Greedy Algorithms

Greedy Algorithms

- A greedy algorithm is an algorithm design paradigm which finds the solution for an optimization problem.
- A greedy algorithm works in phases. At each phase:
  - » You take the best you can get right now, without regard for future consequences
  - » You hope that by choosing a *local* optimum at each step, you will end up at a *global* optimum

Optimization problems

- An optimization problem is one in which you want to find, not just *a* solution, but the *best* solution.
- In an optimization problem, one or more objectives are given as Objective functions along with a set of constraints.
- $minimize\ f(x)$

  $$\text{subject to. } \mathrm{g}(x) \leq k$$
  $$\mathrm{h}(x) = l$$

- A "greedy algorithm" sometimes works well for optimization problems
- A greedy algorithm does not guarantees optimal solution for an optimization problem.

Example: Counting money

- Suppose you want to count out a certain amount of money, using the fewest possible bills and coins
- A greedy algorithm would do this would be: At each step, take the largest possible bill or coin that does not overshoot
  - » Example: To make $6.39, you can choose:
    - a $5 bill
    - a $1 bill, to make $6
    - a 25¢ coin, to make $6.25
    - A 10¢ coin, to make $6.35
    - four 1¢ coins, to make $6.39
- For US money, the greedy algorithm always gives the optimum solution

A failure of the greedy algorithm

- In some (fictional) monetary system, “krons” come in 1 kron, 7 kron, and 10 kron coins
- Using a greedy algorithm to count out 15 krons, you would get
  - » A 10 kron piece
  - » Five 1 kron pieces, for a total of 15 krons
  - » This requires six coins
- A better solution would be to use two 7 kron pieces and one 1 kron piece
  - » This only requires three coins
- The greedy algorithm results in a solution, but not in an optimal solution

Optimal Substructure

- The problems which can be solved by Greedy algorithms efficiently need to satisfy two properties:

    1. Optimal substructure
    2. Greedy-choice Property

Optimal substructure: Problems must exhibit optimal substructure (like Dynamic programming).

  » A problem has optimal substructure, if an optimal solution to the problem contains within it the solutions to the sub problems.

  » We have to identify the optimal substructure of a problem.

Greedy-choice Property

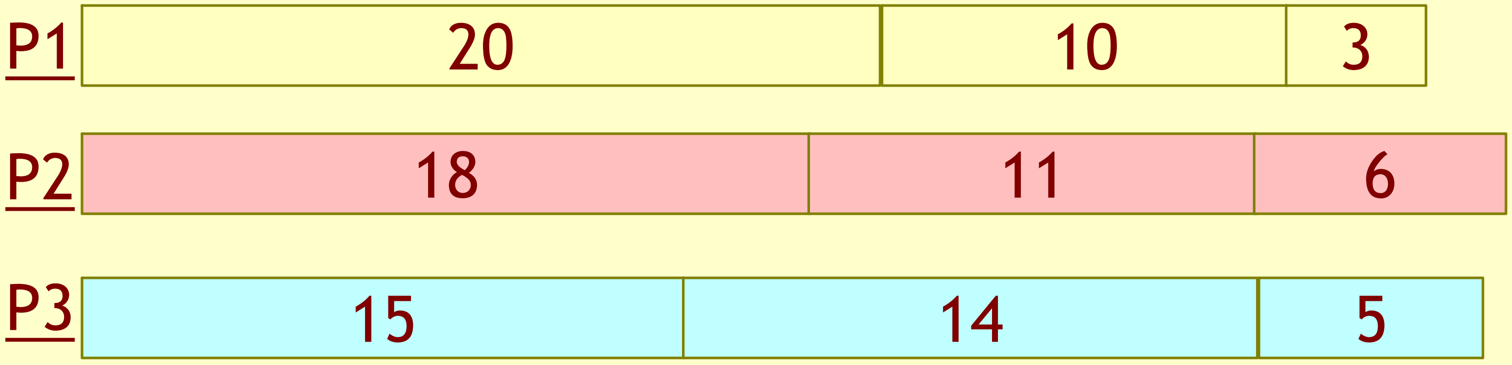
- Problems also exhibit the **greedy-choice** property.
  - » When we have a choice to make, make the one that looks best *right now*.
  - » Make a **locally optimal choice** in hope of getting a **globally optimal solution**.
  - » The greedy choice at any stage may depends on the choices made so far but never on the future choices or all solutions to the sub problems.
  - » Gradually make greedy choices, to reduce the problem size.
  - » Never reconsiders the solution we obtained so far.
  - » We have to prove the greedy choice property.

Application

1. Shortest path
2. MST
3. Huffman code
4. Interval Scheduling
5. Optimal Caching

A scheduling problem

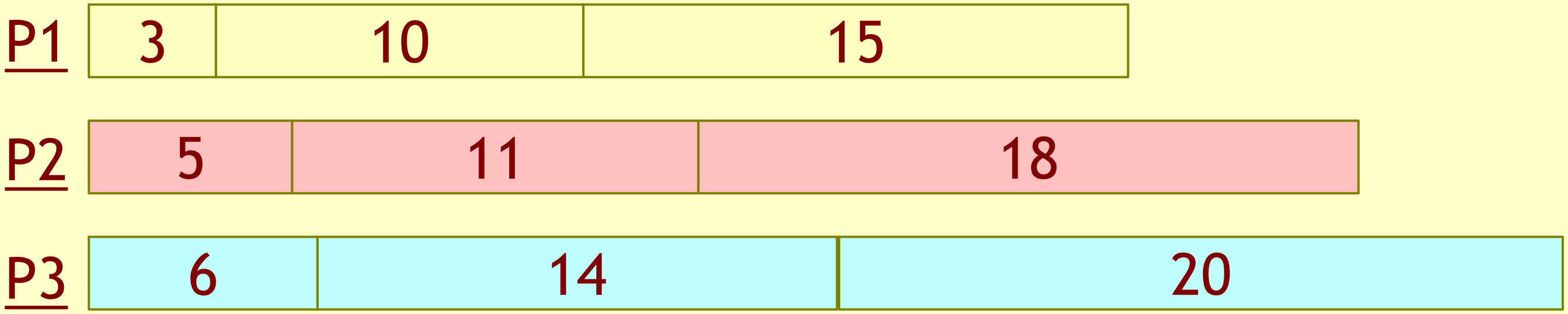
- You have to run nine jobs, with running times of 3, 5, 6, 10, 11, 14, 15, 18, and 20 minutes
- You have three processors on which you can run these jobs
- You decide to do the longest-running jobs first, on whatever processor is available



- Time to completion: 18 + 11 + 6 = 35 minutes
- This solution isn't bad, but we might be able to do better

