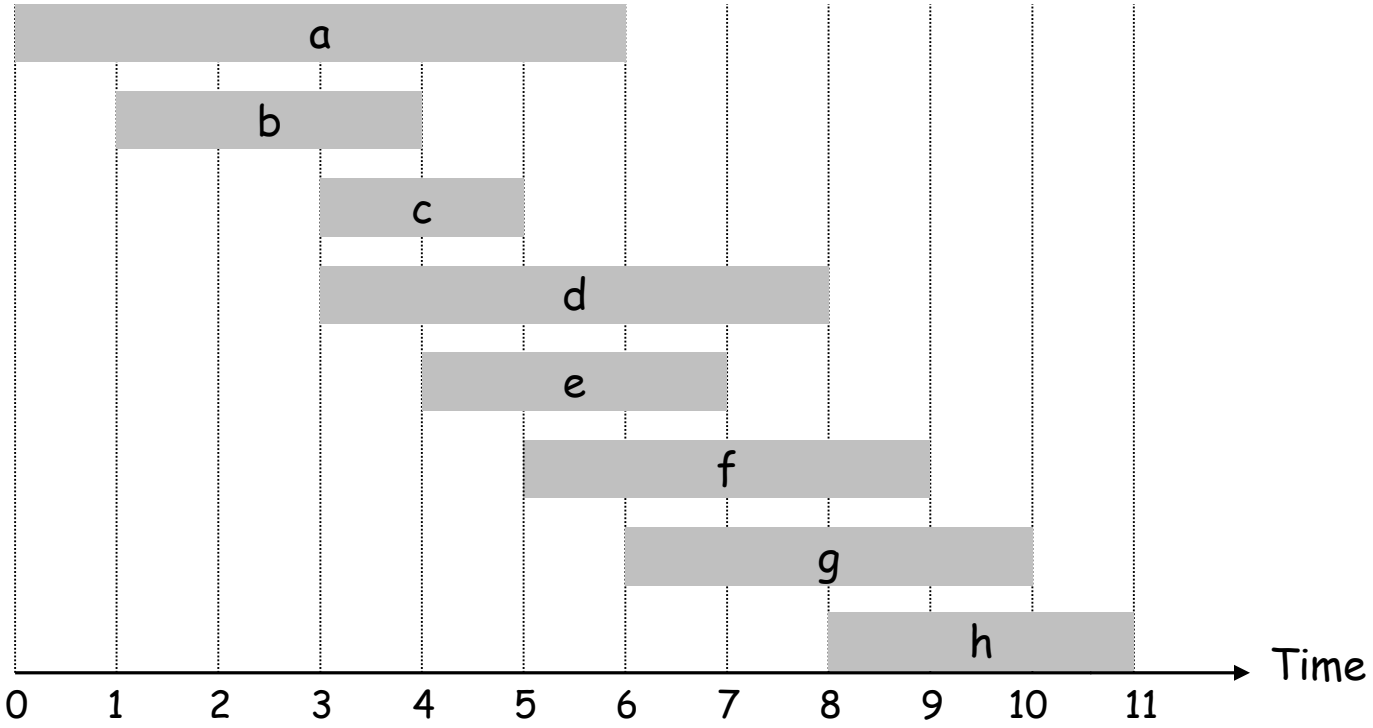
Interval Scheduling

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.

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

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

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

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

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?

shortest interval?

fewest conflicts?

