Online Algorithms

Offline algorithms:

- All inputs are given in advance.
- E.g., given n numbers, the merge-sort algorithm can sort the numbers in $O(n \log n)$ time.

Online algorithms:

- Inputs (requests) are given ***one*** at a time;
  - the algorithm must serve each input before it receives the next one.
- Usually assume no knowledge of the future inputs.
- Why difficult?
  - the offline version is NP-complete; and/or
  - lack of future information.

Example: Ski Rental

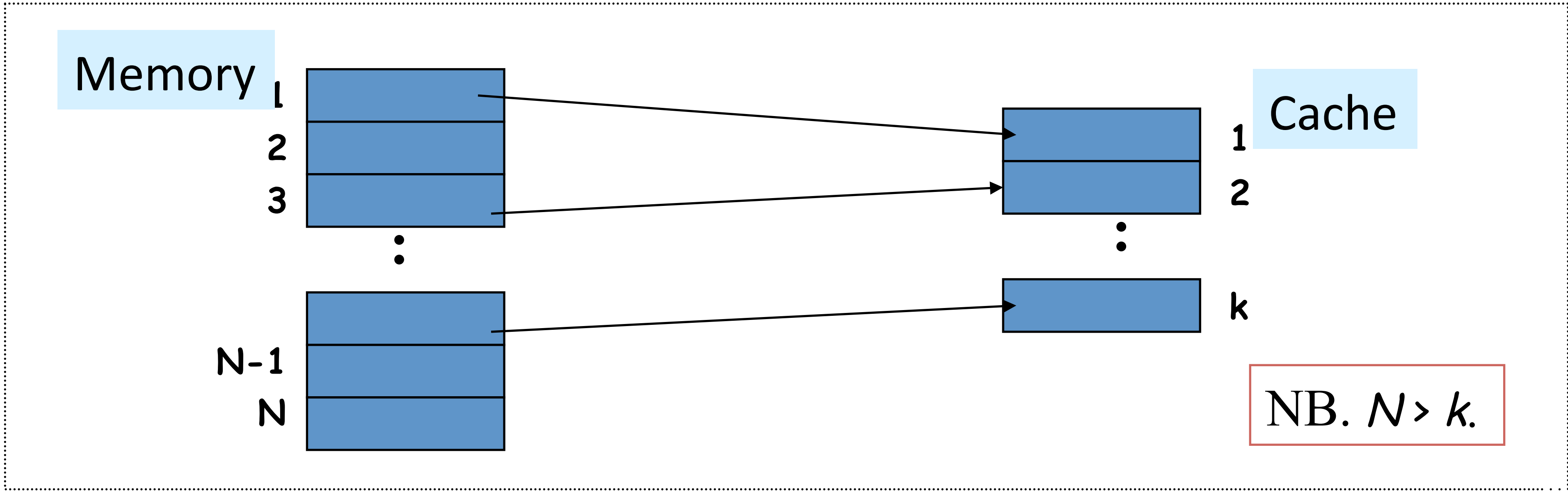
- You are a novice skier and have no idea whether you really like skiing (and when you will give up skiing).
- Each time you go to ski, you need to make a decision of renting or buying a pair of skis.
- Rental: \$100; Buying cost: \$1200
- What is a good strategy?

  (a) Buy the skis the first time you go to ski.

  (b) Always rent the skis

  (c) Buy the skis after you've skied 3/6/12 times.

Example: Paging algorithms



- A memory request is the range [1,N].
- To serve a request, bring the requested memory item into the cache if it is not already there (i.e., a miss).
- A paging algorithm (like LRU, LFU) determines which item in the cache to be evicted so as to make room for the requested item.

Why studying online algorithms?

- For many real-life online problems, we do have some algorithms (heuristics, ideas) for solving them.
  - In many cases, we believe some algorithms are better than others, but often without justification.
- Theory folks always want to explain (or disprove) such phenomena.
- A typical example is the paging problem.

  Can we prove LRU is better than LFU ?

  Or even better, prove LRU is the best possible online paging algorithm.

Least recently used

Least frequently used

Performance measure

- Cost = number of memory access
- Assumption: An algorithm assesses the memory only when there is a miss.
  - Given an algorithm that may access the memory at any time, we can construct another algorithm that accesses the memory only when a miss occurs with no increase in total number of access.
- Cost: Serving a request is associated with a cost.

