Lecture 18: Offline alg Part 2: Paging

Today we'll do one long example that illustrates many concepts:
competitive ratio, determinism vs. randomized alg, amortized
analysis...

"Paging" is a funny old-fashioned word; it's all about...

Cache

Disk vs. memory, mem vs. cache, CPU vs. motherboard
... extremely important for practical algo. run time.

Simple model:

k fast-memory

cache slots

N data/disk/slow-mem items/pages

Model: CPU will access a seq. \( I = r_1, r_2, r_3, \ldots \)

request

If item in cache: \( \in \) [free]

Else: incur cost 1 \( \in \) "cache miss" / "page fault"

\( \in \) "must move item into cache"

\( \in \) "must "evict" one item from cache"

\( \in \) "maybe at "the beginning of time" cache is not

full yet, but we'll ignore, assuming cache starts full"

Alg = Eviction strategy. [Ideas?]

"Offline Optimal": If you psychically knew

\( I \), optimal alg is...

\( \in \) Evict page whose next req. is

farthest in future

\( \in \) e.g.: \( k=3, N=4, \) requests:

\( I = 1, 2, 3, 2, 4, 3, 4, 1, 2, 3, 4 \ldots \)
Takes a teeny bit of thought to see this greedy strat is indeed offline optimal. But it is.

On e.g.: 1 2 3 2 4 3 4 1 2 x → kick out 2 x → kick out 1 3 4

But we don't know future, so offline alg?!

"LRU" (Least Recently Used): evict page that's L.R.U.

On e.g.: 1, 2, 3, 2, 4 x → evict 1 3 4 1 x → evict 2 x → evict 3 x → evict 4

Hmm... kinda bad if next req. is whatever was just evicted.

"Adversary's" worst I for LRU: 1, 2, 3, 4, 1, 2, 3, 4, 1, ...

Offline opt has a "1 3 1" (in k=3 case; always just uses \( N = k+1 \)).

In general k.

\[ \Rightarrow \text{C.R. for LRU no better than } k. \]

Better alg?!

"FIFO": evict page that's been in cache longest

ex: [bad example I implies] [C.R. no better than k is.]

Fact: [No det. alg. can have C.R. < k.]

probs: Given Alg. Let I be a long seq. that always requests the page Alg just evicted. Need to know Alg to design I.

\( \Rightarrow \) miss every 1 reqs

Claim: for any I with \( N = k+1 \), offline opt. only misses once every \( \geq k \) reqs.

because when item i evicted, all other k-1 items will subsequently be req'd earlier: \( \Rightarrow k-1 \) successes per miss.
There's a twist you can invent to make this situation less sad... I

LRU on input I
with cache size k
might be k times worse:

vs. Offline opt. on I
with cache size k

Force it to have cache size \( \frac{k}{2} \).

Now, "C.R." is \( \leq 2 \) 

? I kinda cheating, not apples to apples.
But... "Resource Augmentation": maybe okay to imagine you can double your cache... II

Proof sketch: Given any seq. \( I = 1, 3, 1, 1, \{4, 7, 2, 2, 3, 6, 5\} \ldots \)

divide into consecutive "phases":
phase = maximal sequence of k distinct items

Fact 1: LRU (\& FIFO) have cost \( \leq k \) in each phase.

\[ \text{evident from def. of phase} \]

Fact 2: Offline opt. with cache of size \( \frac{k}{2} \) must have \( \geq \frac{k}{2} \) misses per phase.

\[ \Rightarrow \text{"C.R." of} \ \leq \ \frac{k}{k/2} = 2 \]

ex: For LRU with k vs. Offline opt. with h \( \leq k \),

"C.R." is \( \leq \frac{k}{k-h+1} \)

\[ \Rightarrow \text{so k for h=k, but} \]

\[ \leq 10 \text{ for h=0.9k, e.g.} \]

