1 Introduction

This chapter introduces the concept of computing online, after briefly looking at algorithms from a more general perspective. As a starting point, we consider optimization problems and $\mathcal{NP}$-hardness on a very high level. Next, we describe one of the strategies used to attack such problems, namely approximation algorithms. As an example, we have a quick look at the simple knapsack problem, which is one of the most famous $\mathcal{NP}$-hard maximization problems. We study this problem as we will meet it again later in this book; in Chapter 6, we formulate an online version of the problem where we make use of our insights into the greedy strategy for the offline version. The goal is to quickly recall what we understand as computing problems and algorithms, not to give an introduction to the topic; we assume that the reader is already familiar with these concepts. The objective of these first few pages is to put into context what we learn next.

After that, we formally introduce the main topic of this book: *online computation*, that is, algorithms that work on problem instances which they do not know from the start, but that get revealed piece by piece. These algorithms are called *online algorithms*, and they are commonly analyzed using so-called *competitive analysis*, which compares their solution to an optimal one; this is a concept that is strongly related to studying the approximation ratios of algorithms described in the aforementioned setting. We start by giving a formal definition, which we link to the paging problem. Paging, as it is defined in this book, is a very simplified version of what is met in practical settings; a simple two-level memory hierarchy is assumed that consists of a small fast memory, called the *cache*, and a larger slow memory. The objective is to manage the memory such that the slow memory is accessed as rarely as possible. The problem is motivated by the difference in speed between the main memory and the CPU. Note that, from a practical point of view, we should be speaking of caching in this context, but we use the terminology that is established in the area of online algorithms. We are not concerned with the technical details of this problem, but we want to break it down to its essence. Paging will accompany

D. Komm, *An Introduction to Online Computation*,
Texts in Theoretical Computer Science. An EATCS Series,
DOI 10.1007/978-3-319-42749-2_1

us in Chapters 2 and 3, where we introduce two different models of computation, namely randomized computation and computation with advice. In this chapter, we state some basic results and present fundamental techniques to analyze deterministic online algorithms, and make clear to which points we need to pay special attention. For paging, we introduce some important strategies and give upper and lower bounds on their competitive ratios (analogously to the approximation ratio, the competitive ratio roughly corresponds to the factor by which a solution computed by an online algorithm is worse than an optimal solution). To analyze the latter, we consider worst-case instances and introduce a hypothetical *adversary* that tries to make the online algorithm at hand perform as badly as possible. We show an upper bound on the competitive ratio that solely depends on the given cache size k; this is done by proving that an online algorithm that implements a simple *first in first out* (FIFO) strategy (basically treating the cache like a queue) achieves this competitive ratio. This bound is tight as there is an adversary that can make sure that no online algorithm can be more successful in general. Interestingly, we also show that FIFO's counterpart, that is, a *last in first out* (LIFO) strategy, is a lot worse, as is a *least frequently used* (LFU) strategy. After that, we introduce the general concept of *marking algorithms* for the paging problem. This class of algorithms, which contains algorithms that implement a *least recently used* (LRU) strategy, is also shown to achieve a competitive ratio of k. Finally, we quickly touch upon two ideas to possibly grant an online algorithm an advantage over the adversary, namely seeing into the future for a little bit, or having a larger cache than the optimal algorithm that the online algorithm is compared against.

1.1 Offline Algorithms

The term *algorithm* may without hesitation be called the central notion of computer science; it is the formal description of a strategy to solve a given instance of a problem. It is important to point out that this description is finite, but it should be applicable to *every* instance of the problem although there may be infinitely many. The origin of the word dates back to Muḥammad ibn Mūsā al-Khwārizmī, a Persian mathematician, who lived in the eighth and ninth century. In computer science, we are concerned with the study of these algorithms to both explore what is doable by means of computers, that is, which kind of work can be automated, and how well it can be done when satisfying certain conditions such as, for instance, bounding the time spent to solve the problem.

The investigation of the first point is based on one of the major breakthrough results of twentieth-century science. There are well-defined problems that cannot be solved algorithmically, that is, no matter how powerful a computer's resources will be at some point, it will not be able to answer these questions. In 1936, Alan Turing wrote his pioneering paper "On computable numbers with an application to the Entscheidungsproblem," introducing a formal definition of the notion of *algorithm* and then (using arguments similar to those in the proof of Gödel's fundamental

Incompleteness Theorem) showing that, for some particular problems, algorithms cannot give a correct answer. Today, we call his formalization the *Turing machine* in his honor. The informal term *algorithm* is formalized by Turing machines that always finish their work in finite time (they "halt"). Most computer scientists agree that these hypothetical machines do indeed formalize what we understand as algorithms. Since then, a lot of effort has been made to further refine Turing's result, and nowadays the field of *computability* is one of the cornerstones of theoretical computer science.

One such question that we cannot answer algorithmically in general is the *halting problem*, which asks

> *Does a given Turing machine halt on a given input or does it run forever?*

Even though there are only two possible answers, namely "yes" and "no," no algorithm can figure out the correct answer for all possible Turing machines. We call such a problem a *decision problem*. In this book, we only deal with computable problems, that is, problems for which, in principle, algorithms can compute a solution. As an example, such a computable decision problem can be given by the question

> *Is a given natural number a prime number?*

Obviously, there are infinitely many instances of this problem, namely all natural numbers. Moreover, we can find a finite description to answer this question for any given such number $x \in \mathbb{N}$. If x is either 0 or 1, we answer "no," and otherwise we check for every number $y \in \{2, 3, \ldots, \lfloor \sqrt{x} \rfloor\}$ whether it divides x. If we find such a y, we answer "no," otherwise we answer "yes."

In what follows, we usually ask questions that do not have a simple answer like "yes" or "no," but are more involved. A typical such question is

> *Given a traffic network, what is the fastest tour that visits all cities on the map exactly once and returns to the starting point?*

The above problem is the famous *traveling salesman problem* (TSP) and we know that, given enough time, we can answer it with the fastest tour there is, for any given instance. The next definition formalizes such an *optimization problem*; there are two different objectives, either to minimize some cost or to maximize some gain; we thus speak of *minimization* or *maximization problems*. For the TSP, we want to minimize some cost, namely the total traveling time that is associated with every tour.

Definition 1.1 (Optimization Problem). An *optimization problem* Π consists of a set of *instances* $\mathcal{I}$, a set of *solutions* $\mathcal{O}$, and three functions $\text{sol}\colon \mathcal{I} \to \mathcal{P}(\mathcal{O})$, $\text{quality}\colon \mathcal{I} \times \mathcal{O} \to \mathbb{R}$, and $\text{goal} \in \{\min, \max\}$. For every instance $I \in \mathcal{I}$, $\text{sol}(I) \subseteq \mathcal{O}$ denotes the set of *feasible solutions* for I. For every instance $I \in \mathcal{I}$ and every feasible solution $O \in \text{sol}(I)$, $\text{quality}(I, O)$ denotes the *measure* of I and O. An *optimal solution* for an instance $I \in \mathcal{I}$ of Π is a solution $\textsc{Opt}(I) \in \text{sol}(I)$ such that

$$\text{quality}(I, \textsc{Opt}(I)) = \text{goal}\{\text{quality}(I, O) \mid O \in \text{sol}(I)\} .$$

If $\text{goal} = \min$, we call Π a *minimization problem* and write "cost" instead of "quality." Conversely, if $\text{goal} = \max$, we say that Π is a *maximization problem* and write "gain" instead of "quality."

We call an algorithm *consistent* for a given problem Π if it computes a feasible solution for every given instance (input). We denote the solution computed by an algorithm Alg on an instance I by $\textsc{Alg}(I)$. When, for instance, considering a minimization problem, we denote the cost incurred by Alg on the instance I by $\text{cost}(I, \textsc{Alg}(I))$; likewise, for maximization problems, we write $\text{gain}(I, \textsc{Alg}(I))$ for the gain of Alg's solution when given I. To have an easier notation, we usually simply write $\text{cost}(\textsc{Alg}(I))$ or $\text{gain}(\textsc{Alg}(I))$, respectively.

We now have a framework of (computable) optimization problems and algorithms to solve them. However, not only from a practical point of view, we are typically not satisfied to merely know that we are able to design an algorithm to solve some given problem; we would also like to get the solution while obeying some given restrictions. One such restriction might be an upper bound on the running time of the algorithm at hand. Here, computer scientists consider an algorithm *efficient* if its running time is in $\mathcal{O}(n^k)$ for all inputs of length n and some natural number k which is independent of n; we call such algorithms *polynomial-time algorithms*. Consider our algorithm for testing whether a given natural number is a prime number, and let us call this algorithm Prime. Furthermore, assume that the input x is encoded as a binary string of length n. We can roughly estimate the running time of Prime as follows. Due to its length, x has a size of around 2^n; if x is prime, Prime tests the divisibility of x for roughly $\sqrt{2^n} = 2^{n/2}$ natural numbers and thus its running time grows exponentially in n; hence, Prime is not efficient. Of course, if x is not a prime, but, say, divisible by 2, Prime finishes with the answer "no" very quickly; but we are interested in the worst-case behavior of the algorithms we study, and we keep this point of view throughout this book.

All decision problems that can be solved in polynomial time are members of the class $\mathcal{P}$. The class $\mathcal{NP}$ contains all decision problems Π for which we can *verify* in polynomial time that, for all instances of Π that have the answer "yes," the answer is indeed "yes." The exact relation between the two classes $\mathcal{P}$ and $\mathcal{NP}$ is surely one of the most important and famous questions in computer science and mathematics.

