Online and Other Myopic Algorithms

Allan Borodin Denis Pankratov

DRAFT: January 13, 2021

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 provide the motivation for the continuing active interest and what we consider to be 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 a (**) indicating that to the best of our knowledge this is not a known result. With the exception of Chapter 1, we will not mention specific references in the main sections of the chapters. Rather, we will defer some history and the most relevant citations for results until the end of the chapter. There we shall also often present some relevant related work that is not mentioned in the main text of the chapter. In each chapter, our convention will be to initially refer to all authors for papers having at most three authors; for papers having more than three authors we will use “author et al.”

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 a few motivating examples. These examples (Ski Rental, Line Search and Paging) are often used to introduce online algorithms and they form the basis for many extensions that more closely model important applications. We analyze online algorithms for these problems using the competitive analysis performance measure. While competitive analysis has been applied extensively in diverse appications, the results can significantly deviate from performance on real data. This is particularly true for 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 or vanilla 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 does not refer to the Internet, although a lot of the applications of online algorithms are in networking and some are specific to 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*. The analysis of online algorithms in terms of the competitve ratio is called *competitve analysis*.

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 the goal is to 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 (i.e., advice), limited mechanisms for 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, 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 any amount of computation to make its decision for each input. 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 rarely happens. 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, the information-theoretic nature of the online model leads to unconditional results, i.e., results that do not depend on unproven complexity assumptions, such as $\mathcal{P} \neq \mathcal{NP}$.

We shall take a tour of various problems, models, and analysis techniques with the goal being 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 will accompany us on this 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 many scheduling and resource allocation applications. Consider, for example, a data center that schedules computing jobs: clearly it is not a good idea to wait for all jobs to arrive in order to come up with an optimal schedule that minimizes the *makespan* – the latest completion time. The jobs have to be scheduled as they come in. Some delay might be tolerated, but not much. As another example, consider patients arriving at a walk-in clinic that need to be seen by a relevant doctor. The receptionist plays the role of an online algorithm, and his or her decisions can be analyzed using the online framework. In online (as in "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 ignoring an item.

One should also note that the term "online algorithm" is used in related but different ways in different fields. For example, in many scheduling results, "online" could arguably be more appropriately called "real time computation" where inputs arrive with respect to continuous time 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 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 an ordering of the input items and then processing the items in an online fashion. Quite often such algorithms give rise to conceptually simple and efficient offline approximation algorithms. This can be helpful not only for achieving non-trivial approximation ratios for $\mathcal{NP}$-hard problems, but also for problems in $\mathcal{P}$ (such as bipartite matching), since optimal algorithms can be too slow for large practical instances. Many 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. In fact, streaming is an online model but we tend to refer to "online algorithms" in a different senses. The setting of streaming algorithms can be viewed figuratively as trying to drink out of a fire hose. There is a massive stream of data passing through a processing unit, and there is no way to store the entire stream for post processing. Streaming algorithms may or may not be required to output results throughout the computation. In any case, streaming algorithms are usually most concerned with minimizing memory requirements in order to compute a certain function of the sequence of input items. Meanwhile 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. For example, Line Search was considered by Anatole Beck [28] and Richard Bellman [31]. One can also 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 [64] 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, Favrholdt and Giel [7] 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 [137] suggested that worst case competitive analysis provided a better than existing "average case" analysis explaination for the success of algorithms such as $MoveToFront$ 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 computer science 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 [92] online greedy algorithm for the identical machines Makespan problem and even more explicitly Yao's [144] analysis of online Bin Packing algorithms. None of these works used the term competitve ratio; this terminology was introduced by Karlin *et al.* [106] 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 1: 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 sounds 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 can be formalized by variants of an online problem called Ski Rental. This problem 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 $\$b$. (Clealry, the currency doesn't matter so we will drop the use of the $ sign.) By scaling we can assume that $r = 1$. For simplicity, we will assume that b is an integer greater or equal to 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 knows the number g of good-weather days in advance. If $g \le b$ 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, which we denote by OPT, is $\min(g, b)$. Even without knowing g it is possible to keep expenses roughly within a factor of 2 of the optimum. This is achieved by the so-called $BreakEven$ algorithm: the idea is to rent skis for $b-1$ days and buy skis on the following day after that. We wish to argue that $BreakEven$ guarantees a cost within the factor $2-1/b$ of optimal, i.e., $BreakEven \le (2-1/b)OPT$. To prove this we consider several cases. If $g < b$ then $BreakEven = g$, i.e., the algorithm incurs the cost g, since the weather would spoil on day $g+1 \le b$ and you would leave before buying skis on day b. Observe that in this case $OPT = g$ as well, and due to our assumption that $b \ge 1$ we have $BreakEven \le (2-1/b)OPT$ in this case. If $g \ge b$ then our strategy incurs cost $BreakEven = b-1+b = (2-1/b)b$. In this case, $OPT = \min(g, b) = b$, so $BreakEven \le (2-1/b)OPT$ holds. Thus, we say that in all cases this strategy achieves competitive ratio $2-1/b$, which is slightly better than 2 and approaches 2 as b increases. Note: scaling allowed us to ignore the rental cost in the competitive ratio. Without scaling, the competitive ratio becomes $2-\frac{r}{b}$. As such, when $r \to 0$ (equivalently, $\frac{r}{b} \to 0$), the competitive ratio becomes 2 in the limit.

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 on the following day. If you buy skis on day i then the adversary declares the weather to be bad on day $i+1$. If $i \le b-1$ then it is optimal to rent skis for a total cost of i, but you incurred the cost of $(i-1)+b \ge (i-1)+(i+1) = 2i$; that is, twice the optimal. If $i \ge b$ then an optimal strategy is to buy skis on the very first day with a cost of b, whereas you spent $i-1+b \ge b-1+b = (2-1/b)b$. Thus, no matter what you do an adversary can force you to spend at least $(2-1/b)$ 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 a stronger adversary that adapts to your decisions. The assumption of a weak adversary for the Ski Rental problem is reasonable because the weather shouldn't conspire against you based on the outcomes of your coin flips. Instead of presenting a randomized Ski Rental algorithm right away and then analyzing it, we shall describe how one could conceivably design such an algorithm from scratch. This will hopefully remove some of the mystery as to where such an algorithm comes from.

What form should a randomized algorithm take? Examining the above deterministic algorithm, it is reasonable to suppose that even with a randomized strategy you should buy skis before or on day b. It is easy to see that buying skis before day b deterministically does not help improve the competitive ratio. What happens if you buy skis before day b with some probability? A simplest form of a randomized algorithm that fits this description picks a random integer $i \in [0, b-1]$ from some probability distribution p, rents for i days, and buys skis on day $i+1$ (if the weather is still good). The distribution p can be represented as a vector $p = (p_0, p_1, \ldots, p_{b-1})$, where p_i is the probability of renting for i days. Note that the p_i must satisfy the following constraints in order for p to be a valid distribution: $p_i \ge 0$ and $\sum_{i=0}^{b-1} p_i = 1$. 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 a decision. To summarize, so far we have a reasonable template for a randomized algorithm for Ski Rental which is shown in Algorithm 1. We say that this is a template because we haven't yet specified how probabilities p_i should be chosen. The rest of this section is dedicated to this.

Algorithm 1 A template of randomized algorithm for Ski Rental.

```
procedure RandomizedSkiRental($b$)
    Compute probabilities $p_i$ for $i \in [0, b-1]$        ▷ Later we discuss how this should be done
    Choose $i \in [0, b-1]$ with probability $p_i$
    while day $j$ is good do
        if $j \le i$ then
            Rent skis on day $j$
        else if $j = i+1$ then
            Buy skis on day $j$
        else
            No decision – skis have already been bought
```

We measure the competitive ratio of 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. In order to see which choices of probabilities p_i lead to the smallest competitive ratio, we leave them as undetermined variables or parameters. Then we express the competitive ratio of Algorithm 1 as a *function* of p. Finding the value of p that minimizes this function is equivalent to specifying how p should be computed in Algorithm 1 to achieve the smallest competitive ratio among all algorithms of this form. This approach of giving a parameterized version of an algorithm, expressing the competitive ratio in terms of the parameters, and then finding an optimal setting of those parameters is quite common in algorithmic analysis.

In order to express the competitive ratio of Algorithm 1 in terms of p, we consider two cases depending on when the adversary decides to spoil the weather.

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, competitive ratio c has to satisfy the following inequality:

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

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 (LP) to minimize c subject to the above inequalities together with the constraints ensuring that p is a valid probability distribution. An interested reader may consult Chapter 8 for a quick refresher on linear programming.

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

Solving the above linear program gives us the values of parameters p that we are after. Instead of solving the LP from scratch, we simply present a solution and verify it. More specifically, 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:

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

where the second equality follows by setting $j = b-1-i$.

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 other than $p_i \geq 0$ with equality. Since an optimal solution occurs at a vertex of the feasible polytope and it is easy to see that setting any $p_i = 0$ is detrimental, the claimed solution p_i and c is optimal and unique. We conclude that our randomized algorithm achieves the competitive ratio $\frac{1}{1-(1-1/b)^b}$. Since $(1-1/b)^b \to e^{-1}$, 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 2: Line Search Problem

A robot starts at the origin of the x-axis[1]. 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 (and also known as Linear Search or Cow Path[2]) problem.

Suppose that the object has been placed at distance $d \geq 1$ 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 correct direction, finding the object in time d. This is an optimal offline solution, i.e., $OPT = d$.

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, without loss of generality say the robot picks the positive direction, and then 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. (Equivalently, a phase ends when the robot doubles the distance.) 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, the 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 the object is at distance d, $2^i < d \leq 2^{i+1}$ in direction $(-1)^i$, the total distance travelled is $2(1+2+\cdots+2^i+2^{i+1})+d \leq 2 \cdot 2^{i+2} + d < 8d + d = 9d$. Thus, this doubling strategy gives a 9-competitive algorithm for the line search problem. We note that a doubling strategy is often utilized in competitive analysis.

[1]In this section we can think of the x-axis as the continuous space $(-\infty, \infty)$ or as an infinite discrete graph with nodes at $\{\ldots, -k, -k+1, \ldots, 0, 1, 2, \ldots\}$.

[2]The *cow path problem* is also used to refer to the more general m-ray problem which entails searching for an object in a star graph starting at the root node. In particular, the line search problem is then the 2-ray 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 an 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 3: 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. Currently, 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 a "cache hit"), the CPU retrieves it directly from the cache. If the requested data is not in the cache (called a "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 a 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 x_{i+1}. 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. As before, we will measure the performance of an algorithm by the competitive ratio; that is, 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 $n-1+k$ cache misses while processing X (where the additive k term assumes that the optimal algorithm also starts with an empty cache). Therefore, the competitive ratio of $FIFO$ is at most $\frac{nk}{n-1+k} = \frac{k+(n-1)k}{k+(n-1)} < k$ and converges 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; that is, when $x_j = x_i$ with $j > i$ and x_i is 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 ALG 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 ALG incur a cache miss on every single time step. An optimal offline algorithm LFD (Longest Forward Distance) 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 are at least $k-1$ pages in future inputs that are going to be requested before the cache page that is furthest in the future. 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 ALG incurs essentially $|X|$ cache misses. Thus, ALG has competitive ratio at least k.

We finish this section by noting that while competitive ratio gives some 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 the 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 better* than $FIFO$, 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 the 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 initiated the area of online algorithm analysis. However, being a worst case measure, it may not model reality well in many applications. This has led researchers to consider other models, such as stochastic inputs, advice, lookahead, and parameterized complexity, among others. We shall cover these topics in later chapters of this book.

1.7 Exercises

1. Fill in the details of the analysis of the randomized algorithm for the Ski Rental problem.

2. Consider the following randomized algorithm for the Ski Rental problem: with a chance of

$\frac{b-1}{b}$, buy the skis on the first day, and with a chance of $\frac{1}{b}$ always rent and never buy. Recall that the cost of buying is $b \in \mathbb{N}$ and the cost of renting is 1 per day. Suppose that the weather spoils on day k.

(a) What is the expected cost of the algorithm in terms of b and k?

(b) Does the algorithm achieve a constant competitive ratio?

(c) Now consider a different randomized algorithm for the Ski Rental problem. Rent equipment for the first $b-1$ days. For each following day, buy equipment with a chance of $1/3$ and continue renting otherwise. Of course, if you decide to buy on a particular day you don't make a decision the following days. What is the competitive ratio of this algorithm?