Another approach

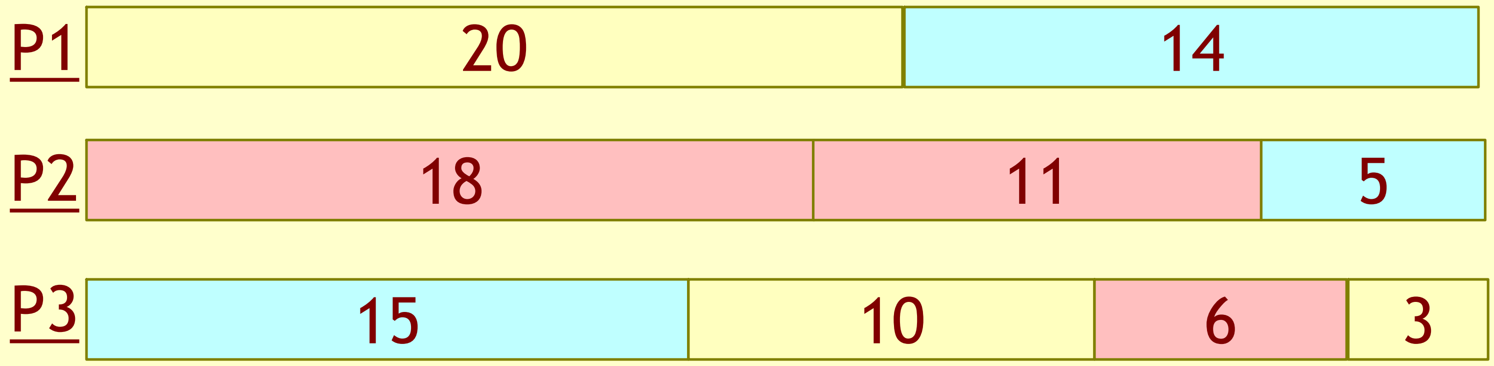
- What would be the result if you ran the *shortest* job first?
- Again, the running times are 3, 5, 6, 10, 11, 14, 15, 18, and 20 minutes



- That wasn't such a good idea; time to completion is now 6 + 14 + 20 = 40 minutes
- Note, however, that the greedy algorithm itself is fast
  - » All we had to do at each stage was pick the minimum or maximum

An optimum solution

- Better solutions do exist:

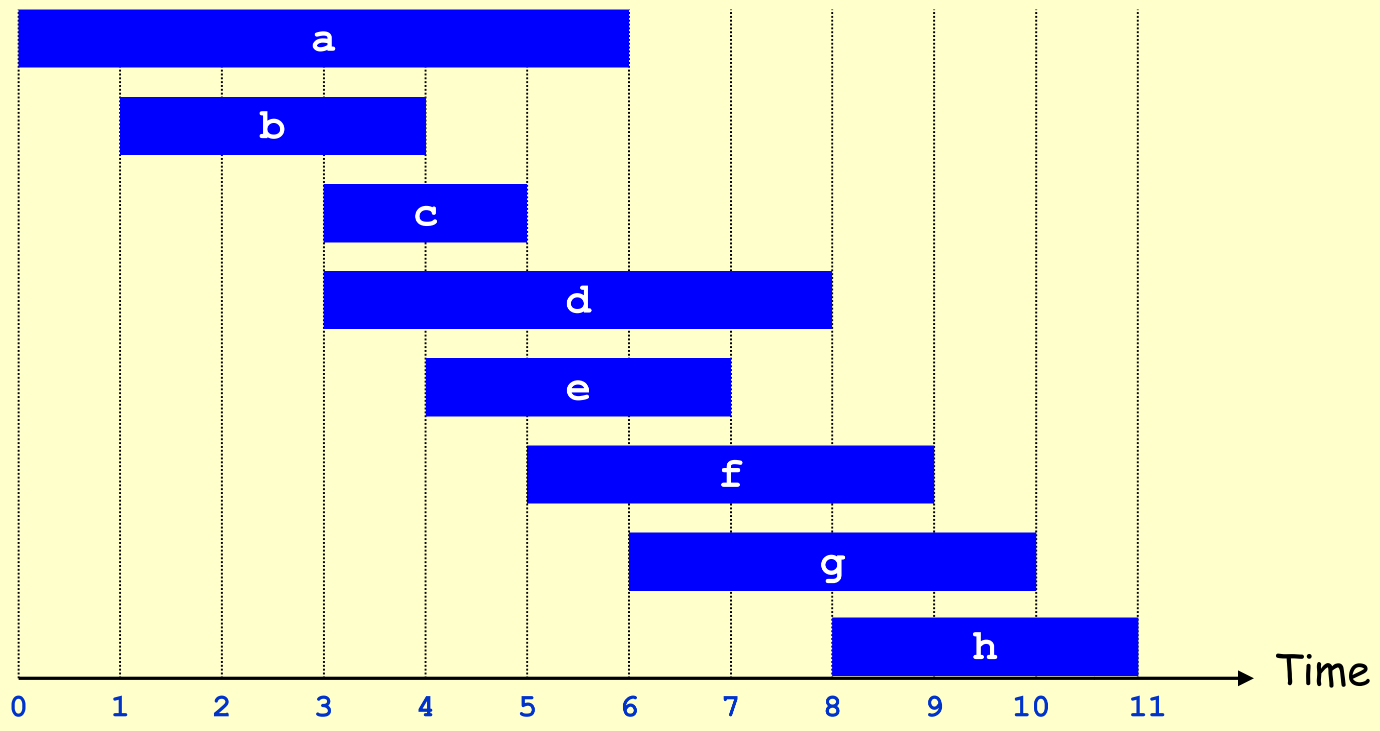


- This solution is clearly optimal
- There may be other optimal solutions
- How do we find such a solution?
  - One way: Try all possible assignments of jobs to processors
  - Unfortunately, this approach can take exponential time

Interval Scheduling

Interval Scheduling

- 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.

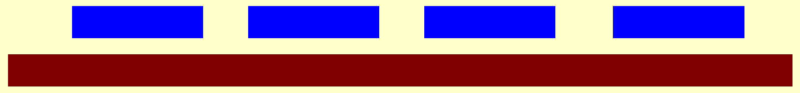


Interval Scheduling: Greedy Algorithm

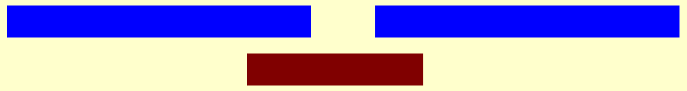
- Greedy templates: Consider jobs in some natural order. Take each job provided it's compatible with the ones already taken.
  - » [Earliest start time] Consider jobs in ascending order of s_j.
  - » [Earliest finish time] Consider jobs in ascending order of f_j.
  - » [Shortest interval] Consider jobs in ascending order of $f_j - s_j$.
  - » [Fewest conflicts] For each job j, count the number of conflicting jobs c_j. Schedule in ascending order of c_j.

Interval Scheduling: Greedy Algorithm

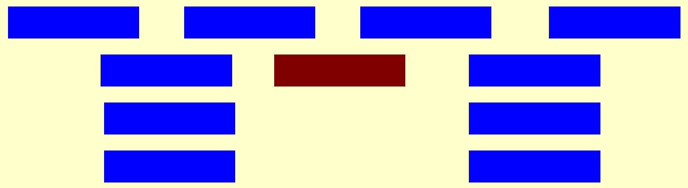
- Greedy template: Consider jobs in some natural order. Take each job provided it's compatible with the ones already taken.



