Algorithms I May 21, 2025

Lecture 26: Online Algorithms

Notes by Ola Svensson[1]

The first two sections are basically a verbatim copy of the notes [1]. The third section is based on the notes [2].

In the study of algorithms, we often assume that the entire input is available at the outset. Online algorithms, however, address a different paradigm where the input arrives sequentially, piece by piece. The core challenge in this "online" setting is the necessity of making irrevocable decisions at each step, based only on the data observed thus far and without knowledge of future inputs. The objective remains to optimize a specific goal, but this must be achieved by committing to decisions incrementally as the input stream unfolds. A prominent real-world application illustrating this concept is ad allocation: when a user performs a search query, an algorithm must instantaneously decide which advertisements to display, without foreseeing subsequent search queries or user interactions. We will now explore a foundational example to further illustrate these principles.

1 Rent or buy (ski rental) and competitive ratios

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 doesn't seem very smart 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 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 1 *Assume we have an online problem and some algorithm Alg such that given an instance I of our minimization problem $ALG(I)$ gives the cost of the solution Alg comes up with. Furthermore, assume that $OPT(I)$ is the cost of the best possible solution of the problem for instance I. Then*

$$\max_I \frac{ALG(I)}{OPT(I)}$$

is called the competitive ratio *of the algorithm Alg. (For a maximization problem, we are interested in the ratio* $\min_I \frac{ALG(I)}{OPT(I)}$ *just as in approximation algorithms.)*

There is also the following alternative definition where the constant c is so as to allow some initial start up cost:

Definition 2 *Assume the same as above, then we also call r the* competitive ratio *if*

$$ALG(I) \leq r \cdot OPT(I) + c$$

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

[1] **Disclaimer:** These notes were written as notes for the lecturer. They have not been peer-reviewed and may contain inconsistent notation, typos, and omit citations of relevant works.

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, or we can choose not to place the page we brought from main memory in the cache. 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. This time we could decide to just read page 1 and not place it in the cache at all. 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 $\mathrm{ALG}(I)$ 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. Similarly, we let $\mathrm{OPT}(I)$ denote the minimal number of cache misses possible for the sequence σ. Our goal is to design an r-competitive algorithm for a small r. Recall that an online algorithm Alg has a competitive ratio r if for all σ,

$$\mathrm{ALG}(I) \leq r \cdot \mathrm{OPT}(I) + c\,,$$

where c is a constant independent of σ (usually an unimportant startup cost).

2.1 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 this memory page and 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. If the next time p is requested comes after the next time all the pages in the cache are requested again, we don't put 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. Since page 4 is requested again later than pages 1 through 3 are, the algorithm doesn't put 4 in the cache and doesn't evict any page. The next miss occurs when page 5 is requested. Once again,

the next times pages 1 through 3 are requested occur before page 5 is requested the next time, so page 5 is not brought in the cache. The next cache miss happens when page 4 is requested. At that point, page 3 gets evicted from the cache and is replaced with page 4 because pages $1, 2$, and 4 get requested again before page 3 does. Finally, the last cache miss occurs when page 3 is requested (last term in the request sequence). At this point, the algorithm could choose to evict page 1 and put page 3 in the cache instead. Thus, the optimal algorithm has 4 cache misses on this request sequence.

Notice that in the case of caching, we have an optimal algorithm assuming that we know the entire input (in our case the input is the request sequence). This fact makes the analysis of online algorithms for caching easier because we know what algorithm we compare the online algorithms against.

For the remainder of this lecture, we will **assume that a caching algorithm always has to bring the memory page in the cache on a cache miss**, and doesn't have the option of just looking at it and not putting it in the cache, as it will make our proofs simpler. This version is also much closer to what happens in hardware.

2.2 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 evicts 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.

Lemma 3 *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 $\{1, 2, ..., k+1\}$ 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 other requests. Those requests will be for pages in the cache. Thus, another miss will occur after at least k 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. ■

Lemma 4 *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 follows:

$$\underbrace{[4\ 1\ 2\ 1]}_{\text{Phase 1}} \quad \underbrace{[5\ 3\ 4\ 4]}_{\text{Phase 2}} \quad \underbrace{[1\ 2\ 3]}_{\text{Phase 3}}$$

