Chapter 1

Introduction

In this chapter we introduce online problems and online algorithms, give a brief history of the area, and present several motivating examples. The first two examples are the ski rental problem and the line search problem. We analyze several online algorithms for these problems using the classical notion of competitive analysis. The last problem we consider is paging. While competitive analysis can be applied to algorithms for this problem as well, the results significantly deviate from the "practical" performance of these paging algorithms. This highlights the necessity of new tools and ideas that will be explored throughout this text.

1.1 What is this Book About?

This book is about the analysis of online problems. In the basic formulation of an online problem, an input instance is given as a sequence of input items. After each input item is presented, an algorithm needs to output a decision and that decision is final, i.e., cannot be changed upon seeing any future items. The goal is to maximize or minimize an objective function, which is a function of all decisions for a given instance. (We postpone a formal definition of an online problem but hopefully the examples that follow will provide a clear intuitive meaning.) The term "online" in "online algorithms" refers to the notion of irrevocable decisions and has nothing to do with Internet, although a lot of the applications of the theory of online algorithms are in networking and online applications on the internet. The main limitation of an online algorithm is that it has to make a decision in the absence of the entire input. The value of the objective achieved by an online algorithm is compared against an optimal value of the objective that is achieved by an ideal "offline algorithm," i.e., an algorithm having access to the entire input. The ratio of the two values is called the competitive ratio.

We shall study online problems at different levels of granularity. At each level of granularity, we are interested in both positive and negative results. For instance, at the level of individual algorithms, we fix a problem, present an algorithm, and prove that it achieves a certain performance (positive result) and that the performance analysis is tight (negative result). At the higher level of models, we fix a problem, and ask what is the best performance achievable by an algorithm of a certain type (positive result) and what is an absolute bound on the performance achievable by all algorithms of a certain type (negative result). The basic model of deterministic online algorithms can be extended to allow randomness, side information (a.k.a. advice), limited mechanisms of revoking decisions, multiple rounds, and so on. Negative results can often be proved by interpreting an execution of an algorithm as a game between the algorithm and an adversary. The adversary constructs an input sequence so as to fool an algorithm into making bad online decisions. What

determines the nature and order of input arrivals? In the standard version of competitive analysis, the input items and their arrival order is arbitrary and can be viewed as being determined by an all powerful adversary. While helpful in many situations, traditional worst-case analysis is often too pessimistic to be of practical value. Thus, it is sometimes necessary to consider limited adversaries. In the *random order model* the adversary chooses the set of input items but then the order is determined randomly; in stochastic models, an adversary chooses an input distribution which then determines the sequence of input item arrivals. Hence, in all these models there is some concept of an adversary attempting to force the "worst-case" behavior for a given online algorithm or the worst-case performance against *all* online algorithms.

A notable feature of the vanilla online model is that it is information-theoretic. This means that there are no computational restrictions on an online algorithm. It is completely legal for an algorithm to perform an exponential amount of computation to make the nth decision. At first, it might seem like a terrible idea, since such algorithms wouldn't be of any practical value whatsoever. This is a valid concern, but it simply doesn't happen. Most of the positive results are achieved by very efficient algorithms, and the absence of computational restrictions on the model makes negative results really strong. Perhaps, most importantly information-theoretic nature of the online model leads to unconditional results, i.e., results that do not depend on unproven assumptions, such as $P \neq NP$.

We shall take a tour of various problems, models, analysis techniques with the goal to cover a selection of classical and more modern results, which will reflect our personal preferences to some degree. The area of online algorithms has become too large to provide a full treatment of it within a single book. We hope that you can accompany us on our journey and that you will find our selection of results both interesting and useful!

1.2 Motivation

Systematic theoretical study of online algorithms is important for several reasons. Sometimes, the online nature of input items and decisions is forced upon us. This happens in a lot of scheduling or resource allocation applications. Consider, for example, a data center that schedules computing jobs: clearly it is not feasible to wait for all processes to arrive in order to come up with an optimal schedule that minimizes makespan. The jobs have to be schedules as they come in. Some delay might be tolerated, but not much. As another example, consider patients arriving at a walk-in clinic and need to be seen by a relevant doctor. Then the receptionist plays the role of an online algorithm, and his or her decisions can be analyzed using the online framework. In online (=Internet) advertising, when a user clicks on a webpage, some advertiser needs to be matched to a banner immediately. We will see many more applications of this sort in this book. In such applications, an algorithm makes a decision no matter what: if an algorithm takes too long to make a decision it becomes equivalent to the decision of completely ignoring an item.

