Online Algorithms

- In some problems, data is given in an online fashion & we must make decisions without knowing full data.

Example: Rent_or_Buy (ski rental)

- We want to ski this winter, we don't know how many days we will go.
- Rent ski for a day: cost = R

  Buy ski/pass and use unbounded: cost = B
- Clearly if we knew we knew # of days $> \frac{B}{R}$ we should buy; o.w. we should rent.
- What is the best algorithm?

Definition (competitiveness): if an algorithm Alg given instance I has solution with value Alg(I) and opt(I) is the best possible (offline) solution then

$$\max_I \frac{Alg(I)}{Opt(I)}$$

↓ minimization

is called competitive ratio of Alg.

Strong Competitive: Assume the same as above, then we call r the strong competitive ratio if

$$Alg(I) \leq r \cdot Opt(I) + c$$ → constant.

For simplicity assume $R=1$.

Deterministic rent_or_buy: suppose we decide to rent until a total of $B-1$ and then decide to buy on day B.

— doesn't seem smart but:

lemma: This deterministic algorithm has competive ratio $2-\frac{R}{B}$ and is best possible for any deterministic algorithm.

proof: Suppose j is the actual # of days we ski.

- Case $j \leq B$: we pay the same as opt.
- Case $j > B$: we pay $2B-1$ while opt $= B$.

$\longrightarrow$ competive $\leq \frac{2B-1}{B} = 2-\frac{1}{B}$

To see tightness: Clearly any deterministic must decide on some day to buy (hence rents until a day)

- Suppose Alg_i is the algorithm that rents for $i-1$ days & then buys on day i.

For any i, if:

- $i \geq B$: let the # of days be B.

then Alg_i pays $i-1+B \longrightarrow$ ratio $\geq 2-\frac{1}{B}$

then Alg_i pays $i-1+B \longrightarrow$ ratio $\geq 2-\frac{1}{B}$

- $i=1$: let $j=1$ then Alg_i pays B and ratio $=B$
- $1<i<B$: let # of days $j=\left\lfloor \frac{i-1+B}{2-\frac{1}{B}} \right\rfloor \geq 1$

$\longrightarrow$ ratio $\geq 2-\frac{1}{B}$

Can randomization help?

Yes! It can be shown that randomized strategy gives competitive ratio $\frac{e}{e-1}$ (exercise!).

Example: Paging (cache)

- Tradeoff between large (slow) memory & fast/small memory (cache)
- Whenever we need data from memory $\longrightarrow$ needs to go to cache. If a page is already in cache: cache hit ; otherwise chache miss
- If cache is full we need to evict some data to make room.
- Cache miss: slows the system; minimize the # of cache misses.
- n: total # of memory space

k: size of cache (# of pages we can keep)

e.g. $n=6$, $k=3$

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

Page miss

- Optimal algorithm if we know the sequence?
  Evict the page for which the next request happes the latest in the future (among all the pages currently in the cache.
- We don't know the sequence of requests in advance.

Deterministic Caching

Many deterministic online algorithms

- LRU (Least Recently Used): evict the page that hasn't been used the longest time
- FIFO (First In First Out): Cache is like a queue
- LFU (Least Frequently Used): evict the least frequently used one
- LIFO (Last In First Out): cache works like a stack.
- We will see that LRU and FIFO have comp. ratio k while the otter two have unbounded ratio.

Bad example for LFU & LIFO: Suppose we load pages 1..k initially and consider the sequence: $k+1, k, k+1, k, k+1, \ldots$

- We evict k, then $k+1$, then k, ...
- We have a cache miss for every request.
- optimum alg would evict 1 instead and no more miss.

Lemma: No deterministic algorithm can have better than k competitive ratio.

proof: Suppose $n > k$ and for any deterministic alg the adversary can request a page not in cache in each step. $\longrightarrow$ every request is a miss; n

- optimum algorithm can always evict the page that will be requested furtherest in future.
- So when a miss occurs at least k other requests must have been in the cache

  $\longrightarrow$ optimum misses one every k or $\frac{n}{k}$

  $\longrightarrow$ competitive ratio of any deterministic $\geq k$.

Lemma: LRU has competetive ratio k.

proof: Exercise.