Counter example for earliest start time



Counter example for shortest interval



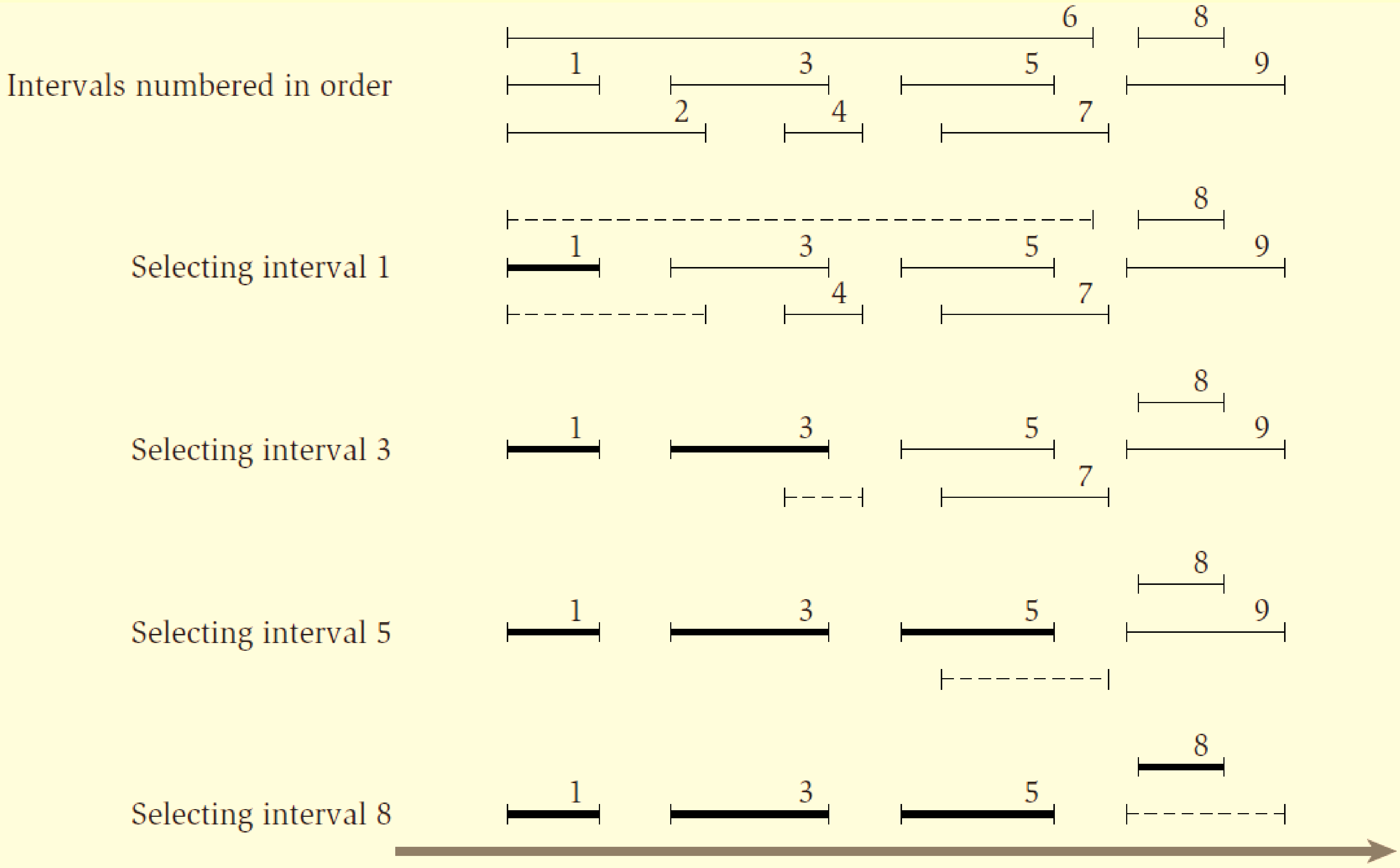
Counter example for fewest conflicts

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 $f_1 \le f_2 \le ... \le f_i \le ... \le f_n$.
$R \leftarrow$ all requests $i$
$A \leftarrow \emptyset$
**while** $R \neq \emptyset$ **do**
        Choose a request $i \in R$ that has the smallest finishing time
        $A \leftarrow A \cup \{i\}$
        Delete all requests from R that are not compatible with request $i$
**endWhile**
**return** A
```

Example



Intervals numbered in order
6
8
1
3
5
9
2
4
7
Selecting interval 1
Selecting interval 3
Selecting interval 5
Selecting interval 8

Interval Scheduling: Greedy Algorithm

- Running time = O(n log n) sorting time of finishing time of n jobs.
  - » Remember job j* that was added last to A.
  - » Job j is compatible with A if $s_j \geq f_{j*}$.
  - » O(n) time to go through the sorted list of n jobs in the while loop

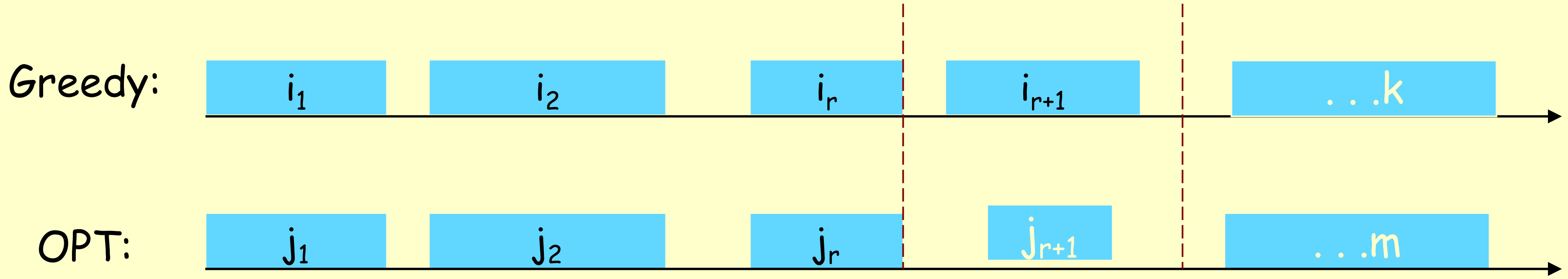
Interval Scheduling: Analysis

» Let i_1, i_2, ... i_k denote set of jobs selected by greedy.

» Let j_1, j_2, ... j_m denote set of jobs in the optimal.

» No. of jobs in greedy = k

» No. of jobs in greedy = m

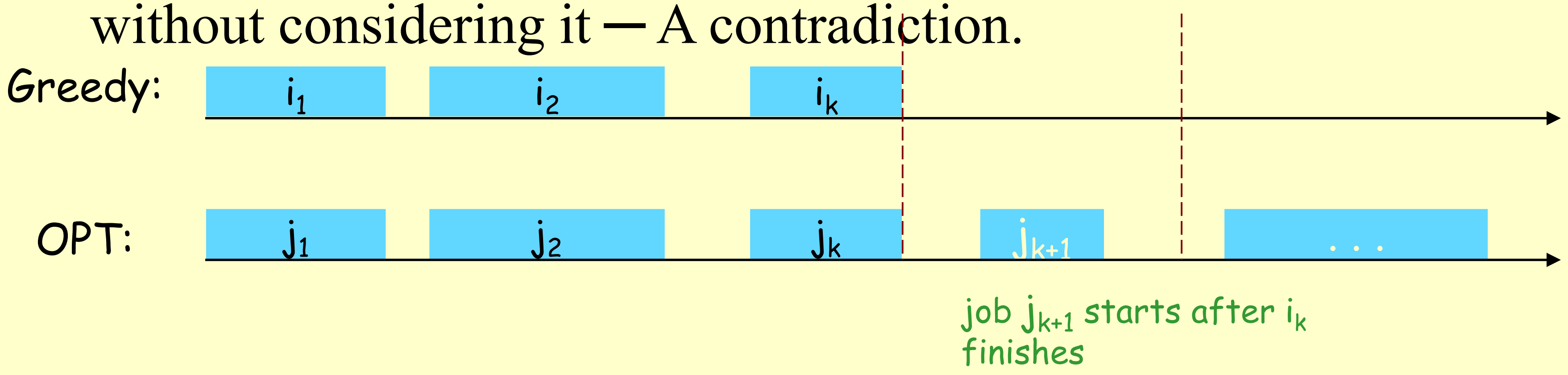


Interval Scheduling: Analysis

- Theorem. For all indices $r \leq k$ *we have* $f(i_r) \leq f(j_r)$.
- Pf. (by Induction)
- For r =1, the statement is true as the greedy solution starts by selecting the request i_1 with minimum finish time.
- Inductive Hypothesis: Assume that $f(i_{r-1}) \leq f(j_{r-1})$.
- Inductive step: In r_{th} iteration, the greedy selects the available job with minimum finishing time.
- So, clearly $f(i_r) \leq f(j_r)$.

Interval Scheduling: Analysis

- Theorem. Greedy algorithm is optimal.