3. Consider the following version of the Ski Rental problem. Let b be a positive integer divisible by 6. Each day you have the following options:

   (i) rent skis at cost 1 per day; or

   (ii) pay an upfront fee of $b/3$ and the cost of rent per day becomes $r = 1/3$; or

   (iii) buy skis at cost b.

   Consider the following algorithm: use option (i) for $b/2$ days, then use option (ii) and rent for $b/2 - 1$ additional days, then use option (iii) to buy skis on day b. Of course, the algorithm terminates as soon as the weather spoils. Your goal is to find a tight bound on the competitive ratio, meaning you need to prove matching upper and lower bounds.

   (a) State and prove the tightest upper bound on the competitive ratio of this algorithm.

   (b) State and prove the tightest lower bound on the competitive ratio of this algorithm.

4. (*) 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 the uniform distribution on $[n] = \{1, 2, \ldots, n\}$. Design an optimal deterministic online algorithm assuming that the algorithm knows n.

5. 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.

6. Consider the Line Search problem. A slightly more general zig-zag algorithm is presented in Algorithm 2. In $ZigZag(h)$ the initial line segment travelled is h. Thus, the algorithm given in the main text is $ZigZag(1)$ in this terminology.

   (a) Show that $ZigZag(h)$ does not guarantee a finite competitive ratio when OPT can be arbitrarily small.

   (b) Compute the competitive ratio of $ZigZag(h)$ (this includes both upper and lower bounds) under the assumption that $OPT \geq h$. Does this competitive ratio depend on h?

7. Consider the following algorithm for the Line Search problem. The robot starts at the origin, and works in phases $i = 0, 1, 2, \ldots$ In phase i the robot moves 3^i distance to the right. If the treasure is not found, the robot returns to the origin and moves 3^i distance to the left. If the

Algorithm 2 The generalized zig-zag algorithm.

```
procedure $ZigZag(h)$
    $dir \leftarrow +1$
    $dist \leftarrow h$
    while treasure not found do
        Travel in direction $dir$ distance $dist$
        Return to the origin
        $dir \leftarrow dir \times (-1)$
        $dist \leftarrow dist \times 2$
```

treasure is not found, the robot returns to the origin and starts the next phase. Equivalently, a new phase begins when the distance is tripled.

(a) What is the worst case total distance travelled by the robot when the treasure is located at coordinate $+28$?

(b) What is the competitive ratio achieved by the algorithm?

(c) Consider the generalization of this algorithm. Instead of multiplying the distance travelled in each phase by 3, the distance travelled in each phase is now multiplied by k. That is, in phase i, the distance is k^i. Use ia similar analysis to part (b) to compute the competitive ratio in this case.

8. 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 ℓ^∞ norm, i.e., $||(x, y)||_\infty = \max(|x|, |y|)$. 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?

9. In the standard Line Search problem a robot moves along a line to find a treasure. Consider the following modification of this problem. The robot starts out at the crossroads connecting 4 roads. The treasure is located at an unknown distance d along one of the roads. The robot is restricted to moving along the lines only, i.e., it cannot cut across. The standard Euclidean distance is assumed. See the figure below.

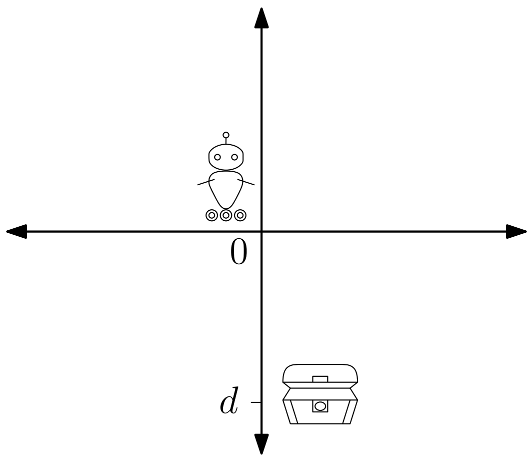
   Consider the following algorithm for finding the treasure. The algorithm works in phases labelled by $i = 0, 1, 2, \ldots$ In phase i the robot walks along each of the 4 roads in some predetermined order to distance 2^i and returns back to the crossroads if the treasure is not found. In this question, your goal is to compute the competitive ratio of this algorithm.

   (a) What is the worst case total distance travelled by the robot when $d = 5$?

   (b) What is the worst case total distance travelled by the robot when $d = 0.2$?

   (c) Assume that $d = 2^k + \epsilon$ for some $0 < \epsilon < 2^k$ and $k \in \mathbb{N}$. What is the worst case total distance travelled by the robot in this case? What is the competitive ratio achieved by the algorithm?

   (d) Instead of 4 roads meeting at the crossroads, now there are m roads meeting at the crossroads. Now, in phase i the robot walks along each of the m roads in some predetermined order to distance 2^i and returns back to the crossroads if the treasure is not



found. Use similar analysis to part (b) to compute the competitive ratio in this case. Your competitive ratio should be expressed as a function of m.

10. Consider the following Search For Treasure problem. There are two robots located at the point O in a Euclidean 2D plane. A circle of radius 1 is centered at O. The robots move at unit speeds and can instantaneously change direction. The robots can only move inside the circle and along its perimeter. The robots do not communicate with each other. A treasure is located somewhere on the perimeter. The robots don't know the location of the treasure and they only find out whether there is a treasure or not at a particular point by visiting that point (i.e., their "visibility radius" is 0). The goal is to specify a trajectory for each robot so that one of the robots (we don't care which one) finds the treasure as fast as possible. We are interested in the worst case discovery time, where worst case is taken over possible locations of the treasure.

    Observe that the offline solution has cost 1, since if robots knew the location X of the treasure, one of the robots could just move directly along the line segment OX.

    (a) Positive result: Describe some optimal trajectories of robots for this task. What is the competitive ratio achieved by these trajectories?

    (b) Negative result: Prove that it is not possible to improve the worst-case discovery time of your trajectories.

11. Consider FWF, which stands for the Flush When Full, algorithm for Paging; that is, when a cache miss occurs and the entire cache is full, evict *all* pages from the cache. What is the competitive ratio of FWF?

12. (a) Find a request sequence X such that $FIFO$ incurs fewer cache misses than LRU on X.

    (b) Find a request sequence X such that LRU incurs fewer cache misses than $FIFO$ on X.

    For the above, you can fix some particular $k \in \mathbb{N}$ – the size of the cache; and some particular $n \in \mathbb{N}$ – the number of distinct pages in the slow memory.

13. 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?

14. Consider the Paging problem where each page $p \in [n]$ has an associated cost $c_p \in \mathbb{R}_{\geq 0}$ for bringing that page into the cache. Instead of minimizing the number of cache misses, now you are interested in minimizing the cost of bringing pages into the cache. Consider the greedy algorithm: if a cache miss occurs and cache is full, evict the page with the smallest associated cost (it will be cheapest to recover).

    (a) State the problem formally using the notation **Input:** ..., **Output:** ..., **Objective:** ....

    (b) Write down the pseudocode of the greedy algorithm described above.

    (c) Prove that the greedy algorithm does not achieve a constant competitive ratio.

15. Consider the Paging problem and the *LRU* algorithm. Does there exist an input sequence I such that *LRU* with cache size $k+1$ on I incurs strictly more page faults than *LRU* with cache size k on I? Note that both runs initially have empty caches.

    If you answer "YES", then give such an instance I (the shorter the better, and you can consider a particular value of k, e.g., $k = 3$ vs. $k + 1 = 4$). If you answer "NO", then give a formal and complete proof of this fact for general k and arbitrary I.

16. Answer the same question as the previous one, but for $FIFO$.

17. Consider the following Paging algorithm, which we call $\widehat{LRU}$: on a page fault, $\widehat{LRU}$ evicts the page whose *second to last request is the least recent.* If there are pages in cache which have been requested fewer than two times, then the least recently used page among those only requested once is evicted. For example, with cache size 3 and request sequence $\langle p_1, p_2, p_3, p_3, p_2, p_1, p_4 \rangle$, on the last request, $\widehat{LRU}$ evicts page p_1, whereas *LRU* would have evicted p_3. Note that most deterministic paging algorithms seem to have competitive ratio k, where k is the size of the cache, but $\widehat{LRU}$ has competitive ratio $2k$. You will prove that here.

    (a) Prove a lower bound of $2k$ on the competitive ratio of $\widehat{LRU}$. Note that you cannot fix k to a particular value, you have to prove the result for all values k simultaneously. Thus, your adversarial analysis has to be in terms of variable k.

    (b) Prove an upper bound of $2k$ on the competitive ratio of $\widehat{LRU}$. Hint: adapt the proof of k-competitiveness for *LRU* to the new algorithm.

1.8 Historical Notes and References

Karp [109] attributes formulating the Ski Rental problem to Rudolf. The problem arose as a special case in Karlin *et al.* [106] in their study of "snoopy caching". They proved the optimal deterministic competitive ratio for the problem. They also introduced the competitive ratio and competitive analysis terminology. The randomized $\frac{e}{e-1}$ competitive ratio is due to Karlin *et al.* [105] and has been generalized in a number of papers as to be discussed in Chapter 4.

The Line Search problem (also known as linear search or cow path problem) has an interesting history preceding competitive analyis. The problem was independently introduced by Beck [28] and Richard Bellman [31]. Beck and Newman [29] proved that the doubling strategy discussed in Section 1.5 is 9-competitive and that this is the best possible deterministic ratio. This problem and its generalization to the m-ray problem has been rediscovered in various papers including Baetza *et al.* [15] which brought attention to this problem in the context of competitive analysis. The extension to the m-ray problem (i.e., searching in a star graph with m rays starting at the root) is due to Gal [83, 84] who proved that the optimal deterministic ratio is $1 + 2\frac{m^m}{(m-1)^{m-1}}$. See Jaillet *et al.* [98] for an historical note on the m-ray problem.

Sleator and Tarjan [137] established the competitive ratio of $FIFO$ and LRU. The optimal offline paging algorithm LFD is due to Belady [30].

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 the Time-Series Search and One-Way Trading problems.

2.1 Request-Answer Games

In Chapter 1, we gave an informal description of an online problem that is applicable to most online problems in the competitive analysis literature including the Ski Rental and Paging problems. Keeping these problems in mind, we can define the general request-answer framework that formalizes the class of online problems. This framework 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 $n = 0, 1, \ldots$.

In the range of a cost function, ∞ is used to indicate that certain solutions (answers) are not allowed. Such solutions are also sometimes referred to as *infeasible*. 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/infeasible. Algorithm 3 provides a template for deterministic online algorithms for any problem within the request-answer game framework.

Algorithm 3 Deterministic online algorithm template.

```
On an instance $\mathcal{I} = (x_1, \ldots, x_n)$, including the ordering of the data items:
$i \leftarrow 1$
while there are unprocessed data items do
    The algorithm receives $x_i \in R$
    The algorithms makes an irrevocable decision $d_i \in A$ for $x_i$ based on $x_1, \ldots, x_i$
    $i \leftarrow i + 1$
```

[1]For certain results concerning request-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.

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.

It is important to note that the same computational problem can have several representations as a request-answer game depending on the information in the request set and answer set. Often there will be natural request and answer sets for a given problem.

2.2 Competitive Ratio for Minimization Problems

In this section we formally define the notion of *competitive ratio*. Let ALG be an online algorithm and $\mathcal{I} = (x_1, x_2, \ldots x_n)$ be an input sequence. We shall abuse the notation and let $ALG(\mathcal{I})$ denote both the output of the algorithm as well as the objective value[2] 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 *(asymptotic) 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 at most ρ if $ALG(\mathcal{I}) \leq \rho \cdot OPT(\mathcal{I}) + o(OPT(\mathcal{I}))$. 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.

This convention is very important so it is worth stressing it again. By default the term competitive ratio means asymptotic competitive ratio. When we wish to use the notion of strict competitive ratio we shall state so explicitly. Also, when we wish to stress the asymptotic nature of the argument we can occasionally say asymptotic competitive ratio explicitly. To distinguish between strict and asymptotic competitive ratios we shall use notations ρ_{strict} and ρ, respectively. Observe that for any algorithm ALG for a minimization problem we have $\rho(ALG) \leq \rho_{\text{strict}}(ALG)$. Therefore, if we prove that $\rho_{\text{strict}}(ALG) \leq k$, it automatically implies that $\rho(ALG) \leq k$. Thus, when it comes to *upper bounds* on competitive ratio for minimization problems, a strict competitive ratio result is stronger than an asymptotic ratio. Also if we prove that $\rho(ALG) \geq k$, it automatically implies that $\rho_{\text{strict}}(ALG) \geq k$, so when it comes to *lower bound* results the relationship is reversed: an asymptotic lower bound is stronger than a strict lower bound. The gold standard, which is not always achievable, happens when one shows an upper bound on the strict competitive ratio, i.e., $\rho_{\text{strict}}(ALG) \leq k$, and a *matching* lower bound on the asymptotic competitive ratio, i.e., $\rho(ALG) \geq k$. In particular, this implies that $\rho_{\text{strict}}(ALG) = \rho(ALG) = k$.