One should also note that the term "online algorithm" is used in a related but different way by many people with respect to scheduling algorithms. Namely, in many scheduling results, "online" could arguably be more appropriately be called "real time computation" where inputs arrive with respect to continuous time t and algorithmic decisions can be delayed at the performance cost of "wasted time". In machine learning, the concept of *regret* is the analogue of the competitive ratio (see Chapter 18). Economists have long studied market analysis within the lens of online decision making. Navigation in geometric spaces and mazes and other aspects of "search" have also been viewed as online computation. Our main perspective and focus falls within the area of algorithmic analysis for discrete computational problems. In online computation, we view input

items as arriving in discrete steps and in the initial basic model used in competitive analysis, an irrevocable decision must be made for each input item before the next item arrives. Rather than the concept of a real time clock determining time, we view time in terms of these discrete time steps.

Setting aside applications where input order and irrevocable decisions are forced upon us, in some offline applications it might be worthwhile fixing the order of input items and considering online algorithms. Quite often such online algorithms give rise to conceptually simple and efficient offline approximation algorithms. This can be helpful not only for achieving non-trivial approximation ratios for NP-hard problems, but also for problems in P (such as bipartite matching), since optimal algorithms can be too slow for large practical instances. Simple greedy algorithms tend to fall within this framework consisting of two steps: sorting input items, followed by a single online pass over the sorted items. In fact, there is a formal model for this style of algorithms called the priority model, and we will study it in detail in Chapter 17.

Online algorithms also share a lot of features with streaming algorithms. The setting of streaming algorithms can be viewed figuratively as trying to drink out of a firehose. There is a massive stream of data passing through a processing unit, and there is no way to store the entire stream for postprocessing. Thus, streaming algorithms are concerned with minimizing memory requirements in order to compute a certain function of a sequence of input items. While online algorithms do not have limits on memory or per-item processing time, some positive results from the world of online algorithms are both memory and time efficient. Such algorithms can be useful in streaming settings, which are frequent in networking and scientific computing.

1.3 Brief History of Online Algorithms

It is difficult to establish the first published analysis of an online algorithm but, for example, one can believe that there has been substantial interest in main memory paging since paging was introduced into operating systems. A seminal paper in this regard is Peter Denning's [4] introduction of the working set model for paging. It is interesting to note that almost 50 years after Denning's insightful approach to capturing locality of reference, Albers et al [1] established a precise result that characterizes the page fault rate in terms of a parameter $f(n)$ that measures the number of distinct page references in the next n consecutive page requests.

Online algorithms has been an active area of research within theoretical computer science since 1985 when Sleator and Tarjan [8] suggested that worst-case competitive analysis provided a better (than existing "average-case" analysis) explaination for the success of algorithms such as *move to front* for the list accessing problem (see chapter 4). In fact, as in almost any research area, there are previous worst case results that can be seen as at least foreshadowing the interest in competitive analysis, where one compares the performance of an online algorithm relative to what can be achieved optimally with respect to all possible inputs. Namely, Graham's [5] online greedy algorithm for the identical machines makespan problem and even more explicitly Yao's [9] analysis of online bin packing algorithms. None of these works used the term competitve ratio; this terminology was introduced by Karlin et al. [7] in their study of "snoopy caching" following the Sleator and Tarjan paper.

Perhaps remarkably, the theoretical study of online algorithms has remained an active field and one might even argue that there is now a renaissance of interest in online algorithms. This growing interest in online algorithms and analysis is due to several factors, including new applications, online model extensions, new performance measures and constraints, and an increasing interest in experimental studies that validate or challenge the theoretical analysis. And somewhat ironically,

average-case analysis (i.e. stochastic analysis) has become more prominent in the theory, design and analysis of algorithms. We believe the field of online algorithms has been (and will continue to be) a very successful field. It has led to new algorithms, new methods of analysis and a deeper understanding of well known existing algorithms.

