

## Lecture 9: Online Algorithms Part I

# 1 Online Problem Examples

## 1.1 Canadian Traveler's Problem

It is winter in Canada. A Canadian traveler has a road map and must get from point A to point B, but some sections of road are blocked by snow.

In the offline version of this problem, the traveler has full information of which road sections are blocked. Thus, a traditional shortest path algorithm can be employed to determine an optimal path from point A to point B.

In the online version of the problem, at each intersection, the traveler must decide which way to go with awareness only of a given road section being blocked by snow if he/she has already passed through an endpoint of said road section.

## 1.2 Ski Rental Problem

It costs \$1/day to rent, \$ $T$  to buy. After  $S$  days, you will break your leg and never ski again.

In the offline version of this problem, you know the value of  $S$  prior to your first day of skiing. As a result the decision is easy (if  $T \leq S$  then buy, if  $T > S$ , rent every day until you break your leg, and if  $S = T$  apply either strategy).

The online version of this problem is not so trivial. In this version of the problem we do not know  $S$  in advance. Observe that any deterministic strategy we apply can be described as follows: we will rent for  $k$  days, and then buy on day  $k + 1$  if we have not broken our leg by then.

### 1.3 Paging Problem

We are allowed  $k$  pages of data to be cached in "fast" memory storage and all other data pages must be stored in slow memory and swapped into the fast memory if and when they are requested. The user requests a sequence of pages  $\langle \sigma \rangle$ .

If page  $\sigma_i$  is in fast memory, the cost for retrieving it is zero, and if  $\sigma_i$  is not in fast memory, then  $\sigma_i$  must be swapped in, and some other page  $\sigma_j$  must be removed to create space for it, and the retrieval cost is 1. The latter case is referred to as a *page fault*. The cost of an algorithm on a given sequence  $\langle \sigma \rangle$  is simply the number of page faults.

In the offline formulation of this problem, we know the values of our sequence in advance. In the online formulation, we only learn what pages will be requested as the requests arrive; that is we learn  $\sigma_i$ 's value at time  $i$ , and need to base our decision on knowledge of only the values of  $\sigma_1, \sigma_2, \dots, \sigma_i$  without knowing the values of  $\sigma_{i+1}, \sigma_{i+2}, \dots, \sigma_n$ .

## 2 Measuring Online Algorithm Performance

How do we measure performance of online algorithms?

### 2.1 Competitive Analysis

Competitive analysis measures how well an online algorithm performs versus the optimal offline strategy.

**Definition 2.1.1** *An online algorithm  $A$  is said to be  $\alpha$ -competitive if for any input sequence  $\langle \sigma \rangle$ ,  $COST_A(\sigma) \leq \alpha COST_{MIN}(\sigma)$ , where  $COST_A(\sigma)$  denotes cost of online algorithm  $A$  on  $\langle \sigma \rangle$  and  $COST_{MIN}(\sigma)$  denotes the cost of the optimal offline algorithm on  $\langle \sigma \rangle$ .*

## 2.2 Ski Rental Problem

For this problem, there exists an optimal offline algorithm  $MIN$ , which is to simply buy if and only if  $S \geq T$ . Thus we either rent for  $S$  days and spend  $\$S$ , or we buy immediately and spend  $\$T$ . Hence the value of  $MIN$  on any given input is simply  $\min(S, T)$ . Let us consider two possible online algorithms for the ski rental problem.

Define simple algorithm  $A_1$  as follows:

- Buy on the very first day.

Regardless of the value of  $S$ ,  $A_1$  will spend  $T$ . Note that if  $S \geq T$  then  $MIN$  would have elected to buy as well, and the two algorithms have made an identical decision.  $A_1$  will only deviate from  $MIN$  when  $S < T$ . The worst case scenario is that we break our leg the very first day. In this case  $MIN$  provides the optimal solution of  $\$1$ , while  $A_1$  led to a cost of  $\$T$ . Thus  $A_1$ , in the worst-case, is a factor of  $T$  greater than  $MIN$ . By our definition above, we may say that  $A_1$  is  $T$ -competitive.

Define simple algorithm  $A_2$  as follows:

- Rent for the first  $T - 1$  days.
- If we still have not broken our leg by day  $T$ , then buy on day  $T$ .

If  $S \leq T - 1$ , the result of  $A_2$  matches the result of  $MIN$ . If we break our leg after day  $T$ , then  $A_2$  spent  $2 * T - 1 + k$  while  $MIN$  spent  $T + k$ , where  $k = S - T$ ; the number of days that follow  $T$  before the day on which our leg is broken. Since  $2T - 1 + k \leq 2(T + k)$ , this algorithm is 2-competitive. Observe that this inequality becomes tightest when  $S = T$  (and thus  $k = 0$ ); in this worst-case we have  $2T - 1 \leq 2T$ .

## 2.3 Paging Problem

The optimal offline algorithm knows the future, and applies a strategy which we will refer to as Longest Forward Distance ( $LFD$ ). At any given time

$i$ , *LFD* will remove either a page that is never again requested within the sequence if such a page exists, otherwise it will remove the page that is requested again the farthest into the future.