  E.g., paging algorithms: hit $\rightarrow$ 0; miss $\rightarrow$ 1.

Competitive ratio

- Cost: Serving a request is associated with a cost.
  - E.g., paging algorithms: hit → 0; miss → 1.
- Given a sequence I of requests, we want an online algorithm **A** to minimize the total cost.
- Let **A**(I) denote the total cost that **A** needs to serve I.
- We say that
  - A is (asymptotic) **c**-competitive: $\forall I$, **A**(I) ≤ **c OPT**(I) + **d**, where **OPT**(I) denotes the smallest possible cost to serve I, and **d** is a constant;

    P.S. Roughly speaking, 5-competitive implies a cost at most 5 times of the minimum cost in the worst case.
  - A is optimal: $\forall I$, **A**(I) = **OPT**(I).

c is called the **competitive ratio** or the competitiveness coefficient.

Why competitive analysis ?

- Suppose we simply measure the worst-case cost in terms of the length (denoted *n*) of input sequence. Say, log *n*, *n*, etc.
- Meaningless in many cases. For example, we can show that given any paging algorithm *A*, there exists a sequence I such that *A*(I) = the length of $I = n$.
- What does it mean? All online paging algorithms are equally bad.
- Intuitively, an algorithm shouldn't be regarded as poor if whenever it performs poorly, the optimal offline algorithm also performs poorly.

LRU is better than LFU

- Lower bound: No paging algorithm has competitive ratio smaller than *k*, where *k* is the size of the cache.

  [Sleator and Tarjan 1985]

- LRU is *k*-competitive.

  [Sleator and Tarjan 1985; Goemans 1993]

- LFU cannot achieve a bounded competitive ratio. (That means, there are some input sequences for which LFU performs poorly.)

Outline

1. LRU is k-competitive.
2. LFU cannot achieve a constant competitive ratio.
3. Lower bound: no (deterministic) online algorithm can be better than k-competitive
4. Randomized algorithm: O(ln k)-competitive

LRU is k-competitive

LRU: When eviction in necessary, pick the item whose "recent use" was earliest.

Let I be any sequence of jobs.

Divide I into phases as follows:

Phase 1: *k* distinct items	Phase 2 : *k* distinct items	Phase 3 : *k* distinct items

Phase 1: the longest sequence requesting for at most *k* distinct items.

....

Phase i: the longest sequence following Phase i-1 requesting at most *k* distinct items.

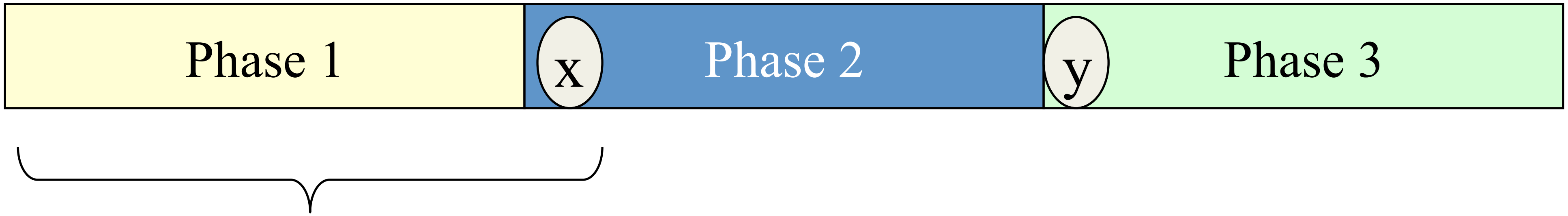
....

Proof Framework

- Optimal offline algorithm: at least one miss in every phase; cost ≥ 1.

- LRU: at most k misses in every phase; cost $\leq k$.