- Pf. (by contradiction)
  - » Assume greedy is not optimal.
  - » So, the OPT schedule has more jobs than the Greedy. i.e. $m > k$
  - » When the last job i_k is scheduled in Greedy, j_k is scheduled in OPT and we have $f(i_k) \leq f(j_k)$
  - » The next job in OPT is j_{k+1} and $s(j_{k+1}) \geq f(j_k)$.
  - » So, $s(j_{k+1}) \geq f(i_k)$. i.e. At time $f(i_k)$, one mutually compatible job is available in R, but still Greedy has finished its execution without considering it ─ A contradiction.

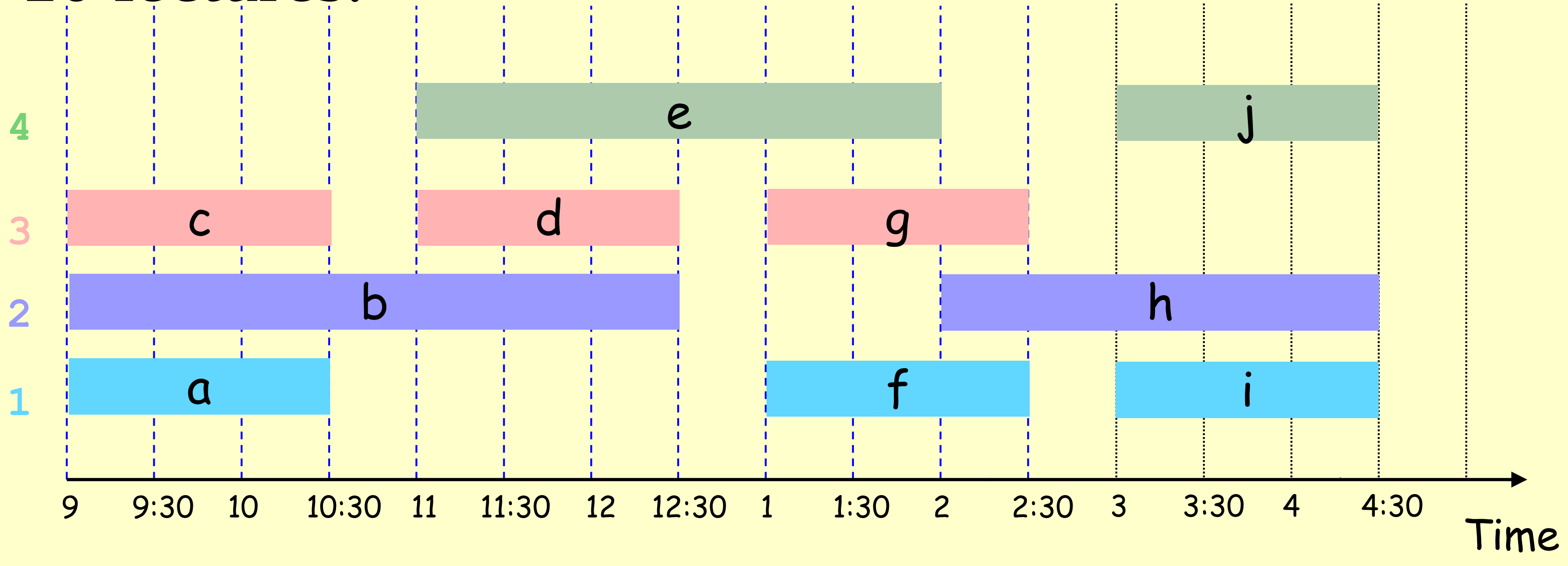


Interval Partitioning

- Interval scheduling so far is using a single resource.
- Extension: What if we have multiple resources for scheduling tasks?
- Assumption: all resources are equivalent

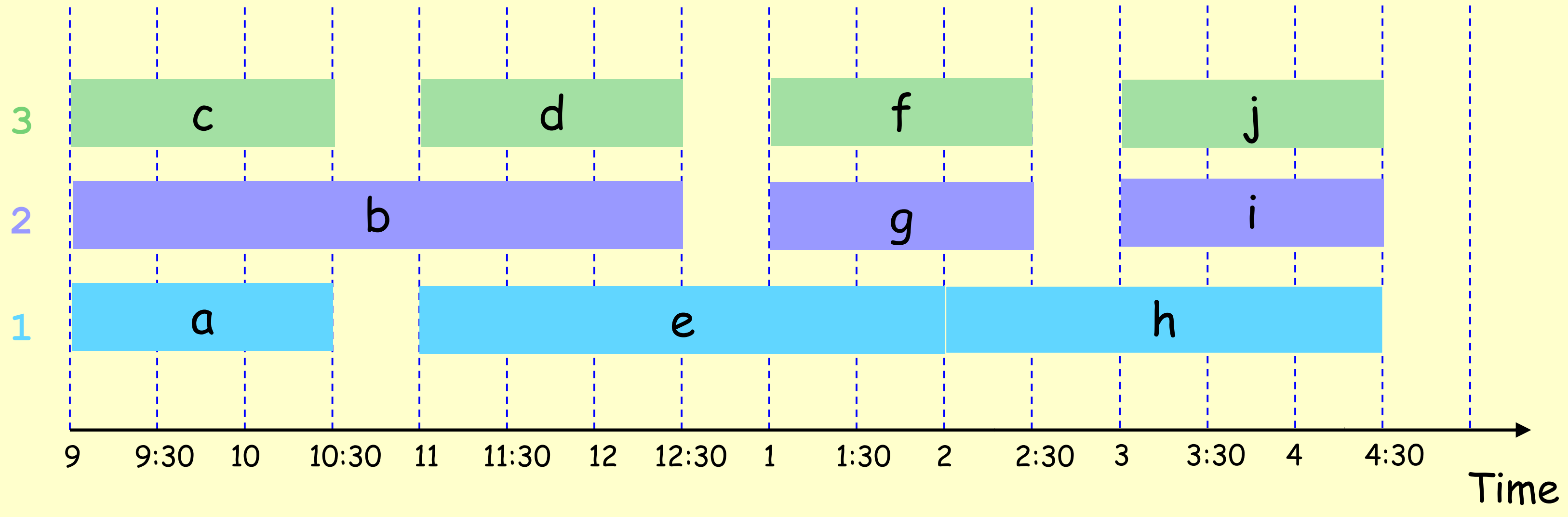
Interval Partitioning

- 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 lectures occur at the same time in the same room.
- Ex: This schedule uses 4 classrooms to schedule 10 lectures.



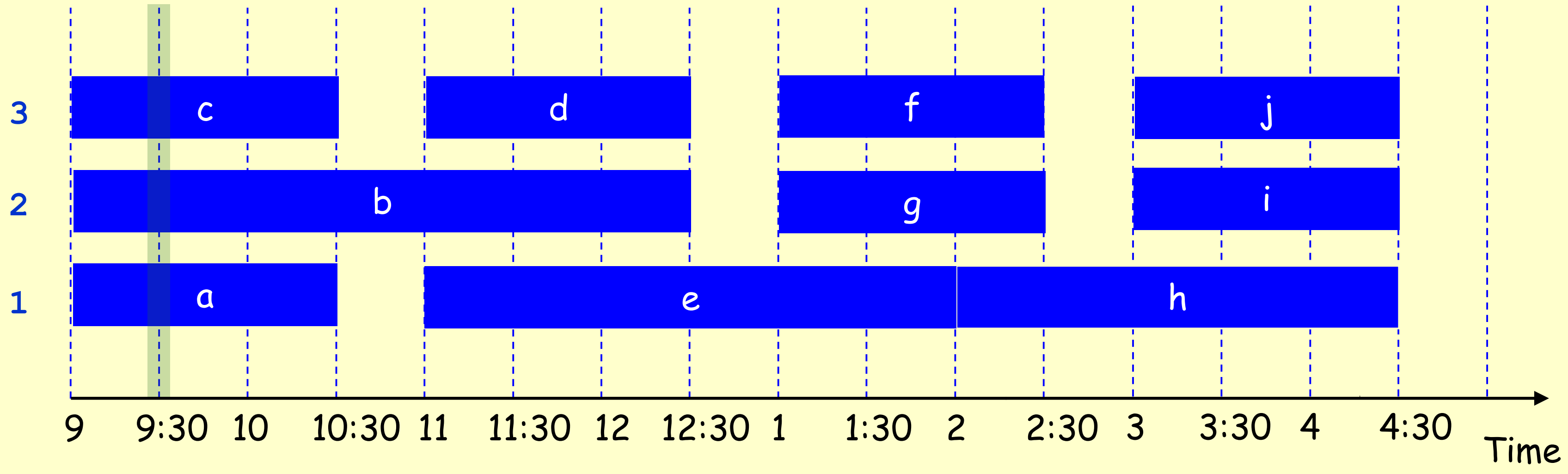
Interval Partitioning

- 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 only 3.



Interval Partitioning: Lower Bound on Optimal Solution

- The depth of a set of open intervals is the maximum number of concurrent intervals at any given time.
- Number of classrooms needed $\geq$ depth.
- Ex: Depth of schedule below = 3 $\Rightarrow$ schedule below is optimal.



Interval Partitioning: Greedy Algorithm

- Greedy approach:
  - » Consider lectures in increasing order of start time.
  - » Assign lecture to any compatible classroom.
  - » Each resource is represented by a label {1, 2,…, d}
  - » We assign a label to each interval
  - » Assignment has the property that overlapping intervals are labeled with different numbers.
- Implementation. O(n log n).

Interval Partitioning: Greedy Algorithm

