COMP 7720 - Online Algorithms

Caching (Paging) Problem

Shahin Kamali

University of Manitoba
There are two types of memory: a fast ‘cache’ of size $k$, and a slow memory of unbounded size.

- The input is an online sequence of requests to pages of size 1.
- To serve a request to page $x$, it should be in the cache.
  - In case $x$ is not in the cache, a fault of cost 1 happens.
  - In case $x$ is in the cache, a hit of cost 0 happens.
- The goal is to minimize the total number of faults.
- To bring $x$ to the cache, we might need to evict a page.
  - A caching algorithm is defined through its eviction policy.
Problem Definition

- There are two types of memory: a fast ‘cache’ of size $k$, and a slow memory of unbounded size.
  - The input is an online sequence of requests to pages of size 1.
- To serve a request to page $x$, it should be in the cache.
  - In case $x$ is not in the cache, a fault of cost 1 happens.
  - In case $x$ is in the cache, a hit of cost 0 happens.
  - The goal is to minimize the total number of faults.
- To bring $x$ to the cache, we might need to evict a page.
  - A caching algorithm is defined through its eviction policy.

Cost (number of faults): 0

\[
\sigma = \begin{bmatrix}
\_ & \_ & \_ & \_ \\
\end{bmatrix}
\]
Problem Definition

There are two types of memory: a fast ‘cache’ of size $k$, and a slow memory of unbounded size

- The input is an online sequence of requests to pages of size 1.

To serve a request to page $x$, it should be in the cache

- In case $x$ is not in the cache, a fault of cost 1 happens
- In case $x$ is in the cache, a hit of cost 0 happens
- The goal is to minimize the total number of faults

To bring $x$ to the cache, we might need to evict a page.

- A caching algorithm is defined through its eviction policy

Cost (number of faults): 1

\[
\sigma = a
\]

\[
\begin{array}{ccc}
\text{a} & & \\
\end{array}
\]
Problem Definition

There are two types of memory: a fast ‘cache’ of size $k$, and a slow memory of unbounded size.

- The input is an online sequence of requests to pages of size 1.
- To serve a request to page $x$, it should be in the cache.
  - In case $x$ is not in the cache, a fault of cost 1 happens.
  - In case $x$ is in the cache, a hit of cost 0 happens.
- The goal is to minimize the total number of faults.
- To bring $x$ to the cache, we might need to evict a page.

A caching algorithm is defined through its eviction policy.

Cost (number of faults): 2

$$\sigma = \begin{bmatrix} a & b \end{bmatrix}$$

\[
\begin{array}{|c|c|}
\hline
a & b \\
\hline
\end{array}
\]
Problem Definition

- There are two types of memory: a fast ‘cache’ of size $k$, and a slow memory of unbounded size.
  - The input is an online sequence of requests to pages of size 1.
- To serve a request to page $x$, it should be in the cache.
  - In case $x$ is not in the cache, a fault of cost 1 happens.
  - In case $x$ is in the cache, a hit of cost 0 happens.
- The goal is to minimize the total number of faults.
- To bring $x$ to the cache, we might need to evict a page.
  - A caching algorithm is defined through its eviction policy.

Cost (number of faults): 3

\[
\sigma = a \ b \ c
\]

\[
\begin{array}{|c|c|c|}
\hline
a & b & c \\
\hline
\end{array}
\]
Problem Definition

- There are two types of memory: a fast ‘cache’ of size $k$, and a slow memory of unbounded size.
  - The input is an online sequence of requests to pages of size 1.
- To serve a request to page $x$, it should be in the cache.
  - In case $x$ is not in the cache, a fault of cost 1 happens.
  - In case $x$ is in the cache, a hit of cost 0 happens.
- The goal is to minimize the total number of faults.
- To bring $x$ to the cache, we might need to evict a page.
  - A caching algorithm is defined through its eviction policy.

Cost (number of faults): 3

$$\sigma = a \ b \ c \ b$$

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
</table>
Problem Definition

- There are two types of memory: a fast ‘cache’ of size $k$, and a slow memory of unbounded size
  - The input is an online sequence of requests to pages of size 1.
- To serve a request to page $x$, it should be in the cache
  - In case $x$ is not in the cache, a fault of cost 1 happens
  - In case $x$ is in the cache, a hit of cost 0 happens
  - The goal is to minimize the total number of faults
- To bring $x$ to the cache, we might need to evict a page.
  - A caching algorithm is defined through its eviction policy

Cost (number of faults): 3

$$\sigma = a \ b \ c \ b \ a$$

| a | b | c |
---|---|---|
Problem Definition

- There are two types of memory: a fast ‘cache’ of size $k$, and a slow memory of unbounded size.
  - The input is an online sequence of requests to pages of size 1.
- To serve a request to page $x$, it should be in the cache.
  - In case $x$ is not in the cache, a fault of cost 1 happens.
  - In case $x$ is in the cache, a hit of cost 0 happens.
  - The goal is to minimize the total number of faults.
- To bring $x$ to the cache, we might need to evict a page.
  - A caching algorithm is defined through its eviction policy.

Cost (number of faults): 4

$\sigma = a \ b \ c \ b \ a \ d$

| a | b | c | d |
Problem Definition

- There are two types of memory: a fast ‘cache’ of size $k$, and a slow memory of unbounded size
  - The input is an online sequence of requests to pages of size 1.
- To serve a request to page $x$, it should be in the cache
  - In case $x$ is not in the cache, a fault of cost 1 happens
  - In case $x$ is in the cache, a hit of cost 0 happens
  - The goal is to minimize the total number of faults
- To bring $x$ to the cache, we might need to evict a page.
  - A caching algorithm is defined through its eviction policy

Cost (number of faults): 4

$\sigma = a \ b \ c \ b \ a \ d \ c$

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
</table>

COMP 7720 - Online Algorithms  Caching (Paging) Problem
Problem Definition

- There are two types of memory: a fast ‘cache’ of size $k$, and a slow memory of unbounded size.
  - The input is an online sequence of requests to pages of size 1.

- To serve a request to page $x$, it should be in the cache.
  - In case $x$ is not in the cache, a fault of cost 1 happens.
  - In case $x$ is in the cache, a hit of cost 0 happens.
  - The goal is to minimize the total number of faults.

- To bring $x$ to the cache, we might need to evict a page.
  - A caching algorithm is defined through its eviction policy.

Cost (number of faults): $\sigma = a \ b \ c \ b \ a \ d \ c \ e$

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
</tr>
</tbody>
</table>
Problem Definition

- There are two types of memory: a fast ‘cache’ of size \( k \), and a slow memory of unbounded size.
  - The input is an online sequence of requests to pages of size 1.
- To serve a request to page \( x \), it should be in the cache.
  - In case \( x \) is not in the cache, a fault of cost 1 happens.
  - In case \( x \) is in the cache, a hit of cost 0 happens.
- The goal is to minimize the total number of faults.
- To bring \( x \) to the cache, we might need to evict a page.
  - A caching algorithm is defined through its eviction policy.

Cost (number of faults): 5

\[
\sigma = a \ b \ c \ b \ a \ d \ c \ e
\]

| a | e | c | d |
Least-Recently-Used (LRU)

- LRU algorithm: if eviction is necessary, evict the least recently used item.

Cost (number of faults): 5

\[ \sigma = a \ b \ c \ b \ a \ d \ c \ e \]

```
  a  b  c  d
```
Least-Recently-Used (LRU)

- LRU algorithm: if eviction is necessary, evict the least recently used item.

Cost (number of faults): 5

$$\sigma = a \ b \ c \ b \ a \ d \ c \ e$$

\[
\begin{array}{cccc}
a & e & c & d \\
\end{array}
\]
LRU algorithm: if eviction is necessary, evict the least recently used item.

Cost (number of faults): 6

\[ \sigma = a \ b \ c \ b \ a \ d \ c \ e \ f \]
Least-Recently-Used (LRU)

- LRU algorithm: if eviction is necessary, evict the least recently used item.

Cost (number of faults): 6

$$\sigma = a\ b\ c\ b\ a\ d\ c\ e\ f$$

<table>
<thead>
<tr>
<th>f</th>
<th>e</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
</table>
Least-Recently-Used (LRU)

- LRU algorithm: if eviction is necessary, evict the least recently used item.

Cost (number of faults): 7

\[ \sigma = a \; b \; c \; b \; a \; d \; c \; e \; f \; a \]

\[
\begin{array}{cccc}
  f & e & c & d \\
\end{array}
\]
Least-Recently-Used (LRU)

- LRU algorithm: if eviction is necessary, evict the least recently used item.

Cost (number of faults): 7

\[ \sigma = a \ b \ c \ b \ a \ d \ c \ e \ f \ a \]

| f | e | c | a |
First-In-First-Out (FIFO)

FIFO algorithm: if eviction is necessary, evict the oldest page in the cache (the one that came earlier).

Cost (number of faults): 5

\[ \sigma = a \ b \ c \ b \ a \ d \ c \ e \]

\[ \begin{array}{cccc}
  a & b & c & d \\
\end{array} \]
First-In-First-Out (FIFO)

- FIFO algorithm: if eviction is necessary, evict the oldest page in the cache (the one that came earlier).

Cost (number of faults): 5

\[ \sigma = a \ b \ c \ b \ a \ d \ c \ e \]

| e | b | c | d |
First-In-First-Out (FIFO)

FIFO algorithm: if eviction is necessary, evict the oldest page in the cache (the one that came earlier).

Cost (number of faults): 6

\[ \sigma = a \ b \ c \ b \ a \ d \ c \ e \ f \]

| e | b | c | d |
First-In-First-Out (FIFO)

FIFO algorithm: if eviction is necessary, evict the oldest page in the cache (the one that came earlier).

