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

- *Gordon Gecko (Michael Douglas)*

Analysis of Algorithms

Piyush Kumar

(Lecture 5: Greedy Algorithms)

Welcome to 4531

Source: K. Wayne, …

Greedy Algorithms

- Optimization problem: Min/Max an objective.
  - Minimize the total length of a spanning tree.
  - Minimize the size of a file using compression
  - … (The mother of all problems)
- Greedy Algorithm
  - Attempt to do best at each step without consideration of future consideration
    - For some problems, Locally optimal choice leads to global opt.
    - Follows *"Greed is good"* philosophy
    - Requires *"Optimal Substructure"*
- What examples have we already seen?

Greedy Algorithms

- For some problems, "Greed is good" works.
- For some, it finds a good solution which is not global opt
  - Heuristics
  - Approximation Algorithms
- For some, it can do very bad.

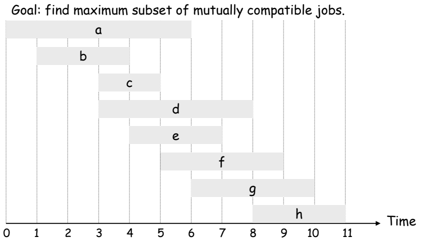
Problem of Change

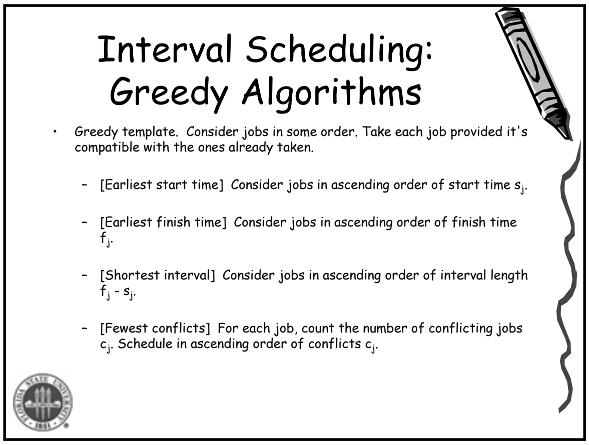
- Vending machine has quarters, nickels, pennies and dimes. Needs to return N cents change.
- Wanted: An algorithm to return the N cents in minimum number of coins.
- What do we do?

4.1 Interval Scheduling

Interval Scheduling

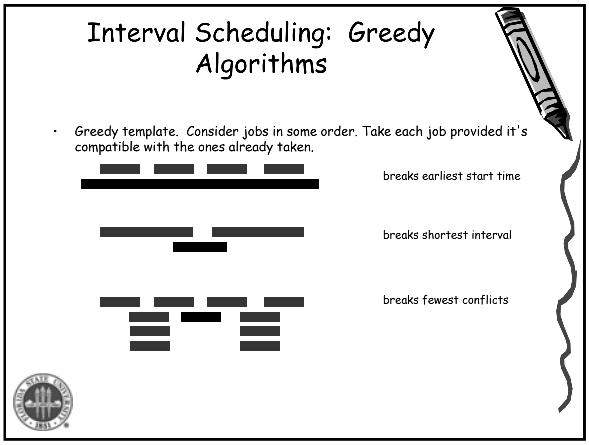
- Interval scheduling.
  - Job j starts at s_j and finishes at f_j.
  - Two jobs compatible if they don't overlap.
  - Goal: find maximum subset of mutually compatible jobs.





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.



