

# CS 580: Algorithm Design and Analysis

---

Jeremiah Blocki  
Purdue University  
Spring 2019

**Announcement:** Homework 1 released!  
**Due:** January 24<sup>th</sup> at 11:59PM (Gradescope)

# Recap: Graphs

## Definition of a Graph

- Representations
  - Adjacency Matrix
  - Adjacency List
- Connectivity, Cycles
- Trees (Connected + No Cycles)
  - Rooted Trees/Binary Trees/Balanced Binary Trees

## Breadth First Search

- BFS Tree
- $O(m+n)$  algorithm
- Applications: Connected Components, Shortest Path etc...

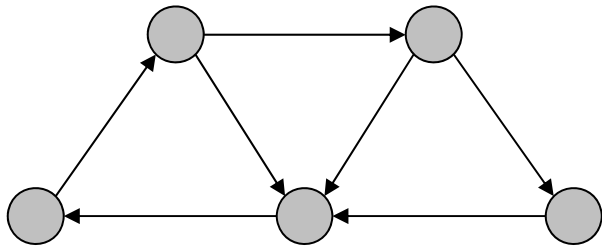
## Testing Bipartiteness

## Directed Graphs and Strong Connectivity

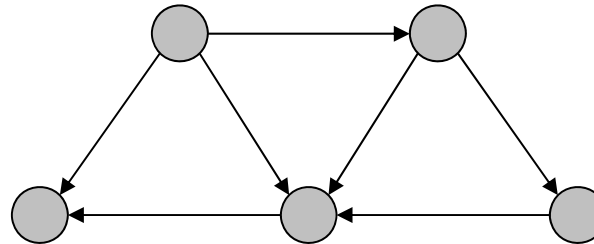
# Strong Connectivity: Algorithm

**Theorem.** Can determine if  $G$  is strongly connected in  $O(m + n)$  time.  
**Pf.**

- Pick any node  $s$ .
- Run BFS from  $s$  in  $G$ .
- Run BFS from  $s$  in  $G^{rev}$ . reverse orientation of every edge in  $G$
- Return true iff all nodes reached in both BFS executions.
- Correctness follows immediately from previous lemma. ▪



strongly connected



not strongly connected

## 3.6 DAGs and Topological Ordering

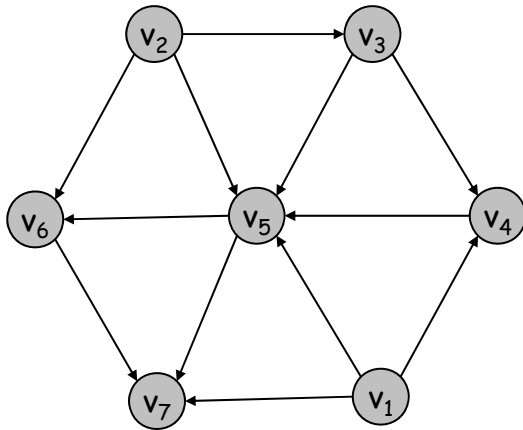
---

# Directed Acyclic Graphs

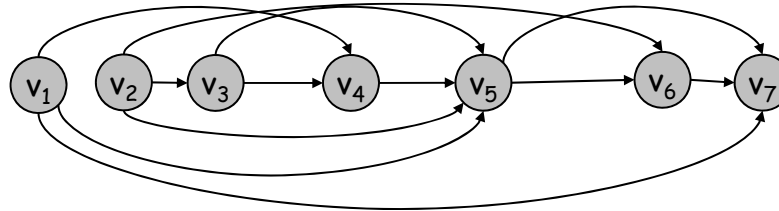
**Def.** An **DAG** is a directed graph that contains no directed cycles.

**Ex.** Precedence constraints: edge  $(v_i, v_j)$  means  $v_i$  must precede  $v_j$ .

**Def.** A **topological order** of a directed graph  $G = (V, E)$  is an ordering of its nodes as  $v_1, v_2, \dots, v_n$  so that for every edge  $(v_i, v_j)$  we have  $i < j$ .



a DAG



a topological ordering

# Precedence Constraints

Precedence constraints. Edge  $(v_i, v_j)$  means task  $v_i$  must occur before  $v_j$ .

## Applications.

- Course prerequisite graph: course  $v_i$  must be taken before  $v_j$ .
- Compilation: module  $v_i$  must be compiled before  $v_j$ . Pipeline of computing jobs: output of job  $v_i$  needed to determine input of job  $v_j$ .

Function  $F(V)$

$W := 2 * V;$

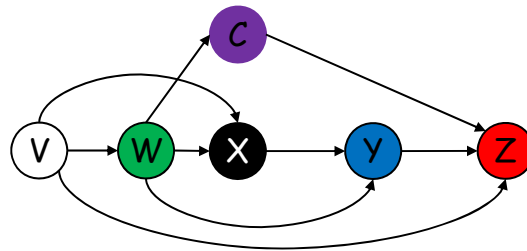
$X := W + V;$

$Y := X * W;$

$C := W * W;$

$Z := Y + V;$

return  $Z$

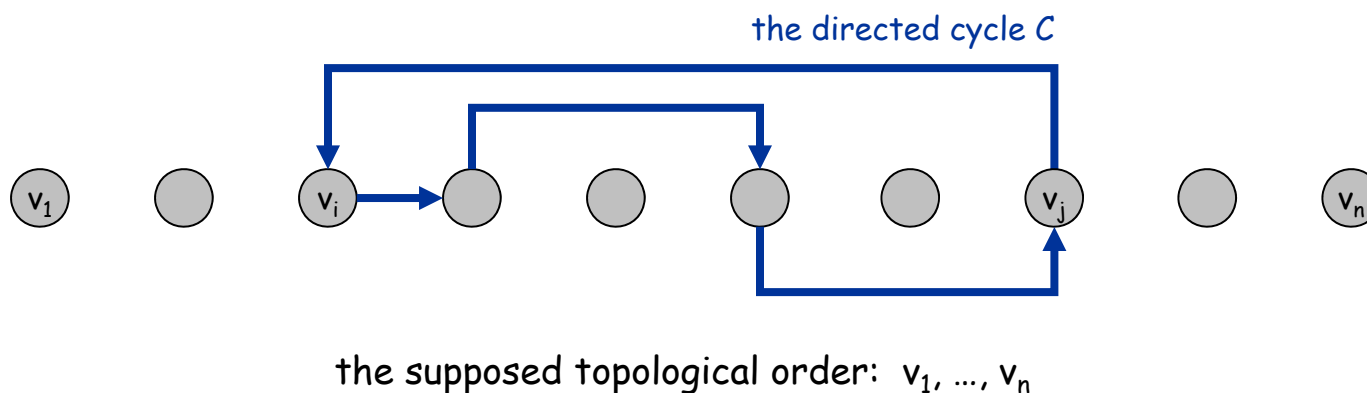


## Directed Acyclic Graphs

**Lemma.** If  $G$  has a topological order, then  $G$  is a DAG.

**Pf.** (by contradiction)

- Suppose that  $G$  has a topological order  $v_1, \dots, v_n$  and that  $G$  also has a directed cycle  $C$ . Let's see what happens.
- Let  $v_i$  be the lowest-indexed node in  $C$ , and let  $v_j$  be the node just before  $v_i$ ; thus  $(v_j, v_i)$  is an edge.
- By our choice of  $i$ , we have  $i < j$ .
- On the other hand, since  $(v_j, v_i)$  is an edge and  $v_1, \dots, v_n$  is a topological order, we must have  $j < i$ , a contradiction. ▪



## Directed Acyclic Graphs

**Lemma.** If  $G$  has a topological order, then  $G$  is a DAG.

Q. Does every DAG have a topological ordering?

Q. If so, how do we compute one?

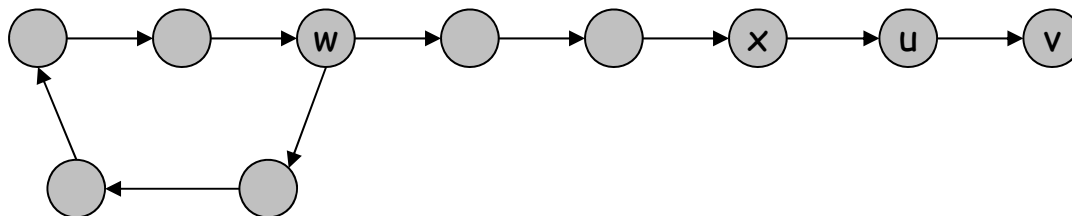


# Directed Acyclic Graphs

**Lemma.** If  $G$  is a DAG, then  $G$  has a node with no incoming edges.

**Pf.** (by contradiction)

- Suppose that  $G$  is a DAG and every node has at least one incoming edge. Let's see what happens.
- Pick any node  $v$ , and begin following edges backward from  $v$ . Since  $v$  has at least one incoming edge  $(u, v)$  we can walk backward to  $u$ .
- Then, since  $u$  has at least one incoming edge  $(x, u)$ , we can walk backward to  $x$ .
- Repeat until we visit a node, say  $w$ , twice.
- Let  $C$  denote the sequence of nodes encountered between successive visits to  $w$ .  $C$  is a cycle. ▪



