Online Algorithms

Allan Borodin Denis Pankratov

DRAFT: January 26, 2019

Contents

Preface

In 1998, Allan Borodin and Ran El-Yaniv co-authored the text "Online computation and competitive analysis". As in most texts, some important topics were not covered. To some extent, this text aims to rectify some of those omissions. Furthermore, because the field of Online Algorithms has remained active, many results have been improved. But perhaps most notable, is that the basic adversarial model of competitive analysis for online algorithms has evolved to include new models of algorithm design that are important theoretically as well as having a significant impact on many current applications.

In Part I, we first review the basic definitions and some of the "classical" online topics and algorithms; that is, those already well studied as of 1998. We then present some newer online results for graph problems, scheduling problems, max-sat and submodular function maximization. We also present the seminal primal dual analysis for online algorithms introduced by Buchbinder and Naor that provides an elegant unifying model for many online results. Part I concludes with an update on some recent progress for some of the problems introduced earlier.

The focus of Part II is the study of extensions of the basic online competitive analysis model. In some chapters, we discuss alternative "online-like" algorithmic models. In other chapters, the analysis of online algorithms goes beyond the worst case perspective of the basic competitive analysis in terms of both stochastic analysis as well as alternative performance measures.

In Part III, we discuss some additional application areas, many of these applications providing the motivation for the continuing active interest (and indeed for what we consider is a renaissance) in the study of online and online-like algorithms. We view our text primarily as an introduction to the study of online and online-like algorithms for use in advanced undergraduate and graduate courses. In addition, we believe that the text offers a number of interesting potential research topics.

At the end of each chapter, we present some exercises. Some exercises are denoted by a (*) indicating that the exercise is technically more challenging. Some other exercises are denoted by as (**) indicating that to the best of our knowledge this is not a known result. We have also provided some history and the most relevant citations for results presented in the chapter and some relevant related work that is not mentioned in the chapter.

We thank ....

Part I

Basics

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 [6] 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 [13] 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 [7] online greedy algorithm for the identical machines makespan problem and even more explicitly Yao's [14] analysis of online bin packing algorithms. None of these works used the term competitve ratio; this terminology was introduced by Karlin et al. [12] 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

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

In the second case, the adversary spoils the weather on day $g+1$ where $g \ge b$. 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 $\sum_{i=0}^{b-1} ip_i + b \le cb$.

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

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

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.

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

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 $2(1+2+\cdots+2^i+2^{i+1})+d \le 2\cdot 2^{i+2}+d < 8d+d = 9d$. 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?

Chapter 2

Deterministic Online Algorithms

In this chapter we formally define a general class of online problems, called *request-answer games* and then define the competitive ratio of deterministic online algorithms with respect to request-answer games. The definitions depend on whether we are dealing with a minimization or a maximization problem. As such, we present examples of each. For minimization problems we consider the makespan problem and the bin packing problem.For maximization problems we consider time-series search and the one-way trading problems.

2.1 Request-Answer Games

In the Chapter 1, we gave an informal description of an online problem, one that is applicable to most online problems in the competitive analysis literature including the ski rental, paging, and makespan problems. Keeping these problems in mind, we can define the general request-answer framework that formalizes the class of online problems that abstracts almost all the problems in this text with the notable exception of the line search problem and more generally the navigation and exploration problems in Chapter 23.

Definition 2.1.1. A *request-answer* game for a minimization problem consists of a request set R, an answer set[1] A, and cost functions $f_n : R^n \times A^n \to \mathbb{R} \cup \{\infty\}$ for $n = 0, 1, \ldots$.

Here ∞ indicates that certain solutions are not allowed. For a maximization problem the f_n refer to profit functions and we have $f_n : R^n \times A^n \to \mathbb{R} \cup \{-\infty\}$. Now $-\infty$ indicates that certain solutions are not allowed. We can now provide a template for deterministic online algorithms for any problem within the request-answer game framework.

```
   Online Algorithm Template
1: On an instance $I$, including an ordering of the data items $(x_1, \ldots, x_n)$:
2: $i := 1$
3: While there are unprocessed data items
4:    The algorithm receives $x_i \in R$ and makes an irrevocable decision $d_i \in A$ for $x_i$
      (based on $x_i$ and all previously seen data items and decisions).
5:    $i := i + 1$
6: EndWhile
```

[1]For certain results concerning requst-answer games, it is required that the answer set be a finite set. However, for the purposes of defining a general framework and for results in this text, this is not necessary.

2.2 Competitive Ratio for Minimization Problems

In this section we formally define the notion of *competitive ratio.* In a later section we will analyze the performance of Graham's greedy makespan algorithm in terms of its competitive ratio. Let ALG be an online algorithm and $\mathcal{I} = \langle x_1, x_2, \ldots x_n \rangle$ be an input sequence. We shall abuse notation and let $ALG(\mathcal{I})$ denote both the output of the algorithm as well as the objective value of the algorithm when the context is clear. If we want to distinguish the objective value we will write $|ALG(\mathcal{I})|$.

Definition 2.2.1. Let ALG be an online algorithm for a minimization problem and let OPT denote an optimal solution to the problem. *The competitive ratio* of ALG, denoted by $\rho(ALG)$, is defined as follows:

$$\rho(ALG) = \limsup_{|OPT(\mathcal{I})| \to \infty} \frac{ALG(\mathcal{I})}{OPT(\mathcal{I})}.$$

Equivalently, we can say that the competitive ratio of ALG is ρ if for all sufficiently large inputs $\mathcal{I}$, we have $ALG(\mathcal{I}) \leq \rho \cdot OPT(\mathcal{I}) + o(OPT(\mathcal{I}))$. This then is just a renaming of the asymptotic approximation ratio as is widely used in the study of offline optimization algorithms. We reserve the competitive ratio terminology for online algorithms and use *approximation ratio* otherwise. In offline algorithms, when $ALG(\mathcal{I}) \leq \rho \cdot OPT(\mathcal{I})$ for all $\mathcal{I}$, we simply say approximation ratio; for online algorithms we will say that ALG is *strictly ρ competitive* when there is no additive term.

In the literature you will often see a slightly different definition of the competitive ratio. Namely, an online algorithm is said to achieve competitive ratio ρ if there is a constant $\alpha \geq 0$ such that for all inputs $\mathcal{I}$ we have $ALG(\mathcal{I}) \leq \rho \cdot OPT(\mathcal{I}) + \alpha$. We shall refer to this as the *classical definition.* The difference between the two definitions is that our definition allows us to ignore additive terms that are small compared to OPT, whereas the classical definition allows us to ignore constant additive terms. The two definitions are "almost" identical in the following sense. If an algorithm achieves the competitive ratio ρ with respect to the classical definition then it achieves the competitive ratio ρ with respect to our definition as well (assuming $OPT(\mathcal{I}) \to \infty$). Conversely, if an algorithm achieves the competitive ratio ρ with respect to our definition then it achieves the competitive ratio $\rho + \epsilon$ for any constant $\epsilon > 0$ with respect to the classical definition. The latter part is because we have $ALG \leq \rho OPT + o(OPT) = (\rho + \epsilon)OPT + o(OPT) - \epsilon OPT \leq (\rho + \epsilon)OPT + \alpha$ for a suitably chosen *constant* α such that α dominates the term $o(OPT) - \epsilon OPT$. We prefer our definition over the classical one because it makes stating the results simpler sometimes. In any case, as outlined above the difference between the two definitions is minor.

Implicit in this definition is the worst-case aspect of this performance measure. To establish a lower bound (i.e. an *inapproximation* for an online algorithm, there is a game between the algorithm and an adversary. Once the algorithm is stated the adversary creates a nemesis instance (or set of instances if we are trying to establish asymptotic inapproximation results). Sometimes it will be convenient to view this game in extensive form where the game alternates between the adversary announcing the next input item and the algorithm making a decision for this item. However, since the adversary knows the algorithm, the nemesis input sequence can be determined in advance and this becomes a game in normal (i.e., matrix) form. In addition to establishing inapproxination bounds (i.e. lower bounds on the competitive ratio) for specific algorithms, we sometimes wish to establish an inapproximation bound for *all* online algorithms. In this case, we need to show how to derive an appropriate nemesis sequence for any possible online algorithm.

2.3 Minimization Problem Example: Makespan

For definiteness, we consider the makespan problem for identical machines and Graham's online greedy algorithm. As stated earlier, this algorithm is perhaps the earliest example of a competitive analysis argument.

Makespan for identical machines

Input: $(p_1, p_2, \ldots, p_n)$ where p_j is the load or processing time for a job j; m — the number of identical machines.

Output: $\sigma : \{1, 2, \ldots, n\} \rightarrow \{1, 2, \ldots m\}$ where $\sigma(j) = i$ denotes that the j^{th} job has been assigned to machine i.

Objective: To find σ so as to minimize $\max_i \sum_{i:\sigma(j)=i} p_j$.

Algorithm 1 The online greedy makespan algorithm.

```
procedure GREEDY MAKESPAN
    initialize $s(i) \leftarrow 0$ for $1 \leq i \leq m$          ▷ $s(i)$ is the current load on machine $i$
    $j \leftarrow 1$
    while $j \leq n$ do
        $i' \leftarrow \arg\min_i s(i)$          ▷ The algorithm can break ties arbitrarily
        $\sigma(j) \leftarrow i'$
        $s(i') \leftarrow s(i') + p_j$
        $j \leftarrow j + 1$
    return $\sigma$
```

The makespan of a machine is its current load and the greedy algorithm schedules each job on some least loaded machine. The algorithm is online in the sense that the scheduling of the i^{th} job takes place before seeing the remaining jobs. The algorithm is *greedy* in the sense that at any step the algorithm schedules so as optimize as best as it can given the current state of the computation without regard to possible future jobs. We note that as stated, the algorithm is not fully defined. That is, there may be more than one machine whose current makespan is minimal. When we do not specify a "tie-breaking" rule, we mean that how the algorithm breaks ties is not needed for the correctness and performance analysis. That is the analysis holds even if we assume that an adversary is breaking the ties.

As an additional convention, when we use a WHILE loop we are assuming that the number n of online inputs is not known to the algorithm; otherwise, we will use a FOR loop to indicate that n is known a priori.

We now provide an example of competitive analysis by analyzing the online greedy algorithm for the makespan problem.

Theorem 2.3.1. *Let $G(\mathcal{I})$ denote the makespan of Graham's online greedy algorithm on m machines when executed on the input sequence $\mathcal{I}$.*

Then for all inputs $\mathcal{I}$ we have $G(\mathcal{I}) \leq (2 - 1/m) \cdot OPT(\mathcal{I})$. That is, this online greedy algorithm has a strict competitive ratio which is at most $(2 - 1/m)$.

