Chapter 9
Online Algorithms

Algorithm Theory
WS 2015/16

Fabian Kuhn

Online Computations

I → ALG → O

- Sometimes, an algorithm has to start processing the input before the complete input is known

- For example, when storing data in a data structure, the sequence of operations on the data structure is not known

Online Algorithm: An algorithm that has to produce the output step-by-step when new parts of the input become available.

Offline Algorithm: An algorithm that has access to the whole input before computing the output.

- Some problems are inherently online
  - Especially when real-time requests have to be processed over a significant period of time

Competitive Ratio

- Let's again consider optimization problems
  - For simplicity, assume, we have a minimization problem

Optimal offline solution $\mathbf{OPT}(\boldsymbol{I})$:

- Best objective value that an offline algorithm can achieve for a given input sequence I

Online solution $\mathbf{ALG}(\boldsymbol{I})$:

- Objective value achieved by an online algorithm ALG on I

Competitive Ratio: An algorithm has competitive ratio $c \geq 1$ if

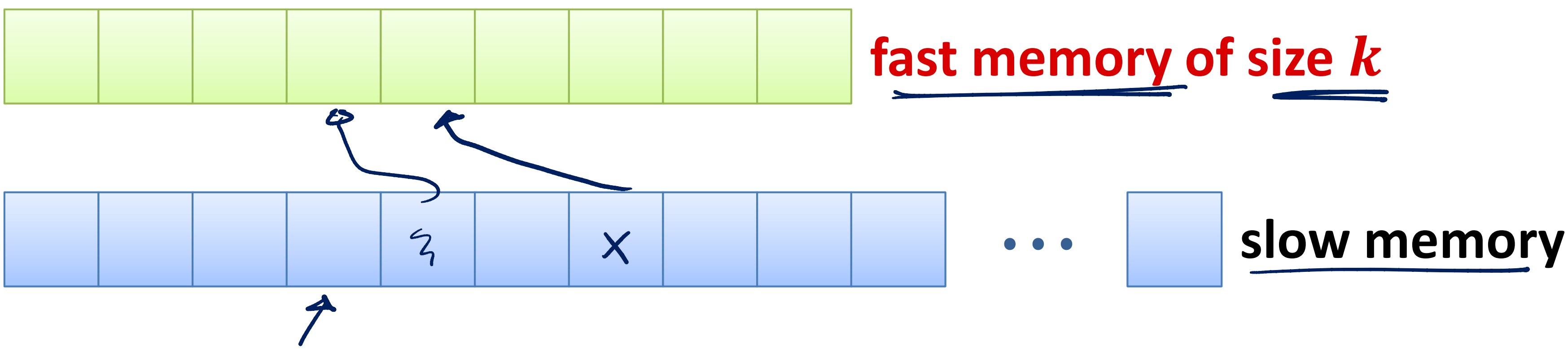
$$\mathbf{ALG}(\boldsymbol{I}) \leq \boldsymbol{c} \cdot \mathbf{OPT}(\boldsymbol{I}) + \boldsymbol{\alpha}.$$

additive term

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

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 in 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

OPT

Longest Forward Distance (LFD): not an online strategy

- 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: σ: seq. of pages

- For contradiction, assume that LFD is not optimal
- Then there exists a finite input sequence σ on which LFD is not optimal (assume that the length of σ is $|\sigma| = n$)
- Let OPT be an optimal solution for σ 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 → we have $i < n$
- Goal: Construct OPT′ that is identical with LFD for req. $1, \dots, i + 1$

LFD is Optimal

same fast mem. content

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

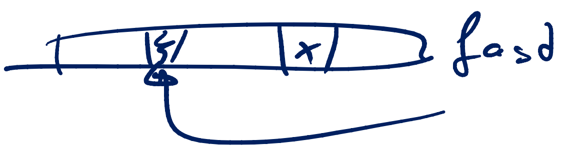
Proof:

OPT : 1, 2, ..., i, $i+1$

LFT : 1, 2, ..., i, $i+1$

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′

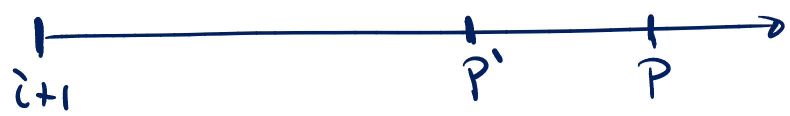
LFD is Optimal

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**

$j > j'$

p is still in mem.