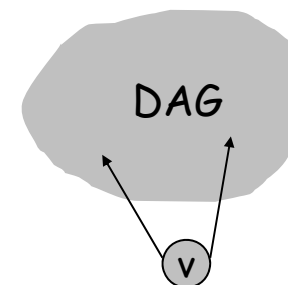
# Directed Acyclic Graphs

**Lemma.** If  $G$  is a DAG, then  $G$  has a topological ordering.



**Pf.** (by induction on  $n$ )

- Base case: true if  $n = 1$ .
- Given DAG on  $n > 1$  nodes, find a node  $v$  with no incoming edges.
- $G - \{v\}$  is a DAG, since deleting  $v$  cannot create cycles.
- By inductive hypothesis,  $G - \{v\}$  has a topological ordering.
- Place  $v$  first in topological ordering; then append nodes of  $G - \{v\}$  in topological order. This is valid since  $v$  has no incoming edges. ▪



---

To compute a topological ordering of  $G$ :

Find a node  $v$  with no incoming edges and order it first

Delete  $v$  from  $G$

Recursively compute a topological ordering of  $G - \{v\}$

and append this order after  $v$

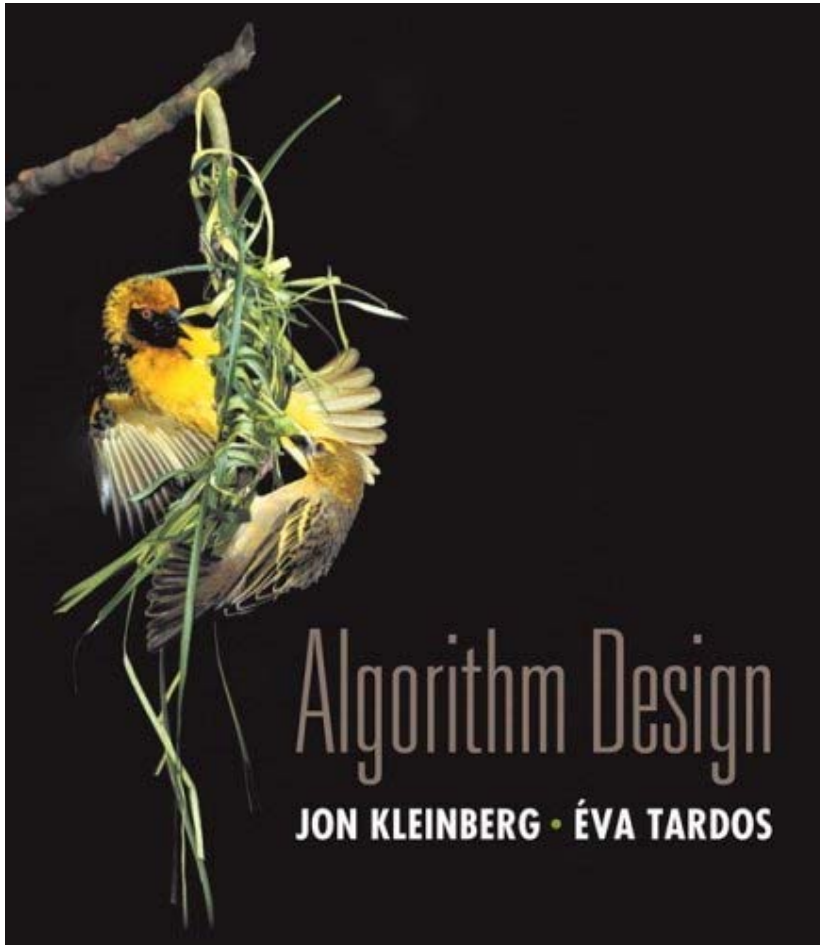
---

## Topological Sorting Algorithm: Running Time

**Theorem.** Algorithm finds a topological order in  $O(m + n)$  time.

**Pf.**

- Maintain the following information:
  - $\text{count}[w]$  = remaining number of incoming edges
  - $S$  = set of remaining nodes with no incoming edges
- Initialization:  $O(m + n)$  via single scan through graph.
- Update: to delete  $v$ 
  - remove  $v$  from  $S$
  - decrement  $\text{count}[w]$  for all edges from  $v$  to  $w$ , and add  $w$  to  $S$  if  $\text{count}[w]$  hits 0
  - this is  $O(1)$  per edge   ▪



# Greedy Algorithms



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

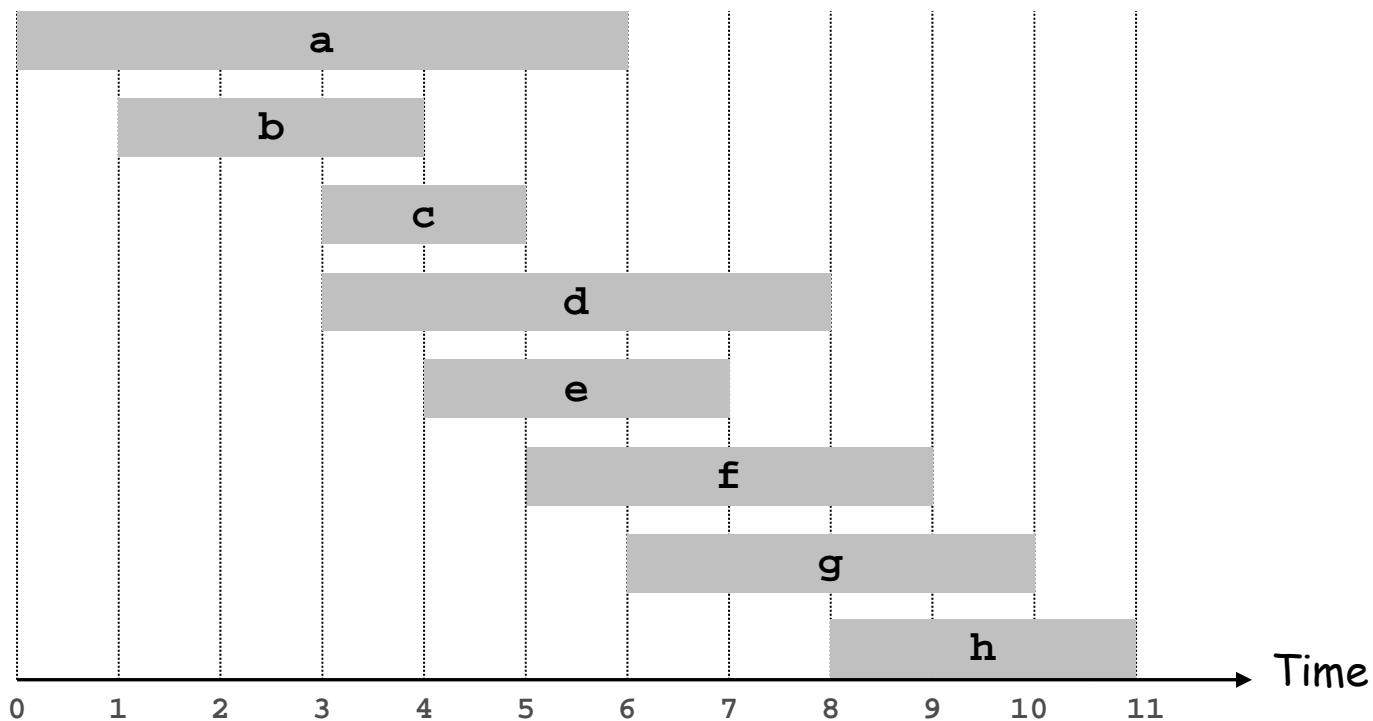
## 4.1 Interval Scheduling

---

# 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

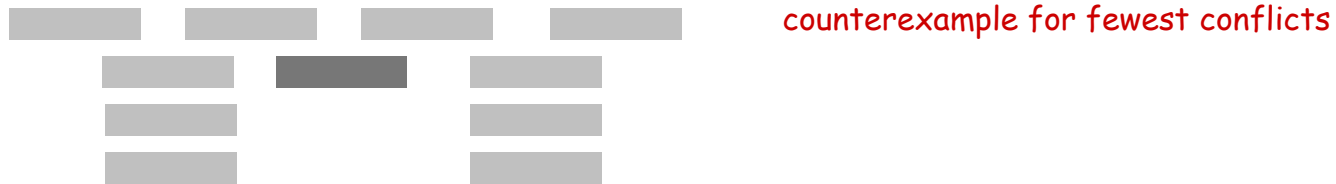
**Greedy template.** Consider jobs in some natural order.

