

# Greedy Algorithms\*

Xiaofeng Gao

Department of Computer Science and Engineering  
Shanghai Jiao Tong University, P.R.China

Algorithm Course: Shanghai Jiao Tong University

---

\* Special thanks is given to Prof. Kevin Wayne@Princeton for sharing his lecture notes, and also given to Mr. Hongjian Cao from CS2015@SJTU for producing this slide.

# Outline

- 1 Basic Methodology
  - Interval Scheduling
  - Interval Partitioning
  - Scheduling to Minimize Lateness
  
- 2 More Examples
  - Optimal Caching
  - Coin Changing

# Outline

- 1 Basic Methodology
  - Interval Scheduling
  - Interval Partitioning
  - Scheduling to Minimize Lateness
- 2 More Examples
  - Optimal Caching
  - Coin Changing

# Interval Scheduling: An Introductory Example

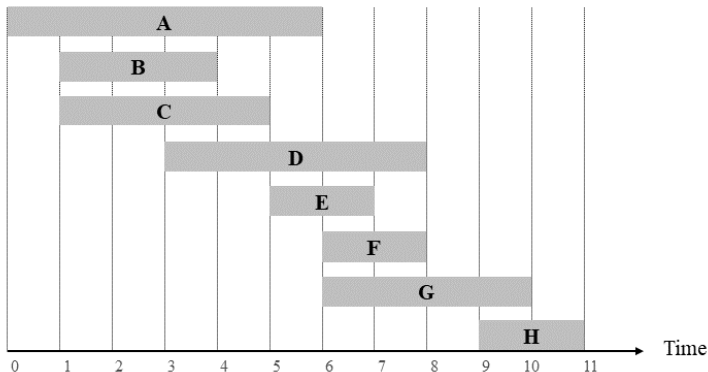
- Job  $j$  starts at  $s_j$  and finishes at  $f_j$ .
- Two jobs are **compatible** if they don't overlap.

**Goal:** find maximum subset of mutually compatible jobs.

# Interval Scheduling: An Introductory Example

- 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.



# Greedy Strategy

**Optimization Problem:** Given a problem  $\Pi$  with domain  $\mathbf{X}$ , choose a subset or determine a sequence according to some **maximization** or **minimization** objective. (Each  $X \in \mathbf{X}$  is an instance of  $\Pi$ )

# Greedy Strategy

**Optimization Problem:** Given a problem  $\Pi$  with domain  $\mathbf{X}$ , choose a subset or determine a sequence according to some **maximization** or **minimization** objective. (Each  $X \in \mathbf{X}$  is an instance of  $\Pi$ )

**General Template:** Consider each item  $x_i \in X$  of problem  $\Pi$  (in some order), make choice that looks **best** at the moment.

Note: it makes a *locally optimal* choice in hope that this choice will lead to a *globally optimal* solution.

# Greedy Strategy

**Optimization Problem:** Given a problem  $\Pi$  with domain  $\mathbf{X}$ , choose a subset or determine a sequence according to some **maximization** or **minimization** objective. (Each  $X \in \mathbf{X}$  is an instance of  $\Pi$ )

**General Template:** Consider each item  $x_i \in X$  of problem  $\Pi$  (in some order), make choice that looks **best** at the moment.

Note: it makes a *locally optimal* choice in hope that this choice will lead to a *globally optimal* solution.

**Interval Scheduling Problem:** Consider jobs in some natural order. Take each job provided to judge its compatibility with the ones already taken.



# Attempts

[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  $c_j$ .

# Attempts

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

Counter Example:



[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  $c_j$ .

# Attempts

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

Counter Example:



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

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

Counter Example:



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

# Attempts

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

Counter Example:



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

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

Counter Example:



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

Counter Example:



# Greedy Interval Scheduling Algorithm

---

## Algorithm 1: Greedy Interval Scheduling

---

```
1 Sort jobs by finish times so that  $f_1 \leq f_2 \leq \dots \leq f_n$ ;  
2  $A \leftarrow \emptyset$ ; // set of jobs selected  
3 for  $j = 1$  to  $n$  do  
4   if job  $j$  is compatible with  $A$  then  
5      $A \leftarrow A \cup \{j\}$ ;  
6 return  $A$ ;
```

---

# Greedy Interval Scheduling Algorithm

---

## Algorithm 1: Greedy Interval Scheduling

---

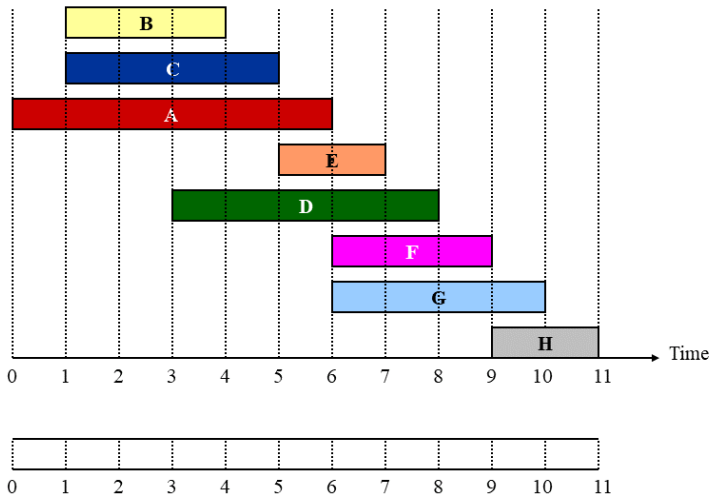
```
1 Sort jobs by finish times so that  $f_1 \leq f_2 \leq \dots \leq f_n$ ;  
2  $A \leftarrow \emptyset$ ; // set of jobs selected  
3 for  $j = 1$  to  $n$  do  
4   if job  $j$  is compatible with  $A$  then  
5      $A \leftarrow A \cup \{j\}$ ;  
6 return  $A$ ;
```