Cost (number of faults): 6

\[ \sigma = a \ b \ c \ b \ a \ d \ c \ e \ f \]

| e | f | c | d |
First-In-First-Out (FIFO)

- FIFO algorithm: if eviction is necessary, evict the oldest page in the cache (the one that came earlier).

Cost (number of faults): 7

\[ \sigma = a \ b\ c\ b\ a\ d\ c\ e\ f\ a \]

| e | f | c | d |

COMP 7720 - Online Algorithms Caching (Paging) Problem
First-In-First-Out (FIFO)

- FIFO algorithm: if eviction is necessary, evict the oldest page in the cache (the one that came earlier).

Cost (number of faults): 7

\[ \sigma = a \ b \ c \ b \ a \ d \ c \ e \ f \ a \]

\[
\begin{array}{cccc}
e & f & a & d \\
\end{array}
\]
Flash-When-Full (FWF)

FWF algorithm: if eviction is necessary, evict all pages in the cache (flash).

Cost (number of faults): 5

\[ \sigma = a \ b \ c \ b \ a \ d \ c \ e \]

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
</table>
Flash-When-Full (FWF)

FWF algorithm: if eviction is necessary, evict all pages in the cache (flash).

Cost (number of faults): 5

$$\sigma = a \ b \ c \ b \ a \ d \ c \ e$$

**e** □ □ □
Flash-When-Full (FWF)

- FWF algorithm: if eviction is necessary, evict all pages in the cache (flash).

Cost (number of faults): 6

\[ \sigma = a \ b \ c \ b \ a \ d \ c \ e \ f \]

| e | f |   |   |
Flash-When-Full (FWF)

- FWF algorithm: if eviction is necessary, evict all pages in the cache (flash).

Cost (number of faults): 6

\[ \sigma = a \; b \; c \; b \; a \; d \; c \; e \; f \]

<table>
<thead>
<tr>
<th>e</th>
<th>f</th>
<th></th>
</tr>
</thead>
</table>
Flash-When-Full (FWF)

- FWF algorithm: if eviction is necessary, evict all pages in the cache (flash).

Cost (number of faults): \(7\)

\[\sigma = a \ b \ c \ b \ a \ d \ c \ e \ f \ a\]

\[\begin{array}{cc}
e & f \\
\end{array}\]
Flash-When-Full (FWF)

FWF algorithm: if eviction is necessary, evict all pages in the cache (flash).

Cost (number of faults): 7

\[ \sigma = a \quad b \quad c \quad b \quad a \quad d \quad c \quad e \quad f \quad a \]

\[
\begin{array}{ccc}
e & f & a \\
\end{array}
\]
An Offline Algorithm

- Furthest-In-Future: Evict the page whose next request is furthest in the future among all pages in the cache.

Cost (number of faults): 5

\[ \sigma = a \ b \ c \ b \ a \ d \ c \ e \ f \ a \ c \ d \ c \ f \ a \ b \ a \ e \]

| a | b | c | d |
An Offline Algorithm

- Furthest-In-Future: Evict the page whose next request is furthest in the future among all pages in the cache.

Cost (number of faults): 5

\[ \sigma = a \ b \ c \ b \ a \ d \ c \ e \ f \ a \ c \ d \ c \ f \ a \ b \ a \ e \]

<table>
<thead>
<tr>
<th>a</th>
<th>e</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
</table>
An Offline Algorithm

- Furthest-In-Future: Evict the page whose next request is furthest in the future among all pages in the cache.

Cost (number of faults): 6

\[ \sigma = a \ b \ c \ b \ a \ d \ c \ e \ f \ a \ c \ d \ c \ f \ a \ b \ a \ e \]

```
 a  e  c  d
```
An Offline Algorithm

- Furthest-In-Future: Evict the page whose next request is furthest in the future among all pages in the cache.

Cost (number of faults): 6

\[ \sigma = a \ b \ c \ b \ a \ d \ c \ e \ f \ a \ c \ d \ c \ f \ a \ b \ a \ e \]

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>f</td>
<td>c</td>
<td>d</td>
</tr>
</tbody>
</table>
**An Offline Algorithm**

- **Furthest-In-Future**: Evict the page whose next request is furthest in the future among all pages in the cache.

  Cost (number of faults): 6

  \[ \sigma = a \ b \ c \ b \ a \ d \ c \ e \ f \ a \ c \ d \ c \ f \ a \ b \ a \ e \]

  \[
  \begin{array}{cccc}
  a & f & c & d \\
  \end{array}
  \]
An Offline Algorithm

- Furthest-In-Future: Evict the page whose next request is furthest in the future among all pages in the cache.

Cost (number of faults): 6

$$\sigma = a \ b \ c \ b \ a \ d \ c \ e \ f \ a \ c \ d \ c \ f \ a \ b \ a \ e$$

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>f</td>
<td>c</td>
<td>d</td>
</tr>
</tbody>
</table>
An Offline Algorithm

Furthest-In-Future: Evict the page whose next request is furthest in the future among all pages in the cache.

Cost (number of faults): 6

\[ \sigma = a \ b \ c \ b \ a \ d \ c \ e \ f \ a \ c \ d \ c \ f \ a \ b \ a \ e \]

| a | f | c | d |
Theorem

**Furthest-In-Future (FIF) is the optimal offline algorithm for Caching.**

- Idea: we can modify any optimal algorithm $\text{Off}$ to work similar to FIF without increasing its cost.
- Assume on an access to $z$, $\text{Off}$ evicts $y$ while $x$ is furthest in future.
- Change $\text{Off}$ so that instead of $y$, $x$ is evicted.
  - We skip the details; a case analysis is required.
Theorem

For a cache of size $k$, no deterministic caching algorithm can have a competitive ratio better than $k$. 
Caching Algorithms & Competitive Ratio

Theorem

For a cache of size $k$, no deterministic caching algorithm can have a competitive ratio better than $k$.

Consider any online algorithm $A$

Create an adversarial sequence of length $n$ on $k + 1$ pages so that $A$ faults on every single request.

- At any time, at least one page is outside of the cache $\rightarrow$ adversary asks for that!
- The cost of $A$ will be $n$.

Question: The cost of FIF for any sequence formed by $k + 1$ pages is at most:

- (a) $O(k)$
- (b) $O(k^2)$
- (c) $n/k + O(k)$
- $O(n + k)$
Caching Algorithms & Competitive Ratio

Theorem

For a cache of size $k$, no deterministic caching algorithm can have a competitive ratio better than $k$.

- After the first $k$ “cold” misses, if FIF misses at one request, it hits in the next $k - 1$ requests.
  - Assume FIF evicts page $x$ for a request to $z$; so all $k + 1$ pages except $x$ are in the cache.
  - The next fault happens on a request to $x$.
  - But we know all $k - 1$ pages (all pages in the cache except potentially $z$) have been request before the next request to $x$.
  - In FIF, for each fault, there are at least $k - 1$ hits.
    - The cost of FIF is $k + (n - k)/k = n/k + O(k)$. 

COMP 7720 - Online Algorithms  Caching ( Paging ) Problem
**Theorem**

For a cache of size $k$, no deterministic caching algorithm can have a competitive ratio better than $k$.

- On an adversarial sequence of length $n$ on $k + 1$ pages:
  - A has a cost of $n$
  - FIF has a cost of at most $n/k + O(k)$
- The ratio between the cost of A and FIF is at least $k$, assuming $k \in o(n)$
Competitive Ratio of LRU

Theorem

LRU has a competitive ratio of at most $k$. 
Theorem

**LRU has a competitive ratio of at most $k$.**

- Use a **phase partitioning** technique.
- Define a phase as a sequence $\sigma_i, \sigma_{i+1}, \ldots, \sigma_{i+m}$ so that requests in this range involve $k$ different pages.
  - The next request $\sigma_{i+m+1}$ is different from all these $k$ requests.

$$\sigma = a\ b\ c\ b\ a\ d\ c\ e\ f\ a\ c\ d\ c\ d\ f\ a\ b\ a\ e\ \ldots \quad k = 4$$
Competitive Ratio of LRU

**Theorem**

*LRU has a competitive ratio of at most $k$.***

- Use a **phase partitioning** technique.
- Define a phase as a sequence $\sigma_i, \sigma_{i+1}, \ldots, \sigma_{i+m}$ so that requests in this range involve $k$ different pages.
  - The next request $\sigma_{i+m+1}$ is different from all these $k$ requests.

$$\sigma = \underbrace{a b c b a d c e f a c d c d f a b a e \ldots}_{\text{phase 1}} \quad k = 4$$
Competitive Ratio of LRU

**Theorem**

LRU has a competitive ratio of at most $k$.

- Use a *phase partitioning* technique.
- Define a phase as a sequence $\sigma_i, \sigma_{i+1}, \ldots, \sigma_{i+m}$ so that requests in this range involve $k$ different pages.
  - The next request $\sigma_{i+m+1}$ is different from all these $k$ requests.

$$\sigma = a\underbrace{b\ c\ b\ a\ d\ c}_{\text{phase 1}} \ e\ f\ a\ c\ d\ c\ d\ f\ a\ b\ a\ e\ \ldots \quad k = 4$$

COMP 7720 - Online Algorithms  Caching (Paging) Problem
Competitive Ratio of LRU

**Theorem**

*LRU has a competitive ratio of at most $k$.***

- Use a **phase partitioning** technique.
- Define a phase as a sequence $\sigma_i, \sigma_{i+1}, \ldots, \sigma_{i+m}$ so that requests in this range involve $k$ different pages.
  - The next request $\sigma_{i+m+1}$ is different from all these $k$ requests.

\[
\sigma = \underbrace{a \ b \ c \ b \ a \ d \ c}_{\text{phase 1}} \quad \underbrace{e \ f \ a \ c}_{\text{phase 2}} \quad \underbrace{d \ c \ d \ f \ a \ b \ a \ e \ldots}_{\text{phase 3}} \quad k = 4
\]
Theorem

LRU has a competitive ratio of at most $k$.