There are many possible online algorithm strategies:

- Least Recently Used (*LRU*). This algorithm will keep track of how long it has been since each page has been accessed, and will swap out the least-recently used page.
- First-In First-Out (*FIFO*). This algorithm will always remove the page that has been in memory for the greatest amount of time.
- Last-In First-Out (*LIFO*), This algorithm will always remove the page that has been in memory for the least amount of time.
- Random Selection. This algorithm selects a page to remove at random.

The listing above is not comprehensive; there are many other paging algorithms as well.

*LIFO* can be shown to perform very poorly. Assume  $k$  page slots and consider the following sequence:  $\langle \sigma_1, \dots, \sigma_{k-1}, p, q, p, q, p, q, \dots \rangle$ , where  $\sigma_i \neq p, \sigma_i \neq q, \forall i \in [1..k-1]$ .

In the above sequence, *LIFO* will choose to swap out  $p$  to make room for  $q$ , then it will swap out  $q$  to make room for  $p$ , etc, for as long a sequence as we choose to construct, ensuring that there is no bound on the worst-case scenario of our competitive analysis.

We will compare *LRU* to *LFD*, but first we must demonstrate that *LFD* is in fact an optimal offline algorithm.

**Theorem 2.3.1** *LFD is an optimal algorithm for the offline paging problem.*

**Proof 2.3.2** *Consider a finite sequence of page requests. Suppose  $\exists$  algorithm  $A$  that produces fewer page faults than *LFD* on some input sequence. In order to produce a different result, at some unique point in time  $A$  and*

*LFD must choose to do something different. Specifically,  $\exists i$  such that  $\sigma_i$  is not in fast memory and  $A$  removes some page  $p$  and  $LFD$  removes page  $q$ , with  $p \neq q$ .*

*Define  $t$  as the first time at which algorithm  $A$  removes page  $q$  after time  $i$ .*

*Let time  $j$  be the time at which the next request for  $p$  after time  $i$  occurs, and let  $l$  be the time at which the next request for  $q$  after time  $i$  occurs. Note that By definition of  $LFD$ ,  $i < j < l$  (otherwise  $LFD$  would have removed  $q$  at time  $j$ , not  $p$ ).*

*Now we define algorithm  $A_1$  which is identical to  $A$ , except at time  $i$   $A_1$  removes  $q$  instead of  $p$  (mimicking  $LFD$ ); and at time  $t$   $A_1$  removes  $p$  (assuming  $\sigma_i \neq p$ ; if  $\sigma_i = p$  then  $A_1$  would do nothing at time  $t$  as there would be no page fault). Our analysis can now be divided into two cases:*

- *Case 1: if  $t < j$ , then  $COST_{A_1}(\sigma) = COST_A(\sigma)$ , both have a page fault at time  $j$  when requesting  $p$ .*
- *Case 2: if  $j \leq t < l$ ,  $A$  faults at time  $t$  and removes  $q$ , while  $A_1$  does not incur a page fault. Thus after  $t$ ,  $COST_{A_1}(\sigma) \leq COST_A(\sigma) + 1$ , that is,  $A_1$  suffers one fewer page faults than  $A$ .*

*Thus,  $A_1$  is either same or better performance-wise than  $A$ , but the page fault sets of  $A_1$  and  $LFD$  are identical at least to time  $t$  which is strictly greater than  $i$ . This argument can be repeated iteratively to generate an algorithm  $A_{i+1}$  from  $A_i$ , where  $A_{i+1}$  matches  $LFD$  for a strictly greater length of time than  $A_i \forall i$ . Since the sequence has finite length, it must be the case that eventually, we will derive algorithm  $A_n$  which matches  $LFD$  for the entire sequence which has the same or better performance than  $A$ . More precisely we have  $COST_{LFD}(\sigma) = COST_{A_n}(\sigma) \leq COST_{A_{n-1}}(\sigma) \leq \dots \leq COST_{A_1}(\sigma) \leq COST_{LFD}(\sigma)$ , which a contradiction.*

*We may thus conclude  $\neg \exists$  any algorithm  $A$  which outperforms  $LFD$ .*

**Claim 2.3.3** *Least-Recently-Used online algorithm (LRU) is  $k$ -competitive.*

**Proof 2.3.4** *Assume an input sequence with non-trivial quantity of distinct pages ( $k+1$  or more). First observe that both algorithms start with at least  $k$*

successes (to fill up the initial  $k$  slots). We want to show that  $COST_{LRU}(\sigma) \leq k * COST_{LFD}(\sigma)$ .

$$\text{Let } \langle \sigma \rangle = \sigma_1, \sigma_2, \dots, \sigma_i, [\sigma_{i+1}, \dots, \sigma_j], [\sigma_{j+1} \dots \sigma_m], \dots,$$

where  $\sigma_i$  is the first page fault and  $\sigma_j$  is page fault  $k + 1$ , and  $\sigma_m$  is page fault  $2k + 1$ . We will refer to the time encapsulated within a bracket as a "phase". Note that it is the final element within a bracket that is a page fault; the first element within a bracket is not necessarily a page fault.