---

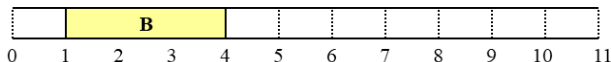
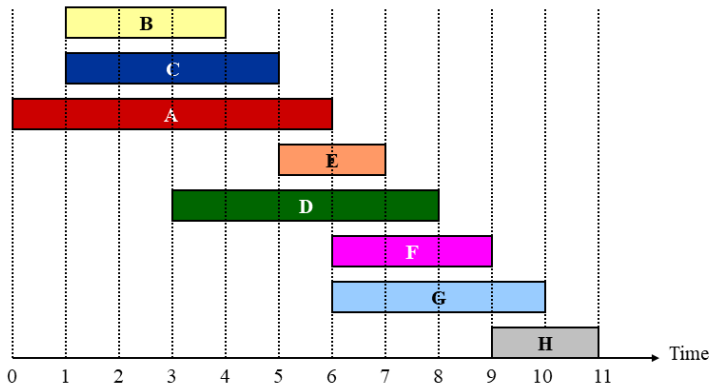
Implementation:  $O(n \log n)$ .

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

## Demo

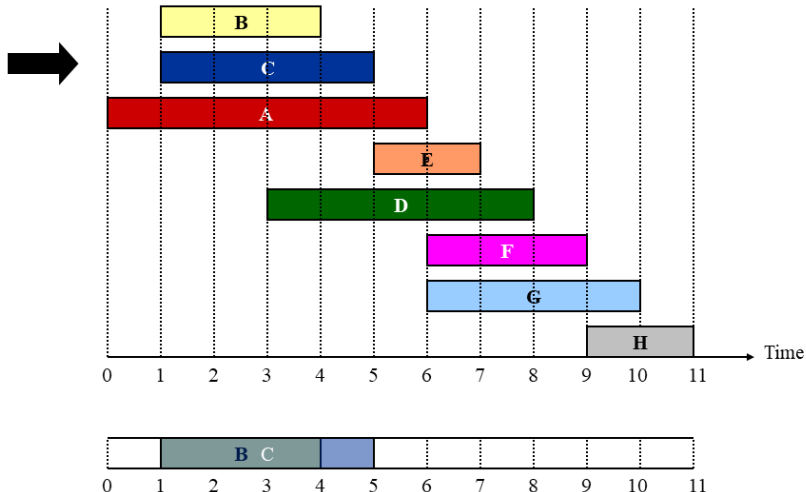


## Demo

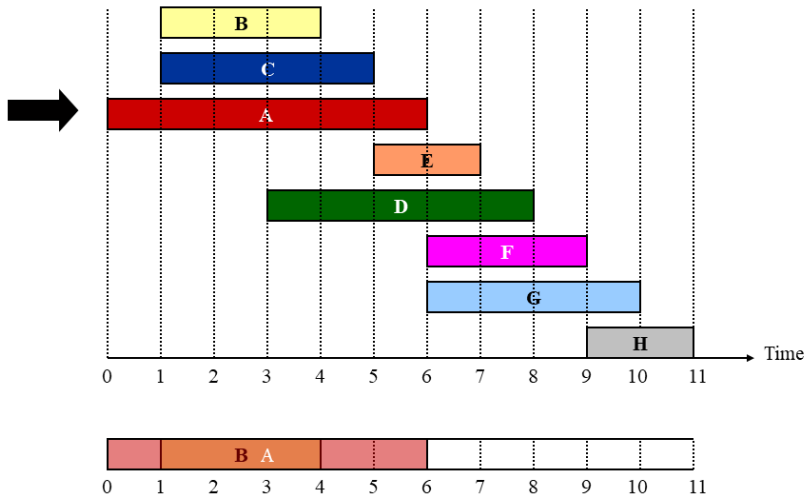




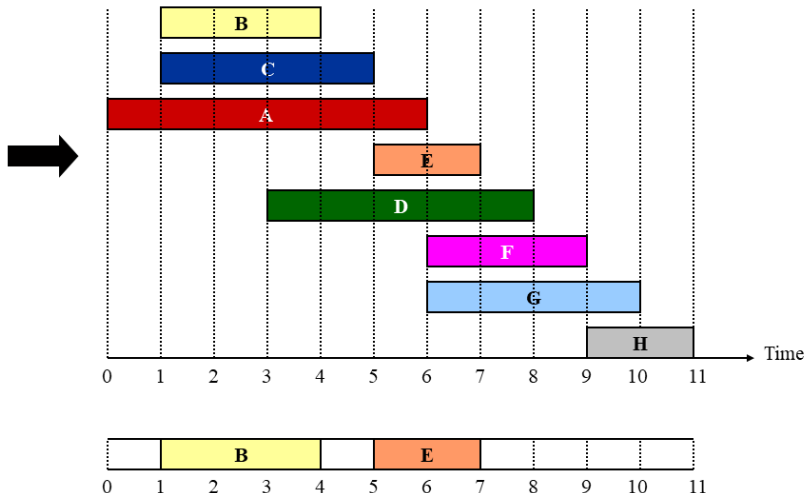
## Demo



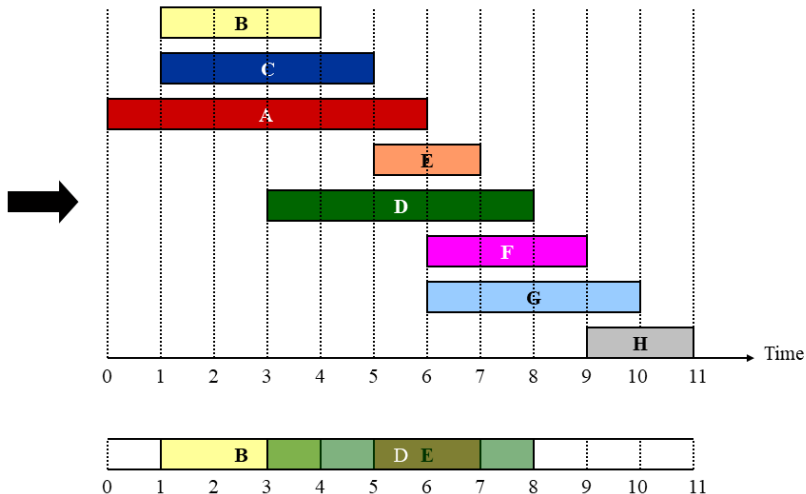
## Demo



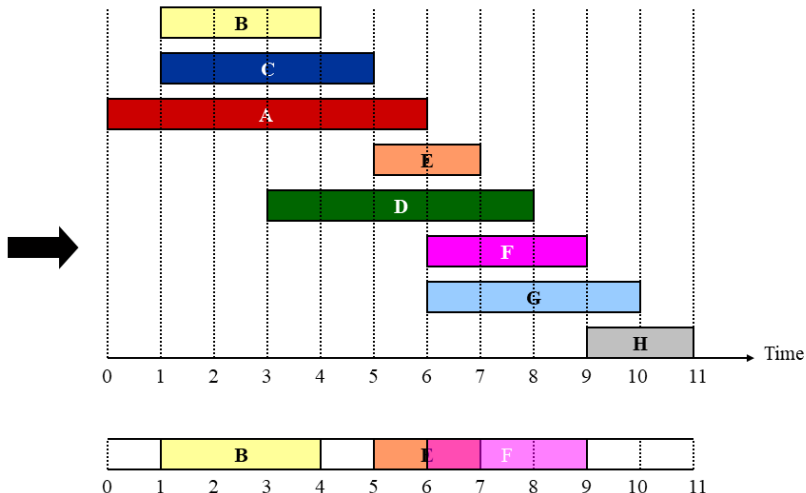
## Demo



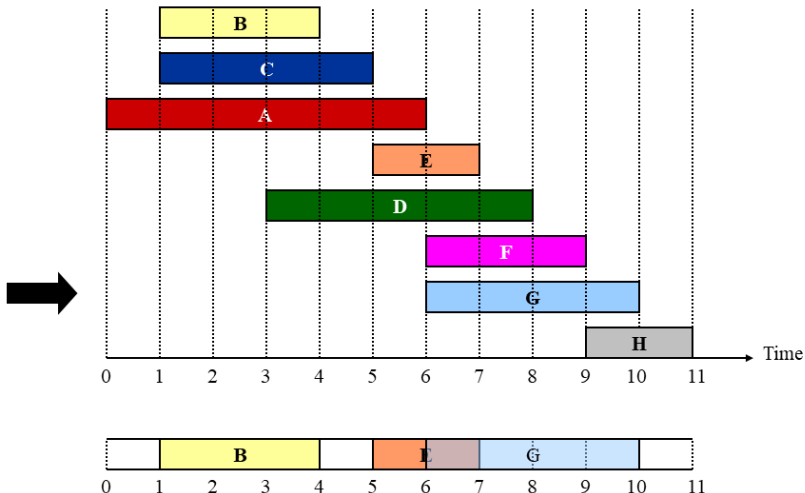
## Demo



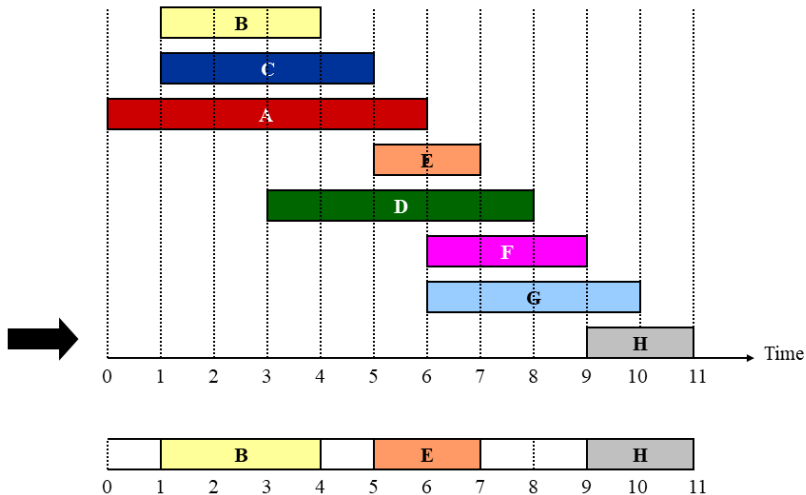
## Demo



# Demo



## Demo



# Notation

**Greedy Solution:**  $\{B, E, H\}$

**Optimal Solutions:** (not necessarily unique)

$\{A, F, H\}, \{B, E, H\}, \{B, F, H\}, \{C, E, H\}, \{C, F, H\}$

**Feasible Solutions:** (can work, but may not be the best)

$\emptyset, \{A\}, \{B\}, \{C\}, \{D\}, \{E\}, \{F\}, \{G\}, \{H\};$   
 $\{A, F\}, \{A, G\}, \{A, H\}, \{B, E\}, \{B, F\}, \{B, G\}, \{B, H\},$   
 $\{C, E\}, \{C, F\}, \{C, G\}, \{C, H\}, \{D, H\}, \{E, H\}, \{F, H\};$   
 $\{A, F, H\}, \{B, E, H\}, \{B, F, H\}, \{C, E, H\}, \{C, F, H\}.$