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(I) makes at least one cache miss each time a new phase begins. Denote the j-th distinct page in phase i as p_j^i. Consider pages $p_2^i - p_k^i$ and page p_1^{i+1}. These are k distinct pages 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 $p_2^i - p_k^i$ and p_1^{i+1} are k distinct pages, page p_1^i is in the case, and only k pages can reside in the cache. Let N be the number of phases. Then we have $\text{OPT}(I) \geq N - 1$. On the other hand, LRU makes at most k misses per phase. Thus $\text{LRU}(\sigma) \leq kN$. As a result, LRU has a competitive ratio of k. ■

2.3 Randomized Caching

Consider the following randomized strategy (also known as the marking algorithm). The strategy associates each page in the cache with 1 bit. Similar to LRU: 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 pages are unmarked.
- Whenever a page is requested

- If the page is in the cache, mark the page;
- Otherwise:
  - If there is an unmarked page in the cache, evict an unmarked page chosen uniformly at random, bring the requested page in, and mark it;
  - Otherwise, unmark all the pages and start a new phase.

One can prove the following lemma (see [1]):

Lemma 5 *The above strategy achieves a competitive ratio of* $2H_k = 2 \cdot \left(\frac{1}{1} + \frac{1}{2} + \ldots + \frac{1}{k}\right) = O(\log k)$.

The above is almost tight:

Lemma 6 *No randomized online algorithm has a competitive ratio better than* H_k.

3 Secretary Problem

In the previous examples, we assumed that we had *no* information about the future when designing the algorithms and we considered the *worst case* competitive ratio. In many settings this can be seen to be overly pessimistic. A model that gives more power to the algorithm is to assume that the items arrive in a random order the so-called random arrival model. In this setting, a classic problem is called the secretary problem:

- n candidates arrive in a random order (each with a unique ranking).
- When a candidate arrives you need take an irrevocable decision to hire the candidate or not (and at most one candidate can be hired).

The goal is to design a strategy to maximize the probability (over the random order of arrivals) of hiring the best candidate.

3.1 Simple strategies

We consider two basic strategies (one bad and one good):

1. Selecting the first candidate. For this strategy
$$\Pr[\text{hiring best candidate}] = 1/n\,.$$
2. Exploring the first $n/2$ candidates, then take the first candidate that is better than the first $n/2$. For this strategy we have
$$\begin{aligned}\Pr[\text{hiring best candidate}] &\geq \Pr[\text{second best among first } n/2] \cdot \Pr[\text{best candidate in 2nd half}] \\ &\geq (1/2)(1/2) = 1/4\,.\end{aligned}$$

3.2 Optimal strategy

The optimal strategy is in fact very similar to the second strategy above but instead of starting to select after observing $n/2$ candidates we start selecting after $r-1$ candidates. Then

$$\begin{aligned}
\Pr[\text{hiring best candidate}] &= \sum_{i=1}^{n} \Pr[\text{selecting i} \bigcap i \text{ is the best}] \\
&= \sum_{i=1}^{n} \Pr[\text{selecting i} | i \text{ is the best}] \Pr[i \text{ is the best}] \\
&= \sum_{i=1}^{r-1} 0 + \frac{1}{n} \sum_{i=r}^{n} \Pr[\text{second best of the first } i \text{ applicants is in the first } r-1 | i \text{ is the best}] \\
&= \frac{1}{n} \sum_{i=r}^{n} \frac{r-1}{i-1} \\
&= \frac{r-1}{n} \sum_{i=r}^{n} \frac{1}{i-1} \,.
\end{aligned}$$

We now optimize the selection of r to maximize the above expression. In other words we need to select r so as to maximize $(r-1)/n \cdot \sum_{i=r}^{n} \frac{1}{i-1}$. Between friends (for large enough r), we have

$$\frac{r-1}{n} \cdot \sum_{i=r}^{n} \frac{1}{i-1} \approx \frac{r}{n} \int_{r}^{n} \frac{1}{x} dx = \frac{r}{n} \cdot \ln(n/r) \,,$$

whose maximum is attained when $r = n/e$. This gives a probability of selecting the best candidate to be at least $1/e$ which turns out to be optimal!

References

[1] Shuchi Chawla, Yiying Zhang, Dan Rosendorf: *Scribes of Lecture 19 in Advanced Algorithms, University of Wisconsin, 2007.*
http://pages.cs.wisc.edu/˜shuchi/courses/787-F07/scribe-notes/lecture19.pdf

[2] Ashish Goel, Raghav Ramesh, Reza Zadeh: *Scribes of Lecture 8 in Algorithms for Modern Data Models, Stanford, 2014.*
https://stanford.edu/˜rezab/amdm/notes/lecture8.pdf