Back to usual apples to apples again.
The sad thing about deterministic algos is "adversary" at beginning always knows what's in your cache, knows what you just evicted, can always choose \( I \) so that next req. = last evict.
How can we make it so adversary doesn't "know" what you just evicted? Randomness!
Random model: Adversary knows your randomized eviction policy
"doesn't see" random coin flips/choices, can't see your cache.
"It's like the O.S.'s alg's spec is published, but cache is in "private memory"!"
May as well assume Adversary fixes all $I$ in advance.
But now you can't change your alg! $\mathbb{E}[\text{cost of Alg on } I] = \max_I$ 
Offline Opt $(I)$
Motivates defn: $C.R. = \max_I \mathbb{E}[\text{cost of Alg on } I] - \text{you next, randomized}$

Thm: $\exists$ randomized alg., "Marking," with $C.R. = O(\log k)$ !

$\exists$ alg., even randomized, has $C.R. = \Omega(\log k)$

Won't show $\exists$. We'll sketch proof of $\exists$. !
Marking: Initialize cache to $1, 2, \ldots, k$, & all pages "unmarked"
- Randomize cache to $1, 2, \ldots, k$.
- When $i$ requested: if in cache, "mark" it
- Else, evict a random unmarked page:
  - except if all pages marked ... unmark-all first
Then mark $i$ when it's brought to cache.

We'll show $C.R. = O(\log k)$ assuming $N = k + 1$.
For $N > k + 1$, proof is $\approx 25\%$ harder. !

Observation: Unmark-all's happen exactly at "phase" boundaries.

<table>
<thead>
<tr>
<th>$3$</th>
<th>$1$</th>
<th>$1$</th>
<th>$2$</th>
<th>$1$</th>
<th>$4$</th>
</tr>
</thead>
</table>

unmark all $\uparrow$ in phase, # marked = # diff. items seen so far in phase.
Q: What is \( E[\text{cost of a phase}] \)?

So only newly-requested items were not in cache, others free.

WLOG, say prev. phase ended with \( 1, 2, \ldots, k \) in cache.

\( k = 4 \), \( N = 5 \) example

Let's make a diagram showing all \( N = k + 1 \) items, annotated with probability.

Start of phase:

\[
\begin{array}{cccccc}
1 & 1 & 1 & 1 & 1 & 0 \\
& 1 & 2 & 3 & 4 & 5
\end{array}
\]

// \( 1, 2, 3, 4 \) are in cache
with prob \( 1 \), \( 5 \) in cache
with prob \( 0 \), no marks

Because we're starting
a new phase, \( 5 \) must
have just been requested.

\[
\begin{array}{cccc}
\frac{3}{4} & \frac{3}{4} & \frac{3}{4} & \frac{3}{4} & 1 \\
& 1 & 2 & 3 & 4 & 5
\end{array}
\]

Perhaps \( 5 \) now req'd many
times. All free so let's
move to next req.

\[
\begin{array}{cccc}
\frac{1}{3} & \frac{1}{3} & \frac{1}{3} & 1 \\
& \frac{1}{3} & \frac{1}{3} & \frac{1}{3} & 1
\end{array}
\]

\[
\begin{array}{cccc}
1 & 1 & \frac{1}{2} & 1 \\
& \frac{1}{3}
\end{array}
\]

\[
\begin{array}{cccc}
\frac{1}{2} & 1 & 0 & 1 & 1 \\
& \frac{1}{2}
\end{array}
\]

Some more cache hits, then phase ends.

\[ \uparrow \] requests \[ \uparrow \]

\# [cost]
In general, $E[\text{cost of phase}]=1+\frac{1}{k}+\frac{1}{k-1}+\cdots+\frac{1}{2}$.

Claim: Opt offline alg pays 1 per phase.

Test if $x$ exists. 1. x exists?
2. x does it exist? 3.
4. x does it exist? 4.

4 1 2 3 5 2 1 1 2 3 4 4 1 2 5 3 1 2 1 1 4 5 4 3 2 1