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 Algorithms

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





## 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  $f_1 \leq f_2 \leq \dots \leq f_n$ .
```

↙ set of jobs selected

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



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

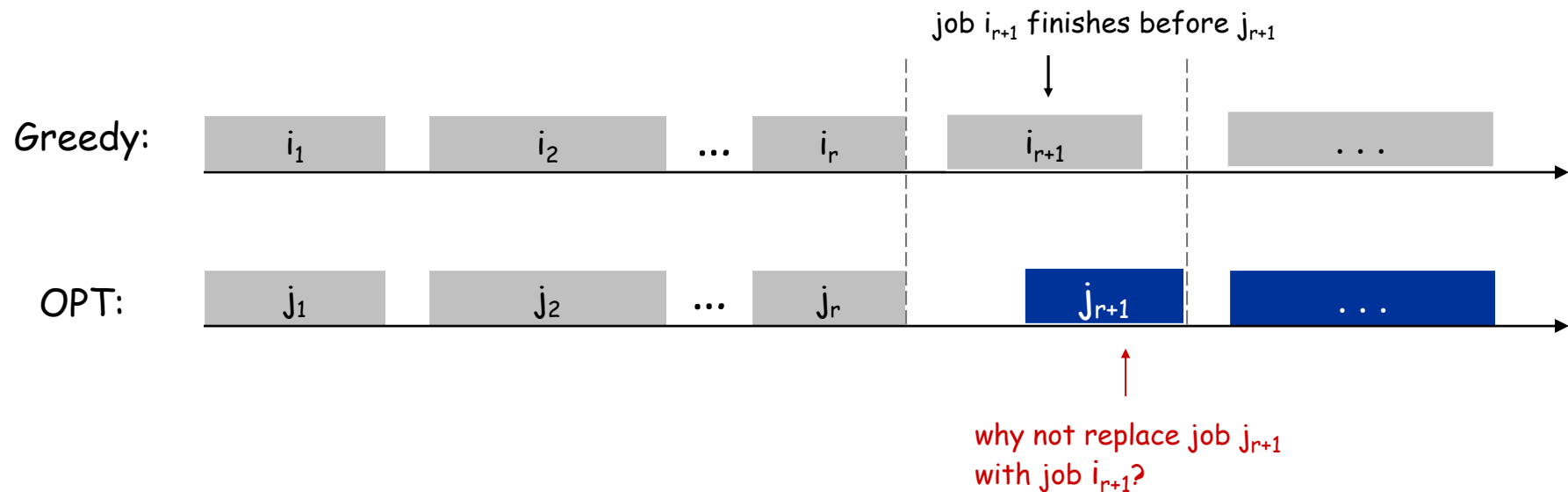
- Remember job  $j^*$  that was added last to  $A$ .
- Job  $j$  is compatible with  $A$  if  $s_j \geq f_{j^*}$ .

# Interval Scheduling: Analysis

**Theorem.** Greedy algorithm is optimal.

**Pf.** (by contradiction)

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

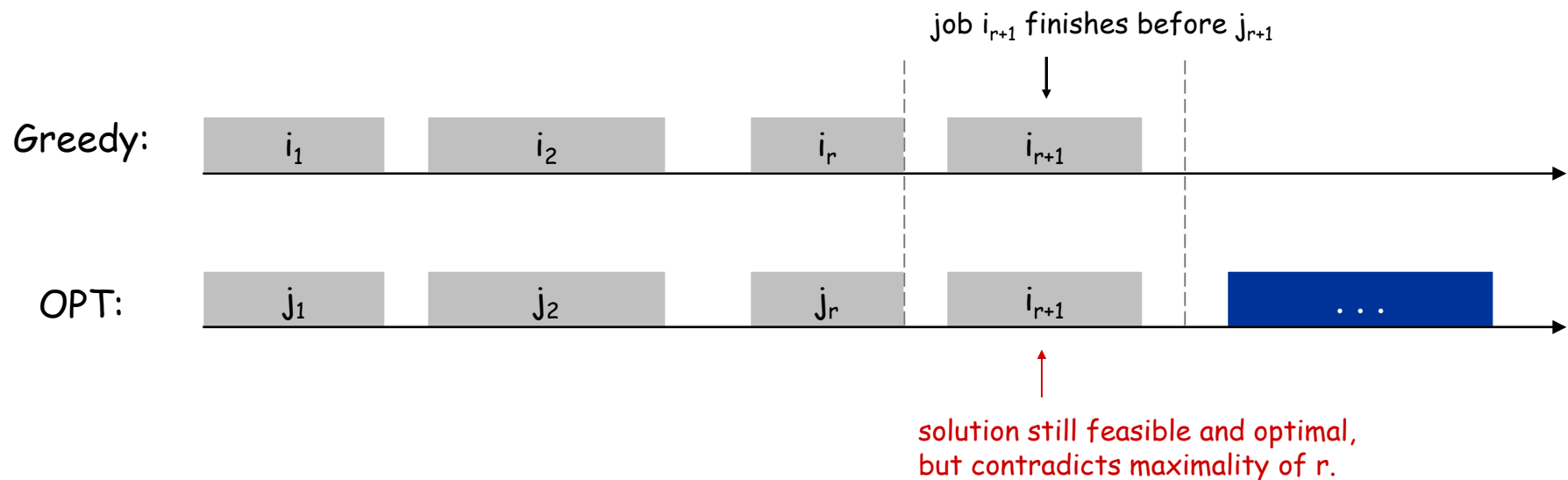


# Interval Scheduling: Analysis

**Theorem.** Greedy algorithm is optimal.

**Pf.** (by contradiction)

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



## 4.1 Interval Partitioning

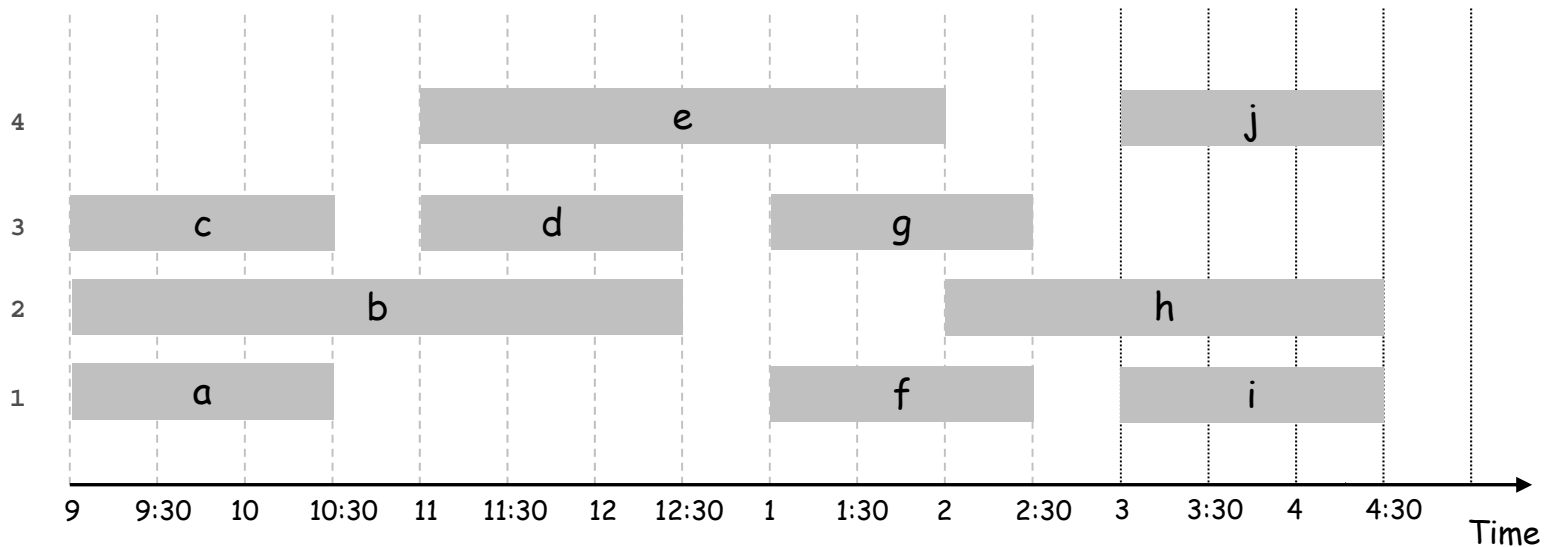
---

# 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.

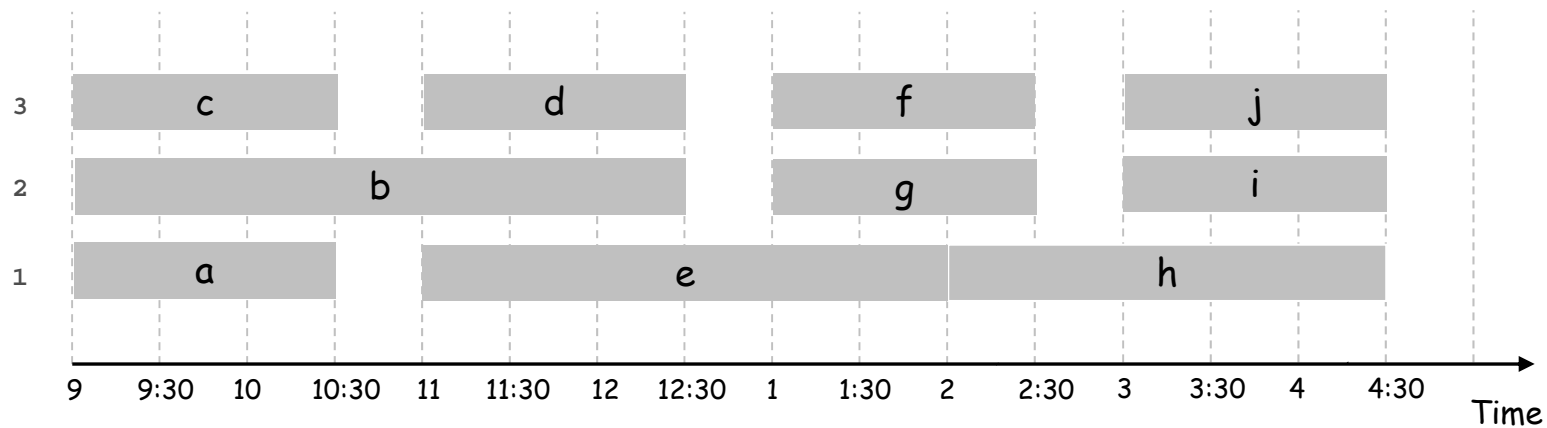


# Interval Partitioning

## 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

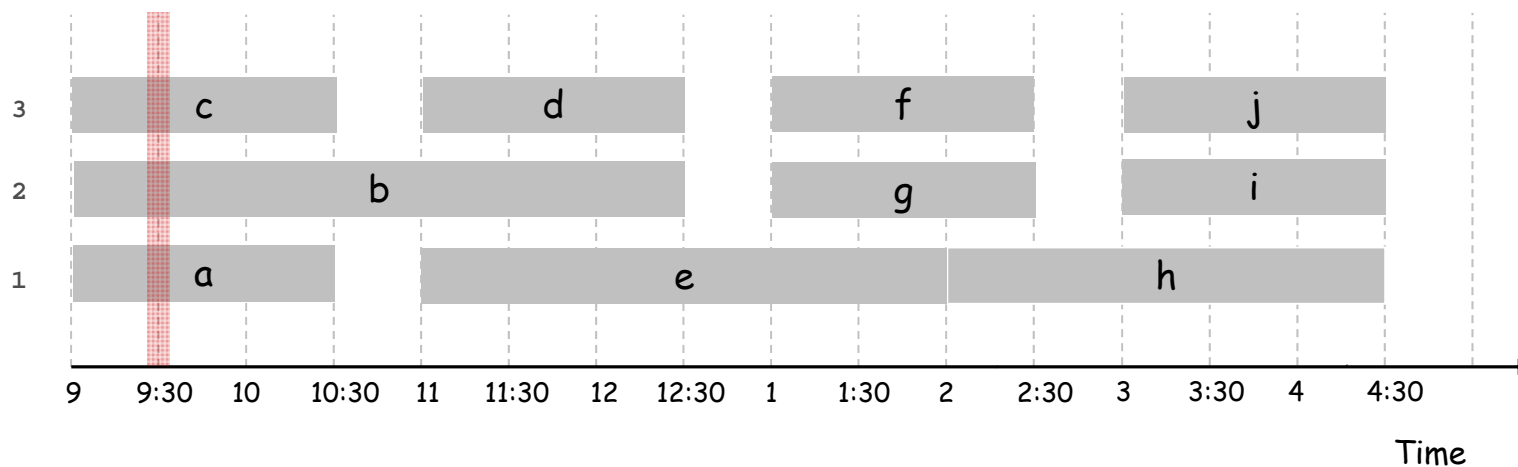
**Def.** The **depth** of a set of open intervals is the maximum number that contain any given time.

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

**Ex:** Depth of schedule below = 3  $\Rightarrow$  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  $s_1 \leq s_2 \leq \dots \leq s_n$ .  
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

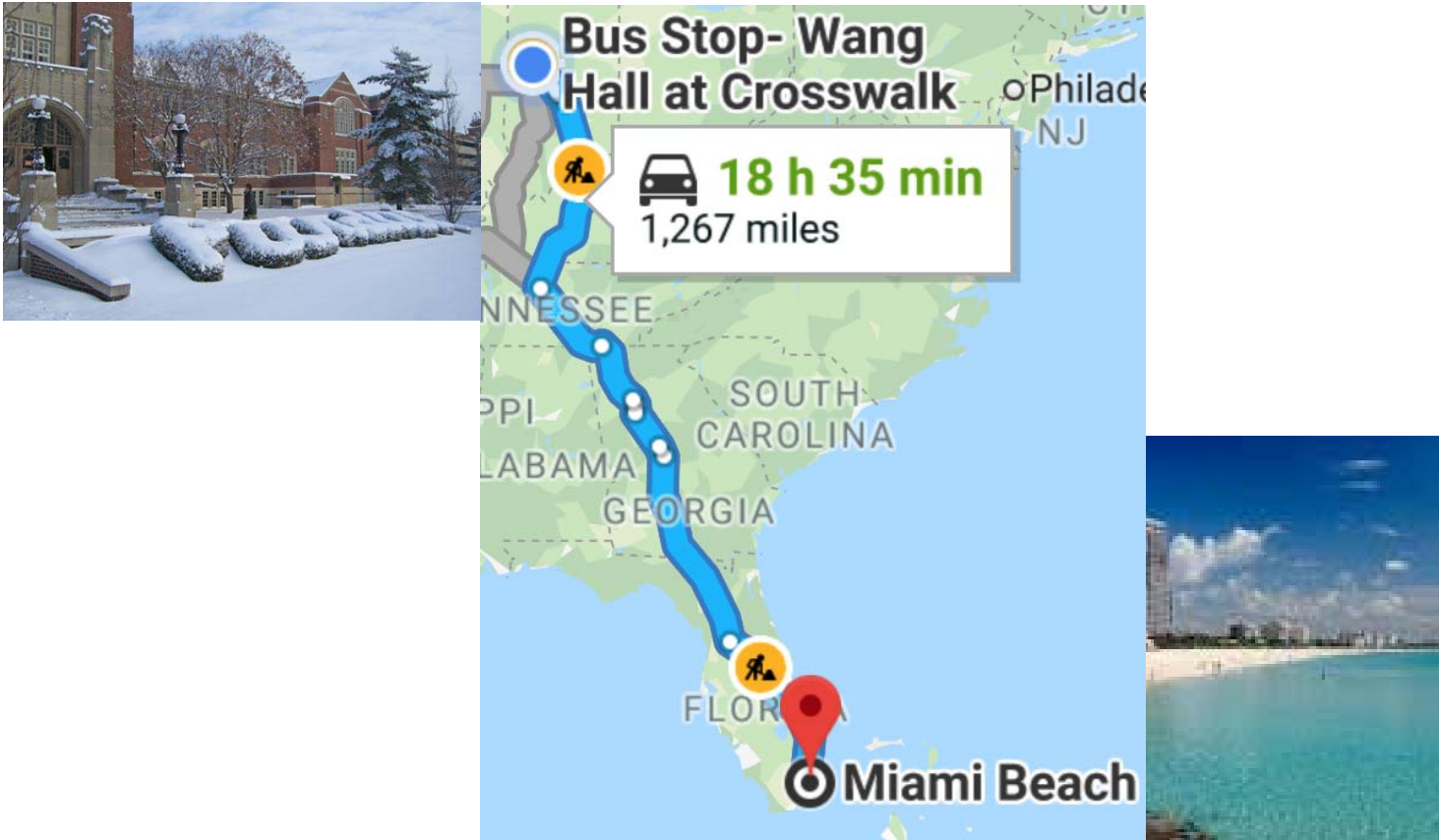