Opt = optimal offline algorithm

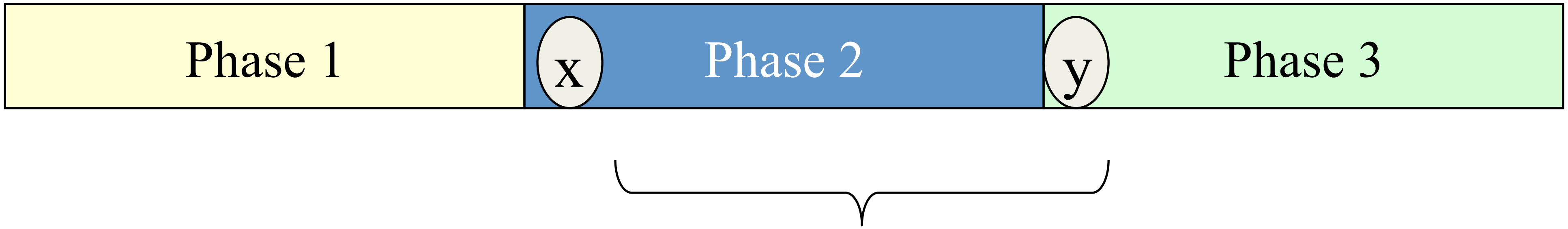


Let us consider the subsequence S comprising
Phase 1 and the first request of Phase 2.

By definition of a phase, S requests k+1 distinct items.

Opt must incur a miss. Why?

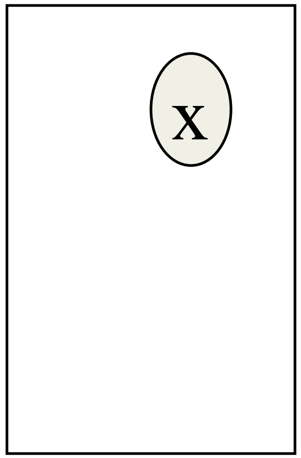
Suppose on the contrary that there isn't a miss.
Then Opt with a cache of k items can serve requests for k+1 distinct items.
A contradiction occurs.



Let us consider the subsequence S comprising Phase 2 (except the first item x) and the first request of Phase 3.

S contains k distinct items ≠ x.

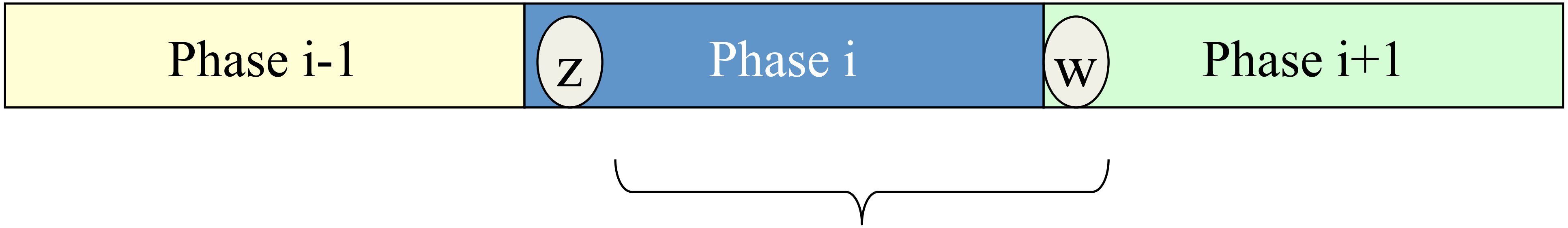
Opt must incur a miss. Why?



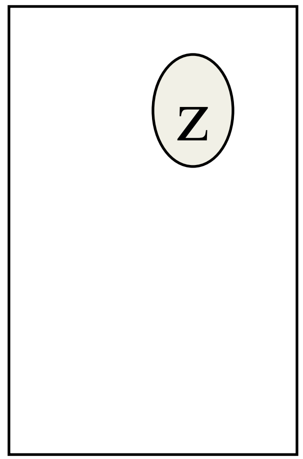
Suppose on the contrary that there isn't a miss.

Then Opt with a cache containing x and k-1 items can serve requests for k distinct items not equal to x.

A contradiction occurs.



Let us consider the subsequence S comprising Phase i (except the first item z) and the first request of Phase i+1.
S contains k distinct items not equal to z.
Opt must incur a miss.