1.4 Motivating Example: Ski Rental

Has it ever happened to you that you bought an item on an impulse, used it once or twice, and then stored it in a closet never to be used again? Even if you absolutely needed to use the item, there may have been an option to rent a similar item and get the job done at a much lower cost. If this seems familiar, you probably thought that there has to be a better way for deciding whether to buy or rent. It turns out that many such rent versus buy scenarios are represented by a single problem. This problem is called "ski rental" and it can be analyzed using the theory of online algorithms. Let's see how.

The setting is as follows. You have arrived at a ski resort and you will be staying there for an unspecified number of days. As soon as the weather turns bad and the resort closes down the slopes, you will leave never to return again. Each morning you have to make a choice either to rent skis for $r\$$ or to buy skis for $b\$$. By scaling we can assume that $r = 1\$$. For simplicity, we will assume that b is an integer ≥ 1. If you rent for the first k days and buy skis on day $k+1$, you will incur the cost of $k+b$ for the entire stay at the resort, that is, after buying skis you can use them an unlimited number of times free of charge. The problem is that due to unpredictable weather conditions, the weather might deteriorate rapidly. It could happen that the day after you buy the skis, the weather will force the resort to close down. Note that the weather forecast is accurate for a single day only, thus each morning you know with perfect certainty whether you can ski on that day or not before deciding to buy or rent, but you have no information about the following day. In addition, unfortunately for you, the resort does not accept returns on purchases. In such unpredictable conditions, what is the best strategy to minimize the cost of skiing during all good-weather days of your stay?

An optimal offline algorithm simply computes the number g of good-weather days. If $g \leq b$ then an optimal strategy is to rent skis on all good-weather days. If $g > b$ then the optimal strategy is to buy skis on the first day. Thus, the offline optimum is $\min(g, b)$. Even without knowing g it is possible to keep expenses roughly within a factor of 2 of the optimum. The idea is to rent skis for $b-1$ days and buy skis on the following day after that. If $g < b$ then this strategy costs $g \leq (2-1/b)g$, since the weather would spoil on day $g+1 \leq b$ and you would leave before buying skis on day b. Otherwise, $g \geq b$ and our strategy incurs cost $b-1+b = (2-1/b)b$, which is slightly better than twice the offline optimum, since for this case we have $b = \min(g, b)$. This strategy achieves competitive ratio $2-1/b$, which approaches 2 as b increases.

Can we do better? If our strategy is deterministic, then no. On each day, an adversary sees whether you decided to rent or buy skis, and based on that decision and past history declares whether the weather is going to be good or bad starting from the next day onward. If you buy skis on day $i \leq b-1$ then the adversary declares the weather to be bad from day $i+1$ onward. This way an optimal strategy is to rent skis for a total cost of i, but you incurred the cost of $(i-1)+b \geq (i-1)+(i+1) = 2i$; that is, twice the optimal. If you buy skis on day $i \geq b$, then the adversary declares bad weather after the first $2b$ days. An optimal strategy is to buy skis on the very first day with a cost of b, whereas you spent $i-1+b \geq b-1+b = (2-1/b)b$. Thus, no matter what you do an adversary can force you to spend $(2-1/b)$ times the optimal.

Can we do better if we use randomness? We assume a weak adversary — such an adversary

knows your algorithm, but has to commit to spoiling weather on some day $g+1$ without seeing your random coins or seeing any of your decisions. Observe that for deterministic algorithms, a weak adversary can simulate the stronger one that adapts to your decisions. The assumption of a weak adversary for the ski rental problem is reasonable because the weather doesn't seem to conspire against you based on the outcomes of your coin flips. It is reasonable to conjecture that even with randomized strategy you should buy skis before or on day b. You might improve the competitive ratio if you buy skis before day b with some probability. One of the simplest randomized algorithms satisfying these conditions is to pick a random integer $i \in [0, b-1]$ from some distribution p and rent for i days and buy skis on day $i+1$ (if the weather is still good). Intuitively, the distribution should allocate more probability mass to larger values of i, since buying skis very early (think of the first day) makes it easier for the adversary to punish such decision. We measure the competitive ratio achieved by a randomized algorithm by the ratio of the expected cost of the solution found by the algorithm to the cost of an optimal offline solution. To analyze our strategy, we consider two cases.