**Observation.** Greedy algorithm never schedules two incompatible lectures in the same classroom.

**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.
- These  $d$  jobs (including  $j$ ) each end after  $s_j$ .
- 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  $s_j + \epsilon$ .
- Key observation  $\Rightarrow$  all schedules use  $\geq d$  classrooms. ▪

## 4.4 Shortest Paths in a Graph



shortest route from Wang Hall to Miami Beach

# Shortest Path Problem

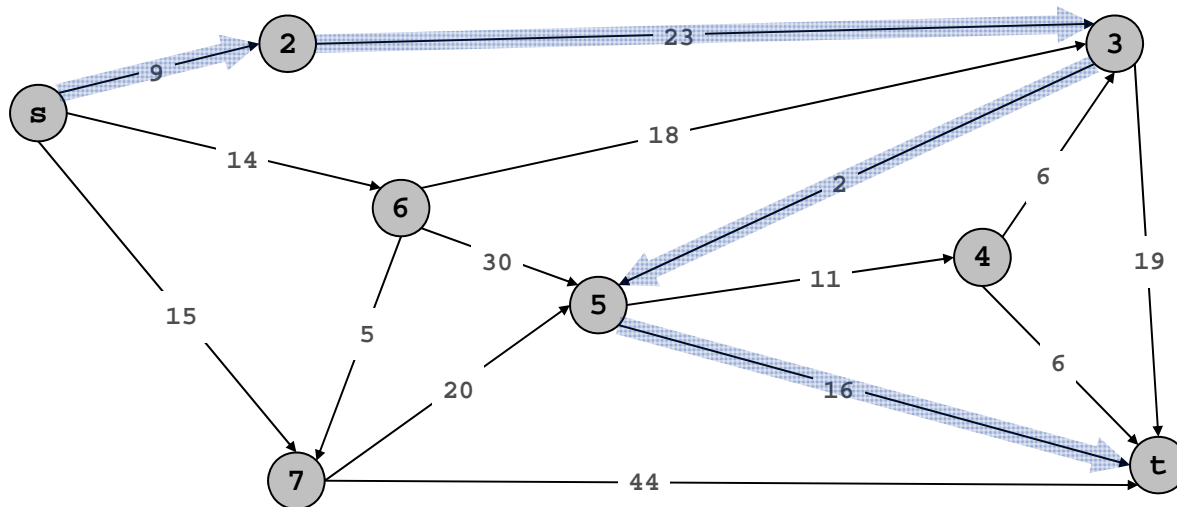
## Shortest path network.

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

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

Cost of path  $s$ -2-3-5- $t$   
=  $9 + 23 + 2 + 16$   
= 50.

cost of path = sum of edge costs in path



# Dijkstra's Algorithm

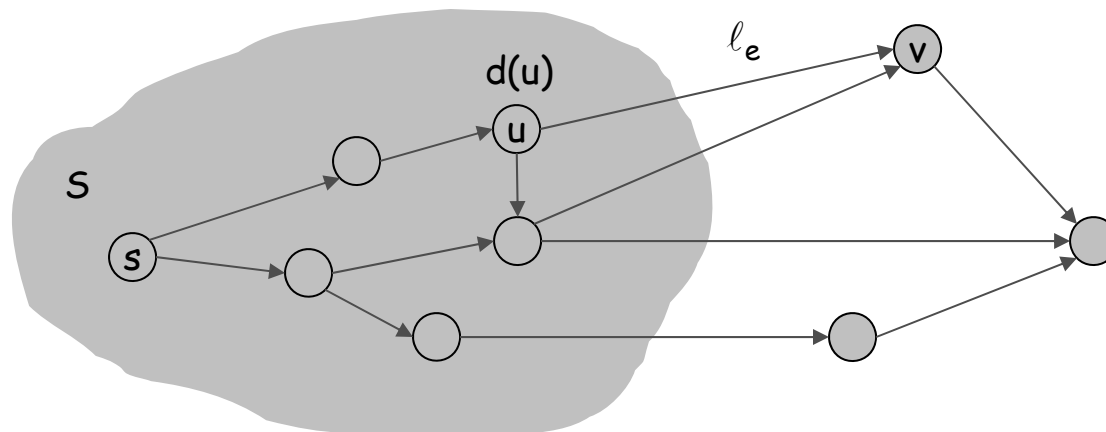
## Dijkstra's algorithm.

- 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  $d(v) = \pi(v)$ .

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



# Dijkstra's Algorithm

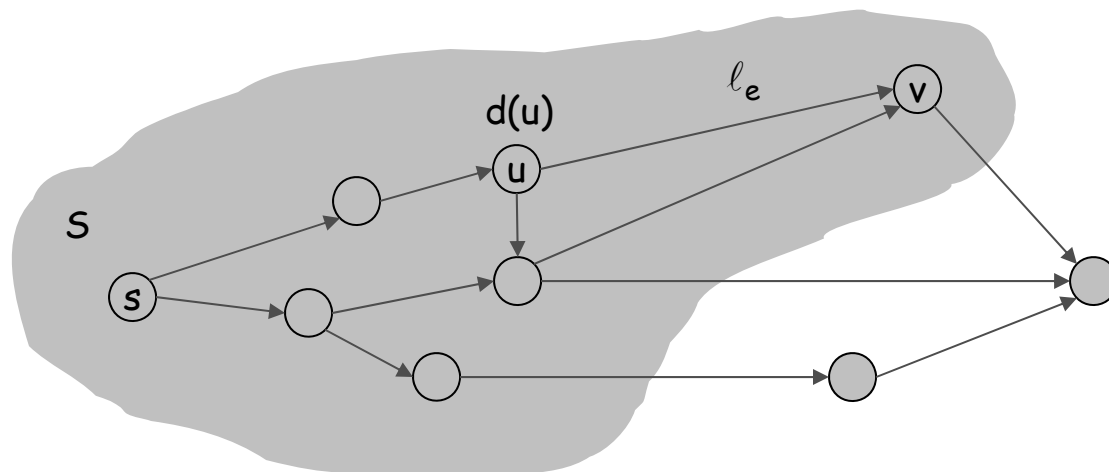
## Dijkstra's algorithm.

- 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  $d(v) = \pi(v)$ .

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



# Dijkstra's Algorithm: Proof of Correctness

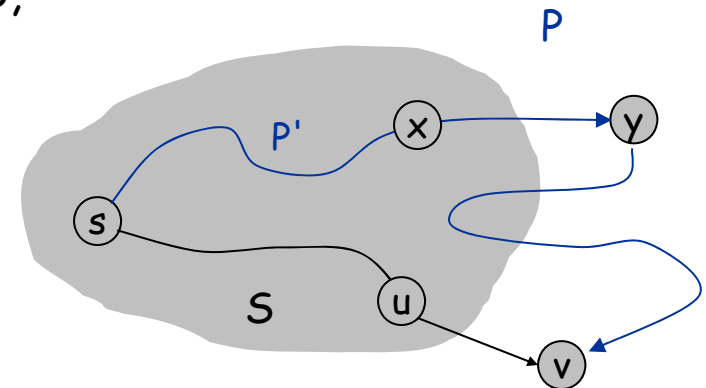
**Invariant.** For each node  $u \in S$ ,  $d(u)$  is the length of the shortest  $s$ - $u$  path.

**Pf.** (by induction on  $|S|$ )

**Base case:**  $|S| = 1$  is trivial.

**Inductive hypothesis:** Assume true for  $|S| = k \geq 1$ .

- Let  $v$  be next node added to  $S$ , and let  $u$ - $v$  be the chosen edge.
