We now shift focus to a different kind of algorithmic problem where we need to perform some optimization without knowing the input in advance. Algorithms of this kind are called online algorithms. We wish to perform nearly as well as an optimal algorithm that knows the input in advance. Since the optimal algorithm we compare against has more power than our algorithm, technically the gap in the performance of our algorithm and the optimal performance is not an approximation factor. Instead we call this gap the "competitive ratio" of the algorithm. Just like approximation factors, competitive ratios are usually written as numbers greater than 1, and the closer they are to 1, the better the algorithm is. Note that the best possible competitive ratio for a problem may not be 1 even if we use unlimited computational power, the gap from 1 being due to the lack of information rather than the lack of computational power.

We illustrate these concepts through the rent-or-buy (a.k.a. ski rental) problem.

14.1 Rent-or-buy problem

In the rent or buy problem sometimes also called the ski rental problem we need to figure out whether to rent or buy skis. Suppose we go skiing every winter and thus need skis. We can either buy a pair for some cost B and then use them forever, or we can rent them every winter for cost R but then we will have to rent them again when we want to go skiing next year. If we know how many times we will go skiing in our lifetime it is easy to choose whether to buy or rent just by comparing the total cost of each option, but we do not actually know how many times we will go until it's far too late. Thus we need to come up with another solution. An easy algorithm is to just rent every winter until we have payed B in rent and then buy the skis once that point comes. While this algorithm is not the best in hindsight, the amount we pay is always within a factor of 2 from the optimal solution. Since if we go skiing fewer times than B/R then we have actually chosen the optimal course of action, and if not then the optimal solution is to buy the skis in the first place so the optimal cost is B and we have paid exactly $2B$ (assuming B/R is an integer). Note that the 2 here is not an approximation ratio it is something we call a competitive ratio. That is the ratio of our cost to the best cost we can get given that we would actually know everything in advance.

Definition 14.1.1 *Assume we have a problem and some algorithm* $\mathrm{A}LG$ *such that given an instance* I *of our problem* $\mathrm{A}LG(I)$ *gives the cost of the solution* $\mathrm{A}LG$ *comes up with. Furthermore assume* $\mathrm{O}PT(I)$ *is the cost of the best possible solution of the problem for instance* I*. Then*

$$\max_I \frac{\mathrm{A}LG(I)}{\mathrm{O}PT(I)}$$

is called the competitive ratio of the algorithm $\mathrm{A}LG$*.*

Sometimes we choose to ignore constant additive factors in the cost of the algorithm. This leads to the following alternative definition of competitive ratio. In this case, when the algorithm's cost involves no constant additive factor, we call the corresponding ratio a strong competitive ratio.

Definition 14.1.2 *Assume the same as above, and suppose that*

$$\mathrm{ALG}(I) \leq r\mathrm{O}PT(I) + c$$

holds for all I and some constant c independent of I. Then the competitive ratio of the algorithm is r. When c is 0, r is the strong competitive ratio.

Next we will study a more advanced online problem—caching.

14.2 Caching

In today's computers, we have to make tradeoffs between the amount of memory we have and the speed at which we can access it. We solve the problem by having a larger but slower *main memory*. Whenever we need data from some page of the main memory, we must bring it into a smaller section of fast memory called the *cache*. It may happen that the memory page we request is already in the cache. If this is the case, we say that we have a cache *hit*. Otherwise we have a cache *miss*, and we must go in the main memory to bring the requested page. It may happen that the cache is full when we do that, so it is necessary to *evict* some other page of memory from the cache and replace it with the page we read from main memory. Cache misses slow down programs because the program cannot continue executing until the requested page is fetched from the main memory. Managing the cache in a good way is, therefore, necessary in order for programs to run fast. The goal of a *caching algorithm* is to evict pages from the cache in a way that minimizes the number of cache misses. A similar problem, called *paging*, arises when we bring pages from a hard drive to the main memory. In this case, we can view main memory as a cache for the hard drive.

