



# **Chapter 8**

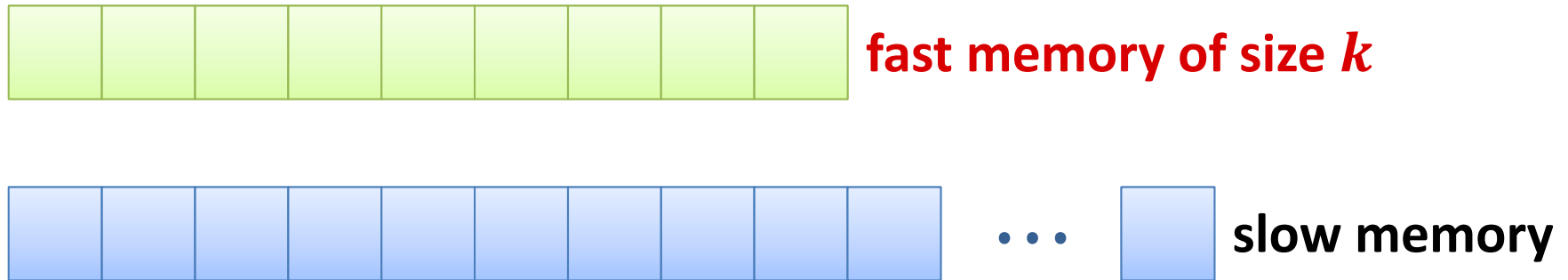
# **Online Algorithms**

**Algorithm Theory**  
**WS 2014/15**

**Fabian Kuhn**

# Paging Algorithm

Assume a simple memory hierarchy:



If a memory page has to be accessed:

- Page in fast memory (hit): take page from there
- Page not fast memory (miss): leads to a page fault
- Page fault: the page is loaded into the fast memory and some page has to be evicted from the fast memory
- Paging algorithm: decides which page to evict
- Classical online problem: we don't know the future accesses

# Paging Strategies

---

## Least Recently Used (**LRU**):

- Replace the page that hasn't been used for the longest time

## First In First Out (**FIFO**):

- Replace the page that has been in the fast memory longest

## Last In First Out (**LIFO**):

- Replace the page most recently moved to fast memory

## Least Frequently Used (**LFU**):

- Replace the page that has been used the least

## Longest Forward Distance (**LFD**):

- Replace the page whose next request is latest (in the future)
- LFD is **not an online strategy!**

# LFD is Optimal

---

**Theorem:** LFD (longest forward distance) is an optimal offline alg.

**Proof:**

- For contradiction, assume that LFD is not optimal
- Then there exists a finite input sequence  $\sigma$  on which LFD is not optimal (assume that the length of  $\sigma$  is  $|\sigma| = n$ )
- Let OPT be an optimal solution for  $\sigma$  such that
  - OPT processes requests  $1, \dots, i$  in exactly the same way as LFD
  - OPT processes request  $i + 1$  differently than LFD
  - Any other optimal strategy processes one of the first  $i + 1$  requests differently than LFD
- Hence, OPT is the optimal solution that behaves in the same way as LFD for as long as possible  $\rightarrow$  we have  $i < n$
- Goal: Construct OPT' that is identical with LFD for req.  $1, \dots, i + 1$

# LFD is Optimal

---

**Theorem:** LFD (longest forward distance) is an optimal offline alg.

**Proof:**

**Case 1:** Request  $i + 1$  does **not** lead to a page fault

- LFD does not change the content of the fast memory
- OPT behaves differently than LFD
  - OPT replaces some page in the fast memory
    - As up to request  $i + 1$ , both algorithms behave in the same way, they also have the same fast memory content
    - OPT therefore does not require the new page for request  $i + 1$
    - Hence, OPT can also load that page later (without extra cost) → OPT'