- The shortest  $s$ - $u$  path plus  $(u, v)$  is an  $s$ - $v$  path of length  $\pi(v)$ .
- Consider any  $s$ - $v$  path  $P$ . We'll see that it's no shorter than  $\pi(v)$ .
- Let  $x$ - $y$  be the first edge in  $P$  that leaves  $S$ , and let  $P'$  be the subpath to  $x$ .
- $P$  is already too long as soon as it leaves  $S$ .



$$l(P) \geq l(P') + l(x, y) \geq d(x) + l(x, y) \geq \pi(y) \geq \pi(v)$$

$\uparrow$  nonnegative weights       $\uparrow$  inductive hypothesis       $\uparrow$  defn of  $\pi(y)$        $\uparrow$  Dijkstra chose  $v$  instead of  $y$

## Dijkstra's Algorithm: Implementation

For each unexplored node, explicitly maintain  $\pi(v) = \min_{e=(u,v): u \in S} d(u) + \ell_e$ .

- Next node to explore = node with minimum  $\pi(v)$ .
- When exploring  $v$ , for each incident edge  $e = (v, w)$ , update

$$\pi(w) = \min \{ \pi(w), \pi(v) + \ell_e \}.$$

**Efficient implementation.** Maintain a priority queue of unexplored nodes, prioritized by  $\pi(v)$ .



PQ Operation	Dijkstra	Array	Binary heap	d-way Heap	Fib heap †
Insert	$n$	$n$	$\log n$	$d \log_d n$	1
ExtractMin	$n$	$n$	$\log n$	$d \log_d n$	$\log n$
ChangeKey	$m$	1	$\log n$	$\log_d n$	1
IsEmpty	$n$	1	1	1	1
Total		$n^2$	$m \log n$	$m \log_{m/n} n$	$m + n \log n$

† Individual ops are amortized bounds

# Extra Slides

---



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

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  $d_j - t_j$ .

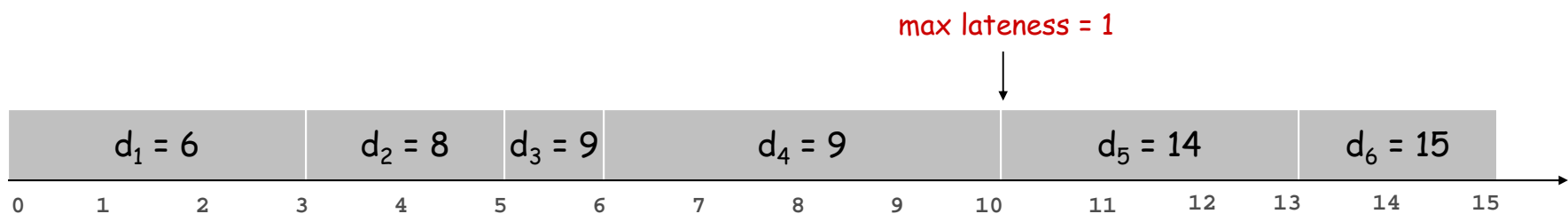
	1	2
$t_j$	1	10
$d_j$	2	10

counterexample

# Minimizing Lateness: Greedy Algorithm

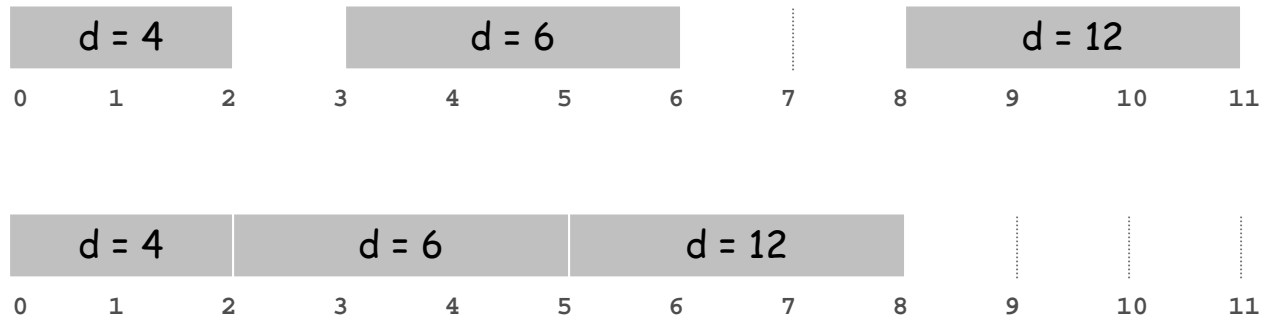
Greedy algorithm. Earliest deadline first.