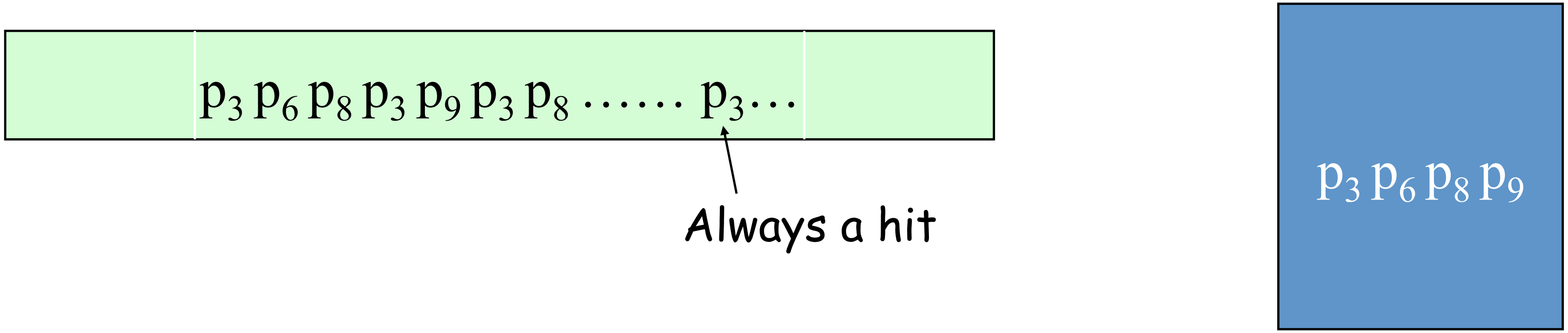
Conclusion on opt

Except the last phase, we can charge at least one miss to every phase.

Total cost $\geq$ number of phases – 1

LRU causes at most k misses within a phase.

Lemma 1: Within a phase, once an item p is brought into the cache, p won't be evicted by LRU.



Corollary: Within a phase, each of the k distinct items can cause at most one miss, which can only occur at its first occurrence.

In other words, the number of misses is at most k.

Proof of Lemma 1 – by contradiction

Suppose on the contrary that within a phase, an item p is evicted after it has been requested.

| …p ….. p…..p' |

I_x

p *is evicted here.*

Assume this eviction occurs at the request I_y.
Denote I_x be the last request for p before I_y. I.e., x < y.

- **Immediately after** I_x : p is the most recently used item.
- **Just before** I_y : p is the least recently used (i.e., k-th most recently used) item.
- To make p the least recently used item and then evicts it, how many distinct items other than p must be requested from I_{x+1} to I_y ?

From I_{x+1} to I_y

At the point the 1st item other than p is requested, p is still the most recently used.

At the point the 2nd distinct item other than p is requested, p is the 2nd-most recently used.

At the point the 3rd distinct item…., p is the 3rd most recently used.

…

At the point the $(k\text{-}1)^{st}$ …., p is the $(k\text{-}1)^{st}$ most recently used. Thus, if eviction is required here, p is not yet the candidate according LRU.

In other words, we need at least k distinct items other than p to cause an eviction of p .

This contradicts that a phase contains distinct items.

LRU is k-competitive

Consider any sequence I.
Suppose I is partitioned into m phases.

$\mathrm{LRU}(I) \leq km$

$\mathrm{Opt}(I) \geq m - 1$

Therefore, $\mathrm{LRU}(I) \leq km \leq k(\mathrm{Opt}(I) + 1) \leq k\,\mathrm{Opt}(I) + k$

Outline

1. LRU is *k*-competitive.
2. LFU cannot achieve a constant competitive ratio.
3. Lower bound: no (deterministic) online algorithm can be better than k-competitive
4. Randomized algorithm: O(ln k)-competitive

LFU is not competitive

LFU: When eviction is necessary, pick the item p in the cache that has the smallest number of access so far.

Lemma. For any positive integer c, there exists I such that LFU(I) ≥ c Opt(I).

LFU is not competitive

LFU: When eviction is necessary, pick the item p in the cache that has the smallest number of access so far.

Lemma. For any positive integer c, there exists I such that $\text{LFU}(I) \geq c\,\text{Opt}(I)$.

I: $(p_1)^c (p_2)^c (p_3)^c \ldots (p_{k-1})^c (p_k p_{k+1})^{c-1}$

Opt: 1 miss

LFU is not competitive

LFU: When eviction is necessary, pick the item p in the cache that has the smallest number of access so far.