# LFD is Optimal

---

**Theorem:** LFD (longest forward distance) is an optimal offline alg.

**Proof:**

**Case 2:** Request  $i + 1$  does lead to a **page fault**

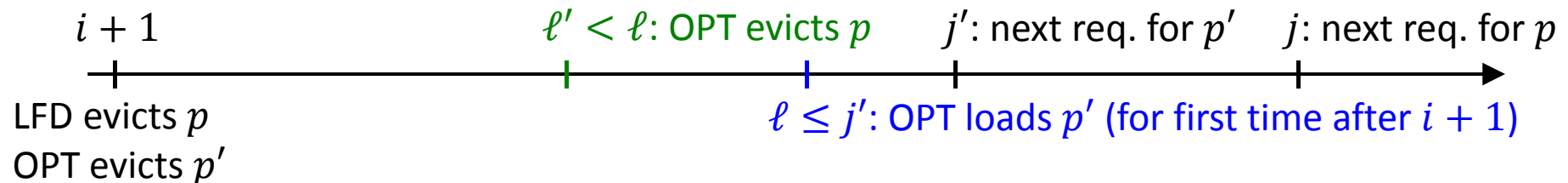
- LFD and OPT move the same page into the fast memory, but they evict different pages
  - If OPT loads more than one page, all pages that are not required for request  $i + 1$  can also be loaded later
- Say, LFD evicts page  $p$  and OPT evicts page  $p'$
- By the definition of LFD,  $p'$  is required again before page  $p$

# LFD is Optimal

**Theorem:** LFD (longest forward distance) is an optimal offline alg.

**Proof:**

**Case 2:** Request  $i + 1$  does lead to a **page fault**



- a) OPT keeps  $p$  in fast memory until request  $\ell$ 
  - Evict  $p$  at request  $i + 1$ , keep  $p'$  instead and load  $p$  (instead of  $p'$ ) back into the fast memory at request  $\ell$
- b) OPT evicts  $p$  at request  $\ell' < \ell$ 
  - Evict  $p$  at request  $i + 1$  and  $p'$  at request  $\ell'$  (switch evictions of  $p$  and  $p'$ )

# Phase Partition

We **partition** a given **request sequence**  $\sigma$  into phases as follows:

- **Phase 0**: empty sequence
- **Phase  $i$** : maximal sequence that immediately follows phase  $i - 1$  and contains at most  $k$  distinct page requests

**Example sequence ( $k = 4$ ):**

2, 5, 12, 5, 4, 2, 10, 8, 3, 6, 2, 2, 6, 6, 8, 3, 2, 6, 9, 10, 6, 3, 10, 2, 1, 3, 5

**Phase  $i$  Interval**: interval starting with the second request of phase  $i$  and ending with the first request of phase  $i + 1$

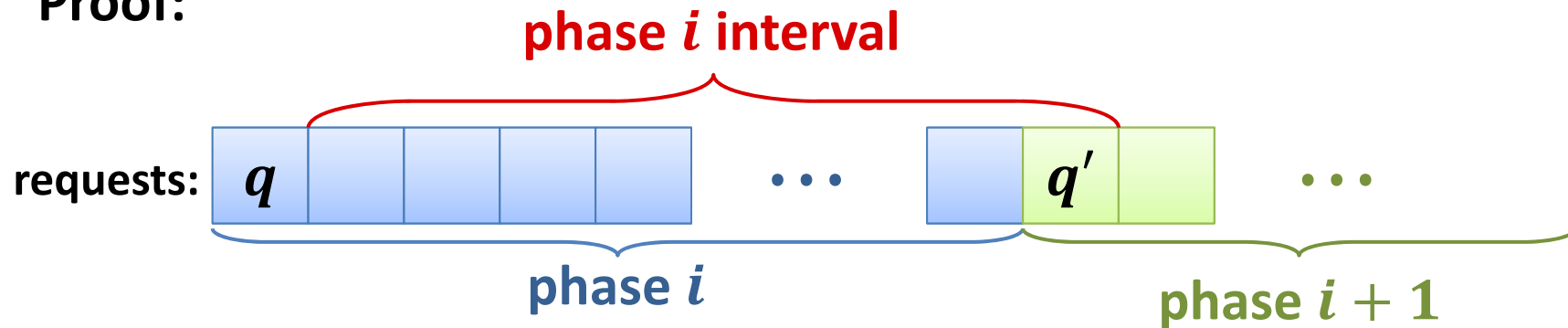
- If the last phase is phase  $p$ , phase-interval  $i$  is defined for  $i = 1, \dots, p - 1$



