CMPUT 498/501: Advanced Algorithms **Fall 2025**

Lecture 23 (Dec 1, 2025): Online algorithms, Paging, Secretary problem

Lecturer: Mohammad R. Salavatipour *Scribe: Lyndon Hallett, Haoxin Sang*

23.1 Online Algorithms

In some problems, data is given in an online fashion and we must make decisions without knowing the full data. These decisions are irrevocable.

23.1.1 Example: Rent-or-Buy (Ski Rental)

We want to ski this winter, but we don't know how many days we will go.

- Rent Ski for a day: cost $= R$
- Buy Ski/Pass (use unbounded): cost $= B$

Clearly, if we knew the # of days $> \frac{B}{R}$, we should buy; otherwise, we should rent.

Question: What is the best algorithm?

Definition 1 (Competitiveness) *An algorithm Alg is given. If instance I has a solution with value $Alg(I)$ and $Opt(I)$ is the best possible (offline) solution, then*

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

is called the competitive ratio of Alg (for minimization).

Definition 2 (Strong Competitiveness) *Assume the same as above, then we call r the strong competitive ratio if*

$$Alg(I) \leq r \cdot Opt(I) + C$$

where C is a constant.

A deterministic algorithm for Rent-or-Buy

For simplicity, assume $R = 1$.

Algorithm (deterministic rent-or-buy): Suppose we decide to rent until a total of $B - 1$ days, and then decide to buy on day B.

This doesn't seem smart, but:

Lemma 1 *This deterministic algorithm has competitive ratio $2 - \frac{1}{B}$ and is the 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 $B-1$ (renting) $+B$ (buying) $= 2B-1$. While $Opt = B$.

$$\text{Competitive Ratio} \leq \frac{2B-1}{B} = 2 - \frac{1}{B}$$

Tightness: Clearly any deterministic algorithm must decide some day to buy (hence rents until a day). Suppose Alg_i is the algorithm that rents for i days and then buys on day i. For any i:

- If $i = 1$: Let the # of days be 1. Then Alg_i pays B. Ratio $= B$.
- If $i \geq B$: Let the # of days be B. Then Alg_i pays $i - 1 + B$. Ratio $= \frac{i-1+B}{B} \geq 2 - \frac{1}{B}$.
- If $1 < i < B$: Let the # of days be $\left\lfloor \frac{i-1+B}{2-1/B} \right\rfloor \geq 1$. Then Alg_i pays $i - 1 + B$. Ratio $= \frac{i-1+B}{B} \geq 2 - \frac{1}{B}$.

■

Can randomization help? Yes! It can be shown that a randomized strategy gives a ratio of $\frac{e}{e-1}$ (Exercise!).

23.1.2 Example: Paging (Caching)

Tradeoff between large (slow) memory and fast/small memory (Cache).

- Whenever we need data from memory, we go to Cache.
- If a page is already in Cache: **Cache hit**; otherwise: **Cache miss**.
- If cache is full, we need to **evict** some data to make room.
- Cache miss slows the system; minimize the # of cache misses.

Parameters:

- n: total # of memory space.
- k: size of cache (# of pages we can keep).

Example: $n = 6, k = 3$. Sequence: $4, 3, 3, 2, 4, 1(\text{miss}), 1, 5(\text{miss}), 3, 2 \ldots$

23.1.2.1 Optimal Algorithm (Offline)

If we know the sequence? Evict the page for which the next request happens the latest in the future (among all pages currently in the Cache).

23.1.2.2 Deterministic Caching

We don't know the sequence of requests in advance. 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 competitive ratio k, while the other two have unbounded ratio.

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

$$1, 2, \dots, k, k+1, k, k+1, k, k+1, \dots$$

After the first k requests, we evict k, then $k+1$, then $k \dots$ We have a cache miss for every request. Optimum algorithm would evict 1 instead and have no more misses.

Lemma 2 *No deterministic algorithm can have better than k competitive ratio.*

Proof. Suppose $n > k$. For any deterministic algorithm, the adversary can request a page not in cache in each step. Then every request is a miss, and n misses in total.

Optimum algorithm can always evict the page that will be requested furthest in the future. So when a miss occurs, at least k other requests must have been in the cache. Then optimum misses one every k (or roughly $\frac{n}{k}$). Therefore, competitive ratio of any deterministic alg $\geq k$. ■

Lemma 3 *LRU has competitive ratio k.*

Proof is left as an exercise.

23.1.2.3 Deterministic 1-Bit LRU (Marking Algorithm)

There is a variant of LRU: maintain a single bit for each page. For each marked/unmarked page: bit is 0/1, respectively. The procedure is as follows:

Procedure:

- Initially all pages are unmarked in phase.
- When a page is requested:
  - If the page is in the cache, mark it.
  - Otherwise:
    - If $\exists$ an unmarked page, then evict an arbitrary unmarked page, bring in the request & mark it.
    - Else, unmark all pages and start next phase.