Intuitively, the asymptotic competitive ratio provides a guarantee on the performance of an algorithm only for sufficiently large inputs while the strict competitive ratio provides the guarantee on all inputs. Depending on the problem and how the problem is formalized in the request-answer

[2]Unless otherwise stated, for the problems we shall consider, we will assume that the objective value is a positive rational number. If we assume the ability to do real arithmetic, then we could also entertain real numbers. Our emphasis is primarily on the competitive ratio and not on the algorithmic complexity of our algorithms. Although the algorithms in this text are efficient we again emphasize that competitive analysis does not impose any restrictions on algorithmic complexity or even computability.

framework, the two competitive ratios might coincide for trivial reasons. As one example of this phenomenon, consider the following *direct sum property.* For an input $\mathcal{I}$ and an integer $k \geq 1$, denote by $\mathcal{I}^k$ the input consisting of k independent copies of $\mathcal{I}$ presented one after another. The direct sum property holds for a given problem and an online algorithm ALG when for all inputs $\mathcal{I}$ it holds that $OPT(\mathcal{I}^k) = k \cdot OPT(\mathcal{I})$ and $ALG(\mathcal{I}^k) = k \cdot ALG(\mathcal{I})$. Then we can argue that $\rho_{\text{strict}}(ALG) = \rho(ALG)$ as follows. For an arbitrary $\epsilon > 0$ take an input instance $\mathcal{I}$ such that $ALG(\mathcal{I}) \geq (\rho_{\text{strict}}(ALG) - \epsilon)OPT(\mathcal{I})$. Existence of such $\mathcal{I}$ is guaranteed by the definition of ρ_{strict}. Since $OPT(\mathcal{I}^k) \to \infty$ as $k \to \infty$ we can consider the infinite sequence of inputs $\{\mathcal{I}^k\}_{k=1}^{\infty}$. We have that

$$ALG(\mathcal{I}^k) = k \cdot ALG(\mathcal{I}) \geq k(\rho_{\text{strict}}(ALG) - \epsilon)OPT(\mathcal{I}) = (\rho_{\text{strict}}(ALG) - \epsilon)OPT(\mathcal{I}^k).$$

This shows that $\rho(ALG) \geq \rho_{\text{strict}}(ALG) - \epsilon$. Since $\epsilon > 0$ was arbitrary, the inequality $\rho(ALG) \geq \rho_{\text{strict}}(ALG)$ follows, while the other inequality $\rho(ALG) \leq \rho_{\text{strict}}(ALG)$ always holds as discussed above. Although the direct sum property seems rather strong, there are natural settings in which it holds for a wide range of problems and algorithms. For example, many graph problems reduce to analyzing individual connected components because there is a direct sum property with respect to connected components (as a concrete example consider the problem of computing a minimum spanning forest). Other examples of the phenomenon when strict and asymptotic competitive ratios coincide for trivial reasons include padding the input and scaling input weights (for weighted problems). Such trivial reasons are undesirable because they don't capture the true difference between the behavior in the limit versus the behavior on small inputs. One can often redefine the problem to rule out these trivial reasons and focus on the true asymptotic nature of large inputs for the problem. In our example of graph problems, one could insist on the input graph being connected. Then one cannot scale the performance of an algorithm on graph G simply by presenting independent copies of G, as those would result in different connected components. In the following section we shall see another example of this phenomenon and discuss it in more concrete terms.

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* of the competitive ratio. 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 since $OPT(\mathcal{I}) \to \infty$ implies that $\alpha = o(OPT(\mathcal{I}))$ for any constant α. 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 $ALG(\mathcal{I}) \leq \rho \cdot OPT(\mathcal{I}) + o(OPT(\mathcal{I})) = (\rho + \epsilon) \cdot OPT(\mathcal{I}) + o(OPT(\mathcal{I})) - \epsilon \cdot OPT(\mathcal{I}) \leq (\rho + \epsilon) \cdot OPT(\mathcal{I}) + \alpha$ for a suitably chosen *constant* α such that α dominates the term $o(OPT(\mathcal{I})) - \epsilon \cdot OPT(\mathcal{I})$ for all $\mathcal{I}$ (not just $\mathcal{I}$ with sufficiently large $OPT(\mathcal{I})$). We prefer our definition over the classical one because it can sometimes simplify the presentation of results. 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 an infinite family of instances if we are trying to establish asymptotic inapproximation results). Sometimes it will be convenient to view this game in an 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 inapproximation 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

In this section we consider the Makespan problem for identical machines. In this problem, there is a fixed number m of identical machines that need to process a sequence of jobs with jobs arriving one after another. Each job is described by a single number p, which indicates the time it will take a single machine to process this job. Since machines are identical the number p is independent of which machine the job gets scheduled on. An algorithm assigns a job to a machine immediately after the job arrives. In particular, this assignment is done without the knowledge of future jobs. A job assigned to a machine cannot migrate to another machine in the future – the decision is final. Once a job is assigned to a machine it will be processed immediately after all previous jobs assigned to the machine have been processed. The *makespan* of a machine is its current load: the sum of processing times of all jobs assigned to that machine so far. The objective is to minimize the largest makespan of any machine at the end of the input. Formally the problem is defined as follows.

Makespan (Identical Machines)

Input: $(p_1, p_2, \ldots, p_n)$ where $p_j > 0$ is the load or processing time for a job j; m is the number of identical machines.

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

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

Next, we present a natural online greedy algorithm, which is perhaps the earliest example of a competitive analysis argument and precedes the more explicit discussion in the literature of online algorithms and competitive analysis.

Algorithm 4 The online greedy makespan algorithm.

```
procedure GreedyMakespan
    Initialize $s(i) \leftarrow 0$ for $1 \le i \le m$          ▷ $s(i)$ is the current load on machine $i$
    $j \leftarrow 1$
    while $j \le n$ 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 greedy algorithm repeatedly schedules each job on some least loaded machine. The algorithm is online in the sense that the scheduling of the j^{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. In other words, a greedy algorithm receiving the j^{th} input item acts as if this item was the last input item.

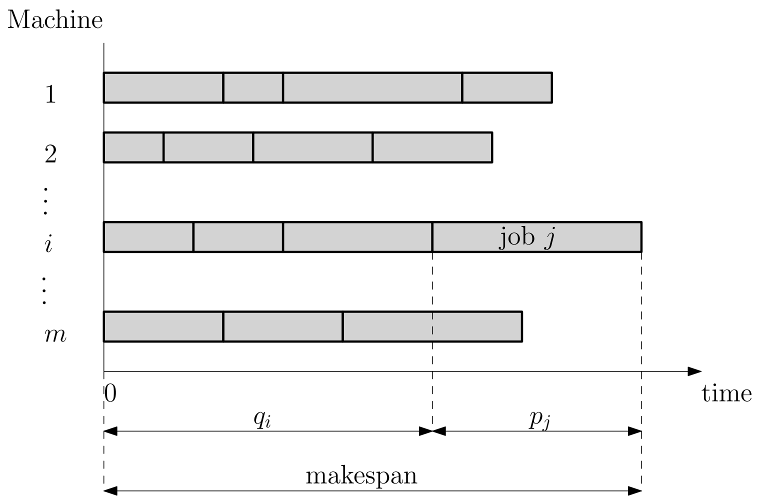


Figure 2.1: This figure illustrates the situation described in the proof of Theorem 2.3.1.

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.

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

Theorem 2.3.1. *For all m and all input sequences $\mathcal{I} = (p_1, p_2, \ldots, p_n)$ we have*

$$GreedyMakespan(\mathcal{I}) \leq \left(2 - \frac{1}{m}\right) \cdot OPT(\mathcal{I}).$$

That is, $\rho_{strict}(GreedyMakespan) \leq (2 - 1/m)$.

Proof. Let $p_{\max}$ denote the maximum processing time of a job in $\mathcal{I}$, i.e., $p_{\max} = \max_{j \in [n]} p_j$. We begin by establishing two necessary lower bounds for any solution and thus for $OPT(\mathcal{I})$:

- $OPT(\mathcal{I}) \geq (\sum_{k=1}^{n} p_k)/m$; that is the maximum load must be at least the average load.
- $OPT(\mathcal{I}) \geq p_{max}$; that is an item of maximum processing time must be scheduled on some machine.

Now we want to upper bound the makespan of *GreedyMakespan* in terms of these necessary OPT bounds. Let machine i be one of the machines with maximum makespan in the schedule produced by *GreedyMakespan* 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 (see Figure 2.1)

$$|GreedyMakspan(\mathcal{I})| = q_i + p_j.$$

Consider what happened at the time job j was scheduled. By the greedy nature, the load of machine i, that is q_i, was minimum at the time p_j was scheduled on machine i. The average of loads of all machines immediately prior to scheduling job j is $\sum_{k<j} p_k/m$, since all past jobs must have been scheduled on m machines. Since the minimum load of a machine can be at most the average load of a machine we have

$$q_i \leq \sum_{k<j} p_k/m \leq \sum_{k \neq j} p_k/m.$$

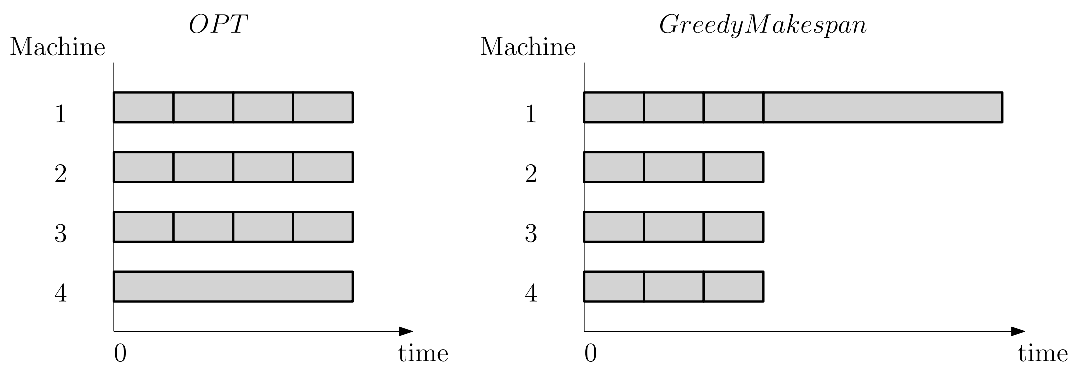


Figure 2.2: This figure illustrates the situation described in the proof of Theorem 2.3.2 for $m = 4$. All the small jobs arrive before the large job. On the left hand side OPT schedule is shown, which is perfectly balanced. Due to the greedy nature of $GreedyMakespan$ all unit jobs are first balanced on the machines, but the last large job still has to be scheduled on some machine, almost doubling the required makespan.

Combining all of the above we obtain

$$\begin{aligned} GreedyMakespan(\mathcal{I}) = q_i + p_j &\le \left(\sum_{k\neq j} \frac{p_k}{m}\right) + p_j = \sum_{k\in[n]} \frac{p_k}{m} + \left(1 - \frac{1}{m}\right) p_j \\ &\le OPT(\mathcal{I}) + \left(1 - \frac{1}{m}\right)\cdot OPT(\mathcal{I}) = \left(2 - \frac{1}{m}\right)\cdot OPT(\mathcal{I}), \end{aligned}$$

where the last inequality follows from the two necessary lower bounds on $OPT(\mathcal{I})$ mentioned at the beginning of the proof. □

The following theorem demonstrates that this *strict competitive ratio* is "tight".

Theorem 2.3.2. *For every m there exists an input sequence $\mathcal{I}$ such that*

$$GreedyMakespan(\mathcal{I}) = \left(2 - \frac{1}{m}\right)\cdot OPT(\mathcal{I}).$$

That is, $\rho_{strict}(GreedyMakespan) \ge (2 - 1/m)$.

Proof. 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, $GreedyMakespan$ 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 $|GreedyMakespan(\mathcal{I})|$ 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. This construction for $m = 4$ is illustrated in Figure 2.2. □

Our formalization of the Makespan problem and the greedy algorithm posses the following scaling property. For an input sequence $\mathcal{I} = (p_1, \ldots, p_n)$ and a number $s > 0$ define the scaled instance $s\cdot\mathcal{I} := (s\cdot p_1, \ldots, s\cdot p_n)$ and observe that

$$OPT(s\cdot\mathcal{I}) = s\cdot OPT(\mathcal{I}) \text{ and } GreedyMakespan(s\cdot\mathcal{I}) = s\cdot GreedyMakespan(\mathcal{I}).$$

This implies that $\rho_{\text{strict}}(GreedyMakespan) = \rho(GreedyMakespan)$, since a nemesis instance $\mathcal{I}$ can be converted into a family of nemesis instances $\{k \cdot \mathcal{I}\}_{k=1}^{\infty}$ with $OPT(k \cdot \mathcal{I}) \to \infty$ as $k \to \infty$. The rest of the argument establishing $\rho_{\text{strict}}(GreedyMakespan) = \rho(GreedyMakespan)$ is similar to the discussion in Section 2.2. In order to rule out such trivial reasons for the equality of the two types of competitive ratios, one could restrict inputs to the Makespan problem to the *normalized* input sequences $(p_1, \dots, p_n)$ with the condition that $\min_j p_j = 1$. Any input sequence $\mathcal{I}$ without trivial 0-duration jobs can be converted into a normalized input sequence $\frac{1}{\min_j p_j} \cdot \mathcal{I}$, therefore this restriction does not have any conceptual impact on the problem. The discussion of the asymptotic considerations below is with respect to these restricted inputs.

We note that for $m = 2$ and $m = 3$, it is not difficult to show that the bound $2 - \frac{1}{m}$ 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 schedules the two initial unit jobs on the same machine, 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 $m \geq 80$ is 1.85358.

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 $p_1 \geq p_2 \geq \cdots \geq p_n$. This is the LPT algorithm ("longest processing time") algorithm which has a tight approximation ratio of $\frac{4}{3} - \frac{1}{3m}$.

2.4 Minimization Problem Example: Bin Packing

In the Bin Packing problem, items arrive in a sequence $(x_1, x_2, \dots, x_n)$ where each item is described by its weight $x_i \in [0, 1]$. The goal is to pack all items into as few bins as possible. Each bin is restricted to hold the maximum total weight 1. This restriction is often referred to as bins being of "unit capacity." We have an unlimited supply of bins. Formally, the problem is stated as follows.

Bin Packing

Input: $(x_1, x_2, \dots, x_n)$ where x_i is the weight of an item i.

Output: $\sigma : \{1, 2, \dots, n\} \to \{1, 2, \dots m\}$ where $\sigma(j) = i$ indicates that item j is packed in bin i and m denotes the maximum number of bins used.

Objective: To find σ so as to minimize m subject to the constraints $\sum_{j:\sigma(j)=i} x_j \leq 1$ for each $i \in [m]$.

The Bin Packing problem is extensively studied within the context of offline and online approximation algorithms. Like the Makespan problem, it is an $\mathcal{NP}$-hard optimization problem. In fact, the hardness of both Makespan and Bin Packing is derived by a reduction from the Subset Sum problem. In 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 5 for pseudocode.
- $FirstFit$: find the first (according to the order in which bins were opened) 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 6 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 7 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 the tightest possible packing. The algorithm *NextFit* is not greedy since it always considers only the most recently opened bin and does not check any of the older bins that could accommodate the current item.

Algorithm 5 The *NextFit* algorithm

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

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*.

Theorem 2.4.1.

$$\rho_{strict}(NextFit) = 2.$$

Proof. First we show $\rho(NextFit) \leq 2$. 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 $B[1] + B[2] > 1$, 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 $i \in \{1, \ldots, m/2\}$. 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 $\sum_{j=1}^{n} x_j$ and that $OPT \geq \sum_{j=1}^{n} x_j$. Combining these observations with the above inequality we get $OPT > m/2 = NextFit/2$. Therefore, we have $NextFit < 2 \cdot OPT$.

Algorithm 6 The *FirstFit* algorithm

