# Lecture 15: Online Algorithms

Lecturer: *Huacheng Yu*

Lecture notes sourced from Avrim Blum's lecture notes here:
http://www.cs.cmu.edu/ avrim/Approx00/lectures/lect0301 .

## 1 Online Algorithms

This lecture will focus on *online algorithms*. That is, algorithms which have to make decisions "online" without knowing the entire input. We already saw one example of this paradigm earlier with the multiplicative weights update rule, and the spirit of what we'll cover today is similar.

We'll focus on two core examples, the *ski rental problem* and *list update problem*. For both problems we'll touch on how to design algorithms, and prove lower bounds. The lower bounds won't be computational and won't depend on assumptions like P vs. NP.

The main measure of quality for online algorithms are their *competitive ratio*. For a cost minimization problem (both problems we'll see today are cost-minimization), we'll say that an online algorithm has competitive ratio $C$ if for *all* inputs, the cost incurred by the online algorithm is at most $C \cdot \mathsf{OPT}$, where $\mathsf{OPT}$ denotes the cost incurred by the best *offline* solution that knows the entire input before making any decisions.

## 2 Ski Rental

We'll first look at the Ski Rental problem. Winter is coming, and you're trying to figure out skiing plans and whether to buy your own skis (without your own skis, you'll need to rent them every time you go). The good news is that the pricing scheme is transparent: buying your own skis will incur a one-time cost of $B$ (and you can buy them whenever you want). Renting skis will cost $R < B$ per trip. The bad news is you have absolutely no idea how often your friends will want to ski. You might buy skis today, and your friends decide to never go again. Alternatively, you might decide to never buy and wind up renting every weekend.

You could certainly come up with a probabilistic model of your likelihood of skiing and figure out the solution minimizing expected cost, but this is hard and also doesn't provide much insight. It turns out that using the competitive ratio instead gives a simple, insightful suggestion.

To be clear, the problem develops like so: On day 1, you decide whether to rent or buy. If you buy, you pay $B$ and the game ends (you now have skis for all future days). If you rent, you pay $R$, and the game continues. Now, day 2 may or may not arrive. If it doesn't, the game ends. If it does, you must again decide whether to rent or buy. If you buy, you pay an (additional) $B$ and the game ends. If you rent, you pay $R$ and the game continues. In general, when day $i$ comes, you decide whether to rent or buy. If you buy, you pay $B$

and the game ends. If you rent, you pay $R$ and the game continues. Day $i+1$ may or may not arrive (if it doesn't, the game ends).

Let's first figure out what the optimal offline solution is. If you knew exactly how many days $D$ you were skiing, what should you do? If $DR < B$, then rent all $D$ days. Otherwise, buy immediately. So our goal will be to design a strategy that guarantees a total cost at most $C \cdot \min\{DR, B\}$ when you ski for $D$ days for a small $C$.

Let's now also consider, for any deterministic algorithm, what the worst possible $D$ is. Note that deterministic algorithms can be completely described by the maximum number of days $T$ to rent before buying. That is, every deterministic algorithm is of the form "rent for the first $\min\{T, D\}$ days, and buy on day $T+1$ if $D > T$." What's the worst possible $D$ for a given $T$?

**Proposition 1.** *The maximum ratio between any deterministic algorithm and the offline optimum is achieved when the last day of skiing is the day the algorithm purchases skis. That is, the maximum ratio of:*

$$\frac{R \cdot \min\{T, D\} + B \cdot I(D > T)}{\min\{DR, B\}}$$

*is acheived at $D = T + 1$.*

*Proof.* Note that if $D > T$, then every day that passes $T$ our algorithm incurs additional cost 0, while the optimum incurs additional cost at least 0 (maybe the optimum already bought, in which case its 0, or they're still renting in which case the extra cost per day is $R$). So the worst case among all $D > T$ is if your last day skiing is the same day you buy. This makes sense: the worst outcome for you conditioned on buying is that you never use the skis again. Now let's think about the worst case among $D \leq T$. Note that the ratio in this regime is just $\frac{DR}{\min\{DR,B\}}$. When $DR < B$, this ratio is 1. When $DR > B$, the ratio is strictly increasing in $D$ (because the numerator increases at a rate of $R$ and the denominator is static).