Proof. Let $p_1, p_2, \ldots p_n$ be the sequence of job sizes and let $p_{max} = \max_j p_j$. Lets first establish two necessary bounds for any solution and thus for OPT. The following are simple claims:

- $OPT \geq (\sum_{k=1}^{n} p_k)/m$; that is the maximum load must be at least the average load.
- $OPT \geq p_{max}$

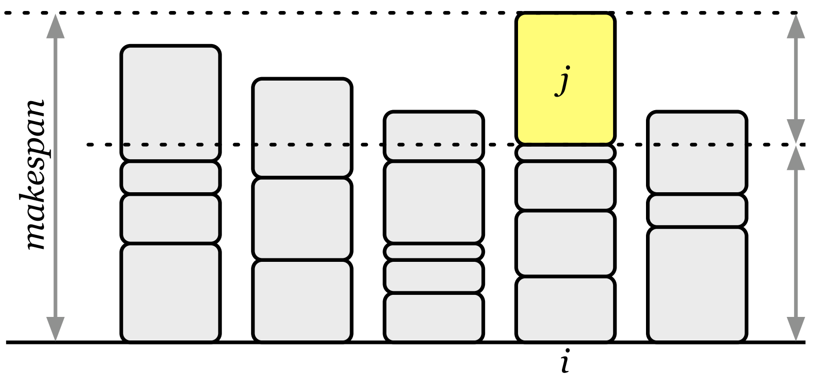

Figure 2.1: Figure from Jeff Erickson's lecture notes

Now we want to bound the makespan of greedy in terms of these necessary OPT bounds. Let machine i be one of the machines defining the makespan for the greedy algorithm G and lets say that job j is the last item to be scheduled on machine i. If we let q_i be the load on machine i just before job j is scheduled, then $G's$ makespan is $q_i + p_j$. By the greedy nature of G and the fact that the minimum load must be less than the average load (see Figure 2.1), we have $q_i \leq \sum_{k \neq j} p_k/m$ so that

$$G(p_1, \ldots p_n) \leq \sum_{k \neq j} p_k/m + p_j = \sum_{k \neq j} p_k/m + p_j/m - p_j/m + p_j = \sum_{k=1}^{n} p_k + (1 - 1/m)p_j$$

Since $\sum_{k=1}^{n} p_k \leq OPT$ and $p_j \leq p_{max} \leq OPT$ we have our desired competitive ratio. $\square$

We shall see that this *strict competitive ratio* is "tight" in the sense that there exists an input $\mathcal{I}$ such that $G(\mathcal{I}) = (2-1/m) \cdot OPT(\mathcal{I})$. We construct such an input sequence $\mathcal{I}$ with $n = m(m-1)+1$ jobs, comprised of $m(m-1)$ initial jobs having unit load $p_j = 1$ followed by a final job having load $p_n = m$. The optimal solution would balance the unit load jobs on say the first $m-1$ machines leaving the last machine to accommodate the final big job having load m. Thus each machine has load m and $OPT = m$. On the other hand, the greedy algorithm G would balance the unit job of all m machines and then be forced to place the last job on some machine which already has load $m-1$ so that the G's makespan is $m + (m-1)$. It follows that for this sequence the ratio is $\frac{2m-1}{m} = 2 - 1/m$ matching the bound in Theorem 2.3.1.

We note that for $m = 2$ and $m = 3$, it is not difficult to show that the bound is tight for *any* (not necessarily greedy) online algorithm. For example, for $m = 2$, an adversary can either provide the input sequence (1,1) or (1,1,2). If the algorithm spreads the two initial unit jobs, the adversary ends the input having only presented (1,1); otherwise the adversarial input is (1,1,2). Even though it may seem that this problem is pretty well understood, there is still much to reflect upon concerning the makespan problem and the greedy algorithm analysis. We note that the lower bound as given relies on the number n of inputs not being known initially by the algorithm. Moreover, the given inapproximation holds for input sequences restricted to $n \leq 3$ and does not establish an *asymptotic inapproximation*. For $m \geq 4$, there are online (non-greedy) algorithms that improve upon the greedy bound. The general idea for an improved competitive ratio is to leave some space for potentially large jobs. Currently, the best known "upper bound" that holds for all m is 1.901 and the "lower bound" for sufficiently large m is 1.88.

Although the greedy inapproximation is not an asymptotic result, the example suggests a simple greedy (but not online) algorithm. The nemesis sequence for all m relies on the last job being a large job. This suggests sorting the input items so that $p_1 \geq p_2 \ldots \geq p_n$. This is Graham's LPT ("longest processing time") algorithm which has a tight approximation ratio of $\frac{4}{3} - \frac{1}{3m}$.

2.4 Minimization Problem Example: Bin Packing

Bin packing is extensively studied within the context of offline and online approximation algorithms. Like the makespan problem (even for identical machines), it is an NP-hard optimization problem. In fact, the hardness of both makespan and bin packing is derived by a reduction from the subset sum problem. In the basic bin packing problem, we are given a sequence of items described by their weights $(x_1, x_2, \ldots, x_n)$ such that $x_i \in [0, 1]$. We have an unlimited supply of bins each of unit weight capacity. The goal is to pack all items in the *smallest* number of bins. Formally, it is stated as

Bin Packing
Input: $(x_1, x_2, \ldots, x_n)$; x_i is the weight of an item i.
Output: $\sigma : \{1, 2, \ldots, n\} \to \{1, 2, \ldots m\}$ for some integer m.
Objective: To find σ so as to minimize m subject to the constraints $\sum_{j:\sigma(j)=i} x_j \leq 1$ for each $i \in [m]$.

In this section, we shall analyze the competitive ratios of the following three online algorithms: $NextFit, FirstFit$, and $BestFit$.

- $NextFit$: if the newly arriving item does not fit in the *most recently opened* bin, then open a new bin and place the new item in that bin. See Algorithm 2 for pseudocode.

- $FirstFit$: find the first bin among *all opened bins* that has enough remaining space to accommodate the newly arriving item. If such a bin exists, place the new item there. Otherwise, open a new bin and place the new item in the new bin. See Algorithm 3 for pseudocode.

- $BestFit$: find a bin among *all opened bins* that has *minimum* remaining space among all bins that have enough space to accommodate the newly arriving item. If there are no bins that can accommodate the newly arriving item, open a new bin and place the new item in the new bin. See Algorithm 4 for pseudocode.

The algorithms $FirstFit$ and $BestFit$ are greedy in the sense that they will never open a new bin unless it is absolutely necessary to do so. The difference between these two algorithms is in how they break ties when there are several existing bins that could accommodate the new item: $FirstFit$ simply picks the first such bin, while $BestFit$ picks the bin that would result in tightest possible packing. The algorithm $NextFit$ is not greedy — it always considers only the most recently opened bin, and does not check any of the older bins.

The simplest algorithm to analyze is $NextFit$ so we start with it. Later we introduce the weighting technique that is used to analyze both $FirstFit$ and $BestFit$ (in fact, simultaneously).

Theorem 2.4.1.
$$\rho(NextFit) \leq 2.$$

Proof. Define $B[i] = 1 - R[i]$, which keeps track of how much weight is occupied by bin i. Assume for simplicity that $NextFit$ created an even number of bins, i.e., m is even. Then we have $B[1] + B[2] >$

Algorithm 2 The *NextFit* algorithm

```
procedure NextFit
    $m \leftarrow 0$                              ▷ total number of opened bins so far
    $R \leftarrow 0$                              ▷ remaining space in the most recently opened bin
    while $j \le n$ do
        if $x_j < R$ then
            $m \leftarrow m + 1$
            $R \leftarrow 1 - x_j$
        else
            $R \leftarrow R - x_j$
        $\sigma(j) \leftarrow m$
        $j \leftarrow j + 1$
```

Algorithm 3 The *FirstFit* algorithm

```
procedure FirstFit
    $m \leftarrow 0$                              ▷ total number of opened bins so far
    $R$ — a map keeping track of remaining space in all opened bins
    while $j \le n$ do
        $flag \leftarrow False$
        for $i = 1$ to $m$ do
            if $x_j < R[i]$ then
                $R[i] \leftarrow R[i] - x_j$
                $\sigma(j) \leftarrow i$
                $flag \leftarrow True$
                break
        if $flag = False$ then
            $m \leftarrow m + 1$
            $R[m] \leftarrow 1 - x_j$
            $\sigma(j) \leftarrow m$
        $j \leftarrow j + 1$
```

1, since the first item of bin 2 could not fit into the remaining space of bin 1. Similarly we get $B[2i-1] + B[2i] > 1$ for all $i \in \{1, \dots, m/2\}$. Adding all these inequalities, we have

$$\sum_{i=1}^{m/2} B[2i-1] + B[2i] > m/2.$$

Now, observe that the left hand side is simply $\sum_{j=1}^{n} w_j$ and that $OPT \ge \sum_{j=1}^{n} w_j$. Combining these observations with the above inequality we get $OPT > m/2 = NextFit/2$. Therefore, we have $NextFit < 2OPT$. □

Next, we show that $\rho(NextFit) \ge 2$. Fix arbitrary small $\epsilon > 0$ such that $n := 1/\epsilon \in \mathbb{N}$. The input will consist of $3n$ items. The first $2n$ items consist of repeating pairs $1-\epsilon, 2\epsilon$. The remaining n items are all ϵ. Thus, the input looks like this: $1-\epsilon, 2\epsilon, 1-\epsilon, 2\epsilon, \dots, 1-\epsilon, 2\epsilon, \epsilon, \dots, \epsilon$, where the dot dot dots indicate that the corresponding pattern repeats n times. Observe that *NextFit* on this instance uses at least $2n$ bins, since the repeating pattern of pairs $1-\epsilon, 2\epsilon$ forces the algorithm to use a new bin on each input item ($1 - \epsilon + 2\epsilon = 1 + \epsilon > 1$). An optimal algorithm can match

Algorithm 4 The *BestFit* algorithm

```
procedure BestFit
    $m \leftarrow 0$                                        ▷ total number of opened bins so far
    $R$ — a map keeping track of remaining space in all opened bins
    while $j \leq n$ do
        $ind \leftarrow -1$
        for $i = 1$ to $m$ do
            if $x_j < R[i]$ then
                if $ind = -1$ or $R[i] < R[ind]$ then
                    $ind \leftarrow i$
        if $ind = -1$ then
            $m \leftarrow ind \leftarrow m + 1$
            $R[m] \leftarrow 1$
        $\sigma(j) \leftarrow ind$
        $R[ind] \leftarrow R[ind] - x_j$
        $j \leftarrow j + 1$
```

items of weight $1 - \epsilon$ with items of weight ϵ for n bins total, and the remaining n items of weight 2ϵ can be placed into $(2\epsilon)n = (2\epsilon)/\epsilon = 2$ bins. Thus, we have $OPT \leq n + 2$ whereas $NextFit \geq 2n$. This means that $\rho(NextFit) \geq 2n/(n + 2) \to 2$ as $n \to \infty$.

The weighting technique is a general technique for analyzing *asymptotic* competitive ratios of algorithms for the bin packing problem. The idea behind the weighting technique is to define a weight function $w : [0, 1] \to \mathbb{R}$ with the three properties described below. But first we need a few more definitions. Let ALG be an algorithm that we want to analyze. Fix an input sequence $x_1, \dots, x_n$ and suppose ALG uses m bins. Define S_i to be the set of indices of items that were placed in bin i by ALG, and extend the weight function to index sets $S \subseteq [n]$ as $w(S) := \sum_{i \in S} w(x_i)$. Now, we are ready to state the properties:

1. $w(x) \geq x$;
2. there exist an absolute constant $k_0 \in \mathbb{N}$ (independent of input) and numbers $\beta_j \geq 0$ (dependent on input) such that for all $j \in [m]$ we have $w(S_j) \geq 1 - \beta_j$ and $\sum_{j=1}^{m} \beta_j \leq k_0$;
3. for all $k \in \mathbb{N}, y_1, \dots, y_k \in [0, 1]$ we have $\sum_{i=1}^{k} y_i \leq 1$ implies that $\sum_{i=1}^{k} w(y_i) \leq \gamma$.