```
procedure FirstFit
    $m \leftarrow 0$                                        ▷ total number of opened bins so far
    $R \leftarrow$ a dynamically growing array, initially empty
    ▷ $R$ keeps track of the remaining space in all opened bins
    $j \leftarrow 1$
    while $j \le n$ do
        $flag \leftarrow False$
        for $i = 1$ to $m$ do
            if $x_j \le R[i]$ then
                $R[i] \leftarrow R[i] - x_j$
                $\sigma(j) \leftarrow i$
                $flag \leftarrow True$
                break
        if $flag = False$ then
            $m \leftarrow m + 1$
            Grow the size of $R$ by 1
            $R[m] \leftarrow 1 - x_j$
            $\sigma(j) \leftarrow m$
        $j \leftarrow j + 1$
```

Algorithm 7 The *BestFit* algorithm

```
procedure BestFit
    $m \leftarrow 0$                                        ▷ total number of opened bins so far
    $R \leftarrow 1$                    ▷ array $R$ keeps track of remaining space in all opened bins
    $j \leftarrow 1$
    while $j \le n$ do
        $ind \leftarrow -1$                    ▷ $ind$ will be the index of the bin having the best fit
        for $i = 1$ to $m$ do
            if $x_j \le R[i]$ then
                if $ind = -1$ or $R[i] < R[ind]$ then
                    $ind \leftarrow i$
        if $ind = -1$ then
            $m \leftarrow ind \leftarrow m + 1$
            $R[m] \leftarrow 1$
        $\sigma(j) \leftarrow ind$
        $R[ind] \leftarrow R[ind] - x_j$
        $j \leftarrow j + 1$
```

Next we show that $\rho(NextFit) \geq 2$. Fix an arbitrary small $\epsilon > 0$ such that $n := 1/(2\epsilon) \in \mathbb{N}$. The input consists of n pairs of items $1-\epsilon, 2\epsilon$. Thus, the input looks like $1-\epsilon, 2\epsilon, 1-\epsilon, 2\epsilon, \ldots, 1-\epsilon, 2\epsilon$, where the "$\ldots$" indicate that the corresponding pattern repeats n times. Observe that *NextFit* on this instance uses $2n$ bins, since the repeating pattern of pairs $1-\epsilon, 2\epsilon$ forces the algorithm to use a new bin on each input item ($1-\epsilon+2\epsilon = 1+\epsilon > 1$). The optimal solution places items of weight $1-\epsilon$ in distinct bins and places all items of weight 2ϵ into a single bin. This is possible by the choice of ϵ and n. Thus, we have $OPT = n+1$ whereas $NextFit = 2n$. This means that

$\rho(NextFit) \geq 2n/(n+1) \to 2$ as $n \to \infty$.

□

Before we analyze the competitive ratio of *FirstFit* we introduce some notation. We use S_i to denote the set of items that *FirstFit* packs into bin i for $i \in [m]$. We use Q_i to denote the set of items that *OPT* packs into bin i. The upper bound on the performance of *FirstFit* is established by means of a *weighting technique.* This technique is applicable not only to Bin Packing, but to a variety of other packing problems and can be derived via the primal-dual framework (see Chapter 8) for the Gilmore-Gomory integer programming formulation of Bin Packing. The basic version of the technique is based on the following idea. Let $w : [0,1] \to \mathbb{R}$ be a function that accepts an item size as an input and produces a modified item size as an output. This function naturally extends to accept a (multi) set of items as an input and to output the sum of the modified item sizes in the set. That is for a (multi) set of items S the weight function is defined as $w(S) = \sum_{x \in S} w(x)$. Suppose that the weight function w satisfies the following properties:

1. The function is nonnegative, i.e., $w(y) \geq 0$ for all $y \in [0,1]$.
2. For all, but a constant number, of bins produced by *FirstFit* the weight function has value at least 1, i.e.,
$$w(S_i) \geq 1 \text{ for all but a constant number of } i \in [m].$$
3. For an arbitrary (multi) set of item sizes that could fit in a single bin, the weight function has value at most γ, i.e.,
$$w(S) \leq \gamma \text{ for all } S \text{ such that } S \text{ consists of item sizes in } [0,1] \text{ and } \sum_{y \in S} y \leq 1.$$

Observe that the second property is with respect to (multi) sets of items that correspond to bins of *FirstFit*, while the last property is with respect to arbitrary (multi) sets of arbitrary item sizes (as long as they fit into a single bin) and not necessarily items coming from $(x_1, \ldots, x_n)$.

The key observation is that the existence of such a weight function implies that the asymptotic competitive ratio of *FirstFit* is at most γ. Consider an arbitrary input $(x_1, \ldots, x_n)$ to *FirstFit*. Let k denote the number of bins to which the second property doesn't apply. Then we have

$$\sum_{i=1}^{n} w(x_i) = \sum_{i=1}^{m} w(S_i) \geq m - k = FirstFit(x_1, \ldots, x_n) - k.$$

The first equality above follows from the fact that *FirstFit* packs all items, so $S_1, \ldots, S_m$ form a partition of all items $(x_1, \ldots, x_n)$. The inequality follows from the fact that $w(S_i) \geq 1$ for all but at most k bins and that $w(S_i) \geq 0$ by the first property for the bins to which $w(S_i) \geq 1$ doesn't apply. Next, observe that we can rewrite $\sum_{i=1}^{n} w(x_i)$ in another way, namely, according to the partition of items due to *OPT*:

$$\sum_{i=1}^{n} w(x_i) = \sum_{i=1}^{OPT} w(Q_i) \leq \gamma \cdot OPT(x_1, \ldots, x_n).$$

The inequality follows from the third property of the weight function. Combining the two inequalities, we obtain that $FirstFit(x_1, \ldots, x_n) \leq \gamma \cdot OPT(x_1, \ldots, x_n) + k$. Since k is constant, the upper bound of γ on the asymptotic competitive ratio of *FirstFit* follows.

This technique was used to establish the first tight bound on the competitive ratio of *FirstFit*, except that the second property turns out to be to restrictive and needs to be replaced by a more relaxed version of the property. The proof based on this version of the weighting technique is still quite involved and unintuitive precisely because of the difficulty in handling the second property. Intuitively, the difficulty of the earlier proofs lies in the fact that the above properties are stated for bins independently of each other, whereas the defining property of $FirstFit$ bins involves relationships between items packed into bins S_i and S_j for $i < j$. If one hopes to prove a ratio that holds for $FirstFit$ but doesn't for $NextFit$, the defining property of $FirstFit$ bins has to come up in the proof somewhere and it is difficult to do that (although possible) with the original weighting technique.

A much simpler proof, which we are about to present, is based on a *modified weighting technique*. The idea is to decompose the weight function w into a sum of two components, i.e., $w(x) := \overline{w}(x) + \overline{\overline{w}}(x)$, where the first component $\overline{w}(x)$ is called a scaled size of an item and the second component $\overline{\overline{w}}(x)$ is called a bonus associated with an item. Consider a version of the second property where the scaled size of one bin plus the bonus of a *related later* bin is at least 1. This modified version of the second property immediately refers to the defining property of $FirstFit$ bins, so establishing it requires less argumentation and results in a much easier and shorter proof (this is not to say that the proof becomes "trivial" or "obvious" – far from it, it is simply much easier than original proofs). Of course, this is just a high-level intuitive explanation. Next, we present the formal proof with all the details.

Theorem 2.4.2.
$$\rho(FirstFit) \le 1.7.$$

Proof. Define the weight function $w(x) = \overline{w}(x) + \overline{\overline{w}}(x)$ for $x \in [0, 1]$ as follows:

- The component $\overline{w}(x)$ is referred to as the *scaled size* of an item x and is defined as
$$\overline{w}(x) = \frac{6}{5}x.$$
- The component $\overline{\overline{w}}(x)$ is referred to as the *bonus* of an item x and is defined as
$$\overline{\overline{w}}(x) = \begin{cases} 0 & \text{if } x \le \frac{1}{6}, \\ \frac{3}{5}\left(x - \frac{1}{6}\right) & \text{if } x \in \left(\frac{1}{6}, \frac{1}{3}\right), \\ 0.1 & \text{if } x \in \left[\frac{1}{3}, \frac{1}{2}\right], \\ 0.4 & \text{if } x > \frac{1}{2}. \end{cases}$$

This weight function and its two component functions extend to (multi) sets of items in a natural way as discussed before the statement of this theorem. Next, we establish the three properties sufficient to prove the stated competitive ratio.

Property 1. For every $x \in [0, 1]$ we clearly have $w(x) \ge 0$.

Property 2. Consider bins i and j created by $FirstFit$ such that $i < j$ (i.e., bin i is opened before bin j). If $\sum_{x \in S_i} x \ge 2/3$ (i.e., bin i has total load at least 2/3) and $|S_j| \ge 2$ (i.e., bin j has at least 2 items) then
$$\overline{w}(S_i) + \overline{\overline{w}}(S_j) \ge 1.$$

Proof of property 2. Let y_1 and y_2 be two distinct items in S_j guaranteed to exist since $|S_j| \ge 2$. Since $\sum_{x \in S_i} x \ge \frac{2}{3}$ we have $\overline{w}(S_i) \ge \frac{6}{5} \cdot \frac{2}{3} = 0.8$. If either $y_1 > 1/2$ or $y_2 > 1/2$ then $\overline{\overline{w}}(S_j) \ge 0.4$ and the claim follows immediately. Moreover, if $y_1 \ge \frac{1}{3}$ and $y_2 \ge \frac{1}{3}$ then $\overline{\overline{w}}(S_j) \ge 0.2$ and the claim also follows. Thus, it is only left to consider the *subcases*: (1) $y_1 < \frac{1}{3}$ and $y_2 \in \left[\frac{1}{3}, \frac{1}{2}\right]$; and (2) $y_1 < \frac{1}{3}$ and $y_2 < \frac{1}{3}$.

Examining bin i, observe that if $\sum_{x \in S_i} x \geq \frac{5}{6}$ then $\overline{w}(S_i) \geq 1$, so the claim follows immediately. Thus, suppose that $\sum_{x \in S_i} x < \frac{5}{6}$ and let z denote the slack, i.e., $z > 0$ and $\sum_{x \in S_i} x = \frac{5}{6} - z$. Observe that $z \leq \frac{1}{6}$ since otherwise $\sum_{x \in S_i} x < 2/3$ violating one of the assumptions. Since bin j was opened after bin i, every item placed in bin j has to be of size greater than the empty space in bin i. Then $y_1, y_2 > \frac{1}{6} + z$. Therefore we have

Subcase (1). $\overline{w}(S_i) + \overline{\overline{w}}(S_j) \geq \frac{6}{5}\left(\frac{5}{6} - z\right) + \overline{\overline{w}}(y_1) + \overline{\overline{w}}(y_2) > 1 - \frac{6}{5}z + \frac{3}{2}z + 0.1 = 1.1 - \frac{3}{5}z \geq 1.1 - \frac{3}{5} \cdot \frac{1}{6} = 1.$

Subcase (2). $\overline{w}(S_i) + \overline{\overline{w}}(S_j) \geq \frac{6}{5}\left(\frac{5}{6} - z\right) + \overline{\overline{w}}(y_1) + \overline{\overline{w}}(y_2) > 1 - \frac{6}{5}z + \frac{3}{5}z + \frac{3}{5}z = 1.$

Property 3. Let S be a (multi) set of items that can fit into a single bin, i.e., $\sum_{x \in S} x \leq 1$. Then

$$w(S) \leq 1.7.$$

Proof of property 3. Observe that $\overline{w}(S) \leq \frac{6}{5} = 1.2$. Thus, it suffices to check that $\overline{\overline{w}}(S) \leq 0.5$. We split it into several cases.

Case $\forall x \in S$ we have $x \leq 1/2$. In this case S has at most 5 items with non-zero bonus and each such item can contribute at most 0.1. Therefore $\overline{\overline{w}}(S) \leq 0.5$, as desired.

Case $\exists x \in S$ such that $x > 1/2$. Then $\overline{\overline{w}}(x) = 0.4$. Note that either all remaining items have zero bonus (in which case we are done), or one item can have non-zero remaining bonus and that bonus is at most 0.1 (we are also done in this case), or two items, say, y_1 and y_2, have non-zero remaining bonus. In the latter case, we have $y_1, y_2 \in \left(\frac{1}{6}, \frac{1}{3}\right)$ and $y_1 + y_2 < \frac{1}{2}$, so we have

$$\overline{\overline{w}}(S) = \overline{\overline{w}}(y_1) + \overline{\overline{w}}(y_2) = \frac{3}{5}\left(y_1 - \frac{1}{6}\right) + \frac{3}{5}\left(y_2 - \frac{1}{6}\right) = \frac{3}{5}(y_1 + y_2) - \frac{2}{10} < \frac{3}{10} - \frac{2}{10} = 0.1.$$

This finishes the verification of all three properties. Next, we show how to combine these properties to derive the bound on the competitive ratio of $FirstFit$.

