COMPSCI 532: Design and Analysis of Algorithms	November 4, 2015
Lecture 19	
Lecturer: Debmalya Panigrahi	*Scribe: Allen Xiao*

1 Overview

In this lecture, we motivate online algorithms and introduce some of the algorithmic tools we use to design and analyze online algorithms. The online model is quite rich in the sense that it encapsulates many practical problems where we do not receive the entire input before we need to act.

2 Online Algorithms

Online algorithms use a different model from what we've seen so far: the algorithm does not have the entire input at its beginning, but receives it piece by piece, sequentially, over time. After receiving each piece, the algorithm must immediately make a decision (updating a data structure, buying/selling stocks) which each have an associated cost. Ultimately, the goal of the algorithm is to minimize the total cost over time.

Under this model, an offline algorithm is equivalent to receiving the entire input first, then making all decisions at the end with the full information. If we placed no restrictions on the ability to *retract* decisions (by making it expensive or disallowing it entirely), we could have our online algorithm perform a full recomputation after receiving each new piece of input, and obtain a solution equivalent to the offline version. There are various flavors of this restriction, some which are more domain-focused:

- Dynamic data structures: cost is the update time of a data structure, expensive to rebuild from scratch.
- Streaming algorithms: cost is the space (memory) of a representation of past inputs. Often required to be sub-linear, so we cannot store the raw input history.

For this lecture, the examples we discuss will use the model where decisions are irrevocable. For example, in vertex cover, we receive edges online and must maintain a cover of the current graph, but we cannot un-choose vertices already chosen. Obviously, depending on the sequence of input pieces, the algorithm may be forced to make decisions that are sub-optimal when the rest of the input is known. Online algorithms describe the problem of trying to plan for the future, without knowing it.

2.1 Ski rental problem (rent-or-buy)

The following is a classic example of an online problem. Suppose you go on a ski trip, for as many days as possible until you break your leg (this is the only condition under which the trip ends). You do not own skis, so you have the option of either renting skis for 10 dollars per day, or buying them for 100 dollars. Each day, you receive one new bit of information (did your leg break?) and you must decide (if you haven't already bought skis) whether you should rent or buy that day. The goal is to minimize the total amount spent on skis.

If you knew your leg would break within the first 10 days, it would be cheapest to rent every day. On the other hand, if the trip ended after 10 days, the ideal solution would be to buy on the first day. When your leg

breaks is unknown, however, so intuitively we could design an algorithm somewhere in the middle ground: rent first, then buy after enough days have passed that the purchase price does not greatly overshadow the amount paid for rentals.

Formally, suppose the trip lasts x days (unknown), and that it costs 1 per day to rent, k to buy skis. Our algorithm will be to rent for k days, then buy on the $(k+1)$-st day. The cost of this algorithm is:

$$\text{cost} = \begin{cases} x & \text{if } x \leq k \\ 2k & \text{if } x > k \end{cases}$$

To analyze the performance of this algorithm, we first describe competitive analysis.

2.2 Competitive analysis

Analyzing an algorithm for an online problem seems tricky, since optimality seems contingent on the unknown future input. In the ski rental example, consider the cases where your leg breaks on day 2 versus day $100k$. To an algorithm on the first day, both these inputs look the same. However, the ideal solutions for each look very different: rent for a day, versus buy immediately. It seems unclear what the criteria for an optimal online algorithm would be, because of incomplete information.

Instead, we will compare our online algorithm against the *offline optimal* algorithm (with complete information). This is the traditional setting with which we are fimilar. This technique is called **competitive analysis**, and the resulting performance ratio is called the **competitive ratio**. For a minimization problem with instances I:

$$\max_I \frac{\text{ALGO}(I)}{\text{OPT}_{\text{offline}}(I)}$$

2.3 Competitive analysis of ski rental

As we determined earlier, the offline optimal solution for the ski rental problem is to buy immediately if the trip lasts longer than k days, otherwise rent all days. The cost of the offline optimal solution is therefore:

$$\text{cost}(\text{OPT}) = \begin{cases} x & \text{if } x \leq k \\ k & \text{if } x > k \end{cases}$$

The worst case ratio between the solutions is when $x > k$, and the competitive ratio is 2.