# Correctness Analysis

**Theorem.** Greedy Interval Scheduling algorithm is optimal.

# Correctness Analysis

**Theorem.** Greedy Interval Scheduling algorithm is optimal.

**Proof.** (by contradiction) Assume greedy is not optimal.

# Correctness Analysis

**Theorem.** Greedy Interval Scheduling algorithm is optimal.

**Proof.** (by contradiction) Assume greedy is not optimal.

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

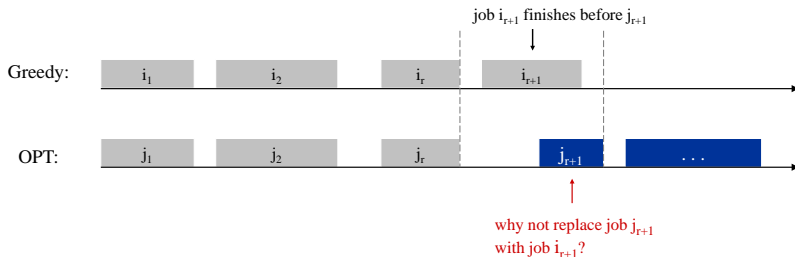
# Correctness Analysis

**Theorem.** Greedy Interval Scheduling algorithm is optimal.

**Proof.** (by contradiction) Assume greedy is not optimal.

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



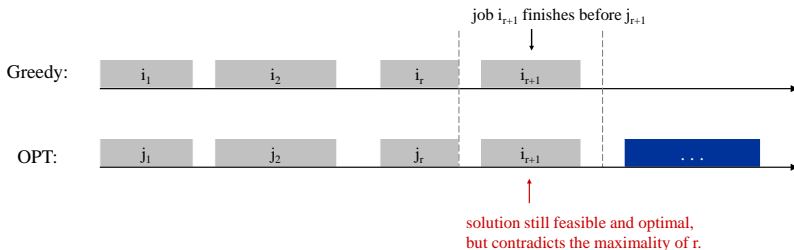
# Correctness Analysis

**Theorem.** Greedy Interval Scheduling algorithm is optimal.

**Proof.** (by contradiction) Assume greedy is not optimal.

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



# Outline

- 1 Basic Methodology
  - Interval Scheduling
  - Interval Partitioning
  - Scheduling to Minimize Lateness
  
- 2 More Examples
  - Optimal Caching
  - Coin Changing

# Interval Partitioning

Lecture  $j$  starts at  $s_j$  and finishes at  $f_j$ .

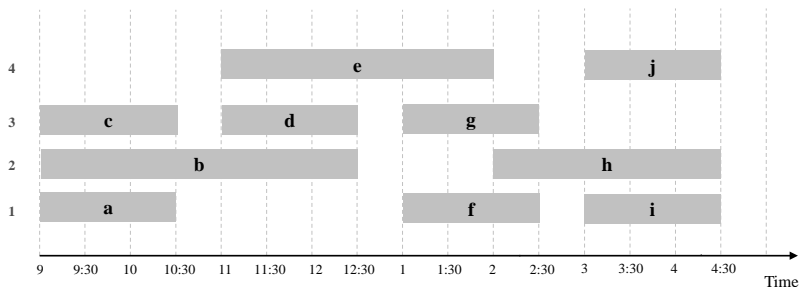
**Goal:** find minimum number of classrooms to schedule all lectures so that no two occur at the same time in the same room.

# 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.

**Example:** This schedule uses 4 classrooms to schedule 10 lectures.



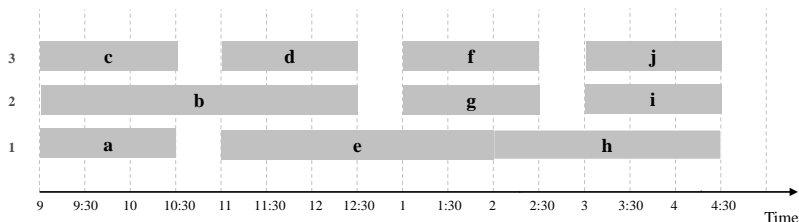


# Interval Partitioning

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.

**Example:** This schedule uses only 3.



# Lower Bound on Optimal Solution

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

# Lower Bound on Optimal Solution

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

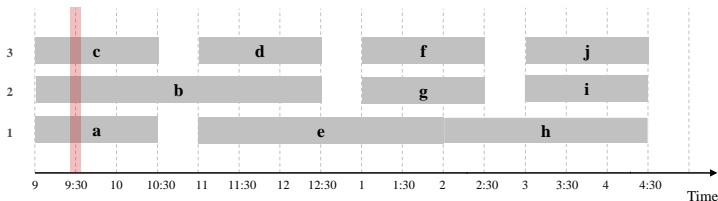
**Key observation.** Number of classrooms needed  $\geq$  depth.