Interval Scheduling: Greedy Algorithm

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

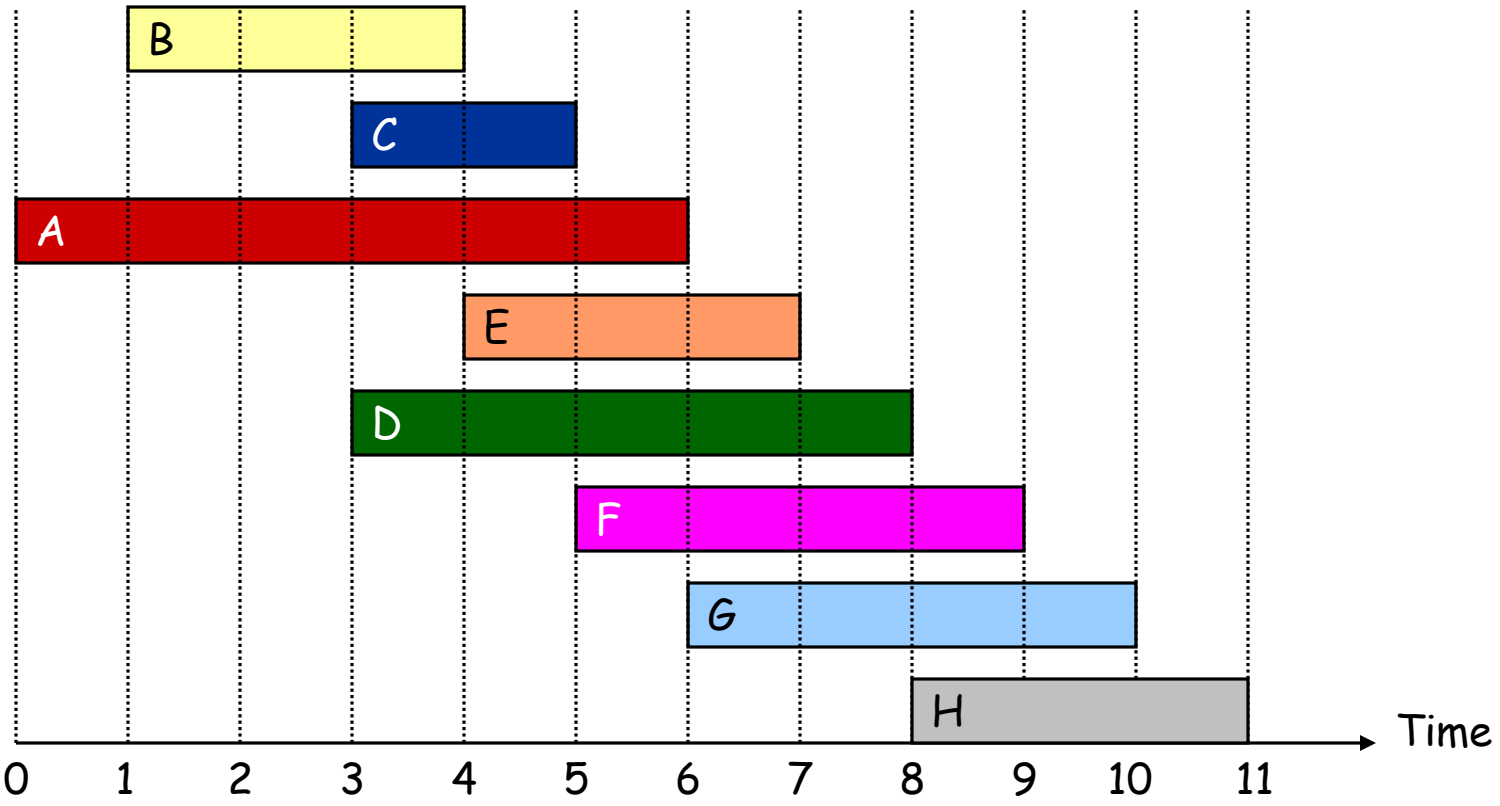
```
Sort jobs by finish times so that f_1 ≤ f_2 ≤ ... ≤ f_n.

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

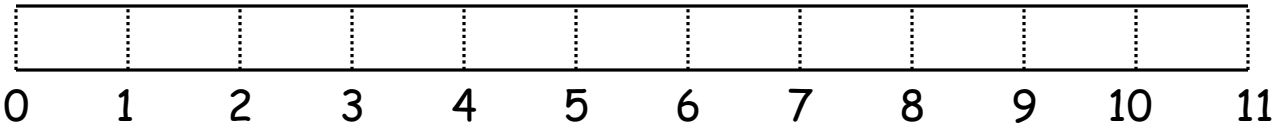
Implementation. $O(n \log n)$.

Remember job j^* that was added last to A.

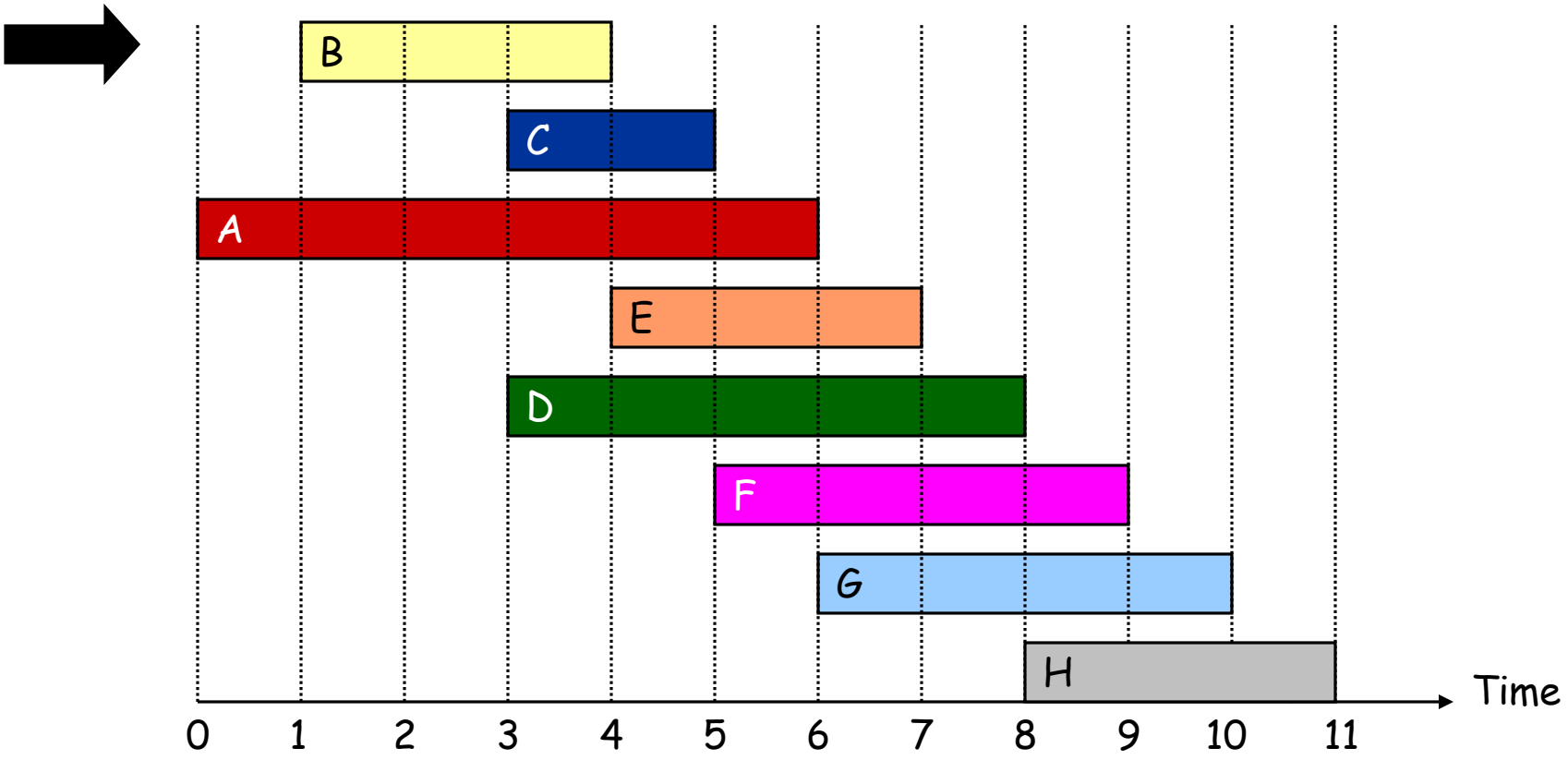
Job j is compatible with A iff $s_j \geq f_{j^*}$.

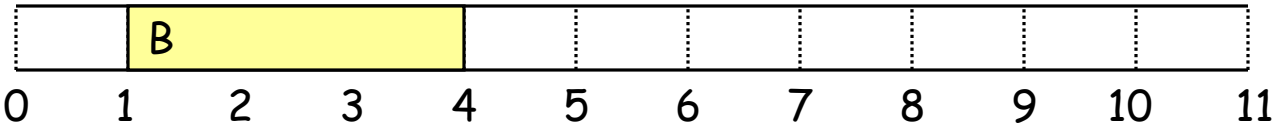
Interval Scheduling



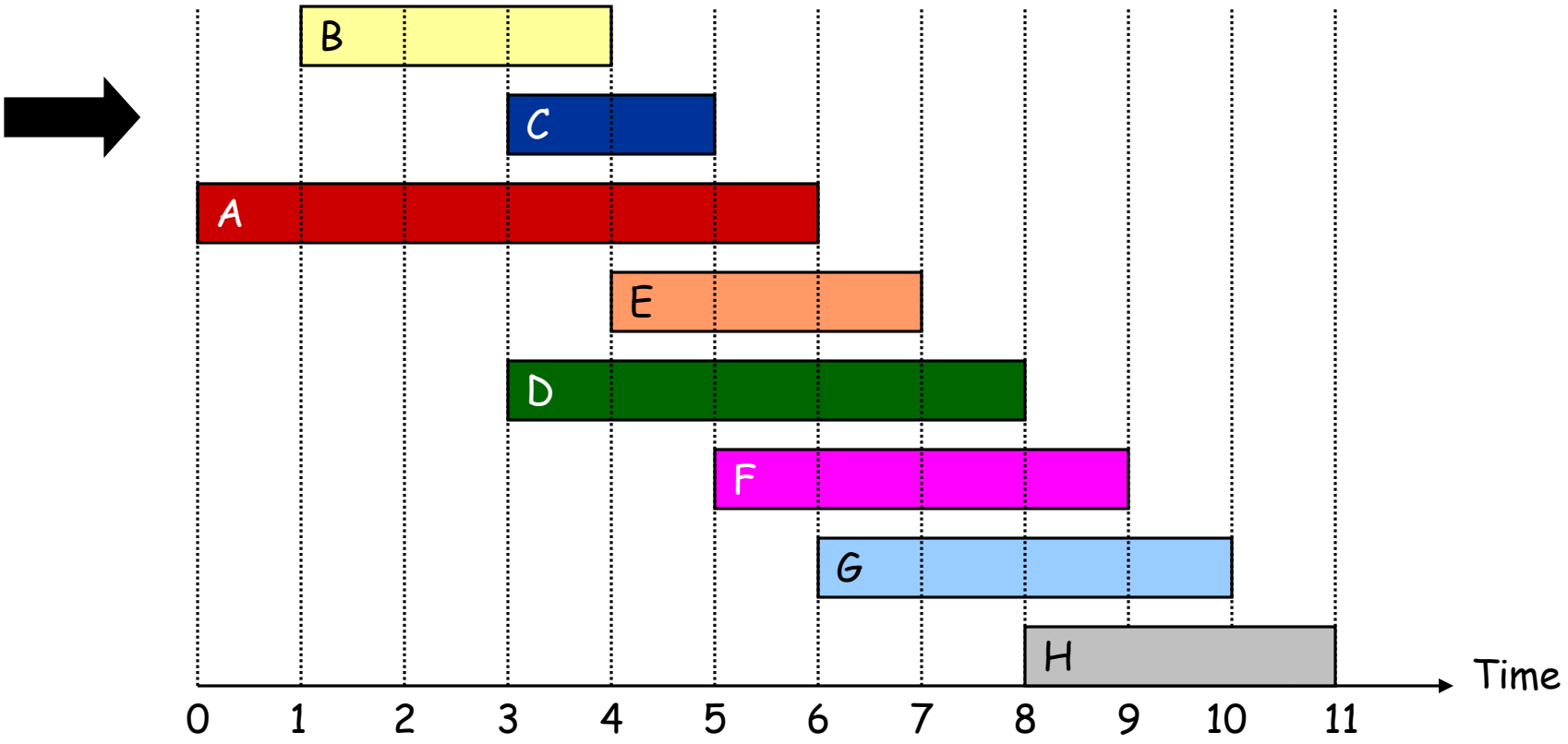


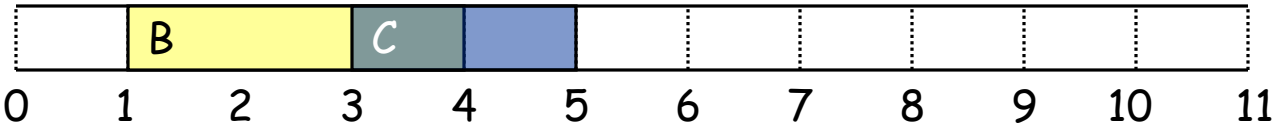
Interval Scheduling



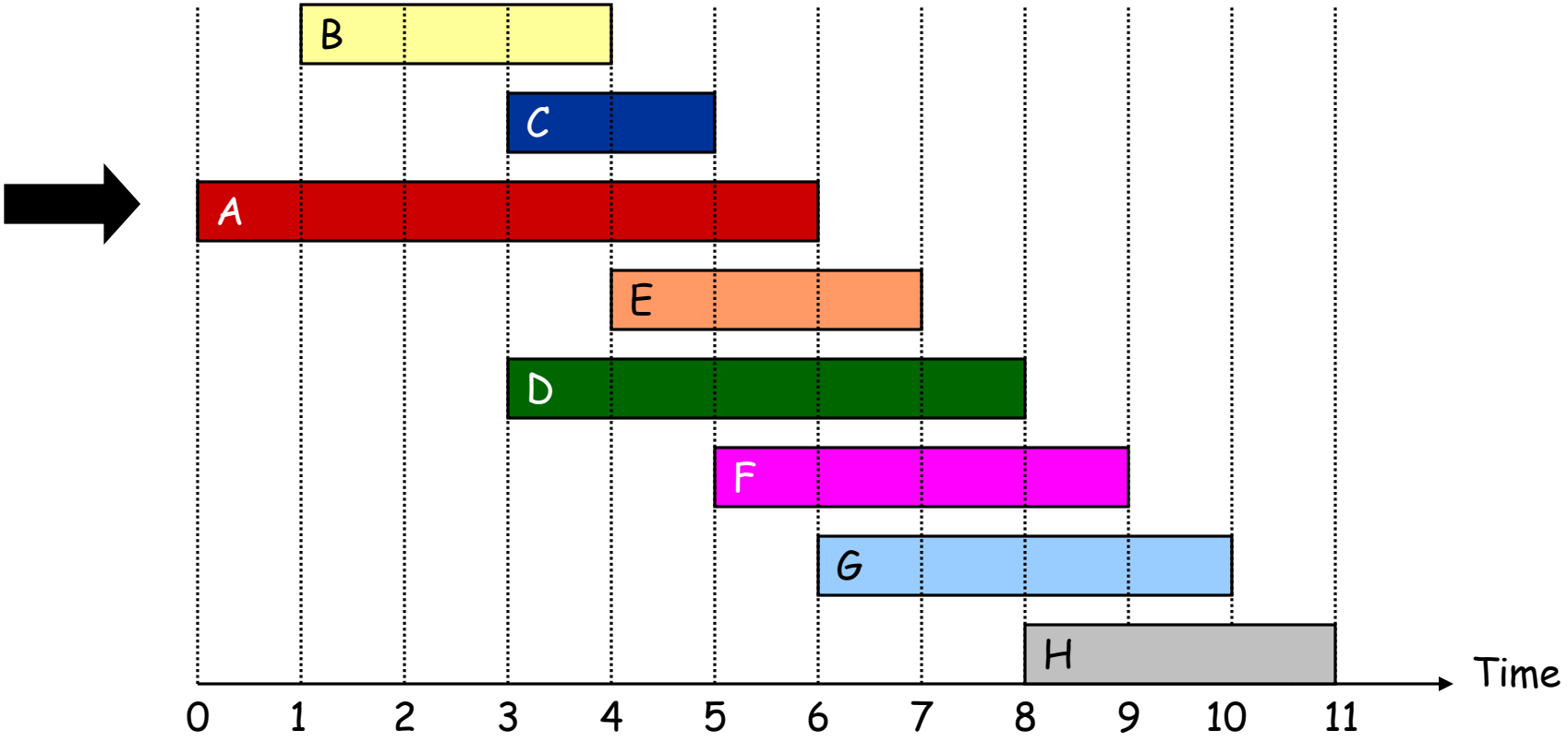