2.4 Caching/paging

In the caching problem, we have a cache of size k and n different pages which may be requested at each time step. When page i is requested but not stored in the cache, we suffer a *cache fault* and need to pring i into the cache. If the cache is already full, we must choose something to evict and replace with i. The goal is to minimize the total number of cache faults, by choosing the page-to-evict carefully.

The offline optimal solution evicts the page whose next request is furthest in the future (i.e. "all other pages in the cache will be requested again before this page is"). We will not state the formal proof of optimality here. Like before, it is critical for this algorithm that we have the entire input. A classical (non-clairvoyant) surrogate is a replacement policy called *Least Recently Used* (LRU): evict the page requested furthest in the past.

To analyze the two eviction policies, we recursively partition the request sequence into a sequence of *rounds*, a maximal subsequence of requests for at most k different page types (starting from the beginning of the sequence). For example, with $k = 3$ and the request sequence:

$$5,3,5,4,5,3,2,9,9,3,5,9,2,3$$

The partitioning into rounds is:

$$(5,3,5,4,5,3),(2,9,9,3),(5,9,2),(3)$$

Each round has no more than k different page types. Now, how well does LRU perform?

Lemma 1. *LRU has at most 1 fault per page type each round, and therefore at most k faults per round.*

Proof. Assume for contradiction that in a single round, page i is fetched, evicted, then fetched again (producing 2 faults). At eviction, i was the last requested furthest in the past of all other elements in the cache. Then there are $k-1$ elements in the cache which were requested after the most recent request for i (within this round). Combined with the new request which causes i's eviction, there are a total of $(k-1)+1+1 = k+1$ unique page types in this round, a contradiction of our definition for rounds. □

On the other hand, OPT faults at least once per round. Each round is a maximal sequence of k types (filling the cache), so the $(k+1)$-st must fault for OPT; this new page would not have been kept in the cache across rounds, since it would be accessed furthest in the future when the other $k-1$ cache entries filled with in-round elements. Summing over rounds, the competitive ratio for LRU is most k.

3 Randomization and Online Algorithms

Without randomization, the previous examples we gave are tight for ski rental and caching (the competitive ratio meet known lower bounds). In fact, randomization gives a lot of power in the online setting for competitive ratio, which is a worst-case analysis. To see why, we can think of these online lower bounds as a game between the algorithm and an *adversary* designing the input. Each step,

1. The adversary moves, providing the next piece of online input.
2. The algorithm moves, making whatever decisions it will on the input.

The adversary attempts to make the competitive ratio as poor as possible.

3.1 Lower bound for ski rental

An adversarial strategy for ski rental is to choose x to be exactly the day you buy skis (break your leg the day you commit to buying). Any ski rental algorithm has the form: "buy skis after renting T times." With this adversarial strategy, that ski rental algorithm will always have cost $T+k$. The competitive ratio is therefore:

$$\frac{T+k}{\min(T,k)} \geq 2$$

3.2 Lower bound for caching

The adversarial strategy this time is to present any page not in the cache, necessarily causing a fault. Remember, in this adversarial model the adversary observes ALGO's progress as the game proceeds, and designs the input based on its progress. In this case, it is ALGO's cache which must fault on every input, while the offline OPT may do much better.

Let the number of requests in the sequence be $|S|$. No matter what the algorithm is, the adversarial choice means that it faults on every request for a total of $|S|$ faults. On the other hand, using the partitioning into rounds we used previously, OPT must have no more than $|S|/k$ faults (one for each round, each round at least length k). The competitive ratio is lower bounded by k. LRU is optimal among deterministic caching algorithms in competitve ratio.

3.3 Randomized competitive analysis

An adversary which knows the random bits of a random algorithm can treat the algorithm as deterministic – it could simulate the algorithm using the random bits to observe the future progress of the algorithm on any input it chooses. Instead, we will often use the model that the adversary can observe the algorithm, but is blind to the random bits used by the algorithm. Formally, this is a distinction between *adaptive* and *oblivious* adversaries.

1. An **oblivious** adversary knows the algorithm code, but must generate the entire input before execution begins (i.e. without knowing any random bits).

2. An **adaptive** adversary knows the algorithm and generates the input online as the algorithm works on it – it can see the results of the algorithm's randomized responses. The input sequence is therefore a random variable dependent on the random bits of the algorithm.