In the first case, the adversary spoils the weather on day $g+1$ where $g < b$. Then the expected cost of our solution is $\sum_{i=0}^{g-1}(i+b)p_i + \sum_{i=g}^{b-1} gp_i$. Since an optimal solution has cost g in this case, we are interested in finding the minimum value of c such that

$$\sum_{i=0}^{g-1}(i+b)p_i + \sum_{i=g}^{b-1} gp_i \leq cg.$$

In the second case, the adversary spoils the weather on day $g+1$ where $g \geq b$. Then the expected cost of our solution is $\sum_{i=0}^{b-1} ip_i + b$. Since an optimal solution has cost b in this case, we need to ensure $\sum_{i=0}^{b-1} ip_i + b \leq cb$.

We can write down a linear program to minimize c subject to the above inequalities together with the constraint $p_0 + p_1 + \cdots + p_{b-1} = 1$.

$$\begin{aligned}
&\text{minimize} && c \\
&\text{subject to} && \sum_{i=0}^{g-1}(i+b)p_i + \sum_{i=g}^{b-1} gp_i \leq cg \quad \text{for } g \in [b-1] \\
& && \sum_{i=0}^{b-1} ip_i + b \leq cb \\
& && p_0 + p_1 + \cdots + p_{b-1} = 1
\end{aligned}$$

We claim that $p_i = \frac{c}{b}(1-1/b)^{b-1-i}$ and $c = \frac{1}{1-(1-1/b)^b}$ is a solution to the above LP. Thus, we need to check that all constraints are satisfied. First, let's check that p_i form a probability distribution:

$$\sum_{i=0}^{b-1} p_i = \sum_{i=0}^{b-1} \frac{c}{b}(1-1/b)^{b-1-i} = \frac{c}{b}\sum_{i=0}^{b-1}(1-1/b)^i = \frac{c}{b}\frac{1-(1-1/b)^b}{1-(1-1/b)} = 1.$$

Next, we check all constraints involving g.

$$\begin{aligned}\sum_{i=0}^{g-1}(i+b)p_i + \sum_{i=g}^{b-1} gp_i &= \sum_{i=0}^{g-1}(i+b)\frac{c}{b}(1-1/b)^{b-1-i} + \sum_{i=g}^{b-1} g\frac{c}{b}(1-1/b)^{b-1-i} \\ &= (1-1/b)^{b-g}cg + \left((1-1/b)^g - (1-1/b)^b\right)(1-1/b)^{-g}cg \\ &= cg\end{aligned}$$

In the above, we skipped some tedious algebraic computations (we invite the reader to verify each of the above equalities). Similarly, we can check that the p_i satisfy the second constraint of the LP. Notably, the solution p_i and c given above satisfies each constraint with equality. We conclude that our randomized algorithm achieves the competitive ratio $\frac{1}{1-(1-1/b)^b}$. Since $(1-1/n)^n \to e^{-1}$, the competitive ratio of our randomized algorithm approaches $\frac{e}{e-1} \approx 1.5819\ldots$ as b goes to infinity.

1.5 Motivating Example: Line Search Problem

A robot starts at the origin of the x-axis. It can travel one unit of distance per one unit of time along the x-axis in either direction. An object has been placed somewhere on the x-axis. The robot can switch direction of travel instantaneously, but in order for the robot to determine that there is an object at location x', the robot has to be physically present at x'. How should the robot explore the x-axis in order to find the object as soon as possible? This problem is known as the line search problem or the cow path problem.

Suppose that the object has been placed at distance d from the origin. If the robot knew whether the object was placed to the right of the origin or to the left of the origin, the robot could start moving in the right direction, finding the object in time d. This is an optimal "offline" solution.

Since the robot does not know in which direction it should be moving to find the object, it needs to explore both directions. This leads to a natural zig-zag strategy. Initially the robot picks the positive direction and walks for 1 unit of distance in that direction. If no object is found, the robot returns to the origin, flips the direction and doubles the distance. We call each such trip in one direction and then back to the origin a phase, and we start counting phases from 0. These phases are repeated until the object is found. If you have seen the implementation and amortized analysis of an automatically resizeable array implementation, then this doubling strategy will be familiar. In phase i robot visits location $(-2)^i$ and travels the distance $2 \cdot 2^i$. Worst case is when an object is located just outside of the radius covered in some phase. Then the robot returns to the origin, doubles the distance and travels in the "wrong direction", returns to the origin, and discovers the object by travelling in the "right direction." In other words when an object is at distance $d = 2^i + \epsilon > 2^i$ in direction $(-1)^i$, the total distance travelled is $2(1+2+\cdots+2^i+2^{i+1})+d \leq 2\cdot 2^{i+2}+d < 8d+d = 9d$. Thus, this doubling strategy gives a 9-competitive algorithm for the line search problem.