Lemma 4 *The Marking algorithm is k-competitive.*

Proof. Consider some i-th phase. We have k distinct pages accessed 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. Therefore, 1-bit LRU has competitive ratio k ■

23.1.2.4 Randomized 1-Bit LRU (Marking)

Suppose instead of evicting an arbitrary unmarked page, we choose one uniformly at random.

Lemma 5 *The randomized marking is $O(\log k)$-competitive.*

Proof. We show a competitiveness of $2H_k$, where $H_k = 1 + \frac{1}{2} + \cdots + \frac{1}{k} \approx \ln k$. Let S_i be the set of pages in cache at start of phase i. Let $\Delta_i = |S_{i+1} \setminus S_i|$ denote the number of new pages added during phase i.

We show $E[\#\text{ of misses in phase } i] \leq \Delta_i(H_k + 1)$. Suppose R_i are the distinct requests in phase i. There are two types of requests:

- **Clean request:** Request page not in S_i.
- **Stale request:** Otherwise.

A cache miss is either clean or stale.

Note that the # of missed clean requests $\leq \Delta_i$. Each clean request brings in a new request and adds to cache & marks it in S_{i+1}. To bound the # of stale cache misses: Suppose there are c clean requests so far and consider the $(s+1)$-th stale request. The probability that this causes a miss is $\leq \frac{c}{k-s}$ since we have evicted c random pages out of the remaining $k-s$ stale requests. Since $c \leq \Delta_i$, we have:

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

So $E[\#\text{ of cache misses in phase } i] \leq \Delta_i(1 + H_k)$ (1).

Now we show for OPT, the # 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. Therefore, we have

$$n_{i-1} + n_i \geq \Delta_i \quad (2)$$

, and consequently,

$$Opt \geq \frac{1}{2}\sum \Delta_i$$

Combining (1) & (2):

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

■

23.1.3 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 function $d : V \times V \to \mathbb{R}^+$ that satisfies the 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 to a_t, we 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 $x, y \in V$, set $d(x, y) = 1$.

There is a deterministic $(2k-1)$-competitive algorithm for k-server. It is conjectured that there is a k-competitive deterministic algorithm for k-server. For a long time a poly-logarithmic competitive randomized algorithm was open until a work by Bansal/Buchbinder/Madry/Naor in 2011 [BBMN11].

23.2 Online Decision Making: The Secretary Problem

Suppose we want to interview people for a secretary position. We have a set of n candidates and we want to hire the best one. We can interview one-by-one and we have to make a decision (to hire or pass and continue) after each interview. These decisions are irrevocable!

- 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 and stop, or pass (continue) to see next.

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

Intuition: If we choose early on: chance of picking the best is low, and if we wait too long we miss out on the good ones.

How about:

- Pass on first $\frac{n}{2}$
- Then pick first that is better than all seen.

Lemma 6 *This strategy finds the best with probability* $\geq 1/4$.

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.

$$\Pr[\text{win}] \geq \Pr[E_1 \wedge E_2] = \Pr[E_1] \cdot \Pr[E_2|E_1] = \frac{1}{2} \cdot \frac{n/2}{n-1} = \frac{1}{4} + o(1)$$

. ■

Can we improve this? Note that the analysis is loose; we still find the best when, for example:

- The third best element is in the first half, the best and second best elements are in the second half, and the best element appears before the second best element.

Improved strategy: 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 7 *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:

1. E_1: a_i is the best.
2. E_2: the best among $a_1 \ldots a_{i-1}$ has rank $\leq k$.

$$\begin{aligned}
\Pr[\text{win}] &= \sum_{i=k+1}^{n} \Pr[a_i \text{ is the best and alg picks it}] \\
&= \sum_{i=k+1}^{n} \Pr[E_1 \wedge E_2] \\
&= \sum_{i=k+1}^{n} \Pr[E_1] \cdot \Pr[E_2] \\
&= \sum_{i=k+1}^{n} \frac{1}{n} \cdot \frac{k}{i-1} \\
&= \frac{k}{n} \sum_{i=k+1}^{n} \frac{1}{i-1} \\
&= \frac{k}{n}(H_{n-1} - H_{k-1}) \\
&\simeq \frac{k}{n}(\ln(n-1) - \ln(k-1)) \\
&= \frac{k}{n} \ln \frac{n-1}{k-1} = (1 - o(1))\frac{k}{n} \ln \frac{n}{k}
\end{aligned}$$

■

Since we want to maximize this, we solve for the best k: $f(x) = \frac{x}{n} \ln \frac{n}{x}$.

Take derivative (1st & 2nd):

$$f'(x) = \frac{1}{n}(\ln \frac{n}{x} - 1); \quad f''(x) = \frac{1}{n}(-\frac{1}{x})$$

In $(0,1)$, f is concave and the maximum is when $f'(x) = 0 \to \ln(\frac{n}{x}) = 1 \to x = \frac{n}{e}$.

So the best value for k is n/e. Therefore,

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

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

23.3 References

[BBMN11] Nikhil Bansal and Niv Buchbinder and Aleksander Madry and Joseph Naor. A Polylogarithmic-Competitive Algorithm for the k-Server Problem. In Journal of the ACM (JACM), vol. 62, no. 40, pp. 1-49. 2015.