For the rest of this lecture, assume that the cache has a capacity of k pages and that main memory has a capacity of N pages. For example consider a cache with $k = 3$ pages and a main memory with $N = 5$ pages. A program could request page 4 from main memory, then page 1, then 2 etc. We call the sequence of memory pages requested by the program the *request sequence*. One request sequence could be 4, 1, 2, 1, 5, 3, 4, 4, 1, 2, 3 in our case. Initially, the cache could contain pages 1, 2, and 3. When we execute this program, we need access to page number 4. Since it's not in the cache, we have a cache miss. We could evict page 2 from the cache and replace it with page 4. Next, our program needs to access page 1, which is in the cache. Hence, we have a cache hit. Now our program needs page 2, which is not in the cache, so we have a miss. We could choose to evict page 1 from the cache and put page 2 in its place. Next, the program requests page 1, so we have another miss and the program brings 1 into the cache, evicting 4. We continue this way until all the memory requests are processed.

In general, we don't know what the next terms in the request sequence are going to be. Thus, caching is a place where we can try to apply an online algorithm. For a request sequence σ and a given algorithm ALG, we call ALG(σ) the *cost* of the algorithm ALG on request sequence σ, and define it to be the number of cache misses that happen when ALG processes the memory requests and maintains the cache.

14.3 Optimal Caching Algorithm

If we knew the entire request sequence before we started processing the memory requests, we could use a greedy algorithm that minimizes the number of cache misses that occur. Whenever we have a cache miss, we go to main memory to fetch the memory page p we need. Then we look at all the memory pages in the cache. We evict the page for which the next request occurs the latest in the future from among all the pages currently in the cache, and replace that page with p in the cache.

Once again, take our sample request sequence 4, 1, 2, 1, 5, 3, 4, 4, 1, 2, 3 and assume that pages 1 through 3 are in the cache. The optimal algorithm has a cache miss when page 4 is requested. The algorithm brings page 4 into the cache and must evict one of the first three pages. Since 3 is accessed last in this subset of pages, it is evicted from the cache. The next miss occurs when page 5 is requested. This time, of pages 1, 2, and 4, 2 is accessed last, so the algorithm replaces page 2 by page 5. The next cache miss occurs when page 3 is requested. The algorithm evicts page 5 which is never requested again, and brings in 3. The cache at this point contains pages 1, 3, and 4. The final cache miss occurs when page 2 is requested. At this point the algorithm evicts either one of 1 and 4 and brings in page 2. There are no more cache misses. Thus, the optimal algorithm has 4 cache misses on this request sequence.

14.4 Deterministic Caching

There are many deterministic online algorithms for caching. We give some examples below. In each of the cases, when a cache miss occurs, the new memory page is brought into the cache. The name of the algorithm suggests which page should be evicted from the cache if the cache is full.

LRU (Least Recently Used) The page that has been in the cache for the longest time without being used gets evicted.

FIFO (First In First Out) The cache works like a queue. We evict the page that's at the head of the queue and then enqueue the new page that was brought into the cache.

LFU (Least Frequently Used) The page that has been used the least from among all the pages in the cache gets evicted.

LIFO (Last In First Out) The cache works like a stack. We evict the page that's on the top of the stack and then push the new page that was brought in the cache on the stack.

The first two algorithms, LRU and FIFO have a competitive ratio of k where k is the size of the cache. The last two, LFU and LIFO have an *unbounded competitive ratio*. This means that the competitive ratio is not bounded in terms of the parameters of the problem (in our case k and N), but rather by the size of the input (in our case the length of the request sequence).

First we show that LFU and LIFO have unbounded competitive ratios. Suppose we have a cache of size k. The cache initially contains pages 1 through k. Also suppose that the number of pages of main memory is $N > k$. Suppose that the last page loaded in the cache was k, and consider the request sequence $\sigma = k + 1, k, k + 1, k, \ldots, k + 1, k$. Since k is the last page that was put in

the cache, it will be evicted and replaced with page $k+1$. The next request is for page k (which is not in the cache), so we have a cache miss. We bring k in the cache and evict $k+1$ because it was brought in the cache last. This continues until the entire request sequence is processed. We have a cache miss for each request in σ, whereas we have only one cache miss if we use the optimal algorithm. This cache miss occurs when we bring page $k+1$ at the beginning and evict page 1. There are no cache misses after that. Hence, LIFO has an unbounded competitive ratio.