It is easy to see that $\mathcal{P} \subseteq \mathcal{NP}$; of course, if we can decide whether a given instance of a decision problem is a "yes" instance, we can also verify that this is the case. However, we do not know yet whether the above inclusion is strict, that is, whether

$$\mathcal{P} \subsetneq \mathcal{NP} \quad \text{or} \quad \mathcal{P} = \mathcal{NP}$$

is true. In the following, we will assume the former, that is, $\mathcal{P} \neq \mathcal{NP}$. To at least identify a class of problems in $\mathcal{NP}$ that are promising candidates to be outside $\mathcal{P}$, one defines a class of problems that are "hard" in the sense that the ability to solve any of them in polynomial time immediately allows us to solve all problems in $\mathcal{NP}$ in polynomial time. These problems, which are not necessarily decision problems, are called *$\mathcal{NP}$-hard*. The TSP is such a problem; primality testing is not. If a problem is $\mathcal{NP}$-hard and a member of $\mathcal{NP}$, it is called *$\mathcal{NP}$-complete*.

Unless $\mathcal{P} = \mathcal{NP}$, we cannot hope for an algorithm that works in polynomial time for an $\mathcal{NP}$-hard optimization problem. We can, however, sometimes at least hope to get a "good" solution in a time that is acceptable. This means that we pay with accuracy (such a solution will not be optimal in general), but we can get a satisfactory upper bound on the time we need to spend; such solutions are computed by *approximation algorithms*, which we formally define in what follows.

Definition 1.2 (Approximation Algorithm). Let Π be an optimization problem, and let ALG be a consistent algorithm for Π. For $r \geq 1$, ALG is an *r-approximation algorithm* for Π if, for every $I \in \mathcal{I}$,

$$\text{gain}(\text{OPT}(I)) \leq r \cdot \text{gain}(\text{ALG}(I))$$

if Π is a maximization problem, or

$$\text{cost}(\text{ALG}(I)) \leq r \cdot \text{cost}(\text{OPT}(I))$$

if Π is a minimization problem.

The *approximation ratio of* ALG is defined as

$$r_{\text{ALG}} := \inf\{r \geq 1 \mid \text{ALG is an } r\text{-approximation algorithm for } \Pi\} .$$

In general, r (and thus r_{ALG}) is not necessarily constant, but may be a function that depends on the input length n. Intuitively, a 2-approximation algorithm for a minimization problem Π is thus an algorithm that computes, for any instance I of Π, an output such that the cost of this solution is never more than twice as large as the cost of an optimal solution. What we want are of course approximation algorithms that are efficient, that is, work in polynomial time. For the TSP, which we described above, there is, for instance, a polynomial-time 3/2-approximation algorithm, known as the *Christofides algorithm*, if the input satisfies certain natural conditions. However, if these conditions are not met, it can be proven that there are

instances of the TSP of length n such that there is no polynomial-time approximation algorithm with an approximation ratio bounded by any polynomial in n.

Now let us consider the following maximization problem. Suppose you want to pack a number of objects into a knapsack with a given weight capacity. Each such object has an assigned weight that also corresponds to its value and is given by a positive integer. The input is described by the weights of the objects and the weight capacity of the knapsack. The goal is to maximize the total value of the objects packed.

Definition 1.3 (Simple Knapsack Problem). The simple knapsack problem is a maximization problem. An instance I is given by a sequence of $n+1$ positive integers $B, w_1, w_2, \ldots, w_n$, where we consider w_i with $1 \leq i \leq n$ to be the weight of the ith object; B is the capacity of the knapsack. A feasible solution for I is any set $O \subseteq \{1, 2, \ldots, n\}$ such that

$$\sum_{i \in O} w_i \leq B \; .$$

The gain of a solution O and a corresponding instance I is given by

$$\text{gain}(I, O) = \sum_{i \in O} w_i \; .$$

The goal is to maximize this number.

In the following, we assume that the weight of every object is smaller than B. This makes sense as all objects that are heavier than the knapsack's capacity cannot be part of any solution and may thus be neglected. It is well known that there is no polynomial-time algorithm for the simple knapsack problem that solves every given instance optimally, unless $\mathcal{P} = \mathcal{NP}$. This problem will be very interesting for us later in a different setting; for now, we just want to give an easy idea of how to approximate optimal solutions in reasonable time. More precisely, we give a simple 2-approximation algorithm that works in polynomial time. The idea is to first sort the objects $w_1, w_2, \ldots, w_n$ in descending order (with respect to their weights) and then follow what is called a *greedy strategy*.

This simply means to pack objects into the knapsack starting with the heaviest one, then the second heaviest, and so on, as long as there is space left in the knapsack; the corresponding algorithm KnGreedy is shown in Algorithm 1.1.

It is easy to see that the running time of KnGreedy is in $\mathcal{O}(n \log n)$. Sorting n integers can be done in $\mathcal{O}(n \log n)$ and after that, every object is inspected at most one more time. It is not much more difficult to show that the gain of any solution computed by KnGreedy is at least half as large as the optimal gain.

Theorem 1.1. KnGreedy *is a polynomial-time* 2*-approximation algorithm for the simple knapsack problem.*

```
O := ∅;                                   // Initialization
s := 0;
i := 0;
sort w_1, w_2, ..., w_n;                  // Preprocessing; we assume that
                                          // now w_1 ≥ w_2 ≥ ... ≥ w_n
while i < n and s + w_{i+1} ≤ B do         // Pack objects greedily
    O := O ∪ {i + 1};
    s := s + w_{i+1};
    i := i + 1;
output O;
end
```

Algorithm 1.1. KnGreedy for the simple knapsack problem.

Proof. Consider any instance $I = (B, w_1, w_2, \ldots, w_n)$ of the simple knapsack problem, and assume without loss of generality that $w_1 \geq w_2 \geq \ldots \geq w_n$. We distinguish two cases with respect to the total weight of the objects in I.

Case 1. If all objects fit into the knapsack, then KnGreedy is even optimal, as it packs all of them.

Case 2. Thus, we assume that the total weight is larger than B, and distinguish two more cases depending on the weight of the largest object in I.

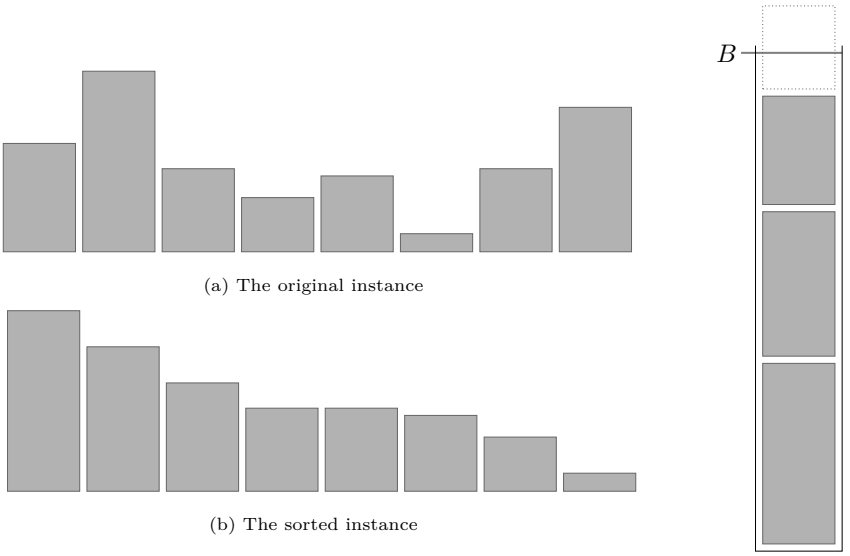
Case 2.1. Suppose there is an object w_i of weight at least $B/2$. We then have $w_1 \geq B/2$ and w_1 is always packed into the knapsack. Since B is an upper bound for any solution, it follows that the approximation ratio of KnGreedy is at most 2 in this case.

Case 2.2. Suppose that the weights of all objects are smaller than $B/2$, and let j be the index of the first object that is too heavy to be packed into the knapsack by KnGreedy. It follows from our assumption that $w_j < B/2$, and this implies that the space that is already occupied by the objects $w_1, w_2, \ldots, w_{j-1}$ must be larger than $B/2$. Thus, we immediately get an approximation ratio of at most 2 also in this case.

We conclude that KnGreedy is a polynomial-time 2-approximation algorithm for the simple knapsack problem. □

An example instance and the corresponding solution computed by KnGreedy are shown in Figure 1.1. Note that the greedy strategy works due to the preceding sorting of the weights of the objects. We now quickly discuss the tightness of our analysis of this algorithm. For any $n \geq 3$ and any arbitrarily large even B, consider an instance I that consists of a capacity B and the sequence

$$\frac{B}{2} + 1, \underbrace{\frac{B}{2}, \frac{B}{2}, \ldots, \frac{B}{2}}_{n-1 \text{ times}} \tag{1.1}$$



(a) The original instance

(b) The sorted instance

(c) The solution

Figure 1.1. The greedy strategy; first sort, then pack greedily what fits.

of weights. If given I as input, KnGreedy packs the first object into the knapsack, which results in a situation where no more objects may be packed; thus, we have $\text{gain}(\text{KnGreedy}(I)) = B/2 + 1$. On the other hand, any optimal solution $\text{Opt}(I)$ for I may safely pack any two objects into the knapsack except the first one, and therefore $\text{gain}(\text{Opt}(I)) = B$. It follows that

$$r_{\text{KnGreedy}} \geq \frac{\text{gain}(\text{Opt}(I))}{\text{gain}(\text{KnGreedy}(I))} = \frac{B}{\frac{B}{2}+1} = \frac{2}{1+\frac{2}{B}} ,$$

which tends to 2 with increasing B.