Our goal is to show that LFD makes at least one page fault per phase, as LRU makes precisely  $k$  page faults in each phase by our construction..

Since there are  $k$  page faults per phase, one of two cases must hold:

- *Case 1: LRU faults twice on some page  $p$  in the phase.  $k + 1$  distinct pages are requested, including  $p$ , during the phase, since  $k$  pages must be requested inbetween the two faults on page  $p$  for LRU to fault on it twice. At least  $k + 1$  distinct pages requested implies LFD must fault at least once, since LFD cannot have all of these pages already in memory.*
- *Case 2: LRU faults on  $k$  distinct pages in the bracket. Let  $p_1$  denote the last page fault before the request phase (in previous phase).*
  - *Case 2a:  $\exists$  page fault on  $p_1$  by LRU at some time in the current phase. Consider  $p_1$  and the bracket following it. We may then conclude that  $k$  distinct pages were requested prior to the fault on  $p_1$ , otherwise LRU would not have removed  $p_1$  and would not have faulted on it. Thus including  $p_1$ ,  $k+1$  distinct pages were requested in the phase, thus at least 1 page fault is incurred by LFD.*
  - *Case 2b:  $\neg \exists$  page fault by LRU on  $p_1$  at some time in the current phase, so we need  $k-1$  distinct pages to max out the remaining  $k-1$  slots ( $p_1$  is in memory at the start of the phase and never faulted on, so it is occupying one page slot throughout the entire phase). Since we have requested  $k$  distinct pages, and have only  $k - 1$  spaces for them, we can be certain at least 1 page fault is incurred by LFD within the phase.*

Thus, LRU is  $k$ -competitive.

Indeed, we cannot construct a deterministic algorithm that is better than  $k$ -competitive, which we will now show.

**Theorem 2.3.5** *For any deterministic online algorithm  $A$ ,  $\exists$  a sequence of requests  $\langle \sigma \rangle$  such that  $COST_A(\sigma)$  is arbitrarily large and  $COST_A(\sigma) \geq k * COST_{MIN}(\sigma)$ , as long as there are at least  $k + 1$  pages in our universe.*

**Proof 2.3.6** *Let  $\exists$  algorithm  $A$ . We will construct a sequence to force  $A$  to produce a solution that is more than  $k$  times worse than the solution of  $MIN$  for the same sequence.*

- *First, request pages  $p_1$  through  $p_k$ .*
- *Request  $p_{k+1}$ , and denote the page  $A$  will remove to make room for  $p_{k+1}$  as  $q$ , which we can determine since  $A$  is deterministic.*
- *Request  $q$ , and denote the page  $A$  will remove to make room for  $q$  as the 'new  $q$ ' and repeat this step  $k * d$  times (for some integer  $d$ ).*

*The sequence constructed above has length  $j = k(d + 1)$ , and  $A$  executing on this sequence yields  $kd$  page faults, as the first  $k$  requests are the only non-faults. Hence  $COST_A(\sigma) = kd$ .*

*If the optimal offline algorithm  $MIN$  faults on some page  $\sigma_i$ , it will not fault on the next at least  $k - 1$  requests. Hence,*

$$COST_{MIN}(\sigma) \leq j/k = (d + 1) = (d + 1) * \frac{kd}{kd} = \frac{COST_A(\sigma)}{k} * \frac{d+1}{d}$$

*Multiply both sides by  $k$  and taking the limit as  $d \rightarrow \infty$  yields*

$$k * COST_{MIN}(\sigma_j) \leq COST_A(\sigma_j).$$

*Therefore no deterministic algorithm can be better than  $k$ -competitive.*

Fortunately it turns out that by introducing non-determinism and using a **randomized online algorithm**, we can find an algorithm that is  $\log(k)$ -competitive.

### 3 Randomized Online Algorithms

**Definition 3.0.7** *A randomized online algorithm  $A$  is a distribution of a deterministic online algorithm  $A_x$  where  $x$  is As set of random event outcomes.*

An "oblivious offline adversary" knows how algorithm  $A$  works, but does not have any prior knowledge of the random variable  $x$ . We now formally define competitive analysis for randomized online algorithms.

**Definition 3.0.8** *A randomized online algorithm is  $\alpha$ -competitive against an oblivious offline adversary if there exists a constant  $C$  such that for all input sequences the following holds:  $E[COST_A(\sigma)] \leq \alpha * COST_{min}(\sigma) + C$ , where the expected value is determined across the distribution of all possible values for  $x$ .*

#### 3.1 Randomized Marking Algorithm for Paging

- Initially, all pages in fast memory are "marked"
- Suppose page  $p$  requested.
  - If  $p$  is in fast memory already, mark and return  $p$ .
  - Otherwise: if all pages in fast memory are marked, first unmark all. Swap  $p$  with uniformly selected unmarked page in fast memory and mark  $p$ .

#### 3.2 To be continued...

The analysis of the algorithm above will be discussed in Online Algorithms part II.

## 4 Acknowledgement

Note that this lecture is heavily adapted from the notes of a class taught by Michel Goemans at MIT in 1992.