Randomized (Online) Algorithms

Part 1

Bob Krekelberg

Algorithms for Decision Support 2024

Organization

- Today and Tueseday's lecture by me.
- **No office hours on Monday.**
- Questions via Teams message or after lecture.

Table of Contents

Table of Contents

When deterministic algorithms do not suffice

Motivation

- Natural hard inputs may exist, that **always** cause a deterministic algorithm to perform poorly.
- It may be acceptable if we can guarantee that, for every given input, the algorithm performs well "on average".

When deterministic algorithms do not suffice

Motivation

- Natural hard inputs may exist, that **always** cause a deterministic algorithm to perform poorly.
- It may be acceptable if we can guarantee that, for every given input, the algorithm performs well "on average".

Average-case analysis?

- The performance of the algorithm, averaged over all possible inputs.
- Need a probability distribution over the inputs.
- Not useful if the natural hard input occurs often.

When deterministic algorithms do not suffice

Motivation

- Natural hard inputs may exist, that **always** cause a deterministic algorithm to perform poorly.
- It may be acceptable if we can guarantee that, for every given input, the algorithm performs well "on average".

Average-case analysis?

- The performance of the algorithm, averaged over all possible inputs.
- Need a probability distribution over the inputs.
- Not useful if the natural hard input occurs often.

Randomized algorithms

- "Flip a coin" and continue computation based on the outcome.
- Algorithm performs different each time on natural hard input.

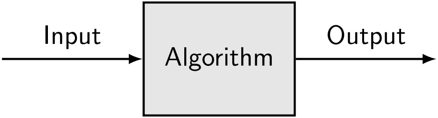
When deterministic algorithms do not suffice

Is well "on average" good enough?

- If we consider a deterministic algorithm to be bad, there are infinitely many inputs for which it **always** produces some output of low quality.
- If a randomized algorithm performs well "on average", it performs bad on some inputs, but only for some random decisions it makes.
- There are no hard instances that **always** cause a good randomized algorithm to fail, but on each instance it fails "sometimes".

Definition

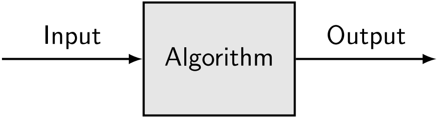
Deterministic Algorithm



- Input $I = (x_1, x_2, \ldots, x_n)$.
- Output $\textsc{Alg}(I) = (y_1, y_2, \ldots, y_n)$.
- y_i depends on $x_1, x_2, \ldots, x_i$ and $y_1, y_2, \ldots y_{i-1}$.

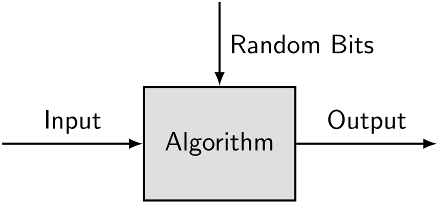
Definition

Deterministic Algorithm



- Input $I = (x_1, x_2, \ldots, x_n)$.
- Output $\textsc{Alg}(I) = (y_1, y_2, \ldots, y_n)$.
- y_i depends on $x_1, x_2, \ldots, x_i$ and $y_1, y_2, \ldots y_{i-1}$.

Randomized Algorithm



- Input $I = (x_1, x_2, \ldots, x_n)$.
- Output $\textsc{Rand}(I) = (y_1, y_2, \ldots, y_n)$.
- ψ binary string, each bit randomly chosen with probability $1/2$.
- y_i depends on $x_1, x_2, \ldots, x_i$, $y_1, y_2, \ldots y_{i-1}$ and ψ.

Random bits

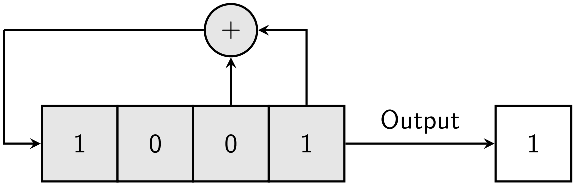
- Real random bits are hard to obtain.
    - Computers are deterministic machines.
- Instead often pseudo-random number generators are used.

Random bits

- Real random bits are hard to obtain.
  - Computers are deterministic machines.
- Instead often pseudo-random number generators are used.

Linear Feedback Shift Registers

- Each iteration shift all bits to the right.
- XOR certain bits to obtain a new leftmost bit.
- Each iteration the rightmost bit is the output random bit.

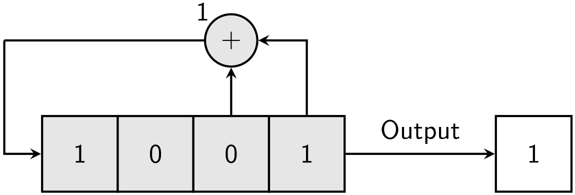


Random bits

- Real random bits are hard to obtain.
    - Computers are deterministic machines.
- Instead often pseudo-random number generators are used.

Linear Feedback Shift Registers

- Each iteration shift all bits to the right.
- XOR certain bits to obtain a new leftmost bit.
- Each iteration the rightmost bit is the output random bit.

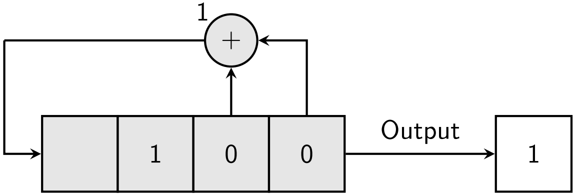


Random bits

- Real random bits are hard to obtain.
    - Computers are deterministic machines.
- Instead often pseudo-random number generators are used.

Linear Feedback Shift Registers

- Each iteration shift all bits to the right.
- XOR certain bits to obtain a new leftmost bit.
- Each iteration the rightmost bit is the output random bit.

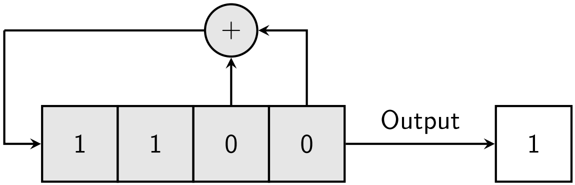


Random bits

- Real random bits are hard to obtain.
    - Computers are deterministic machines.
- Instead often pseudo-random number generators are used.

Linear Feedback Shift Registers

- Each iteration shift all bits to the right.
- XOR certain bits to obtain a new leftmost bit.
- Each iteration the rightmost bit is the output random bit.

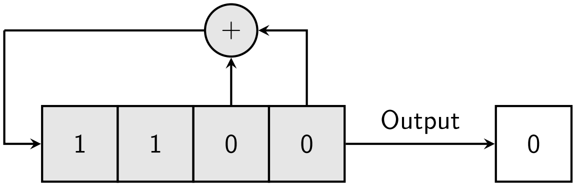


Random bits

- Real random bits are hard to obtain.
    - Computers are deterministic machines.
- Instead often pseudo-random number generators are used.

Linear Feedback Shift Registers

- Each iteration shift all bits to the right.
- XOR certain bits to obtain a new leftmost bit.
- Each iteration the rightmost bit is the output random bit.

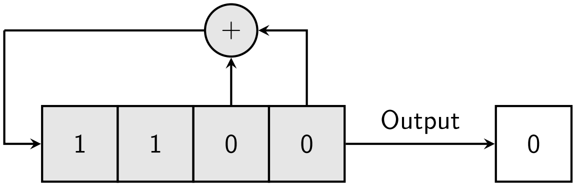


Random bits

- Real random bits are hard to obtain.
  - Computers are deterministic machines.
- Instead often pseudo-random number generators are used.

Linear Feedback Shift Registers

- Each iteration shift all bits to the right.
- XOR certain bits to obtain a new leftmost bit.
- Each iteration the rightmost bit is the output random bit.



Output is statistically very random, however, if you know the state you can predict the next random bit!

Random bits

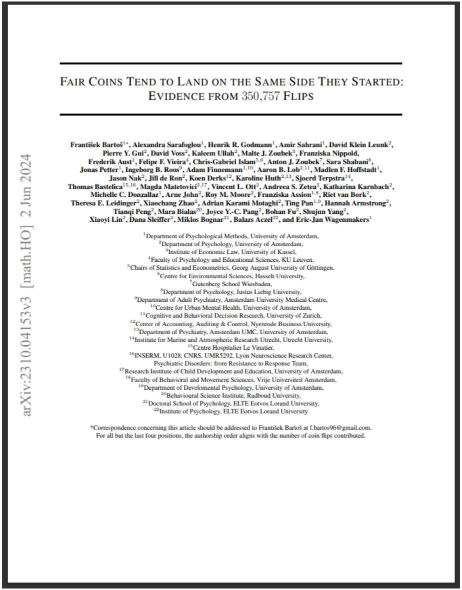
- Neglect difficulty of obtaining "real" randomness.
- Like on a Turing Machine, random bits are read from a *tape* in sequential manner.
- Tape has unbounded length and has an infinite binary string ψ written on it.
- Each bit of ψ is either 1 or 0 with a probability of $1/2$ each.

Random bits

- Neglect difficulty of obtaining "real" randomness.
- Like on a Turing Machine, random bits are read from a *tape* in sequential manner.
- Tape has unbounded length and has an infinite binary string ψ written on it.
- Each bit of ψ is either 1 or 0 with a probability of $1/2$ each.

Random bits

- Neglect difficulty of obtaining "real" randomness.
- Like on a Turing Machine, random bits are read from a *tape* in sequential manner.
- Tape has unbounded length and has an infinite binary string ψ written on it.
- Each bit of ψ is either 1 or 0 with a probability of $1/2$ each.



FAIR COINS TEND TO LAND ON THE SAME SIDE THEY STARTED: EVIDENCE FROM 350,757 FLIPS

František Bartoš[1*], Alexandra Sarafoglou[1], Henrik R. Godmann[1], Amir Sahrani[1], David Klein Leunk[2], Pierre Y. Gui[2], David Voss[2], Kaleem Ullah[2], Malte J. Zoubek[3], Franziska Nippold, Frederik Aust[1], Felipe F. Vieira[4], Chris-Gabriel Islam[5,6], Anton J. Zoubek[7], Sara Shabani[8], Jonas Petter[1], Ingeborg B. Roos[9], Adam Finnemann[1,10], Aaron B. Lob[2,11], Madlen F. Hoffstadt[1], Jason Nak[2], Jill de Ron[2], Koen Derks[12], Karoline Huth[2,13], Sjoerd Terpstra[14], Thomas Bastelica[15,16], Magda Matetovici[2,17], Vincent L. Ott[2], Andreea S. Zetea[2], Katharina Karnbach[2], Michelle C. Donzallaz[1], Arne John[2], Roy M. Moore[2], Franziska Assion[1,8], Riet van Bork[2], Theresa E. Leidinger[2], Xiaochang Zhao[2], Adrian Karami Motaghi[2], Ting Pan[1,9], Hannah Armstrong[2], Tianqi Peng[2], Mara Bialas[20], Joyce Y.-C. Pang[2], Bohan Fu[2], Shujun Yang[2], Xiaoyi Lin[2], Dana Sleiffer[2], Miklos Bognar[21], Balazs Aczel[22], and Eric-Jan Wagenmakers[1]

[1]Department of Psychological Methods, University of Amsterdam,
[2]Department of Psychology, University of Amsterdam,
[3]Institute of Economic Law, University of Kassel,
[4]Faculty of Psychology and Educational Sciences, KU Leuven,
[5]Chairs of Statistics and Econometrics, Georg August University of Göttingen,
[6]Centre for Environmental Sciences, Hasselt University,
[7]Gutenberg School Wiesbaden,
[8]Department of Psychology, Justus Liebig University,
[9]Department of Adult Psychiatry, Amsterdam University Medical Centre,
[10]Centre for Urban Mental Health, University of Amsterdam,
[11]Cognitive and Behavioral Decision Research, University of Zurich,
[12]Center of Accounting, Auditing & Control, Nyenrode Business University,
[13]Department of Psychiatry, Amsterdam UMC, University of Amsterdam,
[14]Institute for Marine and Atmospheric Research Utrecht, Utrecht University,
[15]Centre Hospitalier Le Vinatier,
[16]INSERM, U1028; CNRS, UMR5292, Lyon Neuroscience Research Center, Psychiatric Disorders: from Resistance to Response Team,
[17]Research Institute of Child Development and Education, University of Amsterdam,
[18]Faculty of Behavioral and Movement Sciences, Vrije Universiteit Amsterdam,
[19]Department of Develomental Psychology, University of Amsterdam,
[20]Behavioural Science Institute, Radboud University,
[21]Doctoral School of Psychology, ELTE Eotvos Lorand University,
[22]Institute of Psychology, ELTE Eotvos Lorand University

*Correspondence concerning this article should be addressed to František Bartoš at f.bartos96@gmail.com.

