Lecture 17

In which we introduce online algorithms and discuss the buy-vs-rent problem, the secretary problem, and caching.

In this lecture and the next we will look at various examples of algorithms that operate under partial information. The input to these algorithms is provided as a "stream," and, at each point in time, the algorithms need to make certain decisions, based on the part of the input that they have seen so far, but without knowing the rest of the input. If we knew that the input was coming from a simple distribution, then we could "learn" the distribution based on an initial segment of the input, and then proceed based on a probabilistic prediction of what the rest of the input is going to be like. In our analysis, instead, we will mostly take a worst-case point of view in which, at any point in time, the unknown part of the input could be anything. Interestingly, however, algorithms that are motivated by "learn and predict" heuristics often work well also from the point of view of worst-case analysis.

1 Online Algorithms and Competitive Analysis

We will look at online algorithms for optimization problems, and we will study them from the point of view of *competitive analysis*. The *competitive ratio* of an online algorithm for an optimization problem is simply the approximation ratio achieved by the algorithm, that is, the worst-case ratio between the cost of the solution found by the algorithm and the cost of an optimal solution.

Let us consider a concrete example: we decide to go skiing in Tahoe for the first time. Buying the equipment costs about $500 and renting it for a weekend costs $50. Should we buy or rent? Clearly it depends on how many more times we are going to go skiing in the future. If we will go skiing a total of 11 times or more, then it is better to buy, and to do it now. If we will go 9 times or fewer, then it is better to rent, and if we go 10 times it does not matter. What is an "online algorithm" in this case? Each time we want to go skiing, unless we have bought equipment a previous time, we have to decide whether we are going to buy or rent. After we buy, there is no more decision to make; at any time, the only "input" for the algorithm is the

fact that this is the k-th time we are going skiing, and that we have been renting so far; the algorithm decides whether to buy or rent based on k. For deterministic algorithms, an algorithm is completely described by the time t at which it decides that it is time to buy.

What are the competitive ratios of the possible choices of t? If $t = 1$, that is if we buy before the first time we go skiing, then the competitive ratio is 10, because we always spend \$500, and if it so happens that we never go skiing again after the first time, then the optimum is \$50. If $t = 2$, then the competitive ratio is 5.5, because if we go skiing twice then we rent the first time and buy the second, spending a total of \$550, but the optimum is \$100. In general, for every $t \leq 10$, the competitive ratio is

$$\frac{500 + 50(t-1)}{50t} = 1 + \frac{9}{t}$$

If $t \geq 10$, then the competitive ratio is

$$\frac{500 + 50(t-1)}{500} = .9 + \frac{t}{10}$$

So the best choice of t is $t = 10$, which gives the competitive ratio 1.9.

The general rule for buy-versus-rent problems is to keep renting until what we have spent renting equals the cost of buying. After that, we buy.

(The "predicting" perspective is that if we have gone skiing 10 times already, it makes sense to expect that we will keep going at least 10 more times in the future, which justifies buying the equipment. We are doing worst-case analysis, and so it might instead be that we stop going skiing right after we buy the equipment. But since we have already gone 10 times, the prediction that we are going to go a total of at least 20 times is correct within a factor of two.)

2 The Secretary Problem

Suppose we have joined an online dating site, and that there are n people that we are rather interested in. We would like to end up dating the best one. (We are assuming that people are comparable, and that there is a consistent way, after meeting two people, to decide which one is better.) We could go out with all of them, one after the other, and then pick the best, but our traditional values are such that if we are dating someone, we are not going to go out with anybody else unless we first break up. Under the rather presumptuous assumption that everybody wants to date us, and that the only issue is who we are going to choose, how can we maximize the probability of ending up dating the best person? We are going to pick a random order of the n people, and go out, one after the other, with the first n/e people. In these first n/e dates, we just waste other people's time: no matter how the dates

go, we tell them that it's not them, it's us, that we need some space and so on, and we move on to the next. The purpose of this first "phase" is to calibrate our expectations. After these n/e dates, we continue to go out on more dates following the random order, but as soon as we found someone who is *better than everybody we have seen so far*, that's the one we are going to pick. We will show that this strategy picks the best person with probability about$1/e$, which is about 37%.

How does one prove such a statement? Suppose that our strategy is to reject the people we meet in the first t dates, and then from date $t+1$ on we pick the first person that is better than all the others so far. The above algorithm corresponds to the choice $t = n/e$.