$i+1$ | $\ell' < \ell$: OPT evicts p | j': next req. for p' | j: next req. for p

LFD evicts p
OPT evicts p'

p is mem

p,p' evicted

$\ell \leq j'$: OPT loads p' (for first time after $i+1$)

a) OPT keeps p in fast memory until request ℓ

– Evict p at request $i+1$, keep p' instead and load p (instead of p') back into the fast memory at request ℓ

b) OPT evicts p at request $\ell' < \ell$

– Evict p at request $i+1$ and p' at request ℓ' (switch evictions of p and p')

$\square$

Phase Partition

We partition a given request sequence σ into phases as follows:

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

Example sequence ($\boldsymbol{k=4}$):

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

phase 1 phase 2 phase 3 phase 4 phase 5

phase 1 Interval phase 2 int 3

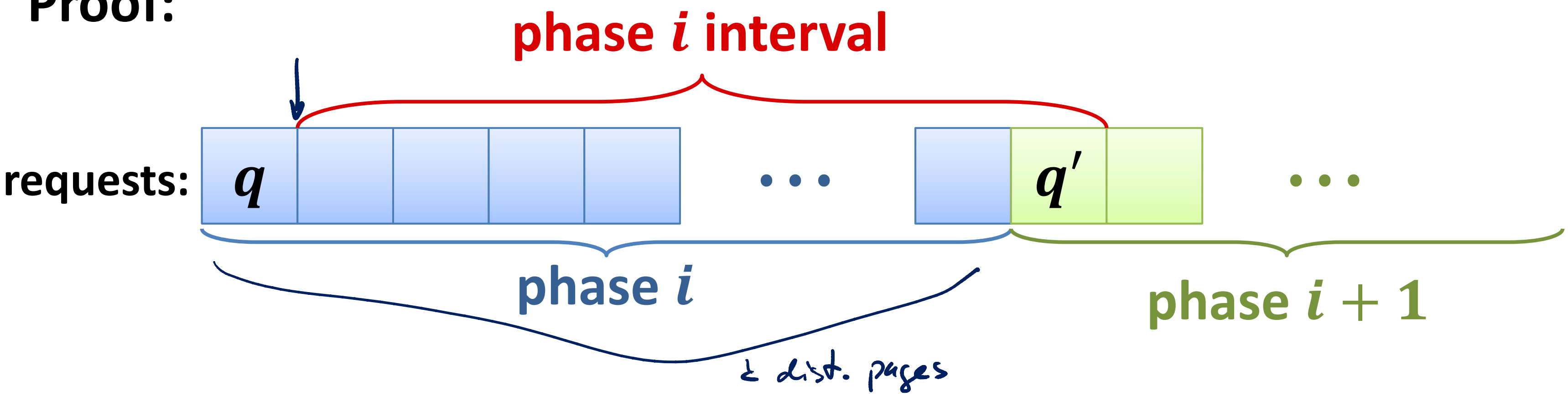
Phase $\boldsymbol{i}$ Interval: interval starting with the second request of phase i and ending with the first request of phase $i+1$

- If the last phase is phase p, phase i interval 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 i interval (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:

phase i

- We will show that both have at most k page faults per phase
- We then have (for every input I):

$$\mathrm{LRU}(I), \mathrm{FIFO}(I) \le k \cdot p \le k \cdot \mathrm{OPT}(I) + k$$

LRU and FIFO Algorithms

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

Proof:

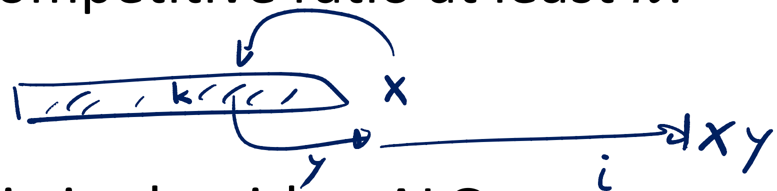
- Need to show that both have at most k page faults per phase
- LRU:
  - The k last distinct pages used are the k ~~least~~ most recently used
  - Throughout a phase i, the k distinct pages of phase i are the ~~l~~m.r.u.
  - Once in the fast memory, these pages are therefore not evicted until the end of the phase
- FIFO:
  - In each page fault in phase i, one of the k pages of phase i is loaded into fast memory
  - Once a page is loaded in a page fault of phase i it belongs to the ~~least~~ most recent $k-1$ pages loaded into fast memory throughout the rest of the phase
  - Hence: Each of the k pages leads to ≤ 1 page fault in phase i

Lower Bound LRU

$\frac{k}{k-h+1}$

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~~ fast 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

  $k-1$
  - 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, adapt. adv. ! random var.