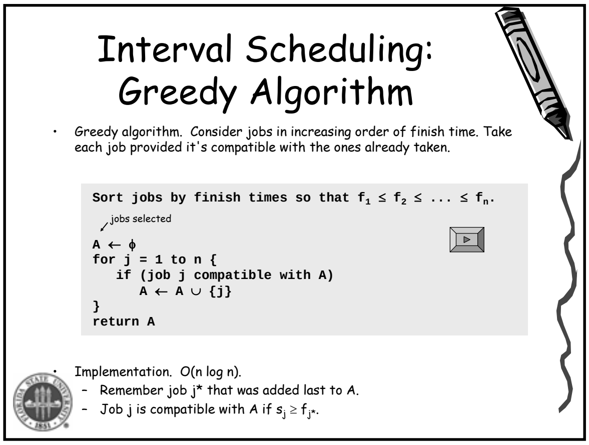
Interval Scheduling: Greedy Algorithms

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

breaks earliest start time

breaks shortest interval

breaks fewest conflicts

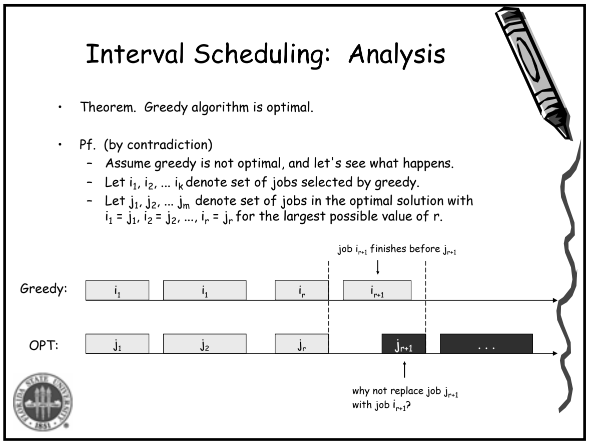


Interval Scheduling: Greedy Algorithm

- 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 f₁ ≤ f₂ ≤ ... ≤ fₙ.
  jobs selected
A ← φ
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 if $s_j \geq f_{j^*}$.

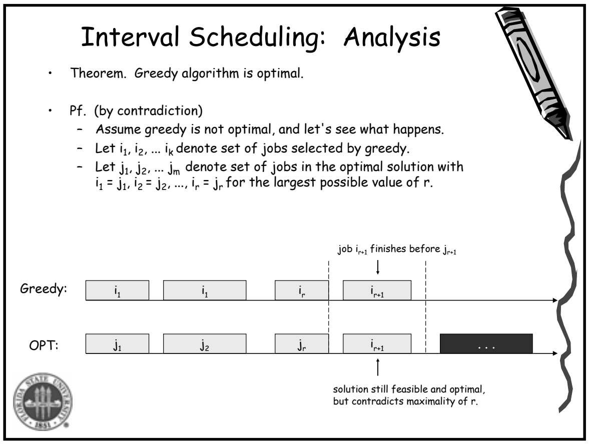


Interval Scheduling: Analysis

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

job i_{r+1} finishes before j_{r+1}

why not replace job j_{r+1} with job i_{r+1}?

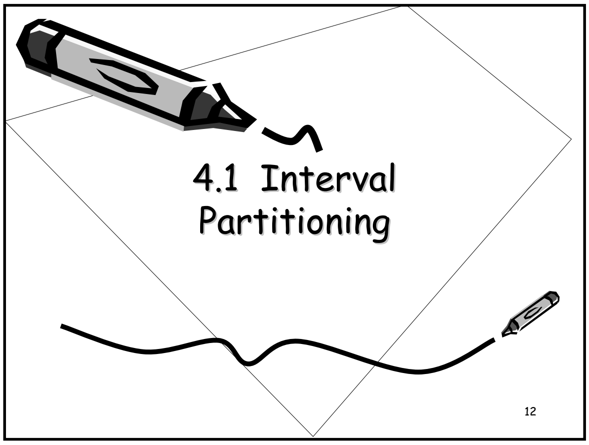


Interval Scheduling: Analysis

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

job i_{r+1} finishes before j_{r+1}

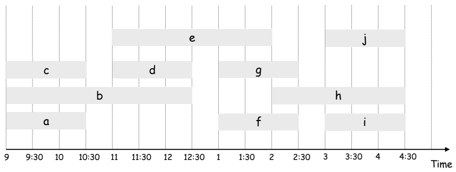
solution still feasible and optimal, but contradicts maximality of r.



4.1 Interval Partitioning

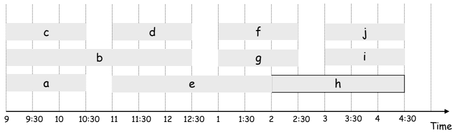
Interval Partitioning

- 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

- 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.

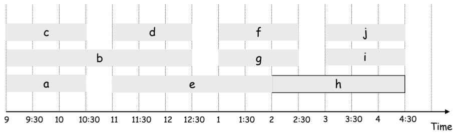


Interval Partitioning: Lower Bound on Optimal Solution

- Def. The depth of a set of open intervals is the maximum number that contain any given time.
- Key observation. Number of classrooms needed ≥ depth.
- Ex: Depth of schedule below = 3 ⇒ schedule below is optimal.

a, b, c all contain 9:30

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



Interval Partitioning: Greedy Algorithm

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

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

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

- Implementation. O(n log n).
  - For each classroom k, maintain the finish time of the last job added.
  - Keep the classrooms in a priority queue.