To demonstrate the unbounded competitive ratio of LFU, we again start with the cache filled with pages 1 through k. First we request each of the pages 1 through $k-1$ m times. After that we request page $k+1$, then k, and alternate them m times. This gives us $2m$ cache misses because each time k is requested, $k+1$ will be the least frequently used page in the cache so it will get evicted, and vice versa. Notice that on the same request sequence, the optimal algorithm makes only one cache miss. This miss occurs during the first request for page $k+1$. At that point, the optimum algorithm evics page 1 and doesn't suffer any cache misses afterwards. Thus, if we make m large, we can get any competitive ratio we want. This shows that LFU has an unbounded competitive ratio.

We now show that no deterministic algorithm can have a better competitive ratio than the size of the cache, k. After that, we demonstrate that the LRU algorithm has this competitive ratio.

Claim 14.4.1 *No deterministic online algorithm for caching can achieve a better competitive ratio than k, where k is the size of the cache.*

Proof: Let ALG be a deterministic online algorithm for caching. Suppose the cache has size k and that it currently contains pages 1 through k. Suppose that $N > k$. Since we know the replacement policy of ALG, we can construct an adversary that causes ALG to have a cache miss for every element of the request sequence. To do that, we simply look at the contents of the cache at any time and make a request for the page in $\{1, 2, \ldots, k+1\}$ that is currently not in the cache.

The only page numbers requested by the adversary are 1 through $k+1$. Thus when the optimal algorithm makes a cache miss, the page it evicts will be requested no sooner than after at least $k-1$ other requests. Those requests will be for pages in the cache. Thus, another miss will occur after at least $k-1$ memory requests. It follows that for every cache miss the optimal algorithm makes, ALG makes at least k cache misses, which means that the competitive ratio of ALG is at least k. ■

Claim 14.4.2 *LRU has a competitive ratio of k.*

Proof: First, we divide the request sequence σ into phases as follows:

- Phase 1 begins at the first page of σ;
- Phase i begins at the first time we see the k-th distinct page after phase $i-1$ has begun.

As an example, suppose $\sigma = 4, 1, 2, 1, 5, 3, 4, 4, 1, 2, 3$ and $k = 3$. We divide σ into three phases as in Figure 14.4.1 below. Phase 2 begins at page 5 since page 5 is the third distinct page after phase 1 began (pages 1 and 2 are the first and second distinct pages, respectively).

Next, we show that OPT makes at least one cache miss each time a new phase begins. Denote the j-th distinct page in phase i as p^i_j. Consider pages p^i_2-p^i_k and page p^{i+1}_1. These are k distinct pages

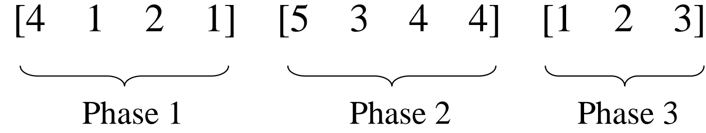

Figure 14.4.1: Three phases of sequence σ.

by the definition of a phase. Then if none of the pages p_2^i-p_k^i incur a cache miss, p_1^{i+1} must incur one. This is because pages p_2^i-p_k^i and p_1^{i+1} are k distinct pages, page p_1^i is in the cache, and only k pages can reside in the cache. Let M be the number of phases. Then we have $\mathrm{OPT}(\sigma) \geq M - 1$. On the other hand, LRU makes at most k misses per phase. Thus $\mathrm{LRU}(\sigma) \leq kM$. As a result, LRU has a competitive ratio of k. ■

14.4.1 Deterministic 1-Bit LRU

As a variant of the above LRU algorithm, the 1-bit LRU algorithm (also known as the marking algorithm) associates each page in the cache with 1 bit. If a cache page is recently used, the corresponding bit value is 1 (marked); otherwise, the bit value is 0 (unmarked). The algorithm works as follows:

- Initially, all cache pages are unmarked;
- Whenever a page is requested:
    - If the page is in the cache, mark the page;
    - Otherwise:
        - If there is at least one unmarked page in the cache, evict an arbitrary unmarked page, bring the requested page in, and mark it;
        - Otherwise, unmark all the pages and start a new phase.

Following the proof of Claim 14.4.2, we can easily see that the above deterministic 1-bit LRU algorithm also has a competitive ratio of k.