```
Sort the intervals by their start times, breaking ties arbitrarily
Let $I_1, I_2, \ldots, I_n$ denote the intervals in this order
For j = 1, 2, 3, . . . , n
    For each interval $I_i$ that precedes $I_j$ in sorted order and overlaps it
        Exclude the label of $I_i$ from consideration for $I_j$
    Endfor
    If there is any label from {1, 2, . . . , d} that has not been
    excluded then
        Assign a nonexcluded label to $I_j$
    Else
        Leave $I_j$ unlabeled
Endif
Endfor
```

Interval Partitioning: Greedy Analysis

- Observation. Greedy algorithm never schedules two incompatible lectures in the same classroom.
- Theorem. Greedy algorithm is optimal.
- Pf.
  - » First let's prove that no interval ends up unlabeled.
    - Consider one of the intervals Ij, and suppose there are t intervals earlier in the sorted order that overlap it.
    - These t intervals, together with Ij, form a set of t + 1 intervals that all pass over a common point on the time-line (namely, the start time of Ij), and so $t + 1 \leq d$.
    - Thus $t \leq d - 1$.
    - It follows that at least one of the d labels is not excluded by this set of t intervals, and so there is a label that can be assigned to Ij.▪

Interval Partitioning: Greedy Analysis

- Observation. Greedy algorithm never schedules two incompatible lectures in the same classroom.
- Theorem. Greedy algorithm is optimal.
- Pf.
  - » Next we prove that no two overlapping intervals are assigned the same label.
    - Consider any two intervals I and I’ that overlap, and suppose I precedes I’ in the sorted order.
    - Then when I’ is considered by the algorithm, I is in the set of intervals whose labels are excluded from consideration
    - Consequently, the algorithm will not assign to I’ the label that it used for I.

Scheduling to Minimize Lateness

- What if all jobs do not have predefined fixed starting time?
- Use a single resource to complete all jobs one after another
- Each job has a due time

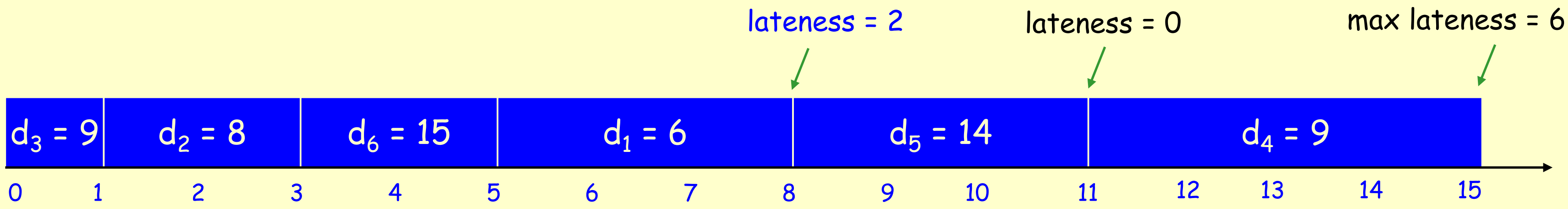
Scheduling to Minimizing Lateness

- 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 $L = \max \ell_j$.

Scheduling to Minimizing Lateness

- Ex:

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



- Job scheduled order: 3, 2, 6, 1, 5, 4

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	5	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

Minimizing Lateness: Greedy Algorithm

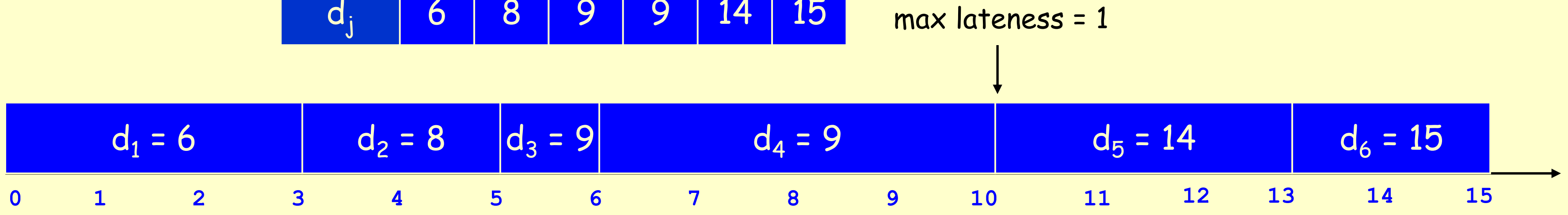
- Greedy algorithm. Earliest deadline first.