Lemma. For any positive integer c, there exists I such that LFU(I) ≥ c Opt(I).

I: $(p_1)^c (p_2)^c (p_3)^c \ldots. (p_{k-1})^c (p_k p_{k+1})^{c-1}$

Opt: 1 miss

LFU: 2c –3 miss

$(p_1)^c (p_2)^c (p_3)^c \ldots. (p_{k-1})^c\ p_k\, \mathbf{p_{k+1} p_k p_{k+1} p_k p_{k+1} \ldots.. p_k p_{k+1}}$

Outline

1. LRU is *k*-competitive.
2. LFU cannot achieve a constant competitive ratio.
3. Lower bound: no (deterministic) online algorithm can be better than k-competitive
4. Randomized algorithm: O(ln k)-competitive

Lower bound: Competitive ratio $\geq k$

Let A be any online paging algorithm.

Claim: There exists a sequence $\mathcal{I}$ of n requests such that
$A(\mathcal{I}) = |\mathcal{I}| = n$.

Proof: $\mathcal{I}$ comprises memory requests the range $[1..k+1]$.

At any time the cache contains at most k items.

The next request is simply the item not in the cache.

Therefore, $A(\mathcal{I}) = n$.

Framework

Let A be any online paging algorithm.

Fact. There exists a sequence $\mathcal{I}$ of n requests such that $A(\mathcal{I}) = n$.

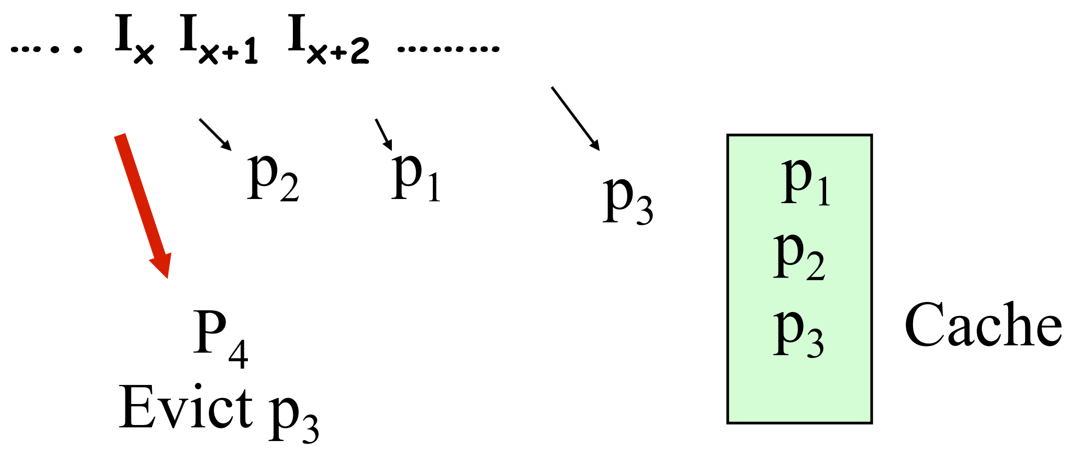
How to argue that competitive ratio of $A \geq k$?

We need to show that the optimal offline algorithm O when working on $\mathcal{I}$ incurs a cost $\leq n/k$.

Then competitive ratio of $A \geq A(\mathcal{I})/Opt(\mathcal{I}) \geq n/\ (n/k) = k$.

Offline algorithm: Opt(I) ≤ LNU(I)

Consider the following offline algorithm LNU (latest next use; LFD):
Initially, the cache is loaded with any k items.
When eviction is necessary, choose the item in the cache that will not be used for the longest period of time.



- LNU is an offline algorithm. *Opt(I)* ≤ LNU*(I)* for all *I*.

Lemma. Consider any request sequence I in the range [1,k+1]. Between every two misses of LNU, there must be at least k-1 hits.

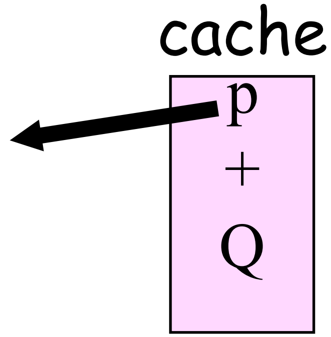
× √√√√√√√√ × √√√√√√ × √√√√√√ ....