For all but the last four positions, the authorship order aligns with the number of coin flips contributed.

Adversaries

Deterministic Algorithm

Adversary can see the algorithm.

(Full knowledge, very strong)

Adversaries

Randomized Algorithm

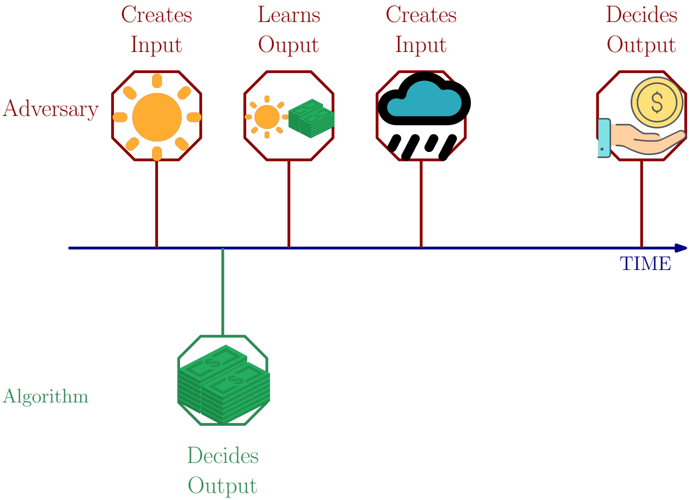
Oblivious Adversary
Can see the algorithm.
Is oblivious to the the random choices made by the algorithm.

Deterministic Algorithm
Adversary can see the algorithm.
(Full knowledge, very strong)

Adaptive Online Adversary
Can see the algorithm.
Learns the decisions of $\textsc{Rand}$ adaptively.
Before generating x_i it must compute y_i.

Adaptive Offline Adversary
Can see the algorithm.
Learns the decisions of $\textsc{Rand}$ adaptively.
Computes $y_1, \ldots y_n$ after generating full input.

Adaptive Offline Adversary



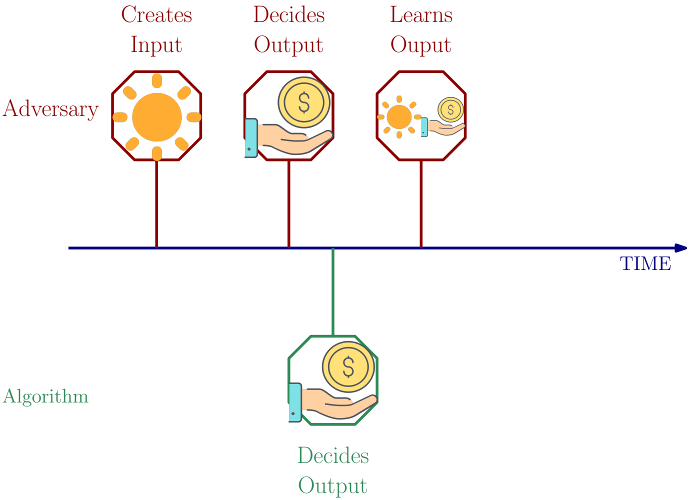
Expected competitive ratio

Expected competitive ratio against an offline adaptive adversary.

- $\textsc{Rand}$ is c-competitive in expectation against an offline adaptive adversary, if for all instances I

$$\frac{\mathbb{E}_{\textsc{Rand}}\left[\textsc{Rand}(I)\right]}{\mathbb{E}_{\textsc{Rand}}\left[\textsc{Opt}(I)\right]} \leq c$$

Adaptive Online Adversary



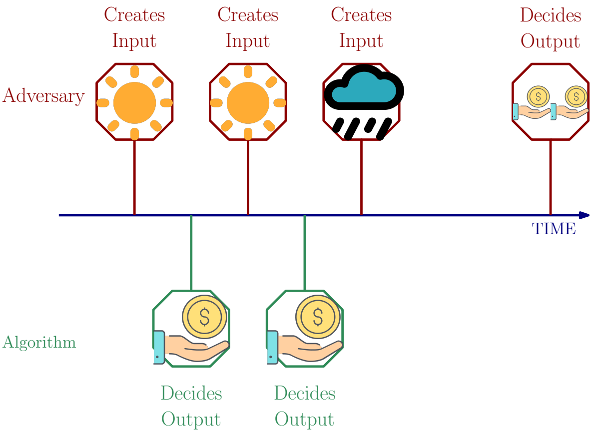
Expected competitive ratio

Expected competitive ratio against an online adaptive adversary.

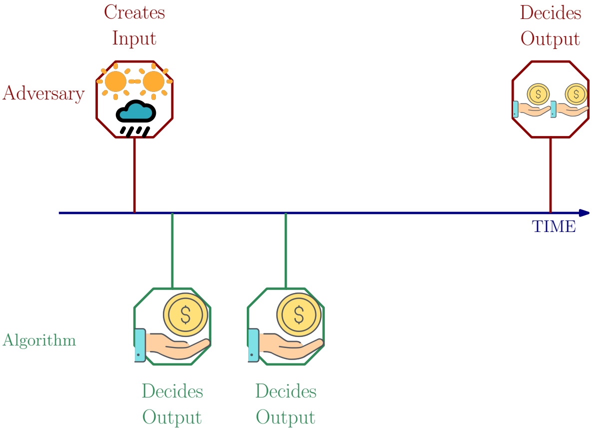
- $\textsc{Rand}$ is c-competitive in expectation against an online adaptive adversary, if for all instances I

$$\frac{\mathbb{E}_{\textsc{Rand}}\left[\textsc{Rand}(I)\right]}{\mathbb{E}_{\textsc{Rand}}\left[\textsc{Opt}(I)\right]} \leq c$$

Oblivious Adversary



Oblivious Adversary



Since the adversary cannot react to the output of the algorithm, this is equivalent to the adversary preparing the input in advance.

Expected competitive ratio

Expected competitive ratio against an oblivious adversary.

- $\textsc{Rand}$ is c-competitive in expectation against an oblivious adversary, if for all instances I

$$\frac{\mathbb{E}_{\textsc{Rand}}\left[\textsc{Rand}(I)\right]}{\textsc{Opt}(I)} \leq c$$

Expected competitive ratio

Expected competitive ratio against an oblivious adversary.

- $\textsc{Rand}$ is c-competitive in expectation against an oblivious adversary, if for all instances I

$$\frac{\mathbb{E}_{\textsc{Rand}}\left[\textsc{Rand}(I)\right]}{\textsc{Opt}(I)} \leq c$$

This is the adversary that we will use in the lectures.

Relationship between adversaries

- The oblivious adversary is weaker than the online adaptive adversary.

Relationship between adversaries

- The oblivious adversary is weaker than the online adaptive adversary.
  - It is easier to get a low expected competitive ratio against an oblivious adversary than against an online adaptive adversary.
  - $R_{\text{OBL}} \leq R_{\text{ADON}}$.

Relationship between adversaries

- The oblivious adversary is weaker than the online adaptive adversary.
  - It is easier to get a low expected competitive ratio against an oblivious adversary than against an online adaptive adversary.
  - $R_{\text{OBL}} \leq R_{\text{ADON}}$.
- The online adaptive adversary is weaker than the offline adaptive adversary.

Relationship between adversaries

- The oblivious adversary is weaker than the online adaptive adversary.
  - It is easier to get a low expected competitive ratio against an oblivious adversary than against an online adaptive adversary.
  - $R_{\textsc{Obl}} \leq R_{\textsc{AdOn}}$.
- The online adaptive adversary is weaker than the offline adaptive adversary.
  - It is easier to get a low expected competitive ratio against an online adaptive adversary than against an offline adaptive adversary.
  - $R_{\textsc{AdOn}} \leq R_{\textsc{AdOff}}$.

Relationship between adversaries

- The oblivious adversary is weaker than the online adaptive adversary.
  - It is easier to get a low expected competitive ratio against an oblivious adversary than against an online adaptive adversary.
  - $R_{\text{OBL}} \leq R_{\text{ADON}}$.
- The online adaptive adversary is weaker than the offline adaptive adversary.
  - It is easier to get a low expected competitive ratio against an online adaptive adversary than against an offline adaptive adversary.
  - $R_{\text{ADON}} \leq R_{\text{ADOFF}}$.
- The offline adaptive adversary is weaker than the deterministic adversary.

Relationship between adversaries

- The oblivious adversary is weaker than the online adaptive adversary.
  - It is easier to get a low expected competitive ratio against an oblivious adversary than against an online adaptive adversary.
  - $R_{\text{OBL}} \leq R_{\text{ADON}}$.
- The online adaptive adversary is weaker than the offline adaptive adversary.
  - It is easier to get a low expected competitive ratio against an online adaptive adversary than against an offline adaptive adversary.
  - $R_{\text{ADON}} \leq R_{\text{ADOFF}}$.
- The offline adaptive adversary is weaker than the deterministic adversary.
  - It is easier to get a low expected competitive ratio against an offline adaptive adversary than a low competitive ratio against a deterministic adversary.
  - $R_{\text{ADOFF}} \leq R_{\text{DET}}$.

Relationship between adversaries

- The oblivious adversary is weaker than the online adaptive adversary.
  - It is easier to get a low expected competitive ratio against an oblivious adversary than against an online adaptive adversary.
  - $R_{\textsc{Obl}} \leq R_{\textsc{AdOn}}$.
- The online adaptive adversary is weaker than the offline adaptive adversary.
  - It is easier to get a low expected competitive ratio against an online adaptive adversary than against an offline adaptive adversary.
  - $R_{\textsc{AdOn}} \leq R_{\textsc{AdOff}}$.
- The offline adaptive adversary is weaker than the deterministic adversary.
  - It is easier to get a low expected competitive ratio against an offline adaptive adversary than a low competitive ratio against a deterministic adversary.
  - $R_{\textsc{AdOff}} \leq R_{\textsc{Det}}$.
- Although the oblivious adversary is the weakest, often it precisely models the problem.
  - E.g. whether or not you buy ski's has not effect on the weather.

A randomized algorithm is a set of deterministic algorithms

Observation

Every randomized algorithm Rand, that uses at most $b(n)$ random bits for inputs of length n, can be viewed as a set strat($\textsc{Rand}$, n) $= \{\textsc{Alg}_1, \textsc{Alg}_2, \ldots \textsc{Alg}_{2^{b(n)}}\}$ (not necessarily distinct) deterministic online algorithms, from which $\textsc{Rand}$ chooses one uniformly at random with probability $1/2^{b(n)}$.

A randomized algorithm is a set of deterministic algorithms

Observation

Every randomized algorithm Rand, that uses at most $b(n)$ random bits for inputs of length n, can be viewed as a set strat$(\textsc{Rand}, n) = \{\textsc{Alg}_1, \textsc{Alg}_2, \ldots \textsc{Alg}_{2^{b(n)}}\}$ (not necessarily distinct) deterministic online algorithms, from which $\textsc{Rand}$ chooses one uniformly at random with probability $1/2^{b(n)}$.

Limitations

- For (most) online algorithms n is not known in advance, so we cannot compute $b(n)$ before the execution.
- However, we can compute $b(n)$ when analysing the algorithm.

A randomized algorithm is a set of deterministic algorithms

Suppose algorithm $\textsc{RandSki}$ is a randomized algorithm for the ski rental problem.

Procedure $\textsc{RandSki}$

We choose a day i, uniformly at random from $\{1, 2, \ldots, B\}$, such that we rent skis until day $i - 1$ and, if there are at least i sunny days, we buy skis on day i.

A randomized algorithm is a set of deterministic algorithms

Suppose algorithm $\textsc{RandSki}$ is a randomized algorithm for the ski rental problem.

Procedure $\textsc{RandSki}$

We choose a day i, uniformly at random from $\{1, 2, \ldots, B\}$, such that we rent skis until day $i - 1$ and, if there are at least i sunny days, we buy skis on day i.

Suppose $\textsc{Alg}_i$ is a deterministic algorithm for the ski rental problem.

Procedure Alg$_i$

For the first $i - 1$ sunny days rent skis, and buy skis on day i.

A randomized algorithm is a set of deterministic algorithms

Suppose algorithm $\textsc{RandSki}$ is a randomized algorithm for the ski rental problem.

Procedure $\textsc{RandSki}$

We choose a day i, uniformly at random from $\{1, 2, \ldots, B\}$, such that we rent skis until day $i - 1$ and, if there are at least i sunny days, we buy skis on day i.

Suppose $\textsc{Alg}_i$ is a deterministic algorithm for the ski rental problem.

Procedure Alg$_i$

For the first $i - 1$ sunny days rent skis, and buy skis on day i.

Clearly strat($\textsc{RandSki}$) $= \{\textsc{Alg}_1, \textsc{Alg}_2, \ldots, \textsc{Alg}_B\}$, where we pick a deterministic algorithm $\textsc{Alg}_i$ with probability $\frac{1}{B}$.