Finally, in comparing $T$ to $T+1$, note that either $TR \geq B$, in which case both ratios have the same denominator (but the numerator for $T+1$ is bigger), or $TR < B$, in which case the ratio for $T$ is $TR/TR = 1$, whereas the ratio for $T+1$ is $> 1$ (as $TR + B > B$ and $TR + B > (T+1)R$). So for all $T$, the worst case is when you go skiing for exactly $T+1$ days, and only use your purchased skis once. □

**Corollary 2.** *The competitive ratio achieved by the deterministic algorithm that purchases skis on day $T+1$ is exactly $\frac{TR+B}{\min\{(T+1)R,B\}}$.*

For simplicity, assume that $B$ is an integer multiple of $R$. In this case, the best competitive ratio is achieved when $T+1 = B/R$, and the ratio is exactly $(2T+1)/(T+1) = 2 - R/B$ (to prove this, again observe that if $T+1 > B/R$, then decreasing $T$ doesn't change the denominator, but decreases the numerator. If $T+1 < B/R$, then the ratio is $1 + \frac{B/R-1}{T+1}$, which will clearly decrease if we increase $T$. Both claims assume $B/R$ is an integer).[1] Note that in developing the algorithm we've also proved that it's optimal among all deterministic algorithms.

[1] If $B/R$ isn't an integer, then the optimal competitive ratio is achieved when $T+1 \in (\lfloor B/R \rfloor, \lceil B/R \rceil)$.

# 3 List Update

Now, imagine that you have a single linked list of elements. Every time the list is queried for element $x$, you have to scan from the start of the list until you hit $x$. But you can edit the list a bit as you go: after accessing $x$, you can pull $x$ as far to the front of the list as you like (keeping all other elements in the same relative order). The total cost for accessing element $x$ is the depth you have to scan in order to find it (moving it forwards is free). The total cost of the algorithm is just the sum of all accesses.

Again, one could try natural probabilistic models: say that each request is for a random element, and element $x$ is requested with probability $p_x$. Then it's not too hard to see that the best solution is just to maintain the elements in decreasing order of $p_x$. But this solution is somewhat unsatisfying: it just pushes all of the work onto estimating the values $p_x$, which realistically won't be independent (if you just accessed element $x$, you might be more likely to access it or something related to it). A reasonable compromise might be to use historical values of $p_x$ (and bring an element up whenever its most recent access makes its historical $p_x$ larger than some elements currently in front of it). But apparently this does poorly in practice. The competitive ratio seems to capture this fact nicely. Formally the algorithm will be as follows, called FrequencyCount:

- Initialize $C(x) = 0$ for all $x$.
- When element $x$ is queried, increment $C(x)$. Then, compare $C(x)$ to $C(y)$ for all $y$ currently ahead of $x$. If $C(x) > C(y)$, move $C(x)$ ahead of $C(y)$.

Note that FrequencyCount maintains the invariant that the list is always sorted in decreasing order of $C(\cdot)$. Now we'll show that FrequencyCount doesn't have a competitive ratio better than $\Omega(n)$, where $n$ is the number of elements. Note that no algorithm can have competitive ratio worse than $n$, since it does at most $n$ scans on every query but opt does at least 1, so this is pretty bad.

**Proposition 3.** *The competitive ratio of FrequencyCount is $\Omega(n)$.*

*Proof.* Consider the following sequence: First, query element $i$ exactly $i$ times (in that order). This gets the list in the order $n, n-1, \ldots, 1$. Next, repeat the following $k$ times (for large $k$): for $i = 1$ to $n$, request the $i$-th element $n$ times in a row.