In the above $\gamma \geq 1$ is a parameter. If such a weight function w exists for a given ALG and with a given γ then we have $\rho(ALG) \leq \gamma$. This is easy to see, since by property 2 we have

$$w([n]) = \sum_{i=1}^{n} w(x_i) = \sum_{i=1}^{m} w(S_i) \geq \sum_{i=1}^{m} (1 - \beta_i) = m - \sum_{i=1}^{m} \beta_i \geq m - k_0.$$

Let Q_i denote the indices of items that are placed in bin i by OPT and suppose that OPT uses m' bins. Then by property 3, we have

$$w([n]) = \sum_{i=1}^{m'} w(Q_i) \leq \gamma m'.$$

Combining the above two inequalities we get that $\gamma m' \geq m - k_0$. That is $ALG \leq \gamma \, \mathrm{OPT} + k_0$, i.e., $\rho(ALG) \leq \gamma$.

We use the weighting technique to prove the following.

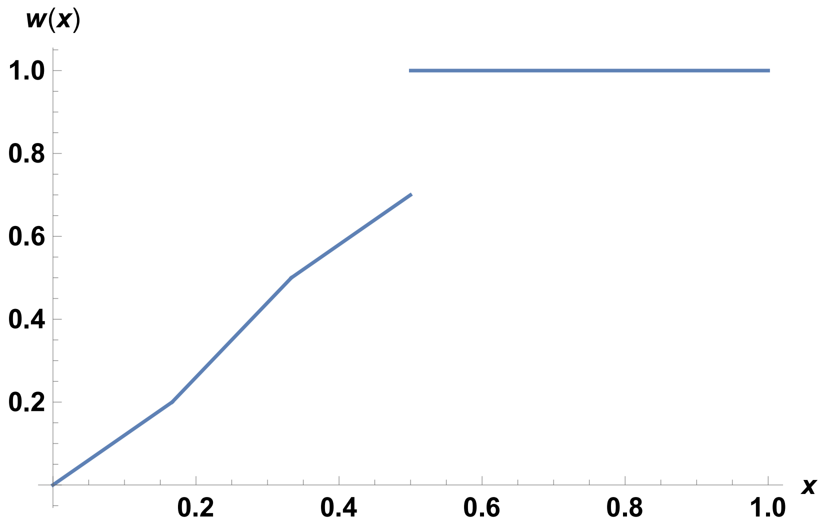

Figure 2.2: The graph of the weight function w used to analyze the competitive ratio of the $FirstFit$ and $BestFit$ algorithms.

Theorem 2.4.2.

$$\rho(FirstFit) \le 17/10; \qquad \rho(BestFit) \le 17/10.$$

The proof follows by verifying the above properties for the following weight function w for $FirstFit$ and $BestFit$:

$$w(x) = \begin{cases} \frac{6}{5}x & \text{for } 0 \le x \le \frac{1}{6}, \\ \frac{9}{5}x - \frac{1}{10} & \text{for } \frac{1}{6} < x \le \frac{1}{3}, \\ \frac{6}{5}x + \frac{1}{10} & \text{for } \frac{1}{3} < x \le \frac{1}{2}, \\ 1 & \text{for } \frac{1}{2} < x \le 1. \end{cases}$$

The first property is easy to check and becomes obvious by looking at the graph of the function w shown in Figure 2.2.

The following lemma establishes the third property. The proof of this lemma is left as an instructive exercise that can be done by a case analysis.

Lemma 2.4.3. *For the w function defined above we have for all $k \in \mathbb{N}$ and all $y_1, \ldots, y_k \in [0, 1]$*

$$\textit{if } \sum_{i=1}^{k} y_i \le 1 \textit{ then } \sum_{i=1}^{k} w(y_i) \le 1.7.$$

Thus, it is left to verify the second property for $FirstFit$ and $BestFit$. Recall that our goal is to show that for every bin i we have $w(S_i) \ge 1 - \beta_i$, such that the sum of β_i converges. Observe that the solutions constructed by $FirstFit$ and $BestFit$ cannot contain two bins that are at most half full (i.e., $R[i] \ge 1/2$) — if two such bins are present, then why weren't the items from the later bin inserted into an earlier bin? Suppose that bin i is the unique bin with $R[i] \ge 1/2$, then we can set $\beta_i = 1$ to guarantee that this bin satisfies the second property. This adds at most 1 to $\sum_j \beta_j$. Thus, we can perform the entire argument with respect to bins that are more than half full. Therefore, from now on we assume that $R[i] < 1/2$ for all $i \in [m]$.

To complete the argument we introduce the idea of *coarseness*. Coarseness of bin i, denoted by α_i, is defined as the maximum remaining space in an earlier bin, i.e., there is an earlier bin $j < i$ with remaining space $R[j] = \alpha_i$. Observe that $0 = \alpha_1 \le \alpha_2 \le \cdots \le \alpha_m < 1/2$. The proof of the second property follows from the following lemma.

Lemma 2.4.4. *1. If $R[i] \le \alpha_i$ then $w(S_i) \ge 1$, i.e., we can set $\beta_i = 0$.*

2. If $R[i] > \alpha_i$ then $w(S_i) \ge 1 - \frac{9}{5}(R[i] - \alpha_i)$, i.e., we can set $\beta_i = \frac{9}{5}(R[i] - \alpha_i)$.

Proof. Let $y_1, \ldots, y_k$ be the weights of the items in bin i. Recall that $B[i] = \sum_{j=1}^{k} y_j$. By renaming variables, if necessary, we can assume that $y_1 \ge y_2 \ge \cdots \ge y_k$.

1. If $y_1 > 1$ then $w(y_1) = 1$ and therefore $w(S_i) \ge 1$. Thus, we can assume that $k \ge 2$. In particular, we must have $y_1 \ge y_2 \ge \alpha_i$ (otherwise, each of the two items could have been placed into an earlier bin corresponding to the coarseness α_i). We consider several cases depending on the range of α_i.

   Case 1: $\alpha_i \le 1/6$. Then we have $B[i] = 1 - R[i] \ge 1 - \alpha_i \ge 5/6$. Observe that on interval $[0, 1/2)$ the slope of w is at least $6/5$, therefore we have $w(S_i)/B[i] \ge 6/5$, so $w(S_i) \ge (6/5)B[i] \ge (6/5)(5/6) = 1$.

   Case 2: $1/3 < \alpha_i < 1/2$. Since $y_1 \ge y_2 > \alpha_i$, we have $y_1, y_2 > 1/3$, so $w(y_1) + w(y_2) \ge 2W(1/3) = 2\left((9/5)(1/3) - 1/10\right) = 1$.

   Case 3: $1/6 < \alpha_i \le 1/3$.

   Subcase (a): $k = 2$. If $y_1, y_2 \ge 1/3$ then it is similar to Case 2 above. Similarly, note that both items cannot be less than $1/3$. Then it only remains to consider $y_1 \ge 1/3 > y_2 > \alpha_i$. Thus, we have

$$w(y_1) + w(y_2) = (6/5)y_1 + 1/10 + (9/5)y_2 - 1/10 = (6/5)(y_1 + y_2) + (3/5)y_2.$$

   Since $y_1 + y_2 \ge 1 - \alpha_i$ and $y_2 > \alpha_i$ we have $w(y_1) + w(y_2) \ge (6/5)(1 - \alpha_i) + 3/5\alpha_i = 1 + 1/5 - 3/5\alpha_i \ge 1$, where the last inequality follows since $\alpha_i \le 1/3$.

   Subcase (b): $k > 2$. As in subcase (a) if $y_1, y_2 \ge 1/3$ then we are done. If $y_1 < 1/3$ then we have

$$\begin{aligned} w(y_1) + w(y_2) + \sum_{j=3}^{k} w(y_j) &= (9/5)(y_1 + y_2) - (1/5) + (6/5)\sum_{j=3}^{k} y_j \\ &\ge (6/5)B[i] + (3/5)(y_1 + y_2) - (1/5) \\ &= (6/5)(1 - \alpha) + (3/5)(2\alpha) - 1/5 = 1. \end{aligned}$$

   If $y_1 \ge 1/3 > y_2$ then we have

$$\begin{aligned} w(y_1) + w(y_2) + \sum_{j=3}^{k} w(y_j) &= (6/5)y_1 + (1/10) + (9/5)y_2 - (1/10) + \sum_{j=3}^{k} w(y_j) \\ &= (6/5)(y_1 + y_2) + (3/5)y_2 + (6/5)\sum_{j=3}^{k} y_j \\ &\ge (6/5)(1 - \alpha) + (3/5)\alpha = 1 + (1/5) - 3/5\alpha \ge 1. \end{aligned}$$

2. We have $B[i] = 1 - R[i] = 1 - \alpha_i + (R[i] - \alpha_i)$. Consider $z_1 = y_1 + (R[i] - \alpha_i)/2$ and $z_2 = y_2 + (R[i] - \alpha_i)/2$. Then we have $z_1 + z_2 + \sum_{j=3}^{k} y_k = 1 - \alpha_i$. Then by the first part of this theorem, we have $w(z_1) + w(z_2) + \sum_{j=3}^{k} w(y_k) \geq 1$. Note that $w(z_1) + w(z_2) \geq w(y_1) + w(y_2) + (9/5)(R[i] - \alpha_i)$ since the slope of w on this range does not exceed $9/5$. Combining the two inequalities we get $w(y_1) + w(y_2) + (9/5)(R[i] - \alpha_i) + \sum_{j=3}^{k} w(y_k) \geq 1$. Collecting the $w(y_j)$ together we get $w(S_i) \geq 1 - (9/5)(R[i] - \alpha_i)$.

□

It is just left to verify that $\sum_{i=1}^{m} \beta_i$ is bounded by an absolute constant. From Lemma 2.4.4 we see that those bins with $R[i] \leq \alpha_i$ do not contribute anything to this sum. Thus, we assume that we only deal with bins such that $R[i] > \alpha_i$ from now on. Observe that for such bins we have

$$\alpha_i \geq R[i-1] = \alpha_{i-1} + (R[i-1] - \alpha_{i-1}) = \alpha_{i-1} + \frac{5}{9}\beta_{i-1}.$$

Alternatively, we write $\beta_{i-1} \leq \frac{9}{5}(\alpha_i - \alpha_{i-1})$. Thus, we have

$$\sum_{i=1}^{m-1} \beta_i = \frac{9}{5}\sum_{i=2}^{m}(\alpha_i - \alpha_{i-1}) = \frac{9}{5}(\alpha_m - \alpha_0) < \frac{9}{5}\cdot\frac{1}{2} < 1.$$

Since $\beta_m \leq 1$ and due to the fact that we have ignored a possible bin that is at most half full, we have established that

$$\sum_{j=1}^{m} \beta_j \leq 3.$$

This finishes the analysis of $FirstFit$ and $BestFit$.

2.5 Competitive Ratio for Maximization Problems

As defined, competitive ratios and approximation ratios for a minimization problem always satisfy $\rho \geq 1$ and equal to 1 if and only the algorithm is (asymptotically) optimal. Clearly, the closer ρ is to 1, the better the approximation.

For maximization problems, there are two ways to state competitive and asymptotic approximation ratios for a maximization algorithm ALG.

1. $\rho(ALG) = \liminf_{OPT(\mathcal{I})\to\infty} \frac{ALG(\mathcal{I})}{OPT(\mathcal{I})}$.
2. $\rho(ALG) = \limsup_{OPT(\mathcal{I})\to\infty} \frac{OPT(\mathcal{I})}{ALG(\mathcal{I})}$.

There is no clear consensus as to which convention to use. In the first definition we always have $\rho \leq 1$. This is becoming more of the standard way of expressing competitive and approximation ratios as the fraction of the optimum value that the algorithm achieves. (For example, see the results in Chapters 7 and 8.) With this convention, we have to be careful in stating results as now an "upper bound" is a negative result and a "lower bound" is a positive result. Using the second definition we would be following the convention for minimization problems where again $\rho \geq 1$ and upper and lower bounds have the standard interpretation for being (respectively) positive and negative results. For both conventions, it is unambiguous when we say, for example, "achieves approximation ratio ..." and "has an inapproximation ratio ...".

