COMP 7720 - Online Algorithms

Caching (Paging) Problem

Shahin Kamali

Lecture 8 - Oct. 3, 2017

University of Manitoba
Review & Plan

Review & Plan
Today’s objectives

- Caching Problem
  - Optimal offline algorithm
  - Lower bound for deterministic algorithms
  - Marking algorithms & upper bounds
  - Randomized algorithms
  - Caching anomalies
Caching Problem
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

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

$$\sigma = \begin{array}{cccc}
\end{array}$$

COMP 7720 - Online Algorithms  Caching (Paging) Problem
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): $1$

$\sigma = a$

| a | | |
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): $2$

$$\sigma = a \ b$$

| a | b |   |   |
Caching 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): 3

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

\[
\begin{array}{|c|c|c|}
\hline
a & b & c \\
\hline
\end{array}
\]
Caching 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): $3$

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

| 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): 3

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

| a | b | c |
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.

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

| a | b | c | d |
Caching 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): 4

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

| a | b | c | d |
Caching 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): $5$

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

| a | b | c | d | e |
Caching 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):

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

$$\begin{array}{|c|c|c|c|}
\hline
a & e & c & d \\
\hline
\end{array}$$
Least-Recently-Used (LRU)

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

Cost (number of faults): 5

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

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

Cost (number of faults): 6

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

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

Cost (number of faults): 6

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

\[
\begin{array}{|c|c|c|c|}
\hline
f & e & c & d \\
\hline
\end{array}
\]
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 | d |
Caching Problem

Least-Recently-Used (LRU)

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

Cost (number of faults): 7

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

\[ \begin{array}{cccc}
f & e & c & a \\
\end{array} \]
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

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

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

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

\[
\begin{array}{cccc}
e & b & c & d \\
\end{array}
\]
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 |
FIFO algorithm: if eviction is necessary, evict the oldest page in the cache (the one that came earlier).

Cost (number of faults): 7

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

<table>
<thead>
<tr>
<th></th>
<th>f</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>e</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
FIFO algorithm: if eviction is necessary, evict the oldest page in the cache (the one that came earlier).

Cost (number of faults): 7

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

\[
\begin{array}{|c|c|}
\hline
e & f \\
\hline
\end{array}
\begin{array}{|c|c|}
\hline
a & d \\
\hline
\end{array}\]
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></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
</table>
Caching Problem

Flash-When-Full (FWF)

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

Cost (number of faults): 5

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

\[
\begin{array}{c}
e \\
\end{array}
\]
Caching Problem

Flash-When-Full (FWF)

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

Cost (number of faults): 6

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

\[\begin{array}{cc}
e & f \\
\end{array}\]
**Caching Problem**

**Flash-When-Full (FWF)**

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

Cost (number of faults): 6

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

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>e</td>
<td>f</td>
<td>c</td>
<td>d</td>
</tr>
</tbody>
</table>
**Flash-When-Full (FWF)**

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

Cost (number of faults): 7

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

<table>
<thead>
<tr>
<th>e</th>
<th>f</th>
<th></th>
</tr>
</thead>
</table>
Caching Problem

Flash-When-Full (FWF)

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

Cost (number of faults): 7

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

\[
\begin{array}{ccc}
e & f & a \\
\end{array}
\]
Caching Problem

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

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

<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>
</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): 5

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

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

\[
\begin{array}{cccc}
a & e & c & d \\
\end{array}
\]
Caching Problem

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

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

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

COMP 7720 - Online Algorithms  Caching (Paging) Problem
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

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

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>f</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

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

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>f</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
</table>
Caching Problem

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
Caching Problem

Caching Algorithms & Competitive Ratio

Theorem

For a cache of size $k$, no deterministic caching algorithm can have a competitive ratio better than $k$. 
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.
  - The cost of $A$ will be $n$. 

Caching Problem

Caching Algorithms & Competitive Ratio
Theorem

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

- For any such sequence, 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.
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 $n/k$
- The ratio between the cost of A and FIF is at least $k$
So, no deterministic algorithm can be better than $k$-competitive.

- No algorithm is ‘competitive’ in the sense that the competitive ratio depends on the input.

Yet, a competitive ratio of $k$ is much better than a ratio that depends on $n$.

