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.

- An item x can be *accessed*. The operation is called $\texttt{Access}(x)$. The cost is the position of x.
- The algorithm can rearrange the list by swapping adjacent elements. The cost of a swap is 1.

So an on-line algorithm is specified by describing which swaps are done and when. The goal is to devise and analyze an on-line algorithm for doing all the accesses $\texttt{Access}(\sigma_1)$, $\texttt{Access}(\sigma_2)$, $\texttt{Access}(\sigma_3)$,... with a small competitive factor. Here are several algorithms to consider.

Problem 2. Do no swaps Consider an online algorithm that does no swaps. What is a worst-case input for this algorithm? What is the resulting competitive ratio?

Problem 3. Single exchange Consider an online algorithm that, for each accessed element, swaps it one position closer to the front of the list. What is a worst-case input for this algorithm? What is the resulting competitive ratio?

Problem 4. Frequency count Consider an online algorithm that remembers the frequency of each element being accessed, and keeps the list sorted by frequency (largest to smallest). What is a worst-case input for this algorithm? What is the resulting competitive ratio?

Okay, those were warm-ups, now its time for the real algorithm.

Algorithm: Move-to-front

After an access to an element x, do a series of swaps to move x to the front of the list.

Theorem 1: Competitive ratio of MTF

MTF is a 4-competitive algorithm for the list-update problem.

Proof. We'll use the potential function method. There will be a potential function that depends on the state of the MTF algorithm and the state of the "opponent" algorithm B, which can be any algorithm, even one which can see the future. Using this potential, we'll show that the amortized cost to MTF of an access is at most 4 times the cost of that access to B.

What is the potential function Φ? Define

$$\Phi_t = 2 \cdot (\text{The number of inversions between } B\text{'s list and MTF's list at time } t)$$

Recall that an inversion is a pair of distinct elements (x, y) that appear in one order in B's list and in a different order in MTF's list. It's a measure of the similarity of the lists.

We can first analyze the amortized cost to MTF of $\texttt{Access}(x)$ (where it pays for the list traversal and its swaps, but B only does its access). Then we separately analyze the amortized cost to MTF that is incurred when B does any swaps. (Note that in the latter case MTF incurs zero cost, but it will have a non-zero amortized cost, since the potential function may change. To be

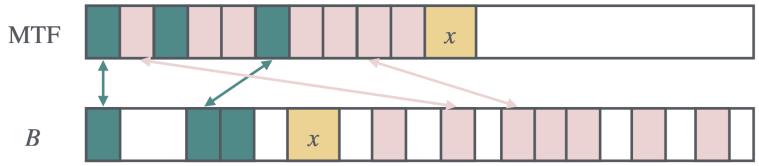
complete the analysis must take this into account.). In each case we'll show that the amortized cost to MTF (which is the actual cost, plus the increase in the potential) is at most 4 times the cost to B.

For any particular step, let C_{MTF} and C_B be the actual costs of MTF and B on this step, and $AC_{MTF} = C_{MTF} + \Delta\Phi$ be the amortized cost. Here $\Delta\Phi = \Phi_{new} - \Phi_{old}$ is the increase in Φ. Hence observe that $\Delta\Phi$ may be negative, and the amortized cost may be less than the actual cost. We want to show that

$$AC_{MTF} \leq 4 \cdot C_B$$

We can then sum the amortized costs, which would equal the actual cost of the entire sequence of accesses to MTF plus the final potential (non-negative) minus the initial potential (zero). This would be the four times total cost of B, which would give the result.

Analysis of `Access`(x). First look at what happens when MTF accesses x and brings it to the front of its list. Say the picture looks like this:



Look at the elements that lie before x in MTF's list, and partition them as follows:

$$S = \{y \mid y \text{ is before } x \text{ in MTF and } y \text{ is before } x \text{ in } B\}$$
$$T = \{y \mid y \text{ is before } x \text{ in MTF and } y \text{ is after } x \text{ in } B\}$$

In the above picture, S is dark/teal, and T is light/pink.

What is the cost of the access to MTF in terms of these sets?

$$C_{MTF} = 1 + \underbrace{|S| + |T|}_{\text{find cost}} + \underbrace{|S| + |T|}_{\text{swap cost}} = 1 + 2(|S| + |T|).$$

On the other hand, since all of S lies before x in B, the cost of the algorithm B is at least

$$C_B \geq |S| + 1.$$

What happens to the potential as a result of this operation? Well, here's MTF after the operation:

The only changes in the inversions involve element x, because all other pairs stay in the same relative order. Hence, for every element of S the the number of inversions increases by 1, and for every element of T the number of inversions decreases by 1. Hence the increase in Φ is precisely:

$$\Delta(\Phi) = 2 \times (|S| - |T|)$$

Now the amortized cost is

$$\begin{aligned} AC_{\text{MTF}} &= C_{MTF} + \Delta(\Phi) = 2(|S| - |T|) + 1 + 2(|S| + |T|) \\ &= 1 + 4|S| \leq 4(1 + |S|) \leq 4\, C_B \end{aligned}$$

This completes the amortized analysis of `Access`(x).

