Offline Caching

- Cache that can store k pages
- Sequence of page requests
- Cache miss happens if requested page not in cache. We need bring the page into cache, and evict some existing page if necessary.
- Cache hit happens if requested page already in cache.
- Goal: minimize the number of cache misses.

page sequence		cache		
1	✗	1		
5	✗	1	5	
4	✗	1	5	4
2	✗	1	2	4
5	✗	1	2	5
3	✗	1	2	3
2	✓	1	2	3
1	✓	1	2	3

misses $= 6$

A Better Solution for Example

page sequence		cache					cache		
1	✗	1				✗	1		
5	✗	1	5			✗	1	5	
4	✗	1	5	4		✗	1	5	4
2	✗	1	2	4		✗	1	5	2
5	✗	1	2	5		✓	1	5	2
3	✗	1	2	3		✗	1	3	2
2	✓	1	2	3		✓	1	3	2
1	✓	1	2	3		✓	1	3	2
		misses = 6					misses = 5		

Offline Caching Problem

Input: k : the size of cache

n : number of pages

We use $[n]$ for $\{1, 2, 3, \cdots, n\}$.

$\rho_1, \rho_2, \rho_3, \cdots, \rho_T \in [n]$: sequence of requests

Output: $i_1, i_2, i_3, \cdots, i_T \in \{\mathrm{hit}, \mathrm{empty}\} \cup [n]$: indices of pages to evict ("hit" means evicting no page, "empty" means evicting empty page)

Offline Caching Problem

Input: k : the size of cache

n : number of pages

$\rho_1, \rho_2, \rho_3, \cdots, \rho_T \in [n]$: sequence of requests

We use $[n]$ for $\{1, 2, 3, \cdots, n\}$.

Output: $i_1, i_2, i_3, \cdots, i_T \in \{\text{hit}, \text{empty}\} \cup [n]$: indices of pages to evict ("hit" means evicting no page, "empty" means evicting empty page)

- Offline Caching: we know the whole sequence ahead of time.
- Online Caching: we have to make decisions on the fly, before seeing future requests.

Offline Caching Problem

Input: k : the size of cache

n : number of pages

We use $[n]$ for $\{1, 2, 3, \cdots, n\}$.

$\rho_1, \rho_2, \rho_3, \cdots, \rho_T \in [n]$: sequence of requests

Output: $i_1, i_2, i_3, \cdots, i_T \in \{\text{hit}, \text{empty}\} \cup [n]$: indices of pages to evict ("hit" means evicting no page, "empty" means evicting empty page)

- Offline Caching: we know the whole sequence ahead of time.
- Online Caching: we have to make decisions on the fly, before seeing future requests.

Q: Which one is more realistic?

Offline Caching Problem

Input: k : the size of cache

n : number of pages

We use $[n]$ for $\{1, 2, 3, \cdots, n\}$.

$\rho_1, \rho_2, \rho_3, \cdots, \rho_T \in [n]$: sequence of requests

Output: $i_1, i_2, i_3, \cdots, i_T \in \{\text{hit}, \text{empty}\} \cup [n]$: indices of pages to evict ("hit" means evicting no page, "empty" means evicting empty page)

- Offline Caching: we know the whole sequence ahead of time.
- Online Caching: we have to make decisions on the fly, before seeing future requests.

Q: Which one is more realistic?

A: Online caching

- Offline Caching: we know the whole sequence ahead of time.
- Online Caching: we have to make decisions on the fly, before seeing future requests.

Q: Which one is more realistic?

A: Online caching

Q: Why do we study the offline caching problem?

- Offline Caching: we know the whole sequence ahead of time.
- Online Caching: we have to make decisions on the fly, before seeing future requests.

Q: Which one is more realistic?

A: Online caching

Q: Why do we study the offline caching problem?

A: Use the offline solution as a benchmark to measure the "competitive ratio" of online algorithms

Offline Caching: Potential Greedy Algorithms

- FIFO(First-In-First-Out): Evict the first-in page in cache

Offline Caching: Potential Greedy Algorithms

- FIFO(First-In-First-Out): Evict the first-in page in cache
- LRU(Least-Recently-Used): Evict page whose most recent access was earliest

Offline Caching: Potential Greedy Algorithms

- FIFO(First-In-First-Out): Evict the first-in page in cache
- LRU(Least-Recently-Used): Evict page whose most recent access was earliest
- LFU(Least-Frequently-Used): Evict page that was least frequently requested

Offline Caching: Potential Greedy Algorithms

- FIFO(First-In-First-Out): Evict the first-in page in cache
- LRU(Least-Recently-Used): Evict page whose most recent access was earliest
- LFU(Least-Frequently-Used): Evict page that was least frequently requested
- LIFO (Last In First Out): Evict the last-in page in cache

Offline Caching: Potential Greedy Algorithms

- FIFO(First-In-First-Out): Evict the first-in page in cache
- LRU(Least-Recently-Used): Evict page whose most recent access was earliest
- LFU(Least-Frequently-Used): Evict page that was least frequently requested
- LIFO (Last In First Out): Evict the last-in page in cache

- All the above algorithms are not optimum!
- Indeed all the algorithms are "online", i.e, the decisions can be made without knowing future requests. Online algorithms can not be optimum.