2.6 Maximization Problem Example: Time-Series Search

As an example of a deterministic algorithm for a maximization problem, we first consider the following time-series search problem. In this problem one needs to exchange the entire savings in one currency into another, e.g., dollars into euros. Over a period of n days, a new exchange rate p_j is posted on each day $j \in [n]$. The goal is to select a day with maximally beneficial exchange rate and exchange *the entire savings* on that day. If you have not made the trade before day n, you are forced to trade on day n. You might not know n in advance, but you will be informed on day n that it is the last day. Without knowing any side information, an adversary can force any deterministic algorithm to perform arbitrarily badly. There are different variations of this problem depending on what is known about currency rates a-priori. We assume that before seeing any of the inputs, you also have access to an upper bound U and a lower bound L on the p_j, that is $L \le p_j \le U$ for all $j \in [n]$. We also assume no transaction costs. Formally, the problem is defined as follows.

Time-series search
Input: $(p_1, p_2, \ldots, p_n)$ where p_j is the rate for day j meaning that one dollar is equal to p_j euros, $U, L \in \mathbb{R}_{\ge 0}$ such that $L \le p_j \le U$ for all $j \in [n]$
Output: $i \in [n]$
Objective: To compute i so as to maximize p_i.

We introduce a parameter $\phi = U/L$ — the ratio between a maximum possible rate and minimum possible rate. Observe that any algorithm achieves competitive ratio ϕ. Can we do better?

The following deterministic *reservation price* online algorithm is particularly simple and improves upon the trivial ratio ϕ. The algorithm trades all of its savings on the first day that the rate is at least $p^* = \sqrt{UL}$. If the rate is always less than p^*, the algorithm will trade all of its savings on the last day. This algorithm is also known as the reservation price policy or RPP for short.

Algorithm 5 The Reservation price policy

```
procedure RESERVATION PRICE
    $p^* \leftarrow \sqrt{UL}$
    $flag \leftarrow 0$
    $j \leftarrow 1$ and
    while $j \le n$ and $flag = 0$ do
        if $j < n$ and $p_j \ge p^*$ then
            Trade all savings on day $j$
            $flag \leftarrow 1$
        else if $j = n$ then
            Trade all savings on day $n$
```

Theorem 2.6.1. *Let $\phi = \frac{U}{L}$. The reservation price algorithm achieves competitive ratio $\sqrt{\phi}$ whether n is known or unknown in advance.*

Proof. Consider the case where $p_j < p^*$ for all $j \in [n]$. Then the reservation price algorithm trades all dollars into euros on the last day achieving the objective value $p_n \ge L$. The optimum is $\max_j p_j \le p^* = \sqrt{UL}$. The ratio between the two is

$$\frac{OPT}{ALG} \le \frac{\sqrt{UL}}{L} = \sqrt{\phi}.$$

Now, consider the case where there exists $p_j \geq p^*$ and let j be earliest such index. Then the reservation price algorithm trades all dollars into euros on day j achieving objective value $p_j \geq \sqrt{UL}$. The optimum is $\max_j p_j \leq U$. The ratio between the two is

$$\frac{OPT}{ALG} \leq \frac{U}{\sqrt{UL}} = \sqrt{\phi}.$$

□

We can also show that $\sqrt{\phi}$ is optimal among all deterministic algorithms for time-series search.

Theorem 2.6.2. *No deterministic online algorithm for time-series search can achieve competitive ratio smaller than $\sqrt{\phi}$ even if n is known.*

Proof. The adversary specifies $p_i = \sqrt{UL}$ for $i \in [n-1]$. If the algorithm trades on day $i \leq n-1$, the adversary then declares $p_n = U$. Thus, OPT trades on day n. In this case, the competitive ratio is $U/\sqrt{UL} = \sqrt{U/L} = \sqrt{\phi}$.

If the algorithm does not trade on day $i \leq n-1$, the adversary declares $p_n = L$. Thus the algorithm is forced to trade on day n with exchange rate L, while OPT trades on an earlier day with exchange rate $\sqrt{UL}$. In this case, the competitive ratio is $\sqrt{UL}/L = \sqrt{U/L} = \sqrt{\phi}$. □

A similar argument presented in the following theorem shows that knowing just ϕ is not sufficient to improve upon the trivial competitive ratio.

Theorem 2.6.3. *Suppose that instead of U and L only $\phi = \frac{U}{L}$ is known to an algorithm a priori. Then competitive ratio of any algorithm for time-series search is at least ϕ.*

Proof. The adversary declares ϕ and presents the input sequence $(p_1, \dots, p_n)$, where $p_i = 1$ for $i \in [n-1]$ to a reservation price algorithm ALG.

If ALG trades on a day $i \leq [n-1]$, then the adversary declares $p_n = \phi$. In this case, an optimal solution is to trade on day p_n achieving objective value p_n, and the algorithm traded when the exchange rate was 1.

If ALG doesn't trade on day $n-1$ or earlier, then the adversary declares $p_n = 1/\phi$. In this case, an optimal solution is to trade on day 1 (for example) achieving objective value 1, and the algorithm trades on the last day achieving $1/\phi$.

In either case, the adversary can achieve competitive ratio ϕ. □

2.7 Maximization Problem Example: One-Way Trading

A natural generalization of the time-series search is the one-way trading problem. In this problem, instead of requiring the algorithm to trade *all of its savings* on a single day, we allow an algorithm to trade a fraction $f_i \in [0,1]$ of its savings on day i for $i \in [n]$. An additional requirement is that by the end of the last day all of the savings have been traded, that is $\sum_{i=1}^n f_i = 1$. As before, the bounds U and L on exchange rates are known in advance. Also as before, we assume that the algorithm is forced to trade all remaining savings at the specified rate on day n.

One-way currency trading

Input: $(p_1, p_2, \dots, p_n)$ where p_j is the rate for day j meaning that one unit of savings is equal to p_j units of new currency; $U, L \in \mathbb{R}_{\geq 0}$ such that the rates must satisfy $L \leq p_i \leq U$ for each day i.
Output: $f_1, \dots f_n \in [0,1]$ where f_i indicates that the fraction f_i of savings are traded on day i and $\sum_{i=1}^n f_i = 1$.

Objective: To compute f so as to maximize $\max_i f_i p_i$; that is, to maximize the aggregate exchange rate.

The ability to trade a fraction of your savings on each day is surprisingly powerful — one can *almost* achieve competitive ratio $\log \phi$ instead of $\sqrt{\phi}$ achievable without this ability. Algorithm 6 shows how this is done. To simplify the presentation we assume that $\phi = 2^k$ for some positive integer k. Consider exponentially spaced reservation prices of the form $L2^i$ where $i \in \{0, 1, \ldots, k-1\}$. We refer to $L2^i$ as the ith reservation price. Upon receiving p_1, the algorithm computes index i of the largest reservation price that is exceeded by p_1 and trades $(i+1)/k$ fraction of its savings. This index i is recorded in i^*. For each newly arriving exchange rate p_j we compute index i of the reservation price that is exceeded by p_j. If $i \le i^*$ the algorithm ignores day j. Otherwise, the algorithm trades $(i - i^*)/k$ fraction of its savings on day j and updates i^* to i. Thus, we can think of i^* as keeping track of the best reservation price that has been exceeded so far, and whenever we have a better reservation price being exceeded we trade the fraction of savings proportional to the difference between indices of the two reservation prices. Thus, the algorithm is computing some kind of *mixture of reservation price policies*, and it is called the Mixture of RPPs algorithm.

Algorithm 6 The Mixture of RPPs

```
procedure RESERVATION PRICE
    ▷ $U, L$, and $\phi = U/L = 2^k$ are known in advance
    $i^* \leftarrow -1$
    for $j \leftarrow 1$ to $n$ do
        $i \leftarrow \max\{i \mid L2^i \le p_j\}$
        if $i = k$ then
            $i \leftarrow k - 1$
        if $i > i^*$ then
            Trade fraction $(i - i^*)/k$ of savings on day $j$
            $i^* \leftarrow i$
    Trade all remaining savings on day $n$
```

Theorem 2.7.1. *The competitive ratio of the Mixture of RPPs algorithm is $c(\phi)\log \phi$ where $c(\phi)$ is a function such that $c(\phi) \to 1$ as $\phi \to \infty$.*

Proof. Consider the input sequence $(p_1, \ldots, p_n)$ and let $i_1, \ldots, i_n$ be the indices of the corresponding reservation prices. That is, i_j is the largest index such that $L2^{i_j} \le p_j$ for all $j \in [n]$. Let ℓ be the day number with the highest exchange rate, that is $p_\ell = \max\{p_j\}$. Clearly, $OPT = p_\ell \le L2^{i_\ell + 1}$.

Note that the i^*s form a non-decreasing sequence during the execution of the algorithm. Consider only those values of i^* that actually change value, and let $i_0^* < i_1^* < \cdots < i_p^*$ denote that sequence of values. We have $i_0^* = -1$ and $i_p^* = i_\ell$. The algorithm achieve at least the following value of the objective function:

$$\sum_{j=1}^{p} \frac{i_j^* - i_{j-1}^*}{k} L2^{i_j^*} + \frac{k - i_\ell}{k} L,$$

where the first term is the lower bound on the contribution of trades until day ℓ and the second term is the contribution of trading the remaining savings on the last day.

In order to bound the first term, we note that if we wish to minimize $\sum_{j=1}^{p}(i_j^* - i_{j-1}^*)2^{i_j^*}$ (E) over all increasing sequences i_j^* with $i_0^* = -1$ and $i_p^* = i_\ell$ then we have $i_j^* = j - 1$. That is the unique minimizer of expression (E) is the sequence $-1, 0, 1, 2, \ldots, i_\ell$, i.e., it doesn't skip any values. In this case we have $\sum_{j=1}^{p}(i^*_j - i^*_{j-1})2^{i^*_j} = \sum_{j=0}^{i_\ell} 2^j = 2^{i_\ell+1} - 1$. Why is this a minimizer? We will show that an increasing sequence that skips over a particular value v cannot be a minimizer. Suppose that you have a sequence such that $i^*_{j-1} < v < i^*_j$ and consider the jth term in (E) corresponding to this sequence. It is $(i^*_j - i^*_{j-1})2^{i^*_j} = (i^*_j - v + v - i^*_{j-1})2^{i^*_j} = (i^*_j - v)2^{i^*_j} + (v - i^*_{j-1})2^{i^*_j} > (i^*_j - v)2^{i^*_j} + (v - i^*_{j-1})2^{v}$ — that is if we change our sequence to include v we strictly decrease the value of (E). Thus, the unique minimizing sequence is the one that doesn't skip any values.

From the above discussion we conclude that we can lower bound ALG as follows:

$$ALG \geq \frac{2^{i_\ell+1} - 1}{k} L + \frac{k - i_\ell}{k} L.$$

Finally, we can bound the competitive ratio:

$$\frac{OPT}{ALG} = \frac{L2^{i_\ell+1}}{(2^{i_\ell+1} - 1)L/k + (k - i_\ell)L/k} = k\frac{2^{i_\ell+1}}{2^{i_\ell+1} + k - i_\ell - 1}.$$

The worst-case competitive ratio is obtained by maximizing the above expression. We can do so analytically (taking derivatives, equating to zero, etc.), which gives $i_\ell = k - 1 + 1/\ln(2)$. Thus, the competitive ratio is $\log \phi = k$ times a factor that is slightly larger than 1 and approaching 1 as $k \to \infty$. □

We saw that with time-series search knowing U or L was crucial and knowing just ϕ was not enough. It turns out that for one-way trading one can prove a similar result to the above assuming that the algorithm only knows ϕ and doesn't know U or L. One of the exercises at the end of this chapter deals is dedicated to this generalization. Another generalization is that we don't need to assume that ϕ is a power of 2. Proving this is tedious and has low pedagogical value, thus we state it here without a proof.