# Lower Bound on Optimal Solution

**Definition:** 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.

**Example:** Depth of schedule = 3  $\Rightarrow$  The schedule is optimal.

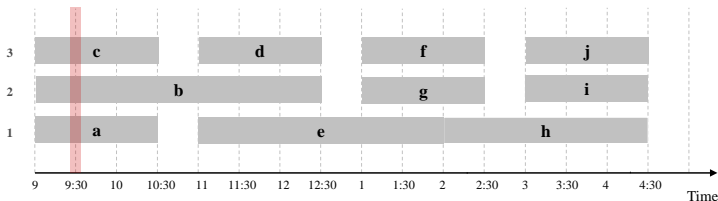


# Lower Bound on Optimal Solution

**Definition:** 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.

**Example:** Depth of schedule = 3  $\Rightarrow$  The schedule is optimal.



**Question.** Does there always exist a schedule equal to depth of intervals?

# Greedy Interval Partitioning Algorithm

---

**Algorithm 2:** Interval Partitioning Greedy Algorithm

---

```
1 Sort intervals by starting time so that  $s_1 \leq s_2 \leq \dots \leq s_n$ ;  
2  $d \leftarrow 0$ ; // number of allocated classrooms  
3 for  $j = 1$  to  $n$  do  
4   if lecture  $j$  is compatible with some classroom  $k$  then  
5     schedule lecture  $j$  in classroom  $k$ ;  
6   else  
7     allocate a new classroom  $d + 1$ ;  
8     schedule lecture  $j$  in classroom  $d + 1$ ;  
9      $d \leftarrow d + 1$ ;  
10 return  $A$ ;
```

---

# Greedy Interval Partitioning Algorithm

---

**Algorithm 2:** Interval Partitioning Greedy Algorithm

---

```
1 Sort intervals by starting time so that  $s_1 \leq s_2 \leq \dots \leq s_n$ ;  
2  $d \leftarrow 0$ ; // number of allocated classrooms  
3 for  $j = 1$  to  $n$  do  
4   if lecture  $j$  is compatible with some classroom  $k$  then  
5     schedule lecture  $j$  in classroom  $k$ ;  
6   else  
7     allocate a new classroom  $d + 1$ ;  
8     schedule lecture  $j$  in classroom  $d + 1$ ;  
9      $d \leftarrow d + 1$ ;  
10 return  $A$ ;
```

---

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

- For classroom  $k$ , maintain the finish time of the last job added.
- Keep the classrooms in a **priority queue**.

# Correctness Proof

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



# Correctness Proof

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

**Theorem.** Greedy algorithm is optimal.

# Correctness Proof

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

**Theorem.** Greedy algorithm is optimal.

**Proof.** Let  $d$  = number of classrooms that the algorithm allocates.

# Correctness Proof

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

**Theorem.** Greedy algorithm is optimal.

**Proof.** Let  $d$  = number of classrooms that the 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 each end after  $s_j$ .)

# Correctness Proof

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

**Theorem.** Greedy algorithm is optimal.

**Proof.** Let  $d$  = number of classrooms that the 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 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  $s_j + \varepsilon$ . □

# Correctness Proof

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

**Theorem.** Greedy algorithm is optimal.

**Proof.** Let  $d$  = number of classrooms that the 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 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  $s_j + \varepsilon$ . □

Key observation  $\Rightarrow$  all schedules use  $\geq d$  classrooms.

# Outline

- 1 Basic Methodology
  - Interval Scheduling
  - Interval Partitioning
  - Scheduling to Minimize Lateness
- 2 More Examples
  - Optimal Caching
  - Coin Changing

# Scheduling to Minimize Lateness

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

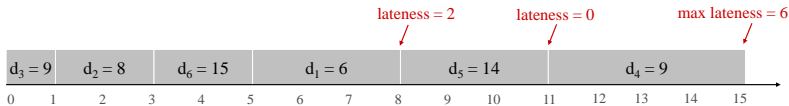
**Goal:** schedule all jobs to minimize maximum lateness  $L = \max l_j$ .

# Scheduling to Minimize Lateness

- 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  $L = \max l_j$ .

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





Attempt: Consider jobs in ascending order by some strategy

[Shortest processing time first] Sort by processing time  $t_j$ .

[Earliest deadline first] Sort by deadline  $d_j$ .

[Smallest slack] Sort by slack  $d_j - t_j$ .

# Attempt: Consider jobs in ascending order by some strategy

[Shortest processing time first] Sort by processing time  $t_j$ .

counterexample

	1	2
$t_j$	1	10
$d_j$	100	10

[Earliest deadline first] Sort by deadline  $d_j$ .

[Smallest slack] Sort by slack  $d_j - t_j$ .

# Attempt: Consider jobs in ascending order by some strategy

[Shortest processing time first] Sort by processing time  $t_j$ .

counterexample

	1	2
$t_j$	1	10
$d_j$	100	10

[Earliest deadline first] Sort by deadline  $d_j$ .

[Smallest slack] Sort by slack  $d_i - t_i$ .

counterexample

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

# A Greedy Algorithm: Earliest Deadline First

---

**Algorithm 3:** Greedy Minimizing Lateness

---

```
1 Sort  $n$  jobs by deadline so that  $d_1 \leq d_2 \leq \dots \leq d_n$ ;  
2  $t \leftarrow 0$ ;  
3 for  $j = 1$  to  $n$  do  
4   Assign job  $j$  to interval  $[t, t + t_j]$ ;  
5    $s_j \leftarrow t, f_j \leftarrow t + t_j$ ;  
6    $t \leftarrow t + t_j$ ;  
7 return intervals  $[s_j, f_j]$ ;
```

---

# A Greedy Algorithm: Earliest Deadline First

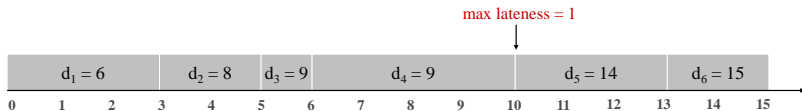
---

**Algorithm 3:** Greedy Minimizing Lateness

---

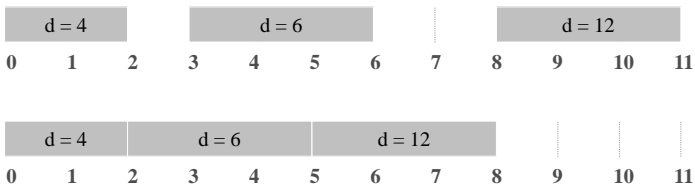
- 1 Sort  $n$  jobs by deadline so that  $d_1 \leq d_2 \leq \dots \leq d_n$ ;
  - 2  $t \leftarrow 0$ ;
  - 3 **for**  $j = 1$  **to**  $n$  **do**
  - 4     Assign job  $j$  to interval  $[t, t + t_j]$ ;
  - 5      $s_j \leftarrow t, f_j \leftarrow t + t_j$ ;
  - 6      $t \leftarrow t + t_j$ ;
  - 7 **return** intervals  $[s_j, f_j]$ ;
- 