# Optimal Algorithm

**Lemma:** Algorithm LFD has at least one page fault in each phase  $i$  interval (for  $i = 1, \dots, p - 1$ , where  $p$  is the number of phases).

**Proof:**



- $q$  is in fast memory after first request of phase  $i$
- Number of distinct requests in phase  $i$ :  $k$
- By maximality of phase  $i$ :  $q'$  does not occur in phase  $i$
- Number of distinct requests  $\neq q$  in phase interval  $i$ :  $k$

→ at least one page fault

# LRU and FIFO Algorithms

---

**Lemma:** Algorithm LFD has at least one page fault in each phase interval  $i$  (for  $i = 1, \dots, p - 1$ , where  $p$  is the number of phases).

**Corollary:** The number of page faults of an optimal offline algorithm is at least  $p - 1$ , where  $p$  is the number of phases

**Theorem:** The LRU and the FIFO algorithms both have a competitive ratio of at most  $k$ .

**Proof:**

- In phase  $i$  only pages from phases before phase  $i$  are evicted from the fast memory  $\rightarrow \leq k$  page faults per phase
  - As long as not all  $k$  pages from phase  $i$  have been requested, the least recently used and the first inserted are from phases before  $i$
  - When all  $k$  pages have been requested, the  $k$  pages of phase  $i$  are in fast memory and there are no more page faults in phase  $i$

# Lower Bound

---

**Theorem:** Even if the slow memory contains only  $k + 1$  pages, any deterministic algorithm has competitive ratio at least  $k$ .

## Proof:

- Consider some given deterministic algorithm ALG
- Because ALG is deterministic, the content of the fast memory after the first  $i$  requests is determined by the first  $i$  requests.
- Construct a request sequence inductively as follows:
  - Assume some initial slow memory content
  - The  $(i + 1)^{\text{st}}$  request is for the page which is not in fast memory after the first  $i$  requests (throughout we only use  $k + 1$  different pages)
- There is a page fault for every request
- OPT has a page fault at most every  $k$  requests
  - There is always a page that is not required for the next  $k - 1$  requests

# Randomized Algorithms

---

- We have seen that deterministic paging algorithms cannot be better than  $k$ -competitive
- Does it help to use randomization?

**Competitive Ratio:** A randomized online algorithm has competitive ratio  $c \geq 1$  if for all inputs  $I$ ,

$$\mathbb{E}[\mathbf{ALG}(I)] \leq c \cdot \mathbf{OPT}(I) + \alpha.$$

- If  $\alpha \leq 0$ , we say that ALG is **strictly  $c$ -competitive**.

# Adversaries

---

- For randomized algorithm, we need to distinguish between different kinds of adversaries (providing the input)

## **Oblivious Adversary:**

- Has to determine the complete input sequence before the algorithm starts
  - The adversary cannot adapt to random decisions of the algorithm

## **Adaptive Adversary:**

- The adversary knows how the algorithm reacted to earlier inputs
- **online adaptive:** adversary has no access to the randomness used to react to the current input
- **offline adaptive:** adversary knows the random bits used by the algorithm to serve the current input

# Lower Bound

---

The adversaries can be ordered according to their strength

oblivious  $<$  online adaptive  $<$  offline adaptive

- An algorithm that works with an adaptive adversary also works with an oblivious one
- A lower bound that holds against an oblivious adversary also holds for the other 2
- ...

**Theorem:** No randomized paging algorithm can be better than  $k$ -competitive against an online (or offline) adaptive adversary.