Recall that $S_1, \ldots, S_m$ denote the bins opened by $FirstFit$. Observe that at most one bin can have total load less than $1/2$, otherwise items from a later bin of load $< 1/2$ could have been placed in an earlier bin of load $< 1/2$, which violates the definition of $FirstFit$. Without loss of generality suppose that bin is S_m and ignore this bin from future consideration. Now, split the remaining bins into two categories. The first category consists of those bins that contain a single item each. Let C_1 denote the indices of such bins. The second category consists of those bins that contain at least two items each. Let C_2 denote the indices of such bins. Therefore, we have $C_1 \cup C_2 = \{1, \ldots, m-1\}$ and $C_1 \cap C_2 = \emptyset$. Observe that at most one bin from C_2 has total load $< 2/3$: suppose that two bins $i_1, i_2 \in C_2$ have total loads $< 2/3$ and suppose $i_1 < i_2$ without loss of generality. Since bin i_2 contains at least two items and has total load $< 2/3$ then S_{i_2} contains an item of size $< 1/3$; however, then this item should have been placed in S_{i_1} by the definition of $FirstFit$. This is a contradiction, therefore at most one bin from C_2 has total load $< 2/3$. Let C_2' be C_2 without that bin. Observe that since we have omitted at most two bins in C_1 and C_2' we have $|C_1| + |C_2'| \geq m - 2$. With this notation, we are ready to finish the argument.

Observe that for all $i \in C_1$ load of bin i is greater than $1/2$ and bin i contains a single item. Thus, the bonus of that item is 0.4 whereas the scaled size of that item is at least $\frac{6}{5} \cdot \frac{1}{2} = 0.6$. Therefore, we have $w(S_i) \geq 1$ and consequently:

$$\sum_{i \in C_1} w(S_i) \geq \sum_{i \in C_1} 1 = |C_1|. \tag{2.1}$$

Let $C_2' = \{p_1 < p_2 < \cdots < p_\ell\}$. Since the load of each S_{p_k} is at least $2/3$ and each S_{p_k} contains at least two items we can repeatedly apply property 2 to subsequent pairs of bins, i.e., S_{p_k} playing the role of S_i in property 2 and $S_{p_{k+1}}$ playing the role of S_j in property 2, for $k \in \{1, \ldots, \ell - 1\}$. In other words, we have

$$\sum_{k=1}^{\ell} w(S_{p_k}) = \sum_{k=1}^{\ell} \overline{w}(S_{p_k}) + \overline{\overline{w}}(S_{p_k}) \geq \sum_{k=1}^{\ell-1} \overline{w}(S_{p_k}) + \overline{\overline{w}}(S_{p_{k+1}}) \geq \sum_{k=1}^{\ell-1} 1 = \ell - 1 = |C_2'| - 1. \qquad (2.2)$$

Combining Equations (2.1) and (2.2), we obtain:

$$\sum_{i=1}^{n} w(x_i) = \sum_{i=1}^{m} w(S_i) \geq \sum_{i \in C_1} w(S_i) + \sum_{i \in C_2'} w(S_i) \geq |C_1| + |C_2'| - 1 \geq m - 2 - 1 = FirstFit - 3.$$

Recall that $Q_1, \ldots, Q_{OPT}$ denote the contents of bins created by OPT. Now, by applying property 3 we get

$$\sum_{i=1}^{n} w(x_i) = \sum_{i=1}^{OPT} w(Q_i) \leq \sum_{i=1}^{OPT} 1.7 = 1.7 \cdot OPT.$$

Combining the above two equations we obtain $FirstFit(x_1, \ldots, x_n) \leq 1.7 \cdot OPT(x_1, \ldots, x_n) + 3$. This finishes the proof of the theorem. □

The big mystery in the above proof is where the weight function w comes from and how does one go about finding a weight function for problems similar to Bin Packing. Is there a recipe? Is there a method? Unfortunately, there isn't a completely satisfactory answer to this question. We didn't come up with the above proof and research papers do not mention how one could discover such a weight function. We suspect that it is mostly an educated guess based on analysis of many particular instances and specifically of hard instances on which $FirstFit$ gets competitive ratio close to 1.7. The first step towards finding the appropriate weight function is to realize that the weight function method proves the competitive ratio based on properties 1, 2, and 3. The next step is to "guess" a weight function of some general form, for example, a linear function $w(x) = \alpha \cdot x$ for some constant α. Then one could try to find what value of α proves the best competitive ratio by leaving it as a parameter in the analysis and optimizing the parameter at the end. This is similar to the analysis of $RandomizedSkiRental$ in Chapter 1. Unfortunately, one finds that a linear function is not good enough to establish the competitive ratio 1.7, so a richer class of weight function needs to be considered. The next natural choice seems to be piece-wise linear functions – this is when one can conceivably arrive at the w used in the above proof. The mystery behind the "right" weight function is akin to the mystery behind finding the right potential function in the potential function method to be discussed in Chapter 4. Alternatively, the weight function may be found more or less systematically by using the primal-dual framework which we discuss in Chapter 8, but even then it is hard to find such a nice, succinct and general description of the function.

The same weight function as in the above proof can be used to establish an asymptotic competitive ratio 1.7 for $BestFit$ with some modifications to the way the weight function is applied. Properties 1 and 3 continue to hold since they do not depend on the algorithm being analyzed. In the proof for $FirstFit$, property 2 was applied to a particular sequence of bins, two bins at a time. Unfortunately, this property breaks down when applied to the same sequence of bins for $BestFit$. The problem lies in the fact that $FirstFit$ places item y in bin S_j because y exceeds free space in S_i for $i < j$. This is no longer true for $BestFit$ since y might be placed in bin S_j even when it could also fit in S_i for $i < j$ provided that placing y in S_j resulted in a tighter packing. Thus, property 2 needs to be applied to a carefully chosen sequence of bins, two at a time, that is different from the sequence of bins with at least two elements each in the order of bins being opened. We omit the details and simply state the result. At the end of this chapter, we refer the interested reader to related literature where full details may be found.

Theorem 2.4.3.

$$\rho(BestFit) \leq 1.7.$$

We finish this section by demonstrating that the asymptotic competitive ratio of 1.7 is tight for both $FirstFit$ and $BestFit$. The same adversarial input is used to prove both results simultaneously.

Theorem 2.4.4.

$$\rho(FirstFit) \geq 1.7 \text{ and } \rho(BestFit) \geq 1.7.$$

Proof. Let $k \in \mathbb{N}$, $\delta \in \mathbb{R}$ so that $0 < \delta \ll 18^{-k}$, and define $\delta_i = \delta \cdot 18^{k-i}$. The adversary presents a sequence of $30k$ items arriving in three regions. The first region consists of items whose weights are close to $1/6$ – we call these *type A* items. The second region consists of so-called *type B* items whose weights are close to $1/3$. The last region consists of *type C* items whose weights are close to $1/2$. Each region consists of k blocks and each block consists of 10 items.

Type A items (first region) in block i are defined as follows for $i \in [k]$:

- $a_{1,i} = \frac{1}{6} + 33\delta_i$,
- $a_{2,i} = \frac{1}{6} - 3\delta_i$,
- $a_{3,i} = a_{4,i} = \frac{1}{6} - 7\delta_i$,
- $a_{5,i} = \frac{1}{6} - 13\delta_i$,
- $a_{6,i} = \frac{1}{6} + 9\delta_i$,
- $a_{7,i} = a_{8,i} = a_{9,i} = a_{10,i} = \frac{1}{6} - 2\delta_i$.

Type B items (second region) in block i are defined as follows for $i \in [k]$:

- $b_{1,i} = \frac{1}{3} + 46\delta_i$,
- $b_{2,i} = \frac{1}{3} - 34\delta_i$,
- $b_{3,i} = b_{4,i} = \frac{1}{3} + 6\delta_i$,
- $b_{5,i} = \frac{1}{3} + 12\delta_i$,
- $b_{6,i} = \frac{1}{3} - 10\delta_i$,
- $b_{7,i} = b_{8,i} = b_{9,i} = b_{10,i} = \frac{1}{3} + \delta_i$.

Type C items (third region) in block i are defined as follows for $i \in [k]$:

- $c_{1,i} = c_{2,i} = \cdots = c_{10,i} = \frac{1}{2} + \delta$.

This finishes the description of the adversarial sequence. Specifically, the items are presented in order:

$$\underbrace{a_{1,1}, a_{2,1}, \cdots, a_{10,1}}_{\text{block 1, type A}}, \underbrace{a_{1,2}, a_{2,2} \cdots, a_{10,2}}_{\text{block 2, type A}}, \cdots, \underbrace{a_{1,k}, a_{2,k} \cdots, a_{10,k}}_{\text{block } k\text{, type A}},$$

followed by a similar presentation of type B items, followed by a similar presentation of type C items. Let x denote the entire sequence.

Let ALG denote either $FirstFit$ or $BestFit$. Observe that ALG creates the following bins in processing this input sequence:

- $a_{1,i}, \ldots, a_{5,i}$ are placed in bin $2i-1$ resulting in load $\frac{5}{6} + 3\delta_i$ for $i \in [k]$,
- $a_{6,i}, \ldots, a_{10,i}$ are placed in bin $2i$ resulting in load $\frac{5}{6} + \delta_i$ for $i \in [k]$,
- $b_{1,i}, b_{2,i}$ are placed in bin $2k + 5i - 4$ resulting in load $\frac{2}{3} + 12\delta_i$ for $i \in [k]$,
- $b_{3,i}, b_{4,i}$ are placed in bin $2k + 5i - 3$ resulting in load $\frac{2}{3} + 12\delta_i$ for $i \in [k]$,
- $b_{5,i}, b_{6,i}$ are placed in bin $2k + 5i - 2$ resulting in load $\frac{2}{3} + 2\delta_i$ for $i \in [k]$,
- $b_{7,i}, b_{8,i}$ are placed in bin $2k + 5i - 1$ resulting in load $\frac{2}{3} + 2\delta_i$ for $i \in [k]$,
- $b_{9,i}, b_{10,i}$ are placed in bin $2k + 5i$ resulting in load $\frac{2}{3} + 2\delta_i$ for $i \in [k]$,
- $c_{j,i}$ is placed bin bin $7k + 10(i-1) + j$ resulting in load $1/2 + \delta$ for $i \in [k]$ and $j \in [10]$.

Verification of this claim is omitted, as it is an easy although tedious exercise. The property that $\delta_{i-1} = 18\delta_i$ is crucially important for this claim. Thus, ALG opens $17k$ bins to pack this sequence.

Next, we show that $OPT \leq 10k+1$. We start by filling in $10k-1$ bins with type C items. Then we pad these bins with one of the following combinations:

- $a_{j,i}, b_{j,i}$ for $j \in \{3, 4, \ldots, 10\}$ and $i \in [k]$,
- $a_{1,i}, b_{2,i}$ for $i \in [k]$,
- $a_{2,i}, b_{1,i+1}$ for $i \in [k-1]$.

This leaves $b_{1,1}, a_{2,k}$ and one item of weight $\frac{1}{2} + \delta$, which may be packed in two more bins. Therefore, we have

$$\frac{ALG(x)}{OPT(x)} \geq \frac{17k}{10k+1} \rightarrow 1.7 \text{ as } k \rightarrow \infty.$$

□

2.5 Competitive Ratio for Maximization Problems

As defined, competitive and approximation ratios for a minimization problem always satisfy $\rho \geq 1$ and equal to 1 if and only if 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 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 $\rho \leq 1$. It is becoming more standard to express competitive and approximation ratios as the fraction of the optimum value that the algorithm achieves especially if the ratio is a non-parameterized constant. For some examples, see the results in Chapter 5 for Bipartite Matching and Chapter 6) for MaxSAT. 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 ...". We will use a mixture of these two conventions, stating constant ratios as fractions while, as in this section and many of the results in Chapter 5), we will use ratios $\rho \geq 1$ when ρ is expressed as a function of some input parameter.

As for minimization problems, the term "competitive ratio" will refer by default to the asymptotic competitive ratio that holds for large enough inputs (as measured by OPT) whereas the term "strict competitive ratio", denoted by ρ_{strict}, is reserved for the competitive ratio that holds for all inputs regardless of the size of the input.

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 single day with a 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 rates p_j, that is $L \leq p_j \leq U$ 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; $U, L \in \mathbb{R}_{\geq 0}$ where $L \leq p_j \leq U$ for all $j \in [n]$.
Output: $i \in [n]$.
Objective: To compute i so as to maximize p_i.

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

The following deterministic algorithm is particularly simple and improves upon the trivial ratio ϕ. The algorithm trades all of its savings on the first day that the exchange 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. The value p^* is referred to as the *reservation price*, and the algorithm is known as the *reservation price policy* or *RPP* for short.

Theorem 2.6.1.

$$\rho_{strict}(ReservationPricePolicy) \leq \sqrt{\phi}.$$

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 $p_n \geq L$. The optimum is $\max_j p_j \leq p^* = \sqrt{UL}$. The ratio between the two is

$$\frac{OPT(p_1, \ldots, p_n)}{ReservationPricePolicy(p_1, \ldots, p_n)} \leq \frac{\sqrt{UL}}{L} = \sqrt{\phi}.$$

Algorithm 8 The reservation price policy algorithm.