Deterministic 1-Bit LRU (Marking Algorithm)

- There is a variant of LRU: maintain a single bit for each page
- Marked / unmarke page: bit is 0/1

  - Initially all pages are unmarked in phase.
  - when a page is requested:
    - * if the page is in the cache, mark it.
    - * o.w.
      - if ∃ an unmarked then evict an arbitrary unmarked, bring in the request & mark it
      - else, umark all pages and start next phase

Lemma: The Marking algorithm is k-competitive.

proof: Consider some i'th phase.

- We have k distinct pages access in a phase. so we do at most k page faults per phase. (as each page fault results in a marked page).
- When a phase ends we must have had k+1 distinct page requests. clearly opt must have at least one page fault.

⟶ 1-bit LRU has competitive ratio k. ∎

Randomized 1-bit LRU (Marking)

– Suppose instead of evicting an arbitrary unmarked page, we choose one u.r.

Lemma: The randomized marking is $O(\log k)$-competitive.

proof: We show a competitiveness of $2H_k$

$$H_k = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{k} \approx \ln k$$

– let S_i = set of pages in cache at start of phase i

$$\Delta_i = |S_{i+1} \setminus S_i|$$

↳ # of new pages added during phase i

– We show $E[\#\text{ of misses in phase } i] \leq \Delta_i \cdot (H_k + 1)$

– Suppose R_i are the distinct requests in phase i

{ clean request: request page not in S_i

stale request: o.w.

– A cache miss is either clean or stale

– # of missed clean requests: $\leq \Delta_i$

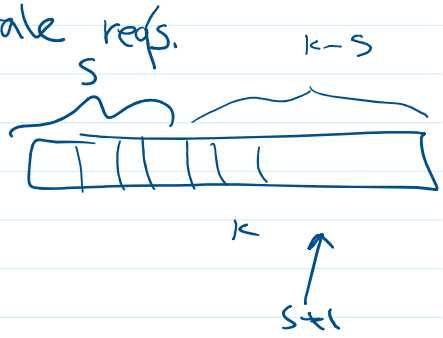
each clean requests brings in a new request and adds to cache & mark it ⟶ in S_{i+1}

- to bound # of stale cache misses:

suppose there are $\leq c$ clean requests so far
$\leq s$ stale

and consider $(s+1)$'th stale request.

the probability that this causes a miss is $\leq \frac{c}{k-s}$ since we have evicted $\leq$ random pages out of the remaining $k-s$ stale reqs.



since $c \leq \Delta_i$:

$$\sum_{s=0}^{k-1} \frac{c}{k-s} \leq \Delta_i \sum_{s=0}^{k-1} \frac{1}{k-s} = \Delta_i H_k$$

So $\mathbb{E}[\#$ of cache misses in phase $i] =$

$$\Delta_i (1 + H_k) \qquad (1)$$

- Now we show for opt # of cache misses $\geq \frac{1}{2}\sum \Delta_i$
- let n_i : # of cache misses of opt in phase i
- Note that there are $k + \Delta_i$ distinct pages in phases $i-1$ & i. So at least Δ_i cause a miss. $\quad n_{i-1} + n_i \geq \Delta_i$

$$\rightarrow \text{opt} \geq \frac{1}{2}\sum \Delta_i \qquad (2)$$

Combining (1) & (2)

$$\mathbb{E}[\#\text{ of cache misses}] \leq 2(H_k + 1) \cdot \text{opt}.$$

Generalizing paging: k-Server Problem

- The following generalization of paging has received a lot of attention, known as the k-server problem.

- Suppose we are given a metric space (V, d) with distance func. $d: V \times V \longrightarrow \mathbb{R}^+$ that satisfies triangle inequality. Say $|V| = m$ and we have k servers that are located at different points of V.
- At each time t we get a request $a_t \in V$ and we need a server at point a_t to perform the task
- if there is not a server there we have to move a server to a_t and if this server moves from point x we have to pay cost $d(x, a_t)$.

- Goal: Find a strategy to place the servers to serve all the requests while minimizing server moving cost.

Paging as special case: we can model paging as k-server: for all points $x, y \in V$, $x \neq y$ set $d(x,y) = 1$

There is a deterministic $(2k-1)$-competitive alg for k-server. It is conjectured that there is a k-comp. deterministic alg. for k-server.

- For a long time a poly-logarithmic-comp. rand algorithm was open until a work by Bansal/Buchbinder/Madry/Naor in 2011.

Online Decision Making

Want to interview people for a secretary position.

- We have a set of n candidates & want to hire the best one.
- We can interview one-by-one & we have to make a decision (to hire or pass and continue) after each interview.

  these decisions are irrevocable!