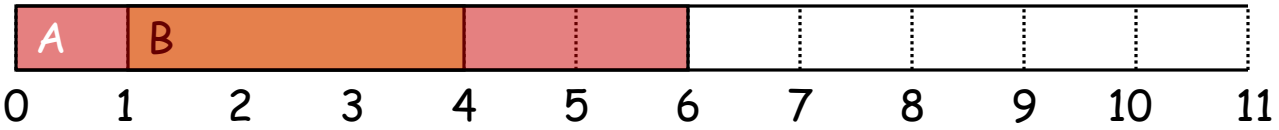
Interval Scheduling



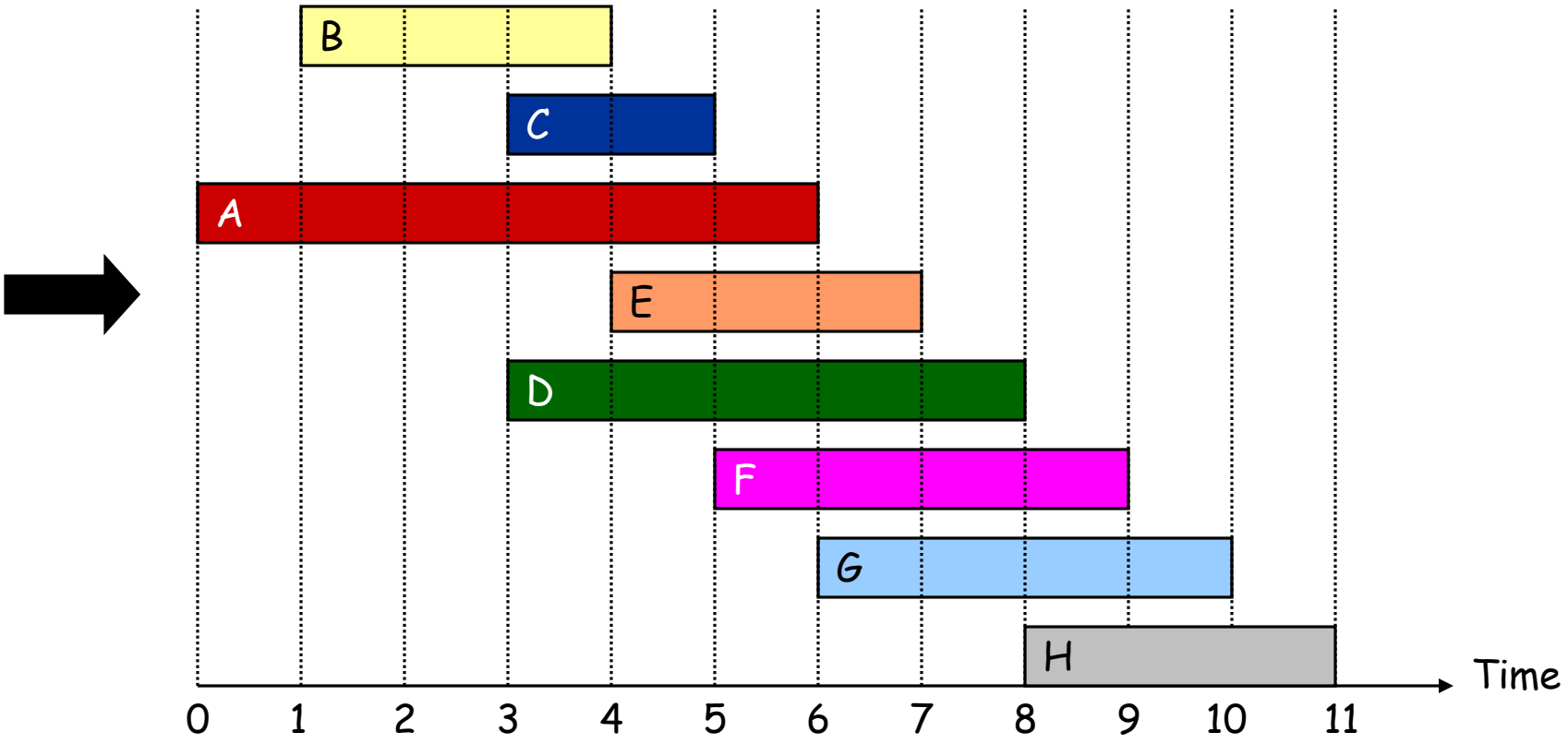


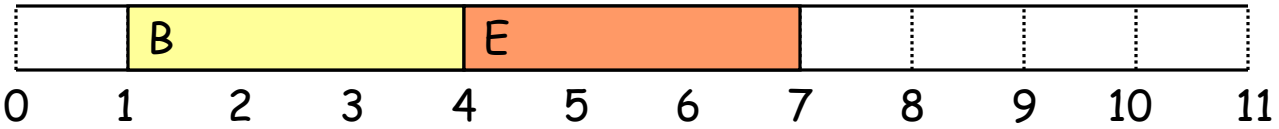
Interval Scheduling



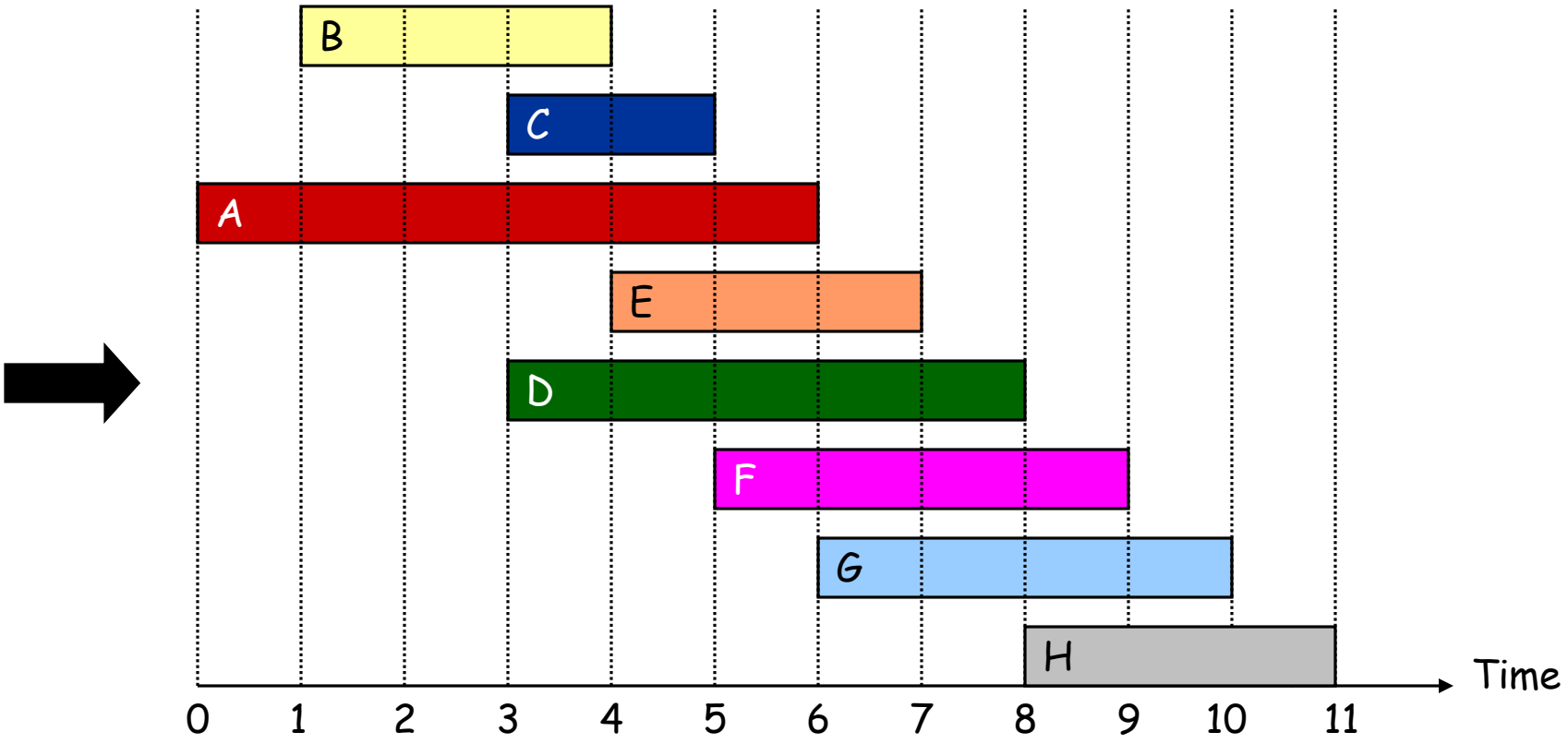


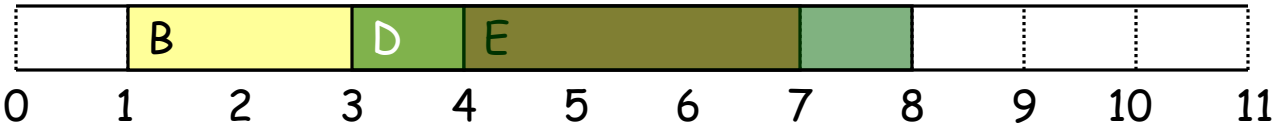
Interval Scheduling



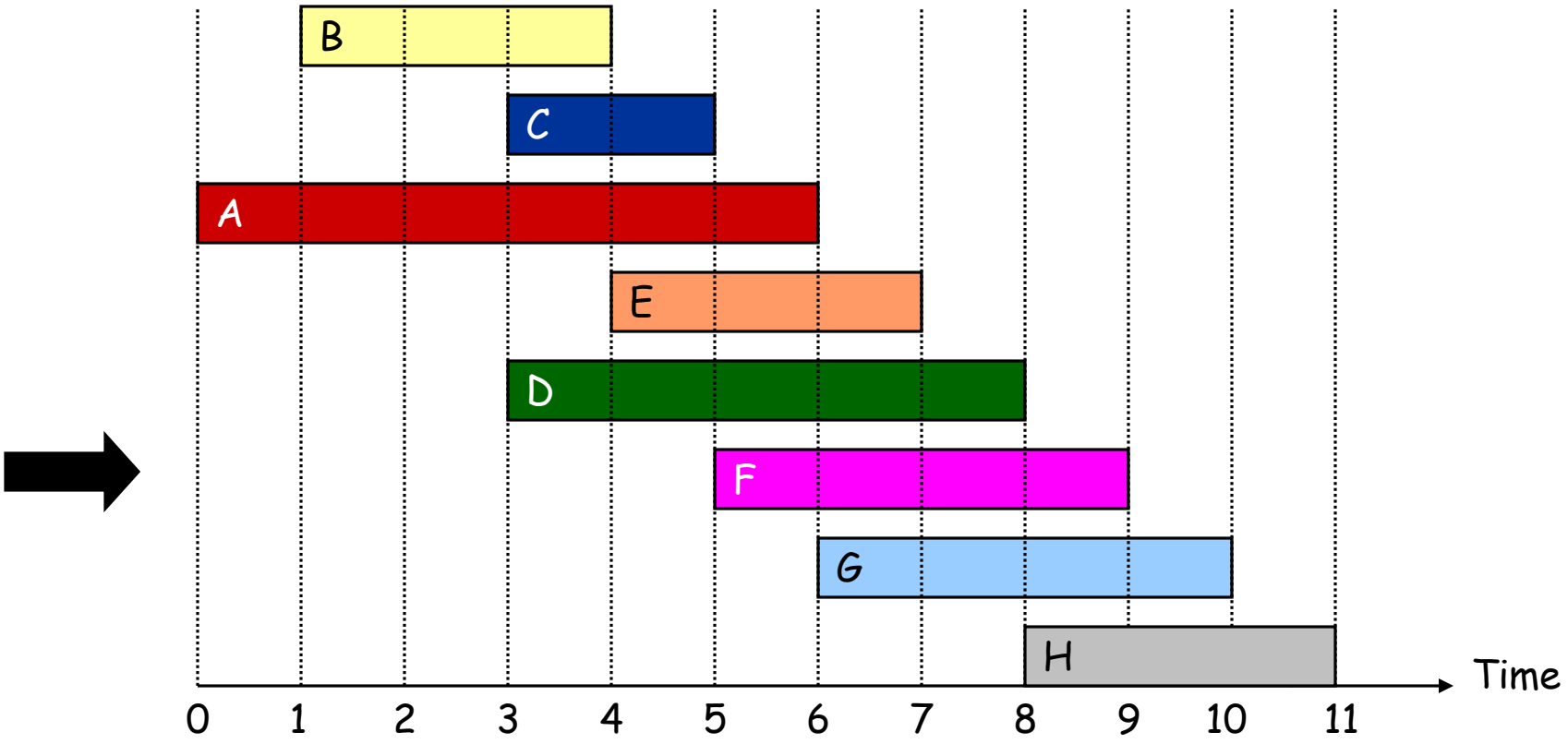


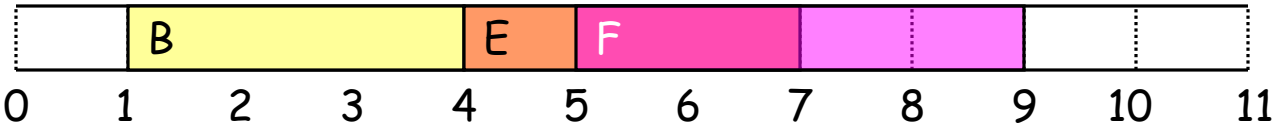
Interval Scheduling



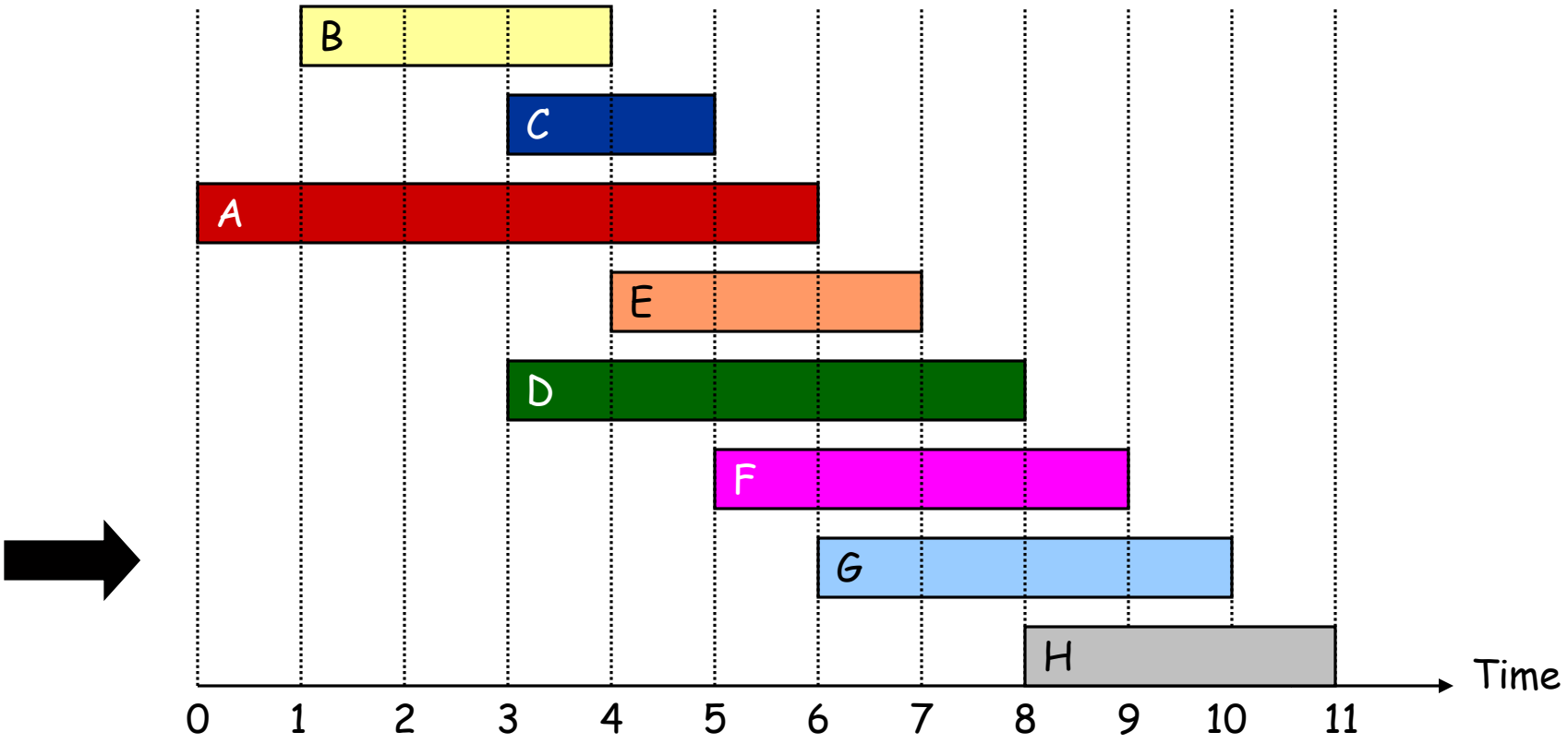


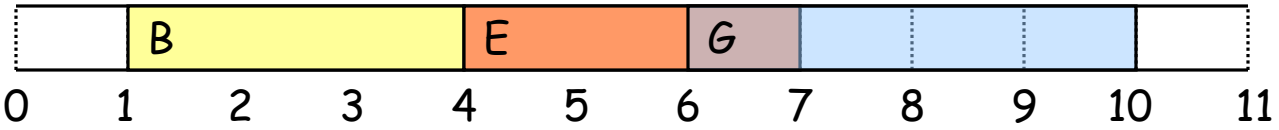
Interval Scheduling



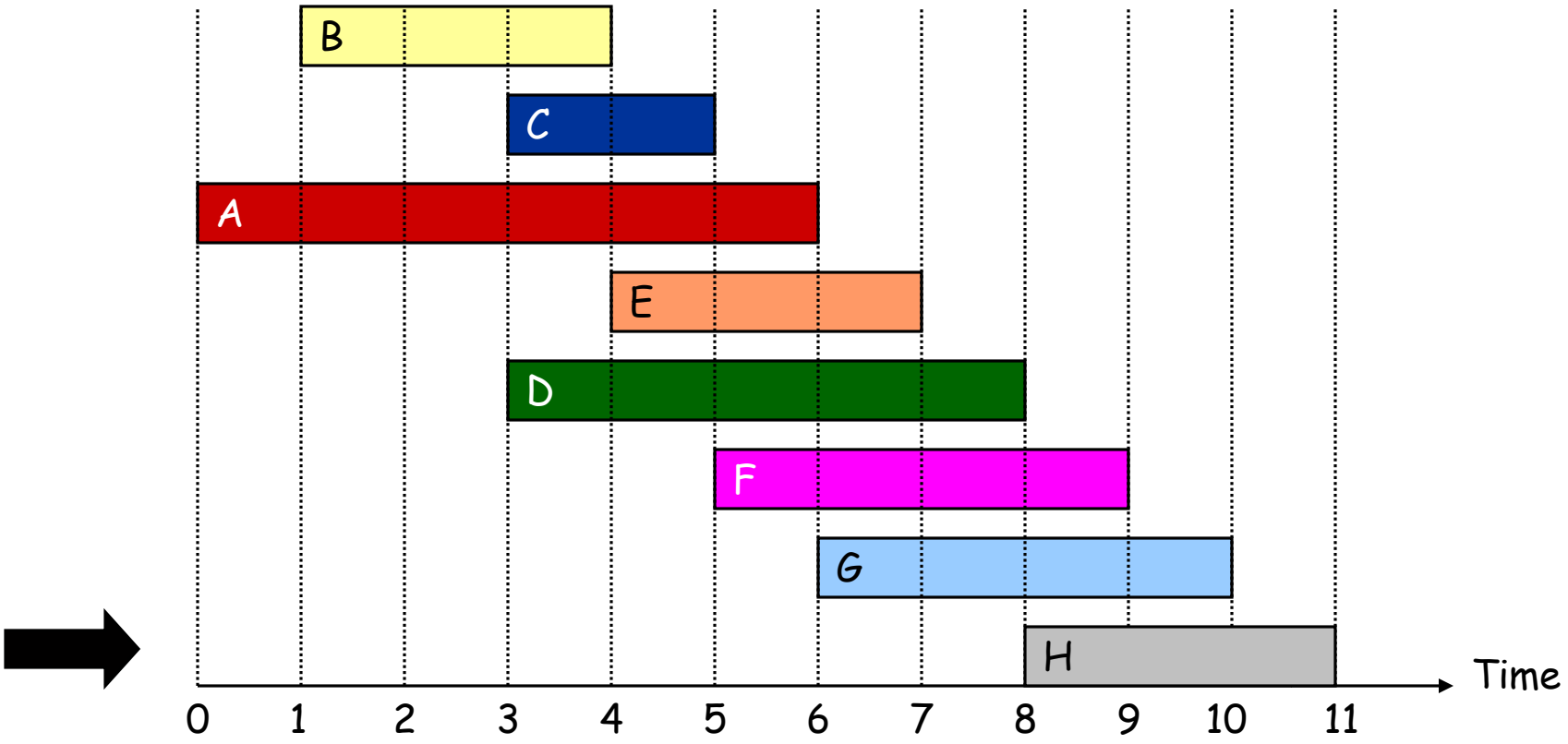


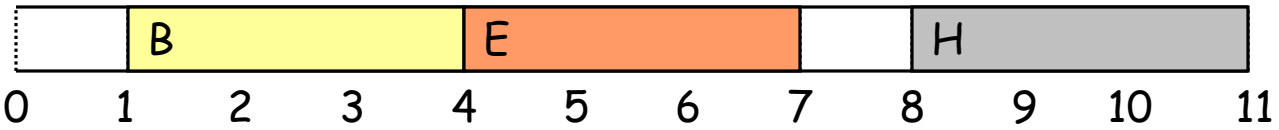
Interval Scheduling





Interval Scheduling





Interval Scheduling: Analysis

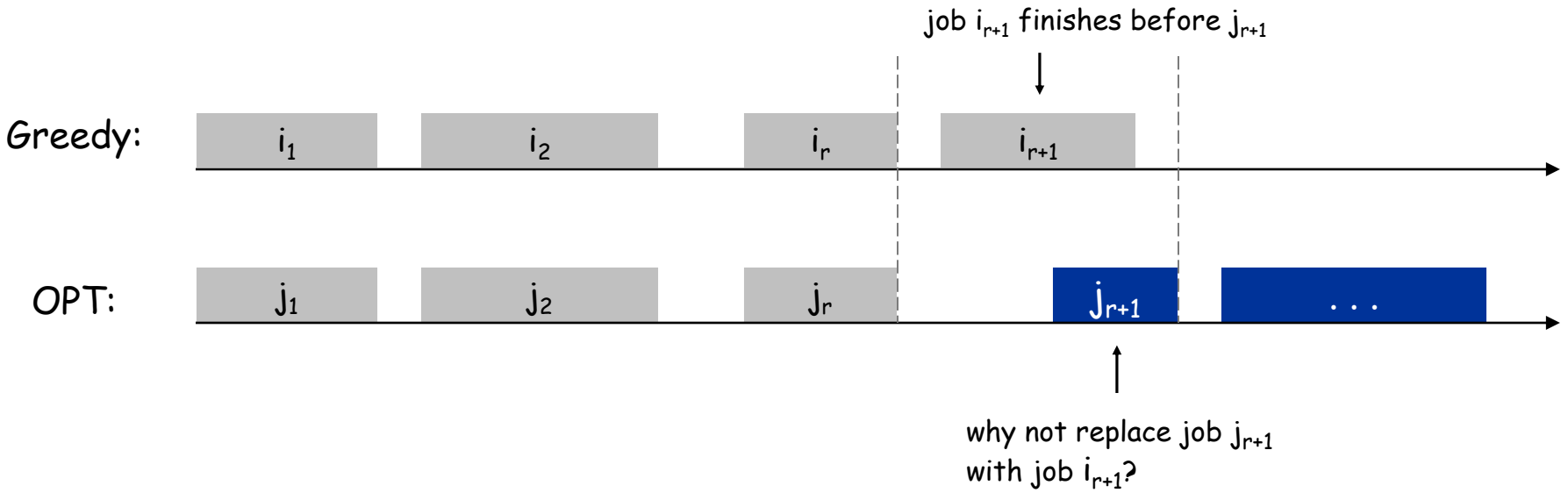