The typical definition of these adversary models also distinguishes between two possible cost schemes for the adaptive adversary (whether the competitive ratio compares the algorithm's result against the offline optimal of this sequence or something else). For more information, a good reference is the textbook by Borodin and El-Yaniv [BEY05]. We will focus on the oblivious model, which allows better bounds than either adaptive adversary.

3.4 Randomized ski rental

We randomize ski rental by setting a distribution over dates to buy, rather than fixing a buy date. Recall that our two daily options were to buy skis for k, or pay 1 to rent for that day. Suppose now that we buy after $t > 0$ days with probability p_t, such that $\int_0^\infty p_t = 1$. Our goal is to pick a distribution of p_t which minimizes the competitive ratio in expectation.

Although we initially described the problem in terms of discrete time steps, we will use a continuous-time version in the analysis. The discrete-time version is a bit more involved; a note by Claire Mathieu [Mat07] covers both versions, if you are curious. Finally, more work is required to show that the competitive ratio in the continuous version bounds the competitive ratio for the discrete one, which we omit.

Suppose the skiing ends after x days. The expected cost of our algorithm is:

$$\int_0^x p_t(t+k)dt + x\int_x^\infty p_t dt$$

The first term describes the cases where the skis are bought before the trip ends ($t \leq x$); while the second term adds the cases when the skis are not purchased before the trip ends ($t > x$). Suppose our goal is a constant competitive ratio of c, which we determine later. To determine the correct values of p_t to choose, we examine the competitive ratio under two cases on x:

1. If $x > k$, then the optimal solution is to buy immediately, spending k. We can write:

$$ck = \int_0^x p_t(t+k)dt + x\int_x^\infty p_t dt$$

   Taking the derivative with respect to x:

$$\begin{aligned} 0 &= (k+x)p_x + \int_x^\infty p_t dt - xp_x \\ &= kp_x + \int_x^\infty p_t dt \end{aligned}$$

   Since all $p_t \geq 0$ and $k > 0$, for this equation to hold we must use $p_t = 0$ for all $t > k$.

2. For $x \leq k$, the optimal solution is to rent daily for a total cost of x. We can write:

$$cx = \int_0^x p_t(t+k)dt + x\int_x^\infty p_t dt$$

   Taking the derivative with respect to x:

$$\begin{aligned} c &= (k+x)p_x + \int_x^\infty p_t dt - xp_x \\ &= kp_x + \int_x^\infty p_t dt \end{aligned}$$

   Taking another derivative with respect to x:

$$0 = kp'_x - p_x$$

   Rearranging, this gives $p_x/p'_x = k$, a first-order differential equation which has solutions of the form $p_x = Ae^{x/k}$ for some constant A. Thus, for all $t \leq k$, we will use $p_t = Ae^{t/k}$.

With these two cases, we have an expression for p_t:

$$p_t = \begin{cases} Ae^{t/k} & t \leq k \\ 0 & t > k \end{cases}$$

It remains to determine the constant A and the exact competitive ratio c this choice of p_t affords us. To solve for A, recall that $\int_0^\infty p_t dt = 1$.

$$\begin{aligned} 1 &= \int_0^\infty p_t dt \\ &= \int_0^k Ae^{t/k} dt \\ &= \left[Ake^{t/k}\right]_{t=0}^k \\ &= Ak(e-1) \end{aligned}$$

We can rearrange to obtain $A = 1/(k(e-1))$. Plugging the expression for p_t back into the earlier formula for c:

$$\begin{aligned} c &= kp_x + \int_x^\infty p_t dt \\ &= k \cdot \frac{e^{x/k}}{k(e-1)} + \int_x^k \frac{e^{t/k}}{k(e-1)} dt \\ &= \frac{e^{x/k}}{e-1} + \left[\frac{e^{t/k}}{e-1} \right]_{t=x}^{k} \\ &= \frac{1}{e-1} \left(e^{x/k} + e^{k/k} - e^{x/k} \right) \\ &= \frac{e}{e-1} \end{aligned}$$

Using this choice of p_t gives an expected competitive ratio of $e/(e-1) \approx 1.58$.

3.5 Randomized caching

Again, suppose we have a cache of size k and a sequence of requests over n pages, and we wish to minimize the number of faults. We introduce a variation of LRU which, using randomization, gives a slightly better competitive ratio. For the reader familiar with the *clock/second-chance algorithm*, this is a version which chooses randomly among the un-marked pages instead of using the first one found. We call this the *marking* algorithm.