Analysis of *B* swapping. As explained above, we view B doing a swap as an event in its own right, not associated with any of the access events. For each such swap, observe that $C_{\text{MTF}} = 0$ and $C_B = 1$. Moreover, $\Delta(\Phi) \leq 2$, since the swap may introduce at most one new inversion. Hence,

$$AC_{MTF} \leq 2\, C_B \leq 4\, C_B$$

Putting the parts together. Summing the amortized costs we get:

$$\text{Total Cost to MTF} + \leq 4(\text{Total Cost to } B) + \Phi_{\text{init}} - \Phi_{\text{final}}$$

But $\Phi_{\text{init}} = 0$, since we start off with the same list as B. And $\Phi_{\text{final}} \geq 0$. Hence $\Phi_{\text{init}} - \Phi_{\text{final}} \leq 0$. Hence,

$$\text{Total Cost to MTF} \leq 4 \times (\text{Total Cost to } B).$$

Hence the MTF algorithm is 4-competitive. □

Key Idea: Using a potential function to analyze online algorithms

We saw potential functions earlier in the course for coming up with *amortized* cost bounds. Another powerful use of them is to analyze online algorithms. They help us relate the cost of our algorithm to the cost of the optimal omniscient algorithm, which is often very tricky to figure out without a potential function.

4 Paging

Our last problem shows up frequently in systems but also has relevance to efficient algorithms. Its called the *paging* problem.

Problem: Paging

In the paging problem, we say we have:

- a disk with N pages of memory,
- a *cache* with space for $k < N$ pages. Initially, the cache stores pages $1, 2, \ldots, k$.

We have to process a sequence of *requests*. When a request is made, if the page is in the cache, the request is free. If the page is not in the cache, we have a *page fault* (this is also called a *cache miss*). We then need to bring the page into the cache and throw something else out if the cache is full. The decision here is *which page* to throw out. Our goal / cost

model is to minimize the number of page faults.

The paging problem has several important applications in the area of computer systems. For example, operating systems have to manage virtual memory and decide how to map it onto physical memory (which is much smaller). CPU caches need to decide which cache lines to keep and which ones to throw out when they incur a cache miss, and many database systems store caches of recently/frequently accessed items to improve their performance. Essentially, any time you have a large amount of stuff but you want to make repeatedly accessing the same small amount of stuff much faster, you have the paging problem!

An online algorithm: LRU A standard online algorithm is the *least recently used* heuristic (LRU): "throw out the least recently used page". This is used commonly in practice because it is simple and performs reasonably well empirically. What's a bad case for LRU? What if the request sequence looks like 1,2,3,4,1,2,3,4,1,2,3,4... Notice that in this case, the algorithm makes a page fault every time and yet if we knew the future we could have thrown out a page whose next request was 3 time steps ahead. More generally, this type of example shows that the competitive ratio of LRU is at least k.

In fact, it's not hard to show that you can't do better than a competitive ratio of k with *any deterministic* algorithm.

Theorem 2

Any deterministic algorithm cannot have a competitive ratio better than k.

Proof. Set $N = k+1$ and consider a request sequence that always requests whichever page the algorithm *doesn't* have in its cache. By design, this will cause the algorithm to have a page fault every time. However, if we knew the future, every time we had a page fault we could always throw out the item whose next request is farthest in the future (instread of the least recently used page). Since there are k pages in our cache, for one of them, this next request has to be at least k time steps in the future, and since $N = k+1$, this means we won't have a page fault for at least $k-1$ more steps (until that one is requested). So, the algorithm that knows the future has a page fault at most once every k steps, and the ratio is k. □

We can also show that LRU can indeed achieve a competitive ratio of k. In other words, it is (one of) the optimal deterministic algorithm(s).

Theorem 3

LRU achieves a competitive ratio of k.

Proof. Define a phase to contain k distinct requests, and moreover, the next request is distinct from all requests in the phase. First, it is not hard to see that LRU incurs at most k page faults in each phase, and thus the total number of page faults is at most $k \cdot m$ where m is the number

of phases. Now, we show that any algorithm must pay a cost of at least $m-1$ for m phases. This shows a competitive ratio of k as m grows large. □

Claim 1

Any algorithm must incur a cost of $m-1$ for m phases.