A randomized algorithm is a set of deterministic algorithms

Suppose algorithm RandSki2 is another randomized algorithm for the ski rental problem.

Procedure RandSki2

Every sunny day, until we have bought skis, we decide with probability $\frac{1}{B}$ if we buy skis this day, or keep renting.

A randomized algorithm is a set of deterministic algorithms

Suppose algorithm $\textsc{RandSki2}$ is another randomized algorithm for the ski rental problem.

Procedure $\textsc{RandSki2}$

Every sunny day, until we have bought skis, we decide with probability $\frac{1}{B}$ if we buy skis this day, or keep renting.

Suppose $\textsc{Alg}_i$ is a deterministic algorithm for the ski rental problem.

Procedure $\textsc{Alg}_i$

For the first $i - 1$ sunny days rent skis, and buy skis on day i.

A randomized algorithm is a set of deterministic algorithms

Suppose algorithm $\textsc{RandSki2}$ is another randomized algorithm for the ski rental problem.

Procedure $\textsc{RandSki2}$

Every sunny day, until we have bought skis, we decide with probability $\frac{1}{B}$ if we buy skis this day, or keep renting.

Suppose $\textsc{Alg}_i$ is a deterministic algorithm for the ski rental problem.

Procedure $\textsc{Alg}_i$

For the first $i-1$ sunny days rent skis, and buy skis on day i.

Let $\mathcal{E}_i$ be the event where we buy on day i. Then,
$\mathbf{Pr}\left(\mathcal{E}_i\right) = \frac{1}{B}(1-\frac{1}{B})^{i-1} = \frac{1}{B}(\frac{B-1}{B})^{i-1}$

A randomized algorithm is a set of deterministic algorithms

Suppose algorithm $\textsc{RandSki2}$ is another randomized algorithm for the ski rental problem.

Procedure $\textsc{RandSki2}$

Every sunny day, until we have bought skis, we decide with probability $\frac{1}{B}$ if we buy skis this day, or keep renting.

Suppose $\textsc{Alg}_i$ is a deterministic algorithm for the ski rental problem.

Procedure $\textsc{Alg}_i$

For the first $i-1$ sunny days rent skis, and buy skis on day i.

Then, $\text{strat}(\textsc{RandSki}) = \{\underbrace{\textsc{Alg}_1, \ldots, \textsc{Alg}_1}_{(\frac{B}{B-1})^{N-1}}, \underbrace{\textsc{Alg}_2, \ldots, \textsc{Alg}_2}_{(\frac{B}{B-1})^{N-2}}, \ldots, \textsc{Alg}_N\}$, where N is the number of sunny days, and we pick a deterministic algorithm $\textsc{Alg}_i$ with uniformly at random with probability $\frac{1}{\sum_{i=1}^{N}(\frac{B}{B-1})^{i-1}}$.

No 1-competitive randomized algorithm

Theorem

If for some online problem there exists a 1-competitive randomized online algorithm, then there also exists a 1-competitive deterministic online algorithm.

No 1-competitive randomized algorithm

Theorem

If for some online problem there exists a 1-competitive randomized online algorithm, then there also exists a 1-competitive deterministic online algorithm.

Proof.

Suppose that $\textsc{Rand}$ is a 1-competitive randomized algorithm for the problem.

No 1-competitive randomized algorithm

Theorem

If for some online problem there exists a 1-competitive randomized online algorithm, then there also exists a 1-competitive deterministic online algorithm.

Proof.

Suppose that $\textsc{Rand}$ is a 1-competitive randomized algorithm for the problem. Now suppose that we fix a bit string ψ'. By the previous observation this corresponds to a deterministic algorithm $\textsc{Alg}_i$.

No 1-competitive randomized algorithm

Theorem

If for some online problem there exists a 1-competitive randomized online algorithm, then there also exists a 1-competitive deterministic online algorithm.

Proof.

Suppose that $\textsc{Rand}$ is a 1-competitive randomized algorithm for the problem. Now suppose that we fix a bit string ψ'. By the previous observation this corresponds to a deterministic algorithm $\textsc{Alg}_i$. Since the randomized algorithm is 1-competitive, for every input and every bit string, it should return a 1-competitive result. Then, also, the result for every input and fixed bit string ψ' is 1-competitive.

No 1-competitive randomized algorithm

Theorem

If for some online problem there exists a 1-competitive randomized online algorithm, then there also exists a 1-competitive deterministic online algorithm.

Proof.

Suppose that $\textsc{Rand}$ is a 1-competitive randomized algorithm for the problem. Now suppose that we fix a bit string ψ'. By the previous observation this corresponds to a deterministic algorithm $\textsc{Alg}_i$. Since the randomized algorithm is 1-competitive, for every input and every bit string, it should return a 1-competitive result. Then, also, the result for every input and fixed bit string ψ' is 1-competitive. Thus algorithm $\textsc{Alg}_i$ is 1-competitive. □

No 1-competitive randomized algorithm

Theorem

If for some online problem there exists a 1-competitive randomized online algorithm, then there also exists a 1-competitive deterministic online algorithm.

Proof.
Suppose that $\textsc{Rand}$ is a 1-competitive randomized algorithm for the problem. Now suppose that we fix a bit string ψ'. By the previous observation this corresponds to a deterministic algorithm $\textsc{Alg}_i$. Since the randomized algorithm is 1-competitive, for every input and every bit string, it should return a 1-competitive result. Then, also, the result for every input and fixed bit string ψ' is 1-competitive. Thus algorithm $\textsc{Alg}_i$ is 1-competitive. □

As a consequence, if we shown that there exists no 1-competitive deterministic online algorithm for some online problem, we can conclude that there also exists no 1-competitive randomized one.

Overview

- Randomized algorithms are dependent on both input and random string.
- A randomized algorithm is a probability distribution over all deterministic algorithms.
- 3 types of adversaries, Oblivious, Adaptive Online and Adaptive Offline.
- A 1-competitive randomized algorithm implies the existence of a 1-competitive deterministic algorithm.

Table of Contents

Paging problem

Paging.

- Given a computer system with two-level memory, **main memory** and **cache**.

- When the processor needs a page p_i

  - If p_i is in the cache, the system does not do anything. (cost 0)

  - If p_i is not in the cache, the system must copy the page p_i from main memory to the cache. (cost 1)

Cache $\boxed{1}\ \boxed{2}\ \boxed{3}$ Main Memory $\boxed{1}\ \boxed{2}\ \boxed{3}\ \boxed{4}\ \boxed{5}\ \boxed{6}$

Paging problem

Paging.

- Given a computer system with two-level memory, **main memory** and **cache**.

- When the processor needs a page p_i

  - If p_i is in the cache, the system does not do anything. (cost 0)

  - If p_i is not in the cache, the system must copy the page p_i from main memory to the cache. (cost 1)

Cache $\boxed{1}\ \boxed{2}\ \boxed{3}$ Main Memory $\boxed{1}\ \boxed{2}\ \boxed{3}\ \boxed{4}\ \boxed{5}\ \boxed{6}$

Paging problem

Paging.

- Given a computer system with two-level memory, **main memory** and **cache**.
- When the processor needs a page p_i
  - If p_i is in the cache, the system does not do anything. (cost 0)
  - If p_i is not in the cache, the system must copy the page p_i from main memory to the cache. (cost 1)

Cache $\boxed{1}\ \boxed{2}\ \boxed{3}$ Main Memory $\boxed{1}\ \boxed{2}\ \boxed{3}\ \boxed{4}\ \boxed{5}\ \boxed{6}$

Paging problem

Paging.

- Given a computer system with two-level memory, **main memory** and **cache**.

- When the processor needs a page p_i

  - If p_i is in the cache, the system does not do anything. (cost 0)

  - If p_i is not in the cache, the system must copy the page p_i from main memory to the cache. (cost 1)

Cache $\boxed{1}\ \boxed{2}\ \boxed{3}$ Main Memory $\boxed{1}\ \boxed{2}\ \boxed{3}\ \boxed{4}\ \boxed{5}\ \boxed{6}$

Paging problem

Paging.

- Given a computer system with two-level memory, **main memory** and **cache**.
- When the processor needs a page p_i
  - If p_i is in the cache, the system does not do anything. (cost 0)
  - If p_i is not in the cache, the system must copy the page p_i from main memory to the cache. (cost 1)

Cache $\boxed{1}\ \boxed{2}\ \boxed{3}$ Main Memory $\boxed{1}\ \boxed{2}\ \boxed{3}\ \boxed{4}\ \boxed{5}\ \boxed{6}$

Paging problem

Paging.

- Given a computer system with two-level memory, **main memory** and **cache**.

- When the processor needs a page p_i

  - If p_i is in the cache, the system does not do anything. (cost 0)

  - If p_i is not in the cache, the system must copy the page p_i from main memory to the cache. (cost 1)

Cache $\boxed{1}\ \boxed{2}\ \boxed{3}$ Main Memory $\boxed{1}\ \boxed{2}\ \boxed{3}\ \boxed{4}\ \boxed{5}\ \boxed{6}$

Paging problem

Paging.

- Given a computer system with two-level memory, **main memory** and **cache**.

- When the processor needs a page p_i
  - If p_i is in the cache, the system does not do anything. (cost 0)
  - If p_i is not in the cache, the system must copy the page p_i from main memory to the cache. (cost 1)

Paging problem

Paging.

- Given a computer system with two-level memory, **main memory** and **cache**.
- When the processor needs a page p_i
  - If p_i is in the cache, the system does not do anything. (cost 0)
  - If p_i is not in the cache, the system must copy the page p_i from main memory to the cache. (cost 1)

Cache | 1 | 2 | 4 |

Main Memory | 1 | 2 | 3 | 4 | 5 | 6 |

Paging, so far...

We have seen:

- Two deterministic algorithms for paging.
  - LFU (Least-Frequently-Used).
  - LRU (Least-Recently-Used).
- Optimal offline algorithm for paging.
  - LFD (Longest-Forward-Distance).
- Problem competitive ratio lower bound.

Paging, so far...

We have seen:

- Two deterministic algorithms for paging.
  - LFU (Least-Frequently-Used).
    - Unbounded competitive ratio.
  - LRU (Least-Recently-Used).
- Optimal offline algorithm for paging.
  - LFD (Longest-Forward-Distance).
- Problem competitive ratio lower bound.

Paging, so far...

We have seen:

- Two deterministic algorithms for paging.
  - LFU (Least-Frequently-Used).
    - Unbounded competitive ratio.
  - LRU (Least-Recently-Used).
    - k-competitive.
- Optimal offline algorithm for paging.
  - LFD (Longest-Forward-Distance).
- Problem competitive ratio lower bound.

Paging, so far...

We have seen:

- Two deterministic algorithms for paging.
  - LFU (Least-Frequently-Used).
    - Unbounded competitive ratio.
  - LRU (Least-Recently-Used).
    - k-competitive.
- Optimal offline algorithm for paging.
  - LFD (Longest-Forward-Distance).
- Problem competitive ratio lower bound.
  - k-competitive.

Paging, so far...

We have seen:

- Two deterministic algorithms for paging.
  - LFU (Least-Frequently-Used).
    - Unbounded competitive ratio.
  - LRU (Least-Recently-Used).
    - k-competitive.
- Optimal offline algorithm for paging.
  - LFD (Longest-Forward-Distance).
- Problem competitive ratio lower bound.
  - k-competitive.

Only for deterministic algorithms!

A first attempt at randomization

Procedure RANDOM

Whenever a page fault occurs, remove a page from the cache chosen uniformly at random.

A first attempt at randomization

Each page is chosen
with equal probability

Whenever page fault occurs, remove a page from the cache chosen uniformly at random.

Analysis of RANDOM

Theorem

Algorithm $\textsc{Random}$ is at least k-competitive in expectation, even when $M = k + 1$.

Analysis of RANDOM

Theorem

Algorithm RANDOM is at least k-competitive in expectation, even when $M = k + 1$.

To show that algorithm RANDOM is at least k-competitive in expectation we will construct an input where RANDOM removes in expectation k pages from the cache where it suffices for OPT to only remove a single page.

Analysis of RANDOM

Theorem

Algorithm RANDOM is at least k-competitive in expectation, even when $M = k + 1$.

To show that algorithm RANDOM is at least k-competitive in expectation we will construct an input where RANDOM removes in expectation k pages from the cache where it suffices for OPT to only remove a single page.

Proof.

Assume without loss of generality that RANDOM en OPT start with a cache containing pages $1, \ldots, k$.

Analysis of RANDOM

Theorem

Algorithm $\textsc{Random}$ is at least k-competitive in expectation, even when $M = k + 1$.

To show that algorithm $\textsc{Random}$ is at least k-competitive in expectation we will construct an input where $\textsc{Random}$ removes in expectation k pages from the cache where it suffices for $\textsc{Opt}$ to only remove a single page.