**Proof:** The same proof as for deterministic algorithms works.

- Are there better algorithms with an oblivious adversary?

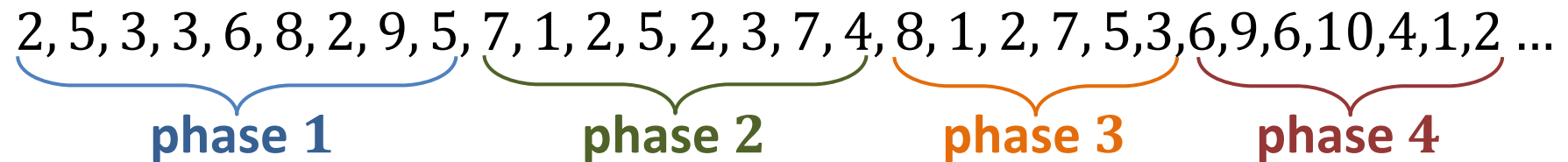
# The Randomized Marking Algorithm

---

- Every entry in fast memory has a marked flag
- Initially, all entries are unmarked.
- If a page in fast memory is accessed, it gets marked
- When a **page fault** occurs:
  - If all  $k$  pages in fast memory are marked, all marked bits are set to 0
  - The page to be evicted is chosen uniformly at random among the unmarked pages
  - The marked bit of the new page in fast memory is set to 1

# Example

**Input Sequence (k=6):**



**Fast Memory:**



**Observations:**

- At the end of a phase, the fast memory entries are exactly the  $k$  pages of that phase
- At the beginning of a phase, all entries get unmarked
- #page faults depends on #new pages in a phase



# Page Faults per Phase

---

## Consider a fixed phase $i$ :

- Assume that of the  $k$  pages of phase  $i$ ,  $m_i$  are **new** and  $k - m_i$  are **old** (i.e., they already appear in phase  $i - 1$ )
- All  $m_i$  new pages lead to page faults (when they are requested for the first time)
- When requested for the first time, an old page leads to a page fault, if the page was evicted in one of the previous page faults
- We need to count the number of page faults for old pages

# Page Faults per Phase

**Phase  $i$ ,  $j^{\text{th}}$  old page that is requested (for the first time):**

- There is a page fault if the page has been evicted
- There have been at most  $m_i + j - 1$  distinct requests before
- The old places of the  $j - 1$  first old pages are occupied
- The other  $\leq m_i$  pages are at uniformly random places among the remaining  $k - (j - 1)$  places (oblivious adv.)
- Probability that the old place of the  $j^{\text{th}}$  old page is taken:

$$\leq \frac{m_i}{k - (j - 1)}$$

# Page Faults per Phase

Phase  $i > 1$ ,  $j^{\text{th}}$  old page that is requested (for the first time):

- Probability that there is a page fault:

$$\leq \frac{m_i}{k - (j - 1)}$$

Number of page faults for old pages in phase  $i$ :  $F_i$

$$\begin{aligned} \mathbb{E}[F_i] &= \sum_{j=1}^{k-m_i} \mathbb{P}(j^{\text{th}} \text{ old page incurs page fault}) \\ &\leq \sum_{j=1}^{k-m_i} \frac{m_i}{k - (j - 1)} = m_i \cdot \sum_{\ell=m_i+1}^k \frac{1}{\ell} \\ &= m_i \cdot (H(k) - H(m_i)) \leq m_i \cdot (H(k) - 1) \end{aligned}$$

# Competitive Ratio

**Theorem:** Against an oblivious adversary, the randomized marking algorithm has a competitive ratio of at most  $2H(k) \leq 2 \ln(k) + 2$ .