- Why?
Caching Problem

Competitive Ratio of LRU

Theorem

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

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

\[ \sigma = \underbrace{a b c b a d c e f a c d c d f a b a e \ldots}_{\text{phase 1}} \]

\[ k = 4 \]
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.

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

$k = 4$
Caching 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.

\[
\sigma = \underbrace{a \ b \ c \ b \ a \ d \ c}_{\text{phase1}} \underbrace{e \ f \ a \ c}_{\text{phase2}} \underbrace{d \ c \ d \ f \ a \ b \ a \ e \ \ldots}_{\text{phase3}} \quad k = 4
\]
Caching Problem

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 $\text{OPT}$ **per phase**?
  - Each phase + next item has $k + 1$ distinct pages
  - $\text{OPT}$ has to pay a cost of 1 per phase!

$$\sigma = \underbrace{a b c b a d c}_{\text{phase 1}} \quad \underbrace{e f a c}_{\text{phase 2}} \quad \underbrace{d c d f a b a e \ldots}_{\text{phase 3}} \quad k = 4$$
Caching Problem

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!

$\sigma = \underbrace{a b c b a d c}_{\text{phase1}} \underbrace{e f a c}_{\text{phase2}} \underbrace{d c d f a b a e \ldots}_{\text{phase3}}$  \hspace{1cm} k = 4
Theorem

**LRU has a competitive ratio of at most \( k \).**

- The ratio between the cost of LRU and \( \text{OPT} \) is at most \( k \) per phase

\[
c.r.(\text{LRU}) = \frac{\text{LRU}(\text{phase}_1) + \text{LRU}(\text{phase}_2) + \ldots + \text{LRU}(\text{phase}_N)}{\text{OPT}(\text{phase}_1) + \text{OPT}(\text{phase}_2) + \ldots + \text{OPT}(\text{phase}_N)} \\
\leq \text{Max}_i \frac{\text{LRU}(\text{phase}_i)}{\text{OPT}(\text{phase}_i)} \leq k
\]

\( \sigma = \underline{a \ b \ c \ b \ a \ d \ c} \underline{e \ f \ a \ c} \underline{d \ c \ d \ f \ a \ b \ a \ e \ldots} \quad k = 4 \)
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 $k$
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.
- 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!

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

- Start with all pages unmarked.
- Upon a hit, mark the page.
- 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!

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

- Start with all pages unmarked.
- Upon a hit, mark the page.
- 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!

\[ \sigma = a \ b \]
Caching Problem

Marking 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.
- 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!

\[
\sigma = a \ b
\]

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<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.
- 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!

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

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td></td>
</tr>
<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.
- 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!

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

- Start with all pages unmarked.
- Upon a hit, mark the page.
- 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!

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

<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 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.
- 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!

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

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</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.
- 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!

\[ \sigma = a \ b \ c \ b \ e \]
Marking 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.
  - 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!

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

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
</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.
- 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!

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

```
  a   b   e   ✓   d
```
A marking algorithm maintains a bit (‘mark’) for each page in the cache.

- Start with all pages unmarked.
- Upon a hit, mark the page.
- 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!

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

<table>
<thead>
<tr>
<th></th>
<th>a</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.
- 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!

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

Marking 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.
  - 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!

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

<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.
- 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!

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

<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.
- 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!

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

<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.
- 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!

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

<table>
<thead>
<tr>
<th></th>
<th>f</th>
<th>a</th>
<th>e</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
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

$$\sigma = \underbrace{a b c b a d c}_{\text{phase 1}} \underbrace{e f a c}_{\text{phase 2}} \underbrace{d c d f a b a e \ldots}_{\text{phase 3}} \quad k = 4$$
Caching Problem

Marking 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
- 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

\[
\sigma = \underbrace{a \ b \ c \ b \ a \ d \ c}_\text{phase1} \ \underbrace{e \ f \ a \ c}_\text{phase2} \ \underbrace{d \ c \ d \ f \ a \ b \ a \ e \ \ldots}_\text{phase3}
\]

$k = 4$
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 $\text{OPT}$ has to pay a cost of 1 per 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

**LRU is a marking algorithm**
**Theorem**

*LRU is a marking algorithm*

- Assume LRU is not marking
  - So, it evicts a marked page $x$ at some phase for a request to $y$
    - Both $x$ and $y$ are among $k$ pages that define the phase
  - In order to evict $x$, it should be least-recently used, i.e., there should be $k - 1$ pages requested after $x$ and before $y$.
    - Adding $x$ and $y$, there will be $k + 1$ pages in the phase $\rightarrow$ contradiction
Marking Algorithms Remarks

- LRU and Flash-When-Full are marking algorithms
  - They have competitive ratio $k$
Caching Problem

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.
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 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 \( e \)

\[ \begin{array}{cccc}
  f & b & e & d \\
  \checkmark & \checkmark & \\
\end{array} \]
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.