```
Sort n jobs by deadline so that  $d_1 \leq d_2 \leq \dots \leq d_n$   
  
t ← 0  
for j = 1 to n  
    Assign job j to interval [t, t + tj]  
    sj ← t, fj ← t + tj  
    t ← t + tj  
output intervals [sj, fj]
```



## 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.** Given a schedule  $S$ , an **inversion** is a pair of jobs  $i$  and  $j$  such that:  $i < j$  but  $j$  scheduled before  $i$ .



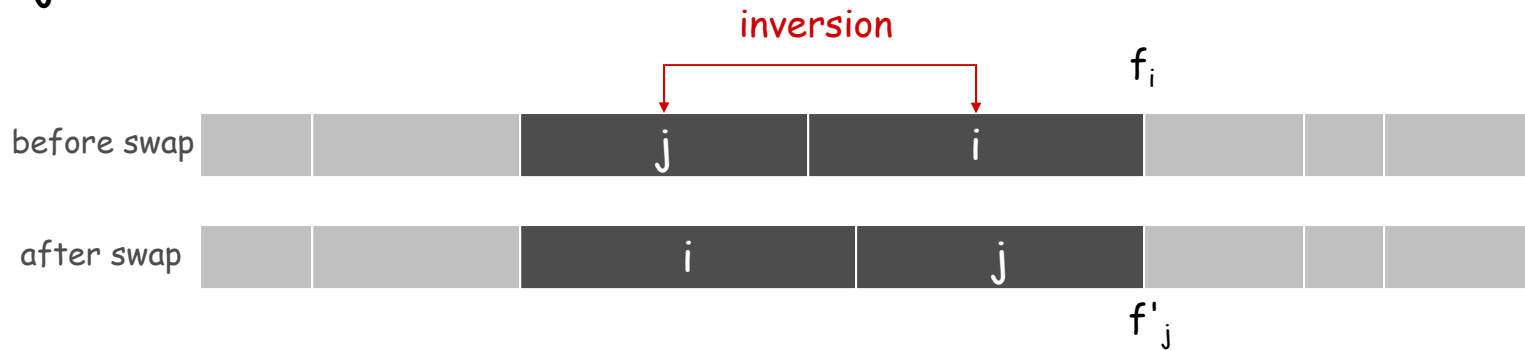
[ as before, we assume jobs are numbered so that  $d_1 \leq d_2 \leq \dots \leq d_n$  ]

**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.** Given a schedule  $S$ , an **inversion** is a pair of jobs  $i$  and  $j$  such that:  $i < j$  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  $\ell$  be the lateness before the swap, and let  $\ell'$  be it afterwards.

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

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



## 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.

*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.

*Other greedy algorithms.* Kruskal, Prim, Dijkstra, Huffman, ...

## 4.3 Optimal Caching

---

# Optimal Offline Caching

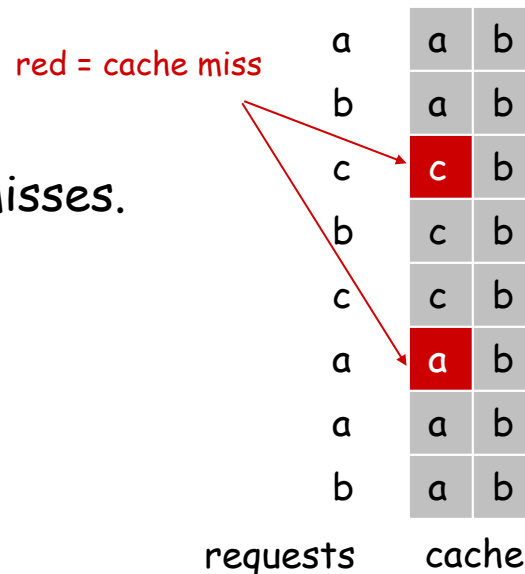
## Caching.

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

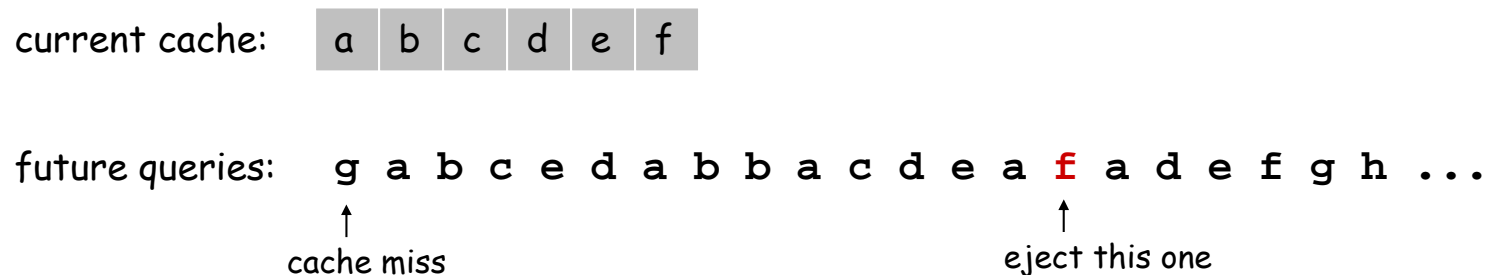
**Ex:**  $k = 2$ , initial cache =  $ab$ ,  
requests:  $a, b, c, b, c, a, a, b$ .

**Optimal eviction schedule:** 2 cache misses.



# 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 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

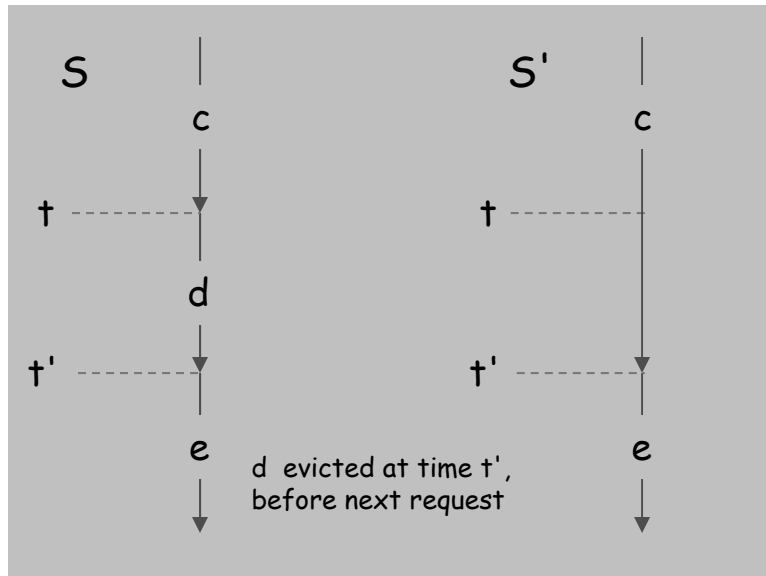
a reduced schedule

## Reduced Eviction Schedules

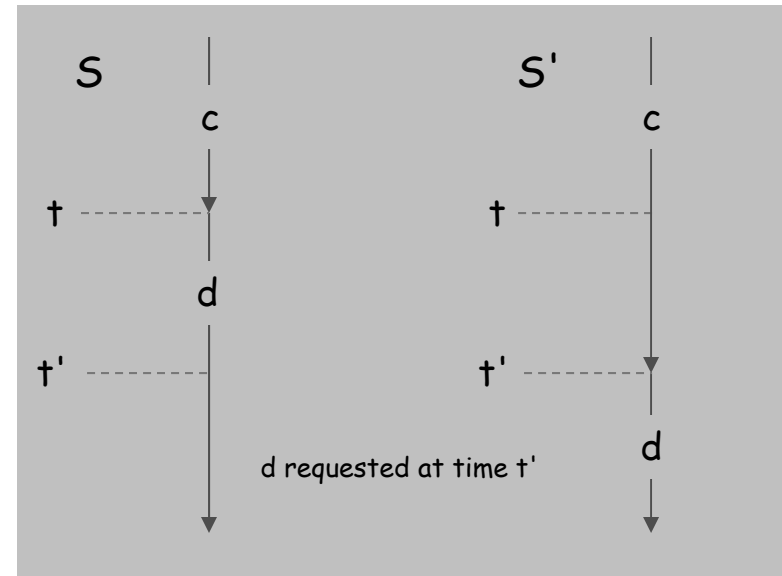
**Claim.** Given any unreduced schedule  $S$ , can transform it into a reduced schedule  $S'$  with no more cache misses.

**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 of 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.

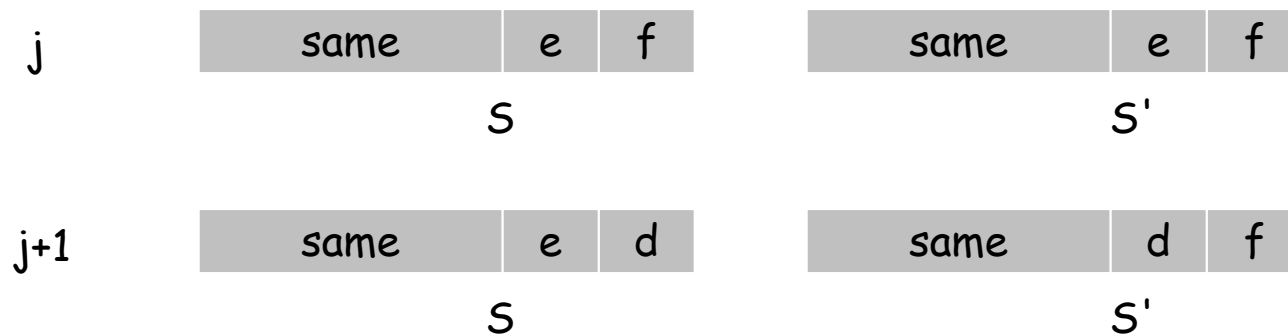