```
procedure ReservationPricePolicy
    $p^* \leftarrow \sqrt{UL}$
    $flag \leftarrow 0$
    $j \leftarrow 1$
    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 $n$
```

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

$$\frac{OPT(p_1, \ldots, p_n)}{ReservationPricePolicy(p_1, \ldots, p_n)} \le \frac{U}{\sqrt{UL}} = \sqrt{\phi}.$$

□

We can also show that $\sqrt{\phi}$ is optimal among all deterministic algorithms for Time-Series Search.

Theorem 2.6.2. *Let ALG be an arbitrary deterministic algorithm for Time-Series Search. Then*

$$\rho(ALG) \ge \sqrt{\phi}.$$

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

$$\frac{OPT(p_1, \ldots, p_n)}{ALG(p_1, \ldots, p_n)} = \frac{U}{\sqrt{UL}} = \sqrt{\frac{U}{L}} = \sqrt{\phi}.$$

If the algorithm does not trade on day $i \le n-1$, 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

$$\frac{OPT(p_1, \ldots, p_n)}{ALG(p_1, \ldots, p_n)} = \frac{\sqrt{UL}}{L} = \sqrt{\frac{U}{L}} = \sqrt{\phi}.$$

□

Observe that *ReservationPricePolicy* algorithm requires the knowledge of both U and L. What if instead of providing both U and L to an algorithm, we provided only the ratio ϕ to an algorithm? A similar argument to the one used in Theorem 2.6.2 shows that knowing just ϕ is not sufficient to improve upon the trivial competitive ratio of ϕ.

Theorem 2.6.3. *Suppose that instead of U and L only $\phi = \frac{U}{L} \ge 1$ is known to an algorithm a priori. Let ALG be an arbitrary deterministic algorithm for this version of Time-Series Search. Then*

$$\rho(ALG) \ge \phi.$$

Proof. The adversary declares ϕ and presents the input sequence $(p_1, \ldots, p_n)$, where $p_i = 1$ for $i \in [n-1]$.

If ALG trades on 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 $p_n = \phi$, 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 achieving objective value 1, and the algorithm trades on the last day achieving $1/\phi$.

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

2.7 Maximization Problem Example: One-Way Trading

A natural generalization of Time-Series Search is the One-Way Trading problem. In this problem, instead of requiring an algorithm to trade *all of its savings* on a single day, we allow the algorithm to trade a fraction $f_j \in [0, 1]$ of its savings on day j for $j \in [n]$. An additional requirement is that by the end of the last day all of the savings have to be traded, that is $\sum_{j=1}^{n} f_j = 1$. 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 of the remaining savings at the specified rate on day n.

One-Way Trading

Input: $(p_1, p_2, \ldots, p_n)$ where p_j is the rate for day j meaning that one unit of savings is equal to p_j units of new currency; $U, L \in \mathbb{R}_{\geq 0}$ where the rates must satisfy $L \leq p_j \leq U$ for $j \in [n]$.

Output: $f_1, \ldots f_n \in [0, 1]$ where f_j indicates that the fraction f_j of savings are traded on day j and $\sum_{j=1}^{n} f_j = 1$.

Objective: To compute f so as to maximize the aggregate exchange rate $\sum_j f_j p_j$.

The ability to trade a fraction of savings on each day is surprisingly powerful resulting in an exponential improvement in the best achievable competitive ratio as compared to Time-Series Search. More specifically, next we show that one can almost achieve competitive ratio $\log \phi$ instead of $\sqrt{\phi}$ that is best possible when one is forced to trade entire savings on a single day.

Algorithm 9, called $MixtureOfRPP$, shows how this is done. To simplify the presentation we assume that $\phi = 2^k$ for some positive integer k. Consider exponentially spaced classes of reservation prices $\{p_j\}$. More specifically, we are partitioning the prices into k classes $\{p_j | L \cdot 2^i \leq p_j < 2^{i+1}\}$ for $i \in \{0, 1, \ldots, k-2\}$ and $\{p_j | L \cdot 2^{k-1} \leq p_j \leq L \cdot 2^k\}$. We refer to $L \cdot 2^i$ as the i^{th} 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 $i \leq i^*$ 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 a *mixture of reservation price policies*, hence the name of the algorithm[3].

Theorem 2.7.1.

$$\rho(MixtureOfRPPs) \leq c(\phi) \log \phi,$$

where $c(\phi)$ is a function such that $c(\phi) \to 1$ as $\phi \to \infty$.

[3]See Exercise 3 in Chapter 3 for further motivation of the naming of the Algorithm

Algorithm 9 The mixture of RPPs algorithm.