Proof.

Assume without loss of generality that $\textsc{Random}$ en $\textsc{Opt}$ start with a cache containing pages $1, \ldots, k$.

Now consider the input sequence $(2, \ldots k, k+1, 2, \ldots, k, k+1, 2, \ldots)$.

Analysis of RANDOM

Theorem

Algorithm $\textsc{Random}$ is at least k-competitive in expectation, even when $M = k + 1$.

To show that algorithm $\textsc{Random}$ is at least k-competitive in expectation we will construct an input where $\textsc{Random}$ removes in expectation k pages from the cache where it suffices for $\textsc{Opt}$ to only remove a single page.

Proof.

Assume without loss of generality that $\textsc{Random}$ en $\textsc{Opt}$ start with a cache containing pages $1, \ldots, k$.

Now consider the input sequence $(2, \ldots k, k+1, 2, \ldots, k, k+1, 2, \ldots)$.

Clearly $\textsc{Opt}$ removes page 1 at the first request and never has to remove a page again.

Analysis of RANDOM

Theorem

Algorithm Random is at least k-competitive in expectation, even when $M = k + 1$.

To show that algorithm Random is at least k-competitive in expectation we will construct an input where Random removes in expectation k pages from the cache where it suffices for Opt to only remove a single page.

Proof.

Assume without loss of generality that Random en Opt start with a cache containing pages $1, \dots, k$.

Now consider the input sequence $(2, \dots k, k+1, 2, \dots, k, k+1, 2, \dots)$.

Clearly Opt removes page 1 at the first request and never has to remove a page again.

We now need to determine when Random in expectation removes page 1.

Let X be the random variable that counts the number of page faults until Random removes page 1.

Random has probability $\frac{1}{k}$ to remove the right page at each page fault.

Analysis of RANDOM

Theorem

Algorithm $\textsc{Random}$ is at least k-competitive in expectation, even when $M = k + 1$.

To show that algorithm $\textsc{Random}$ is at least k-competitive in expectation we will construct an input where $\textsc{Random}$ removes in expectation k pages from the cache where it suffices for $\textsc{Opt}$ to only remove a single page.

Proof.

Assume without loss of generality that $\textsc{Random}$ en $\textsc{Opt}$ start with a cache containing pages $1, \dots, k$.

Now consider the input sequence $(2, \dots k, k + 1, 2, \dots, k, k + 1, 2, \dots)$.

Clearly $\textsc{Opt}$ removes page 1 at the first request and never has to remove a page again.

We now need to determine when $\textsc{Random}$ in expectation removes page 1.

Let X be the random variable that counts the number of page faults until $\textsc{Random}$ removes page 1.

$\textsc{Random}$ has probability $\frac{1}{k}$ to remove the right page at each page fault.

Since X follows a geometric distribution, it follows that $\mathbb{E}[X] = k$.

Analysis of RANDOM

Theorem

Algorithm RANDOM is at least k-competitive in expectation, even when $M = k + 1$.

To show that algorithm RANDOM is at least k-competitive in expectation we will construct an input where RANDOM removes in expectation k pages from the cache where it suffices for OPT to only remove a single page.

Proof.

Assume without loss of generality that RANDOM en OPT start with a cache containing pages $1, \dots, k$.

Now consider the input sequence $(2, \dots k, k+1, 2, \dots, k, k+1, 2, \dots)$.

Clearly OPT removes page 1 at the first request and never has to remove a page again.

The probability distribution of the number of tails one must flip before the first head using a weighted coin.

We now need ... ves page 1.

Let X be the ra... faults until RANDOM removes page 1.

RANDOM has probability $\frac{1}{k}$ to ...emove the right page at each page fault.

Since X follows a geometric distribution, it follows that $\mathbb{E}[X] = k$.

Analysis of RANDOM

Theorem

Algorithm $\textsc{Random}$ is at least k-competitive in expectation, even when $M = k + 1$.

To show that algorithm $\textsc{Random}$ is at least k-competitive in expectation we will construct an input where $\textsc{Random}$ removes in expectation k pages from the cache where it suffices for $\textsc{Opt}$ to only remove a single page.

Proof.

Assume without loss of generality that $\textsc{Random}$ en $\textsc{Opt}$ start with a cache containing pages $1, \ldots, k$.

Now consider the input sequence $(2, \ldots k, k+1, 2, \ldots, k, k+1, 2, \ldots)$.

Clearly $\textsc{Opt}$ removes page 1 at the first request and never has to remove a page again.

We now need to determine when $\textsc{Random}$ in expectation removes page 1.

Let X be the random variable that counts the number of page faults until $\textsc{Random}$ removes page 1.

$\textsc{Random}$ has probability $\frac{1}{k}$ to remove the right page at each page fault.

Since X follows a geometric distribution, it follows that $\mathbb{E}[X] = k$.

This gives a competitive ratio of $\frac{\textsc{Random}}{\textsc{Opt}} = \frac{k}{1} = k$. $\square$

A second attempt at randomization

- Just making a random decision does not always work.
- Adapt competitive deterministic strategies.
- Let's take a closer look into the LRU algorithm.

Phase partitioning

Phase Partioning: Partition the request sequence into phases.

- Phase 0 contains all requests untill the first page fault (that is, requests that are already in the cache).
- Phase i is the maximal sequence following phase $i-1$ that contains exactly k distinct page requests (that is, phase $i+1$ begins on the request that is the $(k+1)$-th distinct page).
- Phase N contains at most k distinct page requests.

Phase partitioning

Phase Partioning: Partition the request sequence into phases.

- Phase 0 is empty.

- Phase i is the maximal sequence following phase $i-1$ that contains exactly k distinct page requests (that is, phase $i+1$ begins on the request that is the $(k+1)$-th distinct page).

- Phase N contains at most k distinct page requests.

Phase partitioning

Phase Partioning: Partition the request sequence into phases.

- Phase 0 is empty.
- Phase i is the maximal sequence following phase $i-1$ that contains exactly k distinct page requests (that is, phase $i+1$ begins on the request that is the $(k+1)$-th distinct page).
- Phase N contains exactly k distinct page request.

Phase partitioning

Phase Partioning: Partition the request sequence into phases.

- Phase 0 is empty.
- Phase i is the maximal sequence following phase $i-1$ that contains exactly k distinct page requests (that is, phase $i+1$ begins on the request that is the $(k+1)$-th distinct page).
- Phase N contains exactly k distinct page request.

For $k = 3$:

Request: 1, 3, 3, 5, 4, 3, 2, 5, 2, 1, 1, 3, 2, 3, 1, 3, 3, 5, 3, 5, 2

Phase partitioning

Phase Partioning: Partition the request sequence into phases.

- Phase 0 is empty.
- Phase i is the maximal sequence following phase $i-1$ that contains exactly k distinct page requests (that is, phase $i+1$ begins on the request that is the $(k+1)$-th distinct page).
- Phase N contains exactly k distinct page request.

For $k = 3$:

Request: $\underbrace{\boxed{1, 3, 3, 5,}}_{\text{Phase 1}}$ 4, 3, 2, 5, 2, 1, 1, 3, 2, 3, 1, 3, 3, 5, 3, 5, 2

Phase partitioning

Phase Partioning: Partition the request sequence into phases.

- Phase 0 is empty.
- Phase i is the maximal sequence following phase $i-1$ that contains exactly k distinct page requests (that is, phase $i+1$ begins on the request that is the $(k+1)$-th distinct page).
- Phase N contains exactly k distinct page request.

For $k = 3$:

Request: $\underbrace{\boxed{1,\ 3,\ 3,\ 5,}}_{\text{Phase 1}}\underbrace{\boxed{4,\ 3,\ 2,}}_{\text{Phase 2}}$ 5, 2, 1, 1, 3, 2, 3, 1, 3, 3, 5, 3, 5, 2

Phase partitioning

Phase Partioning: Partition the request sequence into phases.

- Phase 0 is empty.
- Phase i is the maximal sequence following phase $i-1$ that contains exactly k distinct page requests (that is, phase $i+1$ begins on the request that is the $(k+1)$-th distinct page).
- Phase N contains exactly k distinct page request.

For $k = 3$:

Request: $\underbrace{\boxed{1,\ 3,\ 3,\ 5,}}_{\text{Phase 1}}\underbrace{\boxed{4,\ 3,\ 2,}}_{\text{Phase 2}}\underbrace{\boxed{5,\ 2,\ 1,\ 1,}}_{\text{Phase 3}}3,\ 2,\ 3,\ 1,\ 3,\ 3,\ 5,\ 3,\ 5,\ 2$

Phase partitioning

Phase Partioning: Partition the request sequence into phases.

- Phase 0 is empty.
- Phase i is the maximal sequence following phase $i-1$ that contains exactly k distinct page requests (that is, phase $i+1$ begins on the request that is the $(k+1)$-th distinct page).
- Phase N contains exactly k distinct page request.

For $k = 3$:

Request: $\underbrace{\boxed{1,\ 3,\ 3,\ 5,}}_{\text{Phase 1}}\ \underbrace{\boxed{4,\ 3,\ 2,}}_{\text{Phase 2}}\ \underbrace{\boxed{5,\ 2,\ 1,\ 1,}}_{\text{Phase 3}}\ \underbrace{\boxed{3,\ 2,\ 3,\ 1,\ 3,\ 3,}}_{\text{Phase 4}}\ 5,\ 3,\ 5,\ 2$

Phase partitioning

Phase Partioning: Partition the request sequence into phases.

- Phase 0 is empty.
- Phase i is the maximal sequence following phase $i-1$ that contains exactly k distinct page requests (that is, phase $i+1$ begins on the request that is the $(k+1)$-th distinct page).
- Phase N contains exactly k distinct page request.

For $k = 3$:

Request:	1, 3, 3, 5,	4, 3, 2,	5, 2, 1, 1,	3, 2, 3, 1, 3, 3,	5, 3, 5, 2
	Phase 1	Phase 2	Phase 3	Phase 4	Phase 5

Marking algorithm

- Works in phases.
- In each phase:
  - Mark a page when requested.
  - Only remove **unmarked** pages.
- When all pages are marked:
  - Unmark all pages.
  - Start a new phase.

Marking algorithm

- Works in phases.
- In each phase:
  - Mark a page when requested.
  - Only remove **unmarked** pages.

How do we choose which one?

- When all pages are marked:
  - Unmark all pages.
  - Start a new phase.

LRU is a marking algorithm

Theorem

LRU is a marking algorithm.

LRU is a marking algorithm

Theorem

LRU is a marking algorithm.

Proof.

For a contradiction, suppose that LRU is not a marking algorithm.

LRU is a marking algorithm

Theorem

LRU is a marking algorithm.

Proof.

For a contradiction, suppose that LRU is not a marking algorithm. Then there exists some instance I such that LRU removes a marked page.

LRU is a marking algorithm

Theorem

LRU is a marking algorithm.

Proof.
For a contradiction, suppose that LRU is not a marking algorithm. Then there exists some instance I such that LRU removes a marked page. Let p be the page for which this happens for the first time, at time step t_j in some phase P_i. Since p is marked, it must have been requested before in phase P_i, say at time $t_{j'}$, with $j' < j$.

LRU is a marking algorithm

Theorem

LRU is a marking algorithm.

Proof.

For a contradiction, suppose that LRU is not a marking algorithm. Then there exists some instance I such that LRU removes a marked page. Let p be the page for which this happens for the first time, at time step t_j in some phase P_i. Since p is marked, it must have been requested before in phase P_i, say at time $t_{j'}$, with $j' < j$. Because p was most recently used at time t_j, there must have been k distinct request, all different from p, after time $t_{j'}$.

LRU is a marking algorithm

Theorem

LRU is a marking algorithm.