2.8 Exercises

1. Consider the makespan problem for temporary jobs, where now each job has both a load p_j and a duration d_j. When a job arrives, it must be scheduled on one of the m machines and remains on that machine for d_j time units after which it is removed. The makespan of a machine is the maximum load of the machine at any point in time. As for permanent jobs, we wish to minimize (over all machines) the maximum makespan.
   Show that the greedy algorithm provides a 2-approxmimation for this problem.

2. Suppose that in the time-series search problem the algorithm only knows L beforehand. Does there exist a deterministic algorithm with competitive ratio better than ϕ?

3. Suppose that in the one-way trading problem the algorithm only knows L beforehand. Does there exist a deterministic algorithm with competitive ratio better than ϕ?

4. (HARD) Suppose that in the one-way trading problem the algorithm only knows ϕ beforehand. Design a deterministic algorithm with competitive ratio as close to $\log \phi$ as possible.

5. Prove Lemma 2.4.3.

6. Which properties of $FirstFit$ and $BestFit$ were needed for the proof of Theorem 2.4.2? Can you find any other algorithms with those properties? That is find an algorithm other than $FirstFit$ and $BestFit$ such that the proof of Theorem 2.4.2 applies to the new algorithm *without any modifications.*

7. Prove that $\rho(FirstFit) \geq 1.7$ and that $\rho(BestFit) \geq 1.7$ (give an adversarial strategy).

8. Prove that no online algorithm can achieve competitive ratio better than 4/3 for the Bin Packing problem.

2.9 Historical Notes and References

The makespan problem for identical machines was studied by Graham [7, 8]. These papers present online and offline greedy approximation algorithms for the makespan problem on identical machines as well as presenting some surprising anomolies. The papers preceed Cook's seminal paper [4] introducing NP completeness but still Graham conjectures that it will not be possible to have an efficient optimal algorithm for this problem. This work also preceeds the explicit introduction of competitive analysis by Sleator and Tarjan [13] and does not emphasize the online nature of the greedy algorithm but still the $2 - \frac{1}{m}$ appears to be the first approximation and competitive bound to be published.

The makespan problem belongs to a wider class of scheduling problems called load balancing problems, including other performance measures (e.g., where the laod on a machine is its L_p norm for $p \geq 1$) as well as other measures, and scheduling and routing problems. In particular, the so-called "Santa Claus problem" [2] considers the $max - min$ performance measure for scheduling jobs the unrelated machines model (see Chapter 4. The name of the problem derives from the motivation of distributing n presents amongst m children (where now $p_{i,j}$ is the value of the j^{th} present when given to the i^{th} child) so as to maximize the the value of the least happy child.

We will be addiing more historical remarks based on the following references

- [3]
- [5]
- [11]
- [9]
- [10]

Chapter 3

Randomized Online Algorithms

The central theme of this chapter is how much randomness can help in solving an online problem. To answer this question, we need to extend the notion of competitive ratio to randomized algorithms. Measuring the performance of randomized algorithms is not as straightforward as measuring the performance of deterministic algorithms, since randomness allows for different kinds of adversaries. We look at the classical notions of **oblivious**, **adaptive online**, and **adaptive offline** adversaries, and explore relationships between them. We cover Yao's minimax theorem, which is a useful technique for proving lower bounds against an oblivious adversary. Lastly, we briefly discuss some issues surrounding randomness as an expensive resource and the topic of derandomization.

3.1 Randomized Online Algorithm Template

We recall that according to our convention random variables are denoted by capital letters and particular outcomes of random variables are denoted by small case letters. For example, suppose that B is the Bernoulli random variable with parameter p. Then B is 1 with probability p and 0 with probability $1-p$, and when we write B the outcome hasn't been determined yet. When we write b we refer to the outcome of sampling from the distribution of B, thus b is fixed to be either 0 or 1. In other words B is what you know about the coin before flipping it, and b is what you see on the coin after flipping it once.

A randomized online algorithm generalizes the deterministic paradigm by allowing the decision in step 5 of the template to be a randomized decision. We view the algorithm as having access to an infinite tape of random bits. We denote the contents of the tape by R, i.e., $R_i \in \{0,1\}$ for $i \geq 1$, and the R_i are distributed uniformly and independently of each other.

```
   **Randomized Online Algorithm Template**
1: $R \leftarrow$ infinite tape of random bits
2: On an instance $I$, including an ordering of the data items $(x_1, \ldots, x_n)$:
3: $i := 1$
4: **While** there are unprocessed data items
5:     The algorithm receives $x_i$ and makes an irrevocable randomized decision $D_i := D_i(x_1, \ldots, x_i, R)$ for $x_i$
       (based on $x_i$, all previously seen data items, and $R$).
6:     $i := i + 1$
7: **EndWhile**
```

Remark 3.1.1. You might wonder if having access to random bits is enough. After all, we often want random variables distributed according to more complicated distributions, e.g., Gaussian with parameters μ and σ. Turns out that you can model any reasonable random variable to any desired accuracy with access to R only. For example, if you need a Binomial random variable with parameters $1/2$ and n, you can write a subprocedure that returns $\sum_{i=1}^{n} R_i$. If you need a new independent sample from that distribution, you can use fresh randomness from another part of the string R. This can be done for all other standard distributions — Bernoulli with parameter p (how?), Binomial with parameters p and n, exponential, Gaussian, etc. We will often skip the details of how to obtain a particular random variable from R and simply assume that we have access to the corresponding subroutine.

Note that the decision in step 4 is now a function not only of all the previous inputs but also of randomness R. Thus each decision D_i is a random variable. However, if we fix R to be particular infinite binary string $r \in \{0,1\}^{\mathbb{N}}$, each decision becomes deterministic. This way, we can view an online randomized algorithm ALG as a distribution over deterministic online algorithms ALG_r indexed by randomness r. Then ALG samples r from $\{0,1\}^{\mathbb{N}}$ uniformly at random and runs ALG_r. This viewpoint is essential for the predominant way to prove inappoximation results for randomized algorithm, namely the use of the von Neumann-Yao principle.

3.2 Types of Adversaries

For this section we recall the view of an execution of an online algorithm as a game between an adversary and the algorithm. In the deterministic case, there is only one kind of adversary. In the randomized case, we distinguish between three different kinds of adversaries: **oblivious**, **adaptive offline**, and **adaptive online**, depending on the information that is available to the adversary when it needs to create the next input item.

Oblivious adversary: this is the weakest kind of an adversary that only knows the (pseudocode of the) algorithm, but not the particular random bits r that are used by the algorithm. The adversary has to come up with the input sequence $x_1, x_2, \ldots, x_n$ in advance — before learning any of the decisions made by the online algorithm on this input. Thus, the oblivious adversary knows the *distribution* of $D_1, D_2, \ldots, D_n$, but it doesn't know which particular decisions $d_1, d_2, \ldots, d_n$ are going to be taken by the algorithm. Let $OBJ(x_1, \ldots, x_n, d_1, \ldots, d_n)$ be the objective function. The performance is measured as the ratio between the expected value of the objective achieved by the algorithm to the offline optimum on $x_1, \ldots, x_n$. More formally it is

$$\frac{\mathbb{E}_{D_1,\ldots,D_n}\left(OBJ(x_1, \ldots, x_n, D_1, \ldots, D_n)\right)}{OPT(x_1, \ldots, x_n))}.$$

Observe that we don't need to take the expectation of the optimum, because input items $x_1, \ldots, x_n$ are *not random.*

Adaptive offline adversary: this is the strongest kind of an adversary that knows the (pseudocode of the) algorithm and its online decisions, but not R. Thus, the adversary creates the first input item x_1. The algorithm makes a decision D_1 and the adversary learns the outcome d_1 prior to creating the next input item x_2. We can think of the input items as being defined recursively $x_i := x_i(x_1, d_1, \ldots, x_{i-1}, d_{i-1})$. After the entire input sequence is created we compare the performance of the algorithm to that of an optimal offline algorithm that knows the entire sequence $x_1, \ldots, x_n$ in advance. More formally it is

$$\frac{\mathbb{E}_{D_1,\ldots,D_n}\left(OBJ(x_1, \ldots, x_n, D_1, \ldots, D_n)\right)}{\mathbb{E}_{D_1,\ldots,D_n}(OPT(x_1, \ldots, x_n))}.$$

Observe that we have to take the expectation of the optimum in this case, because input items $x_1, \ldots, x_n$ are random as they depend on $D_1, \ldots, D_n$ (implicit in our notation).

Adaptive online adversary: this is an intermediate kind of an adversary that creates an input sequence and an output sequence adaptively. As before the adversary knows the (pseudocode of the) algorithm, but not R. The adversary creates the first input item x_1 and makes its own decision on this item d'_1. The algorithm makes a random decision D_1, the outcome d_1 of which is then revealed to the adversary. The adversary then comes up with a new input item x_2 and its own decision d'_2. Then the algorithm makes a random decision D_2, the outcome d_2 of which is then revealed to the adversary. And so on. Thus, the order of steps is as follows: $x_1, d'_1, d_1, x_2, d'_2, d_2, x_3, d'_3, d_3, \ldots$ We say that the adaptive online adversary can create the next input item based on the previous decisions of the algorithm, but *it has to serve this input item immediately itself.* The performance of an online algorithm is measured by the ratio of the objective value achieved by the adversary versus the objective value achieved by the algorithm. More formally, it is

$$\frac{\mathbb{E}_{D_1,\ldots,D_n}\left(OBJ(x_1,\ldots,x_n,D_1,\ldots,D_n)\right)}{\mathbb{E}_{D_1,\ldots,D_n}\left(OBJ(x_1,\ldots,x_n,d'_1,\ldots,d'_n)\right)}.$$

Observe that we have to take the expectation of the objective value achieved by the adversary, since both input items $x_1, \ldots, x_n$ and adversary's decisions $d'_1, \ldots, d'_n$ depend on random decisions of the algorithm $D_1, \ldots, D_n$ (implicit in our notation).

Based on the above description one can easily define competitive ratios for these different kinds of adversaries for both maximization and minimization problems (using lim infs and lim sups in the obvious way). We denote the competitive ratio achieved by a randomized online algorithm ALG with respect to the oblivious adversary, adaptive offline adversary, and adaptive online adversary by $\rho_{\text{OBL}}(ALG), \rho_{\text{ADOFF}}(ALG)$, and $\rho_{\text{ADON}}(ALG)$, respectively.

The most popular kind of adversary in the literature is the oblivious adversary. When we analyze randomized online algorithms we will assume the oblivious adversary unless stated otherwise. The oblivious adversary often makes the most sense from a practical point of view, as well. This happens, when performance of the algorithm is independent of the input. We already saw it for the ski rental problem. Whether you decide to buy or rent skis should (logically) have no affect on the weather — this is precisely modelled by an oblivious adversary. However, there are problems for which decisions of the algorithm can affect the behaviour of the future inputs. This happens, for example, for paging. Depending on which pages are in the cache, the future pages will either behave as cache misses or as cache hits. In addition, one can write programs that alter their behaviour completely depending on cache miss or cache hit. One real-life example of such (nefarious) programs are Spectre and Meltdown that use cache miss information together with speculative execution to gain read-only access to protected parts of computer memory. Thus, there are some real-world applications which are better modelled by adaptive adversaries, since decisions of the algorithm can alter the future input items.