FIFO is not optimum

FIFO

requests

1

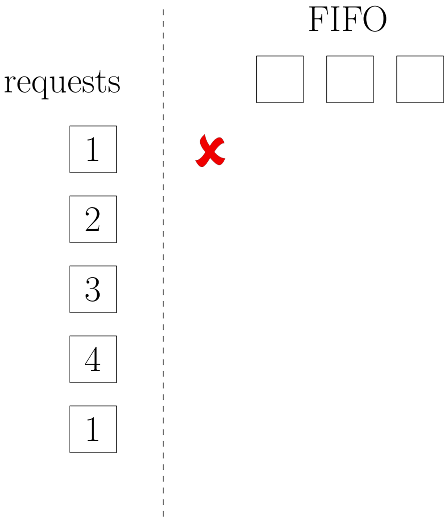
2

3

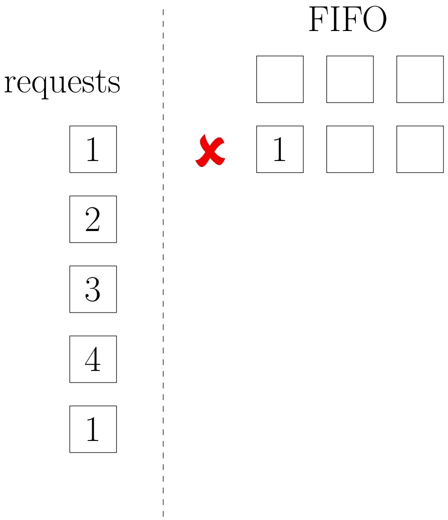
4

1

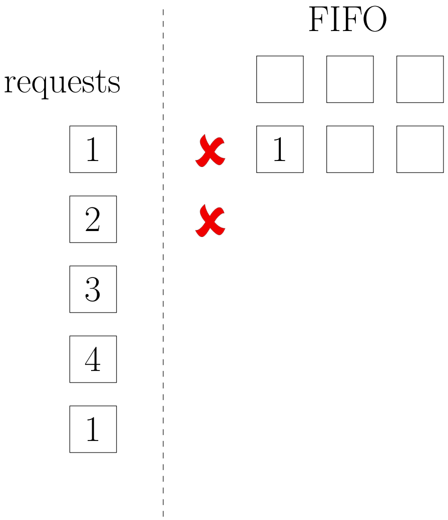
FIFO is not optimum



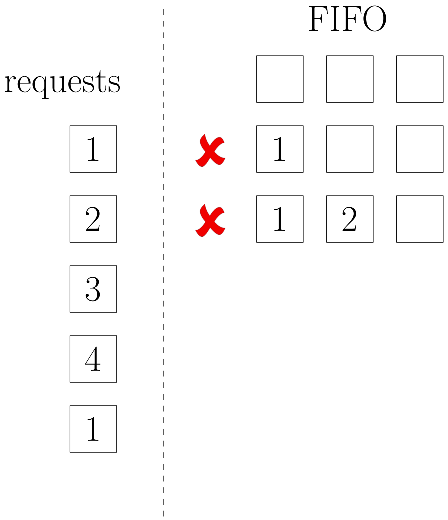
FIFO is not optimum



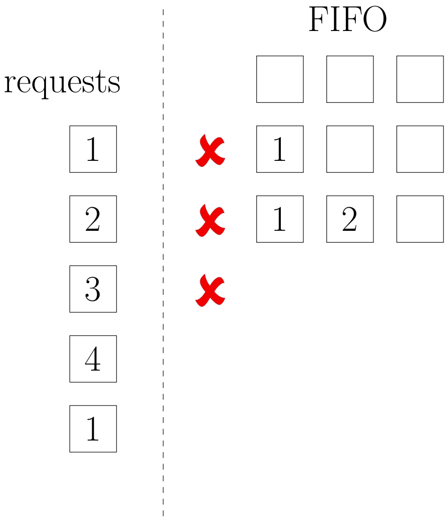
FIFO is not optimum



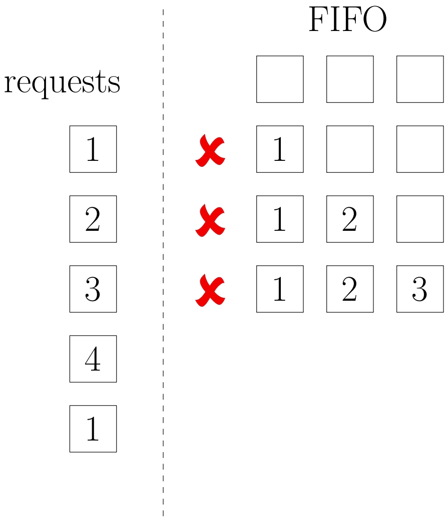
FIFO is not optimum



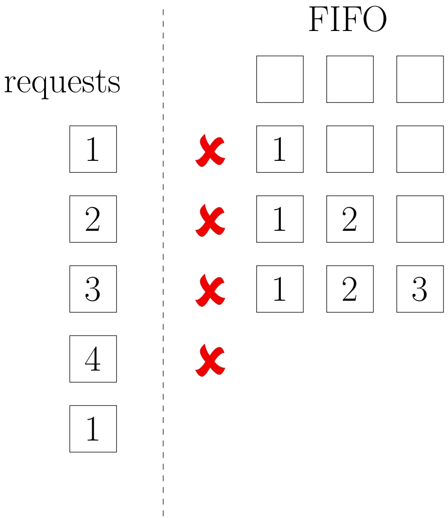
FIFO is not optimum



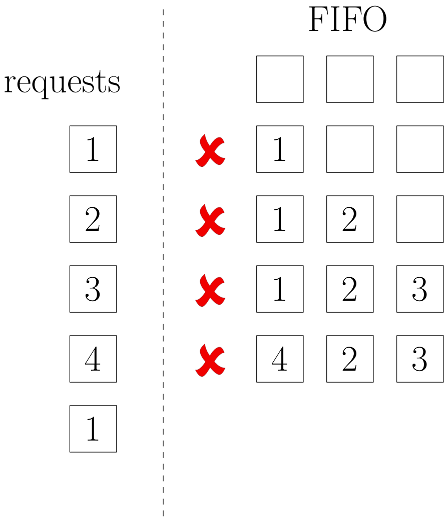
FIFO is not optimum



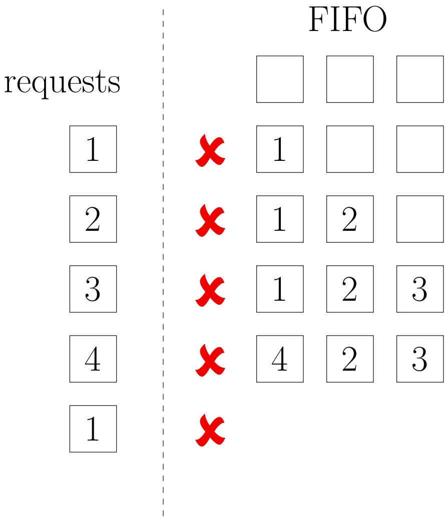
FIFO is not optimum



FIFO is not optimum



FIFO is not optimum



FIFO is not optimum

requests		FIFO		
1	✘	1		
2	✘	1	2	
3	✘	1	2	3
4	✘	4	2	3
1	✘	4	1	3

FIFO is not optimum

requests		FIFO		
1	✗	1		
2	✗	1	2	
3	✗	1	2	3
4	✗	4	2	3
1	✗	4	1	3

misses $= 5$

FIFO is not optimum

requests		FIFO				Furthest-in-Future		
1	✗	1			✗	1		
2	✗	1	2		✗	1	2	
3	✗	1	2	3	✗	1	2	3
4	✗	4	2	3	✗	1	4	3
1	✗	4	1	3	✓	1	4	3
		misses = 5				misses = 4		

Optimum Offline Caching

Furthest-in-Future (FF)

- Algorithm: every time, evict the page that is not requested until furthest in the future, if we need to evict one.
- The algorithm is not an online algorithm, since the decision at a step depends on the request sequence in the future.