Proof.
For a contradiction, suppose that LRU is not a marking algorithm. Then there exists some instance I such that LRU removes a marked page. Let p be the page for which this happens for the first time, at time step t_j in some phase P_i. Since p is marked, it must have been requested before in phase P_i, say at time $t_{j'}$, with $j' < j$. Because p was most recently used at time t_j, there must have been k distinct request, all different from p, after time $t_{j'}$. Then P_i consists of at least $k+1$ distinct requests, which is a contradiction. $\square$

Marking algorithm

Procedure Marking

```
mark all pages in the cache                          //first page fault starts a new phase
for every requests x do
   if x is in the cache
      if x is unmarked
         mark x
   else
      if there is no unmarked page
         unmark all pages in the cache               //start new phase
      p ← somehow chosen page among all unmarked cached pages
      remove p
      insert x at the old position of p
      mark x
```

Marking algorithm

Procedure Marking

```
mark all pages in the cache                //first page fault starts a new phase
for every requests x do
    if x is in the cache
        if x is unmarked
            mark x
    else
        if there is no unmarked page
            unmark all pages in the cache          //start new phase
        p ← somehow chosen page among all unmarked cached pages
        remove p
        insert x at the old position of p
        mark x
```

Request:

Cache | 1 | 2 | 3 |

Main Memory | 1 | 2 | 3 | 4 | 5 | 6 |

Marking algorithm

Procedure Marking

```
mark all pages in the cache                        //first page fault starts a new phase
for every requests x do
    if x is in the cache
        if x is unmarked
            mark x
    else
        if there is no unmarked page
            unmark all pages in the cache                    //start new phase
        p ← somehow chosen page among all unmarked cached pages
        remove p
        insert x at the old position of p
        mark x
```

Request:

Cache | 1 | 2 | 3 |

Main Memory | 1 | 2 | 3 | 4 | 5 | 6 |

Marking algorithm

Procedure Marking

```
mark all pages in the cache                       //first page fault starts a new phase
for every requests x do
    if x is in the cache
        if x is unmarked
            mark x
    else
        if there is no unmarked page
            unmark all pages in the cache          //start new phase
        p ← somehow chosen page among all unmarked cached pages
        remove p
        insert x at the old position of p
        mark x
```

Request:

Cache | 1 | 2 | 3 |

Main Memory | 1 | 2 | 3 | 4 | 5 | 6 |

Marking algorithm

Procedure Marking

mark all pages in the cache //first page fault starts a new phase
for every requests x **do**
  if x is in the cache
    if x is unmarked
      mark x
  else
    if there is no unmarked page
      unmark all pages in the cache //start new phase
    $p \leftarrow$ somehow chosen page among all unmarked cached pages
    remove p
    insert x at the old position of p
    mark x

Request: 1

Cache | 1 | 2 | 3 |

Main Memory | 1 | 2 | 3 | 4 | 5 | 6 |

Marking algorithm

Procedure Marking

```
mark all pages in the cache                          //first page fault starts a new phase
for every requests x do
    if x is in the cache
        if x is unmarked
            mark x
    else
        if there is no unmarked page
            unmark all pages in the cache                          //start new phase
        p ← somehow chosen page among all unmarked cached pages
        remove p
        insert x at the old position of p
        mark x
```

Request: 1

Cache | 1 | 2 | 3 |

Main Memory | 1 | 2 | 3 | 4 | 5 | 6 |

Marking algorithm

Procedure Marking

```
mark all pages in the cache                          //first page fault starts a new phase
for every requests x do
    if x is in the cache
        if x is unmarked
            mark x
    else
        if there is no unmarked page
            unmark all pages in the cache                              //start new phase
        p ← somehow chosen page among all unmarked cached pages
        remove p
        insert x at the old position of p
        mark x
```

Request: 1

Cache | 1 | 2 | 3 |

Main Memory | 1 | 2 | 3 | 4 | 5 | 6 |

Marking algorithm

Procedure Marking

```
mark all pages in the cache                        //first page fault starts a new phase
for every requests x do
    if x is in the cache
        if x is unmarked
            mark x
    else
        if there is no unmarked page
            unmark all pages in the cache                          //start new phase
        p ← somehow chosen page among all unmarked cached pages
        remove p
        insert x at the old position of p
        mark x
```

Request: 1 ,4

Cache	1	2	3

Main Memory	1	2	3	4	5	6

Marking algorithm

Procedure Marking

mark all pages in the cache   //first page fault starts a new phase
for every requests x **do**
  if x is in the cache
    if x is unmarked
      mark x
  else
    if there is no unmarked page
      unmark all pages in the cache   //start new phase
    $p \leftarrow$ somehow chosen page among all unmarked cached pages
    remove p
    insert x at the old position of p
    mark x

Request: 1 ,4

Cache | 1 | 2 | 3 |

Main Memory | 1 | 2 | 3 | 4 | 5 | 6 |

Marking algorithm

Procedure Marking

> **mark** all pages in the cache //first page fault starts a new phase
> **for** every requests x **do**
>   **if** x is in the cache
>   **if** x is unmarked
>   **mark** x
>   **else**
>   **if** there is no unmarked page
>   **unmark** all pages in the cache //start new phase
> $p \leftarrow$ somehow chosen page among all unmarked cached pages
>   **remove** p
>   **insert** x at the old position of p
>   **mark** x

Request: 1 ,4

Cache | 1 | 2 | 3 |

Main Memory | 1 | 2 | 3 | 4 | 5 | 6 |

Marking algorithm

Procedure Marking

mark all pages in the cache //first page fault starts a new phase
for every requests x **do**
  if x is in the cache
    if x is unmarked
      mark x
  else
    if there is no unmarked page
      unmark all pages in the cache //start new phase
    $p \leftarrow$ somehow chosen page among all unmarked cached pages
    remove p
    insert x at the old position of p
    mark x

Request: 1 ,4

Cache | 1 | 2 | 3 |

Main Memory | 1 | 2 | 3 | 4 | 5 | 6 |

Marking algorithm

Procedure Marking

mark all pages in the cache //first page fault starts a new phase
for every requests x **do**
  if x is in the cache
    if x is unmarked
      mark x
  else
    if there is no unmarked page
      unmark all pages in the cache //start new phase
    $p \leftarrow$ somehow chosen page among all unmarked cached pages
    remove p
    insert x at the old position of p
    mark x

Request: 1 ,4

Cache	1	2	3

Main Memory	1	2	3	4	5	6

Marking algorithm

Procedure Marking

> **mark** all pages in the cache //first page fault starts a new phase
> **for** every requests x **do**
>   **if** x is in the cache
>   **if** x is unmarked
>   **mark** x
>   **else**
>   **if** there is no unmarked page
>   **unmark** all pages in the cache //start new phase
> $p \leftarrow$ somehow chosen page among all unmarked cached pages
>   **remove** p
>   **insert** x at the old position of p
>   **mark** x

Request: 1 ,4

Cache

1	2	3

Main Memory

1	2	3	4	5	6

Marking algorithm

Procedure Marking

mark all pages in the cache //first page fault starts a new phase
for every requests x **do**
  if x is in the cache
    if x is unmarked
      mark x
  else
    if there is no unmarked page
      unmark all pages in the cache //start new phase
    $p \leftarrow$ somehow chosen page among all unmarked cached pages
    remove p
    insert x at the old position of p
    mark x

Request: 1 ,4

Cache | 1 | 2 | 3 |

Main Memory | 1 | 2 | 3 | 4 | 5 | 6 |

Marking algorithm

Procedure Marking

mark all pages in the cache //first page fault starts a new phase
for every requests x **do**
  if x is in the cache
    if x is unmarked
      mark x
  else
    if there is no unmarked page
      unmark all pages in the cache //start new phase
    $p \leftarrow$ somehow chosen page among all unmarked cached pages
    remove p
    insert x at the old position of p
    mark x

Request: 1 ,4

Cache	1	2	3

Main Memory	1	2	3	4	5	6

Procedure Marking

```
mark all pages in the cache                         //first page fault starts a new phase
for every requests x do
    if x is in the cache
        if x is unmarked
            mark x
    else
        if there is no unmarked page
            unmark all pages in the cache           //start new phase
        p ← somehow chosen page among all unmarked cached pages
        remove p
        insert x at the old position of p
        mark x
```

Request: 1 ,4

Cache		2	3

Main Memory	1	2	3	4	5	6

Marking algorithm

Procedure Marking

mark all pages in the cache //first page fault starts a new phase
for every requests x **do**
  if x is in the cache
    if x is unmarked
      mark x
  else
    if there is no unmarked page
      unmark all pages in the cache //start new phase
    $p \leftarrow$ somehow chosen page among all unmarked cached pages
    remove p
    insert x at the old position of p
    mark x

Request: 1 ,4

Cache | | 2 | 3 |

Main Memory | 1 | 2 | 3 | 4 | 5 | 6 |

Marking algorithm

Procedure Marking

mark all pages in the cache //first page fault starts a new phase
for every requests x **do**
  if x is in the cache
    if x is unmarked
      mark x
  else
    if there is no unmarked page
      unmark all pages in the cache //start new phase
    $p \leftarrow$ somehow chosen page among all unmarked cached pages
    remove p
    insert x at the old position of p
    mark x

Request: 1 ,4

Cache	4	2	3

Main Memory	1	2	3	4	5	6

Marking algorithm

Procedure Marking

```
mark all pages in the cache                    //first page fault starts a new phase
for every requests x do
    if x is in the cache
        if x is unmarked
            mark x
    else
        if there is no unmarked page
            unmark all pages in the cache                      //start new phase
        p ← somehow chosen page among all unmarked cached pages
        remove p
        insert x at the old position of p
        mark x
```

Request: 1 ,4

Cache	4	2	3

Main Memory	1	2	3	4	5	6

Marking algorithm

Procedure Marking

mark all pages in the cache //first page fault starts a new phase
for every requests x **do**
  if x is in the cache
    if x is unmarked
      mark x
  else
    if there is no unmarked page
      unmark all pages in the cache //start new phase
    $p \leftarrow$ somehow chosen page among all unmarked cached pages
    remove p
    insert x at the old position of p
    mark x

Request: 1 ,4

Cache	4	2	3

Main Memory	1	2	3	4	5	6

Marking algorithm

Procedure Marking

```
mark all pages in the cache                          //first page fault starts a new phase
for every requests x do
    if x is in the cache
        if x is unmarked
            mark x
    else
        if there is no unmarked page
            unmark all pages in the cache            //start new phase
        p ← somehow chosen page among all unmarked cached pages
        remove p
        insert x at the old position of p
        mark x
```

Request: 1 ,4 ,2

Cache | 4 | 2 | 3 |

Main Memory | 1 | 2 | 3 | 4 | 5 | 6 |

Marking algorithm

Procedure Marking

mark all pages in the cache //first page fault starts a new phase
for every requests x **do**
  if x is in the cache
    if x is unmarked
      mark x
  else
    if there is no unmarked page
      unmark all pages in the cache //start new phase
    $p \leftarrow$ somehow chosen page among all unmarked cached pages
    remove p
    insert x at the old position of p
    mark x

Request: 1 ,4 ,2

Cache	4	2	3

Main Memory	1	2	3	4	5	6

Marking algorithm

Procedure Marking

mark all pages in the cache //first page fault starts a new phase
for every requests x **do**
  if x is in the cache
    if x is unmarked
      mark x
  else
    if there is no unmarked page
      unmark all pages in the cache //start new phase
    $p \leftarrow$ somehow chosen page among all unmarked cached pages
    remove p
    insert x at the old position of p
    mark x

Request: 1 ,4 ,2

Cache | 4 | 2 | 3 |

Main Memory | 1 | 2 | 3 | 4 | 5 | 6 |

Marking algorithm

Procedure Marking

mark all pages in the cache //first page fault starts a new phase
for every requests x **do**
  if x is in the cache
    if x is unmarked
      mark x
  else
    if there is no unmarked page
      unmark all pages in the cache //start new phase
    $p \leftarrow$ somehow chosen page among all unmarked cached pages
    remove p
    insert x at the old position of p
    mark x

Request: 1 ,4 ,2

Cache

4	2	3

Main Memory

1	2	3	4	5	6

Marking algorithm

Procedure Marking

mark all pages in the cache //first page fault starts a new phase
for every requests x **do**
  if x is in the cache
    if x is unmarked
      mark x
  else
    if there is no unmarked page
      unmark all pages in the cache //start new phase
    $p \leftarrow$ somehow chosen page among all unmarked cached pages
    remove p
    insert x at the old position of p
    mark x

Request: 1 ,4 ,2

Cache | 4 | 2 | 3 |

Main Memory | 1 | 2 | 3 | 4 | 5 | 6 |

Marking algorithm

Procedure Marking

```
mark all pages in the cache                          //first page fault starts a new phase
for every requests x do
    if x is in the cache
        if x is unmarked
            mark x
    else
        if there is no unmarked page
            unmark all pages in the cache                              //start new phase
        p ← somehow chosen page among all unmarked cached pages
        remove p
        insert x at the old position of p
        mark x
```

Request: 1 ,4 ,2 ,4

Cache | 4 | 2 | 3 |

Main Memory | 1 | 2 | 3 | 4 | 5 | 6 |

Marking algorithm

Procedure Marking

> **mark** all pages in the cache   //first page fault starts a new phase
> **for** every requests x **do**
>   **if** x is in the cache
>   **if** x is unmarked
>   **mark** x
>   **else**
>   **if** there is no unmarked page
>   **unmark** all pages in the cache   //start new phase
> $p \leftarrow$ somehow chosen page among all unmarked cached pages
>   **remove** p
>   **insert** x at the old position of p
>   **mark** x