The Secretary Problem

- We have a sequence of n items/agents arrive $a_1, a_2, \ldots, a_n$ with a total ordering between them ($\forall i,j$: can say if a_i is better than a_j)
- these items arrive in a uniformly random order
- We can only compare an item with previously seen elements
- We can hire (choose) the current item & stop or pass (continue) to see next.

Goal: Find an algorithm/strategy that maximizes the probability of choosing the best item.

intuition:

- We choose early on: chance of the best is low

{ We choose early on : chance of the best is low
{ we wait too long : miss on the good ones

- how about:
  - pass on first $\frac{n}{2}$
  - then pick first that is better than all seen

lemma: this strategy finds the best with prob $\geq \frac{1}{4}$

proof: consider the following two events:

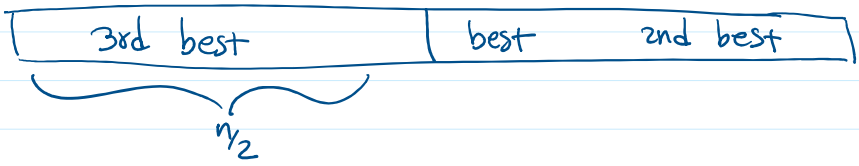
* E_1: the best element is in the second half
* E_2: the 2nd best element is in the first half

- Note that in $E_1 \cap E_2$ the algorithm finds the best element.

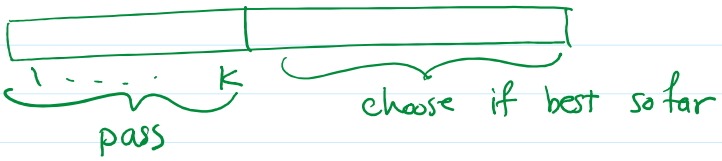
$$Pr[win] \geq Pr[E_1 \wedge E_2] = Pr[E_1] \cdot Pr[E_2 | E_1]$$

$$= \frac{1}{2} \cdot \frac{n/2}{n-1} = \frac{1}{4} + o(1)$$

- Can we improve this? note that the analysis is loose: we sill find the best when



- Pick $k \in \{1, \ldots, n\}$ then pass on the first k and then choose one that is better than all seen so far.



lemma: This modified algorithm has probability $(1-o(1))\frac{k}{n}\ln\frac{n}{k}$ of picking the best.

proof: For any $i>k$ we compute the probability of picking the best when it is a_i.

– For this we need:

both $\begin{cases} E_1: a_i \text{ is the best} \\ E_2: \text{the best among } a_1 \dots a_{i-1} \text{ has rank} \le k \end{cases}$

$$\Pr[\text{win}] = \sum_{i=k+1}^{n} \Pr[a_i \text{ is the best and alg. picks it}]$$

$$= \sum_{i=k+1}^{n} \Pr[E_1 \wedge E_2]$$

$$= \sum_{i=k+1}^{n} \Pr[E_1]\cdot\Pr[E_2]$$

$$= \sum_{i=k+1}^{n} \frac{1}{n}\cdot\frac{k}{i-1}$$

$$= \frac{k}{n}\sum_{i=k+1}^{n}\frac{1}{i-1}$$

$$= \frac{k}{n}\left(H_{n-1} - H_{k-1}\right)$$

$$\simeq \frac{k}{n}\left(\ln(n-1) - \ln(k-1)\right)$$

$$= \frac{k}{n}\ln\frac{n-1}{k-1} = (1-o(1))\frac{k}{n}\ln\frac{n}{k}$$ ∎

Since we want to maximize this, we solve for best

Since we want to maximize this, we solve for best k:

$$f(x) = \frac{x}{n} \ln \frac{n}{x}$$

Take derivative (1st & 2nd)

$$f'(x) = \frac{1}{n}\left(\ln \frac{n}{x} - 1\right) \qquad f''(x) = \frac{1}{n}\left(-\frac{1}{x}\right)$$

In $(0,1)$, f is concave & the maximum is when

$$f'(x) = 0 \longrightarrow \ln\left(\frac{n}{x}\right) = 1 \longrightarrow x = \frac{n}{e}$$

So the best value for k is $\frac{n}{e}$

$$\Pr\left[\text{win with } k = \frac{n}{e}\right] = (1 - o(1))\frac{k}{n}\ln\frac{n}{k} \simeq \frac{1}{e}$$

Note: It can be shown this is the best strategy.