Note that this will do the following to the list: the first time element $i$ is queried, it's all the way in the back. Each time it's queried (except for the first), it moves up one spot. So the total cost paid to access $i$-th element $n$ times is $n + \sum_{j=2}^{n} j = \Omega(n^2)$. So the total cost paid by FrequencyCount is $\Omega(kn^3)$.

Now consider the optimal offline solution: whenever element $i$ is queried for the first time (in each of the $k$ iterations), move it all the way to the front. Then, the total cost paid to access $i$ $n$ times is $2n$, and the total cost paid by the optimal offline solution is $O(kn^2)$. So the competitive ratio of FrequencyCount is $\Omega(n)$. □

It turns out that there's a better solution with a competitive ratio of 2, called MoveToFront: every time you access an element, move it all the way to the front. It might seem a little surprising that this works - why should you expect that the element you just accessed is

going to be accessed again soon? (maybe in practice this makes sense, but it doesn't show up in the model at all). The idea is the following: if recently accessed elements aren't accessed again until very far in the future, then even the offline optimum can't do anything clever, so it's OK if you're a bit wasteful. But if recently accessed elements are accessed again soon, the offline optimum can be very good, so you better also be very good.[2]

**Proposition 4.** *MoveToFront achieves a competitive ratio of at most* 2*.*

*Proof.* There is a really clever (and shorter) proof in the referenced notes due to Adam Kalai. But we'll use the argument below because it demonstrates the key idea of a *potential function.*

Imagine running MoveToFront and the Offline Optimum side-by-side on the same input (for which the offline optimum is optimal). At all times $t$ (that is, after $t$ requests have been made), we'll let $\Phi(t)$ denote the number of pairs $(x, y)$ such that $x$ is in front of $y$ in MoveToFront (at time $t$), but $x$ is behind $y$ in the offline optimum (at time $t$). Note that $\Phi(0) = 0$, that $\Phi$ can increase or decrease, but we will always have $\Phi(t) \geq 0$.

We'll also let $MTF(t)$ denote the cost paid by MoveToFront to access the element requested at time $t$, and $OPT(t)$ denote the cost paid by the offline optimum to access the element requested at time $t$. We'll show that for every $t$, $MTF(t) + (\Phi(t) - \Phi(t-1)) \leq 2OPT(t)$. If we can prove this, then we can sum over all $t$ to get that the cost of MTF is at most twice the cost of the offline optimum, because $\Phi$ is always non-negative.

So now let's prove the claim: Consider at time $t$ the request to element $x$. Say that MoveToFront pays $p$ to access $x$, and that $k$ elements in front of $x$ (before moving $x$ to the front) are also in front of $x$ in the offline optimum (before accessing $x$ and potentially moving it). Then $MTF(t) = p$, and $OPT(t) \geq k + 1$.

Now, let's look at the potential. First, consider the intermediate case where the offline optimum hasn't yet moved $x$. Then $x$ is moved in front of $k$ elements that were also in front of it in the offline optimum, so $\Phi$ goes up by $k$. But $x$ is also moved in front of $p - k - 1$ elements that should have been behind $x$ in the offline optimum, so $\Phi$ also goes **down** by $p - k - 1$. So at this point, we have that $\Phi$ goes up by exactly $2k - p + 1$. Finally, from this point, note that the offline optimum might also move $x$ closer to the front. However, these changes will only *decrease* $\Phi$: $x$ is in front of everything in the MoveToFront list, so every element that $x$ gets moved in front of by the offline optimum makes the lists agree on one additional pair. So we get that:

$$\begin{aligned} MTF(t) &= p \\ OPT(t) &\geq k + 1 \\ \Phi(t) - \Phi(t-1) &\leq 2k - p + 1 \\ \Rightarrow MTF(t) + \Phi(t) - \Phi(t-1) &\leq 2 \cdot OPT(t). \end{aligned}$$

□

[2]The intuition is similar to why we use the multiplicative weight rule: if none of the experts did well in hindsight, how could we be expected to outperform any expert? But as long as one of the experts is good, then MWU does well as well.