1. The offline problem

2. An optimal algorithm for the offline problem: Farthest-into-Future ($FF$)

3. Proof of optimality of $FF$

4. Online problem
1. Data Compression
   - Symbol codes
   - Optimal lossless compression and prefix codes
   - Trees and prefix codes

2. Greedy algorithms
   - A greedy algorithm for optimal lossless compression using symbol codes: the Huffman algorithm
Cache maintenance

- The offline problem
- An optimal algorithm for the offline problem: Farthest-in-Future (FF)
- Proof of optimality of FF
- The online problem
1. The offline problem

2. An optimal algorithm for the offline problem: Farthest-into-Future (FF)

3. Proof of optimality of FF

4. Online problem
Input

- \( n \), the number of pages in the main memory
- \( k \), the size of the cache memory
- a sequence of \( m \) requests for memory pages \( r_1, r_2, \ldots, r_m \)

Example:

- \( n = 3 \)
- \( k = 2 \)
- \( m = 7 \)
- sequence of requests: \( a, b, c, b, c, a, b \)
To service a request, the corresponding page must be in the cache.

⇒ After the first $k$ requests for distinct pages the cache is full.

- **Cache miss**: a request for a page that is not in the cache.
  - We must evict a page from the cache to bring in the requested page.

**Assumption**: a request is received and serviced within the same time step.
At each time step $1 \leq t \leq m$, we must decide which page (if any) to evict from the cache.

**Definition 1 (Scheduling algorithm).**

A schedule is a sequence of eviction decisions so that all $m$ requests are serviced at time $m$. An algorithm that provides such a schedule is a scheduling algorithm.

**Goal:** find the schedule that minimizes the total number of cache misses.
Example

- # pages in main memory: $n = 3$
- cache size: $k = 2$
- sequence of $m = 7$ requests: $a, b, c, b, c, a, b$

<table>
<thead>
<tr>
<th>time $t$:</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>requests:</td>
<td>$a$</td>
<td>$b$</td>
<td>$c$</td>
<td>$b$</td>
<td>$c$</td>
<td>$a$</td>
<td>$b$</td>
</tr>
</tbody>
</table>
Example

- # pages in main memory: \( n = 3 \)
- cache size: \( k = 2 \)
- sequence of \( m = 7 \) requests: \( a, b, c, b, c, a, b \)

<table>
<thead>
<tr>
<th>time ( t ):</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>requests:</td>
<td>( a, b, c, b, c, a, b )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>eviction schedule ( S ):</td>
<td>( -, - , a, - , - , c, - )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cache contents:</td>
<td>{a} {a, b} {b, c} {b, c} {b, c} {b, a} {b, a}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- "-" stands for "no eviction"
- \( S = \{ -, -, a, -, -, c, - \} \) evicts \( a \) at time 3, \( c \) at time 6
- \( S \) incurs 2 cache misses (can’t do better here)
Offline vs online problem

- **Offline** problem: the entire sequence of requests \( \{r_1, r_2, \ldots, r_m\} \) is part of the input (known at time \( t = 0 \))

- **Online** problem (more natural): requests arrive one at a time; \( r_t \) must be serviced at time \( t \), before future requests \( r_{t+1}, \ldots, r_m \) are seen

- A **scheduling algorithm for the online problem** can only base its eviction decision at time \( t \) on
  1. the requests it has seen so far
  2. the eviction decisions it has made so far

- The optimal offline algorithm helps to understand the online problem *(coming up)*
1. The offline problem

2. An optimal algorithm for the offline problem: Farthest-into-Future (FF)

3. Proof of optimality of FF

4. Online problem
**Definition 2 (Farthest-into-Future).**

*FF*: When the page requested at time $i$ is not in the cache, evict from the cache the page that is needed the farthest into the future and bring in the requested page.

**Notation:** we will denote the schedule produced by this algorithm $S_{FF}$.

*Why would $S_{FF}$ be optimal?*
Definition 3 (Reduced schedule).

A reduced schedule brings a page in the cache at time $t$ only if

1. the page is requested at time $t$; and
2. the page is not already in the cache.

