4. **Greedy Algorithms I**

- coin changing
- interval scheduling
- scheduling to minimize lateness
- optimal caching
4. Greedy Algorithms I

- coin changing
- interval scheduling
- scheduling to minimize lateness
- optimal caching
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.
Cashier's algorithm

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

**Cashier's Algorithm** ($x$, $c_1$, $c_2$, ..., $c_n$)

**Sort** $n$ coin denominations so that $c_1 < c_2 < ... < c_n$

$S \leftarrow \emptyset$  
set of coins selected

**While** $x > 0$

$k \leftarrow$ largest coin denomination $c_k$ such that $c_k \leq x$

**If** no such $k$, **Return** "no solution"

**Else**

$x \leftarrow x - c_k$

$S \leftarrow S \cup \{ k \}$

**Return** $S$

**Q.** Is cashier's algorithm optimal?
Properties of optimal solution

**Property.** Number of pennies ≤ 4.

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

**Property.** Number of nickels ≤ 1.

**Property.** Number of quarters ≤ 3.

**Property.** Number of nickels + number of dimes ≤ 2.

**Pf.**
- 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.
Analysis of cashier's algorithm

Theorem. Cashier's algorithm is optimal for U.S. coins: 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 \( c_1, \ldots, c_{k-1} \) 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 cashier's algorithm.

<table>
<thead>
<tr>
<th>( k )</th>
<th>( c_k )</th>
<th>all optimal solutions must satisfy</th>
<th>max value of coins ( c_1, c_2, \ldots, c_{k-1} ) in any OPT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>( P \leq 4 )</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>( N \leq 1 )</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>( N + D \leq 2 )</td>
<td>( 4 + 5 = 9 )</td>
</tr>
<tr>
<td>4</td>
<td>25</td>
<td>( Q \leq 3 )</td>
<td>( 20 + 4 = 24 )</td>
</tr>
<tr>
<td>5</td>
<td>100</td>
<td>no limit</td>
<td>( 75 + 24 = 99 )</td>
</tr>
</tbody>
</table>
Cashier's algorithm for other denominations

Q. Is cashier's algorithm for any set of denominations?

A. No. Consider U.S. postage: 1, 10, 21, 34, 70, 100, 350, 1225, 1500.
   • Cashier's algorithm: $140\$ = 100 + 34 + 1 + 1 + 1 + 1 + 1 + 1$.
   • Optimal: $140\$ = 70 + 70.$

A. No. It may not even lead to a feasible solution if $c_1 > 1$: 7, 8, 9.
   • Cashier's algorithm: $15\$ = 9 + ???$.
   • Optimal: $15\$ = 7 + 8$. 
4. Greedy Algorithms I

- coin changing
- interval scheduling
- scheduling to minimize lateness
- optimal caching

SECTION 4.1
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.

$$S = \{ J_1, \ldots, J_n \}$$

How many jobs in set $S$? $2^n$

jobs d and g are incompatible
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.

**counterexample for earliest start time**

**counterexample for shortest interval**

**counterexample for fewest conflicts**

\[ a \text{ e g h} \]
Interval scheduling: earliest-finish-time-first algorithm

**Earliest-Finish-Time-First** \((n, s_1, s_2, ..., s_n, f_1, f_2, ..., f_n)\)

**Sort** jobs by finish time so that \(f_1 \leq f_2 \leq ... \leq f_n\)

\(A \leftarrow \emptyset\) \hspace{1cm} \text{set of jobs selected}

\[\text{FOR } j = 1 \text{ TO } n\]

\[\text{IF job } j \text{ is compatible with } A\]

\[A \leftarrow A \cup \{j\}\]

**RETURN** \(A\)

---

**Proposition.** Can implement earliest-finish-time first in \(O(n \log n)\) time.

- Keep track of job \(j^*\) that was added last to \(A\).
- Job \(j\) is compatible with \(A\) iff \(s_j \geq f_{j^*}\).
- Sorting by finish time takes \(O(n \log n)\) time.
Interval scheduling: analysis of earliest-finish-time-first algorithm

Theorem. The earliest-finish-time-first algorithm is optimal.