Request: 1 ,4 ,2 ,4

Cache | 4 | 2 | 3 |

Main Memory | 1 | 2 | 3 | 4 | 5 | 6 |

Marking algorithm

Procedure Marking

mark all pages in the cache //first page fault starts a new phase
for every requests x **do**
  if x is in the cache
    if x is unmarked
      mark x
  else
    if there is no unmarked page
      unmark all pages in the cache //start new phase
    $p \leftarrow$ somehow chosen page among all unmarked cached pages
    remove p
    insert x at the old position of p
    mark x

Request: 1 ,4 ,2 ,4

Cache | 4 | 2 | 3 |

Main Memory | 1 | 2 | 3 | 4 | 5 | 6 |

Marking algorithm

Procedure Marking

> **mark** all pages in the cache　　//first page fault starts a new phase
> **for** every requests x **do**
> 　**if** x is in the cache
> 　　**if** x is unmarked
> 　　　**mark** x
> 　**else**
> 　　**if** there is no unmarked page
> 　　　**unmark** all pages in the cache　　//start new phase
> 　　$p \leftarrow$ somehow chosen page among all unmarked cached pages
> 　　**remove** p
> 　　**insert** x at the old position of p
> 　　**mark** x

Request: 1 ,4 ,2 ,4 ,5

Cache | 4 | 2 | 3 |

Main Memory | 1 | 2 | 3 | 4 | 5 | 6 |

Marking algorithm

Procedure Marking

mark all pages in the cache //first page fault starts a new phase
for every requests x **do**
  if x is in the cache
    if x is unmarked
      mark x
  else
    if there is no unmarked page
      unmark all pages in the cache //start new phase
    $p \leftarrow$ somehow chosen page among all unmarked cached pages
    remove p
    insert x at the old position of p
    mark x

Request: 1 ,4 ,2 ,4 ,5

Cache | 4 | 2 | 3 |

Main Memory | 1 | 2 | 3 | 4 | 5 | 6 |

Marking algorithm

Procedure Marking

mark all pages in the cache //first page fault starts a new phase
for every requests x **do**
  if x is in the cache
    if x is unmarked
      mark x
  else
    if there is no unmarked page
      unmark all pages in the cache //start new phase
    $p \leftarrow$ somehow chosen page among all unmarked cached pages
    remove p
    insert x at the old position of p
    mark x

Request: 1 ,4 ,2 ,4 ,5

Cache	4	2	3

Main Memory	1	2	3	4	5	6

Marking algorithm

Procedure Marking

> **mark** all pages in the cache //first page fault starts a new phase
> **for** every requests x **do**
>   **if** x is in the cache
>   **if** x is unmarked
>   **mark** x
>   **else**
>   **if** there is no unmarked page
>   **unmark** all pages in the cache //start new phase
> $p \leftarrow$ somehow chosen page among all unmarked cached pages
>   **remove** p
>   **insert** x at the old position of p
>   **mark** x

Request: 1 ,4 ,2 ,4 ,5

Cache [4] [2] [3]

Main Memory [1] [2] [3] [4] [5] [6]

Marking algorithm

Procedure Marking

mark all pages in the cache //first page fault starts a new phase
for every requests x **do**
  if x is in the cache
    if x is unmarked
      mark x
  else
    if there is no unmarked page
      unmark all pages in the cache //start new phase
    $p \leftarrow$ somehow chosen page among all unmarked cached pages
    remove p
    insert x at the old position of p
    mark x

Request: 1 ,4 ,2 ,4 ,5

Cache | 4 | 2 | 3 |

Main Memory | 1 | 2 | 3 | 4 | 5 | 6 |

Marking algorithm

Procedure Marking

mark all pages in the cache //first page fault starts a new phase
for every requests x **do**
  if x is in the cache
    if x is unmarked
      mark x
  else
    if there is no unmarked page
      unmark all pages in the cache //start new phase
    $p \leftarrow$ somehow chosen page among all unmarked cached pages
    remove p
    insert x at the old position of p
    mark x

Request: 1 ,4 ,2 ,4 ,5

Cache | 4 | 2 | 3 |

Main Memory | 1 | 2 | 3 | 4 | 5 | 6 |

Marking algorithm

Procedure Marking

```
mark all pages in the cache                      //first page fault starts a new phase
for every requests x do
    if x is in the cache
        if x is unmarked
            mark x
    else
        if there is no unmarked page
            unmark all pages in the cache            //start new phase
        p ← somehow chosen page among all unmarked cached pages
        remove p
        insert x at the old position of p
        mark x
```

Request: 1 ,4 ,2 ,4 ,5

Cache | 4 | 2 | |

Main Memory | 1 | 2 | 3 | 4 | 5 | 6 |

Marking algorithm

Procedure Marking

mark all pages in the cache //first page fault starts a new phase
for every requests x **do**
  if x is in the cache
    if x is unmarked
      mark x
  else
    if there is no unmarked page
      unmark all pages in the cache //start new phase
    $p \leftarrow$ somehow chosen page among all unmarked cached pages
    remove p
    insert x at the old position of p
    mark x

Request: 1 ,4 ,2 ,4 ,5

Cache | 4 | 2 | |

Main Memory | 1 | 2 | 3 | 4 | 5 | 6 |

Marking algorithm

Procedure Marking

```
mark all pages in the cache                 //first page fault starts a new phase
for every requests x do
    if x is in the cache
        if x is unmarked
            mark x
    else
        if there is no unmarked page
            unmark all pages in the cache   //start new phase
        p ← somehow chosen page among all unmarked cached pages
        remove p
        insert x at the old position of p
        mark x
```

Request: 1 ,4 ,2 ,4 ,5

Cache 4 2 5

Main Memory 1 2 3 4 5 6

Marking algorithm

Procedure Marking

mark all pages in the cache //first page fault starts a new phase
for every requests x **do**
  if x is in the cache
    if x is unmarked
      mark x
  else
    if there is no unmarked page
      unmark all pages in the cache //start new phase
    $p \leftarrow$ somehow chosen page among all unmarked cached pages
    remove p
    insert x at the old position of p
    mark x

Request: 1 ,4 ,2 ,4 ,5

Marking algorithm

Procedure Marking

```
mark all pages in the cache                          //first page fault starts a new phase
for every requests x do
    if x is in the cache
        if x is unmarked
            mark x
    else
        if there is no unmarked page
            unmark all pages in the cache                              //start new phase
        p ← somehow chosen page among all unmarked cached pages
        remove p
        insert x at the old position of p
        mark x
```

Request: 1 ,4 ,2 ,4 ,5

Cache | 4 | 2 | 5 |

Main Memory | 1 | 2 | 3 | 4 | 5 | 6 |

Competitive marking algorithms

Request: $\boxed{1,\ 3,\ 3,\ 5,}\ \boxed{4,\ 3,\ 2,}\ \boxed{5,\ 2,\ 1,\ 1,}\ \boxed{3,\ 2,\ 3,\ 1,\ 3,\ 3,}\ \boxed{5,\ 3,\ 5,\ 2}$

Phase 1 Phase 2 Phase 3 Phase 4 Phase 5

Theorem

Every deterministic marking algorithm is k-competitive.

Competitive marking algorithms

Request:	1, 3, 3, 5,	4, 3, 2,	5, 2, 1, 1,	3, 2, 3, 1, 3, 3,	5, 3, 5, 2
Phase 1	Phase 2	Phase 3	Phase 4	Phase 5

Theorem

Every deterministic marking algorithm is k-competitive.

Proof.

Consider the k-phase partition $P_1, P_2, \ldots, P_N$ of the input I.

Competitive marking algorithms

Request: $\boxed{1,\ 3,\ 3,\ 5,}\ \boxed{4,\ 3,\ 2,}\ \boxed{5,\ 2,\ 1,\ 1,}\ \boxed{3,\ 2,\ 3,\ 1,\ 3,\ 3,}\ \boxed{5,\ 3,\ 5,\ 2}$

Phase 1 Phase 2 Phase 3 Phase 4 Phase 5

Theorem

Every deterministic marking algorithm is k-competitive.

Proof.

Consider the k-phase partition $P_1, P_2, \ldots, P_N$ of the input I.
Observe that OPT makes at least N page faults.

Competitive marking algorithms

Request:	1, 3, 3, 5,	4, 3, 2,	5, 2, 1, 1,	3, 2, 3, 1, 3, 3,	5, 3, 5, 2
	Phase 1	Phase 2	Phase 3	Phase 4	Phase 5

Theorem

Every deterministic marking algorithm is k-competitive.

Proof.

Consider the k-phase partition $P_1, P_2, \ldots, P_N$ of the input I.
Observe that $\textsc{Opt}$ makes at least N page faults.
Furthermore, consider the phase partition $P_{\textsc{Alg},1}, P_{\textsc{Alg},2}, \ldots P_{\textsc{Alg},N'}$. Since the marking algorithm makes at most k page faults per phase of the algorithm, and thus, kN' page faults in total. It remains to show that $N = N'$.

Competitive marking algorithms

Request: $\boxed{1,\ 3,\ 3,\ 5,}\boxed{4,\ 3,\ 2,}\boxed{5,\ 2,\ 1,\ 1,}\boxed{3,\ 2,\ 3,\ 1,\ 3,\ 3,}\boxed{5,\ 3,\ 5,\ 2}$

Phase 1 Phase 2 Phase 3 Phase 4 Phase 5

Theorem

Every deterministic marking algorithm is k-competitive.

Proof.

Consider the k-phase partition $P_1, P_2, \ldots, P_N$ of the input I.

Observe that $\textsc{Opt}$ makes at least N page faults.

Furthermore, consider the phase partition $P_{\textsc{Alg},1}, P_{\textsc{Alg},2}, \ldots P_{\textsc{Alg},N'}$. Since the marking algorithm makes at most k page faults per phase of the algorithm, and thus, kN' page faults in total. It remains to show that $N = N'$.

This follows, if for every $1 \leq i \leq N$, $P_i = P_{\textsc{Alg},i}$.

Competitive marking algorithms

Request:	1, 3, 3, 5,	4, 3, 2,	5, 2, 1, 1,	3, 2, 3, 1, 3, 3,	5, 3, 5, 2
	Phase 1	Phase 2	Phase 3	Phase 4	Phase 5

Theorem

Every deterministic marking algorithm is k-competitive.

Proof.

Consider the k-phase partition $P_1, P_2, \ldots, P_N$ of the input I.

Observe that $\textsc{Opt}$ makes at least N page faults.

Furthermore, consider the phase partition $P_{\textsc{Alg},1}, P_{\textsc{Alg},2}, \ldots P_{\textsc{Alg},N'}$. Since the marking algorithm makes at most k page faults per phase of the algorithm, and thus, kN' page faults in total. It remains to show that $N = N'$.

This follows, if for every $1 \leq i \leq N$, $P_i = P_{\textsc{Alg},i}$.

P_1 and $P_{\textsc{Alg},1}$ both start with the first request that causes a page fault.

Competitive marking algorithms

Request: $\boxed{1, 3, 3, 5,}\boxed{4, 3, 2,}\boxed{5, 2, 1, 1,}\boxed{3, 2, 3, 1, 3, 3,}\boxed{5, 3, 5, 2}$

Phase 1 Phase 2 Phase 3 Phase 4 Phase 5

Theorem

Every deterministic marking algorithm is k-competitive.

Proof.

Consider the k-phase partition $P_1, P_2, \ldots, P_N$ of the input I.
Observe that $\textsc{Opt}$ makes at least N page faults.
Furthermore, consider the phase partition $P_{\textsc{Alg},1}, P_{\textsc{Alg},2}, \ldots P_{\textsc{Alg},N'}$. Since the marking algorithm makes at most k page faults per phase of the algorithm, and thus, kN' page faults in total. It remains to show that $N = N'$.
This follows, if for every $1 \le i \le N$, $P_i = P_{\textsc{Alg},i}$.
P_1 and $P_{\textsc{Alg},1}$ both start with the first request that causes a page fault.
If k distinct pages were requested within phase $P_{\textsc{Alg},i}$, all pages in the cache are marked. Then with the $(k+1)$-th distinct page, the algorithm starts a new phase $P_{\textsc{Alg},i+1}$.

Competitive marking algorithms

Request:	1, 3, 3, 5,	4, 3, 2,	5, 2, 1, 1,	3, 2, 3, 1, 3, 3,	5, 3, 5, 2
	Phase 1	Phase 2	Phase 3	Phase 4	Phase 5

Theorem

Every deterministic marking algorithm is k-competitive.