- What is the cost of LRU per phase?
  - $k$ different pages; LRU incurs at most $k$ faults

$$
\sigma = \underbrace{a\ b\ c\ b\ a\ d\ c}_\text{phase 1} \underbrace{e\ f\ a\ c}_\text{phase 2} \underbrace{d\ c\ d\ f\ a\ b\ a\ e}_\text{phase 3} \ldots $$

$k = 4$
Competitive Ratio of LRU

Theorem

LRU has a competitive ratio of at most $k$.

- What is the cost of LRU per phase?
  - $k$ different pages; LRU incurs at most $k$ faults

- What is the cost of OPT per phase?
  - Each phase + next item has $k + 1$ distinct pages
  - OPT has to pay a cost of 1 per phase!

$$\sigma = (a\ b\ c\ b\ a\ d\ c\ e\ f\ a\ c\ \underbrace{d\ c\ d\ f\ a\ b\ a\ e\ \ldots}_{k = 4})$$

phase1 phase2 phase3
Competitive Ratio of LRU

**Theorem**

*LRU has a competitive ratio of at most $k$.***

- The ratio between LRU and $OPT$ is at most $k$ per phase

\[
c.r.(LRU) = \frac{LRU(\text{phase 1}) + \ldots + LRU(\text{phase N})}{OPT(\text{phase 1}) + \ldots + OPT(\text{phase N})} \leq \max_i \frac{LRU(\text{phase i})}{OPT(\text{phase i})} \leq k
\]

$$\sigma = \underbrace{a b c b a d c}_\text{phase 1} \quad \underbrace{e f a c}_\text{phase 2} \quad \underbrace{d c d f a b a e \ldots}_\text{phase 3} \quad k = 4$$
Other Algorithms with C.R. $k$?

- In the proof, we just used the fact that LRU has a cost of at most $k$ for each phase.
  - For any subsequence formed by requests to $k$ pages, LRU incurs a cost of at most $k$. 
Other Algorithms with C.R. \( k \)?

- In the proof, we just used the fact that LRU has a cost of at most \( k \) for each phase.
  - For any subsequence formed by requests to \( k \) pages, LRU incurs a cost of at most \( k \)
- Can we extend this proof to other algorithms?
A marking algorithm maintains a bit (‘mark’) for each page in the cache.

- Start with all pages unmarked.
- Upon a hit, mark the page (if it is not already marked).
- Upon a fault, if eviction is required, evict an unmarked page.
  - If all pages in the cache are marked, all of them are unmarked first!
- When an item is brought to the cache it is marked

$$\sigma = a$$
Marking Family of Algorithms

- A marking algorithm maintains a bit (‘mark’) for each page in the cache.
  - Start with all pages unmarked.
  - Upon a hit, mark the page (if it is not already marked).
  - Upon a fault, if eviction is required, evict an unmarked page.
    - If all pages in the cache are marked, all of them are unmarked first!
  - When an item is brought to the cache it is marked

\[ \sigma = a \]
A marking algorithm maintains a bit (‘mark’) for each page in the cache.

- Start with all pages unmarked.
- Upon a hit, mark the page (if it is not already marked).
- Upon a fault, if eviction is required, evict an unmarked page.
  - If all pages in the cache are marked, all of them are unmarked first!
- When an item is brought to the cache it is marked

\[ \sigma = a \ b \]
Marking Family of Algorithms

A marking algorithm maintains a bit (‘mark’) for each page in the cache.

- Start with all pages unmarked.
- Upon a hit, mark the page (if it is not already marked).
- Upon a fault, if eviction is required, evict an unmarked page.
  - If all pages in the cache are marked, all of them are unmarked first!
- When an item is brought to the cache it is marked

\[ \sigma = a \ b \]

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
Marking Family of Algorithms

A marking algorithm maintains a bit ('mark') for each page in the cache.

- Start with all pages unmarked.
- Upon a hit, mark the page (if it is not already marked).
- Upon a fault, if eviction is required, evict an unmarked page.
  - If all pages in the cache are marked, all of them are unmarked first!
- When an item is brought to the cache it is marked

\[ \sigma = a \ b \ c \]

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>
Marking Family of Algorithms

- A marking algorithm maintains a bit (‘mark’) for each page in the cache.
  - Start with all pages unmarked.
  - Upon a hit, mark the page (if it is not already marked).
  - Upon a fault, if eviction is required, evict an unmarked page.
    - If all pages in the cache are marked, all of them are unmarked first!
  - When an item is brought to the cache it is marked

\[ \sigma = a \ b \ c \]

\[
\begin{array}{c|c|c}
 a & b & c \\
 \checkmark & \checkmark & \checkmark \\
\end{array}
\]
A marking algorithm maintains a bit (‘mark’) for each page in the cache.

- Start with all pages unmarked.
- Upon a hit, mark the page (if it is not already marked).
- Upon a fault, if eviction is required, evict an unmarked page.
  - If all pages in the cache are marked, all of them are unmarked first!
- When an item is brought to the cache it is marked

\[ \sigma = a \ b \ c \ d \]

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
A marking algorithm maintains a bit (‘mark’) for each page in the cache.

- Start with all pages unmarked.
- Upon a hit, mark the page (if it is not already marked).
- Upon a fault, if eviction is required, evict an unmarked page.
  - If all pages in the cache are marked, all of them are unmarked first!
- When an item is brought to the cache it is marked

\[ \sigma = a \ b \ c \ d \]

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
A marking algorithm maintains a bit (‘mark’) for each page in the cache.

- Start with all pages unmarked.
- Upon a hit, mark the page (if it is not already marked).
- Upon a fault, if eviction is required, evict an unmarked page.
  - If all pages in the cache are marked, all of them are unmarked first!
- When an item is brought to the cache it is marked

\[
\sigma = a \ b \ c \ b \ e
\]

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
A marking algorithm maintains a bit (‘mark’) for each page in the cache.

- Start with all pages unmarked.
- Upon a hit, mark the page (if it is not already marked).
- Upon a fault, if eviction is required, evict an unmarked page.
  - If all pages in the cache are marked, all of them are unmarked first!
- When an item is brought to the cache it is marked

\[ \sigma = a \ b \ c \ b \ e \]
Marking Family of Algorithms

- A marking algorithm maintains a bit (‘mark’) for each page in the cache.
  - Start with all pages unmarked.
  - Upon a hit, mark the page (if it is not already marked).
  - Upon a fault, if eviction is required, evict an unmarked page.
    - If all pages in the cache are marked, all of them are unmarked first!
  - When an item is brought to the cache it is marked

\[ \sigma = a \ b \ c \ b \ e \]

\[
\begin{array}{c|c|c|c}
 a & b & e & d \\
\hline
 a & b & \checkmark & d \\
\end{array}
\]
A marking algorithm maintains a bit (‘mark’) for each page in the cache.

- Start with all pages unmarked.
- Upon a hit, mark the page (if it is not already marked).
- Upon a fault, if eviction is required, evict an unmarked page.
  - If all pages in the cache are marked, all of them are unmarked first!
- When an item is brought to the cache it is marked

$$\sigma = a \ b \ c \ b \ e \ f$$

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>e</th>
<th>d</th>
</tr>
</thead>
</table>
Marking Family of Algorithms

- A marking algorithm maintains a bit (‘mark’) for each page in the cache.
  - Start with all pages unmarked.
  - Upon a hit, mark the page (if it is not already marked).
  - Upon a fault, if eviction is required, evict an unmarked page.
    - If all pages in the cache are marked, all of them are unmarked first!
  - When an item is brought to the cache it is marked

\[ \sigma = a \ b \ c \ b \ e \ f \]

<table>
<thead>
<tr>
<th></th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓</td>
<td>e</td>
</tr>
<tr>
<td></td>
<td>b</td>
</tr>
<tr>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>
Marking Family of Algorithms

- A marking algorithm maintains a bit (‘mark’) for each page in the cache.
  - Start with all pages unmarked.
  - Upon a hit, mark the page (if it is not already marked).
  - Upon a fault, if eviction is required, evict an unmarked page.
    - If all pages in the cache are marked, all of them are unmarked first!
  - When an item is brought to the cache it is marked

$$\sigma = a \ b \ c \ b \ e \ f \ d$$

<table>
<thead>
<tr>
<th>f</th>
<th>b</th>
<th>e</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>
A marking algorithm maintains a bit (‘mark’) for each page in the cache.

- Start with all pages unmarked.
- Upon a hit, mark the page (if it is not already marked).
- Upon a fault, if eviction is required, evict an unmarked page.
  - If all pages in the cache are marked, all of them are unmarked first!
- When an item is brought to the cache it is marked

$$\sigma = a \ b \ c \ b \ e \ f \ d$$

<table>
<thead>
<tr>
<th></th>
<th>f</th>
<th>b</th>
<th>e</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>
A marking algorithm maintains a bit (‘mark’) for each page in the cache.

- Start with all pages unmarked.
- Upon a hit, mark the page (if it is not already marked).
- Upon a fault, if eviction is required, evict an unmarked page.
  - If all pages in the cache are marked, all of them are unmarked first!
- When an item is brought to the cache it is marked

\[ \sigma = a \ b \ c \ b \ e \ f \ d \ a \]

\[
\begin{array}{cccc}
  f & b & e & d \\
  \checkmark & \checkmark & \checkmark & \checkmark \\
\end{array}
\]
A marking algorithm maintains a bit (‘mark’) for each page in the cache.