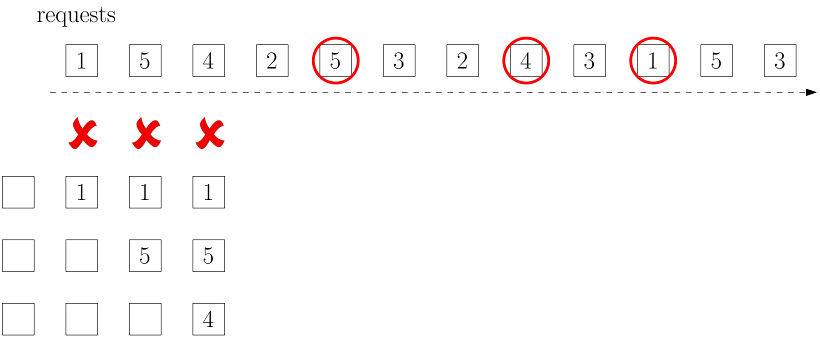
Furthest-in-Future (FF)

requests	FIFO				Furthest-in-Future			
1	✗	1			✗	1		
2	✗	1	2		✗	1	2	
3	✗	1	2	3	✗	1	2	3
4	✗	4	2	3	✗	1	4	3
1	✗	4	1	3	✓	1	4	3
		misses = 5				misses = 4		