Let us identify our n suitors with the integers $1, \ldots, n$, with the meaning that 1 is the best, 2 is the second best, and so on. After we randomly permute the order of the people, we have a random permutation π of the integers $1, \ldots, n$. The process described above corresponds to finding the minimum of $\pi[1], \ldots, \pi[t]$, where $t = n/e$, and then finding the first $j \geq t+1$ such that $\pi[j]$ is smaller than the minimum of $\pi[1], \ldots, \pi[t]$. We want to compute the probability that $\pi[j] = 1$. We can write this probability as

$$\sum_{j=t+1}^{n} \mathbb{P}[\pi[j] = 1 \text{ and we pick the person of the } j\text{-th date }]$$

Now, suppose that, for some $j > t$, $\pi[j] = 1$. When does it happen that we do *not* end up with the best person? We fail to get the best person if, between the $(t+1)$-th date and the $(j-1)$-th date we meet someone who is better than the people met in the first t dates, and so we pick that person instead of the best person. For this to happen, the minimum of $\pi[1], \ldots, \pi[j-1]$ has to occur in locations between $t+1$ and $j-1$. Equivalently, we *do* pick the best person if the best among the first $j-1$ people happen to be one of the first t people.

We can rewrite the probability of picking the best person as

$$\sum_{j=t+1}^{n} \mathbb{P}[\pi[j] = 1 \text{ and the minimum of } \pi[1], \ldots, \pi[j-1] \text{ is in } \pi[1], \ldots, \pi[t]]$$

$$= \sum_{j=t+1}^{n} \frac{1}{n} \cdot \frac{t}{j-1}$$

To see that the above equation is right, $\mathbb{P}[\pi[j] = 1] = 1/n$ because, in a random permutation, 1 is equally likely to be the output of any of n possible inputs. Conditioned on $\pi[j] = 1$, the minimum of $\pi[1], \ldots, \pi[j-1]$ is equally likely to occur in any of the $j-1$ places, and so there is a probability $t/(j-1)$ that it occurs in one of the first t locations. (Some readers may find this claim suspicious; it can be

confirmed by explicitly counting how many permutations are such that $\pi[j] = 1$ and $\pi[i] = \min\{\pi[1], \ldots, \pi[j-1]\}$, and to verify that for each $j > t$ and each $i \leq t$ the number of these permutations is exactly $(n-1)!/(j-1)$.)

So the probability of picking the best person is

$$\frac{t}{n} \sum_{j=t+1}^{n} \frac{1}{j-1} = \frac{t}{n} \left(\sum_{j=1}^{n-1} \frac{1}{j} - \sum_{j=1}^{t} \frac{i}{j} \right) \approx \frac{t}{n} \cdot (\ln n - \ln t) = \frac{t}{n} \ln \frac{n}{t}$$

And the last expression is optimized by $t = n/e$, in which case the expression is $1/e$.

Note that this problem was not in the "competitive analysis" framework, that is, we were not trying to find an approximate solution, but rather to find the optimal solution with high probability.

Note also that, with probability $1/e$, the algorithm causes us to pick nobody, because the best person is one of the first n/e with probability $1/e$, and when this happens we set our standards so high that we are ending up alone.

Suppose instead that we always want to end up with someone, and that we want to optimize the "rank" of the person we pick, that is, the place in which it fits in our ranking from 1 to n. If we apply the above algorithm, with the modification that we pick the last person if we have gotten that far, then with probability $1/e$ we pick the last person which, on average, has rank $n/2$, so the average rank of the person we pick is $\Omega(n)$. (This is not a rigorous argument, but it is close to the argument that establishes rigorously that the average is $\Omega(n)$.)

In general, any algorithm that is based on rejecting the first t people, and then picking the first subsequent one which is better than the first t, or the last one if we have gotten that far, picks a person of average rank $\Omega(\sqrt{n})$.

Quite surprisingly, there is an algorithm that picks a person of average rank $O(1)$, and which is then competitive for the optimization problem of minimizing the rank. The algorithm is rather complicated, and it is based on first computing a series of timesteps $t_0 \leq t_1 \leq \ldots \leq t_k \leq \ldots$ according to a rather complicated formula, and then proceed as follows: we reject the first t_0 people, then if we find someone in the first t_1 dates which is better than all the previous people, we pick that person. Otherwise, between the (t_1+1)-th and the t_2-th date, we are willing to pick someone if that person is either the best or the second best of those seen so far. Between the (t_2+1)-th and t_3-th date, we become willing to pick anybody who is at least the *third-best* person seen so far, and so on. Basically, as time goes on, we become increasingly desperate, and we reduce our expectations accordingly.