This sums up our first ideas of how to deal with optimization problems that are in general regarded as *infeasible*. We pay with accuracy, and we gain speed in return. Of course, there are smarter methods to deal with the (simple) knapsack problem than just following a simple greedy approach. In particular, there is an algorithm that achieves an approximation ratio of $1+\varepsilon$ running in time $\mathcal{O}(n^3 \cdot 1/\varepsilon)$, for every constant $\varepsilon > 0$; such an algorithm that achieves an arbitrarily good approximation ratio $1+\varepsilon$ in a time that is polynomial both in n and $1/\varepsilon$ is called a *fully polynomial-time approximation scheme* (FPTAS).

Exercise 1.1. Algorithm 1.1 shows a very naive implementation of KnGreedy as it already stops packing objects into the knapsack after one object is encountered that is too

heavy. However, there might still be smaller objects in the input. Would it help to consider them? How would the analysis change?

Exercise 1.2. As an input, an algorithm for the TSP expects a complete graph with edge weights that are positive real numbers; a feasible solution is a Hamiltonian cycle in the given graph, and the goal is to output such a cycle with minimum cost (the cost being the sum of all weights of edges that it consists of). The greedy algorithm TSPGreedy starts with an arbitrary vertex and follows an edge of minimum weight to a yet unvisited neighboring vertex; this is iterated until all n vertices are visited. Then, the last and first vertex are connected to obtain a Hamiltonian cycle. Argue on an intuitive level why this approach is bad for the TSP.

We will revisit TSPGreedy in Chapter 8. Although greedy strategies are bad for many other optimization problems, they will play an important role throughout this book. Let us therefore end this section with the remark that there are optimization problems for which they work quite well. One such problem is the *minimum spanning tree problem* (MSTP) to which we will also return in Chapter 8.

Exercise 1.3. An algorithm for the MSTP also expects a complete graph G with edge weights that are positive real numbers as input. The goal is to compute a minimum spanning tree of G, that is, a subgraph of G that is connected, does not contain any cycles, contains all vertices, and that has minimum cost. Consider the following greedy algorithm Kruskal. If n denotes the number of vertices of G, Kruskal works in $n-1$ rounds. In every round, an edge e of G is chosen to be part of the solution; e is an edge of minimum weight that is not yet chosen and that does not close a cycle with respect to the already chosen edges. Prove that Kruskal always computes an optimal solution.

1.2 Online Algorithms and Paging

As described above, an algorithm computes a well-defined output for any given instance of a computational problem. In this context, we have, so far, briefly spoken about efficient approximation algorithms for $\mathcal{NP}$-hard problems. In other words, we have imposed certain requirements on the algorithms we want to study; we demanded that they have a polynomial running time while producing an output for any instance of the given problem. If $\mathcal{P} \neq \mathcal{NP}$, we may thus only hope to get an approximate solution. In the following, we want to focus on another restriction that we encounter in practice. Here, we are not concerned with not taking too much time, but with the fact that we do not know the whole instance of the problem at hand in advance. Until now, we assumed that from the start we have all information available that we need to compute a solution.

This assumption may be unrealistic in scenarios such as the following. From a practical point of view, the basic design principle of modern computers follows the *von Neumann architecture*. Computers suffer from the fact that the CPU is usually a lot faster than its main memory, which leads to a bad overall performance as

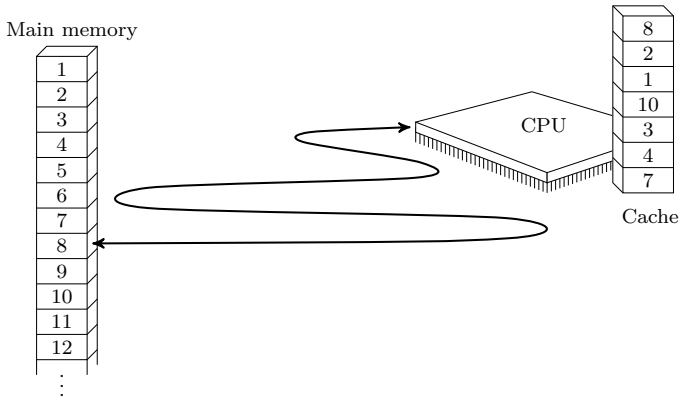


Figure 1.2. A schematic view of an environment in which we study paging. In the main memory, pages with indices $1, 2, \ldots, m$ are stored, while a small subset of them is currently stored in the cache of size $k \ll m$.

the CPU cannot be utilized to its full extent. To overcome this drawback, another memory, the so-called *cache*, is introduced. This memory is a lot faster than the main memory; however, it is therefore also a lot more expensive and thus smaller.

Now we consider the important task of an operating system to manage the cache in such a way that we need to access the main memory as rarely as possible. We want to study the essence of what makes this problem hard, and thus we look at a simplified version of the problem; see Figure 1.2. In a practical setting, we are confronted with a much more complicated situation, and there are usually many different levels of caches; here, we only deal with a two-level memory hierarchy. More precisely, for our theoretical investigations, we make the following assumptions.

- There are two different types of memory, namely the aforementioned main memory and the cache.
- Both of them may store chunks of data of a fixed size, which we call *pages*; we assume that each page has a size of 1.
- The main memory can store m pages, denoted by $p_1, p_2, \ldots, p_m$, the cache can store k pages; we assume that there are m pages in total.
- As the cache is a lot more expensive than the main memory, we have $m \gg k$.
- The input is subdivided into discrete *time steps* $T_1, T_2, \ldots, T_n$. During such a time step T_i with $1 \leq i \leq n$, exactly one page is *requested*, that is, needed in the ongoing computation.
- The CPU can only access pages that are stored in the cache.
- As a consequence, if a requested page is not in the cache in the corresponding time step, it needs to be loaded into it from the main memory, which causes a

cost of 1; we call this situation a *page fault*. In this case, if there is no space left in the cache, a *victim page* has to be selected and removed from the cache to make room.

- Conversely, accessing a page from the cache does not induce any cost.

This problem is called the *paging problem*. For convenience, we will simply refer to it as "paging." An algorithm for paging is basically defined by the strategy it uses to choose the victim pages, that is, which page it replaces in the cache if a page fault occurs.

Paging is a prominent member of a broad class of problems for which the concrete input is revealed piecewise at runtime. Such problems are called *online problems*, and it is obvious that we encounter them in many practical situations, when, for instance, humans frequently interact with computers. We will see many other examples throughout this book. However, at first, we need a formalism that enables us to study strategies to handle such problems. Similarly to "offline" problems, which we described in the previous section, the objective in online problems can either be to minimize some cost or to maximize some gain; for instance, for paging we want to minimize the number of page faults that occur.

We start by defining online problems formally. The following definition is similar to that of offline problems from Definition 1.1, but here we need to introduce the two notions *request* and *answer*, which will be crucial for our further formalizations.

Definition 1.4 (Online Problem). An *online problem* Π consists of a set of instances $\mathcal{I}$, a set of *solutions* $\mathcal{O}$, and three functions sol, quality, and goal with the same meaning as for general optimization problems according to Definition 1.1. Every instance $I \in \mathcal{I}$ is a sequence of *requests* $I = (x_1, x_2, \ldots, x_n)$ and every output $O \in \mathcal{O}$ is a sequence of *answers* $O = (y_1, y_2, \ldots, y_n)$, where $n \in \mathbb{N}^+$ (thus, all instances and solutions are finite). An *optimal solution* for an instance $I \in \mathcal{I}$ of Π is a solution $\textsc{Opt}(I) \in \text{sol}(I)$ such that

$$\text{quality}(I, \textsc{Opt}(I)) = \text{goal}\{\text{quality}(I, O) \mid O \in \text{sol}(I)\} .$$

If goal = min, we call Π an *online minimization problem* and write "cost" instead of "quality." Conversely, if goal = max, we say that Π is an *online maximization problem* and write "gain" instead of "quality."

As in the case of offline problems, we usually simply write $\text{cost}(O)$ instead of $\text{cost}(I, O)$ if I is clear from the context. Definition 1.4 does not yet formalize what we mean by online computation; namely, that the output must be computed with incomplete information. In particular, we want to model that an algorithm that works on such a problem

1. only knows a prefix of the input in every given time step,
2. makes decisions that are based only on this knowledge, and

3. may not revoke any decision it already made.

We formalize these rules in the following definition of online algorithms, which are the main objects of study throughout this book.

Definition 1.5 (Online Algorithm). Let Π be an online problem and let $I = (x_1, x_2, \ldots, x_n)$ be an instance of Π. An *online algorithm* ALG for Π computes the output $\text{ALG}(I) = (y_1, y_2, \ldots, y_n)$, where y_i only depends on $x_1, x_2, \ldots, x_i$ and $y_1, y_2, \ldots, y_{i-1}$; $\text{ALG}(I)$ is a feasible solution for I, that is, $\text{ALG}(I) \in \text{sol}(I)$.

Although the notion *time step* is not explicitly used in Definitions 1.4 and 1.5, we implicitly assign the ith request x_i and the corresponding answer y_i to time step T_i.