- Start with all pages unmarked.
- Upon a hit, mark the page (if it is not already marked).
- Upon a fault, if eviction is required, evict an unmarked page.
  - If all pages in the cache are marked, all of them are unmarked first!
- When an item is brought to the cache it is marked

\[ \sigma = a \ b \ c \ b \ e \ f \ d \ a \]

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>a</td>
<td>e</td>
<td>d</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
Marking Family of Algorithms (cntd.)

Theorem

Any deterministic marking algorithms $M$ has competitive ratio $k$.

- **What is the cost of $M$ per phase?**
  - It starts the phase with all pages unmarked
  - On the first request to $x$, it becomes marked
    - $x$ remains in the cache until the end of the phase
    - $M$ incurs a cost of 1 for $x$ throughout the phase

$$
\sigma = \underbrace{a \ b \ c \ b \ a \ d \ c}_{\text{phase1}} \ \underbrace{e \ f \ a \ c}_{\text{phase2}} \ \underbrace{d \ c \ d \ f \ a \ b \ a \ e \ \ldots}_{\text{phase3}} \ \ k = 4
$$
Theorem

Any deterministic marking algorithms $M$ has competitive ratio $k$.

- What is the cost of $M$ per phase?
  - It starts the phase with all pages unmarked
  - At the end of the phase, all $k$ pages of the phase are marked
  - On the first request to $x$, it becomes marked
    - $x$ remains in the cache until the end of the phase
    - $M$ incurs a cost of 1 for $x$ throughout the phase

\[ \sigma = a \ b \ c \ b \ a \ d \ c \ e \ f \ a \ c \ d \ c \ d \ f \ a \ b \ a \ e \ \ldots \quad k = 4 \]
Theorem

Any deterministic marking algorithms $M$ has competitive ratio $k$.

What is the cost of $M$ per phase?

- It starts the phase with all pages unmarked
- At the end of the phase, all $k$ pages of the phase are marked
- On the first request to $x$, it becomes marked
  - $x$ remains in the cache until the end of the phase
  - $M$ incurs a cost of 1 for $x$ throughout the phase
- For each phase, $M$ incurs a cost of at most $k$
- Recall that $OPT$ has to pay a cost of 1 per phase!

$$\sigma = a \ b \ c \ b \ a \ d \ c \ e \ f \ a \ c \ d \ c \ d \ f \ a \ b \ a \ e \ \ldots \quad k = 4$$

phase1   phase2   phase3
Marking Algorithms Remarks

- LRU and Flash-When-Full are marking algorithms
- They have competitive ratio $k$
Marking Algorithms Remarks

- LRU and Flash-When-Full are marking algorithms
  - They have competitive ratio $k$
- FIFO is Not a marking algorithm
  - Yet, it has a competitive ratio of $k$. 
Randomized Paging Algorithms

- Random Algorithm: in case an eviction is necessary, evict a page selected uniformly at random.
Randomized Paging Algorithms

- Random Algorithm: in case an eviction is necessary, evict a page selected uniformly at random.
- Random has a competitive ratio of $k$
- Is it good?
MARK Algorithm

- MARK Algorithm is a randomized marking algorithm
- In case an eviction is necessary, evict an unmarked page selected uniformly at random from all unmarked pages.
  - If all pages are marked, unmark all of them.
MARK Algorithm

MARK Algorithm is a randomized marking algorithm

In case an eviction is necessary, evict an unmarked page selected uniformly at random from all unmarked pages.

If all pages are marked, unmark all of them.

\[ \sigma = a \ b \ c \ b \ e \ f \ d \ a \]
randomly evict \( b \) or \( e \)

<table>
<thead>
<tr>
<th>f</th>
<th>b</th>
<th>e</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>
MARK Algorithm

- MARK Algorithm is a randomized marking algorithm
- In case an eviction is necessary, evict an unmarked page selected uniformly at random from all unmarked pages.
  - If all pages are marked, unmark all of them.

\[ \sigma = a \ b \ c \ b \ e \ f \ d \ a \] 

\( e \) was selected

<table>
<thead>
<tr>
<th>f</th>
<th>b</th>
<th>a</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
MARK Algorithm

- MARK Algorithm is a randomized marking algorithm.
- In case an eviction is necessary, evict an unmarked page selected uniformly at random from all unmarked pages.
  - If all pages are marked, unmark all of them.

$$\sigma = a \ b \ c \ b \ e \ f \ d \ a \ c$$

only $b$ is unmarked

<table>
<thead>
<tr>
<th></th>
<th>f</th>
<th>b</th>
<th>a</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
MARK Algorithm

- MARK Algorithm is a randomized marking algorithm.
- In case an eviction is necessary, evict an unmarked page selected uniformly at random from all unmarked pages.
  - If all pages are marked, unmark all of them.

\[ \sigma = a \ b \ c \ b \ e \ f \ d \ a \ c \]

\( b \) is evicted

<table>
<thead>
<tr>
<th>f</th>
<th>c</th>
<th>a</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
MARK Algorithm

- MARK Algorithm is a randomized marking algorithm
- In case an eviction is necessary, evict an unmarked page selected uniformly at random from all unmarked pages.
  - If all pages are marked, unmark all of them.

\[ \sigma = a \ b \ c \ b \ e \ f \ d \ a \ c \ e \]

| f | c | a | d |
MARK Algorithm

- MARK Algorithm is a randomized marking algorithm
- In case an eviction is necessary, evict an unmarked page selected uniformly at random from all unmarked pages.
  - If all pages are marked, unmark all of them.

\[ \sigma = a \ b \ c \ b \ e \ f \ d \ a \ c \ e \]
randomly evict from \( f, c, a, d \)

| f | c | a | d |
MARK Algorithm

- MARK Algorithm is a randomized marking algorithm.
- In case an eviction is necessary, evict an unmarked page selected uniformly at random from all unmarked pages.
  - If all pages are marked, unmark all of them.

\[
\sigma = a \ b \ c \ b \ e \ f \ d \ a \ c \ e
\]

\(d\) is evicted

\[
\begin{array}{cccc}
f & c & a & e \\
& & & \checkmark \\
\end{array}
\]
MARK Algorithm is a randomized marking algorithm.

In case an eviction is necessary, evict an unmarked page selected uniformly at random from all unmarked pages.

- If all pages are marked, unmark all of them.

\[ \sigma = a \ b \ c \ b \ e \ f \ d \ a \ c \ e \ b \]
Competitive Ratio of MARK

Theorem

*MARK has a competitive ratio of at most* $2H_k$

- $H_k$ is the $k$’th harmonic number

\[ H_k = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{k} \]
**Theorem**

MARK has a competitive ratio of at most $2H_k$

- $H_k$ is the $k$'th harmonic number

$$H_k = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{k}$$

- For any $k$, we have $\ln k < H_k \leq 1 + \ln k$.
  - So $H_k \in \Theta(\log k)$
**Theorem**

MARK has a competitive ratio of at most $2H_k$

- $H_k$ is the $k$'th harmonic number

$$H_k = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{k}$$

- For any $k$, we have $\ln k < H_k \leq 1 + \ln k$.
  - So $H_k \in \Theta(\log k)$

- No randomized algorithm can have a competitive ratio better than $H_k$
There are two types of memory: a fast ‘cache’ of size $k$, and a slow memory of unbounded size.

- The input is an online sequence of requests to pages of size 1.
- To serve a request to page $x$, it should be in the cache
  - In case $x$ is not in the cache, a fault of cost 1 happens
  - In case $x$ is in the cache, a hit of cost 0 happens
- The goal is to minimize the total number of faults

To bring $x$ to the cache, we might need to evict a page.

- A caching algorithm is defined through its eviction policy.
Caching Problem: A Review

- Latest-In-Future (LIF) is the optimal offline algorithm.
- No deterministic algorithm has a competitive ratio better than $k$. 
Latest-In-Future (LIF) is the optimal offline algorithm.

No deterministic algorithm has a competitive ratio better than \( k \).

An algorithm is marking if it maintains a ‘mark’ for each page.

- After a request to \( x \) mark it.
- Always evict an unmarked page (if all marked, first unmark all pages and then evict one)
Latest-In-Future (LIF) is the optimal offline algorithm.

No deterministic algorithm has a competitive ratio better than $k$.

An algorithm is marking if it maintains a ‘mark’ for each page.

- After a request to $x$ mark it.
- Always evict an unmarked page (if all marked, first unmark all pages and then evict one)

Any deterministic marking algorithm has a competitive ratio of $k$.

- Least-Recently-Used (LRU), and Flash-When-Full (FWF) both have competitive ratio $k$. 
Caching Problem: A Review

- Latest-In-Future (LIF) is the optimal offline algorithm.
- No deterministic algorithm has a competitive ratio better than $k$.
- An algorithm is marking if it maintains a ‘mark’ for each page.
  - After a request to $x$ mark it.
  - Always evict an unmarked page (if all marked, first unmark all pages and then evict one).
- Any deterministic marking algorithm has a competitive ratio of $k$.
  - Least-Recently-Used (LRU), and Flash-When-Full (FWF) both have competitive ratio $k$.
- Fist-In-First-Out (FIFO) also has a competitive ratio of $k$. 
A randomized algorithm which randomly evicts a page has a competitive ratio of \( k \).

A marking algorithm that evicts an unmarked page uniformly at random has a competitive ratio of \( H_k \)

\[ H_k = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{k} \]

For large values of \( k \), we have \( H_k \approx \ln(k) \in \Theta(\log k) \).

In fact, no randomized algorithm can achieve a better competitive ratio (i.e., \( o(\log k) \)).
A randomized algorithm which randomly evict a page has a competitive ratio of $k$.