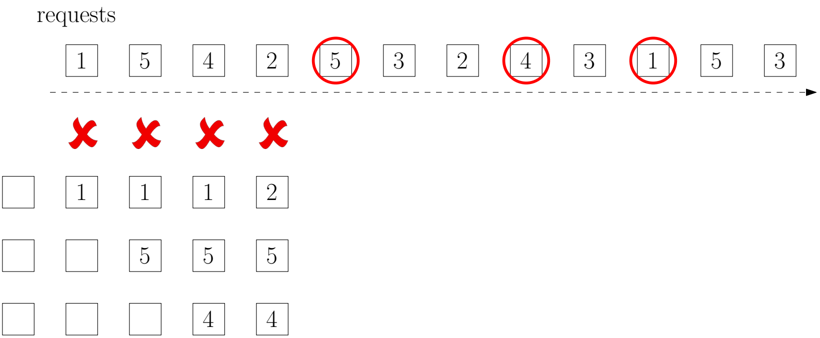
Example

Example

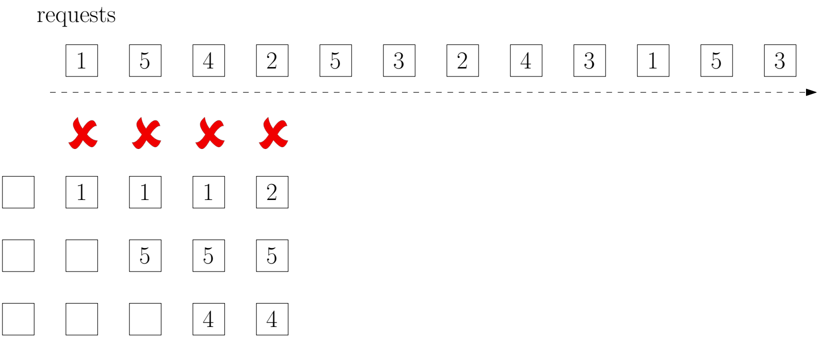
Example



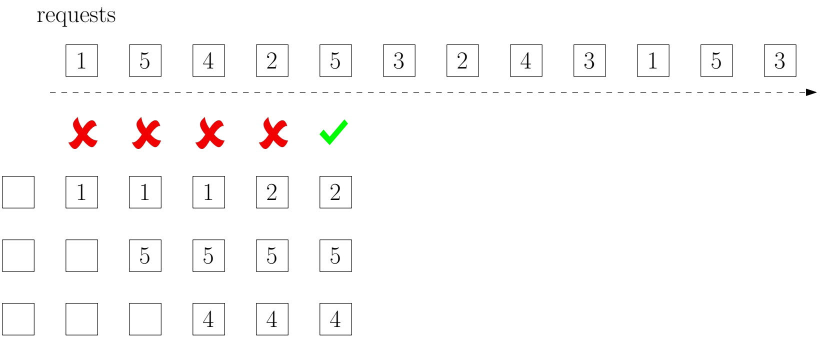
Example



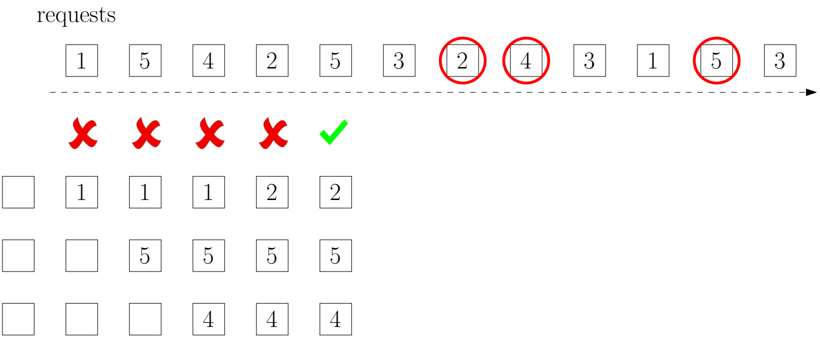
Example



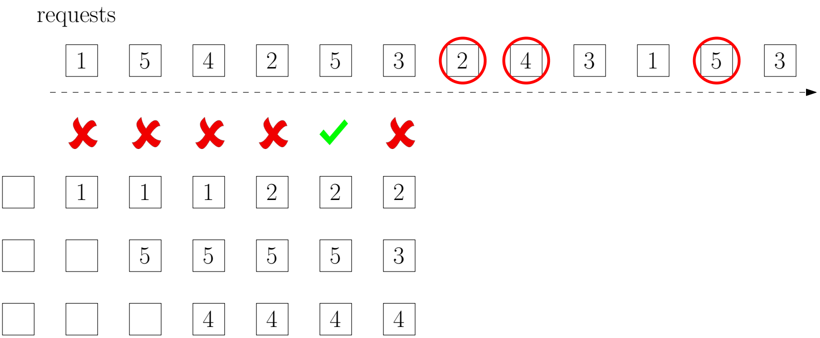
Example



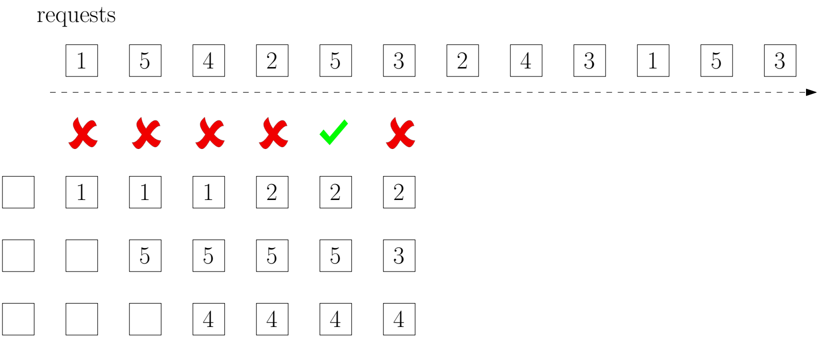
Example



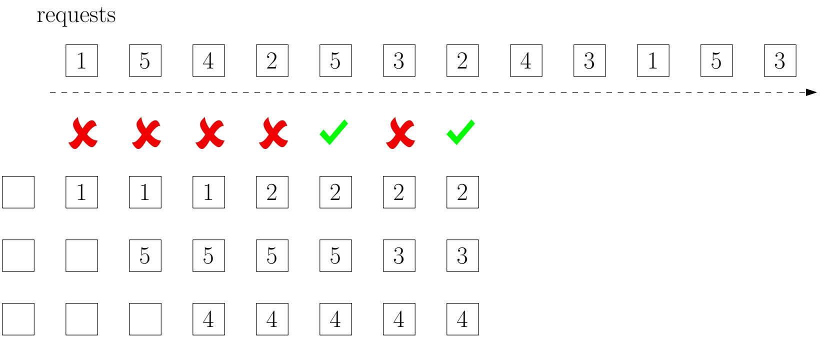
Example



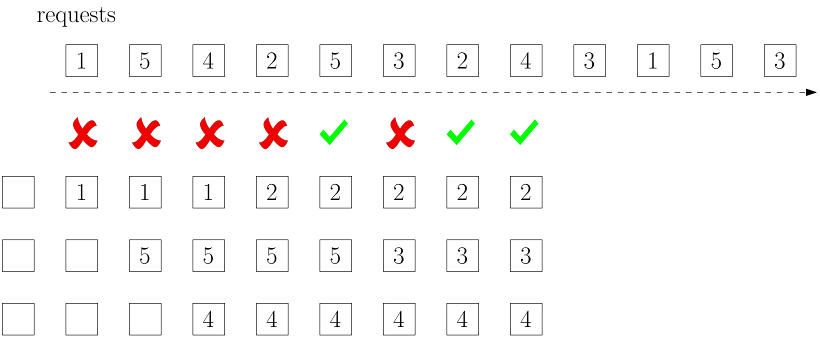
Example



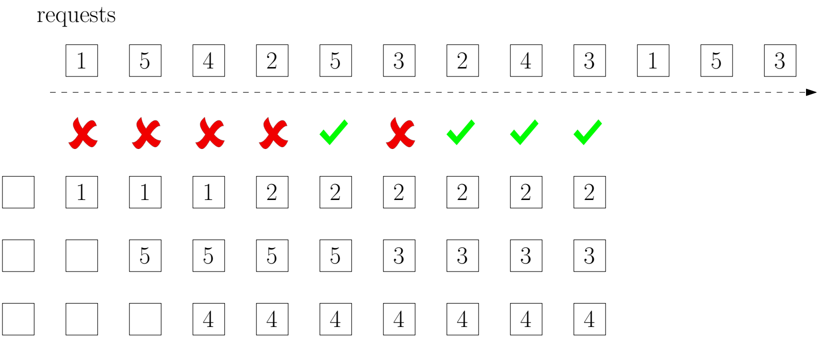
Example



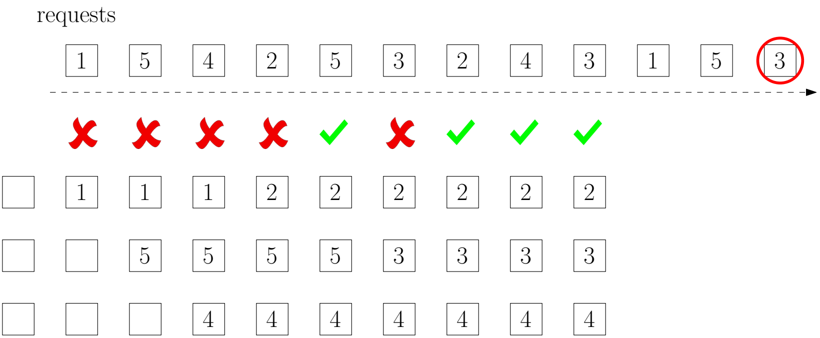
Example



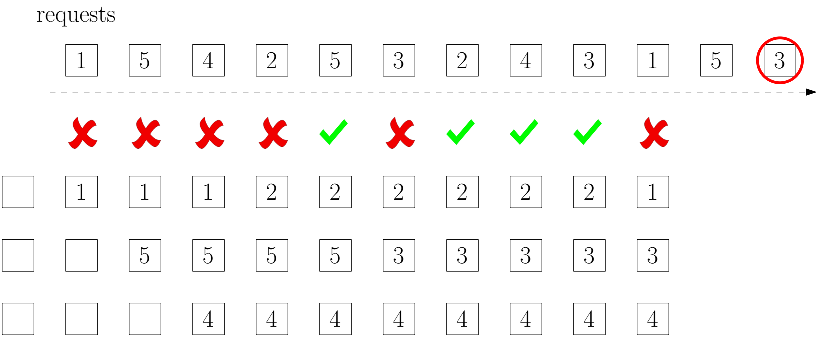
Example



Example



Example



Example

requests

	1	5	4	2	5	3	2	4	3	1	5	3
	✗	✗	✗	✗	✓	✗	✓	✓	✓	✗	✗	
	1	1	1	2	2	2	2	2	2	1	5	
		5	5	5	5	3	3	3	3	3	3	
			4	4	4	4	4	4	4	4	4	

Example

requests

	1	5	4	2	5	3	2	4	3	1	5	3
	✗	✗	✗	✗	✓	✗	✓	✓	✓	✗	✗	✓
	1	1	1	2	2	2	2	2	2	1	5	5
		5	5	5	5	3	3	3	3	3	3	3
			4	4	4	4	4	4	4	4	4	4

Recall: Designing and Analyzing Greedy Algorithms

Greedy Algorithm

- Build up the solutions in steps
- At each step, make an irrevocable decision using a "reasonable" strategy

Analysis of Greedy Algorithm

- Safety: Prove that the reasonable strategy is "safe" (key)
- Self-reduce: Show that the remaining task after applying the strategy is to solve a (many) smaller instance(s) of the same problem (usually easy)

Recall: Designing and Analyzing Greedy Algorithms

Greedy Algorithm

- Build up the solutions in steps
- At each step, make an irrevocable decision using a “reasonable” strategy

Analysis of Greedy Algorithm

- Safety: Prove that the reasonable strategy is “safe” (key)
- Self-reduce: Show that the remaining task after applying the strategy is to solve a (many) smaller instance(s) of the same problem (usually easy)

Offline Caching Problem

Input: k : the size of cache

n : number of pages

$\rho_1, \rho_2, \rho_3, \cdots, \rho_T \in [n]$: sequence of requests

Output: $i_1, i_2, i_3, \cdots, i_t \in \{\mathrm{hit}, \text{empty}\} \cup [n]$

- empty stands for an empty page
- "hit" means evicting no pages

Offline Caching Problem

Input: k : the size of cache

n : number of pages

$\rho_1, \rho_2, \rho_3, \cdots, \rho_T \in [n]$: sequence of requests

$p_1, p_2, \cdots, p_k \in \{\text{empty}\} \cup [n]$: initial set of pages in cache

Output: $i_1, i_2, i_3, \cdots, i_t \in \{\text{hit}, \text{empty}\} \cup [n]$

- empty stands for an empty page
- "hit" means evicting no pages

Analysis of Greedy Algorithm

- Safety: Prove that the reasonable strategy is "safe" (key)
- Self-reduce: Show that the remaining task after applying the strategy is to solve a (many) smaller instance(s) of the same problem (usually easy)

Analysis of Greedy Algorithm

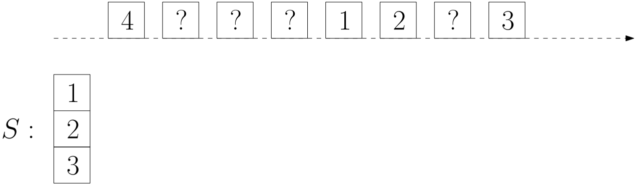
- Safety: Prove that the reasonable strategy is "safe" (key)
- Self-reduce: Show that the remaining task after applying the strategy is to solve a (many) smaller instance(s) of the same problem (usually easy)

Lemma Assume at time 1 a page fault happens and there are no empty pages in the cache. Let p^* be the page in cache that is not requested until furthest in the future. It is safe to evict p^* at time 1.