Proof. For any arbitrary phase i, we can define an associated *offset phase* from the second request of phase i to the first request of phase $i+1$. We claim that any algorithm must incur at least one page fault in each offset phase. In particular, either the first request of phase $i+1$ incurs a page fault, or if it doesn't, then the first request of phase $i+1$ must reside in the cache throughout phase i. Further, when the first request of phase i is served, that page is fetched into the cache. Now, there still remain $k-1$ distinct pages in the offset chunk, and as there are only $k-1$ spots on the cache left, one of them must incur a page fault. There are $m-1$ such offset phases (from the second request of some phase i to the first request of the next phase $i+1$). Thus, the total number of page faults of any algorithm is at least $m-1$. □

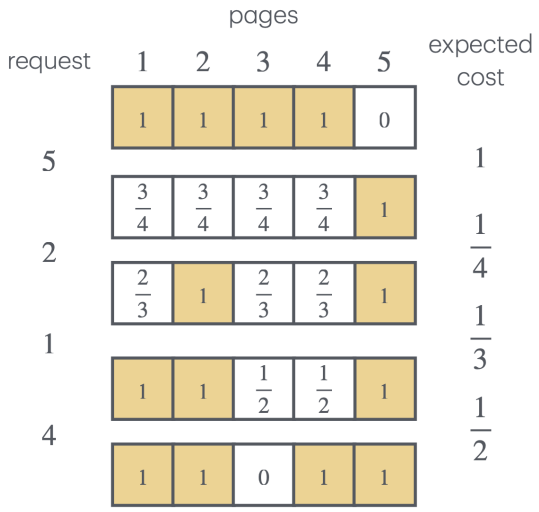
Does this mean that we cannot get any algorithms that are better than k-competitive? No! While deterministic algorithms have this limitation, we can use randomization to help us. Here is a neat *randomized* algorithm with a competitive ratio of $O(\log k)$.

Algorithm: Randomized marking algorithm

- Initiall. pages $1, 2, \ldots, k$ are in the cache. Start with these pages all marked.
- When a page is requested,
  - if it's in cache already, mark it and return.
  - Otherwise,
    - if every page is marked, unmark every page.
    - Throw out a random unmarked page, bring in the requested page, and mark it.

We can think of this as a 1-bit randomized LRU, where marks represent "recently used" vs "not recently used". For analysis purposes, we call the step in which every page is unmarked the beginning of a *phase*.

The figure below illustrates a phase of the algorithm for $N = 5, k = 4$. Each page (shown as a box) contains the probability that that page is in cache. Orange shading means that that page is marked. The column on the left shows the requested page, and the column on the right shows the expected cost of that request. The phase is comprised of accesses to k distinct pages. (Access to pages that are in the cache with probability 1, i.e., that have probability 0 of incurring a page fault, are not shown.) At the end of the phase, all marks are erased, and the next phase begins.



Theorem 4

When the marking algorithm is run on a sequence σ of accesses, we have:

$$\frac{\mathbb{E}[\text{MARKING}(\sigma)]}{\text{OPT}(\sigma)} \leq \begin{cases} H_k & \text{if } N = k+1 \\ 2H_k & \text{if } N > k+1 \end{cases}$$

Note that $H_k := 1 + 1/2 + \ldots + 1/k$ is the kth harmonic number. Recall that $H_k \leq 1 + \ln k$.

Proof. We will only show the proof for the special case of $N = k+1$. For general N, the proof follows similar lines but just is a bit more complicated.

Each phase contains k distinct page requests. The first time each of these pages are requested in the page they will be unmarked, since the first request of a phase will not be in the cache, and at that point the entire cache will be marked from the previous phase. So, we will unmark everything. Thus, the probabilities of these first accesses being to pages not in the cache is nonzero.

So, we can find the expected cost of each of these. The first request, as mentioned, always results in a page fault. Thus, it has a cost of 1.

For any other request $i \in \{2, 3, \ldots, k\}$, at that point there are $i-1$ marked pages in the cache (one for all previous requests). This leaves $N-i-1$ unmarked pages. One of these pages is out of the cache, and it is equally likely for the requested page to be any of them, so the expected cost of this request is

$$\frac{1}{N-i+1}.$$

Summing over all requests in a phase, we have that the total expected cost of a phase is

$$\begin{aligned}
1+\sum_{i=2}^{k}\frac{1}{N-i+1} &= 1+\sum_{i=0}^{k-2}\frac{1}{N-1-i} \\
&= 1+\sum_{i=0}^{k-2}\frac{1}{k-i} \\
&= 1+\sum_{2}^{k}\frac{1}{i} \\
&= H_k.
\end{aligned}$$

So, over m phases, the marking algorithm will incur mH_k expected cost in total. It suffices to show that any algorithm must incur at least m cost for m phases. This was proven in Claim 1. □