$\geq$ k

Therefore, one miss is followed by at least k-1 hits before any miss, and $LNU(I) \leq \lceil n / k \rceil$.

Corollary: For any sequence I of memory requests in the range [1,k+1], $LNU(I) \leq \lceil n / k \rceil$

A miss of LNU



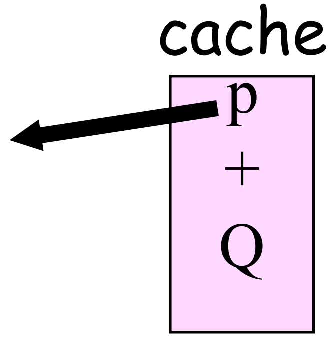
With respect to a request sequence I, suppose LNU incurs a miss at the x-th request (I_x = p') and evicts item p from the cache.

Let Q be the set of the other k-1 items currently in the cache.

Immediately after I_x, what request will cause a hit?

What request will cause a miss?

A miss of LNU



With respect to a request sequence I, suppose LNU incurs a miss at the x-th request (I_x = p') and evicts item p from the cache.

Let Q be the set of the other k-1 items currently in the cache.

Immediately after I_x, what request will cause a hit?

Q U {p'} = {1, .., k+1} – {p}

What request will cause a miss? p

evicts **p** requests for **p**

$I_x \quad I_{x+1} \quad I_{x+2} \quad \ldots\ldots\ldots \quad I_y$

Two cases to consider:

- After I_x, **p** is not requested again and LNU will incur a zero cost.
- After I_x, **p** is requested at I_y.

evicts p   requests for p

$I_x \quad I_{x+1} \quad I_{x+2} \quad \cdots\cdots\cdots \quad I_y$

Why item p is chosen by LNU for eviction when serving I_x ?

Because for each item q in Q, q will be requested before p.

In other words, **after** I_x and **before** I_y, each of the (k-1) items in Q is requested at least once, and there are at least k-1 hits.

Thus, a miss must be followed by at least k-1 hits, and LNU(I) ≤ n / k.

Conclusion

- For any algorithm $\boldsymbol{A}$, there exists a sequence I of n requests such that $\boldsymbol{A}(I) = n$.
- $\mathrm{Opt}(I) \le \mathrm{LNU}(I) \le n / k$.
- The competitive ratio of $A \ge \boldsymbol{A}(I) / \mathrm{Opt}(I) \ge k$.

Outline

1. LRU is *k*-competitive.
2. LFU cannot achieve a constant competitive ratio.
3. Lower bound: no (deterministic) online algorithm can be better than k-competitive
4. Randomized algorithm: O(ln k)-competitive

Can we do better?

Randomized online algorithms: In the worst case (input), can the **expected** value of the competitive ratio < k?

- Evict an item chosen randomly: k-competitive

[Raghavan & Snir 1989]

- A random marker algorithm: **2 H_k**-competitive, where $H_k = 1 + 1/2 + 1/3 + \ldots + 1/k \leq 1 + \ln k$

[Fiat, Karp, Luby, McGeoch, Sleator, Young 1991]
[Achlipptas, Chrobak, and Noga 1996]

- Lower bound: No randomized paging algorithm has competitive ratio smaller than **H_k**.

Restricting the inputs to model the locality of reference:

- LRU is better than FIFO …

[Borodin, Irani, Raghavan, Schieber 95] [Chrobak & Noga 98]…

Algorithm Mark

Initially, all items in the cache, if any, are considered to be unmarked.

When an item in the cache is requested, mark that item.

Whenever eviction is necessary, choose an unmarked item in the cache randomly.

- The new item to be brought into the cache will be marked.
- If all items are marked, unmark ALL of them first.

Remark: Most of the time, recently used pages are given priority to stay in the cache. But occasionally, ignore the history.

Analysis

For any input sequence I, partition I into phases as before (precisely, Phase i captures the longest sequence following Phase i-1 involving at most *k* distinct items).