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



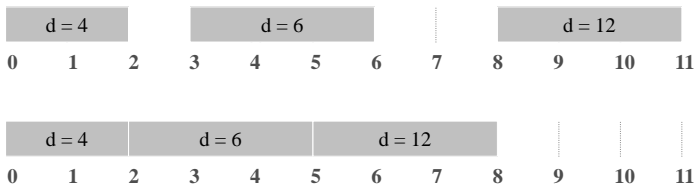
# Correctness Proof: Reduce Optimal Solution

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



# Correctness Proof: Reduce Optimal Solution

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



**Observation.** The greedy schedule has no idle time.

# Correctness Proof: Optimal Solution vs Algorithm Solution

**Definition.** 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  $d_1 \leq d_2 \leq \dots \leq d_n$  ]



# Correctness Proof: Optimal Solution vs Algorithm Solution

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



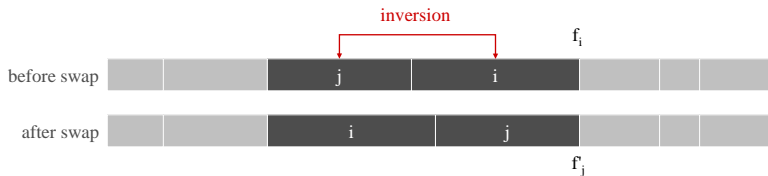
[ as before, we assume jobs are numbered so that  $d_1 \leq d_2 \leq \dots \leq d_n$  ]

**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**.

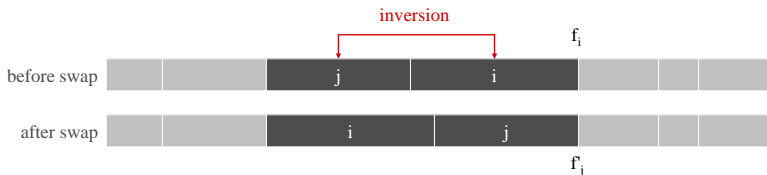
# Correctness Proof: Inversions

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



# Correctness Proof: Inversions

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

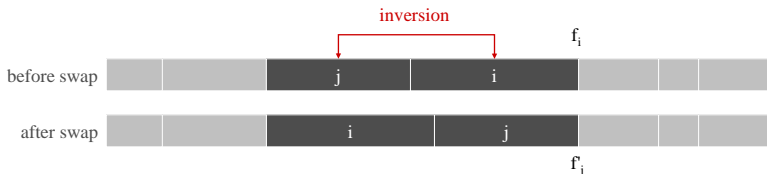


**Proof.** Let  $l$  be the lateness before the swap, and let  $l'$  be it afterwards.

- $l'_k = l_k$  for all  $k \neq i, j$
- $l'_i \leq l_i$
- If job  $j$  is late:

# Correctness Proof: Inversions

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



**Proof.** Let  $l$  be the lateness before the swap, and let  $l'$  be it afterwards.

- $l'_k = l_k$  for all  $k \neq i, j$
  - $l'_i \leq l_i$
  - If job  $j$  is late:
- $$\begin{aligned}
 l'_j &= f'_j - d_j && \text{(definition)} \\
 &= f_i - d_j && (j \text{ finishes at time } f_i) \\
 &\leq f_i - d_i && (i < j) \\
 &\leq l_i && \text{(definition)}
 \end{aligned}$$

# Correctness Proof

**Theorem.** Greedy schedule  $S$  is optimal.

# Correctness Proof

**Theorem.** Greedy schedule  $S$  is optimal.

**Proof.** Define  $S^*$  to be an optimal schedule that has the fewest number of inversions. (We can assume  $S^*$  has no idle time.)

# Correctness Proof

**Theorem.** Greedy schedule  $S$  is optimal.

**Proof.** Define  $S^*$  to be an optimal schedule that has the fewest number of inversions. (We can assume  $S^*$  has no idle time.)

- If  $S^*$  has no inversions, then  $S = S^*$ .

# Correctness Proof

**Theorem.** Greedy schedule  $S$  is optimal.

**Proof.** Define  $S^*$  to be an optimal schedule that has the fewest number of inversions. (We 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.



# Correctness Proof

**Theorem.** Greedy schedule  $S$  is optimal.

**Proof.** Define  $S^*$  to be an optimal schedule that has the fewest number of inversions. (We 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, ...

# Outline

- 1 Basic Methodology
  - Interval Scheduling
  - Interval Partitioning
  - Scheduling to Minimize Lateness
- 2 More Examples
  - Optimal Caching
  - Coin Changing

# Optimal Offline 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 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.

# Optimal Offline 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 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.

**Example.**  $k = 2$ , initial cache =  $ab$ ,

requests:  $a, b, c, b, c, a, a, b$ .

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

red = cache miss

# Optimal Offline 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 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.

**Example.**  $k = 2$ , 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

red = cache miss

# Optimal Strategy: Farthest-In-Future

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

# Optimal Strategy: 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	<b>f</b>	a	d	e	f	g	h	...
↑													↑								
cache miss													eject this one								



## Optimal Strategy: 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 an optimal eviction schedule.

**Proof.** Algorithm and theorem are intuitive; proof is subtle.

# Reduced Eviction Schedules

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

# Reduced Eviction Schedules

**Definition.** 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 (means no more cache insertion/replacement here).



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  $S$ , we can transform it into a reduced schedule  $S'$  with no more cache replacement (insertion).