Proof.
Consider the k-phase partition $P_1, P_2, \ldots, P_N$ of the input I.
Observe that $\textsc{Opt}$ makes at least N page faults.
Furthermore, consider the phase partition $P_{\textsc{Alg},1}, P_{\textsc{Alg},2}, \ldots P_{\textsc{Alg},N'}$. Since the marking algorithm makes at most k page faults per phase of the algorithm, and thus, kN' page faults in total. It remains to show that $N = N'$.
This follows, if for every $1 \leq i \leq N$, $P_i = P_{\textsc{Alg},i}$.
P_1 and $P_{\textsc{Alg},1}$ both start with the first request that causes a page fault.
If k distinct pages were requested within phase $P_{\textsc{Alg},i}$, all pages in the cache are marked. Then with the $(k+1)$-th distinct page, the algorithm starts a new phase $P_{\textsc{Alg},i+1}$.
By definition, also P_{i+1} starts after $(k+1)$ distinct requests. □

How to randomize

How did the adversary manage to beat the deterministic algorithm?

- Every page the algorithm removed from the cache was the wrong choice.

How did the adversary manage to beat the deterministic algorithm?

- Every page the algorithm removed from the cache was the wrong choice.

How did the adversary manage to beat the deterministic algorithm?

- Every page the algorithm removed from the cache was the wrong choice.

How to randomize

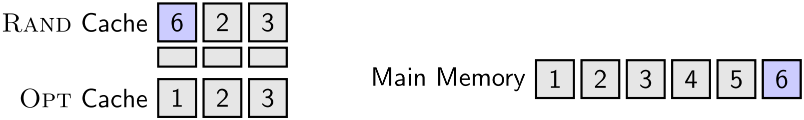
How did the adversary manage to beat the deterministic algorithm?

- Every page the algorithm removed from the cache was the wrong choice.

How to randomize

How did the adversary manage to beat the deterministic algorithm?

- Every page the algorithm removed from the cache was the wrong choice.



How to randomize

How did the adversary manage to beat the deterministic algorithm?

- Every page the algorithm removed from the cache was the wrong choice.

How to randomize

How did the adversary manage to beat the deterministic algorithm?

- Every page the algorithm removed from the cache was the wrong choice.

How did the adversary manage to beat the deterministic algorithm?

- Every page the algorithm removed from the cache was the wrong choice.

How to randomize

How did the adversary manage to beat the deterministic algorithm?

- Every page the algorithm removed from the cache was the wrong choice.

How to randomize

How did the adversary manage to beat the deterministic algorithm?

- Every page the algorithm removed from the cache was the wrong choice.

How to randomize

How did the adversary manage to beat the deterministic algorithm?

- Every page the algorithm removed from the cache was the wrong choice.

How to randomize

How did the adversary manage to beat the deterministic algorithm?

- Every page the algorithm removed from the cache was the wrong choice.

How to randomize

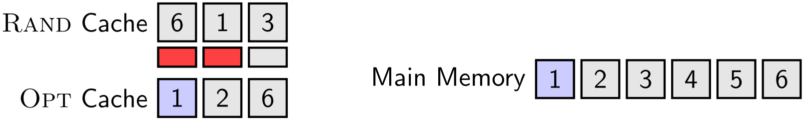
How did the adversary manage to beat the deterministic algorithm?

- Every page the algorithm removed from the cache was the wrong choice.

How to randomize

How did the adversary manage to beat the deterministic algorithm?

- Every page the algorithm removed from the cache was the wrong choice.



How to randomize

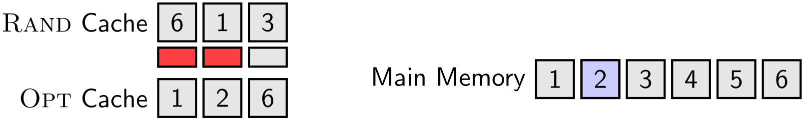
How did the adversary manage to beat the deterministic algorithm?

- Every page the algorithm removed from the cache was the wrong choice.

How to randomize

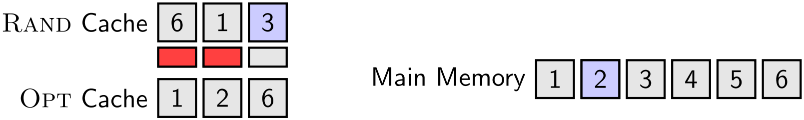
How did the adversary manage to beat the deterministic algorithm?

- Every page the algorithm removed from the cache was the wrong choice.



How did the adversary manage to beat the deterministic algorithm?

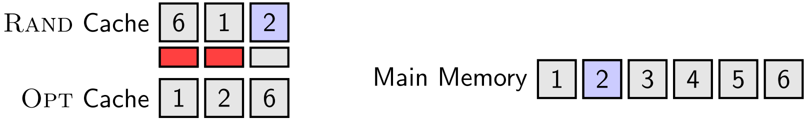
- Every page the algorithm removed from the cache was the wrong choice.



How to randomize

How did the adversary manage to beat the deterministic algorithm?

- Every page the algorithm removed from the cache was the wrong choice.



How to randomize

How did the adversary manage to beat the deterministic algorithm?

- Every page the algorithm removed from the cache was the wrong choice.

How to randomize

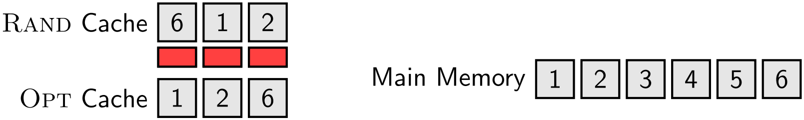
How did the adversary manage to beat the deterministic algorithm?

- Every page the algorithm removed from the cache was the wrong choice.

How to randomize

How did the adversary manage to beat the deterministic algorithm?

- Every page the algorithm removed from the cache was the wrong choice.



How can we avoid this?

- Select a page to remove from the cache **randomly**.

How to randomize

How did the adversary manage to beat the deterministic algorithm?

- Every page the algorithm removed from the cache was the wrong choice.

How can we avoid this?

- Select a page to remove from the cache **randomly**.
- The adversary now has to guess which page was removed by the algorithm.

How to randomize

How did the adversary manage to beat the deterministic algorithm?

- Every page the algorithm removed from the cache was the wrong choice.

How can we avoid this?

- Select a page to remove from the cache **randomly**.
- The adversary now has to guess which page was removed by the algorithm.
- Adversary guesses correctly with probability $1/(\text{cache size} - \#\text{marked pages})$.

Randomized marking algorithm

Procedure Random marking

```
mark all pages in the cache                                //first page fault starts a new phase
for every requests x do
    if x is in the cache
        if x is unmarked
            mark x
    else
        if there is no unmarked page
            unmark all pages in the cache                  //start new phase
        p ← somehow chosen page among all unmarked cached pages
        remove p
        insert x at the old position of p
        mark x
```

Randomized marking algorithm

Procedure Random marking

```
mark all pages in the cache                              //first page fault starts a new phase
for every requests x do
    if x is in the cache
        if x is unmarked
            mark x
    else
        if there is no unmarked page
            unmark all pages in the cache                //start new phase
        p ← somehow chosen page among all unmarked cached pages
        remove p
        insert x at the old position of p
        mark x
```

Randomized marking algorithm

Procedure Random marking

```
mark all pages in the cache                    //first page fault starts a new phase
for every requests x do
    if x is in the cache
        if x is unmarked
            mark x
    else
        if there is no unmarked page
            unmark all pages in the cache                      //start new phase
        p ← randomly chosen page among all unmarked cached pages
        remove p
        insert x at the old position of p
        mark x
```

Harmonic numbers

Harmonic number The *n*-th harmonic number is the sum of multiplicative inverses of the first *n* natural numbers.

$$H_n = 1 + \frac{1}{2} + \cdots + \frac{1}{n} = \sum_{i=1}^{n} \frac{1}{i}$$

- Does not converge but grows slowly.
- $H_n = \ln(n) + \frac{1}{2} + \frac{1}{2n} + 0.077 \approx \ln(n)$.

The RMark algorithm

For the randomized marking algorithms we did not specify how we decide which page to replace, i.e., what distribution we pick.

Let $\mathrm{RM}\textsc{ark}$ be a randomized marking algorithm that replaces unmarked pages uniformly at random.

RMark analysis

Theorem

The $\textsc{RMark}$ algorithm is $2H_k$-competitive in expectation.

RMark analysis

Theorem

The RMARK algorithm is $2H_k$-competitive in expectation.

Assumptions

- The phases of the k-phase partition and the partition by the marking algorithm line up. (As we showed in the proof that deterministic marking is k-competitive).

RMark analysis

Theorem

The RMARK a[...]

Note that the phases only depend on the input and not on the random decisions of the RMARK algorithm.

Assumptions

- The phases of the k-phase partition and the partition by the marking algorithm line up. (As we showed in the proof that deterministic marking is k-competitive).

RMark analysis

Theorem

The $\text{RM}\textsc{ark}$ algorithm is $2H_k$-competitive in expectation.

Assumptions

- The phases of the k-phase partition and the partition by the marking algorithm line up. (As we showed in the proof that deterministic marking is k-competitive).
- In a single phase, no page is requested more then once.

RMark analysis

Theorem

The $\mathrm{RM}\textsc{ark}$ algorithm is $2H_k$-com

A page that is requested for a second time during a phase never causes a page fault, since the first time the page will be marked.

Assumptions

- The phases of the k-pha
  algorithm line up. (As we showe ... marking is k-competitive).
- In a single phase, no page is requested more then once.

RMark analysis

Theorem

The RMARK algorithm is $2H_k$-competitive in expectation.

Assumptions

- The phases of the k-phase partition and the partition by the marking algorithm line up. (As we showed in the proof that deterministic marking is k-competitive).
- In a single phase, no page is requested more then once.

Let us analyze a single phase P_j with $1 \leq j \leq N$.

Definitions

- New pages: pages that entered the cache **during** phase P_j.
- Old pages: pages that were in the cache **at the start** of phase P_j.

RMark analysis

Theorem

The RMARK algorithm is $2H_k$-competitive in expectation.

Assumptions

- The phases of the k-phase partition and the partition by the marking algorithm line up. (As we showed in the proof that deterministic marking is k-competitive).
- In a single phase, no page is requested more then once.

Let us analyze a single phase P_j with $1 \leq j \leq N$.

Definitions

- New pages: pages that entered the cache **during** phase P_j.
- Old pages: pages that were in the cache **at the start** of phase P_j.
- ℓ_j: the number of new pages requested in phase P_j.
- $k - \ell_j$: the number old pages are requested in phase P_j.

RMark analysis

Theorem

The RMARK algorithm is $2H_k$-competitive in expectation.

Assumptions

- The phases of the k-phase partition and the partition by the marking algorithm line up. (As we showed in the proof that deterministic marking is k-competitive).
- In a single phase, no page is requested more then once.
- The adversary first requests all new pages, and then old ones.

Let us analyze a single phase P_j with $1 \leq j \leq N$.

Definitions

- New pages: pages that entered the cache **during** phase P_j.
- Old pages: pages that were in the cache **at the start** of phase P_j.
- ℓ_j: the number of new pages requested in phase P_j.
- $k - \ell_j$: the number old pages are requested in phase P_j.

RMark analysis

Theorem

The RMARK algorithm is $2H_k$-competitive in expectation.

Assumptions

- The phases of the k-phase pa
  algorithm line up. (As we sh
  k-competitive).
- In a single phase, no page is requested mo
- The adversary first requests all new pages, and then old ones.

This is the best strategy for the adversary, as this maximizes the probability of a page fault on the old pages.

Let us analyze a single phase P_j with $1 \leq j \leq N$.

Definitions

- New pages: pages that entered the cache **during** phase P_j.
- Old pages: pages that were in the cache **at the start** of phase P_j.
- ℓ_j: the number of new pages requested in phase P_j.
- $k - \ell_j$: the number old pages are requested in phase P_j.

RMark analysis

Theorem

The RMARK algorithm is $2H_k$-competitive in expectation.

Let $\mathcal{E}_i$ be the event that the i-th old page requested is not in the cache at the moment it is requested. Then,

$$\mathbf{Pr}\left(\mathcal{E}_1\right) = 1 - \frac{k - \ell_j}{k}$$

RMark analysis

$k - \ell_j$: the number of unmarked old pages that have not been removed from the cache.
k: the number of unmarked old pages.

Theorem

The RMark algorit...

Let $\mathcal{E}_i$ be the event that the i-t... page requested is not in the cache at the moment it is requested. Then,

$$\mathbf{Pr}(\mathcal{E}_1) = 1 - \frac{k - \ell_j}{k}$$

RMark analysis

Theorem

The RMark algorithm is $2H_k$-competitive in expectation.

Let $\mathcal{E}_i$ be the event that the i-th old page requested is not in the cache at the moment it is requested. Then,

$$\mathbf{Pr}\left(\mathcal{E}_1\right) = 1 - \frac{k - \ell_j}{k} \qquad \mathbf{Pr}\left(\mathcal{E}_2\right) = 1 - \frac{k - \ell_j - 1}{k - 1}$$

RMark analysis

Theorem

The RMARK algorithm is $2H_k$-competitive in expectation.