Remark 1.

1. In a sense, a reduced schedule performs the least amount of work at every time step.
2. FF is a reduced schedule.
There is an optimal reduced schedule

**Fact 4.**

*We can transform a non-reduced schedule into a reduced one that is at least as good, that is, incurs at most the same number of evictions.*

**Remark 2.**

- The expensive memory operation is the eviction: even when no cache miss is incurred, we still count #evictions.
- In reduced schedules, #cache misses = # evictions.
- Given Fact 4, we can focus solely on reduced schedules.
Proof of Fact 4

- Let $S'$ be a schedule that is not reduced and solves an instance of cache maintenance.

- We will transform $S'$ into a reduced schedule $S$
  - Time $i$, request $r_i \neq a$:
    - if $S'$ evicts a page from the cache to bring in page $a$, not requested at time $i$
    - $S$ pretends it brings in $a$ but in fact does nothing
  - First time step $j > i$ such that $r_j = a$: $S$ brings in $a$
    $\Rightarrow$ charge the cache miss of $S'$ at time $j$ to the eviction of $S$ at the earlier time $i$

- Thus $S$ performs at most as many evictions as $S'$. 
1. The offline problem

2. An optimal algorithm for the offline problem: Farthest-into-Future (FF)

3. Proof of optimality of FF

4. Online problem
Optimality of FF

Claim 1.

Let \( S \) be a reduced schedule that makes the same eviction decisions as \( S_{FF} \) up to time \( t = i \), that is, up to request \( i \). Then there is a reduced schedule \( S' \) that

1. makes the same eviction decisions as \( S_{FF} \) up to time \( t = i + 1 \), that is, up to request \( i + 1 \);
2. the total number of cache misses it incurs is no more than that incurred by \( S \).

Proposition 1.

The schedule \( S_{FF} \) provided by the Farthest-into-Future algorithm is optimal.
Proof of Proposition 1: case $i = 0$

Notation
- $cm(S) =$ total #cache misses of schedule $S$
- $S^*$ is an optimal reduced schedule
- Schedule $S_1$ follows schedule $S_2$ up to request $i$ if $S_1$ makes the same eviction decisions as $S_2$ up to the $i$-th request

$i = 0$: trivially, $S^*$ follows $S_{FF}$ up to request $i = 0$. By Claim 2, we can construct a reduced schedule $S_1$ such that

1. $S_1$ follows $S_{FF}$ up to request $i = 1$
2. $cm(S_1) \leq cm(S^*)$. 
Proof of Proposition 1: case $i > 0$

- $i = 1$: now $S_1$ is a reduced schedule that follows $S_{FF}$ up to request $i = 1$. By Claim 2, we can construct a reduced schedule $S_2$ such that
  1. $S_2$ follows $S_{FF}$ up to request $i = 2$
  2. $cm(S_2) \leq cm(S_1)$.

- $i = 2$: now $S_2$ is a reduced schedule that follows $S_{FF}$ up to request $i = 2$. By Claim 2, we can construct a reduced schedule $S_3$ such that
  1. $S_3$ follows $S_{FF}$ up to request $i = 3$
  2. $cm(S_3) \leq cm(S_2)$. 
Proof of Proposition 1: $S_m = S_{FF}$

- Applying the claim for every $3 \leq i \leq m - 1$, we obtain the reduced schedule $S_m$ that
  1. follows $S_{FF}$ up to time $m$
  2. $cm(S_m) \leq cm(S_{m-1})$.

Tracing back all the inequalities, we obtain $cm(S_m) \leq cm(S^*)$.

- Finally, since $S_m$ follows $S_{FF}$ up to time $m$, $S_{FF} = S_m$.

Hence $cm(S_{FF}) = cm(S_m) \leq cm(S^*)$.

Thus $S_{FF}$ is optimal.
Claim 2.

Let \( S \) be a reduced schedule that makes the same eviction decisions as \( S_{FF} \) up to time \( t = i \), that is, up to request \( i \). Then there is a reduced schedule \( S' \) that

1. makes the same eviction decisions as \( S_{FF} \) up to time \( t = i + 1 \), that is, up to request \( i + 1 \);
2. incurs no more total cache misses than \( S \) does.