```
Sort n jobs by deadline so that $d_1 \leq d_2 \leq \ldots \leq d_n$
$t \leftarrow 0$
for j = 1 to n
   Assign job j to interval $[t, t + t_j]$
   $s_j \leftarrow t$, $f_j \leftarrow t + t_j$
   $t \leftarrow t + t_j$
output intervals $[s_j, f_j]$
```

Minimizing Lateness: Greedy Algorithm

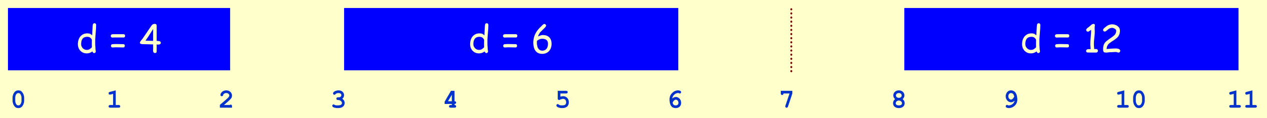
- Example:

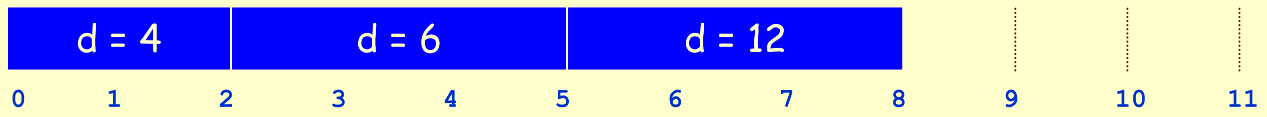
	1	2	3	4	5	6
t_j	3	2	1	4	3	2
d_j	6	8	9	9	14	15



Minimizing Lateness: No Idle Time

- The times period for which the resource is not occupied yet there are jobs left is called *idle time:* there is work to be done, yet for some reason the machine is sitting idle.
- There exists an optimal schedule with no idle time.

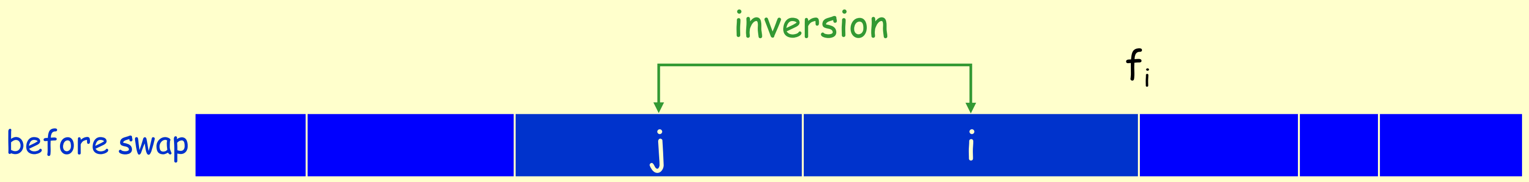




- Observation. The greedy schedule has no idle time.

Minimizing Lateness: Inversions

- Def. Given a schedule S, an inversion is a pair of jobs i and j such that:
  $d_i < d_j$ but j scheduled before i.



[as before, we assume jobs are numbered so that $d_1 \leq d_2 \leq \ldots \leq d_n$]

- Observation. Greedy schedule has no inversions and no idle time.
- Observation. If a schedule (with no idle time) has an inversion, it has at least one inversion with a pair of inverted jobs scheduled consecutively.

Minimizing Lateness

Claim. All schedules with no inversion and no idle time have the same maximum lateness.

Pf.

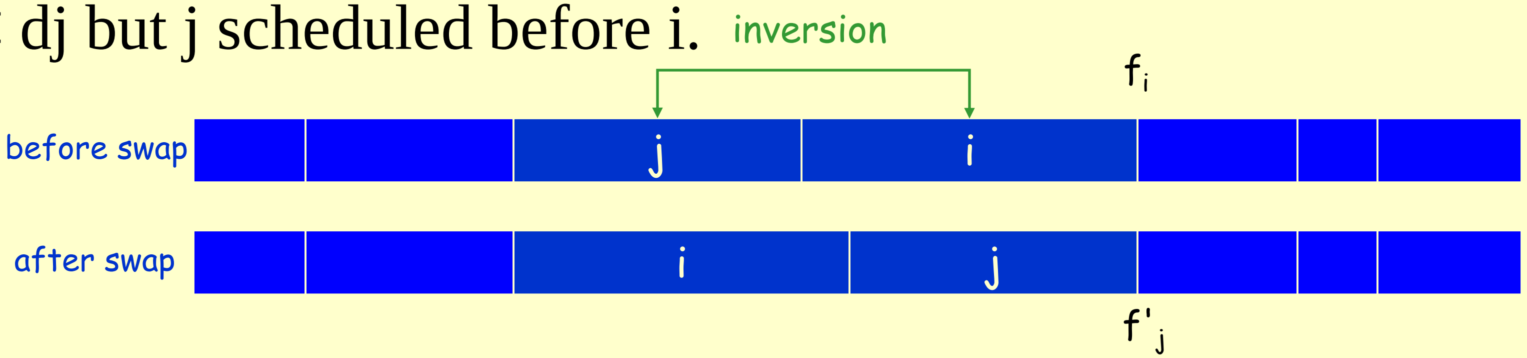
- Two such schedules must differ in the order in which jobs with identical deadlines are scheduled.

- Jobs with this same deadline are scheduled consecutively.

- The last of these jobs have the largest lateness, not depend on the order of these jobs.

Minimizing Lateness: Inversions

- Def. Given a schedule S, an inversion is a pair of jobs i and j such that: di < dj but j scheduled before i.



- Claim. Swapping two consecutive, 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 $k \neq i, j$
  - » $\ell'_i \leq \ell_i$ for i
  - » For j, 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 no idle time and may or may not have inversions.
  - » If S* has no inversions, then S = S* (greedy schedule result S is also with no inversions and idle time and we know, “All schedules with no inversion and no idle time have the same maximum lateness.”)
  - » If S* has an inversion, we keep on swapping the consecutive inverted jobs and it does not increase the maximum lateness and strictly decreases the number of inversions. Ultimately, we will reach at a scheduled S’ which has no idle time and no inversion and has the same lateness as S*.
  - » As “all schedules with no inversion and no idle time have the same maximum lateness”, the lateness of S, S’ and S* will be the same.