```
procedure $MixtureOfRPPs$
    ▷ $U, L$, and $\phi = U/L = 2^k$ are known in advance
    $i^* \leftarrow -1$
    for $j \leftarrow 1$ to $n$ do
        $i \leftarrow \max\{i \mid L \cdot 2^i \leq p_j\}$
        if $i = k$ then Line    ▷ The price $U = L \cdot 2^k$ is in the class $\{p_j | L \cdot 2^{k-2} \leq p_j \leq L \cdot 2^{k-1}\}$
            $i \leftarrow k - 1$
        if $i > i^*$ then
            Trade fraction $(i - i^*)/k$ of savings on day $j$
            $i^* \leftarrow i$
    Trade all remaining savings on day $n$
```

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 $L \cdot 2^{i_j} \leq p_j$ for all $j \in [n]$. Observe that the definition of i_j states that $p_j < L \cdot 2^{i_j+1}$ for all $j \in [n]$ (with the exception of $p_j = U = L \cdot 2^k$). Let ℓ denote the day with the highest exchange rate, that is $p_\ell = \max_{j \in [n]} p_j$. Clearly, we have

$$OPT(p_1, \ldots, p_n) = p_\ell < L \cdot 2^{i_\ell+1}.$$

Note that the value of i^* changes during the execution of $MixtureOfRPPs$. The observed values form a non-decreasing sequence during the execution of the algorithm. Ignore those days when the value of i^* does not change and consider only the *distinct* values of i^* that are observed. Let $i_0^* < i_1^* < \cdots < i_q^*$ denote that sequence of values. We have $i_0^* = -1$ and $i_q^* = i_\ell$. With this notation we can describe the value of the objective function achieved by $MixtureOfRPPs$:

$$\left(\sum_{j=1}^{q} \frac{i_j^* - i_{j-1}^*}{k} L \cdot 2^{i_j^*}\right) + \frac{k - i_\ell}{k} L,$$

where the first term is the lower bound on the contribution of trades until day ℓ and the second term is the lower bound on 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 the expression

$$\sum_{j=1}^{q} (i_j^* - i_{j-1}^*) 2^{i_j^*} \tag{2.3}$$

over all increasing sequences i_j^* with $i_0^* = -1$ and $i_q^* = i_\ell$ then we can set $i_j^* = j - 1$. That is the unique minimizer of expression (2.3) 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 j^{th} term in expression (2.3) 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 expression (2.3). 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 $MixtureOfRPPs$ as follows:

$$MixtureOfRPPs(p_1, \ldots, p_n) \geq \frac{2^{i_\ell+1} - 1}{k} L + \frac{k - i_\ell}{k} L.$$

Finally, we can bound the competitive ratio:

$$\frac{OPT(p_1,\ldots,p_n)}{MixtureOfRPPs(p_1,\ldots,p_n)} \le \frac{L\cdot 2^{i_\ell+1}}{(2^{i_\ell+1}-1)L/k+(k-i_\ell)L/k} = k\frac{2^{i_\ell+1}}{2^{i_\ell+1}+k-i_\ell-1}.$$

The worst case competitive ratio is obtained by maximizing the above expression. We can do so analytically (taking derivatives, equating to zero, and so on), 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. In contrast, it turns out that for One-Way Trading one can prove a similar positive result to the above assuming that the algorithm only knows ϕ and doesn't know U or L. Exercise 10 is dedicated to this generalization. Another generalization is that we don't need to assume that ϕ is a power of 2, which we state here without a proof.

2.8 Exercises

1. Show that $2-\frac{1}{m}$ is an *asymptotic* lower bound for the $GreedyMakespan$ algorithm. That is, for loads in some range $[1,R_m]$, show that there are arbitrarily long input sequences forcing the stated competitive ratio fofr every m.

2. 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-approximation for this problem.

3. Consider the following algorithm called $WorstFit$: find a bin among *all opened bins* that has *maximum* 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.

   (a) Write down pseudocode for $WorstFit$.

   (b) Analyze the competitive ratio of $WorstFit$ by giving matching lower and upper bounds.

4. (*) Find an algorithm ALG different from $FirstFit$ and $BestFit$ that also achieves competitive ratio 1.7 for the Bin Packing problem.

5. (*) Define a general class of algorithms that includes $FirstFit$, $BestFit$ and ALG from the previous exercise as special cases and such that every algorithm in that class achieves competitive ratio 1.7.

6. We have shown that for every input sequence $(x_1,\ldots,x_n)$ we have $FirstFit(x_1,\ldots,x_n)\le \frac{17}{10}OPT(x_1,\ldots,x_n)+3$ using the modified weighting technique. The proof was a bit involved. One can show a much simpler proof if one aims for a slightly weaker result, namely: $FirstFit(x_1,\ldots,x_n)\le\frac{7}{4}OPT(x_1,\ldots,x_n)+2$. In this case one doesn't even have to use the modified weighting technique and can apply the original weighting technique instead. This is what you are asked to do in this question.

(a) Consider the following weight function w

$$w(x) = \begin{cases} \frac{3}{2}x & \text{if } 0 \leq x \leq 1/2 \\ 1 & \text{if } x > 1/2 \end{cases}$$

Prove that for all $k \in \mathbb{N}$ and for all $y_1, \ldots, y_k \in \mathbb{R}_{\geq 0}$ if $\sum_{i=1}^{k} y_i \leq 1$ then $\sum_{i=1}^{k} w(y_i) \leq 7/4$. This shows that applying the weight function to a bin can never stretch the total weight over 7/4.

(b) Consider an arbitrary input sequence $(x_1, \ldots, x_n)$ and the bins produced by the $FirstFit$ algorithm. Define the following bin types:

Type I. Each bin of this type contains a single item of weight $> 1/2$.

Type II. Each bin of this type contains more than one item and the total weight of the bin is $> 2/3$.

Type III. Any bin that is not of Type I or Type II.

Prove that the weight function applied to each bin of Type I or Type II is ≥ 1.

(c) Prove that there can be at most 2 bins of Type III.

(d) Combine all of the above to show that $FirstFit(x_1, \ldots, x_n) \leq \frac{7}{4}OPT(x_1, \ldots, x_n) + 2$.

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

8. 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?

9. 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?

10. (*) 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.

11. Consider the Makespan problem in the random order input model (ROM): adversary creates an input sequence by specifying the processing times $p_1, \ldots, p_n$, the algorithm receives this instance in some random order. That is, *after* the adversary specifies the input sequence, a permutation $\sigma : [n] \to [n]$ is sampled uniformly at random, and the algorithm receives items in the order $p_{\sigma(1)}, p_{\sigma(2)}, \ldots, p_{\sigma(n)}$. Consider the greedy algorithm for the Makespan problem in ROM. Prove that for every $\epsilon > 0$ there exists a sufficiently large m and an input sequence such that $\mathbb{E}[Greedy]/OPT \geq 2 - \epsilon$. Here, the expectation is with respect to the uniform distribution on input arrival order.

2.9 Historical Notes and References

Request-answer games were introduced in Ben-David *et al.* [32]. As mentioned, request-answer games serve as a very general abstract model that applies to almost all optimization problems that we will be considering in this text. Moreover, as we will see in Chapter 3, within this model the different types of adversaries (with respect to randomized online algorithms) can be compared.

The Makespan problem for identical machines was studied by Graham [92, 93]. These papers present online and offline greedy approximation algorithms for the Makespan problem on identical machines as well as presenting some surprising anomalies. The papers precede Cook's seminal paper [58] introducing $\mathcal{NP}$ completeness but still Graham conjectures that it is not be possible to have an efficient optimal algorithm for this problem. This work also precedes the explicit introduction of competitive analysis by Sleator and Tarjan [137] 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.

Once competitive analysis emerged as a prominent research area, attention turned to some classical optimization problems including the Makespan problem. In particular, it was natural to ask if Graham's greedy algorithm was the best online algorithm for any number m of machines. As mentioned, the greedy algorithm is the optimal online algorithm for $m = 2$ and 3. Galambos and Woenginger [85] provided the first improvement over the classical greedy bound for larger values of m. Namely, they show that for every $m \geq 4$, there is a deterministic online algorithm $RLS(m)$ such that the competitive ratio of $RLS(m)$ is equal to $2 - \frac{1}{m} - \epsilon_m$ for some small constant $\epsilon_m > 0$ that converges to 0. This was the beginning of a series of papers establishing improved upper and lower bounds for the Makepsan problem on m identical machines establishing $2 - \epsilon$ bounds for $\epsilon > 0$ independent of m. The first such result is due to Bartal *et al.* [24]. Currently the best approximation ratio is $1 + \sqrt{\frac{1+\ln 2}{2}} \approx 1.9201$ due to Fleischer and Wahl [82] where the bound improves upon previous results for $m \geq 64$. They also provide an account of the papers leading up to their result. In particular, for all $m \geq 2$, Albers [6] established a 1.923 competitive ratio. The current best lower bound is ≈ 1.85358 due to Gormley *et al.* [90]. Their bound is obtained by a general procedure using linear programming to generate deterministic adversaries for a class of request-answer games. Perviously, an explicit adversarial sequence in [6] was used to prove a 1.852 lower bound. These lower bounds are not asymptotic lower bounds.

The Makespan problem belongs to a wider class of scheduling problems called load balancing problems, including other machine models and routing problems (see Chapter 4), other performance measures (e.g., where the load on a machine is its L_p norm for $p \geq 1$) as well as other performance measures (see Chapter 15). In particular, the so-called "Santa Claus problem" [20] considers the $\max - \min$ performance measure for scheduling jobs in the unrelated machines model. 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.

The Bin Packing problem is a classic $\mathcal{NP}$-hard optimization problem and one that continues to be a subject of interest for both the offline and online settings. Early work on the Bin Packing problem popularized the field of approximation algorithms although Graham's Makespan results preceded the results for Bin Packing. This chapter was restricted to the classical one-dimensional Bin Packing problem. Until recently, the most precise results for $FirstFit$ and $BestFit$ appeared in Johnson *et al.* [101] following an earlier conference version by Garey, Graham and Ullman [87] which itself was preceded by a technical report for $FirstFit$ by Ullman [139]. Namely, these results showed the asymptotic competitive ratio to be 1.7 and upper bounds were established by means of a weighting technique. We preented a modified weighting technique and weighting function analysis for $FirstFit$ introduced by Sgall [135] in order to simplify the asymptotic analysis of $BestFit$. Building on this work after more than 40 years since the asymptotic bound, Dósa and Sgall ([67] and [68]) adapt the modified weighting technique to prove that the bounds for $FirstFit$ and $BestFit$ are *strict* competitive ratios. Specifically, they showed that $FirstFit(x_1, \ldots, x_n), BestFit(x_1, \ldots, x_n) \leq \lfloor 1.7 \cdot OPT(x_1, \ldots, x_n) \rfloor$ for any input sequence $(x_1, \ldots, x_n)$. Johnson [100] asked whether or not $FirstFit$ (or $BestFit$) is the best online algorithm for Bin Packing. In what might be the first explicit study of competitive analysis (before the term was introduced), Yao [145] provided an improved online algorithm called $RefinedFirstFit$ with competitive ratio $\frac{5}{3}$. Furthermore, Yao gave the first negative result for competitive analysis showing that no online Bin Packing algorithm can have a competitive ratio better than $\frac{3}{2}$. Since these early Bin Packing online algorithms, there have been a number of improvements in the competitive ratio based on the $Harmonic$ algorithm of Lee and Lee [118]; in particular, the current best ratio is 1.57829 due to Balogh *et al.* [16]. Currently the best lower bound for online Bin Packing is 1.54278 due to Balogh, Békés and Galambos [17] providing a small improvement over a much earlier 1.54037 lower bound by van Vliet [141]. A comprehensive review of the many papers and ideas leading up to the current best ratio can be found in [16]. The lower bounds for bin packing are asymptotic lower bounds in contrast to the offline analysis of the bin packing problem where Karp and Karmacher [107] provide an algorithm ALG such $ALG \leq OPT + o(OPT)$; That is, the asymptotic approximation ratio $= 1$ whereas NP-hardness implies that the strict approximation ratio is at least $\frac{3}{2}$

Given the applications and historical interest in Bin Packing, it is not surprising that there are many variants of the problem. The most important variant is arguably the two dimensional problem, which itself comes in different versions; for example, can the items be rotated 90 degrees or must they fit according to the given axis parallel orientation. We will be considering variants such as two dimensional bin packing and dynamic bin packing in Chapter 7.

Competitive algorithms for the Time-Series Search and One-Way Trading problems were introduced and analyzed in El-Yaniv *et al.* [72]. It is interesting to note the relation between the Time-Series Search problem and the secretary problem in Chapter 16 . In Time-Series we do not know the number n of inputs in advance and we studied the problem in the adversarial input medel assuming we are given the hyper-parameters L, U. In the secretary problem, we assume n is known but inputs arrive according to what is called the random order input model (ROM) and we are not given any initial hyper-parameters. It follows that the positive and negative results concerning deterministic online algorithms for Time-Series Search apply immediately to the secretary problem if the input sequence is generated by an adversary and the hyper-parameters are known. In the same way, the positive and negative results for the secretary problem would apply to the Time-Series Search problem in the random order model.

- two figures for Theorem 2.4.2: one with the plot of the bonus function and another schematic demonstrating how property 2 is applied to $FirstFit$ bins;

- one figure for Theorem 2.4.4 demonstrating different types of bins created by $FirstFit/BestFit$ on the adversarial sequence;

- one figure for an example of Time-Series Search problem, another figure illustrating Theorem 2.6.2;

- one figure illustrating proof of Theorem 2.7.1;

- add a theorem or an exercise about a lower bound for One-Way Trading;

- add more exercises.

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 the 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 notions of **oblivious**, **adaptive online**, and **adaptive offline** adversaries, and explore relationships between them. We introduce Yao's minimax theorem, which is a basic 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.

We refer the reader to Appendix 25 for a brief review of probability theory.

3.1 Randomized Online Algorithm Template

Our convention will be that 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 deterministic 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, for example Bernoulli with parameter p, 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 can sample from a given distribution.

Remark 3.1.2. Letting R be an *infinite* tape of random bits allows online algorithms to sample irrational numbers. This is in line with the information theoretic nature of online algorithms and is done for convenience only. For example, if we wanted to sample from a uniform distribution from interval $[0, 1]$, we could *exactly* sample from that distribution by simply interpreting the entire R as a binary expansion of a random number between 0 and 1. If we wanted to convert an algorithm using such a sample into a more realistic algorithm, we would have to fix some precision k and *approximate* the uniform distribution by $0.R_1R_2 \dots R_k$, where R_i is the i^{th} bit of the infinite random tape R. Then we would have to argue that this approximation does not significantly affect the rest of the algorithm. This typically follows by taking k to be large enough (and still polynomial in the input size) and proving bounds on the approximation error for all computations. The details are typically tedious and routine. We will avoid this issue by always working with distributions that can be efficiently approximated within any specified finite precision. Thus, all our algorithms using potentially irrational samples could be converted into more realistic algorithms in this sense.