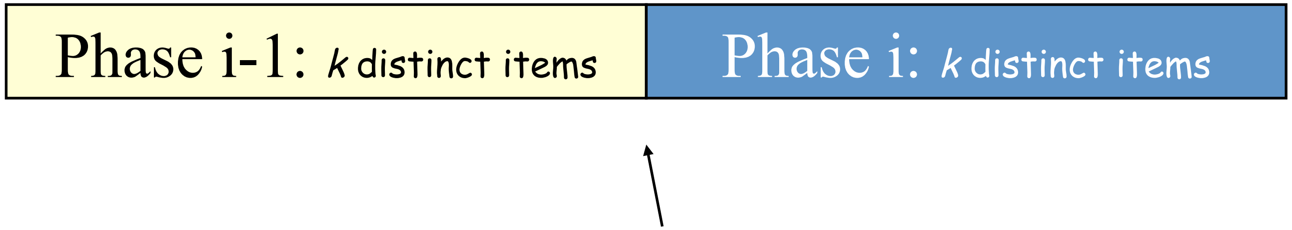
Phase 1: *k* distinct items	Phase 2: *k* distinct items	Phase 3: *k* distinct items

The k distinct items requested in Phase 1 must all be marked and kept in the cache until the end of Phase 1.

Unmark all.

The first item of Phase 2 would trigger unmark-all and then an eviction.

At Phase i



The k distinct items requested in Phase i-1 must be in the cache of Mark.
Call these items old items to Phase i.

Phase i requests for *k* distinct items. Suppose there are $\boldsymbol{n_i}$ new (not old) items and $\boldsymbol{k - n_i}$ old items.

NB. $\boldsymbol{n_1} = k$; for $i > 1$, $\boldsymbol{n_i} \geq 1$

Phase i-1: k distinct items	Phase i: k distinct items

Phase i requests n_i new items and $k - n_i$ old items.

How many misses for Opt (optimal offline algorithm)? At least ..

In Phases i-1 and i, there are $k + n_i$ distinct items requested.
Number of miss $\geq n_i$

Opt's total cost $\geq n_2 + n_4 + n_6 + \ldots$
Alternatively, total cost $\geq n_1 + n_3 + n_5 + \ldots$

In conclusion, Opt's total cost $\geq \frac{1}{2}\,[n_1 + n_2 + n_3 + n_4 + \ldots] = \frac{1}{2} \sum n_i$

Phase i-1: *k* distinct items	Phase i: *k* distinct items

Phase i requests n_i new items and $k - n_i$ old items.

How many misses for Mark in Phase i ? At most …

Misses due to new items in Phase i: exactly n_i

In the worst case, new items should be requested before old items (so that each old item has a better chance to cause a miss).

Miss due to old items: The exact number depends on the random choice of items for eviction (to make room for the new items).

Phase i-1: *k* distinct items | Phase i: *k* distinct items

Phase i requests n_i new items and $k - n_i$ old items.

	Probability in cache	Miss probability
1st old item requested	$(k - n_i) / k$	n_i / k
2nd old item requested	$(k - n_i - 1) / (k-1)$	$n_i / (k-1)$
…..		
$(k - n_i)$-th old item	$1 / (n_i + 1)$	$n_i / (n_i + 1)$

Remark. When the j-th old item is requested,

- all the ni new items and the first (j – 1) old items must be in the cache;
- the j-th old item as well as every other old item have the same probability to be in the cache. I.e., (k – (j-1)) items flight for (k-ni-(j-1) positions.

Expected # of miss in phase i due to old items:

$1\times(n_i\ /\ k)\ +1\times(n_i\ /\ k-1)+\ldots+1\times(n_i\ /\ n_i+1)$

$= n_i\,[\,1\ /\ k\ +1\ /\ k-1+\ldots+1\ /\ n_i+1\,]$

$= n_i\,[\,H_k - H_{ni}\,]$

Expected # of miss in phase i = $n_i + n_i[\,H_k - H_{ni}\,] \le n_i H_k$

Expected # of miss in all phases = $\sum n_i H_k = H_k \sum n_i$

Conclusion

Opt's total cost: $\ge \frac{1}{2} \sum n_i$

Mark (expected cost) : $\le H_k \sum n_i$

Competitive ratio (expected value): $2\,H_k$

Reference

- Online computation & competitive analysis, Cambridge, 1998, Borodin & El-Yaniv