Interval Partitioning: Greedy Analysis

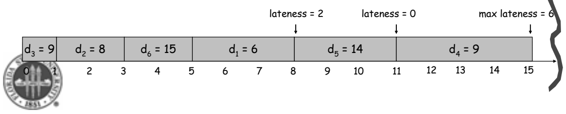
- Observation. Greedy algorithm never schedules two incompatible lectures in the same classroom.
- Theorem. Greedy algorithm is optimal.
- Pf.
  - Let d = number of classrooms that the greedy algorithm allocates.
  - Classroom d is opened because we needed to schedule a job, say j, that is incompatible with all d-1 other classrooms.
  - Since we sorted by start time, all these incompatibilities are caused by lectures that start no later than s_j.
  - Thus, we have d lectures overlapping at time $s_j + \varepsilon$.
  - Key observation ⇒ all schedules use ≥ d classrooms. ▪

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: $_j$ = max { 0, $f_j - d_j$ }.
  - Goal: schedule all jobs to minimize maximum lateness L = max $_j$.
- 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

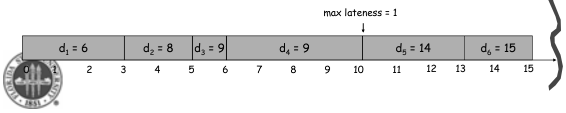
counterexample

Minimizing Lateness: Greedy Algorithm

- Greedy algorithm. Earliest deadline first.

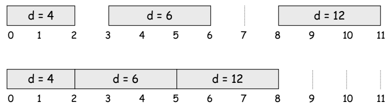
```
Sort n jobs by deadline so that $d_1 \le d_2 \le \ldots \le d_n$

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 [$s_j$, $f_j$]
```



Minimizing Lateness: No Idle Time

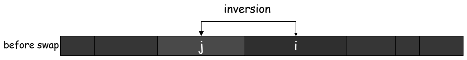
- Observation. There exists an optimal schedule with no idle time.



- Observation. The greedy schedule has no idle time.

Minimizing Lateness: Inversions

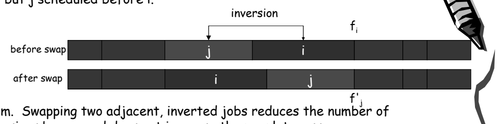
- Def. An inversion in schedule S is a pair of jobs i and j such that: i < j but j scheduled before i.



- 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 be the lateness before the swap, and let ' be it afterwards.
  - $'_k = {}_k$ for all $k \neq i, j$
  - $'_i \leq {}_i$
  - If job j is late:

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

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.

- 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.

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 cache misses.

- Ex: k = 2, initial cache = ab, requests: a, b, c, b, c, a, a, b.
- Optimal eviction schedule: 2 cache misses.

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.

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.

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

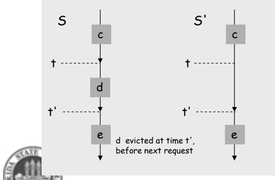
a reduced schedule

Reduced Eviction Schedules

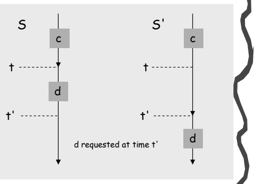
- Claim. Given any unreduced schedule S, can transform it into a reduced schedule S' with no more cache misses.
- Pf. (by induction on number of unreduced items)

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

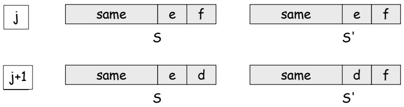
- Theorem. FF is optimal eviction algorithm.
- Pf. (by induction on number or requests j)

> Invariant: There exists an optimal reduced schedule S 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 ≠ e).
    - begin construction of S' from S by evicting e instead of f

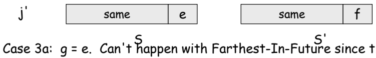


- now S' agrees with S_{FF} on first j+1 requests; we show that having element f in cache is no worse than having element e

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.

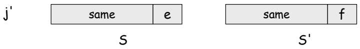
- Case 3b: g = f. Element f can't be in cache of S, so let e' be the element that S evicts.
  - if e' = e, S' accesses f from cache; now S and S' have same cache
  - if e' ≠ e, S' evicts e' and brings e into the cache; now S and S' have the same cache

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

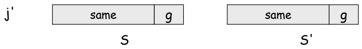
- Let j' be the first time after j+1 that S and S' take a different action, and let g be item requested at time j'.

must involve e or f (or both)



otherwise S' would take the same action

- Case 3c: g ≠ 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. [Section 13.8]
  - LIFO is arbitrarily bad.