Note that the decision in step 4 is now a function not only of all the previous inputs but also of the 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. We will discuss these adversaries in the context of minimization problems; the analogous concepts for maximization problems can be defined following the corresponding discussion of competitive ratio for deterministic setting in Chapter 2.

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 $x_1, x_2, \dots, x_n$ in advance — before learning any of the decisions made by the online algorithm on this input. Thus, the oblivious adversary knows the *distribution* of $D_1, D_2, \dots, D_n$, but it doesn't know which particular decisions $d_1, d_2, \dots, d_n$ are going to be taken by the algorithm. Let $OBJ(x_1, \dots, x_n, d_1, \dots, d_n)$ 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 $x_1, \dots, x_n$. More formally it is

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

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

Remark 3.2.1. Note that good performance in expectation does not preclude that a randomized algorithm may have terrible performance with some small probability.

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 $x_i := x_i(x_1, d_1, \dots, x_{i-1}, d_{i-1})$. 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 $x_1, \dots, x_n$ in advance. More formally it is

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

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

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 d_1' on this item. 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: $x_1, d_1', d_1, x_2, d_2', d_2, x_3, d_3', d_3, \dots$ 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

$$\frac{\mathbb{E}_{D_1,\dots,D_n}\left(OBJ(x_1,\dots,x_n,D_1,\dots,D_n)\right)}{\mathbb{E}_{D_1,\dots,D_n}\left(OBJ(x_1,\dots,x_n,d_1',\dots,d_n')\right)}.$$

Observe that we have to take the expectation of the objective value achieved by the adversary, since both input items $x_1, \dots, x_n$ and adversary's decisions $d_1', \dots, d_n'$ depend on random decisions of the algorithm $D_1, \dots, D_n$ (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 reflects settings where the actions of an algorithm have no impact on future input arrivals. We already assumed an oblivious adversary 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 a 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 the 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 I_z 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 $f_t(x_1, \dots, x_t, d_1, \dots, d_t) \le \rho_{\text{ADOFF}} OPT(x_1, \dots, x_t)$, where d_i are the *deterministic* choices provided by ALG. $\square$

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*

$$\rho_{ADOFF}(ALG) \le (\rho_{ADON}(ALG))^2 .$$

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 a copy of R. We represent ALG as a distribution on deterministic algorithms ALG_r where $r \sim R$. Let $x(R)$ be the requests given by ADV when it runs against ALG_R. Let $d(R)$ denote ithe decisions by ALG_R when it runs against ADV.

Consider a fixed value 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 $x(r)$:

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

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

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

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:

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

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

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

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

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

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

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. The relationships between competitive ratios with respect to different adversaries are summarized in Figure 3.1.

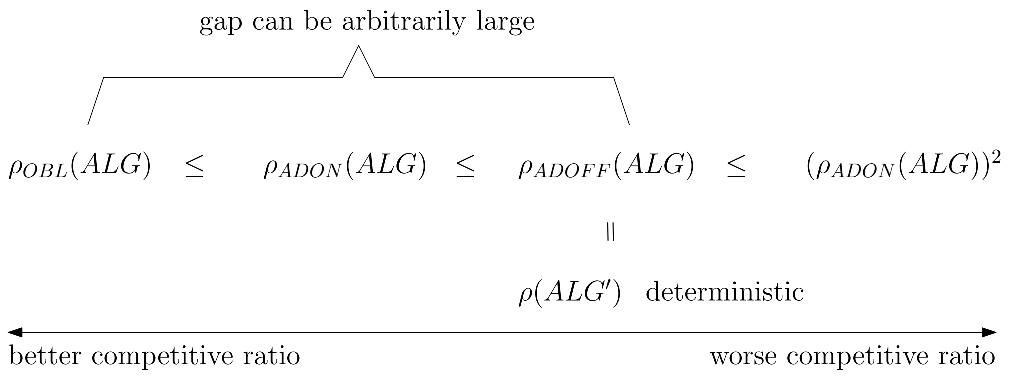

gap can be arbitrarily large
$\rho_{OBL}(ALG) \leq \rho_{ADON}(ALG) \leq \rho_{ADOFF}(ALG) \leq (\rho_{ADON}(ALG))^2$
$\|$
$\rho(ALG')$ deterministic
better competitive ratio
worse competitive ratio

Figure 3.1: The figure summarizes the relationships between the competitive ratios with respect to the different types of adversaries.

3.4 How Much Can Randomness Help?

We exhibit a somewhat contrived algorithmic problem which shows that the gap between the 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 possibility? 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*

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

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 $p_1, \dots, p_n$. Let $f_i(r, p_1, \dots, p_i)$ 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

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

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

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

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

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

□

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

In the proportional knapsack optimization problem[1] the goal is to pack a maximum total weight of items into a single knapsack of weight capacity W (known in advance). Item i is described by

[1]The problem is often referred to as the subset-sum problem. Subset-sum also refers to the NP-complete decision problem; namely, given a set of weights and a target W, is there a subset of weights whose sum equals the target W. The subset-sum decision problem is often used to show that other scheduling problems are NP-hard. Clearly, an optimal algorithm for the proportional knapsack problem provides a solution for the subset-sum decision problem.

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 any constant competitive ratio.

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 does not pack any of the first $n-1$ items, then the adversary declares that the n^{th} item has value 0 which again results in an infinitely large competitive ratio. Otherwise, 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 10 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 10 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 $w_1, \dots, w_n$ 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.

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

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

Case 1: for all $i \in [n]$ we have $w_i < W/2$. Subcase 1(a): $\sum_{i=1}^n w_i \le W$. In this subcase, SimpleRandom running in the first mode packs all of the items. This happens with probability $1/2$, thus we have $\mathbb{E}(\text{SimpleRandom}) \ge 1/2 \sum_i w_i$ and $OPT = \sum_i w_i$. Therefore, it follows that $OPT \le 2 \cdot \mathbb{E}(\text{SimpleRandom})$ 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 $\mathbb{E}(\text{SimpleRandom}) \ge (1/2)(W/2) = W/4 \ge OPT/4$, where the last inequality follows from the trivial observation that $OPT \le W$. Rearranging we have $OPT \le 4 \cdot \mathbb{E}(\text{SimpleRandom})$ in this subcase.

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

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 the one hand, when we are faced with a difficult problem for which no good deterministic algorithm exists, we hope that adding randomness would allow one to design a much better (and often simpler) randomized algorithm. On the other hand, when we have a good randomized algorithm, we hope that we can remove its dependence on randomness, since randomness as a resource is quite expensive. The question then is whether or not a given algorithm can be "de-randomized". It is hard to define a precies definition for what we allow in terms of transforming a randomized algorithm into a deterministic algorithm. We will see some examples of de-randomizations later in the text; for example, see Section 6.1.1 and the method of conditional expectations. However, in what sense can we say that an online algorithm *cannot* be de-randomized? If we show that no deterministic online algorithm exists with (nearly) the same guarantees (e.g., competitive ratio) as the randomized algorithm ALG, we can then say that the ALG cannot be "derandomized". Whether all algorithms can be derandomized or not depends on the computational model. For example, we have seen that, in general, derandomization is not possible for online algorithms with respect to an oblivious adversary, but it is possible for online algorithms with respect to an adaptive offline adversary. In the Turing machine world, it is a fundamental open problem whether all of bounded-error polynomial time algorithms (i.e., the class BPP) can be derandomized or not. Some complexity theorists believe that such a derandomization of BPP should be possible. When derandomization is not possible in general, 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 an important topic and it will come up several times in this book.

In the remainder of this subsection, we would like to briefly discuss why randomness as a resource can be expensive. Prominent scientists throughout 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 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 major 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, it is possible that the only side-effect is that your program might experience a slight degradation in performance, which is not critical. However, there are cases where violating the truly random assumption 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. Without loss of generality we will assume that there are at least $k+1$ pages in slow memory. 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 will result in average competitive ratio at least 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, we can observe that the expected number of page faults is $k + (n-k)/(k+1) \geq n/(k+1)$; that is, 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 $B_1, \dots, B_T$, where block B_1 is the maximal prefix of $x_1, \dots, x_n$ that contains k distinct pages, B_2 is obtained in the same manner after removing B_1 from $x_1, \dots, x_n$, and so on. Arguing as in Section 1.6, we have a bound on $OPT \leq T - 1$. Note that T is a random variable, and $\mathbb{E}(OPT) \leq \mathbb{E}(T) - 1$. 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 (i.e., seeing it for the first time). The last term (-1) means that we terminate B_1 one step before seeing the $k+1$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 the 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 value of the objective function, or for oblivious adversaries, the competitive ratio (strict or asymptotic), in offline algorithms *cost*() can be the approximation ratio or the time complexity, 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 can you 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 distribution μ.

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. In addition, for a maximization problem (for example, see the poof of Theorem 5.5.5 in Chapter 5), the statement of the Yao minimax principle is different than for minimization games (i.e., the inequality is reversed) . 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 require a stand-alone black-box statement.

3.8 Upper Bound for Paging

In this section we present an algorithm called Mark that achieves 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 11 and it works as follows. The algorithm keeps track of cache contents, and associates 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 often be 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 11 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 are 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 the duration of block B_i it will never be evicted). Therefore, the probability that the second page of type (2) is present (unmarked) 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 $H_k \sum_i m_i$. $\square$

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 9 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 a randomized algorithm and show that after applying the conversion in the proof of Theorem 3.4.3, you get back Algorithm 9 in Chapter 2.

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

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

6. If requested pages are restricted to come from $[k + 1]$, prove that 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.

8. Let A be an $n \times n$ matrix with entries in $\{+1, -1\}$. Let a_i denote the i^{th} column of A, i.e., $A = (a_1 a_2 \cdots a_n)$. Let x be an $n \times 1$ vector with entries in $\{+1, -1\}$. Consider the result of the matrix-vector multiplication $Ax = \sum_{i=1}^{n} a_i x_i$. The j^{th} row of this result is denoted by $(Ax)_j$. The j^{th} row is said to be *good* if $(Ax)_j \ge 0$.

   The Maximum Goodness problem is the following: given A the goal is to find x that maximizes the number of good rows in Ax.

In the online version of the problem, the columns of A are revealed one by one, i.e., a_1, a_2, $\ldots$, a_n. An online algorithm responds to a_i with an irrevocable decision on how to set the i^{th} entry of x, i.e., $x_i \in \{+1, -1\}$.

(a) Design a simple randomized online algorithm that for every A outputs x that gives at least $n/2$ good rows in Ax on average. Explain your algorithm in plain English and prove that it achieves the stated expected performance.

(b) Derandomize the algorithm from part (a): present a deterministic algorithm that on every A outputs x that gives at least $n/2$ good rows. Prove that your algorithm satisfies this guarantee.

Hint: let Z denote the number of good rows produced by the randomized algorithm from part (5.1). Let a_1 be the first column of A revealed by adversary and consider x_1 set randomly by the algorithm from part (5.1). Observe that $\mathbb{E}(Z|a_1) = \mathbb{E}\mathbb{E}(Z|x_1, a_1) = \Pr(x_1 = -1)\mathbb{E}(Z|x_1 = -1, a_1) + \Pr(x_1 = +1)\mathbb{E}(Z|x_1 = +1, a_1)$. Provided that you proved $\mathbb{E}(Z) \geq n/2$ in the first part, it implies that either $\mathbb{E}(Z|x_1 = -1, a_1) \geq n/2$ or $\mathbb{E}(Z|x_1 = +1, a_1) \geq n/2$. How can you use this observation to decide on the value of x_1 deterministically?

3.10 Historical Notes and References

The different types of adversaries and their relative power is studied in Ben-David et al [32].

The modified bit guessing game is an adaption of the Böckenhauer el al [38] string guessing game that was used to establish inapproximations for online algorithms with advice.

The $2H_k$ *Mark* algorithm for randomized paging and the H_k lower bound for any randomized online algorithm is due to Fiat et al [80]. This was followed by McGeoch and Sleator who obtained the optimal H_k competitive *Partitioning* algorithm. Randomized paging is the most prominent natural example for which randomization leads to a signiifcant improvement in the competitive ratio. Achlioptas, Chrobak and Noga [3] provide a simpler and more efficient H_k competitive algorithm *Equitable* in which the time complexity for each request does not depend on the number of previous requests.

The minimax theorem was originally proved by John von Neumann [140] in the context of zero-sum games, and it was adapted to randomized algorithms by Andrew Yao [143]. It is the central technique used for proving negative results about randomized algorithms. Yao's application was in proving negative results concerning the minimum time needed for a problem in a given computational model. As we have seen in this chapter, it applies equally well to proving negative results concerning competitive ratios. Since paging is a minimization problem, the application of Yao's minimax principle in Section 3.6 follows the statement given in Section 3.7. For a maximization problem, we have the ambiguity as to whether or not competitive ratios are stated to be at most or at least equal to one. In applying the principle when dealing with ratios $\rho < 1$, we need a different statement of the principle so as to prove an "upper bound" (i.e., a negative result) on the competitive ratio.