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

Why not replace job $j_{r+1}$ with job $i_{r+1}$?
Interval scheduling: analysis of earliest-finish-time-first algorithm

**Theorem.** The earliest-finish-time-first algorithm is optimal.

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

![Diagram](image.png)

- Job $i_{r+1}$ exists and finishes before $j_{r+1}$
- Solution still feasible and optimal (but contradicts maximality of $r$)
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 lectures occur at the same time in the same room.

Ex. This schedule uses 3 classrooms to schedule 10 lectures.
Interval partitioning: greedy algorithms

Greedy template. Consider lectures in some natural order. Assign each lecture to an available classroom (which one?); allocate a new classroom if none are available.

- [Earliest start time] Consider lectures in ascending order of $s_j$.
- [Earliest finish time] Consider lectures in ascending order of $f_j$.
- [Shortest interval] Consider lectures in ascending order of $f_j - s_j$.
- [Fewest conflicts] For each lecture $j$, count the number of conflicting lectures $c_j$. Schedule in ascending order of $c_j$. 
Interval partitioning: greedy algorithms

Greedy template. Consider lectures in some natural order. Assign each lecture to an available classroom (which one?); allocate a new classroom if none are available.

**counterexample for earliest finish time**

3
2
1

**counterexample for shortest interval**

3
2
1

**counterexample for fewest conflicts**

3
2
1
Interval partitioning: earliest-start-time-first algorithm

**Earliest-Start-Time-First** \( (n, s_1, s_2, \ldots, s_n, f_1, f_2, \ldots, f_n) \)

Sort lectures by start time so that \( s_1 \leq s_2 \leq \ldots \leq s_n \).

\( d \leftarrow 0 \quad \text{number of allocated classrooms} \)

For \( j = 1 \) to \( n \)

If lecture \( j \) is compatible with some classroom

Schedule lecture \( j \) in any such classroom \( k \).

Else

Allocate a new classroom \( d + 1 \).

Schedule lecture \( j \) in classroom \( d + 1 \).

\( d \leftarrow d + 1 \)

Return schedule.
Interval partitioning: earliest-start-time-first algorithm

Proposition. The earliest-start-time-first algorithm can be implemented in $O(n \log n)$ time.

Pf. Store classrooms in a priority queue (key = finish time of its last lecture).
- To determine whether lecture $j$ is compatible with some classroom, compare $s_j$ to key of min classroom $k$ in priority queue.
- To add lecture $j$ to classroom $k$, increase key of classroom $k$ to $f_j$.
- Total number of priority queue operations is $O(n)$.
- Sorting by start time takes $O(n \log n)$ time. •

Remark. This implementation chooses the classroom $k$ whose finish time of its last lecture is the earliest.
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.

**Q.** Does number of classrooms needed always equal depth?

**A.** Yes! Moreover, earliest-start-time-first algorithm finds one.
Interval partitioning: analysis of earliest-start-time-first algorithm

Observation. The earliest-start-time first algorithm never schedules two incompatible lectures in the same classroom.

Theorem. Earliest-start-time-first algorithm is optimal.

Pf.
- Let \( d \) = number of classrooms that the algorithm allocates.
- Classroom \( d \) is opened because we needed to schedule a lecture, say \( j \), that is incompatible with all \( d - 1 \) other classrooms.
- These \( d \) lectures 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. •
4. GREEDY ALGORITHMS I

- coin changing
- interval scheduling
- scheduling to minimize lateness
- optimal caching

Section 4.2
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 \( L = \max_j \ell_j \).
Minimizing lateness: greedy algorithms

**Greedy template.** Schedule jobs according to some natural order.

- [Shortest processing time first] Schedule jobs in ascending order of processing time $t_j$.

- [Earliest deadline first] Schedule jobs in ascending order of deadline $d_j$.