---

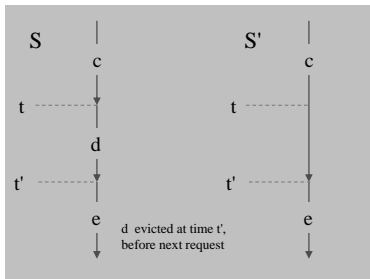
<sup>†</sup>doesn't enter cache at requested time

# Reduced Eviction Schedules

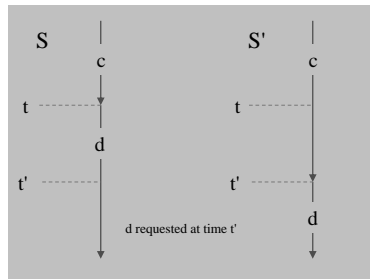
**Claim.** Given any unreduced schedule  $S$ , we can transform it into a reduced schedule  $S'$  with no more cache replacement (insertion).

**Proof.** (by induction on number of unreduced<sup>†</sup> items)

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



Case 2

<sup>†</sup>doesn't enter cache at requested time

# Theorem. FF is an optimal eviction algorithm

**Proof.** (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.

# Theorem. FF is an optimal eviction algorithm

**Proof.** (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.

# Theorem. FF is an optimal eviction algorithm

**Proof.** (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.

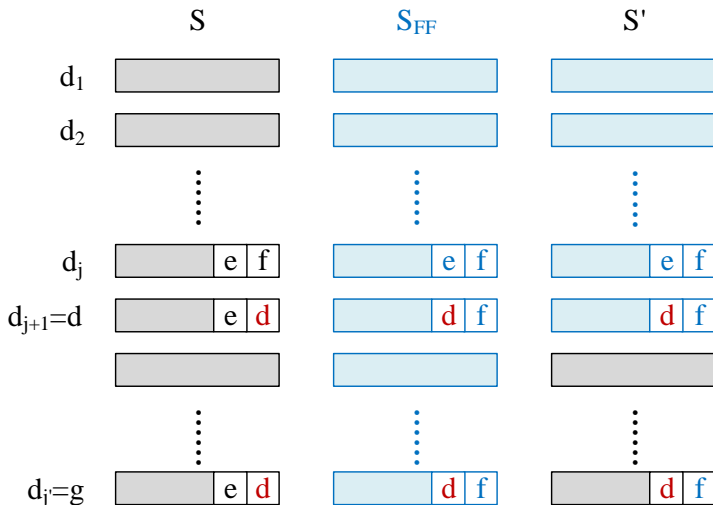
Consider  $(j + 1)^{th}$  request  $d = d_{j+1}$ . Since  $S$  and  $S_{FF}$  have agreed up until now, they have the same cache contents before request  $j + 1$ .

- Case 1: ( $d$  is already in the cache)  $S' = S$  satisfies invariant.
- Case 2: ( $d$  is not in the cache;  $S, S_{FF}$  evict same element)  $S' = S$  satisfies invariant.
- Case 3: ( $d$  is not in the cache;  $S_{FF}$  evicts  $e$ ;  $S$  evicts  $f \neq e$ )

Let  $S'$  agree with  $S_{FF}$  at the  $j + 1$  requests; we show that having element  $f$  in cache is no worse than having element  $e$ .



# An Illustration of Case 3

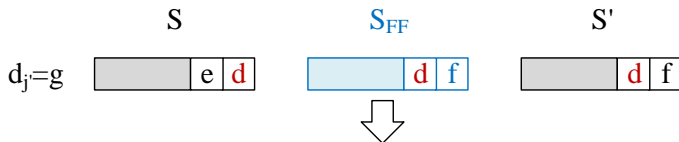


## Correctness Proof (Continued)

Let  $j'$  be the **first** time after  $j + 1$  that  $S$  and  $S'$  take a different action (must involve  $e$  or  $f$  or both), and let  $g$  be item requested at time  $j'$ .

# Correctness Proof (Continued)

Let  $j'$  be the **first** time after  $j + 1$  that  $S$  and  $S'$  take a different action (must involve  $e$  or  $f$  or both), and let  $g$  be item requested at time  $j'$ .



$g=e$

cannot happen due to FF strategy



## Correctness Proof (Continued)

**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.

**Case 3c:**  $g \neq e, f$ .  $S$  must evict  $e$  (otherwise  $S'$  would take the same action). Make  $S'$  evict  $f$ ; now  $S$  and  $S'$  have the same cache.  $\square$

# 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.

---

<sup>‡</sup>FF with direction of time reversed!

# 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<sup>‡</sup>.

---

<sup>‡</sup>FF with direction of time reversed!

# 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<sup>‡</sup>.

**Theorem.** FF is optimal offline eviction algorithm.

- Provides basis for understanding and analyzing online algorithms.
- LRU is  $k$ -competitive.** [Section 13.8 in Cornell Book]
- LIFO is arbitrarily bad.

---

<sup>‡</sup>FF with direction of time reversed!

# Outline

- 1 Basic Methodology
  - Interval Scheduling
  - Interval Partitioning
  - Scheduling to Minimize Lateness
- 2 More Examples
  - Optimal Caching
  - Coin Changing



# Coin Changing

**Goal.** Given US currency denominations:

1 (cent), 5 (nickel), 10 (dime), 25 (quarter), 100 (dollar),

devise a changing method using fewest number of coins.