14.5 Randomized 1-Bit LRU

Given a deterministic LRU algorithm, an adversary can come up with a worst-case scenario by always requesting the page which was just evicted. As a way to improve the competitive ratio, we consider a randomized 1-bit LRU algorithm in which a page is evicted randomly. Before we go into the algorithm, we shall define the competitive ratio for a randomized online algorithm.

Definition 14.5.1 *A randomized online algorithm* ALG *has a* competitive ratio r *if for* $\forall\sigma$*,*

$$E[\mathrm{ALG}(\sigma)] \leq r \cdot \mathrm{OPT}(\sigma) + c,$$

where c is a constant independent of σ.

Note that the competitive ratio of a randomized algorithm is defined over the expected performance of the algorithm rather than the worst-case performance.

The randomized 1-bit LRU algorithm is rather simple: at every step, run the 1-bit LRU algorithm and evict an unmarked page uniformly at random when necessary.

Claim 14.5.2 *The randomized 1-bit LRU algorithm has a competitive ratio of $2H_k$, where H_k is the k-th harmonic number, which is defined as*

$$H_k = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{k} \in (\log k, \log(k) + 1).$$

Proof: Let us first analyze the expected number of cache misses in phase i. Recall that at the end of phase $i-1$, there are k pages in the cache. Let m_i denote the number of "new" pages in phase i, *i.e.*, those pages that were not requested in phase $i-1$. Call the rest of the pages in phase i "old" pages. Requesting a new page causes an eviction while requesting an old page may not. Assume for simplicity that the m_i new pages appear the first in the request sequence of phase i. Actually, one should convince oneself that this assumption indicates the worst-case scenario (think about how the number of cache misses changes if any one of these new page requests is delayed). Once the m_i new pages are requested, they are marked in the cache. The remaining $k - m_i$ pages in the cache are old pages. Now let us consider the probability of a cache miss when we request an old page for the first time, *i.e.*, when we request the (m_i+1)-th distinct page in phase i. This old page by definition was in the cache at the end of phase $i-1$. When requested, however, it may not be in the cache since it can be evicted for caching one of the m_i new pages. Then,

$$Pr[(m_i+1)\text{-th distinct page suffers a cache miss}] = \frac{\binom{k-1}{k-m_i}}{\binom{k}{k-m_i}} = \frac{m_i}{k}.$$

By the same reasoning, we have

$$Pr[(m_i+2)\text{-th distinct page suffers a cache miss}] = \frac{\binom{k-2}{k-m_i-1}}{\binom{k-1}{k-m_i-1}} = \frac{m_i}{k-1},$$

and so on.

Then the expected total number of cache misses in phase i is at most

$$m_i + \frac{m_i}{k} + \frac{m_i}{k-1} + \cdots + \frac{m_i}{k-(k-m_i)+1} \le m_i H_k.$$

Let M be the number of phases. Then

$$E[\text{RAND-1-BIT-LRU}(\sigma)] \le H_k \sum_{i=1}^{M} m_i. \tag{14.5.1}$$

Let us now analyze the number of cache misses of $\text{OPT}(\sigma)$. Note that requesting a new page p in phase i may not cause an eviction. This is because OPT may choose to fix p in the cache when p

was requested in some earlier phase. In contrast, the randomized algorithm unmarks all pages in the cache at the beginning of phase i, and evicts randomly an unmarked page in case of a cache miss. Therefore, at the end of phase i, p cannot be in the cache. Although we cannot bound the number of cache misses in a single phase, we can bound the number in two consecutive phases. Let n_i be the number of cache misses of OPT in phase i. Since in total there are $k + m_i$ distinct pages in phases $i-1$ and i, at least m_i of them cause a miss. Thus we have

$$n_{i-1} + n_i \geq m_i,$$

and

$$\mathrm{OPT}(\sigma) \geq \frac{1}{2} \sum_{i=1}^{M} m_i. \tag{14.5.2}$$

Combining Eq. (14.5.1) and Eq. (14.5.2), we have

$$E[\text{RAND-1-BIT-LRU}(\sigma)] \leq 2H_k \times \mathrm{OPT}(\sigma).$$

In other words, the randomized 1-bit LRU algorithm is $2H_k$-competitive. ■

The guarantee achieved by 1-bit LRU turns out to be the best possible for caching. We state the following lemma without proof.