Analysis of Greedy Algorithm

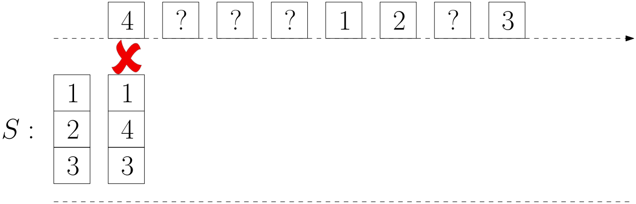
- Safety: Prove that the reasonable strategy is "safe" (key)
- Self-reduce: Show that the remaining task after applying the strategy is to solve a (many) smaller instance(s) of the same problem (usually easy)

Lemma Assume at time 1 a page fault happens and there are no empty pages in the cache. Let p^* be the page in cache that is not requested until furthest in the future. There is an optimum solution in which p^* is evicted at time 1.



Proof.

1. S: any optimum solution
2. p^*: page in cache not requested until furthest in the future.
   - In the example, $p^* = 3$.

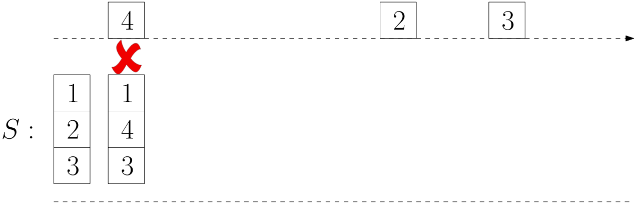


Proof.

1. S: any optimum solution
2. p^*: page in cache not requested until furthest in the future.
   - In the example, $p^* = 3$.
3. Assume S evicts some $p' \neq p^*$ at time 1; otherwise done.
   - In the example, $p' = 2$.

Proof.

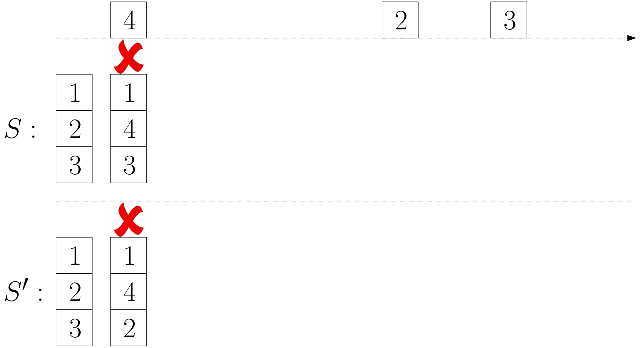
1. S: any optimum solution
2. p^*: page in cache not requested until furthest in the future.
    - In the example, $p^* = 3$.
3. Assume S evicts some $p' \neq p^*$ at time 1; otherwise done.
    - In the example, $p' = 2$.



Proof.

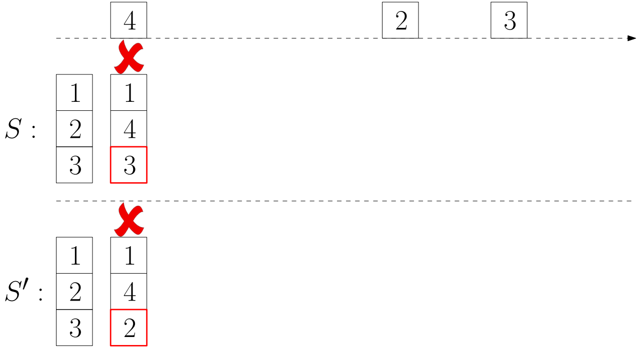
Proof.

4. Create S'. S' evicts $p^*(=3)$ instead of $p'(=2)$ at time 1.



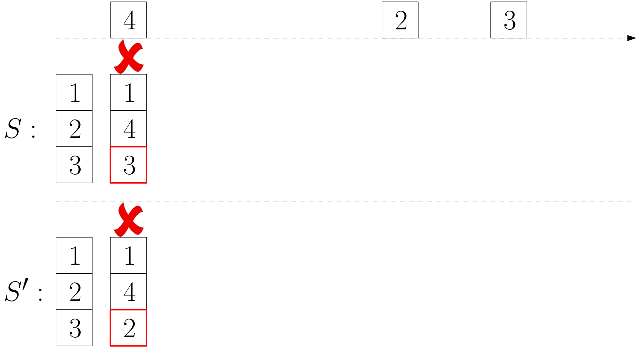
Proof.

4. Create S'. S' evicts $p^*(=3)$ instead of $p'(=2)$ at time 1.



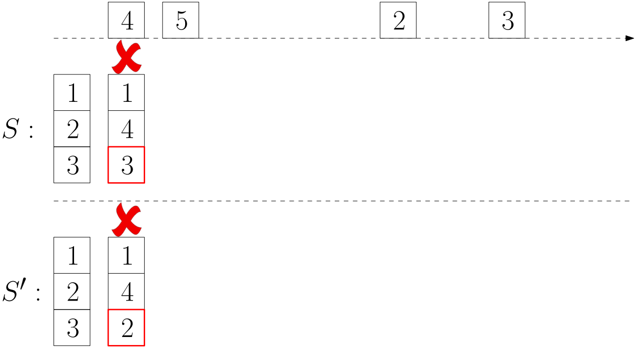
Proof.

4. Create S'. S' evicts $p^*(=3)$ instead of $p'(=2)$ at time 1.
5. After time 1, cache status of S and that of S' differ by only 1 page. S' contains $p'(=2)$ and S contains $p^*(=3)$.



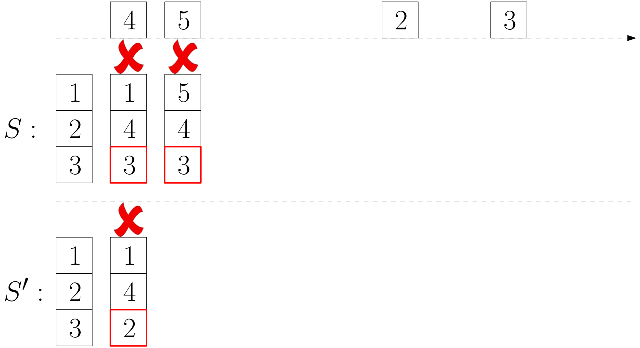
Proof.

4. Create S'. S' evicts $p^*(=3)$ instead of $p'(=2)$ at time 1.
5. After time 1, cache status of S and that of S' differ by only 1 page. S' contains $p'(=2)$ and S contains $p^*(=3)$.
6. From now on, S' will “copy” S.