Theorem. Greedy algorithm is optimal.

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 the optimal solution with $i_1 = j_1, i_2 = j_2, \ldots, i_r = j_r$ for the largest possible value of r.



Interval Scheduling: Analysis

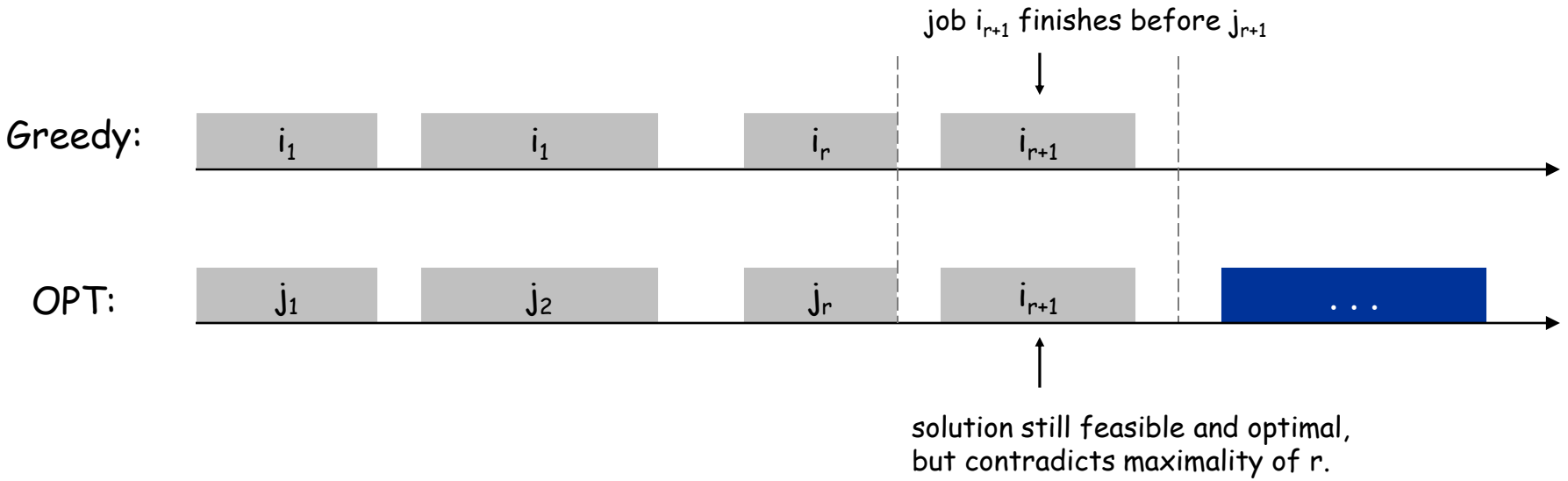
Theorem. Greedy algorithm is optimal.

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 the optimal solution with $i_1 = j_1, i_2 = j_2, \ldots, i_r = j_r$ for the largest possible value of r.



4.2 Scheduling to Minimize Lateness

Scheduling to Minimizing Lateness

Minimizing lateness problem.

Single resource processes one job at a time.

Job j requires t_j units of processing time and is due at time d_j.