A marking algorithm that evicts an unmarked page uniformly at random has a competitive ratio of $H_k$

- $H_k = 1 + 1/2 + 1/3 + \ldots + 1/k$
- For large values of $k$, we have $H_k \approx \ln(k) \in \Theta(\log k)$.

In fact, no randomized algorithm can achieve a better competitive ratio (i.e., $o(\log k)$).
**Question:** How many bits of advice are sufficient to achieve an optimal algorithm?

- $n$: the length of input sequence (number of requests).
- $k$: the size of the cache

(a) $O(k)$  
(b) $O(k \log n)$  
(c) $O(k \log^2 n)$  
(d) $O(n \log k)$

An algorithm has to make at most $n$ decisions about the page to be evicted. One decision per fault to indicate which one of the $k$ pages in the cache should be evicted. At most $O(n \log k)$ bits are sufficient!
Question: How many bits of advice are sufficient to achieve an optimal algorithm?

- $n$: the length of input sequence (number of requests).
- $k$: the size of the cache
- (a) $O(k)$  
- (b) $O(k \log n)$  
- (c) $O(k \log^2 n)$  
- (d) $O(n \log k)$

An algorithm has to make at most $n$ decisions about the page to be evicted.

- One decision per fault to indicate which one of the $k$ pages in the cache should be evicted.

At most $O(n \log k)$ bits are sufficient!
Caching & Advice

- What does the advice encode?
- What is the size of advice?
- How the algorithm works, provided by these bits of advice?
- Why the algorithm has a competitive ratio of 1 (optimal here)?
Caching & Advice

- What does the advice encode? The advice indicates, for each request, what an optimal offline algorithm Opt evicts in case of a failure.
- What is the size of advice?
- How the algorithm works, provided by these bits of advice?
- Why the algorithm has a competitive ratio of 1 (optimal here)?
What does the advice encode? The advice indicates, for each request, what an optimal offline algorithm Opt evicts in case of a failure.

What is the size of advice? Assume Opt makes $m \leq n$ faults for the optimal algorithm. For each fault, the advice indicates which page should be evicted. There are $k$ pages in the cache, and the evicted page can be indicated in $\Theta(\log k)$ bits. The total number of bits will be $m \cdot O(\log k) \in O(n \log k)$.

How the algorithm works, provided by these bits of advice?

Why the algorithm has a competitive ratio of 1 (optimal here)?
Caching & Advice

What does the advice encode? The advice indicates, for each request, what an optimal offline algorithm Opt evicts in case of a failure.

What is the size of advice? Assume Opt makes \( m \leq n \) faults for the optimal algorithm. For each fault, the advice indicates which page should be evicted. There are \( k \) pages in the cache, and the evicted page can be indicated in \( \Theta(\log k) \) bits. The total number of bits will be \( m \cdot O(\log k) \in O(n \log k) \).

How the algorithm works, provided by these bits of advice? It just mimics Opt; whenever there is a fault, it reads the advice to see which page should be evicted.

Why the algorithm has a competitive ratio of 1 (optimal here)?
Caching & Advice

What does the advice encode? The advice indicates, for each request, what an optimal offline algorithm Opt evicts in case of a failure.

What is the size of advice? Assume Opt makes \( m \leq n \) faults for the optimal algorithm. For each fault, the advice indicates which page should be evicted. There are \( k \) pages in the cache, and the evicted page can be indicated in \( \Theta(\log k) \) bits. The total number of bits will be \( m \cdot O(\log k) \in O(n \log k) \).

How the algorithm works, provided by these bits of advice? It just mimics Opt; whenever there is a fault, it reads the advice to see which page should be evicted.

Why the algorithm has a competitive ratio of 1 (optimal here)? It works exactly like Opt.
Theorem

There is an online algorithm that, provided with $O(n \log k)$ bits of advice, can achieve an optimal solution.

- It is a naive solution :'-)
- Can we achieve an optimal solution with a smaller number of bits of advice?
  - For many problems, the answer is no!
  - For caching problem, we can indeed do better.
Assume $OPT$ brings a page $x$ to the cache at time $t$.

- Either $OPT$ evicts $x$ before the next access to $x \rightarrow x$ is mortal.
- $OPT$ keeps $x$ in the cache until the next access to $x \rightarrow x$ is resident.
Caching & Advice (cntd.)

Assume \( \text{OPT} \) brings a page \( x \) to the cache at time \( t \).

- Either \( \text{OPT} \) evict \( x \) before the next access to it \( \rightarrow x \) is mortal.
- \( \text{OPT} \) keeps \( x \) in the cache until the next access to \( x \) \( \rightarrow x \) is resident

\[ a \text{ is resident} \]

\[ \sigma = a \; b \; c \; b \; a \; d \; c \; e \; f \; a \; c \; d \; c \; f \; a \; b \; a \; e \]
Caching & Advice (cntd.)

- Assume \( \text{OPT} \) brings a page \( x \) to the cache at time \( t \).
  - Either \( \text{OPT} \) evict \( x \) before the next access to it \( \Rightarrow x \) is mortal.
  - \( \text{OPT} \) keeps \( x \) in the cache until the next access to \( x \) \( \Rightarrow x \) is resident.

\[ \sigma = a \ b \ c \ b \ a \ d \ c \ e \ f \ a \ c \ d \ c \ f \ a \ b \ a \ e \]

\( a \) is resident
Assume $\text{OPT}$ brings a page $x$ to the cache at time $t$.

- Either $\text{OPT}$ evicts $x$ before the next access to it $\rightarrow x$ is mortal.
- $\text{OPT}$ keeps $x$ in the cache until the next access to $x$ $\rightarrow x$ is resident

$a, b$ are residents

$$\sigma = a \ b \ c \ b \ a \ d \ c \ e \ f \ a \ c \ d \ c \ f \ a \ b \ a \ e$$
Assume $\text{OPT}$ brings a page $x$ to the cache at time $t$.

- Either $\text{OPT}$ evict $x$ before the next access to it $\rightarrow x$ is mortal.
- $\text{OPT}$ keeps $x$ in the cache until the next access to $x$ $\rightarrow x$ is resident.

$a, b$ are residents

$$\sigma = a \ b \ c \ b \ a \ d \ c \ e \ f \ a \ c \ d \ c \ f \ a \ b \ a \ e$$
Assume \( \text{OPT} \) brings a page \( x \) to the cache at time \( t \).

- Either \( \text{OPT} \) evict \( x \) before the next access to it \( \rightarrow x \) is mortal.
- \( \text{OPT} \) keeps \( x \) in the cache until the next access to \( x \) \( \rightarrow x \) is resident

\[
\sigma = a \ b \ c \ b \ a \ d \ c \ e \ f \ a \ c \ d \ c \ f \ a \ b \ a \ e
\]

\( a, b, c \) are residents
Assume \( \text{OPT} \) brings a page \( x \) to the cache at time \( t \).

- Either \( \text{OPT} \) evict \( x \) before the next access to it \( \rightarrow x \) is mortal.
- \( \text{OPT} \) keeps \( x \) in the cache until the next access to \( x \) \( \rightarrow x \) is resident.

\( a, b, c \) are residents

\[ \sigma = a \ b \ c \ b \ a \ d \ c \ e \ f \ a \ c \ d \ c \ f \ a \ b \ a \ e \]
Caching & Advice (cntd.)

Assume OPT brings a page \( x \) to the cache at time \( t \).

- Either OPT evict \( x \) before the next access to it \( \rightarrow x \) is mortal.
- OPT keeps \( x \) in the cache until the next access to \( x \) \( \rightarrow x \) is resident

\[ \sigma = a \ b \ c \ b \ a \ d \ c \ e \ f \ a \ c \ d \ c \ f \ a \ b \ a \ e \]

\( a, c \) are residents \hspace{1cm} b \) is mortal
Caching & Advice (cntd.)

Assume $\text{OPT}$ brings a page $x$ to the cache at time $t$.

- Either $\text{OPT}$ evict $x$ before the next access to it $\rightarrow x$ is mortal.
- $\text{OPT}$ keeps $x$ in the cache until the next access to $x$ $\rightarrow x$ is resident

$a, c$ are residents  
$b$ is mortal

$\sigma = a \ b \ c \ b \ a \ d \ c \ e \ f \ a \ c \ d \ c \ f \ a \ b \ a \ e$
Assume $\text{OPT}$ brings a page $x$ to the cache at time $t$.

- Either $\text{OPT}$ evict $x$ before the next access to it $\rightarrow x$ is mortal.
- $\text{OPT}$ keeps $x$ in the cache until the next access to $x$ $\rightarrow x$ is resident

$a, c$ are residents  $b$ is mortal

$$\sigma = a \ b \ c \ b \ a \ d \ c \ e \ f \ a \ c \ d \ c \ f \ a \ b \ a \ e$$
Assume $\text{OPT}$ brings a page $x$ to the cache at time $t$.

- Either $\text{OPT}$ evict $x$ before the next access to it $\rightarrow x$ is mortal.
- $\text{OPT}$ keeps $x$ in the cache until the next access to $x$ $\rightarrow x$ is resident.

$a, c$ are residents $b$ is mortal

$$\sigma = a \ b \ c \ b \ a \ d \ c \ e \ f \ a \ c \ d \ c \ f \ a \ b \ a \ e$$
Caching & Advice (cntd.)

- Assume $\text{OPT}$ brings a page $x$ to the cache at time $t$.
  - Either $\text{OPT}$ evict $x$ before the next access to it $\rightarrow x$ is mortal.
  - $\text{OPT}$ keeps $x$ in the cache until the next access to $x$ $\rightarrow x$ is resident.

\[
\sigma = a \ b \ c \ b \ a \ d \ c \ e \ f \ a \ c \ d \ c \ f \ a \ b \ a \ e
\]

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>a</td>
</tr>
<tr>
<td></td>
<td>d</td>
<td>c</td>
</tr>
</tbody>
</table>

COMP 7720 - Online Algorithms  Caching (Paging) Problem
Caching & Advice (cntd.)