Typically online problems have well-defined input that makes sense regardless of which algorithm you choose to run, and the input is revealed in an online fashion. For example, in the ski rental problem, the input consists of a sequence of elements, where element i indicates if the weather on day i is good or bad. The line search problem does not have input of this form. Instead, the input is revealed in response to the actions of the algorithm. Yet, we can still interpret this situation as a game between an adversary and the algorithm (the robot). At each newly discovered location, an adversary has to inform the robot whether an object is present at that location or not. The adversary eventually has to disclose the location, but the adversary can delay it as long as needed in order to maximize the distance travelled by the robot in relation to the "offline" solution.

1.6 Motivating Example: Paging

Computer storage comes in different varieties: CPU registers, random access memory (RAM), solid state drives (SSD), hard drives, tapes, etc. Typically, the price per byte is positively correlated with the speed of the storage type. Thus, the fastest type of memory – CPU registers – is also the most expensive, and the slowest type of memory – tapes – is also the cheapest. In addition, certain types of memory are volatile (RAM and CPU registers), while other types (SSDs, hard drives, tapes) are persistent. Thus, a typical architecture has to mix and match different storage types. When information travels from a large-capacity slow storage type to a low-capacity fast storage type, e.g., RAM to CPU registers, some bottlenecking will occur. This bottlenecking can be mitigated by using a cache. For example, rather than accessing RAM directly, the CPU checks a local cache, which stores a local copy of a small number of pages from RAM. If the requested data is in the cache (this event is called "cache hit"), the CPU retrieves it directly from the cache. If the requested data is not in the cache (called "cache miss"), the CPU first brings the requested data from RAM into the cache, and then reads it from the cache. If the cache is full during a "cache miss," some existing page in the cache needs to be evicted. The paging problem is to design an algorithm that decides which page needs to be evicted when the cache is full and cache miss occurs. The objective is to minimize the total number of cache misses. Notice that this is an inherently online problem that can be modelled as follows. The input is a sequence of natural numbers $X = x_1, x_2, \ldots$, where x_i is the number of the page requested by the CPU at time i. Given a cache of size k, initially the cache is empty. The cache is simply an array of size k, such that a single page can be stored at each position in the array. For each arriving x_i, if x_i is in the cache, the algorithm moves on to the next element. If x_i is not in the cache, the algorithm specifies an index $y_i \in [k]$, which points to a location in the cache where page x_i is to be stored evicting any existing page. We will measure the performance by the classical notion of the competitive ratio — the ratio of the number of cache misses of an online algorithm to the minimum number of cache misses achieved by an optimal offline algorithm that sees the entire sequence in advance. Let's consider two natural algorithms for this problem.

FIFO - First In First Out. If the cache is full and a cache miss occurs, this algorithm evicts the page from the cache that was inserted the earliest. We will first argue that this algorithm incurs at most (roughly) k times more cache misses than an optimal algorithm. To see this, subdivide the entire input into consecutive blocks $B_1, B_2, \ldots$. Block B_1 consists of a maximal prefix of X that contains exactly k distinct pages (if the input has fewer than k distinct pages, then any "reasonable" algorithm is optimal). Block B_2 consists of a maximal prefix of $X \setminus B_1$ that contains exactly k distinct pages, and so on. Let n be the number of blocks. Observe that FIFO incurs at most k cache misses while processing each block. Thus, the overall number of cache misses of FIFO is at most nk. Also, observe that the first page of block B_{i+1} is different from *all pages* of B_i due to the maximality of B_i. Therefore while processing B_i and the first page from B_{i+1} any algorithm, including an optimal offline one, incurs a cache miss. Thus, an optimal offline algorithm incurs at least $n - 1$ cache misses while processing X. Therefore, the competitive ratio of FIFO is at most $nk/(n-1) = k + \frac{k}{n-1} \to k$ as $n \to \infty$.