Proposition 2.

The schedule \( S_{FF} \) provided by the Farthest-into-Future algorithm is optimal.
Proof of Claim 2

Notation:

- $cm(S) = \text{total \#cache misses of schedule } S$
- $C_i(S) = \text{contents of the cache of schedule } S \text{ at time } i$

Since $S$ and $S_{FF}$ have made the same scheduling decisions up to time $i$, at the end of time step $i$:

- the contents of their caches are identical, hence

$$C_i(S) = C_i(S_{FF})$$

- so far, $S$ has the same number of cache misses as $S_{FF}$

Suppose that at time $i + 1$, page $p$ is requested, hence $r_{i+1} = p$. 
Proof of Claim 2, case 1: \( r_{i+1} = x \in C_i(S) \)

1. If \( x \in C_i(S) \)
   - \( x \in C_i(S_{FF}) \) since \( C_i(S) = C_i(S_{FF}) \)
   - no cache miss for either schedule

   ![Diagram](image)

   - Set \( S' = S \); then
     1. \( S' \) follows \( S_{FF} \) up to time \( i + 1 \) (\( S \) does!)
     2. \( cm(S') \leq cm(S) \).
Proof of Claim 2, case 2: $r_{i+1} = x \notin C_i(S)$

2. If $x \notin C_i(S)$
   - $x$ also not in $C_i(S_{FF})$ since $C_i(S) = C_i(S_{FF})$
   - both schedules must bring $x$ in, hence incur a cache miss

2.1: If $S$ and $S_{FF}$ both evict the same page $p$, set $S' = S$
   1. $S'$ follows $S_{FF}$ up to time $i + 1$ (since $S$ does)
   2. $cm(S') \leq cm(S)$.
Proof of Claim 2, case 2.2: $r_{i+1} = x \not\in C_i(S)$

2.2: If $S$ evicts $p$ but $S_{FF}$ evicts $q$:

- By construction of $S_{FF}$, $p$ must be requested later in the future than $q$.
- At the end of time step $i + 1$, the cache contents for the two schedules will differ in exactly one item.
Proof of Claim 2, case 2.2: $S$ evicts $p$, $S_{FF}$ evicts $q$

At the end of time step $i + 1$

- the cache of $S$ contains $q$
- the cache of $S_{FF}$ contains $p$
- the remaining $k - 1$ items in both caches are the same
- thus

\[ C_{i+1}(S_{FF}) = C_{i+1}(S) - \{q\} + \{p\}. \]

- Since we want $S'$ to agree with $S_{FF}$ up to time $i + 1$, $S'$
evicts $\sigma$ from its cache as well. Hence

\[ C_{i+1}(S') = C_{i+1}(S_{FF}) = C_{i+1}(S) - \{q\} + \{p\}. \]
Roadmap for case 2.2: $S$ evicts $p$, $S_{FF}$ evicts $q$

- **At the end of time step $i + 1**
  - the cache contents of $S, S'$ differ in exactly one item
  - $\#\text{cache misses of } S = \#\text{cache misses of } S'$

- **Want to ensure that $S'$ will not incur more misses than $S$ for $i + 1 < t \leq m$.**

- **Idea:** set $S' = S$ as soon as the cache contents of $S, S'$ are the same again.

  ⇒ **Goal:** make $C_t(S')$ equal $C_t(S)$ for the earliest $t > i + 1$ possible, while not incurring unnecessary misses.

  - Once $C_t(S') = C_t(S)$, set $S' = S$; if $S'$ has not incurred more misses than $S$ between steps $i + 2$ and $t$, then $cm(S') \leq cm(S)$. 
Case 2.2.1: $r_t = x \not\in \{p, q\}$, $x \not\in C_t(S)$, $S$ evicts $q$

For all $t > i + 1$, $S'$ follows $S$ until one of the following happens for the first time:

2.2.1: $r_t = y \not\in \{p, q\}$, and $y \not\in C_t(S)$, and $S$ evicts $q$.