3 Paging and Caching

The next problem that we study arises in any system that has hierarchical memory, that is, that has a larger but slower storage device and a faster but smaller one that can be used as cache. Consider for example the virtual memory paged on a disk and the real memory, or the content of a hard disk and the cache on the controller, or the RAM in a computer and the level-2 cache on the processor, or the level-2 and the level-1 cache, and so on.

All these applications can be modeled in the following way: there is a cache which is an array with k entries. Each entry contains a copy of an entry of a larger memory device, together with a pointer to the location of that entry. When we want to access a location of the larger device (a *request*), we first look up in the cache whether we have the content of that entry stored there. If so, we have a *hit*, and the access takes negligible time. Otherwise, we have a *miss*, and we need to fetch the entry from the slower large device. In negligible extra time, we can also copy the entry in the cache for later use. If the cache is already full, however, we need to first delete one of the current cache entries in order to make room for the new one. Which one should we delete?

Here we have an online problem in which the data is the sequence of requests, the decisions of the algorithm are the entries to delete from the cache when it is full and there is a miss, and the cost function that we want to minimize is the number of misses. (Which determine the only non-negligible computational time.)

A reasonably good competitive algorithm is to remove the entry for which *the longest time has passed since the last request*. This is the Least Recently Used heuristic, or LRU.

Theorem 1 *Suppose that, for a certain sequence of requests, the optimal sequence of choices for a size-h cache causes m misses. Then, for the same sequence of requests, LRU for a size-k cache causes at most*

$$\frac{k}{k-h+1} \cdot m$$

misses.

This means that, for a size-k cache, LRU is k-competitive against an algorithm that knows the future and makes optimal choices. More interestingly, it says that if LRU caused m misses on a size-k cache on a certain sequence of requests, then, even an optimal algorithm that knew the future, would have caused at least $m/2$ misses using a size $k/2$ cache.

PROOF: Suppose the large memory device has size N, and so a sequence of requests is a sequence $a_1, \ldots, a_n$ of integers in the range $\{1, \ldots, N\}$. Let us divide the sequence into "phases" in the following way. Let t be the time at which we see the $(k+1)$-th new request. Then the first phase is $a_1, \ldots, a_{t-1}$. Next, consider the sequence $a_t, \ldots, a_n$, and recursively divide it into phases. For example, if $k = 3$ and we have the sequence of requests

$$35, 3, 12, 3, 3, 12, 3, 21, 12, 35, 12, 4, 6, 3, 1, 12, 4, 12, 3$$

then the division into phases is

$$(35, 3, 12, 3, 3), (12, 3, 3, 12, 3, 21, 12), (35, 12, 4), (6, 3, 1), (12, 4, 12, 3)$$

In each phase, LRU causes k misses.

Consider now an arbitrary other algorithm, operating with a size-h cache, and consider its behavior over a sequence of phases which is like the above one, but with the first item in each phase moved to the previous phase

$$(35, 3, 12, 3, 3, 12), (3, 3, 12, 3, 21, 12, 35), (12, 4, 6), (3, 1, 12), (4, 12, 3)$$

In the first phase, we have $k+1$ distinct values, and so we definitely have at least $k+1$ misses starting with an empty cache, no matter what the algorithm does. At the beginning of each subsequent phase, we know that the algorithm has in the cache the last request of the previous phase, and then we do not know what is in the remaining $h-1$ entries. We know, however, that we are going to see k distinct requests which are different from the last request, and so at least $k-h+1$ of them must be out of the cache and must cause a miss. So even an optimal algorithm causes at least $k-h+1$ misses per phase, compared with the k misses per phase of LRU, hence the competitive ratio.

(Note: we are glossing over the issue of what happens in the last phase, if the last phase has less than k distinct requests, in which case it could happen that the optimal algorithm has zero misses and LRU has a positive number of misses. In that case, we use the "surplus" that we have in the analysis about the first phase, in which the optimum algorithm and LRU have both k misses.) □

It can be proved that, if we knew the sequence of requests, then the optimal algorithm is to take out of the cache the element *whose next request is further in future*. The LRU algorithm is motivated by the heuristic that the element that has not been used for the longest time is likely to also not be needed for the longest time. It is remarkable, however, that such a heuristic works well even in a worst-case analysis.