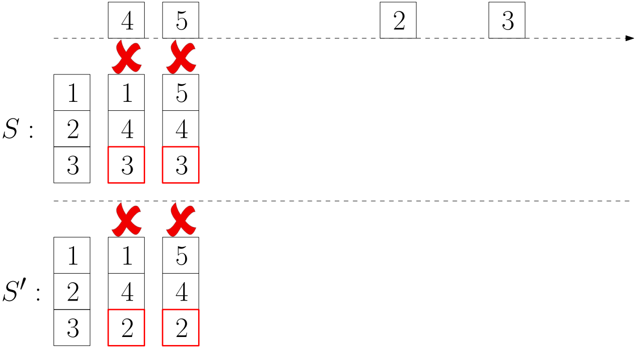
Proof.

4. Create S'. S' evicts $p^*(=3)$ instead of $p'(=2)$ at time 1.
5. After time 1, cache status of S and that of S' differ by only 1 page. S' contains $p'(=2)$ and S contains $p^*(=3)$.
6. From now on, S' will "copy" S.



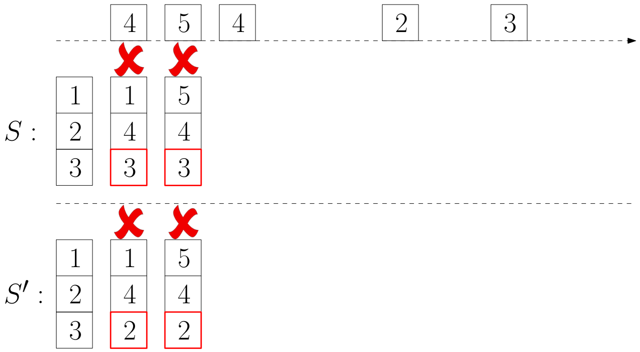
Proof.

1. Create S'. S' evicts $p^*(=3)$ instead of $p'(=2)$ at time 1.
2. After time 1, cache status of S and that of S' differ by only 1 page. S' contains $p'(=2)$ and S contains $p^*(=3)$.
3. From now on, S' will "copy" S.



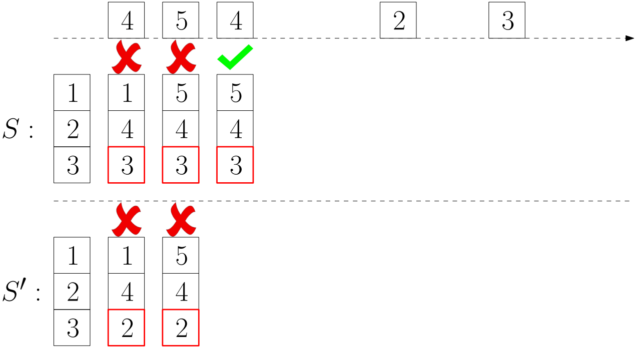
Proof.

4. Create S'. S' evicts $p^*(=3)$ instead of $p'(=2)$ at time 1.
5. After time 1, cache status of S and that of S' differ by only 1 page. S' contains $p'(=2)$ and S contains $p^*(=3)$.
6. From now on, S' will "copy" S.



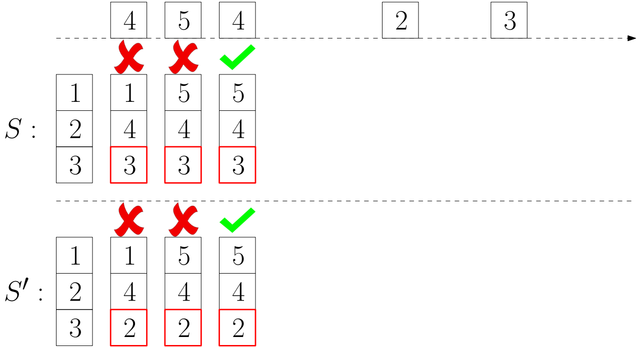
Proof.

4. Create S'. S' evicts $p^*(=3)$ instead of $p'(=2)$ at time 1.
5. After time 1, cache status of S and that of S' differ by only 1 page. S' contains $p'(=2)$ and S contains $p^*(=3)$.
6. From now on, S' will "copy" S.



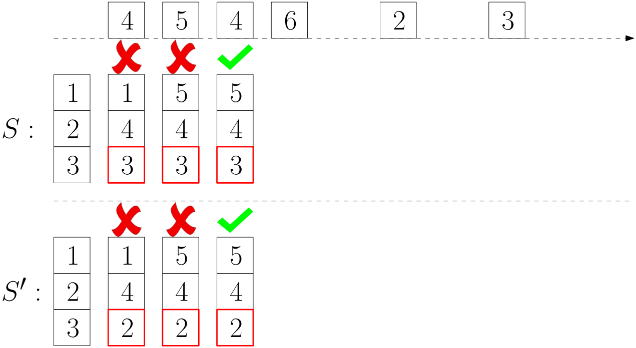
Proof.

1. Create S'. S' evicts $p^*(=3)$ instead of $p'(=2)$ at time 1.
2. After time 1, cache status of S and that of S' differ by only 1 page. S' contains $p'(=2)$ and S contains $p^*(=3)$.
3. From now on, S' will "copy" S.



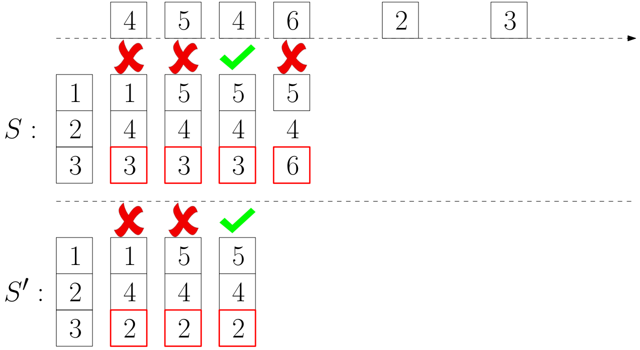
Proof.

4. Create S'. S' evicts $p^*(=3)$ instead of $p'(=2)$ at time 1.
5. After time 1, cache status of S and that of S' differ by only 1 page. S' contains $p'(=2)$ and S contains $p^*(=3)$.
6. From now on, S' will "copy" S.



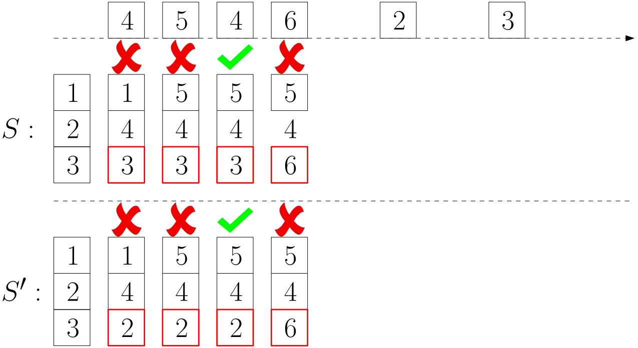
Proof.

4. Create S'. S' evicts $p^*(=3)$ instead of $p'(=2)$ at time 1.
5. After time 1, cache status of S and that of S' differ by only 1 page. S' contains $p'(=2)$ and S contains $p^*(=3)$.
6. From now on, S' will "copy" S.