Since $C_t(S')$ and $C_t(S')$ only differ in $p, q$, then $y \not\in C_t(S')$.
Set $S'$ to evict $p$ and bring in $y$. Then $C_t(S') = C_t(S)$!

Set $S' = S$ henceforth: $S'$ follows $S_{FF}$ up to time $i + 1$ and $cm(S') \leq cm(S)$. 

Case 2.2.2.1: \( r_t = p \), \( S \) evicts \( q \)

2.2.2: \( r_t = p \)

2.2.2.1: If \( S \) evicts \( q \), \( C_t(S) = C_t(S')! \)

Set \( S' = S \) henceforth: \( S' \) follows \( S_{FF} \) up to time \( i + 1 \) and \( cm(S') < cm(S) \).
Case 2.2.2.2: \( r_t = p, \) \( S \) evicts \( y \neq q \)

2.2.2: \( r_t = p \)

2.2.2.2: If \( S \) evicts \( y \neq q \) from its cache, then \( S' \) evicts \( y \) as well and brings in \( q \). Then \( C_t(S') = C_t(S) \).

Set \( S' = S \) henceforth: \( S' \) follows \( S_{FF} \) up to time \( i + 1 \) and \( cm(S') < cm(S) \).
2.2.2.2: resolving the final issue (we’re not done yet!)

- $S'$ is no longer **reduced**: $\sigma$ was brought in when there was no request for $\sigma$ at time $t$ (recall that $r_t = q$).

- Fortunately, we can use Fact 1 to transform $S'$ into a reduced schedule $\overline{S}$ that
  - incurs at most the same number of evictions as $S'$
  - still follows $S_{FF}$ up to time $i + 1$: all the real evictions of the reduced $S'$ will happen **after** time $i + 1$!

- Hence we return $\overline{S}$ as the schedule that satisfies Claim 2.
2.2.3: \( r_t = p \)

Can’t happen! \( S_{FF} \) evicted \( q \) and not \( p \), hence

- \( p \) appears farther in the future than \( q \)
- one of cases i., ii. will happen first
1. The offline problem

2. An optimal algorithm for the offline problem: Farthest-into-Future ($FF'$)

3. Proof of optimality of $FF$

4. Online problem
The online problem

- **Offline** problem: the entire sequence of requests \( \{r_1, r_2, \ldots, r_m\} \) is part of the input (known at time \( t = 0 \))

- **Online** problem (more natural): requests arrive one at a time; \( r_t \) must be serviced at time \( t \), before \( r_{t+1}, \ldots, r_m \) are seen

- An **online scheduling algorithm** can only base its eviction decision at time \( t \) on
  1. the requests it has seen so far
  2. the eviction decisions it has made so far

- The optimal offline algorithm helps to understand the online problem.
The Least Recently Used principle

- The **Least Recently Used (LRU)** principle: evict the page that was requested the **longest ago**

- **Intuition**: a running program will generally keep accessing the things it’s just been accessing (**locality of reference**)

- Essentially Farthest-into-Future (**FF**) reversed in time.

- LRU behaves well on average inputs.

- However an adversary can devise a specific sequence of online requests that will cause LRU to perform very badly compared to the optimal offline algorithm (**how?**).
Worst-case input to LRU

Example

- #pages in main memory: \( n = 3 \)
- size of the cache: \( k = 2 \)
- sequence of online requests

\[
\begin{align*}
\text{\{a, b, c\}} & \quad \text{\{a, b, c\}} & \quad \ldots & \quad \text{\{a, b, c\}} \\
1 & 2 & \ldots & k
\end{align*}
\]

\( \Rightarrow \) LRU: every request starting at time \( t = 3 \) is a miss, hence \( 7 = 3k - 2 \) misses

\( \Rightarrow \) \( FF \): only \( 4 = (3k - 1)/2 \) misses
Average-case performance of LRU

- Experimentally the best scheduling algorithms for the online problem are variants of LRU

- Competitive ratio: the worst-case ratio between the performance of the online algorithm over the performance of the optimal offline algorithm.