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

*EDF*
Minimizing lateness: greedy algorithms

Greedy template. Schedule jobs according to some natural order.

- [Shortest processing time first] Schedule jobs in ascending order of processing time $t_j$.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_j$</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>$d_j$</td>
<td>100</td>
<td>10</td>
</tr>
</tbody>
</table>

$\max l_j = 1$

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

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_j$</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>$d_j$</td>
<td>2</td>
<td>10</td>
</tr>
</tbody>
</table>

$\max l_j = 3$
Minimizing lateness: earliest deadline first

**Earliest-Deadline-First** \((n, t_1, t_2, \ldots, t_n, d_1, d_2, \ldots, d_n)\)

**Sort** \(n\) jobs 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 \([t, t + t_j]\).

\(s_j \leftarrow t; \quad f_j \leftarrow t + t_j\)

\(t \leftarrow t + t_j\)

**Return** intervals \([s_1, f_1], [s_2, f_2], \ldots, [s_n, f_n]\).

\(\text{max lateness} = 1\)

<table>
<thead>
<tr>
<th>(d_1 = 6)</th>
<th>(d_2 = 8)</th>
<th>(d_3 = 9)</th>
<th>(d_4 = 9)</th>
<th>(d_5 = 14)</th>
<th>(d_6 = 15)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>
Minimizing lateness: no idle time

Observation 1. There exists an optimal schedule with no idle time.

Observation 2. The earliest-deadline-first 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 $i$.

Observation 3. The earliest-deadline-first schedule has no inversions.

Observation 4. If a schedule (with no idle time) has an inversion, it has one with a pair of inverted jobs scheduled consecutively.

[ as before, we assume jobs are numbered so that $d_1 \leq d_2 \leq \ldots \leq d_n$ ]
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 adjacent, 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, $\ell'_j = f'_j - d_j$ (definition)
  $= f_i - d_j$ (since $i$ and $j$ inverted)
  $\leq f_i - d_i$ (since $i$ and $j$ inverted)
  $\leq \ell_i$. (definition)
Minimizing lateness: analysis of earliest-deadline-first algorithm

**Theorem.** The earliest-deadline-first schedule \( S \) is optimal.

**Pf.** [by contradiction]

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 max lateness
  - strictly decreases the number of inversions
- This contradicts definition of \( S^* \)

**Claim:**
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.** Gale-Shapley, Kruskal, Prim, Dijkstra, Huffman, ...
4. Greedy Algorithms I

- coin changing
- interval scheduling
- scheduling to minimize lateness
- optimal caching

**memory hierarchy**
- cache memory
- register
- main memory
- disk
- remote disk

**Cache management**
- cost
- time to access
- access latency

**Middleman**
- $W_D$
- $W_L$

**Register**
- $10-1$

**Cut**
- $W_D$

**Cloud**
**Optimal offline caching**

**Caching.**
- Cache with capacity to store \( k \) items.
- Sequence of \( m \) item requests \( d_1, d_2, \ldots, d_m \).
- Cache hit: item already in cache when requested.
- Cache miss: item not already in cache when requested: must bring requested item into cache, and evict some existing item, if full.

**Goal.** Eviction schedule that minimizes number of evictions.

**Ex.** \( k = 2 \), initial cache = ab, requests: a, b, c, b, c, a, a.

**Optimal eviction schedule.** 2 evictions.
Optimal offline caching: greedy algorithms

**LIFO / FIFO.** Evict element brought in most (east) recently.

**LRU.** Evict element whose most recent access was earliest.

**LFU.** Evict element that was least frequently requested.

---

**previous queries**

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>a</th>
<th>w</th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>current cache</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

FIFO: eject a

<table>
<thead>
<tr>
<th></th>
<th>d</th>
<th>a</th>
<th>w</th>
<th>x</th>
<th>d</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>LRU: eject d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>b</th>
<th>a</th>
<th>b</th>
<th>x</th>
<th>d</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>c</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>e</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIFO: eject e</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**future queries**