Assume $\text{OPT}$ brings a page $x$ to the cache at time $t$.

- Either $\text{OPT}$ evict $x$ before the next access to it $\rightarrow x$ is mortal.
- $\text{OPT}$ keeps $x$ in the cache until the next access to $x$ $\rightarrow x$ is resident

$$\sigma = a \ b \ c \ b \ a \ d \ c \ e \ f \ a \ c \ d \ c \ f \ a \ b \ a \ e$$
If \( \text{OPT} \) has a hit for request \( x \) \( \Rightarrow \) \( x \) has been resident in cache since its last access to \( x \).

Consider an algorithm \( \text{ResMor} \) that evicts a mortal page if an eviction is required. \( \text{ResMor} \) always has the same resident pages as \( \text{OPT} \) in its cache. The mortal pages might be different. \( \text{OPT} \) and \( \text{ResMor} \) have the same cost. Assume \( \text{OPT} \) has smaller cost \( \Rightarrow \) there is a request to \( x \) that is a hit by \( \text{OPT} \) and a miss for \( \text{ResMor} \) \( \Rightarrow \) \( x \) is resident in \( \text{OPT} \) and not in \( \text{ResMore} \) \( \Rightarrow \) they maintain different resident pages (\( \text{ResMore} \) has evicted a resident page at some point) \( \Rightarrow \) contradiction.
Caching & Advice (cntd.)

- If $\text{OPT}$ has a hit for request $x \Rightarrow x$ has been resident in cache since its last access to $x$.
- If $\text{OPT}$ has a fault for request $x \Rightarrow$ either it is the first access to $x$ or $x$ has been mortal after its previous access (so that it is evicted at some point).
If $OPT$ has a hit for request $x \Rightarrow x$ has been resident in cache since its last access to $x$.

If $OPT$ has a fault for request $x \Rightarrow$ either it is the first access to $x$ or $x$ has been mortal after its previous access (so that it is evicted at some point).

Consider an algorithm $ResMor$ that evicts a mortal page if an eviction is required.

- $ResMor$ always has the same resident pages as $OPT$ in its cache
- The mortal pages might be different.
If $\text{OPT}$ has a hit for request $x \Rightarrow x$ has been resident in cache since its last access to $x$.

If $\text{OPT}$ has a fault for request $x \Rightarrow$ either it is the first access to $x$ or $x$ has been mortal after its previous access (so that it is evicted at some point).

Consider an algorithm $\text{ResMor}$ that evicts a mortal page if an eviction is required.

- $\text{ResMor}$ always has the same resident pages as $\text{OPT}$ in its cache
- The mortal pages might be different.

$\text{OPT}$ and $\text{ResMor}$ have the same cost

- Assume $\text{OPT}$ has smaller cost $\Rightarrow$ there is a request to $x$ that is a hit by $\text{OPT}$ and a miss for $\text{ResMor}$ $\Rightarrow x$ is resident in $\text{OPT}$ and not in $\text{ResMore}$ $\Rightarrow$ they maintain different resident pages ($\text{ResMore}$ has evicted a resident page at some point) $\Rightarrow$ contradiction
ResMor is an optimal algorithm that, instead of the whole sequence, only needs to know which pages are resident/mortal at each given time.
ResMor is an optimal algorithm that, instead of the whole sequence, only needs to know which pages are resident/mortal at each given time.

Assume with each request, there is one bit of advice that indicates whether the requested page is resident or mortal after the request.
ResMor is an optimal algorithm that, instead of the whole sequence, only needs to know which pages are resident/mortal at each given time.

Assume with each request, there is one bit of advice that indicates whether the requested page is resident or mortal after the request.

We can think of ResMor as an online algorithm with $n$ bits of advice.
Caching & Advice

- What does the advice encode?
- What is the size of advice?
- How the algorithm works, provided by these bits of advice?
- Why the algorithm has a competitive ratio of 1 (optimal here)?
What does the advice encode? For each request there is one bit of advice indicating whether the requested page is evicted before the next request to it (i.e., it is mortal) or not (i.e., it is resident).

What is the size of advice?

How the algorithm works, provided by these bits of advice?

Why the algorithm has a competitive ratio of 1 (optimal here)?
What does the advice encode? For each request there is one bit of advice indicating whether the requested page is evicted before the next request to it (i.e., it is mortal) or not (i.e., it is resident).

What is the size of advice? It is one bit for request, i.e., $n$ bits for a sequence of $n$ requests.

How the algorithm works, provided by these bits of advice?

Why the algorithm has a competitive ratio of 1 (optimal here)?
What does the advice encode? For each request there is one bit of advice indicating whether the requested page is evicted before the next request to it (i.e., it is mortal) or not (i.e., it is resident).

What is the size of advice? It is one bit for request, i.e., $n$ bits for a sequence of $n$ requests.

How the algorithm works, provided by these bits of advice? When eviction is required, it evicts any mortal page.

Why the algorithm has a competitive ratio of 1 (optimal here)?
Caching & Advice

- What does the advice encode? For each request there is one bit of advice indicating whether the requested page is evicted before the next request to it (i.e., it is mortal) or not (i.e., it is resident).

- What is the size of advice? It is one bit for request, i.e., \( n \) bits for a sequence of \( n \) requests.

- How the algorithm works, provided by these bits of advice? When eviction is required, it evicts any mortal page.

- Why the algorithm has a competitive ratio of 1 (optimal here)? It maintains the same resident pages as Opt; so in case of a hit by Opt there will be a hit by the algorithm.
Advice Complexity of Paging

- With $n$ bits of advice, one can achieve an optimal algorithm.
- With roughly $\log\left(\frac{r+1}{r}\right) \cdot n$ bits, one can achieve a competitive ratio of $r$
  - With roughly 0.27$n$ bits, one can achieve a competitive ratio of 2.
  - With roughly 0.24$n$ bits, one can achieve a competitive ratio of 3.

For a potential project, do a survey on advice complexity of paging, and try to deduce new results!
With \( n \) bits of advice, one can achieve an optimal algorithm.

With roughly \( \log\left( \frac{r+1}{r} \right) \cdot n \) bits, one can achieve a competitive ratio of \( r \)

- With roughly 0.27\( n \) bits, one can achieve a competitive ratio of 2.
- With roughly 0.24\( n \) bits, one can achieve a competitive ratio of 3.

For a potential project, do a survey on advice complexity of paging, and try to deduce new results!
Belady’s Anomaly

- Naturally, we expect that having more pages results in less faults.
- In some caching algorithms, the number of page-faults might increase when the number of available pages increases.
  - This is called Belady’s anomaly
Belady’s Anomaly

- Naturally, we expect that having more pages results in less faults.
- In some caching algorithms, the number of page-faults might increase when the number of available pages increases.
- This is called Belady’s anomaly
- FIFO suffers from Belady’s anomaly

Assume $k = 3$. FIFO Cost is: 1

$$\sigma = a\ b\ c\ d\ a\ b\ e\ a\ b\ c\ d\ e$$

$$a$$
Naturally, we expect that having more pages results in less faults.

In some caching algorithms, the number of page-faults might increase when the number of available pages increases.

This is called Belady’s anomaly

FIFO suffers from Belady’s anomaly

Assume \( k = 3 \). FIFO Cost is: 2

\[
\sigma = a \ b \ c \ d \ a \ b \ e \ a \ b \ c \ d \ e
\]

\[
\begin{array}{c}
a \\
\end{array}
\]
Belady’s Anomaly

- Naturally, we expect that having more pages results in less faults.
- In some caching algorithms, the number of page-faults might increase when the number of available pages increases.
  - This is called Belady’s anomaly
- FIFO suffers from Belady’s anomaly

Assume \( k = 3 \). FIFO Cost is: 2

\[ \sigma = a \ b \ c \ d \ a \ b \ e \ a \ b \ c \ d \ e \]

| a | b |
Belady’s Anomaly

- Naturally, we expect that having more pages results in less faults.
- In some caching algorithms, the number of page-faults might increase when the number of available pages increases.
  - This is called Belady’s anomaly
- FIFO suffers from Belady’s anomaly

Assume $k = 3$. FIFO Cost is: 3

$$\sigma = a \ b \ c \ d \ a \ b \ e \ a \ b \ c \ d \ e$$

| a | b |
Belady’s Anomaly

- Naturally, we expect that having more pages results in less faults.
- In some caching algorithms, the number of page-faults might increase when the number of available pages increases.
  - This is called Belady’s anomaly
- FIFO suffers from Belady’s anomaly

Assume $k = 3$. FIFO Cost is: 3

$$\sigma = a \ b \ c \ d \ a \ b \ e \ a \ b \ c \ d \ e$$

\[
\begin{array}{ccc}
  a & b & c \\
\end{array}
\]
Belady’s Anomaly

- Naturally, we expect that having more pages results in less faults.
- In some caching algorithms, the number of page-faults might increase when the number of available pages increases.
  - This is called Belady’s anomaly
- FIFO suffers from Belady’s anomaly

Assume $k = 3$. FIFO Cost is: 4

$$\sigma = a \ b \ c \ d \ a \ b \ e \ a \ b \ c \ d \ e$$

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
</tr>
</tbody>
</table>
Belady’s Anomaly

- Naturally, we expect that having more pages results in less faults.
- In some caching algorithms, the number of page-faults might increase when the number of available pages increases.
- This is called Belady’s anomaly
- FIFO suffers from Belady’s anomaly

Assume $k = 3$. FIFO Cost is: 4

\[ \sigma = a b c d a b e a b c d e \]

\[
\begin{array}{ccc}
  d & b & c \\
\end{array}
\]
Belady’s Anomaly

- Naturally, we expect that having more pages results in less faults.
- In some caching algorithms, the number of page-faults might increase when the number of available pages increases.
  - This is called Belady’s anomaly
- FIFO suffers from Belady’s anomaly

Assume $k = 3$. FIFO Cost is: 5

$$\sigma = a \ b \ c \ d \ a \ b \ e \ a \ b \ c \ d \ e$$

```
  d  b  c
```
Belady’s Anomaly

Naturally, we expect that having more pages results in less faults.