- Consider  $(j+1)^{st}$  request  $d = d_{j+1}$ .
- Since  $S$  and  $S_{FF}$  have agreed up until now, they have the same cache contents before request  $j+1$ .
- Case 1: ( $d$  is already in the cache).  $S' = S$  satisfies invariant.
- Case 2: ( $d$  is not in the cache and  $S$  and  $S_{FF}$  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  $f \neq e$ ).
  - begin construction of  $S'$  from  $S$  by evicting  $e$  instead of  $f$

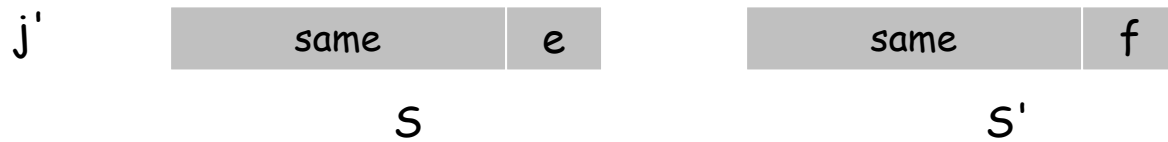


- now  $S'$  agrees with  $S_{FF}$  on first  $j+1$  requests; we show that having element  $f$  in cache is no worse than having element  $e$

## 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)



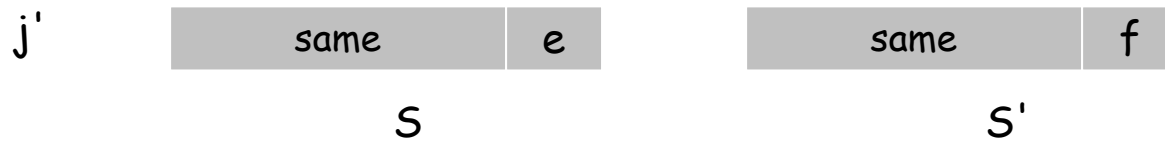
- Case 3a:  $g = e$ . Can't happen with Farthest-In-Future since there must be a request for  $f$  before  $e$ .
  
- Case 3b:  $g = f$ . Element  $f$  can't be in cache of  $S$ , so let  $e'$  be the element that  $S$  evicts.
  - if  $e' = e$ ,  $S'$  accesses  $f$  from cache; now  $S$  and  $S'$  have same cache
  - if  $e' \neq e$ ,  $S'$  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. [Section 13.8]
- LIFO is arbitrarily bad.

# Coin Changing

---

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

*- Gordon Gecko (Michael Douglas)*



## Coin Changing

**Goal.** Given currency denominations: 1, 5, 10, 25, 100, devise a method to pay amount to customer using fewest number of coins.

**Ex:** 34¢.



**Cashier's algorithm.** At each iteration, add coin of the largest value that does not take us past the amount to be paid.

**Ex:** \$2.89.



## Coin-Changing: Greedy Algorithm

**Cashier's algorithm.** At each iteration, add coin of the largest value that does not take us past the amount to be paid.

```
Sort coins denominations by value:  $c_1 < c_2 < \dots < c_n$ .  
  