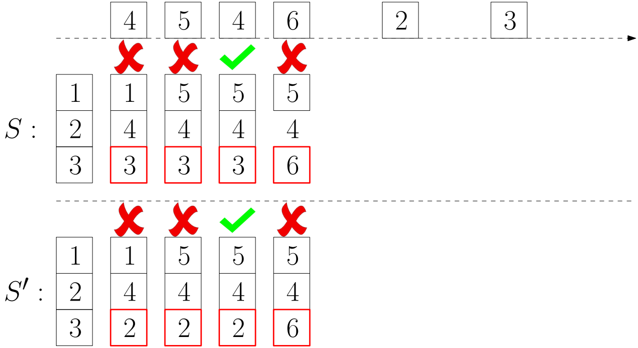
Proof.

4. Create S'. S' evicts $p^*(=3)$ instead of $p'(=2)$ at time 1.
5. After time 1, cache status of S and that of S' differ by only 1 page. S' contains $p'(=2)$ and S contains $p^*(=3)$.
6. From now on, S' will "copy" S.

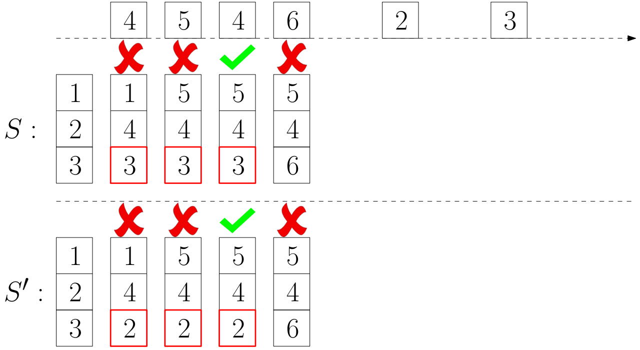


Proof.

4. Create S'. S' evicts $p^*(=3)$ instead of $p'(=2)$ at time 1.
5. After time 1, cache status of S and that of S' differ by only 1 page. S' contains $p'(=2)$ and S contains $p^*(=3)$.
6. From now on, S' will "copy" S.

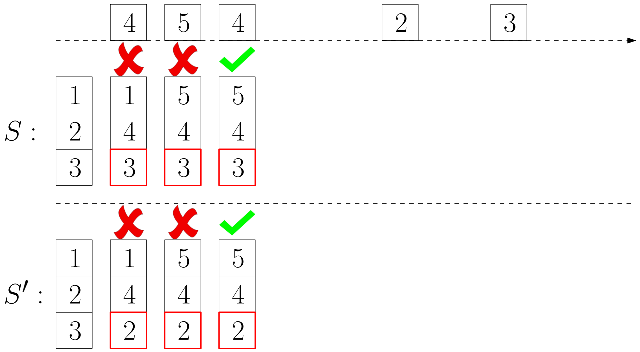


Proof.



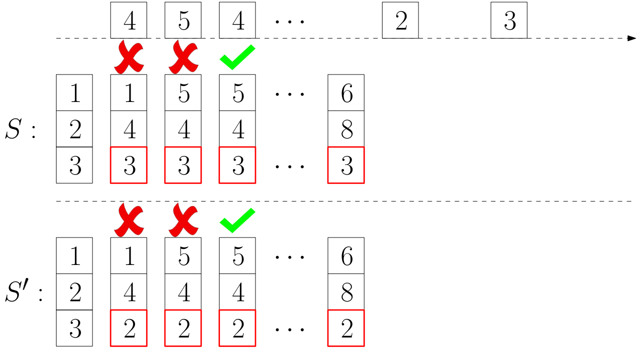
Proof.

7. If S evicted the page p^*, S' will evict the page p'. Then, the cache status of S and that of S' will be the same. S and S' will be exactly the same from now on.



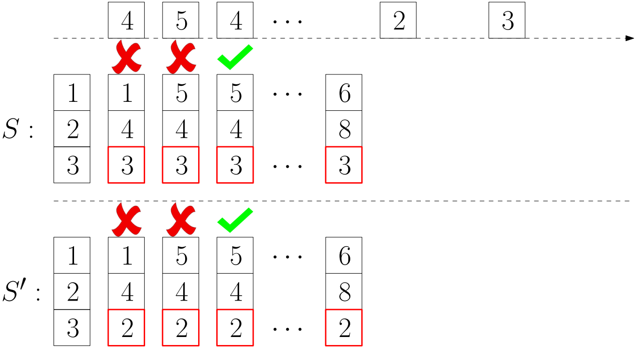
Proof.

7. If S evicted the page p^*, S' will evict the page p'. Then, the cache status of S and that of S' will be the same. S and S' will be exactly the same from now on.
8. Assume S did not evict $p^*(=3)$ before we see $p'(=2)$.

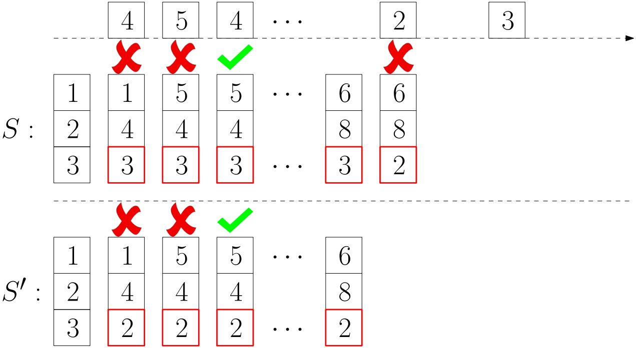


Proof.

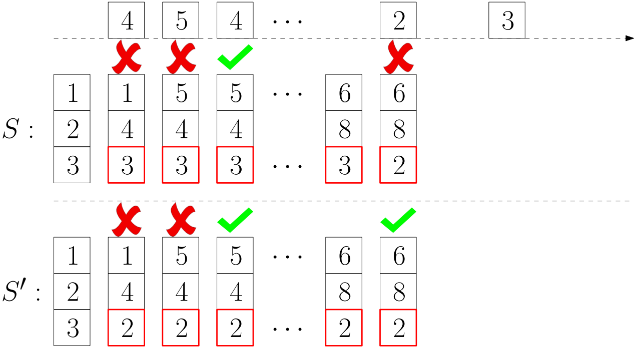
7. If S evicted the page p^*, S' will evict the page p'. Then, the cache status of S and that of S' will be the same. S and S' will be exactly the same from now on.
8. Assume S did not evict $p^*(=3)$ before we see $p'(=2)$.



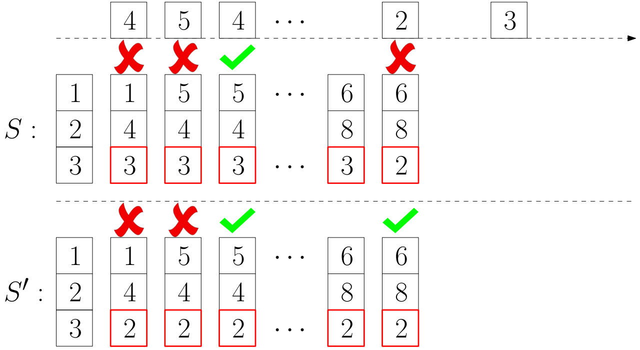
Proof.



Proof.