In some caching algorithms, the number of page-faults might increase when the number of available pages increases.

This is called Belady’s anomaly

FIFO suffers from Belady’s anomaly

Assume $k = 3$. FIFO Cost is: 5

$\sigma = a \ b \ c \ d \ a \ b \ e \ a \ b \ c \ d \ e$

\[
\begin{array}{ccc}
d & a & c \\
\end{array}
\]
Belady’s Anomaly

- Naturally, we expect that having more pages results in less faults.
- In some caching algorithms, the number of page-faults might increase when the number of available pages increases.
  - This is called Belady’s anomaly
- FIFO suffers from Belady’s anomaly

Assume $k = 3$. FIFO Cost is: 6

$$\sigma = a \ b \ c \ d \ a \ b \ e \ a \ b \ c \ d \ e$$

|  d |  a |  c |
Belady’s Anomaly

- Naturally, we expect that having more pages results in less faults.
- In some caching algorithms, the number of page-faults might increase when the number of available pages increases.
  - This is called Belady’s anomaly
- FIFO suffers from Belady’s anomaly

Assume $k = 3$. FIFO Cost is: 6

$\sigma = a \ b \ c \ d \ a \ b \ e \ a \ b \ c \ d \ e$

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>a</td>
<td>b</td>
</tr>
</tbody>
</table>
Belady’s Anomaly

- Naturally, we expect that having more pages results in less faults.
- In some caching algorithms, the number of page-faults might increase when the number of available pages increases.
  - This is called Belady’s anomaly
- FIFO suffers from Belady’s anomaly

Assume $k = 3$. FIFO Cost is: 7

$$\sigma = a \ b \ c \ d \ a \ b \ e \ a \ b \ c \ d \ e$$

\[ \begin{array}{ccc}
  d & a & b \\
\end{array} \]
Belady’s Anomaly

- Naturally, we expect that having more pages results in less faults.
- In some caching algorithms, the number of page-faults might increase when the number of available pages increases.
  - This is called Belady’s anomaly
- FIFO suffers from Belady’s anomaly

Assume $k = 3$. FIFO Cost is: 7

$$\sigma = a \ b \ c \ d \ a \ b \ e \ a \ b \ c \ d \ e$$

```
  e  a  b
```
Belady’s Anomaly

- Naturally, we expect that having more pages results in less faults.
- In some caching algorithms, the number of page-faults might increase when the number of available pages increases.
  - This is called Belady’s anomaly
- FIFO suffers from Belady’s anomaly

Assume $k = 3$. FIFO Cost is: 7

$$\sigma = a \ b \ c \ d \ a \ b \ e \ a \ b \ c \ d \ e$$

| e | a | b |
Naturally, we expect that having more pages results in less faults.

In some caching algorithms, the number of page-faults might increase when the number of available pages increases.

This is called Belady’s anomaly

FIFO suffers from Belady’s anomaly

Assume $k = 3$. FIFO Cost is: 7

$$\sigma = a \ b \ c \ d \ a \ b \ e \ a \ b \ c \ d \ e$$

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>e</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Belady’s Anomaly

- Naturally, we expect that having more pages results in less faults.
- In some caching algorithms, the number of page-faults might increase when the number of available pages increases.
  - This is called Belady’s anomaly
- FIFO suffers from Belady’s anomaly

Assume $k = 3$. FIFO Cost is: 7

$$\sigma = a \ b \ c \ d \ a \ b \ e \ a \ b \ c \ d \ e$$

| e | a | b |
Belady’s Anomaly

- Naturally, we expect that having more pages results in less faults.
- In some caching algorithms, the number of page-faults might increase when the number of available pages increases.
- This is called **Belady’s anomaly**
- FIFO suffers from Belady’s anomaly

Assume \( k = 3 \). FIFO Cost is: 7

\[
\sigma = a \ b \ c \ d \ a \ b \ e \ a \ b \ c \ d \ e
\]

\[
\begin{array}{ccc}
\text{e} & \text{a} & \text{b}
\end{array}
\]
Belady’s Anomaly

- Naturally, we expect that having more pages results in less faults.
- In some caching algorithms, the number of page-faults might increase when the number of available pages increases.
- This is called Belady’s anomaly
- FIFO suffers from Belady’s anomaly

Assume $k = 3$. FIFO Cost is: 8

$$\sigma = a \ b \ c \ d \ a \ b \ e \ a \ b \ c \ d \ e$$

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>e</td>
<td>a</td>
<td>b</td>
</tr>
</tbody>
</table>
Belady’s Anomaly

- Naturally, we expect that having more pages results in less faults.
- In some caching algorithms, the number of page-faults might increase when the number of available pages increases.
  - This is called Belady’s anomaly
- FIFO suffers from Belady’s anomaly

Assume $k = 3$. FIFO Cost is: 8

$$\sigma = a\ b\ c\ d\ a\ b\ e\ a\ b\ c\ d\ e$$

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>e</td>
<td>c</td>
<td>b</td>
</tr>
</tbody>
</table>
Belady’s Anomaly

- Naturally, we expect that having more pages results in less faults.
- In some caching algorithms, the number of page-faults might increase when the number of available pages increases.
  - This is called Belady’s anomaly
- FIFO suffers from Belady’s anomaly

Assume $k = 3$. FIFO Cost is: 9

$$\sigma = a\ b\ c\ d\ a\ b\ e\ a\ b\ c\ d\ e$$

| e | c | b |
Belady’s Anomaly

- Naturally, we expect that having more pages results in less faults.
- In some caching algorithms, the number of page-faults might increase when the number of available pages increases.
  - This is called Belady’s anomaly
- FIFO suffers from Belady’s anomaly

Assume \( k = 3 \). FIFO Cost is: 9

\[
\sigma = a \ b \ c \ d \ a \ b \ e \ a \ b \ c \ d \ e
\]

<table>
<thead>
<tr>
<th></th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>e</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Belady’s Anomaly

- Naturally, we expect that having more pages results in less faults.
- In some caching algorithms, the number of page-faults might increase when the number of available pages increases.
  - This is called Belady’s anomaly
- FIFO suffers from Belady’s anomaly

Assume $k = 3$. FIFO Cost is: 9

$$\sigma = a \ b \ c \ d \ a \ b \ e \ a \ b \ c \ d \ e$$

<table>
<thead>
<tr>
<th></th>
<th>e</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
</table>

Belady’s Anomaly

- Naturally, we expect that having more pages results in less faults.
- In some caching algorithms, the number of page-faults might increase when the number of available pages increases.
  - This is called Belady’s anomaly
- FIFO suffers from Belady’s anomaly

Assume $k = 3$. FIFO Cost is: 9

$$\sigma = a \ b \ c \ d \ a \ b \ e \ a \ b \ c \ d \ e$$

|   | e | c | d |
Belady’s Anomaly

- Naturally, we expect that having more pages results in less faults.
- In some caching algorithms, the number of page-faults might increase when the number of available pages increases.
  - This is called Belady’s anomaly
Belady’s Anomaly

- Naturally, we expect that having more pages results in less faults.
- In some caching algorithms, the number of page-faults might increase when the number of available pages increases.
  - This is called Belady’s anomaly
- FIFO suffers from Belady’s anomaly

Assume $k = 3$. FIFO Cost is: 9

Assume $k = 4$. FIFO Cost is: 1

$\sigma = a \ b \ c \ d \ a \ b \ e \ a \ b \ c \ d \ e$

\[
\begin{array}{c}
\text{a} \\
\text{a} \\
\text{a}
\end{array}
\]
Belady’s Anomaly

- Naturally, we expect that having more pages results in less faults.
- In some caching algorithms, the number of page-faults might increase when the number of available pages increases.
- This is called Belady’s anomaly
- FIFO suffers from Belady’s anomaly

Assume $k = 4$. FIFO Cost is: 2

Assume $k = 3$. FIFO Cost is: 9

$$\sigma = a \ b \ c \ d \ a \ b \ e \ a \ b \ c \ d \ e$$

a -------
Belady’s Anomaly

- Naturally, we expect that having more pages results in less faults.
- In some caching algorithms, the number of page-faults might increase when the number of available pages increases.
  - This is called Belady’s anomaly
- FIFO suffers from Belady’s anomaly

Assume \( k = 4 \). FIFO Cost is: 2

Assume \( k = 3 \).
FIFO Cost is: 9

\[ \sigma = a \ b \ c \ d \ a \ b \ e \ a \ b \ c \ d \ e \]

\[ \begin{array}{c|c|c}
  a & b & \text{ } \\
\end{array} \]
Belady’s Anomaly

- Naturally, we expect that having more pages results in less faults.
- In some caching algorithms, the number of page-faults might increase when the number of available pages increases.
  - This is called Belady’s anomaly
- FIFO suffers from Belady’s anomaly

Assume $k = 4$. FIFO Cost is: 3

Assume $k = 3$.
FIFO Cost is: 9

$\sigma = a \ b \ c \ d \ a \ b \ e \ a \ b \ c \ d \ e$

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td></td>
</tr>
</tbody>
</table>
Belady’s Anomaly