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

\( e \) was selected

<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.

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

only \(b\) is unmarked

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>b</td>
<td>a</td>
<td>d</td>
</tr>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

18 / 23
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.

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

\[b \text{ is evicted}\]

\[
\begin{array}{cccc}
  f & c & a & d \\
  \checkmark & \checkmark & \checkmark & \checkmark \\
\end{array}
\]
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.

\[ \sigma = \text{a b c b e f d a c e} \]

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

\[ \sigma = a \ b \ c \ b \ e \ f \ d \ a \ c \ e \quad d \text{ is evicted} \]

| f | c | a | e | ✓ |
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.

\[ \sigma = a \ b \ c \ b \ e \ f \ d \ a \ c \ e \ b \]
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 $H_k$
No paging algorithm can have a competitive ratio better than $k$

- LRU, FIFI, and FWF all have the optimal competitive ratio of $k$
No paging algorithm can have a competitive ratio better than $k$

- LRU, FIFI, and FWF all have the optimal competitive ratio of $k$

No randomized algorithm can have a competitive ratio better than $H_k \in \Theta(\log k)$.

- MARK has the optimal competitive ratio of $H_k$. 
Caching Problem

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
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

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

\[ a \]
Caching Problem

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

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

<table>
<thead>
<tr>
<th>a</th>
<th></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: 2

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

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<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: 3

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

| 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: 3

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

| a | b | c |
Caching Problem

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

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

| a | b | c |
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

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

| 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

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

| d | b | c |
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

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

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

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

\[
\begin{array}{ccc}
d & a & c \\
\end{array}
\]
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 $k = 3$. 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>
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.

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

| d | 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

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

| 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>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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

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

\[
\begin{array}{ccc}
e & a & b \\
\end{array}
\]
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 $k = 3$. FIFO Cost is: $7$

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

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>e</td>
<td>a</td>
<td></td>
</tr>
</tbody>
</table>
Caching Problem

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

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

\[
\begin{array}{ccc}
e & a & b \\
\end{array}
\]
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 $k = 3$. 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>
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

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

$$\begin{array}{ccc}
  e & c & b \\
\end{array}$$
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 $k = 3$. FIFO Cost is: 9

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

| e | 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 \( k = 3 \). 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
- FIFO suffers from Belady’s anomaly

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

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

\[ \begin{array}{|c|c|c|}
\hline
 e & c & d \\
\hline
\end{array} \]
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
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: 1

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

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

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

\[ \begin{array}{ccc}
\text{a} \\
\end{array} \]
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 $k = 3$. FIFO Cost is: 9

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

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

| 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: 9

Assume \( k = 4 \).
FIFO Cost is: 3

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

\[
\begin{array}{ccc}
\text{a} & \text{b} & \\
\end{array}
\]
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 $k = 3$. FIFO Cost is: 9

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

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

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

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

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

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

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

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

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

<table>
<thead>
<tr>
<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: 4

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

\( \sigma = a \ b \ c \ d \ a \ b \ e \ a \ b \ c \ d \ e \)

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

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

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

<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: 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
```
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

\( \sigma = a \ b \ c \ d \ a \ b \ e \ a \ b \ c \ d \ e \)

```
| a | b | c | d |
```

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

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

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

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

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

\[
\begin{array}{cccc}
  e & a & c & d \\
\end{array}
\]
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 | 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: $7$

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

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

```
e a b d
```
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

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

\[
\begin{array}{cccc}
  e & a & b & d \\
\end{array}
\]
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: 8

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

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

|   | e | a | b | c |
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

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

\[
\begin{array}{cccc}
e & a & b & c \\
\end{array}
\]
Caching Problem

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 |
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  a  b  c
```
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: 10

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

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>e</td>
<td>b</td>
</tr>
</tbody>
</table>
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!