3.3 Relationships between Adversaries

We start with a basic observation that justifies calling oblivious, adaptive offline, and adaptive online adversaries as weak, strong, and intermediate, respectively.

Theorem 3.3.1. *For a minimization problem and a randomized online algorithm ALG we have*

$$\rho_{OBL}(ALG) \leq \rho_{ADON}(ALG) \leq \rho_{ADOFF}(ALG).$$

An analogous statement is true for maximization problems.

The following theorem says that the adaptive offline adversary is so powerful that any randomized algorithm running against it cannot guarantee a better competitive ratio than the one achieved by deterministic algorithms.

Theorem 3.3.2. *Consider a minimization problem given as a request-answer game. Assume that the set of possible answers/decisions is finite (e.g., Ski Rental) and consider a randomized online algorithm ALG for it. Then there is a deterministic online algorithm ALG' such that*

$$\rho(ALG') \leq \rho_{ADOFF}(ALG).$$

An analogous statement is true for maximization problems.

Proof. We refer to $(x_1, \ldots, x_k, d_1, \ldots, d_k)$ as a *position in the game*, where the x_i are input items, provided by an adversary, and d_i are decisions, provided by an algorithm. We say that a position $(x_1, \ldots, x_k, d_1, \ldots, d_k)$ is *immediately winning for adversary* if $f_k(x_1, \ldots, x_k, d_1, \ldots, d_k) > \rho_{\text{ADOFF}}(ALG)OPT(x_1, \ldots, x_k)$, where f_k is the objective function. We call a position *winning for adversary* if there exists $t \in \mathbb{N}$ and an adaptive strategy of choosing requests such that an immediately winning position is reached *no matter what answers are chosen by an algorithm* within t steps.

Note that the initial empty position cannot be a winning position for the adversary. Suppose that it was, for contradiction. The randomized algorithm ALG is a distribution on deterministic algorithms ALG_z for some $z \sim Z$. If the initial empty position was winning for the adversary, then for every z we would have a sequence of requests and answers (depending on z) such that $ALG_z(I_z) > \rho_{\text{ADOFF}}(ALG)OPT(I_z)$. Taking the expectation of both sides, we get $\mathbb{E}_Z(ALG_Z(I_Z)) > \rho_{\text{ADOFF}}(ALG)\mathbb{E}_Z(OPT(I_Z))$, contradicting the definition of $\rho_{\text{ADOFF}}(ALG)$.

Observe that a position $(x_1, \ldots, x_n, d_1, \ldots, d_n)$ is winning if and only if there exists x_{n+1} such that for all d_{n+1} the position $(x_1, \ldots, x_n, x_{n+1}, d_1, \ldots, d_n, d_{n+1})$ is also winning. Thus, if a position is not winning, then for any new input item x_{n+1} there is a decision d_{n+1} that leads to a position that is also not winning. This precisely means that there is a deterministic algorithm ALG' that can keep the adversary in a non-winning position for as long as needed. Since the game has to eventually terminate, it will terminate in a non-winning position, meaning that after any number t of steps of the game we have $f_t(x_1, \ldots, x_t, d_1, \ldots, d_t) \leq \rho_{\text{ADOFF}}OPT(x_1, \ldots, x_t)$, where d_i are the *deterministic* choices provided by ALG. □

The gap between offline adaptive adversary and online adaptive adversary can be at most quadratic.

Theorem 3.3.3. *Consider a minimization problem and a randomized online algorithm ALG for it. Then*

$$\rho_{ADOFF}(ALG) \leq \left(\rho_{ADON}(ALG)\right)^2.$$

An analogous statement is true for maximization problems.

Proof. Let ADV be an arbitrary adaptive offline adversary against ALG. Let R denote the randomness used by ALG, and let R' be its independent copy. Then we can represent ALG as a distribution on deterministic algorithms ALG_r where $r \sim R$. Let $x(R)$ be requests given by ADV when it runs against ALG_R. Let $d(R)$ denote ALG_R responses when it runs against ADV.

Consider a fixed value of r and the well-defined sequence of requests $x(r)$. In order to avoid ambiguity, we are going to label randomness of ALG by a copy of R, namely R'. Since $ALG_{R'}$ is $\rho_{\text{ADON}}(ALG)$-competitive against any adaptive online adversary, it is also $\rho_{\text{ADON}}(ALG)$-competitive against any oblivious adversary (by Theorem 3.3.1). In particular $ALG_{R'}$ is ρ_{ADON}-competitive against the oblivious adversary that presents request sequence $x(r)$:

$$\mathbb{E}_{R'}(ALG_{R'}(x(r))) \le \rho_{\text{ADON}}(ALG)OPT(x(r)).$$

Taking the expectation of both sides with respect to r we get

$$\mathbb{E}_R\mathbb{E}_{R'}(ALG_{R'}(x(R))) \le \rho_{\text{ADON}}(ALG)\mathbb{E}_R(OPT(x(R))). \tag{3.1}$$

Let f denote the objective function. Now, consider a fixed value of r'. Define an adaptive online strategy working against ALG_R that produces a request sequence $x(R)$ and provides its own decision sequence $d(r')$, while ALG_R provides decision sequence $d(R)$. Since ALG is $\rho_{\text{ADON}}(ALG)$-competitive against this adaptive online strategy, we have:

$$\mathbb{E}_R(ALG_R(x(R)) \le \rho_{\text{ADON}}(ALG)\mathbb{E}_R(f(x(R), d(r'))).$$

Taking the expectation of both sides with respect to r' we get

$$\mathbb{E}_R(ALG_R(x(R)) \le \rho_{\text{ADON}}(ALG)\mathbb{E}_{R'}\mathbb{E}_R(f(x(R), d(R'))).$$

The right hand side can be written as $\mathbb{E}_{R'}\mathbb{E}_R(f(x(R), d(R'))) = \mathbb{E}_R\mathbb{E}_{R'}(ALG_{R'}(x(R))$. Combining this with (3.1) we get

$$\mathbb{E}_R(ALG_R(x(R))) \le \rho_{\text{ADON}}(ALG)^2\mathbb{E}(OPT(x(R))).$$

The left hand side is the expected cost of the solution produced by ALG running against *the adaptive offline adversary ADV*. □

In the following section we establish that the gap between ρ_{OBL} and ρ_{ADOFF} (as well as between ρ_{OBL} and ρ_{ADON}) can be arbitrary large.

3.4 How Much Can Randomness Help?

We start by showing that the gap between the best competitive ratio achieved by a randomized algorithm and a deterministic algorithm can be arbitrary large. We begin by fixing a particular gap function $g : \mathbb{N} \to \mathbb{R}$. Consider the following maximization problem:

Modified Bit Guessing Problem
Input: $(x_1, x_2, \ldots, x_n)$ where $x_i \in \{0, 1\}$.
Output: $z = (z_1, z_2, \ldots, z_n)$ where $z_i \in \{0, 1\}$
Objective: To find z such that $z_i = x_{i+1}$ for some $i \in [n-1]$. If such i exists the payoff is $g(n)/(1 - 1/2^{n-1})$, otherwise the payoff is 1.

In this problem, the adversary presents input bits one by one and the goal is to guess the bit arriving in the next time step based on the past history. If the algorithm manages to guess at least one bit correctly, it receives a large payoff of $g(n)/(1 - 1/2^{n-1})$, otherwise it receives a small payoff of 1.

Theorem 3.4.1. *Every deterministic algorithm ALG achieves objective value* 1 *on the Modified Bit Guessing Problem.*

There is a randomized algorithm that achieves expected objective value $g(n)$ against an oblivious adversary on inputs of length n for the Modified Bit Guessing Problem.

Proof. For the first part of the theorem consider a deterministic algorithm ALG. The adversarial strategy is as follows. Present $x_1 = 0$ as the first input item. The algorithm replies with z_1. The adversary defines $x_2 = \neg z_1$. This continues for $n-2$ more steps. In other words, the adversary defines $x_i = \neg x_{i-1}$ for $i = \{2, \ldots, n\}$ making sure that the algorithm does not guess any of the bits. Thus, the algorithm achieves objective function value 1.

Consider the randomized algorithm that selects z_i uniformly at random. The probability that it picks $z_1, \ldots, z_{n-1}$ to be different from $x_2, \ldots, x_n$ in each coordinate is exactly $1/2^{n-1}$. Therefore with probability $1 - 1/2^{n-1}$ it guesses at least one bit correctly. Therefore the expected value of the objective function is at least $g(n)/(1 - 1/2^{n-1}) \cdot (1 - 1/2^{n-1}) = g(n)$. □

Corollary 3.4.2. *The gap between ρ_{OBL} and ρ_{ADOFF} can be arbitrarily large.*

Thus, there are problems for which randomness helps a lot. What about another extreme? Are there problems where randomness does not help at all? It turns out "yes" and, in fact, we have already seen such a problem, namely, the One-Way Trading problem.

Theorem 3.4.3. *Let ALG be a randomized algorithm for the One-Way Trading problem. Then there exists a deterministic algorithm ALG' for the One-Way Trading problem such that*

$$\rho(ALG') \leq \rho_{OBL}(ALG).$$

Proof. Recall that ALG is a distribution on deterministic algorithms ALG_R indexed by randomness R. For each r consider the deterministic algorithmALG_r running on the input sequence $p_1, \ldots, p_n$. Let $f_i(r, p_1, \ldots, p_i)$ be the fraction of savings exchanged on day i. We can define the *average* fraction of savings exchanged on day i, where the average is taken over all deterministic algorithms in the support of ALG. That is

$$\widetilde{f}_i(p_1, \ldots, p_i) := \int f_i(r, p_1, \ldots, p_i)\, dr.$$

Observe that $\widetilde{f}_i(p_1, \ldots, p_i) \geq 0$ and moreover

$$\sum_{i=1}^{n} \widetilde{f}_i(p_1, \ldots, p_i) = \sum_{i=1}^{n} \int f_i(p_1, \ldots, p_i)\, dr = \int \sum_{i=1}^{n} f_i(p_1, \ldots, p_i)\, dr = \int 1\, dr = 1.$$

Therefore, $\widetilde{f}_i$ form valid fractions of savings to be traded on n days. The fraction $\widetilde{f}_i$ depends only on $p_1, \ldots, p_i$ and is independent of randomness r. Thus, these fractions can be computed by a deterministic algorithm (with the knowledge of ALG) in the online fashion. Let ALG' be the algorithm that exchanges $\widetilde{f}_i(p_1, \ldots, p_i)$ of savings on day i. It is left to verify the competitive ratio of ALG'. On input $p_1, \ldots, p_n$ it achieves the value of the objective

$$\begin{aligned}\sum_{i=1}^{n} p_i \widetilde{f}_i(p_1, \ldots, p_i) &= \sum_{i=1}^{n} p_i \int f_i(r, p_1, \ldots, p_i)\, dr = \int \sum_{i=1}^{n} p_i f_i(r, p_1, \ldots, p_i)\, dr \\ &= \mathbb{E}_R(ALG_R(p_1, \ldots, p_n)) \geq OPT(p_1, \ldots, p_n)/\rho_{\text{OBL}}(ALG).\end{aligned}$$

□

The Modified String Guessing problem provides an example of a problem where using randomness improves competitive ratio significantly. Notice that the randomized algorithm uses n bits of randomness to achieve this improvement. Next, we describe another extreme example, where a *single bit* of randomness helps improve the competitive ratio.