coins selected  
S ←  $\phi$   
while (x ≠ 0) {  
    let k be largest integer such that  $c_k \leq x$   
    if (k = 0)  
        return "no solution found"  
    x ← x -  $c_k$   
    S ← S ∪ {k}  
}  
return S
```

Q. Is cashier's algorithm optimal?

## Coin-Changing: Analysis of Greedy Algorithm

**Theorem.** Greedy is optimal for U.S. coinage: 1, 5, 10, 25, 100.

**Pf.** (by induction on  $x$ )

- Consider optimal way to change  $c_k \leq x < c_{k+1}$  : greedy takes coin  $k$ .
- We claim that any optimal solution must also take coin  $k$ .
  - if not, it needs enough coins of type  $c_1, \dots, c_{k-1}$  to add up to  $x$
  - table below indicates no optimal solution can do this
- Problem reduces to coin-changing  $x - c_k$  cents, which, by induction, is optimally solved by greedy algorithm. ▪

$k$	$c_k$	All optimal solutions must satisfy	Max value of coins 1, 2, ..., $k-1$ in any OPT
1	1	$P \leq 4$	-
2	5	$N \leq 1$	4
3	10	$N + D \leq 2$	$4 + 5 = 9$
4	25	$Q \leq 3$	$20 + 4 = 24$
5	100	no limit	$75 + 24 = 99$



# Coin-Changing: Analysis of Greedy Algorithm

**Observation.** Greedy algorithm 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.



# Selecting Breakpoints

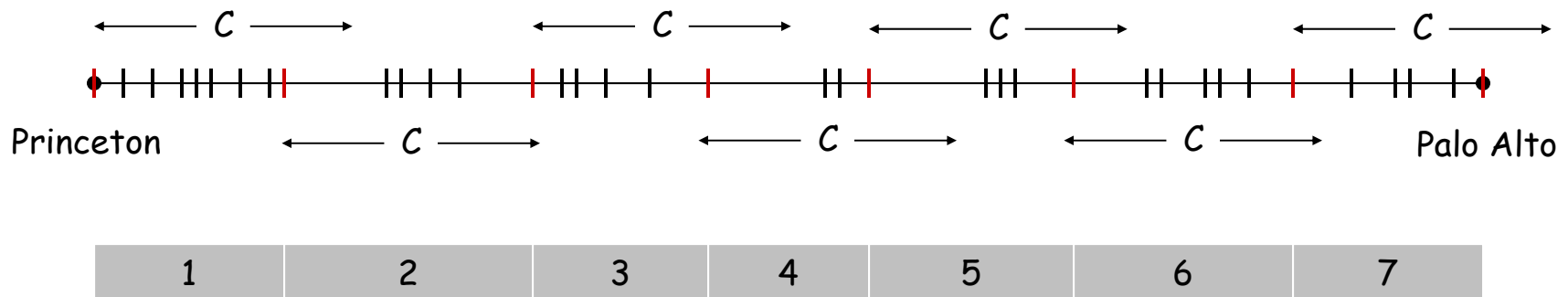
---

# Selecting Breakpoints

## Selecting breakpoints.

- Road trip from Princeton to Palo Alto along fixed route.
- Refueling stations at certain points along the way.
- Fuel capacity =  $C$ .
- Goal: makes as few refueling stops as possible.

**Greedy algorithm.** Go as far as you can before refueling.



## Selecting Breakpoints: Greedy Algorithm

Truck driver's algorithm.

```
Sort breakpoints so that:  $0 = b_0 < b_1 < b_2 < \dots < b_n = L$ 
```

```
 $S \leftarrow \{0\}$  ← breakpoints selected
```

```
 $x \leftarrow 0$  ← current location
```

```
while ( $x \neq b_n$ )
```

```
    let  $p$  be largest integer such that  $b_p \leq x + C$ 
```

```
    if ( $b_p = x$ )
```

```
        return "no solution"
```

```
     $x \leftarrow b_p$ 
```

```
     $S \leftarrow S \cup \{p\}$ 
```

```
return  $S$ 
```

Implementation.  $O(n \log n)$

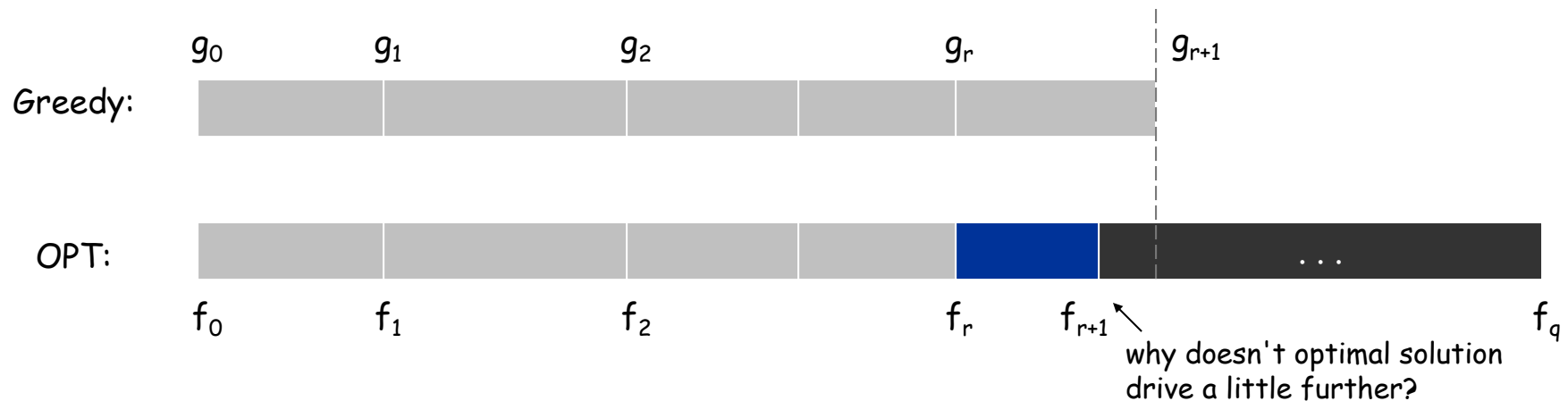
- Use binary search to select each breakpoint  $p$ .

## Selecting Breakpoints: Correctness

**Theorem.** Greedy algorithm is optimal.

**Pf.** (by contradiction)

- Assume greedy is not optimal, and let's see what happens.
- Let  $0 = g_0 < g_1 < \dots < g_p = L$  denote set of breakpoints chosen by greedy.
- Let  $0 = f_0 < f_1 < \dots < f_q = L$  denote set of breakpoints in an optimal solution with  $f_0 = g_0, f_1 = g_1, \dots, f_r = g_r$  for largest possible value of  $r$ .
- Note:  $g_{r+1} > f_{r+1}$  by greedy choice of algorithm.

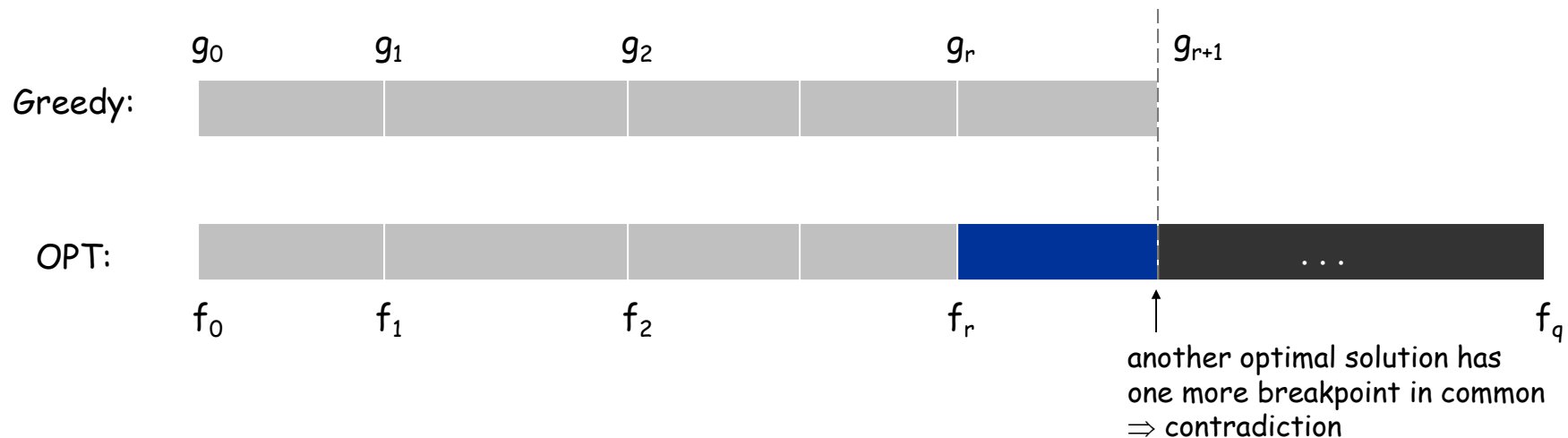


## Selecting Breakpoints: Correctness

**Theorem.** Greedy algorithm is optimal.

**Pf.** (by contradiction)

- Assume greedy is not optimal, and let's see what happens.
- Let  $0 = g_0 < g_1 < \dots < g_p = L$  denote set of breakpoints chosen by greedy.
- Let  $0 = f_0 < f_1 < \dots < f_q = L$  denote set of breakpoints in an optimal solution with  $f_0 = g_0, f_1 = g_1, \dots, f_r = g_r$  for largest possible value of  $r$ .
- Note:  $g_{r+1} > f_{r+1}$  by greedy choice of algorithm.



# Edsger W. Dijkstra

The question of whether computers can think is like the question of whether submarines can swim.

Do only what only you can do.

In their capacity as a tool, computers will be but a ripple on the surface of our culture. In their capacity as intellectual challenge, they are without precedent in the cultural history of mankind.

The use of COBOL cripples the mind; its teaching should, therefore, be regarded as a criminal offence.

APL is a mistake, carried through to perfection. It is the language of the future for the programming techniques of the past: it creates a new generation of coding bums.