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

- 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 input sequence is constructed during the execution
- When determining the next input, the adversary knows how the algorithm reacted to the previous inputs
- Input sequence depends on the random behavior of the alg.
- Sometimes, two adaptive adversaries are distinguished
  - offline, online : different way of measuring the adversary cost

Lower Bound

The adversaries can be ordered according to their strength

$$\text{oblivious} < \text{online adaptive} < \text{offline adaptive}$$

- An algorithm that achieves a given comp. ratio with an adaptive adversary is at least as good with an oblivious one
- A lower bound that holds against an oblivious adversary also holds for the two adaptive adversaries
- …

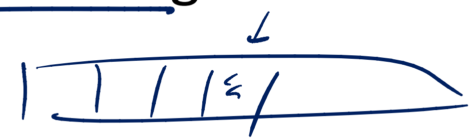
Theorem: No randomized paging algorithm can be better than k-competitive against an adaptive adversary.

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):

$\cancel{2}, 5, 3, 3, 6, 8, 2, 9, 5, 7, 1, 2, 5, 2, 3, 7, 4, 8, 1, 2, 7, 5, 3, 6, 9, 6, 10, 4, 1, 2 \ldots$

phase 1   phase 2   phase 3   phase 4

Fast Memory:

10	6	1	9	4	2

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 $\boxed{k - m_i}$ 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

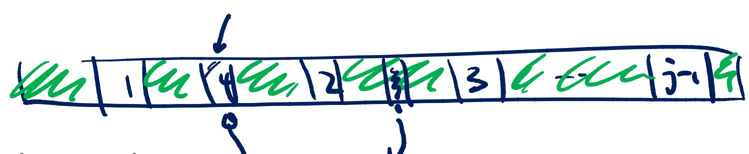
Page Faults per Phase

2, 3, 3, 4, 2, 5, 3

Phase i, j^{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^{th} old page is taken:

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



Page Faults per Phase

$F_{ij} = 1$ if j^{th} old page leads to page fault

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

- Probability that there is a page fault:

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

$F_i = \sum_{j=1}^{k-m_i} F_{ij}$

$\mathbb{E}[F_i] = \sum \mathbb{E}[F_{ij}]$

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

$$\mathbb{E}[F_i] = \sum_{j=1}^{k-m_i} \underbrace{\mathbb{P}\big(j^{\text{th}} \text{ old page incurs page fault}\big)}_{\mathbb{E}[F_{ij}]}$$

$$\leq \sum_{j=1}^{k-m_i} \frac{m_i}{k-(j-1)} = m_i \cdot \sum_{\ell=m_i+1}^{k} \frac{1}{\ell}$$

$$= m_i \cdot \big(H(k) - H(m_i)\big) \leq m_i \cdot (H(k) - 1)$$

$H(t) = 1 + \frac{1}{2} + \ldots + \frac{1}{t}$

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

$1 + \frac{1}{2} + \ldots \frac{1}{m_i}$

$m_i \geq 1$

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:

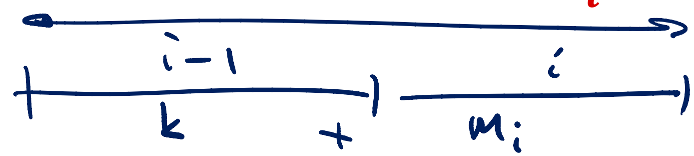
- Assume that there are p phases
- #page faults of rand. marking algorithm in phase i: $F_i + m_i$
- We have seen that
$$\mathbb{E}[F_i] \leq m_i \cdot (H(k) - 1) \leq m_i \cdot \ln(k)$$
- Let F be the total number of page faults of the algorithm:
$$\mathbb{E}[F] \leq \sum_{i=1}^{p} \underbrace{(\mathbb{E}[F_i] + m_i)}_{\leq m_i H(k)} \leq H(k) \cdot \sum_{i=1}^{p} m_i$$

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 replaces → $F_1^* \geq m_1$
- Phase $i > 1$:
  - Number of distinct page requests in phases $i-1$ and i: $\boldsymbol{k + m_i}$
  - Therefore, $\boldsymbol{F_{i-1}^* + F_i^* \geq m_i}$



- Total number of page requests F^*:

$$F^* = \sum_{i=1}^{p} F_i^* \geq \frac{1}{2} \cdot \Bigg(\underbrace{\underbrace{F_1^*}_{\geq m_1} + \sum_{i=2}^{p} \underbrace{(F_{i-1}^* + F_i^*)}_{\geq m_i}}_{\geq \sum m_i} \Bigg) \geq \frac{1}{2} \cdot \sum_{i=1}^{p} m_i$$

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:

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

k

$2H(k)$

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