How do we assess the output quality of an online algorithm? A natural approach is to have a definition analogous to the approximation ratio defined in Definition 1.2 for offline algorithms. As a matter of fact, we are facing a similar situation. The approximation ratio formalizes what we can achieve for an $\mathcal{NP}$-hard problem when computing in polynomial running time (if we assume $\mathcal{P} \neq \mathcal{NP}$). In the context of online computation, we ask what we can achieve when we do not know the whole input in advance. So there are two different restrictions, and in both cases we want to know what we pay for obeying them. In Definition 1.2, we compared the cost or gain of a solution computed by an algorithm to the cost or gain of an optimal solution. This is exactly the same thing that we do with online algorithms. More specifically, we now define the term *competitive ratio* analogously to the approximation ratio; however, there are some small differences.

Definition 1.6 (Competitive Ratio). Let Π be an online problem, and let ALG be a consistent online algorithm for Π. For $c \geq 1$, ALG is *c-competitive* for Π if there is a non-negative constant α such that, for every instance $I \in \mathcal{I}$,

$$\text{gain}(\text{OPT}(I)) \leq c \cdot \text{gain}(\text{ALG}(I)) + \alpha$$

if Π is an online maximization problem, or

$$\text{cost}(\text{ALG}(I)) \leq c \cdot \text{cost}(\text{OPT}(I)) + \alpha$$

if Π is an online minimization problem. If these inequalities hold with $\alpha = 0$, we call ALG *strictly c-competitive*; ALG is called *optimal* if it is strictly 1-competitive.

The *competitive ratio* of ALG is defined as

$$c_{\text{ALG}} := \inf\{c \geq 1 \mid \text{ALG is } c\text{-competitive for } \Pi\} .$$

If the competitive ratio of ALG is constant and the best that is achievable by any online algorithm for Π, we call ALG *strongly c_{ALG}-competitive*.

Note that, in online computation, the term *optimal solution* even has a somewhat stronger meaning than in the context of offline algorithms. In the latter case, optimal solutions to instances of "hard" problems were those that can hypothetically be computed when we are given more time (as opposed to polynomial time). In an online setting, an optimal solution even refers to solutions that usually can only be computed based on some knowledge (for instance, the complete instance) that we generally do not possess; therefore, we call $\textsc{Opt}(I)$ an *optimal offline solution* for I. Additionally, in online computation, we do not take the running time of the algorithms we study into account. For most of the problems we will investigate, the designed online algorithms will be efficient, but there will also be algorithms that take far longer to compute a solution. The same holds for the memory that our algorithms use. In particular, we will always assume that there are no space restrictions, and that an online algorithm is able to remember all choices that it made before in any time step. It is important to keep in mind that "optimal" refers to a solution quality we can generally not guarantee, although Definition 1.6 does speak about "optimal online algorithms." On the other hand, by "strongly competitive," we mean "best possible." Moreover, let us emphasize that the term "optimal" only corresponds to the output quality of an algorithm and is independent of any of its other parameters such as the aforementioned time and space complexities. We can think of $\textsc{Opt}(I)$ as being computed by a hypothetical *offline algorithm* Opt which is optimal in this sense. Throughout this book, we will either speak about Opt or $\textsc{Opt}(I)$ depending on what is more intuitive in the current situation.

Usually, Alg is clear from the context and we simply write "c" instead of "$c_{\textsc{Alg}}$." At times, we have to speak about the cost or gain of an online algorithm Alg on a subsequence of an instance I or even a single request; we will denote this by, for instance, $\text{cost}(\textsc{Alg}((x_j, x_{j+1}, \ldots, x_k)))$, for some j, k with $1 \leq j \leq k \leq n$. Moreover, if we refer to a concrete instance I, we sometimes speak of the *performance of* Alg *on* I; this performance is a lower bound on Alg's competitive ratio.

Note that, similarly to the approximation ratio, an online algorithm has a better solution quality the smaller its competitive ratio c is, and that c is never smaller than 1. Furthermore, both measurements are *worst-case* measures, that is, we are interested in studying how well an algorithm works on its hardest instances for the given problem.

When analyzing and classifying online algorithms with respect to their competitive ratios, we speak of *competitive analysis*. As with the approximation ratio, the competitive ratio is not necessarily constant, but may be a function of the input length n. We use the following terminology.

- If the competitive ratio of some online algorithm Alg is at most c, where c is a constant, we call Alg *competitive* or (depending on whether this holds for $\alpha = 0$) *strictly competitive*. If an online algorithm does not possess a competitive ratio with an upper bound that is independent of the input length, we call it *not competitive*.

- It is fine to call an online algorithm competitive if its competitive ratio depends on some parameter of the studied problem such as, for instance, the cache size k when we are dealing with the paging problem. Such parameters are known to the online algorithm in advance.
- There are, however, problems where we have to be very careful with this classification. For instance, in Chapter 5 we will deal with a problem for which the input length is bounded from above by a parameter of the problem.

Another difference between the competitive ratio and the approximation ratio is that the former uses an additive constant α in its definition, which is not present in the definition of the approximation ratio. This constant plays an important role in an online setting. If we consider an (offline) approximation algorithm that works well on all but finitely many instances, we can always include finitely many exceptions in our algorithm. In an online setting, this is not possible since an online algorithm cannot distinguish these exceptional cases at the beginning of its computation. However, using an appropriate value of α, we can also cope with a finite number of exceptions here. To prove lower bounds, the constant α forces us to construct infinitely many instances with increasing costs or gains. It is not sufficient to construct a finite number of instances, not even an infinite number where the costs or gains are bounded by some constant. Let us give an example of a hypothetical online minimization problem to illustrate this point.

Example 1.1. Consider some online minimization problem Π and suppose we can show that, for every online algorithm ALG for Π, there is an instance I on which it has a cost of at least 10, while the optimal solution $\text{OPT}(I)$ has cost 1. We are now tempted to conclude that every online algorithm for Π is at best 10-competitive. If we have a closer look at Definition 1.6, however, we see that this is actually not allowed. In fact, there may still be 1-competitive online algorithms for Π, because, if we choose $c = 1$ and $\alpha = 9$, the inequality

$$\text{cost}(\text{ALG}(I)) \leq 1 \cdot \text{cost}(\text{OPT}(I)) + 9$$

from Definition 1.6 still holds if we just plug in the values $\text{cost}(\text{ALG}(I)) = 10$ and $\text{cost}(\text{OPT}(I)) = 1$.

On the other hand, suppose we can give a set of infinitely many instances of Π such that, for every such instance of length n, every online algorithm has a cost of at least $10n$, while OPT has cost n. In this case, we may indeed state that every online algorithm for Π is at best 10-competitive. There is no constant α such that, for some arbitrarily small constant $\varepsilon > 0$ and any instance I from above, we have

$$\text{cost}(\text{ALG}(I)) \leq (10 - \varepsilon) \cdot \text{cost}(\text{OPT}(I)) + \alpha \ ,$$

as

$$10n \leq (10 - \varepsilon)n + \alpha \iff \alpha \geq \varepsilon n$$

leads to a contradiction to the fact that α is constant. ♢

Next, we formalize this observation; we start with online minimization problems. Suppose there is a set of instances $\mathcal{I} = \{I_1, I_2, \ldots\}$ such that $|I_i| \leq |I_{i+1}|$, and such that the number of different input lengths in $\mathcal{I}$ is infinite. Furthermore, suppose we can show that, for every online algorithm ALG,

$$\frac{\text{cost}(\text{ALG}(I_i))}{\text{cost}(\text{OPT}(I_i))} \geq c(n) \;, \tag{1.2}$$

where $n = |I_i|$ and $c\colon \mathbb{N}^+ \to \mathbb{R}^+$ is a function that increases unboundedly with n. If ALG were competitive, there would be two constants c' and α such that

$$\text{cost}(\text{ALG}(I_i)) \leq c' \cdot \text{cost}(\text{OPT}(I_i)) + \alpha \;,$$

and together with (1.2) we obtain

$$(c(n) - c') \cdot \text{cost}(\text{OPT}(I_i)) \leq \alpha,$$

which is a contradiction (under the reasonable assumption that the optimal cost is bounded from below by a positive constant). Therefore, whenever we show a lower bound on the competitive ratio that increases unboundedly with the input length, we do not need to consider α at all; in this case, we can conclude that the given online algorithm is not competitive. Note that the same argumentation holds for any function c' with $c'(n) \in o(c(n))$. Now let us consider competitive online algorithms.

Theorem 1.2. *Let Π be an online minimization problem, and let $\mathcal{I} = \{I_1, I_2, \ldots\}$ be an infinite set of instances of Π such that $|I_i| \leq |I_{i+1}|$, and such that the number of different input lengths in $\mathcal{I}$ is infinite. If there is some constant $c \geq 1$ such that*

(i) $\dfrac{\text{cost}(\text{ALG}(I_i))}{\text{cost}(\text{OPT}(I_i))} \geq c$ *, for every $i \in \mathbb{N}^+$, and*

(ii) $\lim\limits_{i\to\infty} \text{cost}(\text{OPT}(I_i)) = \infty$,

for any online algorithm ALG*, then there is no $(c-\varepsilon)$-competitive online algorithm for Π, for any $\varepsilon > 0$.*

Proof. For a contradiction, suppose that both conditions (i) and (ii) are satisfied, but there still is a $(c-\varepsilon)$-competitive online algorithm ALG$'$ for Π, for some $\varepsilon > 0$. Thus, by the definition of the competitive ratio, there is a constant α such that

$$\text{cost}\big(\text{ALG}'(I_i)\big) \leq (c-\varepsilon) \cdot \text{cost}(\text{OPT}(I_i)) + \alpha \;,$$

and thus (assuming that the optimal cost is never zero)

$$\frac{\text{cost}(\text{ALG}'(I_i))}{\text{cost}(\text{OPT}(I_i))} - \frac{\alpha}{\text{cost}(\text{OPT}(I_i))} \leq c - \varepsilon \;, \tag{1.3}$$

for every $i \in \mathbb{N}^+$. Due to condition (i), the first term of (1.3) is at least c. Furthermore, (ii) implies that there are infinitely many instances for which the second term of (1.3) is smaller than ε, which is a direct contradiction. □

Instead of speaking of $c - \varepsilon$, we will sometimes (unless some ε is explicitly used in a proof) simply state that there is no online algorithm that is "better than c-competitive." However, formally it could still be the case that there is, for instance, some $(c - 1/n)$-competitive online algorithm. Then, (1.3) does not necessarily lead to a contradiction with (ii). Throughout this book, in this context "better than" will always mean "better by some arbitrarily small constant ε." Moreover, sometimes we will not define the set $\mathcal{I}$ explicitly, but only speak of one "representative" instance.