Greedy Analysis Strategies

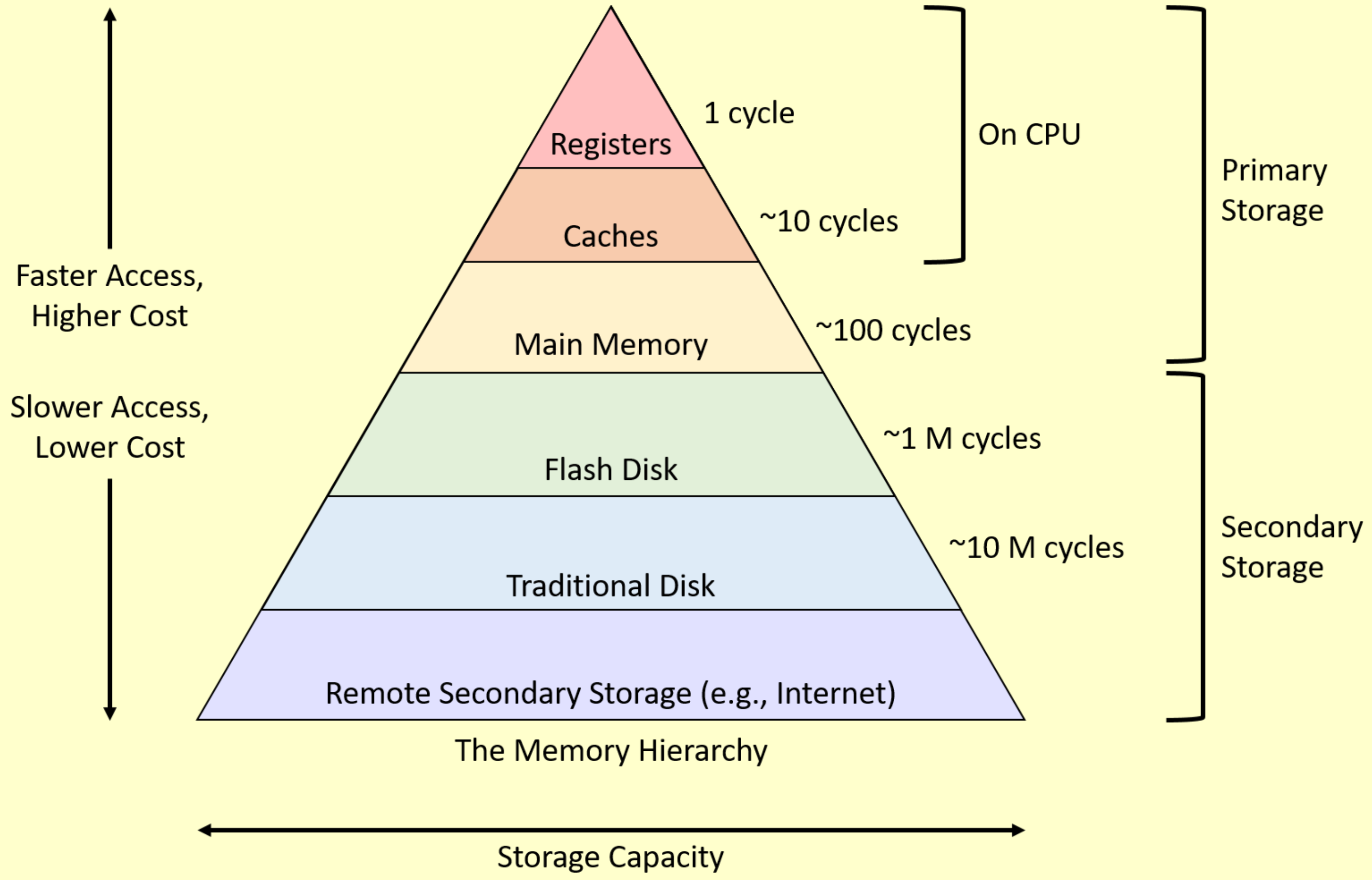
- Greedy algorithm stays ahead. Show that after each step of the greedy algorithm, its solution is at least as good as any other algorithm's. (eg. Interval scheduling)
- Structural. Discover a simple "structural" bound asserting that every possible solution must have a certain value. Then show that your algorithm always achieves this bound.
- Exchange argument. Gradually transform any solution to the one found by the greedy algorithm without hurting its quality. (Minimize Lateness)
- Other greedy algorithms. Kruskal, Prim, Dijkstra, Huffman, …

Greedy Analysis Strategies

Problem name	Problem description	Greedy algorithm	Idea of proof	Run-time
Interval scheduling	Choose as many non-overlapping intervals as possible.	Earliest finishing time first	Greedy algorithm stays ahead (induction)	O(n log n)
Interval partitioning	Divide intervals over least number of resources.	Earliest starting time first	Structural bound	O(n log n)
Minimize lateness	Schedule to minimize the lateness regarding the deadline.	Earliest deadline first	Exchange argument (contradiction)	O(n log n)

Optimal Caching

Cache



The Memory Hierarchy

Cache Memory

- A cache is a small, fast storage device on a CPU that holds limited subsets of data from main memory.
- Cache usually stores the frequently accessed data.
- When a piece of data is needed by the processor, it first checks for a corresponding entry in the cache:
  - » If the data is found on the cache, a **cache hit** has occurs and data is read from cache
  - » If the data is not found on the cache, a **cache miss** occurs and the processor searches the data in main memory. Then the cache allocates a new entry and copies in data from main memory, then the request is fulfilled from the contents of the cache.

Cache Memory

- To achieve optimal caching behavior, a cache maintenance algorithm determines what to keep in the cache and what to evict from the cache when new data needs to be brought in.
- Caches face several important design questions, such as:
  - » *Which* subsets of a program's memory should the cache hold?
  - » *When* should the cache copy a subset of a program's data from main memory to the cache or vice versa?
  - » *Which* data item to evict from the cache when it is full to make room for a new piece of incoming data.

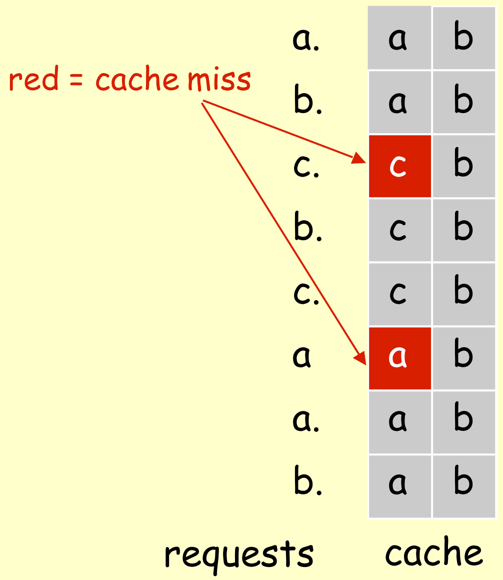
Optimal Caching Problem

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.


red = cache miss
requests
cache

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 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 (under g)