Let $\mathcal{E}_i$ be the event that the i-th old page requested is not in the cache at the moment it is requested. Then,

$$\mathbf{Pr}\left(\mathcal{E}_1\right) = 1 - \frac{k - \ell_j}{k} \qquad \mathbf{Pr}\left(\mathcal{E}_2\right) = 1 - \frac{k - \ell_j - 1}{k - 1}$$

Then, in general, for the i-th requested old page,

$$\mathbf{Pr}\left(\mathcal{E}_i\right) = 1 - \frac{k - \ell_j - (i-1)}{k - (i-1)} = \frac{\ell_j}{k - (i-1)}$$

RMark analysis

Theorem

The $\textsc{RMark}$ algorithm is $2H_k$ competitive in expectation.

Let $\mathcal{C}_j$ be a random variable that is equal tot the cost of the $\textsc{RMark}$ algorithm in phase P_j.
Since the cost for the algorithm is 1 in the case of a page fault, the expected cost equals

$$\mathbb{E}\left[\mathcal{C}_j\right] = \ell_j + \sum_{i=1}^{k-\ell_j} \frac{\ell_j}{k-(i-1)}$$

RMark analysis

Theorem

The $\textsc{RMark}$ algorithm is $2H_k$ competitive in expectation.

Let $\mathcal{C}_j$ be a random variable that is equal tot the cost of the $\textsc{RMark}$ algorithm in phase P_j.
Since the cost for the algorithm is 1 in the case of a page fault, the expected cost equals

$$\mathbb{E}\left[\mathcal{C}_j\right] = \ell_j + \sum_{i=1}^{k-\ell_j} \frac{\ell_j}{k-(i-1)} = \ell_j + \ell_j\left(\frac{1}{k} + \frac{1}{k-1} + \cdots + \frac{1}{\ell_j+1}\right)$$

RMark analysis

Theorem

The $\textsc{RMark}$ algorithm is $2H_k$ competitive in expectation.

Let $\mathcal{C}_j$ be a random variable that is equal tot the cost of the $\textsc{RMark}$ algorithm in phase P_j.
Since the cost for the algorithm is 1 in the case of a page fault, the expected cost equals

$$\mathbb{E}\left[\mathcal{C}_j\right] = \ell_j + \sum_{i=1}^{k-\ell_j} \frac{\ell_j}{k-(i-1)} = \ell_j + \ell_j \left(\frac{1}{k} + \frac{1}{k-1} + \cdots + \frac{1}{\ell_j+1}\right)$$

$$= \ell_j + \ell_j \left(\underbrace{\frac{1}{k} + \frac{1}{k-1} + \cdots + 1}_{H_k} - \underbrace{\left(\frac{1}{\ell_j} + \frac{1}{\ell_j - 1} + \cdots + 1\right)}_{H_{\ell_j}}\right)$$

$$= \ell_j(H_k - H_{\ell_j} + 1) \leq \ell_j H_k$$

RMark analysis

Theorem

The $\textsc{RMark}$ algorithm is $2H_k$ competitive in expectation.

Let $\mathcal{C}_j$ be a random variable that is equal tot the cost of the $\textsc{RMark}$ algorithm in phase P_j.
Since the cost for the algorithm is 1 in the case of a page fault, the expected cost equals

$$\mathbb{E}\left[\mathcal{C}_j\right] = \ell_j + \sum_{i=1}^{k-\ell_j} \frac{\ell_j}{k-(i-1)} = \ell_j + \ell_j \left(\frac{1}{k} + \frac{1}{k-1} + \cdots + \frac{1}{\ell_j+1}\right)$$

$$= \ell_j + \ell_j \left(\underbrace{\frac{1}{k} + \frac{1}{k-1} + \cdots + 1}_{H_k} - \underbrace{\left(\frac{1}{\ell_j} + \frac{1}{\ell_j - 1} + \cdots + 1\right)}_{H_{\ell_j}}\right)$$

$$= \ell_j(H_k - H_{\ell_j} + 1) \leq \ell_j H_k$$

Then,

$$\mathbb{E}\left[\textsc{Rand}\right] = \sum_{j=1}^{N} \mathbb{E}\left[\mathcal{C}_j\right] = \sum_{j=1}^{N} \ell_j H_k$$

RMark analysis

Theorem

The $\mathrm{RM{\scriptsize ARK}}$ algorithm is $2H_k$-competitive in expectation.

For the cost of $\mathrm{O{\scriptsize PT}}$, consider two consecutive phases P_{j-1} and P_j.

RMark analysis

Theorem

The $\mathrm{RM{\small ARK}}$ algorithm is $2H_k$-competitive in expectation.

For the cost of $\mathrm{O{\small PT}}$, consider two consecutive phases P_{j-1} and P_j.
At least $k + \ell_j$ distinct pages were requested.

RMark analysis

Theorem

The $\textsc{RMark}$ algorithm is $2H_k$-competitive in expectation.

For the cost of $\textsc{Opt}$, consider two consecutive phases P_{j-1} and P_j.
At least $k + \ell_j$ distinct pages were requested.
Thus $\textsc{Opt}$ needs to make ℓ_j page faults in these two phases.

Either $\underbrace{P_1, P_2}_{\ell_2 \text{ faults}}, \underbrace{P_3, P_4}_{\ell_4 \text{ faults}}, P_5, \ldots$ or $P_1, \underbrace{P_2, P_3}_{\ell_3 \text{ faults}}, \underbrace{P_4, P_5}_{\ell_5 \text{ faults}}, \ldots$

RMark analysis

Theorem

The $\textsc{RMark}$ algorithm is $2H_k$-competitive in expectation.

For the cost of $\textsc{Opt}$, consider two consecutive phases P_{j-1} and P_j.
At least $k + \ell_j$ distinct pages were requested.
Thus $\textsc{Opt}$ needs to make ℓ_j page faults in these two phases.

$$\text{Either } \underbrace{P_1, P_2}_{\ell_2 \text{ faults}}, \underbrace{P_3, P_4}_{\ell_4 \text{ faults}}, P_5, \ldots \qquad \text{or } P_1, \underbrace{P_2, P_3}_{\ell_3 \text{ faults}}, \underbrace{P_4, P_5}_{\ell_5 \text{ faults}}, \ldots$$

Moreover, $\textsc{Opt}$ causes ℓ_1 page faults in phase P_1 since $\textsc{Opt}$ and $\textsc{Rand}$ start with the same cache contents. Then,

$$\textsc{Opt} \geq \sum_{j=1}^{\lfloor N/2 \rfloor} \ell_{2j} \qquad \text{and} \qquad \textsc{Opt} \geq \sum_{j=1}^{\lceil N/2 \rceil} \ell_{2j-1}$$

RMark analysis

Theorem

The $\textsc{RMark}$ algorithm is $2H_k$-competitive in expectation.

For the cost of $\textsc{Opt}$, consider two consecutive phases P_{j-1} and P_j.
At least $k + \ell_j$ distinct pages were requested.
Thus $\textsc{Opt}$ needs to make ℓ_j page faults in these two phases.

$$\text{Either } \underbrace{P_1, P_2}_{\ell_2 \text{ faults}}, \underbrace{P_3, P_4}_{\ell_4 \text{ faults}}, P_5, \ldots \qquad \text{or } P_1, \underbrace{P_2, P_3}_{\ell_3 \text{ faults}}, \underbrace{P_4, P_5}_{\ell_5 \text{ faults}}, \ldots$$

Moreover, $\textsc{Opt}$ causes ℓ_1 page faults in phase P_1 since $\textsc{Opt}$ and $\textsc{Rand}$ start with the same cache contents. Then,

$$\textsc{Opt} \geq \sum_{j=1}^{\lfloor N/2 \rfloor} \ell_{2j} \qquad \text{and} \qquad \textsc{Opt} \geq \sum_{j=1}^{\lceil N/2 \rceil} \ell_{2j-1}$$

$$\textsc{Opt} \geq \max \left\{ \sum_{j=1}^{\lfloor N/2 \rfloor} \ell_{2j}, \sum_{j=1}^{\lceil N/2 \rceil} \ell_{2j-1} \right\}$$

RMark analysis

Theorem

The $\textsc{RMark}$ algorithm is $2H_k$-competitive in expectation.

For the cost of $\textsc{Opt}$, consider two consecutive phases P_{j-1} and P_j.
At least $k + \ell_j$ distinct pages were requested.
Thus $\textsc{Opt}$ needs to make ℓ_j page faults in these two phases.

$$\text{Either } \underbrace{P_1, P_2}_{\ell_2 \text{ faults}}, \underbrace{P_3, P_4}_{\ell_4 \text{ faults}}, P_5, \ldots \qquad \text{or } P_1, \underbrace{P_2, P_3}_{\ell_3 \text{ faults}}, \underbrace{P_4, P_5}_{\ell_5 \text{ faults}}, \ldots$$

Moreover, $\textsc{Opt}$ causes ℓ_1 page faults in phase P_1 since $\textsc{Opt}$ and $\textsc{Rand}$ start with the same cache contents. Then,

$$\textsc{Opt} \geq \sum_{j=1}^{\lfloor N/2 \rfloor} \ell_{2j} \qquad \text{and} \qquad \textsc{Opt} \geq \sum_{j=1}^{\lceil N/2 \rceil} \ell_{2j-1}$$

$$\textsc{Opt} \geq \max\left\{ \sum_{j=1}^{\lfloor N/2 \rfloor} \ell_{2j}, \sum_{j=1}^{\lceil N/2 \rceil} \ell_{2j-1} \right\} \geq \frac{1}{2}\left(\sum_{j=1}^{\lfloor N/2 \rfloor} \ell_{2j} + \sum_{j=1}^{\lceil N/2 \rceil} \ell_{2j-1} \right)$$

RMark analysis

Theorem

The $\textsc{RMark}$ algorithm is $2H_k$-competitive in expectation.

For the cost of $\textsc{Opt}$, consider two consecutive phases P_{j-1} and P_j.
At least $k + \ell_j$ distinct pages were requested.
Thus $\textsc{Opt}$ needs to make ℓ_j page faults in these two phases.

$$\text{Either } \underbrace{P_1, P_2}_{\ell_2 \text{ faults}}, \underbrace{P_3, P_4}_{\ell_4 \text{ faults}}, P_5, \ldots \qquad \text{or } P_1, \underbrace{P_2, P_3}_{\ell_3 \text{ faults}}, \underbrace{P_4, P_5}_{\ell_5 \text{ faults}}, \ldots$$

Moreover, $\textsc{Opt}$ causes ℓ_1 page faults in phase P_1 since $\textsc{Opt}$ and $\textsc{Rand}$ start with the same cache contents. Then,

$$\textsc{Opt} \geq \sum_{j=1}^{\lfloor N/2 \rfloor} \ell_{2j} \qquad \text{and} \qquad \textsc{Opt} \geq \sum_{j=1}^{\lceil N/2 \rceil} \ell_{2j-1}$$

$$\textsc{Opt} \geq \max \left\{ \sum_{j=1}^{\lfloor N/2 \rfloor} \ell_{2j}, \sum_{j=1}^{\lceil N/2 \rceil} \ell_{2j-1} \right\} \geq \frac{1}{2} \left(\sum_{j=1}^{\lfloor N/2 \rfloor} \ell_{2j} + \sum_{j=1}^{\lceil N/2 \rceil} \ell_{2j-1} \right) = \sum_{j=1}^{N} \frac{1}{2} \ell_j$$

RMark analysis

Theorem

The $\textsc{RMark}$ algorithm is $2H_k$-competitive in expectation.

Thus we conclude that the expected competitive ratio of the $\textsc{RMark}$ algorithm equals

$$\frac{\mathbb{E}\left[\textsc{Rand}\right]}{\textsc{Opt}} \leq \frac{\sum_{j=1}^{N} \ell_j H_k}{\sum_{j=1}^{N} \frac{1}{2}\ell_j} = \frac{H_k \sum_{j=1}^{N} \ell_j}{\frac{1}{2}\sum_{j=1}^{N} \ell_j} = 2H_k$$

□

Overview

- Deterministic marking algorithms are k-competitive.
- $\text{RM}\textsc{ark}$ algorithms are H_k-competitive in expectation.
- How much did we gain?
- Can we do better?

Overview

- Deterministic marking algorithms are k-competitive.
- $\mathrm{RM{\small ARK}}$ algorithms are H_k-competitive in expectation.
- How much did we gain?
  - $H_k \in \mathcal{O}(\log k)$, thus randomization allows for an exponential improvement!
- Can we do better?

Overview

- Deterministic marking algorithms are k-competitive.
- $\textsc{RMark}$ algorithms are H_k-competitive in expectation.
- How much did we gain?
  - $H_k \in \mathcal{O}(\log k)$, thus randomization allows for an exponential improvement!
- Can we do better?
  - Next lecture we will see that this is the best we can hope for asymptotically.