**Proof:**

- Assume that there are  $p$  phases
- #page faults of rand. marking algorithm in phase  $i$ :  $F_i + m_i$

- We have seen that

$$\mathbb{E}[F_i] \leq m_i \cdot (H(k) - 1) \leq m_i \cdot \ln(k)$$

- Let  $F$  be the total number of page faults of the algorithm:

$$\mathbb{E}[F] \leq \sum_{i=1}^p (\mathbb{E}[F_i] + m_i) \leq H(k) \cdot \sum_{i=1}^p m_i$$

# Competitive Ratio

**Theorem:** Against an oblivious adversary, the randomized marking algorithm has a competitive ratio of at most  $2H(k) \leq 2 \ln(k) + 2$ .

**Proof:**

- Let  $F_i^*$  be the number of page faults in phase  $i$  in an opt. exec.
- Phase 1:  $m_1$  pages have to be replaced  $\rightarrow F_1^* \geq m_1$
- Phase  $i > 1$ :
  - Number of distinct page requests in phases  $i - 1$  and  $i$ :  $k + m_i$
  - Therefore,  $F_{i-1}^* + F_i^* \geq m_i$
- Total number of page requests  $F^*$ :

$$F^* = \sum_{i=1}^p F_i^* \geq \frac{1}{2} \cdot \left( F_1^* + \sum_{i=2}^p (F_{i-1}^* + F_i^*) \right) \geq \frac{1}{2} \cdot \sum_{i=1}^p m_i$$

# Competitive Ratio

**Theorem:** Against an oblivious adversary, the randomized marking algorithm has a competitive ratio of at most  $2H(k) \leq 2 \ln(k) + 2$ .

**Proof:**

- Randomized marking algorithm:

$$\mathbb{E}[F] \leq H(k) \cdot \sum_{i=1}^p m_i$$

- Optimal algorithm:

$$F^* \geq \frac{1}{2} \cdot \sum_{i=1}^p m_i$$

**Remark:** It can be shown that no randomized algorithm has a competitive ratio better than  $H(k)$  (against an obl. adversary)

# Self-Adjusting Lists

---

- Linked lists are often inefficient
  - Cost of accessing an item at position  $i$  is linear in  $i$
- But, linked lists are extremely simple
  - And therefore nevertheless interesting
- Can we at least improve the behavior of linked lists?
- In practical applications, not all items are accessed equally often and not equally distributed over time
  - The same items might be used several times over a short period of time
- **Idea:** rearrange list after accesses to optimize the structure for future accesses
- **Problem:** We don't know the future accesses
  - The list rearrangement problems is an online problem!

# Model

---

- Only find operations (i.e., access some item)
  - Let's ignore insert and delete operations
  - Results can be generalized to cover insertions and deletions

## Cost Model:

- Accessing item at position  $i$  costs  $i$
- The only operation allowed for rearranging the list is swapping two adjacent list items
- Swapping any two adjacent items costs 1



# Rearranging The List

---

## Frequency Count (FC):

- For each item keep a count of how many times it was accessed
- Keep items in non-increasing order of these counts
- After accessing an item, increase its count and move it forward past items with smaller count

## Move-To-Front (MTF):

- Whenever an item is accessed, move it all the way to the front

## Transpose (TR):

- After accessing an item, swap it with its predecessor

# Cost

---

## Cost when accessing item at position $i$ :

- Frequency Count (FC): between  $i$  and  $2i - 1$
- Move-To-Front (MTF):  $2i - 1$
- Transpose (TR):  $i + 1$

## Random Accesses:

- If each item  $x$  has an access probability  $p_x$  and the items are accessed independently at random using these probabilities, FC and TR are asymptotically optimal

Real access patterns are not random, TR usually behaves badly and the much simpler MTF often beats FC

# Move-To-Front

---

- We will see that MTF is competitive
- To analyze MTF we need **competitive analysis** and **amortized analysis**

## Operation $k$ :

- Assume, the operation accesses item  $x$  at position  $i$
- $c_k$ : actual cost of the MTF algorithm  
$$c_k = 2i - 1$$
- $a_k$ : amortized cost of the MTF algorithm
- $c_k^*$ : actual cost of an optimal offline strategy
  - Let's call the optimal offline strategy OPT