For maximization problems, we can prove a statement analogous to Theorem 1.2.

Theorem 1.3. *Let Π be an online maximization problem, and let $\mathcal{I} = \{I_1, I_2, \ldots\}$ be an infinite set of instances of Π such that $|I_i| \leq |I_{i+1}|$, and such that the number of different input lengths in $\mathcal{I}$ is infinite. If there is some constant $c \geq 1$ such that*

(i) $\dfrac{\text{gain}(\text{OPT}(I_i))}{\text{gain}(\text{ALG}(I_i))} \geq c$ *, for every $i \in \mathbb{N}^+$, and*

(ii) $\lim\limits_{i \to \infty} \text{gain}(\text{OPT}(I_i)) = \infty$,

for any online algorithm ALG*, then there is no $(c - \varepsilon)$-competitive online algorithm for Π, for any $\varepsilon > 0$.*

Proof. Again, we do not need to consider α when the given online algorithm is not competitive. For a contradiction, suppose that both conditions (i) and (ii) hold, but there is a $(c - \varepsilon)$-competitive online algorithm ALG$'$ for Π, for some $\varepsilon > 0$. It follows that there is some constant α such that

$$\frac{\text{gain}(\text{OPT}(I_i))}{\text{gain}(\text{ALG}'(I_i))} - \frac{\alpha}{\text{gain}(\text{ALG}'(I_i))} \leq c - \varepsilon \,, \tag{1.4}$$

for every $i \in \mathbb{N}^+$. Since, due to (i) and (ii), ALG$'$ would not be competitive if its gain were bounded by a constant, we can assume that

$$\lim_{i \to \infty} \text{gain}\big(\text{ALG}'(I_i)\big) = \infty \,. \tag{1.5}$$

Due to (i), the first term of (1.4) is at least c; due to (1.5), there are infinitely many instances for which the second term is smaller than ε, which is a contradiction. □

To sum up, if we are not speaking about the strict competitive ratio, but allow $\alpha > 0$ when proving lower bounds, we will always try to construct an infinite set of instances such that the conditions (i) and (ii) of Theorem 1.2 (Theorem 1.3, respectively) are satisfied when dealing with online minimization (maximization, respectively) problems.

To this end, for paging, we will use the concept of *phases*, which consist of a number of consecutive time steps. We then show that every online algorithm is worse than an optimal solution by some factor c within one phase, and that it is possible to repeat phases for an arbitrary number of times. Thus, c is a lower bound on the competitive ratio of any online algorithm for paging. Before we start investigating the problem in terms of upper and lower bounds on the achievable competitive ratio, we define it formally.

Definition 1.7 (Paging). The *paging problem* is an online minimization problem. Let there be m memory pages $p_1, p_2, \ldots, p_m$, which are all stored in the main memory, where m is some positive integer. An instance is a sequence $I = (x_1, x_2, \ldots, x_n)$, such that $x_i \in \{p_1, p_2, \ldots, p_m\}$, for all i with $1 \leq i \leq n$, that is, the page x_i is requested in time step T_i. An online algorithm ALG for paging maintains a *cache* memory of size k with $k < m$, which is formalized by a tuple $B_i = (p_{j_1}, p_{j_2}, \ldots, p_{j_k})$ for time step T_i. At the beginning, the cache is initialized as $B_0 = (p_1, p_2, \ldots, p_k)$, that is, with the first k pages. If, in some time step T_i, a page x_i is requested and $x_i \in B_{i-1}$, ALG outputs $y_i = 0$. Conversely, if $x_i \notin B_{i-1}$, ALG has to choose a page $p_j \in B_{i-1}$, which is then removed from the cache to make room for x_i. In this case, ALG outputs $y_i = p_j$ and the new cache content is $B_i = (B_{i-1} \setminus \{p_j\}) \cup \{x_i\}$. The cost is given by $\text{cost}(\text{ALG}(I)) = |\{i \mid y_i \neq 0\}|$ and the goal is to minimize this number.

Note that our definition imposes some restrictions on algorithms designed for the problem, such as that it is impossible to remove pages if there is no page fault. We will see shortly that this is not as restrictive as it may seem; at some points, however, we will allow this constraint to be violated.

To consolidate our feeling for the problem, let us consider a simple instance of paging before we start the formal analysis.

Example 1.2. Suppose $k = 6$, there are m pages $p_1, p_2, \ldots, p_m$ in total, and according to Definition 1.7 the cache is initialized as

p_1	p_2	p_3	p_4	p_5	p_6

.

Now suppose that we are given an instance $I = (p_4, p_7, p_5, p_1, \ldots)$. In time step T_1, page p_4 is requested, which is already in the cache; thus, any online algorithm outputs "0" and there is no cost caused in this time step. The next request is page p_7, and therefore some page needs to be removed from the cache to make room. Assume page p_1 gets chosen to be replaced by p_7, which leads to the situation

p_7	p_2	p_3	p_4	p_5	p_6

.

After that, page p_5 can be loaded directly from the cache and again causes no cost. In time step T_4, however, the cache content needs to be changed once more, as p_1 is not present anymore. Hence, after four time steps, the cost is 2. It is easy to see that a strategy which replaced, for instance, p_4 instead of p_1 in time step T_2, only has a cost of 1 at this point. ♢

Having every algorithm start with the same pages in the cache seems to be a reasonable assumption. What we are interested in is to measure how well an online algorithm works compared to what it could hypothetically achieve in the given situation. Thus, we compare its solution to an optimal one that has the same

starting situation. As a matter of fact, also this assumption is less restrictive than it seems, because the head start that may come with a different cache content at the beginning can be hidden in the constant α from Definition 1.6.

Moreover, as already discussed, Definition 1.7 implies that an online algorithm for paging only removes a page from the cache if the currently requested page is not already in the cache; we call such algorithms *demand paging algorithms*. Of course, we could also think of an alternative definition where a page, or even an arbitrary number of pages, may be removed from and loaded into the cache in every time step. However, it can easily be shown that this does not give an online algorithm any advantage.

Exercise 1.4. Suppose that we change Definition 1.7 such that algorithms may start with different cache contents. Prove that this does not change the competitive ratio of any online algorithm ALG. More precisely, show that if ALG is c-competitive for paging as formalized in Definition 1.7, then ALG is also c-competitive if OPT has a different set of pages in its cache at the beginning.

Exercise 1.5. Show that an online algorithm that is allowed to replace an arbitrary number of pages in every time step can be converted to a demand paging algorithm, that is, an online algorithm that is in accord with Definition 1.7, without increasing its cost. Of course, for such an online algorithm the cost measurement changes. Such an algorithm pays 1 for each replacement of a page in the cache.

Exercise 1.6. So far, it cannot happen that the cache contains empty cells at any point in time, as it is full initially (with the pages $p_1, p_2, \ldots, p_k$), and the only operation to change the cache is to replace a page with another one. In what follows, we will study an online algorithm that is allowed to remove some pages from the cache without loading other pages into it. Show that also such an algorithm can be converted into a demand paging algorithm without increasing its cost. Again, the cost measurement has to be changed. Here, the removal of a page from the cache is free, while loading a page into the cache causes cost 1.

As already mentioned, an (online) algorithm for paging is basically defined by the strategy that it follows when a *page fault* occurs and a page in its cache (the victim page) needs to be replaced. There are many different strategies an algorithm may follow; let us describe a few.

- *First In First Out (FIFO).* With this strategy, the cache is organized as a queue. If a page must be evicted from the cache, the one residing in the cache for the longest time is chosen. The first k pages may be removed arbitrarily.
- *Last In First Out (LIFO).* This strategy is the counterpart to FIFO since it organizes the cache as a stack. In case of a page fault, the page that was most recently loaded into the cache is removed from the cache. On the first page fault, an arbitrary page may be replaced.
- *Least Frequently Used (LFU).* On a page fault, the page is removed that was so far used least frequently. Ties are broken arbitrarily.

- *Least Recently Used (LRU).* Here, on a page fault, the page is removed that was last requested least recently. Also here, the first k pages may be removed arbitrarily.
- *Flush When Full (FWF).* The cache gets completely emptied ("flushed") if a page is requested that is not already in the cache and there is no empty cell. This strategy does not comply with Definition 1.7 since it is not a demand paging strategy; it may remove multiple pages on a page fault, but only loads pages into the cache if the requested page is not already present. As stated in Exercise 1.6, we assume that only loading a page into the cache causes cost 1. An online algorithm that uses the FWF strategy can be converted to be in accord with Definition 1.7 without increasing the cost (as stated in Exercise 1.6).
- *Longest Forward Distance (LFD).* Here, on a page fault, the page is removed whose next request will be the latest.