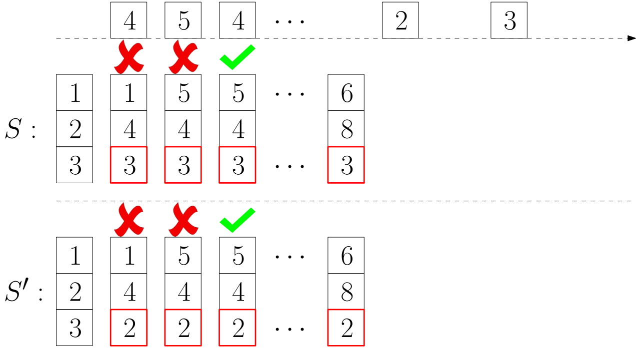


Proof.



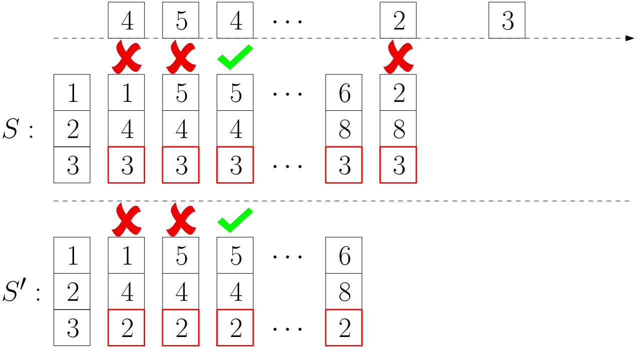
Proof.

1. If S evicts $p^*(=3)$ for $p'(=2)$, then S won't be optimum. Assume otherwise.



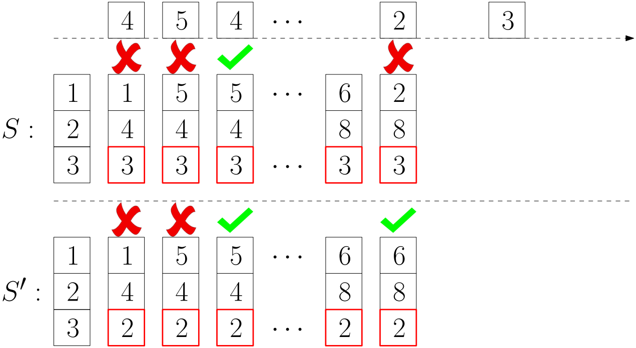
Proof.

9. If S evicts $p^*(=3)$ for $p'(=2)$, then S won't be optimum. Assume otherwise.



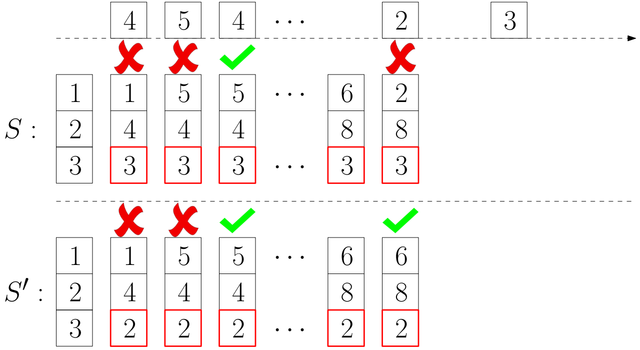
Proof.

9. If S evicts $p^*(=3)$ for $p'(=2)$, then S won't be optimum. Assume otherwise.



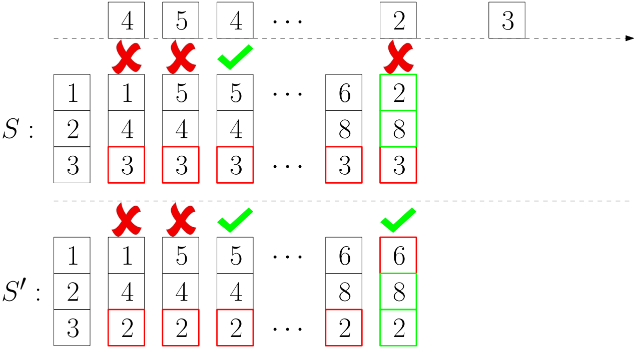
Proof.

9. If S evicts $p^*(=3)$ for $p'(=2)$, then S won't be optimum. Assume otherwise.



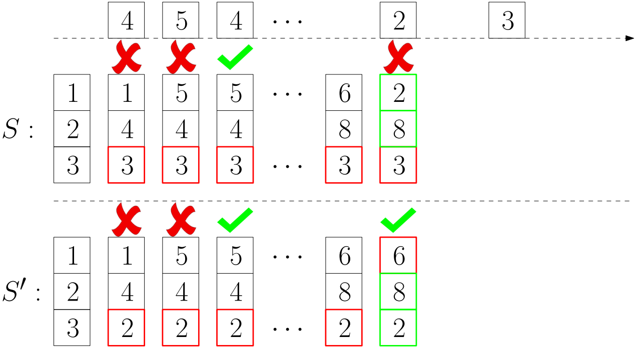
Proof.

9. If S evicts $p^*(=3)$ for $p'(=2)$, then S won't be optimum. Assume otherwise.
10. So far, S' has 1 less page-miss than S does.

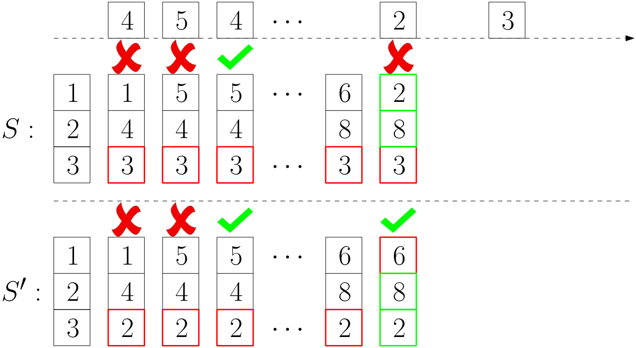


Proof.

9. If S evicts $p^*(=3)$ for $p'(=2)$, then S won't be optimum. Assume otherwise.
10. So far, S' has 1 less page-miss than S does.
11. The status of S' and that of S only differ by 1 page.

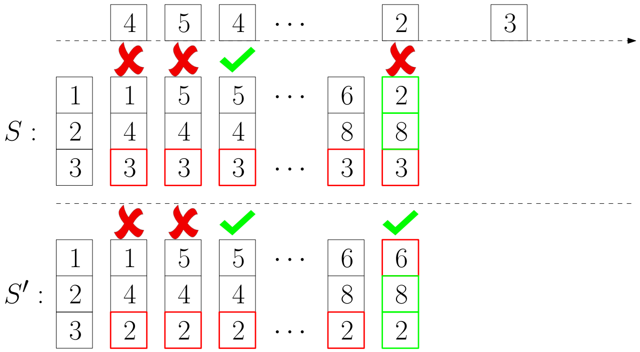


Proof.



Proof.

12 We can then guarantee that S' make at most the same number of page-misses as S does.



Proof.

12. We can then guarantee that S' make at most the same number of page-misses as S does.
    - Idea: if S has a page-hit and S' has a page-miss, we use the opportunity to make the status of S' the same as that of S. □