Online Algorithms

Today we'll be looking at finding approximately-optimal solutions for problems where the difficulty is not that the problem is necessarily *computationally* hard but rather that the algorithm doesn't have all the *information* it needs to solve the problem up front.

Specifically, we will be looking at *online algorithms*, which are algorithms for settings where inputs or data is arriving over time, and we need to make decisions on the fly, without knowing what will happen in the future. This is as opposed to standard problems like sorting where you have all inputs at the start. Data structures are one example of online algorithms (they need to handle sequences of requests, and to do well without knowing in advance which requests will be arriving in the future).

Objectives of this lecture

In this lecture, we will

- define and motivate **online algorithms**
- solve the rent-or-buy problem with an online algorithm and analyze its performance
- analyze various strategies for the *list update* problem. In particular, we will see how *potential functions* are key ingredients in the analysis of online algorithms.
- analyzing online paging algorithms and see how *randomization* allows us to achieve provably better performance than any deterministic algorithm.

1 Framework and Definition

We are given a problem in which the input arrives over time rather than being known entirely up front. At each point in time, we have to make some decision, and each such decision is irrevocable, i.e., we can not change our mind later. Depending on the choices we make, we incur some cost, depending on the cost model of the problem. The goal is to perform well relative to an optimal *omniscient* algorithm, i.e., one that can predict the future and see the entire input in advance.

This is similar to the way in which we analyze approximation algorithms, by comparing the performance of our algorithm to that of the best possible algorithm (except that in this case, our definition of "best" is that it can cheat and see the future). We define the *competitive ratio* of an online algorithm very similarly to the approximation ratio.

Definition: Competitive Ratio

An online algorithm is called *c-competitive* if for all possible inputs σ

$$\text{ALG}(\sigma) \leq c \cdot \text{OPT}(\sigma),$$

where $\text{ALG}(\sigma)$ is the cost incurred by the online algorithm on the input σ (which it does not know in advance) and $\text{OPT}(\sigma)$ is the cost of an optimal omnipotent algorithm that can see σ in advance. The factor c is called the *competitive ratio* of the algorithm.

2 Rent or buy?

Here is a simple online problem that captures a common issue in online decision-making, called the rent-or-buy problem.

Problem: Rent or buy

Its the middle of the snow season and you are planning on going skiing. You can rent a pair of skis at $\$r$ per day, or buy a pair for $\$b$ and keep them forever. You would like to ski for as many days as possible, however, you do not know how many more days of the season will be viable weather for skiing. Each day you find out whether the weather is still good. At some point, you discover that the ski season is over. Your choice is to decide whether to rent or buy skis on each day, with the goal of minimizing the total amount of money that you spend.

Let's walk through a concrete example. You can either rent skis for \$50 or buy them for \$500. If we know the future in advance, the solution is to buy immediately if we know that there are at least 10 days of viable skiing weather, and if not, just always rent. The tricky part is designing an *online* algorithm that doesn't know the future. It has no idea how many days of viable whether there will be. Lets start with some simple but sub-optimal strategies to illustrate:

- **Always immediately buy:** One valid online strategy is to immediate buy on the first day if the weather is good. The worst case input for the this strategy is when we only get to go skiing once, so we could have just paid \$50, so the competitive ratio is $500/50 = 10$, we paid $10\times$ more than we could have.

  Rent forever: Another strategy is to never buy skis and just always rent. In this strategy, the worst case input is that the ski season goes on arbitrarily long, and we end up paying an arbitrarily high amount of money, when the optimal choice would have been to buy immediately, so the competitive ratio is actually ∞ (or unbounded).

In general, since after buying the skis the algorithm has no more decisions to make, we can characterize any online algorithm for the rent-or-buy problem by the day on which it decides to buy. Now observe that in general, the worst-case input for such an algorithm is that the weather is bad on the day after it buys the skis. With this in mind, here is one more bad strategy before we hone in on the optimal one.

- **Rent five times then buy:** How about we rent five times, then decide that it is time to buy. The worst-case input is that the weather is good for six days, then bad. In this case, our algorithm pays $5 \times 50 + 500 = 750$, but the optimal algorithm would just always rent, which costs $6 \times 50 = 300$, so this is 2.5-competitive.

Well that's certainly a lot better than 10. It seems like if we hedge our bets by renting longer, we get a better competitive ratio. At some point, this will stop being true, though. In particular, it never makes sense to plan to rent for more than 10 days, because then we should have just bought the skis for sure. So the most hedging we can do is to rent for 10 days then buy. This is called the *better-late-than-never* algorithm.

Algorithm: Better-late-than-never

We rent for $b/r - 1$ days[a], then we buy. In other words, we buy on day b/r.

[a] If r does not divide b, then we should rent for $\lfloor b/r \rfloor - 1$ days, but we will just assume that r divides b for simplicity

Theorem: Better-late-than-never is 2-competitive

Better-late-than-never is a 2-competitive algorithm for the rent-or-buy problem.

Proof. Suppose the weather is good for n days. We have to consider two cases:

1. If $nr < b$ (i.e., $n < b/r$), then the optimal solution is to always rent, but in this case, our algorithm doesn't buy either, so it is optimal.
2. If $nr \geq b$, the optimal solution buys immediately, but our algorithm first rents for $b/r - 1$ days before buying, so the ratio is

$$\frac{\left(\frac{b}{r} - 1\right) r + b}{b} = \frac{b - r + b}{b} = 2 - \frac{r}{b} \leq 2.$$

□

Now as we will naturally want to ask: can we do better?

Problem 1. Show that *better-late-than-never* has the best possible competitive ratio for the rent-or-buy problem for deterministic algorithms when b is a multiple of r.

3 List Update

This is a nice problem that illustrates some of the ideas one can use to analyze online algorithms. Here are the ground rules for the problem:

Problem: List Update

- We begin with a list of n items $1, 2, \ldots, n$. Imagine a linked list starting with 1 and ending with n.