In what follows, we denote, for instance, by Fifo an (online) algorithm that implements the FIFO strategy. Clearly, Lfd is an offline algorithm as it requires knowledge about the future input to replace a page. The other strategies are online strategies, but they have different solution qualities in terms of competitive analysis. The next sections are devoted to studying them in more detail.

1.3 An Upper Bound for Paging

In the preceding section, we defined that an online algorithm is competitive if its competitive ratio c is bounded by a constant with respect to the input length. For paging, this means that c may depend on both the cache size k and the size of the main memory m. Three of the above online algorithms are k-competitive, so their solution qualities do not at all depend on the number of pages that are available in total. As an example, we will consider Fifo; but before that, we introduce an important tool that will prove to be helpful in the subsequent analysis.

Definition 1.8 (k-Phase Partition). Let $I = (x_1, x_2, \ldots, x_n)$ be an arbitrary instance of paging. A *k-phase partition of I* assigns the requests from I to consecutive disjoint phases $P_1, P_2, \ldots, P_N$ such that

- phase P_1 starts with the first request for a page that is not initially in the cache. Then, P_1 contains a maximum-length subsequence of I that contains at most k distinct pages;
- for any i with $2 \leq i \leq N$, phase P_i is a maximum-length subsequence of I that starts right after P_{i-1} and again contains at most k distinct pages.

It is crucial to note that a phase does not necessarily end right after k distinct pages were requested, but right before a $(k+1)$th one is requested. The last phase of a k-phase partition is not necessarily complete. It is also important to note that a k-phase partition is defined on inputs, and not for algorithms; let us look at an example.

Example 1.3. Suppose we are dealing with paging with cache size 5, and we are given an input

$$(p_3, p_1, p_7, p_5, p_7, p_8, p_3, p_4, p_4, p_2, p_2, p_3, p_5, p_1, p_7, p_3, p_1, p_8, p_7, p_6) .$$

Recall that the cache is initialized as $(p_1, p_2, p_3, p_4, p_5)$. Then we obtain a k-phase partition

$$(p_3, p_1, \underbrace{p_7, p_5, p_7, p_8, p_3, p_4, p_4}_{P_1}, \underbrace{p_2, p_2, p_3, p_5, p_1, p_7, p_3, p_1}_{P_2}, \underbrace{p_8, p_7, p_6}_{P_3}) ,$$

where the last phase P_3 is incomplete. Observe that if we shift the phases by one, that is, if we consider the partition

$$(p_3, p_1, p_7, \underbrace{p_5, p_7, p_8, p_3, p_4, p_4, p_2}_{P'_1}, \underbrace{p_2, p_3, p_5, p_1, p_7, p_3, p_1, p_8}_{P'_2}, \underbrace{p_7, p_6}_{P'_3})$$

instead, there are still at least k distinct pages requested during any one phase (except during the last one P'_3). However, there are two differences between the previous phases and these ones. First, they do not have maximum length (with respect to containing k different pages) anymore; and second, since the first page requested in P_{i+1} was different from all pages in P_i, we observe that in P'_i there are k distinct pages requested that are different from the last page requested before P'_i starts. ♢

We now use a k-phase partition of the given input to analyze Fifo.

Theorem 1.4. Fifo *is strictly k-competitive for paging.*

Proof. Let $I = (x_1, x_2, \ldots, x_n)$ be any instance of paging and consider I's k-phase partition $P_1, P_2, \ldots, P_N$ according to Definition 1.8. Without loss of generality, we assume that $x_1 \notin \{p_1, p_2, \ldots, p_k\}$, that is, the sequence starts with a page fault for any algorithm.

Let us consider a fixed phase P_i with $1 \leq i \leq N$. First, we show that Fifo does not cause more than k page faults during P_i. By definition, there are at most k distinct pages requested during this phase. Let p be the first page that causes a page fault for Fifo during P_i. Then, out of all pages requested during P_i, p will be the first one that is removed again, and can thus cause a second page fault (for the same page). When p gets loaded into the cache, there are $k-1$ pages in the cache that get removed from the cache before p. Thus, p stays in the cache for the next $k-1$

page faults, and consequently no page causes more than one page fault during one phase. Therefore, there are at most k page faults in total during one phase.

Second, we argue that a fixed optimal solution $\textsc{Opt}(I)$ has to make at least one page fault for every phase. To this end, we shift all phases by one as in Example 1.3, leading to a new partition $P'_1, P'_2, \ldots, P'_N$, where P'_N might be the empty sequence; however, the first $N-1$ shifted phases must be complete. As already observed, since the phases P_i with $1 \leq i \leq N-1$ had maximum length, the phase P'_i now contains requests to k pages that differ from the page p' that was last requested before the start of P'_i. We know that p' is in the cache of Opt at the beginning of P'_i. Since there are k more requests different from p', Opt has to cause one page fault during P'_i. This adds up to $N-1$ page faults for $\textsc{Opt}(I)$ plus an additional one on x_1 at the beginning of I.

Since Fifo causes at most $N \cdot k$ page faults in total while Opt causes at least N, it follows that Fifo is strictly k-competitive. □

By similar reasoning to the preceding proof, it can be shown that Lru and (which might be surprising) Fwf are also strictly k-competitive.

Exercise 1.7. Prove that Lru is strictly k-competitive for paging.

Exercise 1.8. Prove that Fwf is also strictly k-competitive for paging. Recall that you need to change the definition of a paging algorithm for this case; Fwf has cost 1 whenever it loads a page into the cache (see Exercise 1.6).

Exercise 1.9. We define a different phase partition to the one in Definition 1.8, which now depends on the online algorithm Fifo. The first phase $P_{\text{Fifo},1}$ ends after the first page fault that is caused by Fifo. Every subsequent phase has a length that is such that Fifo causes exactly k page faults in it; the phase ends right after the kth page fault occurred. Formally, phase $P_{\text{Fifo},i}$ ends immediately after Fifo made $(i-1)k+1$ page faults. The last phase may be shorter. Use this phase partition to show that Fifo is k-competitive.

Does your proof show that Fifo is strictly k-competitive?

Exercise 1.10. Fifo experiences a phenomenon that is known as *Bélády's anomaly*, which states that there are instances on which Fifo causes more page faults if it has a larger cache. Find such an instance.

Hint. It suffices to consider two cache sizes 3 and 4 and a total number of nine pages.

1.4 A Lower Bound for Paging

We now know that there are k-competitive online algorithms for paging. But are these algorithms strongly competitive? In other words, is this the best we can hope for or are there online algorithms which outperform Fifo, Fwf, and Lru?

The answer is that there is nothing better from a worst-case point of view. This means that, for every online algorithm Alg, there are infinitely many instances of paging for which Alg's cost is at least k times larger than the optimal cost. To

model such hard instances, we think of an *adversary* that constructs a hard instance I while knowing the online algorithm Alg we want to analyze. In a way, Alg and the adversary are two players in a game and they have directly opposing goals (in Section 2.4, we will have a closer look at this point of view). As paging is an online minimization problem, this means that the adversary tries to make Alg have a cost that is as large as possible compared to the cost of an optimal solution $\textsc{Opt}(I)$ for I and thereby to maximize the competitive ratio of Alg. If not stated otherwise, we will assume that we are dealing with demand paging algorithms as in Definition 1.7; with the considerations above (in particular, Exercises 1.5 and 1.6), we know that this does not cause any restriction.

Theorem 1.5. *No online algorithm for paging is better than k-competitive.*

Proof. Let $m = k+1$, that is, we only require that there are pages $p_1, p_2, \ldots, p_{k+1}$ in total; let n be some multiple of k. Recall that the cache is initialized as $(p_1, p_2, \ldots, p_k)$, and we consider an arbitrary online algorithm Alg for paging. Obviously, there is exactly one page, at any given time step, that is not in the cache of Alg. The whole idea is that the adversary always requests exactly this page to obtain an instance I of length n. Since it knows Alg, it can always foresee which page will be replaced by Alg if a page fault occurs.

```
output "p_{k+1}";                                   // Inevitable page fault
i := 1;
while i ≤ n − 1 do
    p := the page that is currently not in the cache of Alg;
    output "p";
    i := i + 1;
end
```

Algorithm 1.2. Adversary for any paging algorithm.

More formally, consider Algorithm 1.2, which creates the instance I of length n for Alg by following this strategy. It is easy to see that this instance causes a page fault for Alg in every time step, and thus a total cost of n. However, this is not sufficient to prove the claim. The competitive ratio compares this value to what could have been achieved on I if it had been known; in other words, we need to study the optimal cost on this instance as well.

To do so, we again divide the input into distinct consecutive phases. This time, one phase consists of exactly k time steps, that is, Alg makes exactly k page faults within a phase (recall that n is a multiple of k). If we can show that Opt causes at most one page fault in every phase, we are done. Consider the first phase P_1. In time step T_1, every algorithm causes a page fault as the requested page p_{k+1} is not in the cache by definition. Opt can now choose one of the pages $p_1, p_2, \ldots, p_k$ to be removed to load p_{k+1}. P_1 consists of exactly $k-1$ more time steps, so at most

$k-1$ more distinct pages are requested. Therefore, there is at least one page p' among $p_1, p_2, \ldots, p_k$ that is not requested during this phase, and OPT chooses p' to be removed in time step T_1. There may be more than one such page, in which case OPT chooses the page whose first request is the latest among all such pages (OPT therefore implements the offline strategy LFD).