Proportional Knapsack

Input: $(w_1, \ldots, w_n)$ where $w_i \in \mathbb{R}_{\geq 0}$; W — bin weight capacity, known in advance.
Output: $z = (z_1, z_2, \ldots, z_n)$ where $z_i \in \{0, 1\}$
Objective: To find z such that $\sum_{i=1}^n z_i w_i$ is maximized subject to $\sum_{i=1}^n z_i w_i \leq W$.

In this problem the goal is to pack maximum total weight of items into a single knapsack of weight capacity W (known in advance). Item i is described by its weight w_i. For item i the algorithm provides a decision z_i such that $z_i = 1$ stands for packing item i, and $z_i = 0$ stands for ignoring item i. If an algorithm produces an infeasible solution (that is total weight of packed items exceeds W), the payoff is $-\infty$. Thus, without loss of generality, we assume that an algorithm never packs an item that makes the total weight exceed W. Our first observation is that deterministic algorithms cannot achieve constant competitive ratios.

Theorem 3.4.4. *Let $\epsilon > 0$ be arbitrary and let ALG be a deterministic online algorithm for the Proportional Knapsack problem. Then we have*

$$\rho(ALG) \geq \frac{1-\epsilon}{\epsilon}.$$

Proof. Let $n \in \mathbb{N}$. We describe an adversarial strategy for constructing inputs of size n. First, let $W = n$. Then the adversary presents inputs ϵn until the first time ALG packs such an input. If ALG never packs an input item of weight ϵn, then ALG packs total weight 0, while $OPT \geq \epsilon n$, which leads to an infinitely large competitive ratio.

Suppose that ALG packs $w_i = \epsilon n$ for the first time for some $i < n$. Then the adversary declares $w_{i+1} = n(1-\epsilon) + \epsilon$ and $w_j = 0$ for $j > i + 1$. Therefore ALG cannot pack w_{i+1} since $w_i + w_{i+1} = n + \epsilon > W$. Moreover, packing any of w_j for $j > i + 1$ doesn't affect the value of the objective function. Thus, we have $ALG = \epsilon n$, whereas $OPT = w_{i+1} = n(1-\epsilon) + \epsilon$. We get the competitive ratio of $\frac{n(1-\epsilon)+\epsilon}{\epsilon n} \geq \frac{n(1-\epsilon)}{\epsilon n} = \frac{1-\epsilon}{\epsilon}$. □

Next we show that a randomized algorithm, which we call SimpleRandom, that uses only 1 bit of randomness achieves competitive ratio 4. Such 1 bit randomized algorithms have been termed "barely random". Algorithm 7 provides a pseudocode for this randomized algorithm. The algorithm has two modes of operation. In the first mode, the algorithm packs items greedily — when a new item arrives, the algorithm checks if there is still room for it in the bin and if so packs it. In the second mode, the algorithm waits for an item of weight $\geq W/2$. If there is such an item, the algorithm packs it. The algorithm ignores all other weights in the second mode. The algorithm then requires a single random bit B, which determines which mode the algorithm is going to use in the current run.

Algorithm 7 Simple randomized algorithm for Proportional Knapsack

```
procedure SimpleRandom
    Let $B \in \{0,1\}$ be a uniformly random bit        ▷ $W$ is the knapsack weight capacity
    if $B = 0$ then
        Pack items $w_1, \ldots, w_n$ greedily, that is if $w_i$ still fits in the remaining weight knapsack capacity, pack it; otherwise, ignore it.
    else
        Pack the first item of weight $\geq W/2$ if there is such an item. Ignore the rest of the items.
```

Theorem 3.4.5.

$$\rho_{OBL}(SimpleRandom) \leq 4.$$

Proof. The goal is to show that $OPT \leq 4\mathbb{E}(\text{SimpleRandom})$ on any input sequence $w_1, \ldots, w_n$. We distinguish two cases.

Case 1: for all $i \in [n]$ we have $w_i < W/2$. Subcase 1(a): $\sum_{i=1}^{n} w_i \leq W$. In this subcase, SimpleRandom running in the first mode packs all of the items. This happens with probability 1/2, thus we have $\mathbb{E}(\text{SimpleRandom}) \geq 1/2 \sum_i w_i$ and $OPT = \sum_i w_i$. Therefore, it follows that $OPT \leq 2\mathbb{E}(\text{SimpleRandom})$ in this subcase. Subcase 1(b): $\sum_i w_i > W$. Consider SimpleRandom running in the first mode again. There is an item that SimpleRandom does not pack in this case. Let w_i be the first item that is not packed. The reason w_i is not packed is that the remaining free space is less than w_i, but we also know that $w_i < W/2$. This means that SimpleRandom has packed total weight at least $W/2$ by the time w_i arrives. Since SimpleRandom runs in the first mode with probability 1/2 we have that $\mathbb{E}(\text{SimpleRandom}) \geq (1/2)(W/2) = W/4 \geq OPT/4$, where the last inequality follows from the trivial observation that $OPT \leq W$. Rearranging we have $OPT \leq 4\mathbb{E}(\text{SimpleRandom})$ in this subcase.

Case 2: there exists $i \in [n]$ such that $w_i \geq W/2$. Consider SimpleRandom running in the second mode: it packs the first w_i such that $w_i \geq W/2$. Since SimpleRandom runs in the second mode with probability 1/2 we have $\mathbb{E}(\text{SimpleRandom}) \geq (1/2)(W/2) = W/4 \geq OPT/4$. Thus, it follows that $OPT \leq 4\mathbb{E}(\text{SimpleRandom})$.

This covers all possibilities. We got that in all cases the competitive ratio of SimpleRandom is at most 4. □

3.5 Derandomization

Randomness is a double-edged sword. On one hand, when we are faced with a difficult problem for which no good deterministic algorithm exists, we do hope that adding randomness would allow one to design a much better (and often simpler) randomized algorithm. On another hand, when we have an excellent randomized algorithm, we hope that we can remove its dependence on randomness, since randomness as a resource is quite expensive. If we can design a deterministic algorithm with the same guarantees (e.g., competitive ratio) as the randomized algorithm ALG, we say that ALG has been "derandomized." Whether all algorithms can be derandomized or not depends on the computational model. For example, we have seen that the general-purpose derandomization is not possible for online algorithms versus an oblivious adversary, but it is possible for online algorithms versus an adaptive offline adversary. In the Turing machine world, it is a big open problem whether all of bounded-error polynomial time algorithms (i.e., the class BPP) can be derandomized or not. Many believe that such derandomization of BPP should be possible. When general-purpose derandomization is not possible, it is still an interesting question to see if derandomization is possible for a particular problem. We saw one such example for the One-Way Trading problem. Derandomization is a big topic and it will come up several times in this book.

In the remainder of this section, we would like to discuss why randomness as a resource can be expensive. Best scientists in human history have argued whether "true randomness" exists, or if randomness is simply a measure of our ignorance. The latest word on the subject is given by quantum mechanics, which says that, indeed, to the best of our understanding of how the world works there are truly random events in the nature. In principle, one could build machines that generate truly random bits based on quantum effects, but there are no cheap commercially available solutions like that at this moment (to the best of our knowledge). Instead, random bit generators implemented on the off-the-shelf devices are pseudo-random. The pseudo-random bits can come from different sources — they can be mathematically generated, or they can be generated from some physical processes, such as coordinates of the latest mouse click on the desktop, or voltage noise in the CPU. Standard programming libraries take some combination of these techniques to produce random-looking bits. How can one detect if the bits that are being generated are truly random or pseudo-random? For that we could write a program, called a test, that receives a string of bits and outputs either "YES" for truly random or "NO" for pseudo-random. The test could be as simple as checking sample moments (mean and variance, for example) of the incoming bits and seeing if it falls not too far from the true moments of the distribution. The test could also compute autocorrelation, and so on. Typically, it is not hard to come up with pseudo-random generators that would "fool" such statistical tests to believe that the bits are truly random. But it is again a big open problem to come up with a pseudo-random generator that would provably fool *all* reasonable tests. Fortunately, typically all we need for a randomized algorithm to run correctly is for the pseudo-random bits to pass statistical tests. This is why Quicksort has an excellent empirical performance even with pseudo-random generators. In addition, even if the bits are not truly random the only side-effect that your program might experience is a slight degradation in performance, which is not critical. However, there are cases where breaking truly random guarantee can result in catastrophic losses. This often happens in security, where using pseudo-random tactics (such as generating randomness based on mouse clicks) introduces a vulnerability in the security protocol. Since this book doesn't deal with the topic of security, we will often permit ourselves to use randomness freely. Nonetheless, we realize that random bits might be expensive and we shall often investigate whether a particular randomized algorithm can be derandomized or not.

3.6 Lower Bound for Paging

In this section we revisit the Paging problem. We shall prove a lower bound on the competitive ratio achieved by any randomized algorithm. In the process, we shall discover a general-purpose technique called Yao's minimax principle. In the following section, we will formally state and discuss the principle. To state the result we need to introduce the *nth harmonic number.*

Definition 3.6.1. The nth Harmonic number, denoted by H_n, is defined as

$$H_n = 1 + \frac{1}{2} + \cdots + \frac{1}{n} = \sum_{i=1}^{n} \frac{1}{i}.$$

An exercise at the end of this chapter asks you to show that $H_n \approx \ln(n)$. Now, we are ready to state the theorem:

Theorem 3.6.1. *Let ALG be a randomized online algorithm for the Paging problem with cache size k. Then we have*

$$\rho_{OBL}(ALG) \geq H_k.$$

Proof. We will first show that there is a distribution on input sequences $x_1, \ldots, x_n$ such that every *deterministic* algorithm achieves average competitive ratio greater than H_k with respect to this distribution. We will then see how this implies the statement of the theorem.

Here is the distribution: pick each x_i uniformly at random from $[k+1]$, independently of all other x_j. For every deterministic algorithm, the expected number of page faults is $k + (n-k)/(k+1) \geq n/(k+1)$ — there are k initial page faults, and there is a $1/(k+1)$ chance of selecting a page not currently in the cache in each step after the initial k steps. Next, we analyze the expected value of OPT. As in Section 1.6, let's subdivide the entire input sequence into blocks $B_1, \ldots, B_T$, where block B_1 is the maximal prefix of $x_1, \ldots, x_n$ that contains k distinct pages, B_2 is obtained in the same manner after removing B_1 from $x_1, \ldots, x_n$, and so on. Arguing as in Section 1.6, we have a bound on $OPT \geq T - 1$. Note that T is a random variable, and $\mathbb{E}(OPT) \geq \mathbb{E}(T) - 1$. Thus, the asymptotic competitive ratio is bounded below by $\lim_{n\to\infty} \frac{n}{(k+1)\mathbb{E}(T)}$. If $|B_i|$ were i.i.d., then we could immediately conclude that $\mathbb{E}(T) = n/\mathbb{E}(|B_1|)$. Unfortunately, the $|B_i|$ are not i.i.d., since they have to satisfy $|B_1| + \cdots + |B_T| = n$. For increasing n, $|B_i|$ start behaving more and more as i.i.d. random variables. Formally, this is captured by the Elementary Renewal Theorem from the theory of renewal processes, which for us implies that the competitive ratio is bounded below by $\lim_{n\to\infty} \frac{n}{(k+1)n/\mathbb{E}(W)} = \mathbb{E}(W)/(k+1)$, where W is distributed as $|B_1|$ in the limit (i.e., $n = \infty$).