- Naturally, we expect that having more pages results in less faults.
- In some caching algorithms, the number of page-faults might increase when the number of available pages increases.
  - This is called Belady’s anomaly
- FIFO suffers from Belady’s anomaly

Assume $k = 4$. FIFO Cost is: 3

Assume $k = 3$. FIFO Cost is: 9

$\sigma = a \ b \ c \ d \ a \ b \ e \ a \ b \ c \ d \ e$

| a | b | c |
Belady’s Anomaly

- Naturally, we expect that having more pages results in less faults.
- In some caching algorithms, the number of page-faults might increase when the number of available pages increases.
  - This is called Belady’s anomaly
- FIFO suffers from Belady’s anomaly

Assume $k = 4$. FIFO Cost is: 4

Assume $k = 3$.
FIFO Cost is: 9

$\sigma = a \ b \ c \ d \ a \ b \ e \ a \ b \ c \ d \ e$

| a | b | c |
Belady’s Anomaly

- Naturally, we expect that having more pages results in less faults.
- In some caching algorithms, the number of page-faults might increase when the number of available pages increases.
- This is called Belady’s anomaly
- FIFO suffers from Belady’s anomaly

Assume $k = 3$. FIFO Cost is: 9

Assume $k = 4$. FIFO Cost is: 4

$\sigma = a \ b \ c \ d \ a \ b \ e \ a \ b \ c \ d \ e$

\[
\begin{array}{c|c|c|c}
 a & b & c & d \\
\end{array}
\]
Belady’s Anomaly

- Naturally, we expect that having more pages results in less faults.
- In some caching algorithms, the number of page-faults might increase when the number of available pages increases.
- This is called Belady’s anomaly
- FIFO suffers from Belady’s anomaly

Assume $k = 3$. FIFO Cost is: 9

Assume $k = 4$. FIFO Cost is: 4

$\sigma = a \ b \ c \ d \ a \ b \ e \ a \ b \ c \ d \ e$

| a | b | c | d |
Belady’s Anomaly

- Naturally, we expect that having more pages results in less faults.
- In some caching algorithms, the number of page-faults might increase when the number of available pages increases.
  - This is called **Belady’s anomaly**
- FIFO suffers from Belady’s anomaly

\[ \sigma = a\ b\ c\ d\ a\ b\ e\ a\ b\ c\ d\ e \]

\[ \begin{array}{cccc}
  a & b & c & d \\
\end{array} \]

Assume \( k = 4 \). FIFO Cost is: 4

Assume \( k = 3 \).
FIFO Cost is: 9
Belady’s Anomaly

- Naturally, we expect that having more pages results in less faults.
- In some caching algorithms, the number of page-faults might increase when the number of available pages increases.
  - This is called Belady’s anomaly
- FIFO suffers from Belady’s anomaly

Assume $k = 4$. FIFO Cost is: 4

Assume $k = 3$.
FIFO Cost is: 9

$\sigma = a \ b \ c \ d \ a \ b \ e \ a \ b \ c \ d \ e$

\[
\begin{array}{|c|c|c|c|}
\hline
a & b & c & d \\
\hline
\end{array}
\]
Belady’s Anomaly

- Naturally, we expect that having more pages results in less faults.
- In some caching algorithms, the number of page-faults might increase when the number of available pages increases.
  - This is called Belady’s anomaly
- FIFO suffers from Belady’s anomaly

Assume $k = 4$. FIFO Cost is: 4

Assume $k = 3$.
FIFO Cost is: 9

$\sigma = a \ b \ c \ d \ a \ b \ e \ a \ b \ c \ d \ e$

| a | b | c | d |
Naturally, we expect that having more pages results in less faults.

In some caching algorithms, the number of page-faults might increase when the number of available pages increases.

- This is called Belady’s anomaly
- FIFO suffers from Belady’s anomaly

Assume $k = 4$. FIFO Cost is: 5

Assume $k = 3$.
FIFO Cost is: 9

$$\sigma = a\ b\ c\ d\ a\ b\ e\ a\ b\ c\ d\ e$$

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
</table>
Belady’s Anomaly

- Naturally, we expect that having more pages results in less faults.
- In some caching algorithms, the number of page-faults might increase when the number of available pages increases.
- This is called Belady’s anomaly.
- FIFO suffers from Belady’s anomaly.

Assume $k = 4$. FIFO Cost is: 5

Assume $k = 3$. FIFO Cost is: 9

$\sigma = a \ b \ c \ d \ a \ b \ e \ a \ b \ c \ d \ e$

| e | b | c | d |
Belady’s Anomaly

- Naturally, we expect that having more pages results in less faults.
- In some caching algorithms, the number of page-faults might increase when the number of available pages increases.
- This is called Belady’s anomaly
- FIFO suffers from Belady’s anomaly

Assume $k = 4$. FIFO Cost is: 6

Assume $k = 3$.
FIFO Cost is: 9

$$\sigma = a \ b \ c \ d \ a \ b \ e \ a \ b \ c \ d \ e$$

| e | b | c | d |
Belady’s Anomaly

- Naturally, we expect that having more pages results in less faults.
- In some caching algorithms, the number of page-faults might increase when the number of available pages increases.
  - This is called Belady’s anomaly
- FIFO suffers from Belady’s anomaly

Assume $k = 4$. FIFO Cost is: 6

Assume $k = 3$. FIFO Cost is: 9

$$\sigma = a \ b \ c \ d \ a \ b \ e \ a \ b \ c \ d \ e$$

\[
\begin{array}{cccc}
  e & a & c & d \\
\end{array}
\]
Belady’s Anomaly

- Naturally, we expect that having more pages results in less faults.
- In some caching algorithms, the number of page-faults might increase when the number of available pages increases.
  - This is called Belady’s anomaly
- FIFO suffers from Belady’s anomaly

Assume \( k = 4 \). FIFO Cost is: 7

Assume \( k = 3 \).
FIFO Cost is: 9

\[ \sigma = a\ b\ c\ d\ a\ b\ e\ a\ b\ c\ d\ e \]

\[
\begin{array}{cccc}
e & a & c & d \\
\end{array}
\]
Belady’s Anomaly

- Naturally, we expect that having more pages results in less faults.
- In some caching algorithms, the number of page-faults might increase when the number of available pages increases.
  - This is called Belady’s anomaly
- FIFO suffers from Belady’s anomaly

Assume $k = 4$. FIFO Cost is: 7

Assume $k = 3$. FIFO Cost is: 9

$\sigma = a \ b \ c \ d \ a \ b \ e \ a \ b \ c \ d \ e$

| e | a | b | d |
Belady’s Anomaly

- Naturally, we expect that having more pages results in less faults.
- In some caching algorithms, the number of page-faults might increase when the number of available pages increases.
  - This is called Belady’s anomaly
- FIFO suffers from Belady’s anomaly

Assume $k = 3$. FIFO Cost is: 9

Assume $k = 4$. FIFO Cost is: 8

$\sigma = a \ b \ c \ d \ a \ b \ e \ a \ b \ c \ d \ e$

<table>
<thead>
<tr>
<th></th>
<th>e</th>
<th>a</th>
<th>b</th>
<th>d</th>
</tr>
</thead>
</table>
Belady’s Anomaly

• Naturally, we expect that having more pages results in less faults.
• In some caching algorithms, the number of page-faults might increase when the number of available pages increases.
  • This is called Belady’s anomaly
• FIFO suffers from Belady’s anomaly

Assume \( k = 4 \). FIFO Cost is: 8

Assume \( k = 3 \).
FIFO Cost is: 9

\[ \sigma = a \ b \ c \ d \ a \ b \ e \ a \ b \ c \ d \ e \]

\[
\begin{array}{cccc}
e & a & b & c \\
\end{array}
\]
Belady’s Anomaly

- Naturally, we expect that having more pages results in less faults.
- In some caching algorithms, the number of page-faults might increase when the number of available pages increases.
  - This is called Belady’s anomaly
- FIFO suffers from Belady’s anomaly

Assume $k = 4$. FIFO Cost is: 9

Assume $k = 3$.
FIFO Cost is: 9

$\sigma = a b c d a b e a b c d e$

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
</table>
Belady’s Anomaly

- Naturally, we expect that having more pages results in less faults.
- In some caching algorithms, the number of page-faults might increase when the number of available pages increases.
  - This is called **Belady’s anomaly**
- FIFO suffers from Belady’s anomaly

Assume $k = 4$. FIFO Cost is: 9

Assume $k = 3$. FIFO Cost is: 9

$$\sigma = a \ b \ c \ d \ a \ b \ e \ a \ b \ c \ d \ e$$

```
  d a b c
```
Belady’s Anomaly

- Naturally, we expect that having more pages results in less faults.
- In some caching algorithms, the number of page-faults might increase when the number of available pages increases.
- This is called Belady’s anomaly.
- FIFO suffers from Belady’s anomaly.

\[ \sigma = a \ b \ c \ d \ a \ b \ e \ a \ b \ c \ d \ e \]

Assume \( k = 4 \). FIFO Cost is: 10

Assume \( k = 3 \).
FIFO Cost is: 9

\[
\begin{array}{cccc}
d & a & b & c \\
\end{array}
\]
Naturally, we expect that having more pages results in less faults.

In some caching algorithms, the number of page-faults might increase when the number of available pages increases.

This is called Belady’s anomaly

FIFO suffers from Belady’s anomaly

Assume $k = 4$. FIFO Cost is: 10

Assume $k = 3$. FIFO Cost is: 9

$\sigma = a\ b\ c\ d\ a\ b\ e\ a\ b\ c\ d\ e$

```
  d   e   b   c
```
Anomaly’s Summary

- We see more anomalies in analysis of online algorithms
- Project topic: make a survey on animality of different caching algorithms
  - Do some experiments, try to find anomaly examples by running algorithms on random inputs!