1. Mark a page when it is requested (pages enter the cache marked).
2. When evicting a page, choose uniformly at random among the unmarked pages.
3. When an eviction is required but all cache entries are marked, unmark all pages in the cache.

Intuitively, this is a coarser version of LRU where we use a single bit ("old" or "new") to represent the age of a page, instead of the $\log k$ to order it exactly among the other cached pages. Randomization is crucial to obtaining a better bound, however.

To analyze this algorithm, we again partition the request sequence into rounds (maximal sequence of k different page types).

Lemma 2. *The cache is unmarked exactly at the beginning of each round.*

Proof. We can use induction starting from the first round. In the first round, the cache is empty and begins unmarked. Suppose round i begins with an unmarking, and proceeds with requests to round $i+1$. The first request of round $i+1$ sees a fully marked cache of the k elements in round i, and therefore causes an unmarking event. □

To analyze the number of faults in ALGO, we split the requests during round i into two categories:

1. *Old* requests: the page type was already in the cache from round $i-1$ (it was one of the k page types in round $i-1$).
2. *New* requests: the page type was not in the cache from round $i-1$. The first time a new page is requested during round i, ALGO incurs a fault.

Observe that, to maximize the number of faults in the round, an oblivious adversary would place all new requests before all old requests. This fills up the most cache entries with new pages, giving each old request a greater chance of causing a fault. Let ℓ_i be the number of new requests in round i. Using this request order, there are a guaranteed ℓ_i faults from new pages in round i. It remains to show the expected number of faults from the $k-\ell_i$ requests to old pages for round i.

When we request the j-th old page for the first time in round i, the probability it causes a fault (cache entry was replaced) is exactly $\ell_i/(k-j+1)$. To see this, note that the probability that the it does not cause a fault is:

$$\frac{\text{\# unmarked old pages still in cache}}{\text{\# unmarked old pages left this round}} = \frac{k-\ell_i-(j-1)}{k-(j-1)}$$

That is, it does not fault if its unmarked entry was not replaced by the new requests or any of the $j-1$ old requests preceding it. The probability that it faults is therefore:

$$1-\frac{k-\ell_i-(j-1)}{k-(j-1)} = \frac{\ell_i}{k-j+1}$$

Lemma 3. *The expected number of faults by the marking algorithm in round i is no more than $\ell_i H_k$, where H_k is the k-th harmonic number.*

Proof. The sum of expected faults from old and new requests is:

$$\begin{aligned}
\ell_i + \sum_{j=1}^{k-\ell_i} \frac{\ell_i}{k-j+1} &= \ell_i + \frac{\ell_i}{k} + \frac{\ell_i}{k-1} + \cdots + \frac{\ell_i}{\ell_i+1} \\
&= \ell_i + \ell_i(H_k - H_{\ell_i}) \\
&\leq \ell_i H_k
\end{aligned}$$

□

Summing over all rounds, we see that the expected number of faults is $H_k \sum_i \ell_i$. To complete our analysis, we must lower bound the number of faults made by OPT.

Lemma 4. *The total number of faults in the offline optimal is at least $\frac{1}{2}\sum_i \ell_i$.*

Proof. Observe that between rounds $i-1$ and i, there are $k+\ell_i$ unique page types. Any algorithm has at most k of these pages cached before entering round $i-1$, therefore even the offline optimal must incur at least ℓ_i faults. The number of faults in all of these pairs is at least:

$$\sum_i \ell_i$$

This is not a proper enumeration of the faults, however. The pairs are not disjoint, so if we sum over all pairs of rounds, we double count each round (beside the first and last). Therefore, the number of faults is at least half.

$$\frac{1}{2}\sum_i \ell_i$$

□

Together, these give an competitive ratio of at most $2H_k$, roughly $O(\log k)$ compared to the ratio of k from the deterministic LRU algorithm.

To conclude, we mention a related problem, popular in both theory in practice, called the k-server problem. k-server generalizes the faults to have weights, in a sense. A major open problem is whether k-server admits an algorithm with $O(\log k)$ competitive ratio, like caching.

References

[BEY05] Allan Borodin and Ran El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, 2005.

[Mat07] Claire Mathieu. Online algorithms: Ski rental. URL: `http://cs.brown.edu/~claire/Talks/skirental.pdf`. Computer Science Department, Brown University, 2007.