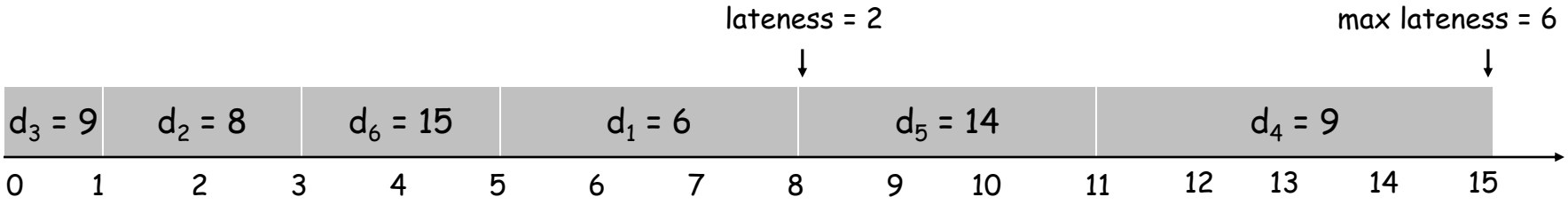
If j starts at time s_j, it finishes at time $f_j = s_j + t_j$.

Lateness: $\ell_j = \max \{ 0, f_j - d_j \}$.

Goal: schedule all jobs to minimize maximum lateness $L = \max \ell_j$.

Ex:

	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 jobs in ascending order of processing time t_j.

	1	2
t_j	1	10
d_j	100	10

counterexample

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

	1	2
t_j	1	10
d_j	2	10

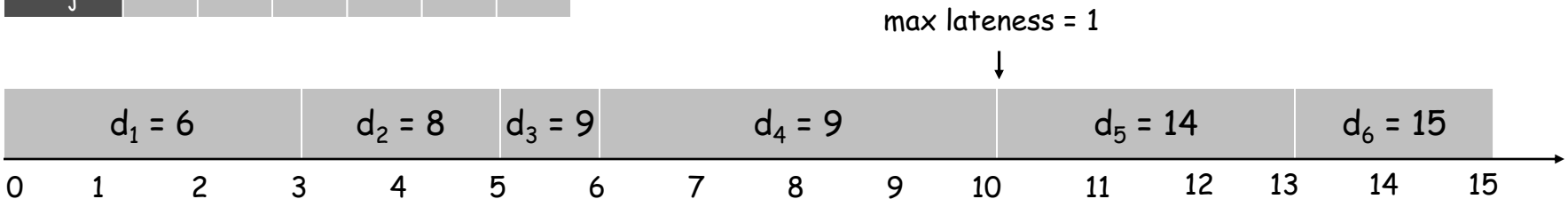
counterexample

Minimizing Lateness: Greedy Algorithm

Greedy algorithm. Earliest deadline first.

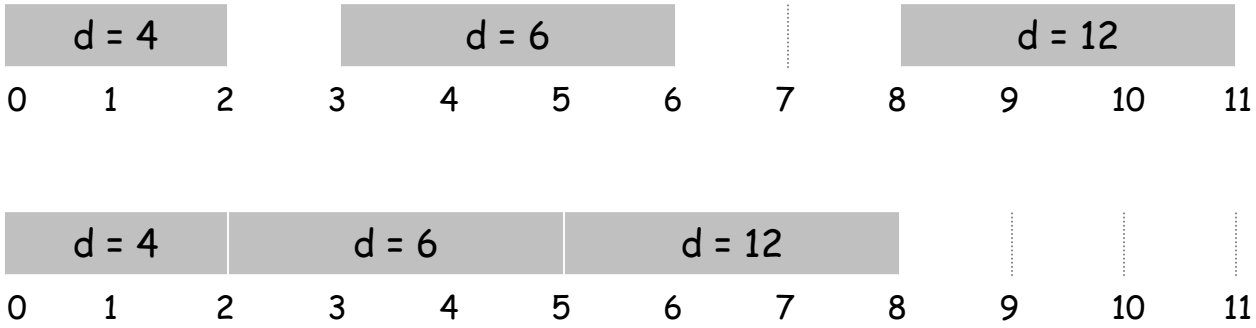
```
Sort n jobs by deadline so that d_1 ≤ d_2 ≤ … ≤ d_n