# Coin Changing

**Goal.** Given US currency denominations:

1 (cent), 5 (nickel), 10 (dime), 25 (quarter), 100 (dollar),

devise a changing method using fewest number of coins.

**Example.** 34¢.



# Coin Changing

**Goal.** Given US currency denominations:

1 (cent), 5 (nickel), 10 (dime), 25 (quarter), 100 (dollar),

devise a changing method using fewest number of coins.

**Example.** 34¢.



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

# Coin Changing

**Goal.** Given US currency denominations:

1 (cent), 5 (nickel), 10 (dime), 25 (quarter), 100 (dollar),

devise a changing method using fewest number of coins.

**Example.** 34¢.



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

**Example.** \$2.89.



# Cashier's Algorithm

---

## Algorithm 4: Cashier's Algorithm

---

```
1 Sort coins denominations by value:  $c_1 < c_2 < \dots < c_n$ ;  
2  $S \leftarrow \emptyset$ ; // coins selected  
3 while  $x \neq 0$  do  
4   let  $k$  be largest integer such that  $c_k \leq x$ ;  
5   if  $k = 0$  then  
6     return "no solution found";  
7    $x \leftarrow x - c_k$ ;  
8    $S \leftarrow S \cup \{k\}$ ;  
9 return  $S$ ;
```

---

# Cashier's Algorithm

---

**Algorithm 4:** Cashier's Algorithm

---

```
1 Sort coins denominations by value:  $c_1 < c_2 < \dots < c_n$ ;  
2  $S \leftarrow \emptyset$ ; // coins selected  
3 while  $x \neq 0$  do  
4   let  $k$  be largest integer such that  $c_k < x$ ;  
5   if  $k = 0$  then  
6     return "no solution found";  
7    $x \leftarrow x - c_k$ ;  
8    $S \leftarrow S \cup \{k\}$ ;  
9 return  $S$ ;
```

---

**Question.** Is cashier's algorithm optimal?

# Properties of Optimal Solution

**Property.** Number of pennies  $\leq 4$ .

**Proof.** Replace 5 pennies with 1 nickel.

# Properties of Optimal Solution

**Property.** Number of pennies  $\leq 4$ .

**Proof.** Replace 5 pennies with 1 nickel.

**Property.** Number of nickels  $\leq 1$ .

**Property.** Number of quarters  $\leq 3$ .



# Properties of Optimal Solution

**Property.** Number of pennies  $\leq 4$ .

**Proof.** Replace 5 pennies with 1 nickel.

**Property.** Number of nickels  $\leq 1$ .

**Property.** Number of quarters  $\leq 3$ .

**Property.** Number of nickels + Number of dimes  $\leq 2$ .

**Proof.**

- Replace 3 dimes and 0 nickels with 1 quarter and 1 nickel;
- Replace 2 dimes and 1 nickel with 1 quarter.
- Recall: at most 1 nickel.



# Correctness Proof

**Theorem.** Greedy algorithm is optimal for U.S. coinage.

# Correctness Proof

**Theorem.** Greedy algorithm is optimal for U.S. coinage.

**Proof.** (by induction on  $x$ ) Consider an optimal way to change  $c_k < x < c_{k+1}$ : greedy takes coin  $k$ . We claim that any optimal solution must also take coin  $k$ .

# Correctness Proof

**Theorem.** Greedy algorithm is optimal for U.S. coinage.

**Proof.** (by induction on  $x$ ) Consider an optimal way to change  $c_k < x < c_{k+1}$ : 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  $x$ .

$k$	$c_k$	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	$75 + 24 = 99$

# Correctness Proof

**Theorem.** Greedy algorithm is optimal for U.S. coinage.

**Proof.** (by induction on  $x$ ) Consider an optimal way to change  $c_k < x < c_{k+1}$ : 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  $x$ .

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	$75 + 24 = 99$

Problem reduces to coin-changing  $x - c_k$  cents, which, by induction, is optimally solved by greedy algorithm.

# Is Cashier's Algorithm Work for Any Denominations?

**Observation 1.** Greedy is sub-optimal for US postal denominations:

1, 10, 21, 34, 70, 100, 350, 1225, 1500.

# Is Cashier's Algorithm Work for Any Denominations?

**Observation 1.** Greedy is sub-optimal for US postal denominations:

1, 10, 21, 34, 70, 100, 350, 1225, 1500.

**Counter example.** 140¢. (Greedy: 100, 34, six 1's; Optimal: 70, 70.)



# Is Cashier's Algorithm Work for Any Denominations?

**Observation 1.** Greedy is sub-optimal for US postal denominations:

1, 10, 21, 34, 70, 100, 350, 1225, 1500.

**Counter example.** 140¢. (Greedy: 100, 34, six 1's; Optimal: 70, 70.)

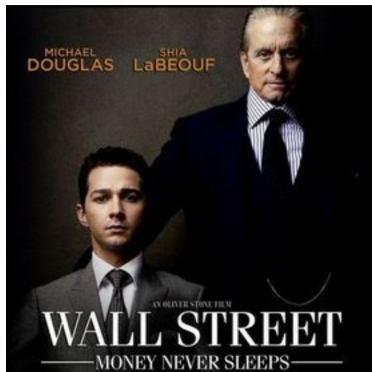


**Observation 2.** Even no feasible solution with system  $\mathcal{c}=\{7, 8, 9\}$ .

- Cashier's algorithm:  $15\text{¢} = 9 + ???$
- Optimal:  $15\text{¢} = 7 + 8$ .



## Movie: Wall Street (1987)



§

*Greed is good.*

*Greed is right.*

*Greed works.*

*Greed clarifies, cuts through, and captures the essence of the evolutionary spirit.*

— Gordon Gecko  
(Michael Douglas)

§ Watch the movie segment at the class webpage.