LRU - Least Recently Used. If the cache is full and a cache miss occurs, this algorithm evicts the page from the cache that was accessed least recently. Note that LRU and FIFO both keep timestamps together with pages in the cache. When x_i is requested and it results in a cache miss, both algorithms initialize the timestamp corresponding to x_i to i. The difference is that FIFO never updates the timestamp until x_i itself is evicted, whereas LRU updates the timestamp to j whenever cache hit occurs, where $x_j = x_i$ with $j > i$ and x_i still in the cache. Nonetheless, the two algorithms are sufficiently similar to each other, that essentially the same analysis as for FIFO can

be used to argue that the competitive ratio of LRU is at most k (when $n \to \infty$).

We note that both FIFO and LRU do not achieve a competitive ratio better than k. Furthermore, no deterministic algorithm for paging can achieve a competitive ratio better than k. To prove this, it is sufficient to consider sequences that use page numbers from $[k+1]$. Let A be a deterministic algorithm, and suppose that it has a full cache. Since the cache is of size k, in each consecutive time step an adversary can always find a page that is not in the cache and request it. Thus, an adversary can make the algorithm A incur a cache miss on every single time step. An optimal offline algorithm evicts the page from the cache that is going to be requested furthest in the future. Since there are k pages in the cache, there is at least $k-1$ pages in the future inputs that are going to be requested before one of the pages in the cache. Thus, the next cache miss can only occur after $k-1$ steps. The overall number of cache misses by an optimal algorithm is at most $|X|/k$, whereas A incurs essentially $|X|$ cache misses. Thus, A has competitive ratio at least k.

We finish this section by noting that while competitive ratio gives useful and practical insight into the ski rental and line search problems, it falls short of providing practical insight into the paging problem. First of all, notice that the closer competitive ratio is to 1 the better. The above paging results show that increasing cache size k makes LRU and FIFO perform worse! This goes directly against the empirical observation that larger cache sizes lead to improved performance. Another problem is that the competitive ratio of LRU and FIFO is the same suggesting that these two algorithms perform equally well. It turns out that in practice LRU is *far superior* to FIFO, because of "locality of reference" – the phenomenon that if some memory was accessed recently, the same or nearby memory will be accessed in the near future. There are many reasons for why this phenomenon is pervasive in practice, not the least of which is a common use of arrays and loops, which naturally exhibit "locality of reference." None of this is captured by competitive analysis as it is traditionally defined.

The competitive ratio is an important tool for analyzing online algorithms having motivated and initiating the area of online algorithm analysis. However, being a worst-case measure, it may not not model reality well in many applications. This has led researchers to consider other models, such as stochastic inputs, advice, look-ahead, and parameterized complexity, among others. We shall cover these topics in the later chapters of this book.

1.7 Exercises

1. Fill in details of the analysis of the randomized algorithm for the ski rental problem.

2. Consider the setting of the ski rental problem with rental cost 1\$ and buying cost b\$, $b \in \mathbb{N}$. Instead of an adversary choosing a day $\in \mathbb{N}$ when the weather is spoiled, this day is generated at random from distribution p. Design an optimal deterministic online algorithm for the following distributions p:

   (a) uniform distribution on $[n]$.

   (b) geometric distribution on $\mathbb{N}$ with parameter $1/2$.

3. What is the competitive ratio achieved by the following randomized algorithm for the line search problem? Rather than always picking initial direction to be $+1$, the robot selects the initial direction to be $+1$ with probability $1/2$ and -1 with probability $1/2$. The rest of the strategy remains the same.

4. Instead of searching for treasure on a line, consider the problem of searching for treasure on a 2-dimensional grid. The robot begins at the origin of $\mathbb{Z}^2$ and the treasure is located

at some coordinate $(x, y) \in \mathbb{Z}^2$ unknown to the robot. The measure of distance is given by the Manhattan metric $||(x, y)|| = |x| + |y|$. The robot has a compass and at each step can move north, south, east, or west one block. Design an algorithm for the robot to find the treasure on the grid. What is the competitive ratio of your algorithm? Can you improve the competitive ratio with another algorithm?

5. Consider Flush When Full algorithm for paging: when a cache miss occurs and the entire cache is full, evict *all* pages from the cache. What is the competitive ratio of Flush When Full?

6. Consider adding the power of limited lookahead to an algorithm for paging. Namely, fix a constant $f \in \mathbb{N}$. Upon receiving the current page p_i, the algorithm also learns $p_{i+1}, p_{i+2}, \ldots, p_{i+f}$. Recall that the best achievable competitive ratio for deterministic algorithms without lookahead is k. Can you improve on this bound with lookahead?