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: No Idle Time

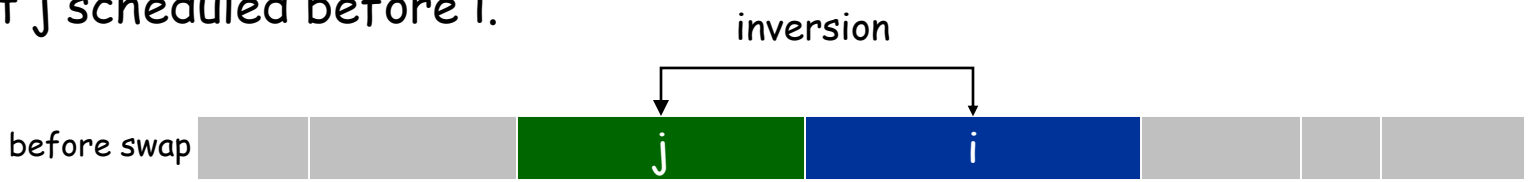
Observation. There exists an optimal schedule with no idle time.



Observation. The greedy schedule has no idle time.

Minimizing Lateness: Inversions

Def. An inversion in schedule S is a pair of jobs i and j such that: i < j but j scheduled before i.

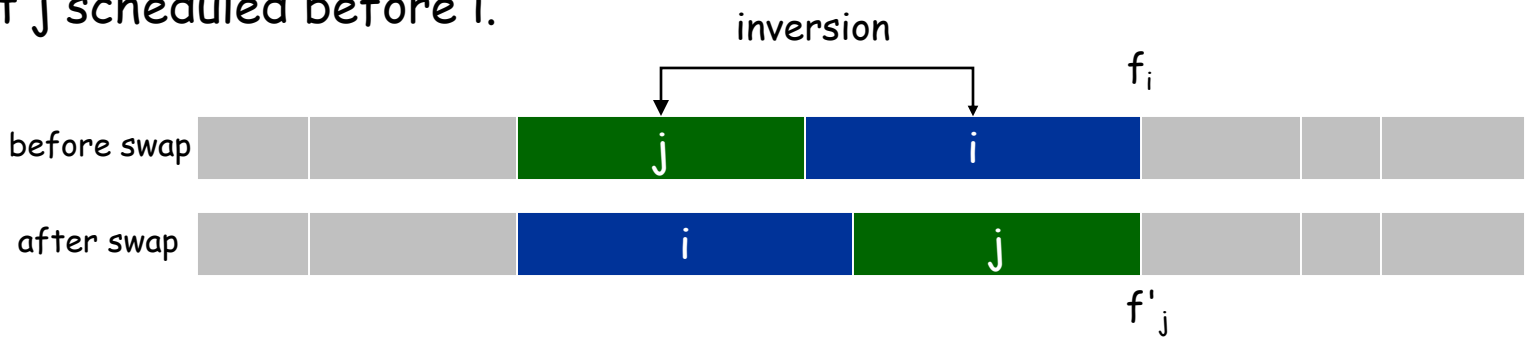


Observation. Greedy schedule has no inversions.