We can use the same argument for any other phase P_i with $2 \leq i \leq N$. The only difference is that OPT does not surely cause a page fault in the first time step $T_{(i-1)k+1}$ of this phase, but it may cause a page fault later or even not at all. However, whenever a page fault occurs, by the same reasoning as for phase P_1, there must be some page that is not requested anymore during phase P_i and that may therefore be safely removed from the cache.

Finally, we need to deal with the additive constant α from Definition 1.6. If the number of page faults caused by OPT is constant, ALG is not competitive. On the other hand, if the number of page faults increases with n, Theorem 1.2 implies that ALG cannot be better than k-competitive. □

We see that the adversary can guarantee that any online algorithm causes a page fault in every time step. Thus, with respect to the pure cost, all online algorithms are equally bad. Then again, for instance, FIFO outperforms LIFO when these strategies are analyzed according to their competitive ratios. This is due to the fact that FIFO keeps pages it just loaded in the cache for a longer time than LIFO.

Theorem 1.6. LIFO *is not competitive for paging.*

Proof. To prove the claim, we show that, for every n, there is an instance I of paging of length n such that $\text{cost}(\text{LIFO}(I))/\text{cost}(\text{OPT}(I))$ grows proportionally with n. To this end, we give an instance of length n that always requests the same two pages; again, it suffices to choose $m = k+1$. The adversary again first requests p_{k+1}, and since all pages $p_1, p_2, \ldots, p_k$ are in the cache at the beginning, LIFO removes some fixed page from the cache, say p_i. Since the adversary knows that LIFO chooses p_i, it requests it in time step T_2 and LIFO removes p_{k+1}, which is now the page that was last loaded into the cache. Then, LIFO must remove p_i in time step T_3 when the adversary again requests p_{k+1}. The adversary continues in this fashion, that is, I is given by

$$(p_{k+1}, p_i, p_{k+1}, p_i, \ldots) .$$

In every time step, LIFO causes a page fault while there is an optimal solution $\text{OPT}(I)$ that removes a page p_j with $j \neq i$ in time step T_1 and has cost 1 overall, because it has p_i and p_{k+1} in its cache from that point on. □

So we see that there is a significant difference between FIFO and LIFO. This is not very surprising; intuitively it seems like a bad idea to immediately remove a page from the cache that was just loaded into it. What about LFU? Here, an intuitive point of view might suggest more success; we learn from what happened so far, namely, we replace a page that was in some sense the least valuable up to now. Unfortunately, this strategy is not much better in the worst case.

Theorem 1.7. LFU *is not competitive for paging.*

Proof. The proof is only slightly more complex than the one for LIFO from Theorem 1.6. For every n', consider the instance I given by

$$(\underbrace{p_1, p_1, \ldots, p_1}_{n' \text{ requests}}, \underbrace{p_2, p_2, \ldots, p_2}_{n' \text{ requests}}, \ldots, \underbrace{p_{k-1}, p_{k-1}, \ldots, p_{k-1}}_{n' \text{ requests}}, \underbrace{p_{k+1}, p_k, \ldots, p_{k+1}, p_k}_{2(n'-1) \text{ requests}})$$

of length $n := (k-1)n' + 2(n'-1)$. In the first $(k-1)n'$ time steps, no online algorithm causes a page fault, and after that, all pages in the cache have been requested n' times except for p_k. Thus, when p_{k+1} is requested in time step $T_{(k-1)n'+1}$, LFU removes p_k, which is the page in the cache that was used least frequently. Next, the adversary requests p_k, and this is iterated until both pages p_k and p_{k+1} have been requested exactly $n' - 1$ times each. Clearly, LFU makes a page fault in each of the last $2(n' - 1)$ time steps. On the other hand, an optimal solution $\text{OPT}(I)$ simply removes a page p_j with $j \neq k$ in time step $T_{(k-1)n'+1}$ and causes no more page faults. Since

$$n' = \frac{n+2}{k+1} ,$$

the competitive ratio of LFU can be bounded from below by

$$2(n' - 1) = \frac{2(n-k+1)}{k+1} ,$$

which is a linear function in n. □

According to Definition 1.6, neither LIFO nor LFU are competitive; however, if we take a closer look, the lower bound on the competitive ratio of LIFO is stronger than that of LFU by a factor which tends to $(k+1)/2$ with growing n.

Exercise 1.11. We have defined LFU such that it keeps track of all m pages and removes one of the pages that was least frequently used in the sum. Suppose the algorithm forgets the number of accesses of pages that are not in the cache and initializes it with 1 for every page that is loaded. Does this give a stronger lower bound?

Exercise 1.12. Now consider the online algorithm MAX that always replaces the page in its cache that has the largest index, that is, for any cache content $p_{i_1}, p_{i_2}, \ldots, p_{i_k}$, the page p_j with $j = \max\{i_1, i_2, \ldots, i_k\}$ is removed in case of a page fault. Is MAX competitive? If so, prove an upper bound on the competitive ratio that is as good as possible. If not, show that MAX has no constant competitive ratio. How about an online algorithm MIN that is defined accordingly?

Exercise 1.13. Consider the following online algorithm WALK that replaces the page at position $1 + ((i-1) \bmod k)$ in the cache on the ith page fault. Less formally, it replaces the pages in the order they are stored in the cache, continuing with the first cell if it used the kth one for the preceding page fault. Argue why WALK is k-competitive.

Exercise 1.14. A phenomenon that is observed in practical settings is *locality of reference*, that is, that pages are likely to be requested consecutively if they are located next to each other. We want to make use of this fact and define an algorithm Local that always removes the page whose page index is farthest away from the requested one on a page fault (ties are broken arbitrarily). Is Local competitive?

1.5 Marking Algorithms

Now that we have established a lower bound on any paging algorithm and a matching upper bound for some specific strategies, we want to focus on a more general concept, the so-called *marking algorithms*. This class of algorithms plays an important role in the context of randomized computation for paging, which we will study in the following chapter.

A marking algorithm works in phases and *marks* pages that were already requested; it only removes pages that are not marked. If all pages in the cache are marked and a page fault occurs, the current phase ends, and a new one starts by first unmarking all pages in the cache. Before processing the first request, all pages get marked such that the first request that causes a page fault starts a new phase. The pseudo-code of a marking algorithm is shown in Algorithm 1.3.

```
mark all pages in the cache;                          // First page fault starts new phase
for every request x do
    if x is in the cache
        if x is unmarked
            mark x;
        output "0";
    else
        if there is no unmarked page
            unmark all pages in the cache;                    // Start new phase
        p := somehow chosen page among all unmarked cached pages;
        remove p and insert x at the old position of p;
        mark x;
        output "p";
end
```

Algorithm 1.3. General scheme of a marking algorithm for paging.

We now show that this general concept allows for strongly competitive online algorithms by using the concept of phases as in the proof of Theorem 1.4. More precisely, we will prove that the phases of marking algorithms correspond to the phases of a k-phase partition from Definition 1.8. Except possibly the last one, a phase of a marking algorithm consists of a maximum-length sequence of requests for k different pages. This makes it very easy for us to argue why such an algorithm makes at most k page faults in one phase.

Theorem 1.8. *Every marking algorithm is strictly k-competitive for paging.*

Proof. Let MARK be a fixed marking algorithm; let I denote the given input and consider its k-phase partition into N phases $P_1, P_2, \ldots, P_N$ according to Definition 1.8. By the same argument as in the proof of Theorem 1.4, we conclude that any optimal algorithm OPT makes at least N page faults in total on I.

What remains to be done is to show that MARK makes at most k page faults in one fixed phase P_i with $1 \leq i \leq N$. We denote the $\overline{N}$ phases explicitly defined by MARK by $P_{\text{MARK},1}, P_{\text{MARK},2}, \ldots, P_{\text{MARK},\overline{N}}$ and claim that both $N = \overline{N}$ and $P_j = P_{\text{MARK},j}$, for all j with $1 \leq j \leq N$. Since MARK makes at most k page faults in one phase $P_{\text{MARK},i}$ (clearly, there cannot be more page faults than pages marked at the end of $P_{\text{MARK},i}$), the claim follows. We first observe that both P_1 and $P_{\text{MARK},1}$ start with the first request that causes a page fault. Every phase P_i except the last one is by definition a maximum-length sequence of k distinct requests. Every requested page gets marked by MARK after being requested. If k distinct pages were requested, all pages in MARK's cache are marked. With the $(k+1)$th distinct page p' being requested since the beginning of P_i, a new phase P_{i+1} starts. In this time step, MARK also starts a new phase $P_{\text{MARK},i+1}$, as there is no unmarked page left in its cache to replace with p'. Thus, the phases P_i and $P_{\text{MARK},i}$ coincide. As a consequence, MARK makes at most k page faults per phase and the claim follows. □

It can be shown that some of the online algorithms we discussed are in fact marking algorithms, although they do not explicitly mark pages.