↑ eject this one (under f)

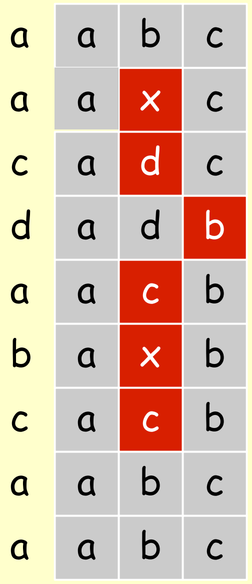
Theorem. [Bellady, 1960s] FF is optimal eviction schedule.

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

Reduced Eviction Schedules

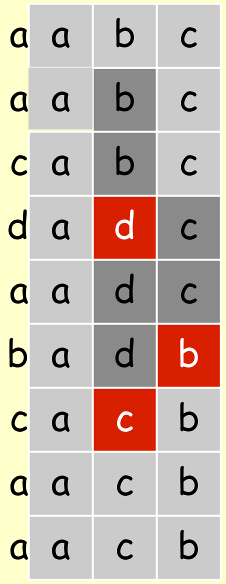
Def. A reduced schedule is a schedule that only inserts an item into the cache in a step in which that item is requested.

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



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

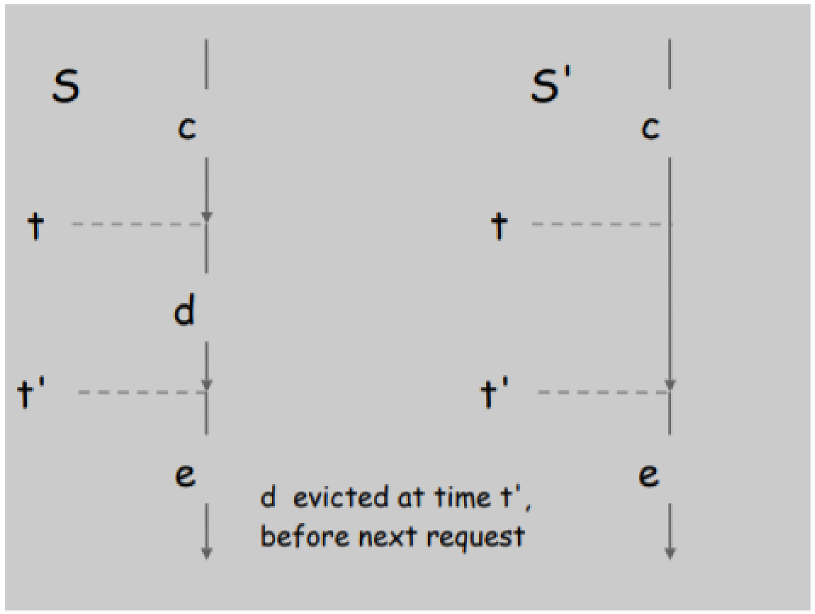
a reduced schedule

Reduced Eviction Schedules

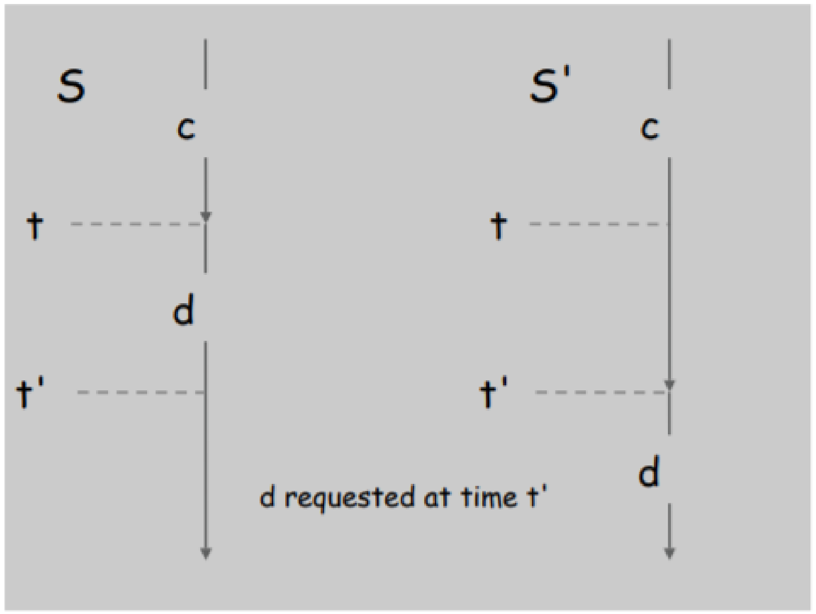
Claim. Given any unreduced schedule S, can transform it into a reduced schedule S' with no more cache misses.

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

Theorem. FF is optimal eviction algorithm.
Pf. (by induction on number or requests j)

Invariant: There exists an optimal reduced schedule that makes the same eviction schedule as S_{FF} through the first j+1 requests.

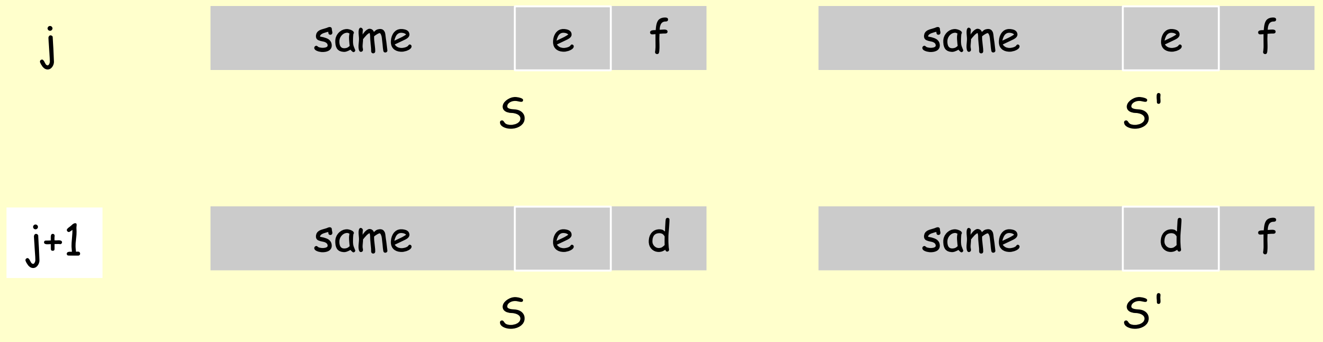
Let S be reduced schedule that satisfies invariant through j requests. We produce S' that satisfies invariant after j+1 requests.

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

Farthest-In-Future: Analysis

Pf. (continued)

- Case 3: (d is not in the cache; S_{FF} evicts e; S evicts f $\neq$ e).
  - begin construction of S' from S by evicting e instead of f

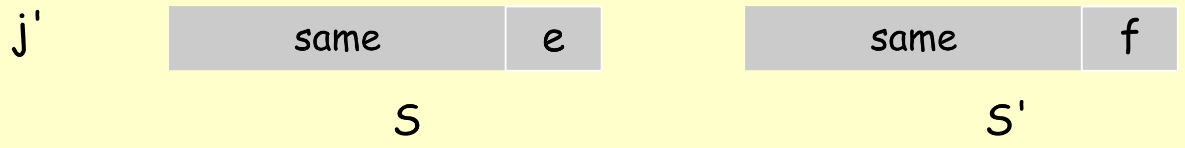


- now S' agrees with S_{FF} on first j+1 requests; we show that having element f in cache is no worse than having element e

Farthest-In-Future: Analysis

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.

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

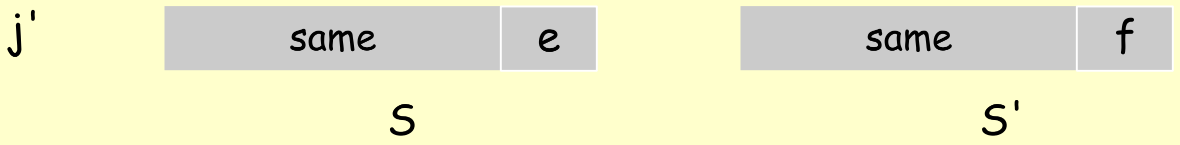
- if e' = e, S' accesses f from cache; now S and S' have same cache
- if e' ≠ e, S' evicts e' and brings e into the cache; now S and S' have the same cache

Note: S' is no longer reduced, but can be transformed into a reduced schedule that agrees with S_{FF} through step j+1

Farthest-In-Future: Analysis

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'.

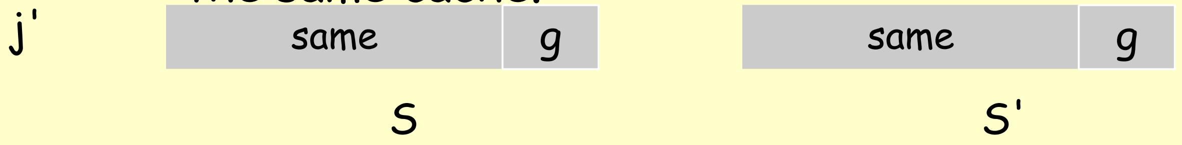
must involve e or f (or both)



otherwise S' would take the same action

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

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



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.

↑ FF with direction of time reversed!

Theorem. FF is optimal offline eviction algorithm.

- Provides basis for understanding and analyzing online algorithms.
- LRU is k-competitive.
- LIFO is arbitrarily bad.