Observation. If a schedule (with no idle time) has an inversion, it has one with a pair of inverted jobs scheduled consecutively.

Minimizing Lateness: Inversions

Def. An inversion in schedule S is a pair of jobs i and j such that: i < j but j scheduled before i.



Claim. Swapping two adjacent, inverted jobs reduces the number of inversions by one and does not increase the max lateness.

Pf. Let ℓ be the lateness before the swap, and let ℓ' be it afterwards.

$\ell'_k = \ell_k$ for all $k \neq i, j$

$\ell'_i \leq \ell_i$

If job j is late:

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

Minimizing Lateness: Analysis of Greedy Algorithm

Theorem. Greedy schedule S is optimal.

Pf. Define S* to be an optimal schedule that has the fewest number of inversions, and let's see what happens.

Can assume S* has no idle time.

If S* has no inversions, then S = S*.

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* ▪

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.

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

4.7 Clustering

Clustering

photos, documents. micro-organisms

Clustering. Given a set U of n objects labeled $p_1, \ldots, p_n$, partition into clusters s.t. objects in different clusters are far apart.

e.g., a large number of corresponding pixels whose intensities differ over some threshold

Fundamental problem. Divide into clusters so that points in different clusters are far apart.

- Routing in mobile ad hoc networks.
- Identify patterns in gene expression.
- Document categorization for web search.
- Similarity searching in medical image databases
- Skycat: cluster 10^9 sky objects into stars, quasars, galaxies.

Clustering of Maximum Spacing

k-clustering. Divide objects into k non-empty groups.

Distance function. Assume it satisfies several natural properties.

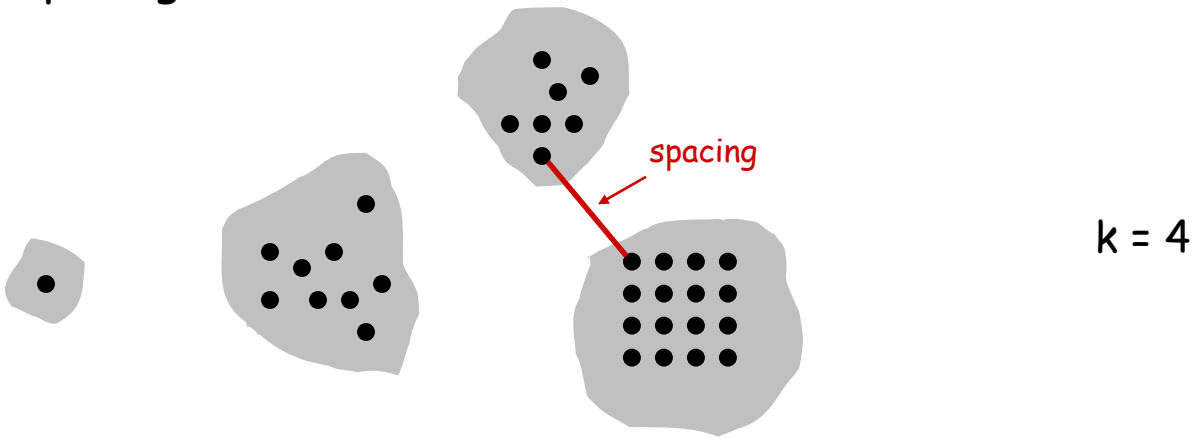
$d(p_i, p_j) = 0$ iff $p_i = p_j$ (identity of indiscernibles)

$d(p_i, p_j) \geq 0$ (nonnegativity)

$d(p_i, p_j) = d(p_j, p_i)$ (symmetry)

Spacing. Min distance between any pair of points in different clusters.

Clustering of maximum spacing. Given an integer k, find a k-clustering of maximum spacing.



Greedy Clustering Algorithm

Single-link k-clustering algorithm.

Create n clusters, one for each object.

Find the closest pair of objects such that each object is in a different cluster; add an edge between them and merge the two clusters.

Repeat n-k times until there are exactly k clusters.

Key observation. This procedure is precisely Kruskal's algorithm (except we stop when there are k connected components).

Remark. Equivalent to finding an MST and deleting the k-1 most expensive edges.

Greedy Clustering Algorithm: Analysis

Theorem. Let C^* denote the clustering $C^*_1, \ldots, C^*_k$ formed by deleting the k-1 most expensive edges of a MST. C^* is a k-clustering of max spacing.

Pf. Let C denote some other clustering $C_1, \ldots, C_k$.

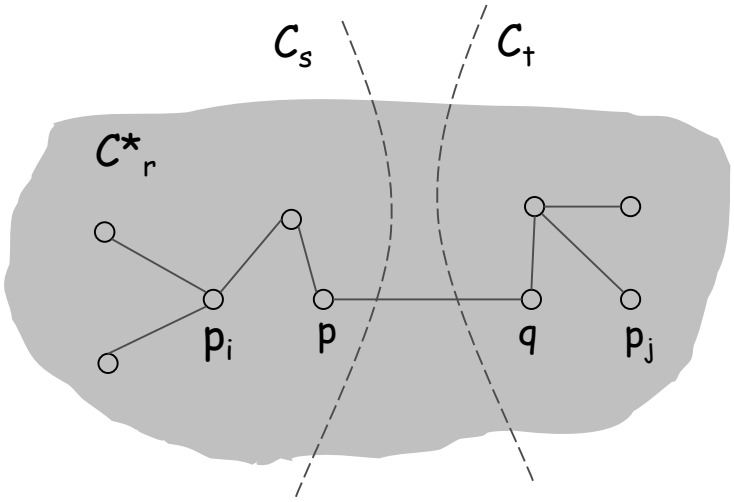
The spacing of C^* is the length d^* of the $(k-1)^{st}$ most expensive edge in MST.

Let p_i, p_j be in the same cluster in C^*, say C^*_r, but different clusters in C, say C_s and C_t.

Some edge (p, q) on p_i-p_j path in C^*_r spans two different clusters in C.

All edges on p_i-p_j path have length $\leq d^*$ since Kruskal chose them.

Spacing of C is $\leq d^*$ since p and q are in different clusters. ▪



4.3 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 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 evictions.

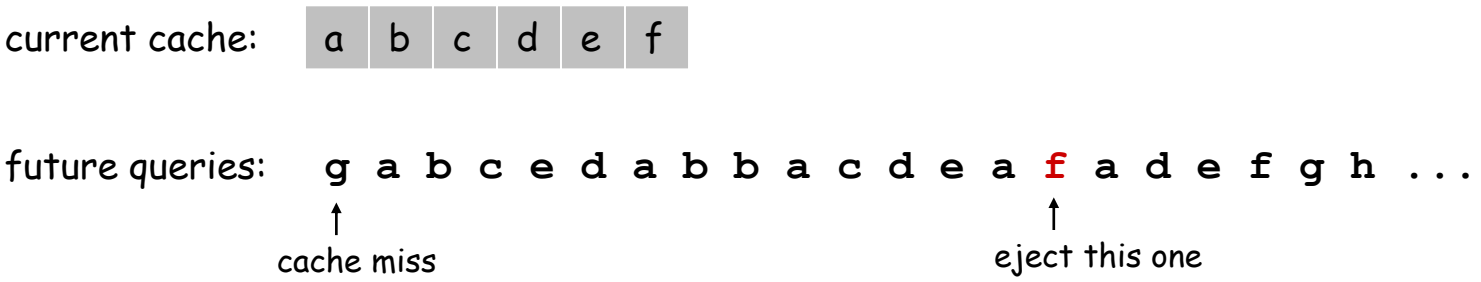
Ex: k = 2, initial cache = ab,
requests: a, b, c, b, c, a, a, b.

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
a	a	b
b	a	b

Optimal Offline Caching: Farthest-In-Future

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



Theorem. [Bellady, 1960s] FF is optimal eviction schedule.

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

Reduced Eviction Schedules

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 evictions.

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

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
a	a	c	b
a	a	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 unreduced items)

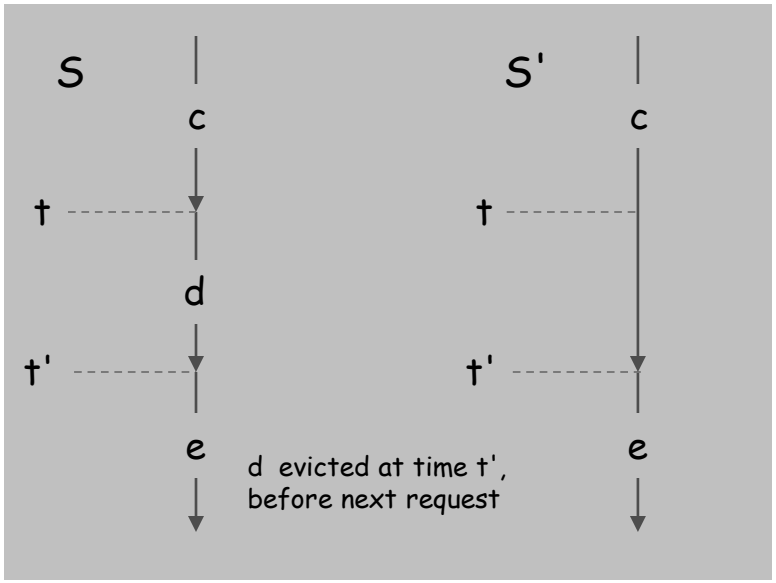
doesn't enter cache at requested time

Suppose S brings d into the cache at time t, without a request.