Theorem 1.9. LRU *is a marking algorithm.*

Proof. To prove the claim means to show that LRU never removes a page that is currently marked by some marking algorithm. For a contradiction, suppose that LRU is not a marking algorithm. Then there is some instance I such that LRU removes a page that is marked. Let p be the page for which this happens for the first time, and denote the corresponding time step by T_j with $1 \leq j \leq n$ during some phase P_i with $1 \leq i \leq N$. Since p is marked, it must have been requested before during P_i, say in time step $T_{j'}$ with $j' < j$. After that, p was most recently used; thus, if LRU removes p in time step T_j, there must have been k distinct requests following time step $T_{j'}$ that are all different from p; the first $k-1$ cause p to become least recently used afterwards, and on the kth such request p is removed according to LRU. As a consequence, P_i consists of at least $k+1$ different requests, which is a direct contradiction to the definition of a k-phase partition. □

Exercise 1.15. Prove that FWF is also a marking algorithm.

Exercise 1.16. How about LIFO and FIFO? Justify your answer.

1.6 Refined Competitive Analysis

Competitive analysis, as we studied it so far, is a pure worst-case measurement; it formalizes a framework in which, for any given online algorithm, the worst possible situation is met. Actually, it might be quite a realistic setting to assume some additional knowledge about the input. There are numerous attempts to get a more realistic model for such situations. In Chapter 3, we will introduce a very general method to deal with additional information and its quantification. At this point, we only want to pick two more specific approaches to give more power to online algorithms for paging.

1.6.1 Lookahead

The straightforward approach to give an online algorithm an advantage compared to the classical model is to enable it to have some *lookahead*, that is, to allow it to look into the future for ℓ time steps. It might be surprising, but this knowledge does not help to improve the competitive ratio. Consider paging with lookahead ℓ; this means that, in any time step, an online algorithm ALG_ℓ sees the current request together with the subsequent ℓ requests. Since the adversary we use to model hard instances knows ALG_ℓ, it surely knows ℓ and may therefore proceed as follows.

Each request is repeated ℓ times such that ALG is still somewhat "in the dark" in the time step where it has to replace a page. We again only need to consider the case where $m = k + 1$. The first $\ell + 1$ requests all ask for the only page that is not in the cache initially, that is, p_{k+1}. In the first time step, ALG_ℓ must replace a page, but it cannot see which page is requested in time step $T_{\ell+2}$. Therefore, the additional knowledge is completely useless, and the adversary can simply request p_i which ALG_ℓ replaces in time step T_1. When ALG_ℓ then must find a page to replace with p_i, it only knows the prefix

$$(p_{k+1}, \underbrace{p_{k+1}, \ldots, p_{k+1}}_{\ell \text{ requests}}, p_i, \underbrace{p_i, \ldots, p_i}_{\ell \text{ requests}}, \ldots)$$

of the input I, which again does not help.

Continuing in this fashion, the adversary can ensure that ALG_ℓ causes a page fault every $\ell + 1$ time steps. With the same reasoning as in the proof of Theorem 1.5, $\text{OPT}(I)$ causes at most one page fault every $k(\ell + 1)$ time steps. For such inputs of length n, ALG_ℓ causes $n/(\ell+1)$ page faults, while $\text{OPT}(I)$ causes at most $n/(k(\ell+1))$. As a result, the competitive ratio of ALG_ℓ has a lower bound of k.

Theorem 1.10. *No online algorithm with lookahead ℓ for paging is better than k-competitive.* □

1.6.2 Resource Augmentation

Another principle to improve the chances of an online algorithm against the adversary is called *resource augmentation*. Here, we allow the online algorithm to use more

resources than the optimal offline algorithm. What this means in detail depends on the problem at hand. For paging, OPT is only allowed to use a cache size of $h \leq k$ for the same input; this problem is called the (h, k)-*paging problem*. We assume that OPT's cache is initialized with the first h pages $p_1, p_2, \ldots, p_h$. This problem is a generalization of paging as we studied it until now, which can just be viewed as (k, k)-paging.

Theorem 1.11. *Every marking algorithm is $k/(k-h+1)$-competitive for (h, k)-paging.*

Proof. The proof follows from an easy modification of the proof of Theorem 1.4. Let MARK be any marking algorithm. Once more, consider the k-phase partition of a given input I. For any given phase P_i with $1 \leq i \leq N$, we know that MARK causes at most k page faults as shown in the proof of Theorem 1.8.

To bound the number of page faults that OPT(I) causes, let us again shift the phases by one to obtain a new partition $P'_1, P'_2, \ldots, P'_N$; again, P'_N may be empty. Let p be the first request during the phase P_i. Then, OPT's cache contains $h-1$ pages at the beginning of P'_i that are different from p, and since, for any i with $1 \leq i \leq N-1$, k distinct pages (that are all different from p) are requested within P'_i, OPT(I) has to make $k-(h-1)$ page faults.

In the first $N-1$ phases, MARK causes $(N-1)k$ page faults whereas OPT(I) causes $(N-1)(k-(h-1))$ page faults. In P_N, MARK causes at most k page faults; on the other hand, OPT(I) causes one additional page fault with the first request of P_1. A competitive ratio of at most $k/(k-(h-1))$ follows, where we set the additive constant α from Definition 1.6 to $k-1$. □

Observe that Theorem 1.11 does not claim *strict* competitiveness, whereas we know from Theorem 1.8 that the competitive ratio is indeed strict for $h = k$. This is due to the fact that we know that OPT(I) has to make one page fault right before P'_1, which we can then assign to the at most k page faults a marking algorithm causes in the last phase P_N. Obviously, an analogous argument for $h < k$ does not work (we would have to assign $k-(h-1)$ page faults of the optimal solution to the at most k page faults of the marking algorithm).

We will briefly revisit resource augmentation for the k-server problem in Chapter 4, and when studying the online knapsack problem in Chapter 6.

Exercise 1.17. What happens if we assume $k < h$ instead of $h \leq k$?

Exercise 1.18. Show that the bound of Theorem 1.11 is tight, that is, that no online algorithm is better than $k/(k-h+1)$-competitive for (h, k)-paging.

1.7 Historical and Bibliographical Notes

As already mentioned, Turing machines were introduced by Turing [138] in 1936. Two major subjects in the kernel of theoretical computer science are *computability*

theory and *complexity theory*, which are both basically built around this model. Of course, our introduction was extremely short and incomplete. There is a very rich literature on computability and computational complexity, for instance, the textbooks written by Arora and Barak [11], Hopcroft et al. [78], Hromkovič [79, 80], Papadimitriou [122], and Sipser [130]. As for approximation algorithms, both Hromkovič [79] and Vazirani [139] give very good introductions.

Today we know that testing whether a given number is prime can be done in polynomial time [1].

The algorithm Kruskal from Exercise 1.3 is known as *Kruskal's algorithm* and named after Kruskal, who first published it in 1956 [110]. In Chapter 8, we will introduce an online version of the MSTP and exploit Kruskal's optimality.

The decision version of the knapsack problem (more precisely, a variant that is called the *subset sum problem*) is among "Karp's 21 $\mathcal{NP}$-complete problems" [93]. In other words, it was one of the first problems ever to be proven to be $\mathcal{NP}$-complete. For the optimization version, there is a pseudo-polynomial-time algorithm that is based on dynamic programming [79, 139]. Ibarra and Kim [84] used this approach to design an FPTAS. Therefore, the offline version of the problem is one of the easier $\mathcal{NP}$-hard problems. More details about the knapsack problem and its variants are, for instance, given in the textbook by Kellerer et al. [94]. The Christofides algorithm for the TSP was introduced in 1976 by Christofides [44].

Competitive analysis was introduced in 1985 by Sleator and Tarjan [131]. The lower bound of k for paging was also proven in this paper (even the more general result from Exercise 1.18 that makes use of resource augmentation); the authors also showed that Lru is k-competitive. Bélády proved that Lfd (which he called Min, not to be confused with the online algorithm from Exercise 1.12) is an optimal offline algorithm for paging [18]. The terms *competitive* and *strongly competitive* were first used in this context by Karlin et al. [92].

"Online Computation and Competitive Analysis" from Borodin and El-Yaniv [34] is certainly the standard textbook on online algorithms and gives both a broad and deep introduction to the topic. Additionally, there are many excellent surveys on online algorithms by, for instance, Albers [4, 6], Fiat and Woeginger [63], and Irani and Karlin [86].

Although Fifo, Fwf, and Lru achieve the same competitive ratio from our theoretical point of view, it has been pointed out that this does not reflect what is observed in practice [34, 47]. The criticism has been made that the idea of competitive analysis is not sufficiently fine-grained as it is, in general, too pessimistic [6, 20, 38, 54, 62, 86, 107]; in other words, many algorithms that perform very well in practice are considered to be very weak with respect to competitive analysis. A more detailed survey of the different refinements of competitive analysis that were proposed since its introduction is given by Fiat and Woeginger [62] and in Chapter 3 of the dissertation of Dorrigiv [56].

Bélády's anomaly (see Exercise 1.10) was first observed by Bélády, Nelson, and Shedler [19].

Due to the fact that a usual lookahead does not help, Albers followed a different approach by introducing and using a so-called *strong* lookahead that enables the algorithm to see ℓ pairwise distinct future requests [5]. This more powerful knowledge about the future does indeed help for paging. Let $\ell \leq k-2$; then there is an online algorithm (basically a variant of the abovementioned strategy LRU) with strong lookahead ℓ that is $(k-\ell)$-competitive, and this bound is tight. The concept of resource augmentation was introduced by Kalyanasundaram and Pruhs [89, 90] (though implicitly used earlier [50, 131]), and since then used for a number of problems [50, 123]. Iwama and Zhang [88] and Han and Makino [75] used this relaxation of pure competitive analysis to study online versions of the knapsack problem; for this problem, we combine resource augmentation and computing with advice in Chapter 6.