Lemma 14.5.3 *No randomized online algorithm has a competitive ratio better than H_k.*

14.6 Online learning and the weighted majority algorithm

We now consider an online problem with applications to machine learning. The stereotypical problem in machine learning is to train a machine for some task based on examples of that task, so that the machine can function correctly in the future (without knowing what specific instances of the task it would need to solve in the future of course). This is naturally modeled as an online problem. For example, you might train a robot to understand speech by giving it examples of real speech.

While we will not directly study algorithms for machine learning, we will look at a related problem – suppose that we had a number of different algorithms for solving a problem, some better at certain instances of the problem than others, then how do we decide which algorithm to use? Here's a hypothetical example of such a problem, that we will call the prediction problem. Our goal is to predict every morning whether or not it will rain that day. There are n sources that we can consult to make our prediction. We will call them the experts. We would like to always follow the best expert, however, we do not know who that is. Instead, we can observe the performance of the experts and over time determine which one to follow. Suppose further that the experts collude with each other to make our performance bad. Could we ensure that our performance is still nearly as good as that of the best expert?

Formally, on day t, each expert makes a prediction $p_{i,t} \in \{-1, 1\}$, where -1 denotes no rain and 1 denotes rain. Based on these, the algorithm must decide whether to predict -1 or 1. At the end of the day, we find out whether the prediction was correct or not. The cost of the algorithm is the

total number of mistakes it makes. Our goal is to bound the number of mistakes against those of the best expert.

The weighted majority algorithm. The weighted majority algorithm is simple to describe. It maintains a collection of weights, one for each expert. Then, it takes a weighted "vote" over the experts using these weights to figure out the prediction for any given day. At the end of the day the algorithm reduces the weight of every expert that made a mistake that day.

In particular, let $w_{i,t}$ denote the weight of expert i on day t. Let $\ell_{i,t}$ denote the "loss" of expert i on day t. $\ell_{i,t}$ is 1 if the expert makes a mistake and 0 otherwise.

The weighted majority algorithm runs as follows.

1. Initialize $w_{i,0} = 1$ for all i.
2. On day t, do:
   (a) If $\sum_i w_{i,t} p_{i,t} \geq 0$ predict "rain".
   (b) Otherwise, $\sum_i w_{i,t} p_{i,t} < 0$, so predict "no rain".
   (c) For all experts i that make a mistake, set their new weights to half of their old weights. That is, $w_{i,t+1} = w_{i,t}(1/2)^{\ell_{i,t}}$ for all i.

Analysis. Our goal is to show that we don't make many more mistakes than the best expert. In other words, if we make many mistakes, then so does the best expert. To prove this, we show two things. One, if we make too many mistakes, then the sum of the weights $\sum_i w_{i,t}$ decreases rapidly over time. On the other hand, if the best expert does not make many mistakes, then its weight cannot be too low. These two together give us a contradiction.

Let $W_t = \sum_i w_{i,t}$ be the total weight on day t. Let us analyze the weight of the best expert first. If the expert makes L^* mistakes in total over T days, then its final weight is $(1/2)^{L^*}$. So

$$W_T \geq \text{ weight of best expert } \geq (1/2)^{L^*}$$

On the other hand, if our algorithm makes a mistake on day t, then since we followed the majority vote, the total weight of experts that made a mistake that same day is at least half the total weight W_t. Since we divide the weight of all of these experts by a factor of 2, we have that $w_{t+1} \leq 3/4W_t$.

Therefore, if we made m mistakes over T days, then

$$W_T \leq W_0(3/4)^m$$

Putting these together and noting that $W_0 = n$, we get that

$$(1/2)^{L^*} \leq n(3/4)^m$$

Taking logs,

$$L^* \geq -\log_2 n + m \log_2(4/3)$$

$$\implies m \le 2.41(L^* + \log_2 n)$$

This means that as long as L^* is large enough compared to $\log n$, we are within a constant factor of L^*.

Can we do any better? Indeed. If instead of dividing the weight of every expert that made a mistake by 2, we were less agressive and multiplied each weight by $1 - \epsilon$ for some $\epsilon > 0$, then we can achieve the following better guarantee:

$$m \le (1 + \epsilon)L^* + O\left(\frac{1}{\epsilon}\log n\right)$$

This analysis works even if the "losses" $\ell_{i,t}$ are $[0, 1]$ variables, and not just $\{0, 1\}$ variables.