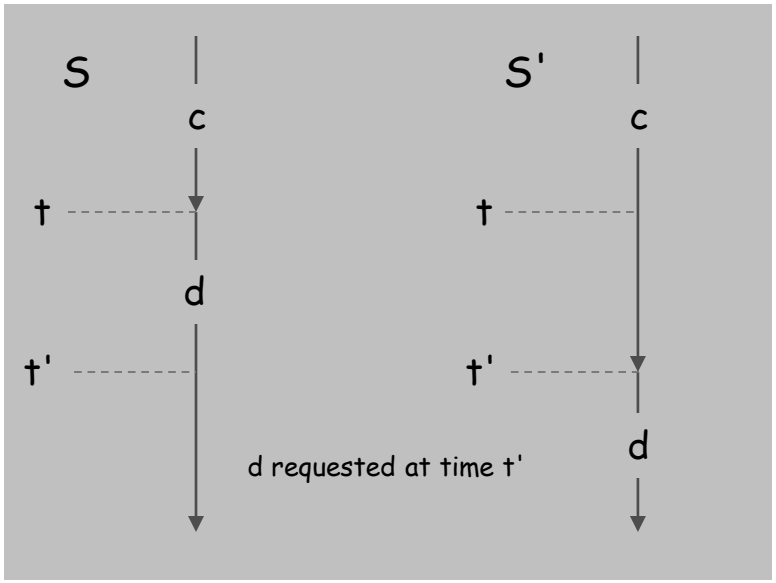
Let c be the item S evicts when it brings d into the cache.

Case 1: d evicted at time t', before next request for d.

Case 2: d requested at time t' before d is evicted. ▪



Case 1



Case 2

Farthest-In-Future: Analysis

Lemma. Let S be a reduced schedule that makes the same schedule as S_{FF} through the first j requests. Then there is a reduced schedule S' that makes the same schedule as S_{FF} through the first j+1 requests, and incurs no more evictions than S does.

Pf.

Consider $(j+1)^{st}$ request $d = d_{j+1}$.

Since S and S_{FF} have agreed up until now, they have the same cache contents before request j+1.

Case 1: (d is already in the cache). S' = S

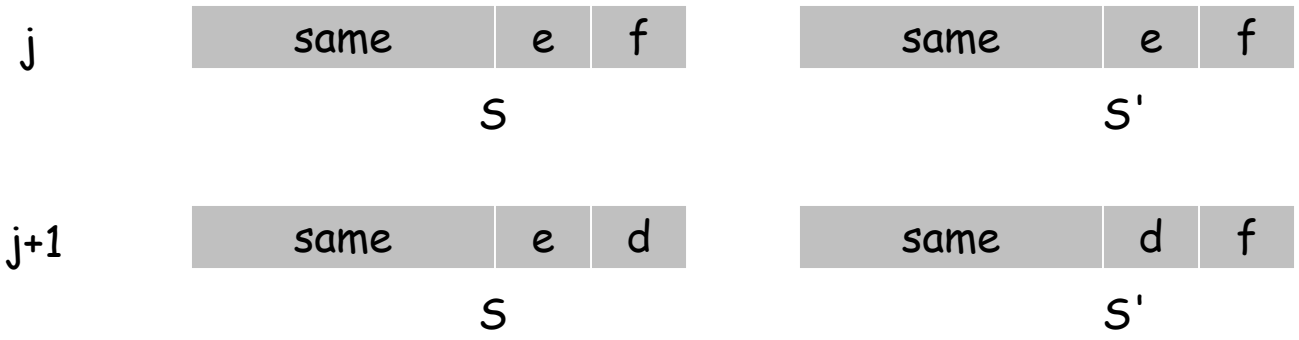
Case 2: (d is not in the cache and S and S_{FF} evict the same element). S' = S

Farthest-In-Future: Analysis

Pf. (continued)

Case 3: (d is not in the cache; S_{FF} evicts e; S evicts f ≠ e).

- begin construction of S' from S by evicting e instead of f



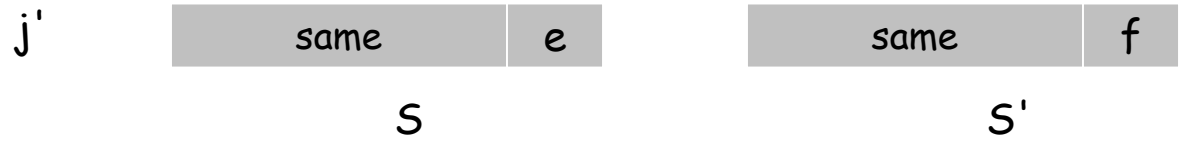
- now S' agrees with S_{FF} on first j+1 requests
- From request j+2 onward, we make S' the same as S, but this becomes impossible when e or f is involved

Farthest-In-Future: Analysis

Pf. (continued)

Let j' be the first time after j+1 that S and S' take a different action, and let g be item requested at time j'.

must involve e or f (or both)

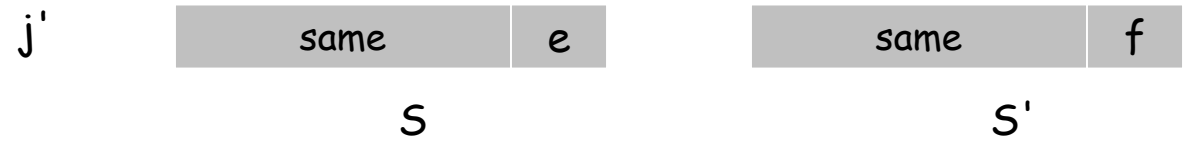


Case 3a: g = e. Can't happen with Farthest-In-Future since there must be a request for f before e.

Farthest-In-Future: Analysis

Pf. (continued)

Let j' be the first time after j+1 that S and S' take a different action, and let g be item requested at time j'.



otherwise S' would take the same action

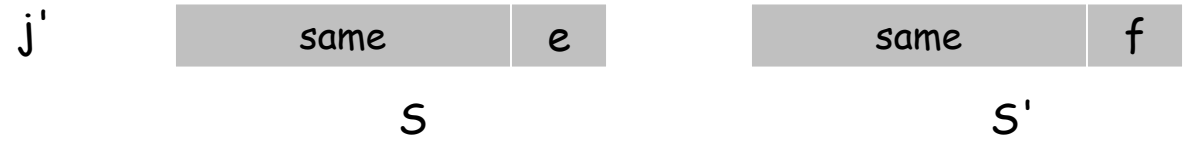
Case 3b: $g \neq e, f$. S must evict e.

Make S' evict f; now S and S' have the same cache. ▪

Farthest-In-Future: Analysis

Pf. (continued)

Let j' be the first time after j+1 that S and S' take a different action, and let g be item requested at time j'.



Case 3c: g = f. Element f can't be in cache of S, so let e' be the element that S evicts.

- if e' = e, S' accesses f from cache; now S and S' have same cache
- if e' ≠ e, S' evicts e' and brings e into the cache; now S and S' have the same cache. S' is no longer reduced, but can be transformed into a reduced schedule which
  a) agrees with S_{FF} through step j+1
  b) has no more evictions than S

Farthest-In-Future: Analysis

Theorem. FF is optimal eviction algorithm.

Pf. (by induction on number of requests j)

Base case (trivial):
There exists an optimal reduced schedule S that makes the same schedule as S_{FF} through the first 0 requests.

Inductive step (implied by the lemma):
If there exists an optimal reduced schedule S that agrees with S_{FF} through the first j requests,
then there exists an optimal reduced schedule S' that agrees with S_{FF} through the first j+1 requests

Caching Perspective

Online vs. offline algorithms.

Offline: full sequence of requests is known a priori.

Online (reality): requests are not known in advance.

Caching is among most fundamental online problems in CS.

LIFO. Evict page brought in most recently.

LRU. Evict page whose most recent access was earliest.

Theorem. FF is optimal offline eviction algorithm.

Provides basis for understanding and analyzing online algorithms.

LRU is k-competitive. [Section 13.8]

LIFO is arbitrarily bad.

Greedy Algorithms: Chapter Summary

Greedy Algorithms

Basic idea

Make the locally optimal choice at each step.

Algorithms

Interval Scheduling
- Choose the job with the earliest finish time

Scheduling to Minimize Lateness
- Choose the job with the earliest deadline

Clustering
- Single-link k-clustering

Optimal Caching
- Evict item that is requested farthest in future

Greedy Algorithms

Proof skills

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.

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