<table>
<thead>
<tr>
<th></th>
<th>g</th>
<th>b</th>
<th>e</th>
<th>d</th>
<th></th>
</tr>
</thead>
</table>

**cache miss**

(which item to eject?)
Optimal offline caching: farthest-in-future (clairvoyant algorithm)

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

Theorem. [Bélády 1966] 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.

an unreduced schedule

![Diagram of unreduced schedule]

a reduced schedule

![Diagram of reduced schedule]
Reduced eviction schedules

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

Pf. [by induction on number of unreduced items]
- 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 $d$.

\[
\begin{array}{c|c|c|c|c|c|c|c|c|c|c|c}
\text{time} & \text{unreduced schedule } S & \text{Case 1} & \text{S'} \\
\hline
\text{t} & \text{c} & \text{c} & \text{c} & \text{c} & \text{c} & \text{c} & \text{c} & \text{c} & \text{c} & \text{c} & \text{c} \\
\hline
\text{t'} & \text{e} & \text{d} & \text{d} & \text{d} & \text{e} & \text{e} & \text{e} & \text{e} & \text{e} & \text{e} & \text{e} \\
\end{array}
\]

- $d$ enters cache without a request
- $d$ evicted before next request
- might as well leave $c$ in cache
Reduced eviction schedules

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

Pf. [by induction on number of unreduced items]

- 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. 

![Diagram](image)

- **unreduced schedule $S$**
  - $c$
  - $c$
  - $c$
  - $d$

- **Case 2**
  - $d$ enters cache without a request
  - $d$ requested before $d$ evicted

- **$S'$**
  - $c$
  - $c$
  - $c$
  - $d$

- **might as well leave $c$ in cache until $d$ is requested**
Farthest-in-future: analysis

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

**Pf.** Follows directly from invariant.

**Invariant.** There exists an optimal reduced schedule $S$ that makes the same eviction schedule as $S_{FF}$ through the first $j$ requests.

**Pf.** [by induction on $j$]

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 \(f\)

\[
\begin{array}{ccc|c|ccc}
S_{FF} & same & e & f & j & same & e & f \\
S & same & e & d & j+1 & same & d & f \\
S' & & & & & & & \\
\end{array}
\]

- 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 \(e\)

- let \(S'\) behave the same as \(S\) until \(S'\) is forced to take a different action (because either \(S\) evicts \(e\); or because either \(e\) or \(f\) is requested)
Farthest-in-future: analysis

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

![Diagram showing same, e, j', same, f]

- **Case 3a:** \( g = e \).
  Can't happen with FF 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 \( e' \neq e \), we make \( S' \) evict \( e' \) and brings \( e \) into the cache;
  now \( S \) and \( S' \) have the same cache
  We let \( S' \) behave exactly like \( S \) for remaining requests.

\( S' \) is no longer reduced, but can be transformed into a reduced schedule that agrees with SFF through step \( j + 1 \)
Farthest-in-future: analysis

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

\[
\begin{array}{c|c|c}
\text{same} & \text{e} & j' \\
S & & S' \\
\end{array}
\]

otherwise \( S' \) could have taken the same action.

- Case 3c: \( g \neq e, f \). \( S \) evicts \( e \).
  Make \( S' \) evict \( f \).

\[
\begin{array}{c|c|c}
\text{same} & g & j' \\
S & & S' \\
\end{array}
\]

Now \( S \) and \( S' \) have the same cache.
(and we let \( S' \) behave exactly like \( S \) for the remaining requests)
Caching perspective

**Online vs. offline algorithms.**
- Offline: full sequence of requests is known a priori.
- Online (reality): requests are not known in advance.
- Caching is among most fundamental online problems in CS.

**LIFO.** Evict page brought in most recently.

**LRU.** Evict page whose most recent access was earliest.

**Theorem.** FF is optimal offline eviction algorithm.
- Provides basis for understanding and analyzing online algorithms.
- LRU is $k$-competitive. [Section 13.8]
- LIFO is arbitrarily bad.