Thus, let's consider $n = \infty$ and compute $|B_1|$. Computing $\mathbb{E}(|B_1|)$ is known as *the coupon collector problem*. We can write $|B_1| = Z_1 + \cdots + Z_k + Z_{k+1} - 1$, where Z_i is the number of pages we see before seeing an ith *new* page (see it for the first time). The last term (-1) means that we terminate B_1 one step before seeing $k+1$st new page for the first time. Then $Z_1 = 1$, i.e., any page that arrives first is the new first page. After that, we have the probability $k/(k+1)$ of seeing a page different from the first one in each consecutive step. Therefore, Z_2 is a geometrically distributed random variable with parameter $p_2 = k/(k+1)$, hence $\mathbb{E}(Z_2) = 1/p_2 = (k+1)/k$. Similarly, we get Z_i is geometrically distributed with parameter $p_i = (k-i+2)/(k+1)$, hence $\mathbb{E}(Z_i) = 1/p_i = (k+1)/(k-i+2)$. Thus, we have $\mathbb{E}(|B_1|) = \mathbb{E}(Z_1) + \mathbb{E}(Z_2) + \cdots + \mathbb{E}(Z_k) = 1 + (k+1)/k + \cdots + (k+1)/(k-i+2) + \cdots + (k+1)/2 + ((k+1)/1 - 1) = (k+1)(1/k + \cdots + 1) = (k+1)H_k$. Combining this with above, we get the bound on competitive ratio of any deterministic algorithm: $(k+1)H_k/(k+1) = H_k$.

Let X^n denote the random variable which is the input sequence generated as above of length n. Also note that ALG is a distribution on deterministic algorithms ALG_R. We have proved above that for each deterministic algorithm ALG_r it holds that

$$\mathbb{E}_{X^n}(ALG_r(X^n)) \geq H_k \mathbb{E}_{X^n}(OPT(X^n)) + o(\mathbb{E}_{X^n}(OPT(X^n))).$$

Taking the expectation of the inequality over r, we get

$$\mathbb{E}_R \mathbb{E}_{X^n}(ALG_R(X^n)) \geq H_k \mathbb{E}_{X^n}(OPT(X^n)) + o(\mathbb{E}_{X^n}(OPT(X^n))).$$

Exchanging the order of expectations, it follows that

$$\mathbb{E}_{X^n} \mathbb{E}_R(ALG_R(X^n)) \geq H_k \mathbb{E}_{X^n}(OPT(X^n)) + o(\mathbb{E}_{X^n}(OPT(X^n))).$$

By the definition of expectation, it means that there exists a sequence of inputs x^n such that

$$\mathbb{E}_R(ALG_R(x^n)) \geq H_k OPT(x^n) + o(OPT(x^n)).$$

□

3.7 Yao's Minimax Principle

In the proof of Theorem 3.6.1 we deduced a lower bound on *randomized algorithms against oblivious adversaries* from a lower bound on *deterministic algorithms against a particular distribution of inputs*. This is a general technique that appears in many branches of computer science and is often referred to as Yao's Minimax Principle. In words, it can be stated as follows: the expected cost of a randomized algorithm on a worst-case input is at least as big as the expected cost of the best deterministic algorithm with respect to a random input sampled from a distribution. Let ALG denote an arbitrary randomized algorithm, i.e., a distribution over deterministic algorithms ALG_R. Let μ denote an arbitrary distribution on inputs x. Then we have

$$\max_x \mathbb{E}_R(cost(ALG_R, x)) \geq \min_{\widetilde{ALG}:\text{deterministic}} \mathbb{E}_{X\sim\mu}(cost(\widetilde{ALG}, X)). \tag{3.2}$$

Observe that on the left-hand side ALG is fixed in advance, and x is chosen to result in the largest possible cost of ALG. On the right-hand side the input distribution μ is fixed in advance, and $\widetilde{ALG}$ is chosen as the best deterministic algorithm for μ. Thus, to apply this principle, we fix some distribution μ and show that the expected cost of every deterministic algorithm with respect to μ has to be large, e.g., at least ρ. This immediately implies that any randomized algorithm has to have cost at least ρ. In the above, *cost*() is some measure function. For example, in online algorithms *cost*() is the competitive ratio (strict or asymptotic), in offline algorithms *cost*() is the approximation ratio, in communication complexity *cost*() is the communication cost, and so on. Yao's minimax principle is by far the most popular technique for proving lower bounds on randomized algorithms. The interesting feature of this technique is that it is often *complete*. This means that under mild conditions you are guaranteed that there is a distribution μ that achieves equality in (3.2) for the best randomized algorithm ALG. This way, not only you can establish a lower bound on performance of all randomized algorithms, but you can, in principle, establish the strongest possible such lower bound, i.e., the tight lower bound, provided that you choose the right μ. Originally, the minimax theorem was proved by John von Neumann in the context of zero-sum games, and it was adapted to algorithms by Andrew Yao.

In order to state Yao's minimax principle formally, we need to have a formal model of algorithms. This makes it a bit awkward to state for online algorithms, since there is no single model. We would have to state it separately for request-answer games, for search problems, and for any other "online-like" problems that do not fit either of these categories. What makes it worse is that the general formal statement of the principle for maximization request-answer games is different from that for minimization games. Moreover, completeness of the principle depends on finiteness of the answer set in the request-answer game formulation. Therefore, we prefer to leave Yao's Minimax Principle in the informal way stated above, especially considering that it's application in each particular case is usually straightforward (as in Theorem 3.6.1) and doesn't warrant a stand-alone black-box statement.

3.8 Upper Bound for Paging

In this section we present an algorithm called Mark that achieves the competitive ratio $\leq 2H_k$ against an oblivious adversary for the Paging problem. In light of Theorem 3.6.1, this algorithm is within a factor of 2 of the best possible *online* algorithm.

The pseudocode of Mark appears in Algorithm 8 and it works as follows. The algorithm keeps track of cache contents, and it associated a Boolean flag with each page in the cache. Initially the cache is empty and all cache positions are unmarked. When a new page arrives, if the page is in the cache, i.e., it's a page hit, then the algorithm marks this page and continues to the next request. If the new page is not in the cache, i.e., it's a page miss, then the algorithm picks an unmarked page uniformly at random, evicts it, brings the new page in its place, and sets the status of the new page to *marked*. If it so happens that there are no unmarked pages at the beginning of this process, then the algorithm *unmarks all pages* in the cache prior to processing the new page.

By tracing the execution of the algorithm on several sample input sequences one may see the intuition behind it: pages that are accessed frequently will be often present in the cache in the marked state, and hence will not be evicted, while other pages are evicted uniformly at random from among all unmarked pages. In the absence of any side information about the future sequence of requested pages, all unmarked pages seem to be equally good candidates. Therefore, picking a page to evict from a uniform distribution is a natural choice.

Algorithm 8 Randomized algorithm for Paging against oblivious adversaries

```
procedure Mark
    $C[1...k]$ stores cache contents
    $M[1...k]$ stores a Boolean flag for each page in the cache
    Initialize $C[i] \leftarrow -1$ for all $i \in [k]$ to indicate that cache is empty
    Initialize $M[i] \leftarrow False$ for all $i \in [k]$
    $j \leftarrow 1$
    while $j \leq n$ do
        if $x_j$ is in $C$ then                                  ▷ page hit!
            Compute $i$ such that $C[i] = x_j$
            if $M[i] = False$ then
                $M[i] \leftarrow True$
        else                                                     ▷ page miss!
            if $M[i] = True$ for all $i$ then
                $M[i] \leftarrow False$ for all $i$
            $S \leftarrow \{i \mid M[i] = False\}$
            $i \leftarrow$ uniformly random element of $S$
            Evict $C[i]$ from the cache
            $C[i] \leftarrow x_j$
            $M[i] \leftarrow True$
        $j \leftarrow j + 1$
```

Theorem 3.8.1.

$$\rho_{OBL}(Mark) \leq 2H_k.$$

Proof. Let $x_1, \ldots, x_n$ be the input sequence chosen by an oblivious adversary. As in Section 1.6, subdivide the entire input sequence into blocks $B_1, \ldots, B_t$, where block B_1 is the maximal prefix of $x_1, \ldots, x_n$ that contains k distinct pages, B_2 is obtained in the same manner after removing B_1 from $x_1, \ldots, x_n$, and so on.

Pages appearing in block B_{i+1} can be split into two groups: (1) new pages that have not appeared in the previous block B_i, and (2) those pages that have appeared in the previous block B_i. Clearly, the case where all pages of type (1) appear *before* all pages of type (2) results in the worst case number of page faults of algorithm Mark. Let m_i be the number of pages of type (1) in block B_i, then block B_i contains $k - m_i$ pages of type (2).

It is easy to see that while processing the first page from each block B_i all existing pages in the cache become unmarked. Every new page of type (1) that is brought in results in a page fault and becomes marked in the cache. A page of type (2) may or may not be present in the cache. If it is not present in the cache, then it is brought in marked; otherwise, it becomes marked. A marked page is never evicted from the cache while processing block B_i.

Consider the first page of type (2) encountered in block B_i. Since all m_i pages of type (1) have already been processed, there are $k - m_i$ unmarked pages of type (2) currently in the cache. Moreover, since the choice of an unmarked page to evict during a page fault is uniform, then the $k - m_i$ unmarked pages of type (2) currently in the cache are equally likely to be any of the original k jobs of type (2) in the cache. Thus, the probability that the first page of type (2) is present (unmarked) in the cache is $\binom{k-1}{m_i}/\binom{k}{m_i} = (k - m_i)/k$. Consider the second page of type (2) encountered in block B_i. We can repeat the above analysis by disregarding the first job of type (2) and pretending that cache size is $k - 1$ (since the first job of type (2) has been marked and for during of block B_i it will never be evicted). Therefore, the probability that the second page of type (2) is present (unmraked) in the cache is $(k - m_i - 1)/(k - 1)$. Proceeding inductively, the probability that the jth job of type (2) in block B_i is present (unmarked) in the cache when it is encountered for the first time is $(k - m_i - j + 1)/(k - j + 1)$. Therefore, the expected number of page faults in block B_i is

$$m_i + \sum_{j=1}^{k-m_i} \left(1 - \frac{k - m_i - j + 1}{k - j + 1}\right) = m_i \sum_{j=1}^{k-m_i} \frac{m_i}{k - j + 1} = m_i + m_i(H_k - H_{m_i}) \le m_i H_k.$$

Note that the number of distinct pages while processing B_{i-1} and B_i is $k + m_i$. Therefore, OPT encounters at least m_i page faults. The number of page faults of OPT in B_1 is m_1. Thus, OPT encounters at least $\left(\sum_i m_i\right)/2$ page faults overall, whereas Mark has expected number of page faults at most $H_k \sum_i m_i$. □

3.9 Exercises

1. Prove Theorem 3.3.1.

2. Prove that $\ln(n) \le H_n \le \ln(n) + 1$.

3. The deterministic Algorithm 6 in Chapter 2 for the One-Way Trading problem was obtained from some randomized algorithm (essentially following the steps of Theorem 3.4.3). Find such randomized algorithm and show that after applying the conversion in the proof of Theorem 3.4.3, you get back Algorithm 6.

4. Prove that $\rho_{\text{OBL}}(Mark) > H_k$.

5. (Harder) Prove that $\rho_{\text{OBL}}(Mark) \ge 2H_k - 1$.

6. Prove that if requested pages are restricted to come from $[k+1]$ then Mark is H_k-competitive.

7. Give an example of an online problem where there is a gap between strict competitive ratio and asymptotic competitive ratio. First, do it for deterministic algorithms, then do it for randomized algorithms against oblivious adversaries. Try to achieve the gap that is as large as possible.