3 Advice Complexity

In this chapter, we introduce another important class of online algorithms. So far, we have considered deterministic and randomized approaches. For paging and the ski rental problem, randomization enables us to construct more powerful algorithms. More specifically, we are able to design randomized online algorithms that are a lot better in terms of their expected competitive ratio; for paging, we even have an exponential improvement. The third model, *online algorithms with advice*, is even more powerful. We introduce this concept and, as in the two preceding chapters, use paging as an example to illustrate the basic ideas. The main motivation to study this model is mostly a theoretical one; it gives us a very intuitive formalization of the notion of "hardness" in the context of online computation. The idea is to measure the amount of information about the yet unknown parts of the input that an online algorithm needs to know in order to achieve some particular output quality. To have a formal framework, we introduce an *oracle* which sees the whole input in advance and may encode any binary information about this input onto a so-called *advice tape*. An online algorithm can then use this tape as a resource during computation. We then ask for the number of bits of advice the online algorithm needs to be, say, c-competitive, for some specific constant c. The number of advice bits used is given by a function of the input length n, similar to the number of random bits, which we studied in the preceding chapter; this number is called the *advice complexity* of the algorithm.

We first introduce the model of advice formally. Next, to be able to prove upper bounds on the advice complexity, we describe the concept of *self-delimiting strings*; these allow an online algorithm to read a number of bits from the advice tape in a situation where it does not have any information about the length of this string. We continue by explaining how to prove lower bounds on the advice complexity; for this reason, we introduce so-called *partition trees*. After that, we study the advice complexity of paging (that is, the advice complexity of online algorithms for paging) both for obtaining optimal solutions and for using only a small number of advice bits.

D. Komm, *An Introduction to Online Computation*,
Texts in Theoretical Computer Science. An EATCS Series,
DOI 10.1007/978-3-319-42749-2_3

Finally, we make some interesting observations on the connection between advice and randomization.

3.1 Introduction

Before we designed the randomized online algorithm RSki for the ski rental problem, we saw that deterministic online algorithms are almost twice as bad as an optimal offline algorithm. We want to have a closer look at this fact. More specifically, we want to answer the question

Why are they twice as bad?

To phrase the question even more exactly, we ask

What are they missing?

It is tempting to give a simple answer. What they miss is the complete input. Sure, if we had a complete and accurate weather forecast, we could simply compute an optimal solution for any instance of the ski rental problem; count the number of days with good weather, and check whether it is at least k. If it is, we buy the skis, otherwise we rent them. But is it really necessary to know the whole input to be optimal? If we think about it, all that we need is to be able to make a simple "yes"/"no" decision, namely, whether to buy the skis on the first day with good weather or to rent them again and again. So what we are missing is basically the smallest amount of information there is; one single bit.

As discussed in Chapter 1, the competitive ratio tells us how much we pay if we work on a specific online problem. The *advice complexity*, on the other hand, tells us what we pay for. For the ski rental problem, every deterministic online algorithm is almost 100% worse than the optimal solution in the worst case, and the "why" can be answered with "because we don't know this single bit." However, of course, in general the question "why" or "what" cannot be answered this easily; for instance, a single bit of information will probably not enable us to get an optimal solution for the paging problem (but surprisingly, later in this chapter, we will see that a constant amount of additional information does help quite a lot). As a matter of fact, different online problems behave very differently when we investigate them with respect to the information that is necessary to obtain some good solution quality. As mentioned above, we can always say that knowing the whole input in advance helps to create an optimal solution, but for some problems we may be able to compress some critical property of the input that already enables us to improve a lot over deterministic or randomized strategies. In a way, we ask about the *information content* of the problems, that is, the information that is hidden in the instances and that needs to be extracted; here, the advice complexity is a powerful tool.

To be able to measure this amount of information, we use a model where an *oracle* is introduced that knows the whole input I of a given online problem in advance. This oracle can write binary information about I on a so-called *advice tape* that

can afterwards be used by an online algorithm that works on I. Informally, we can describe this model as follows.

- We do not simply design online algorithms, but an online algorithm ALG is always created together with an oracle. We call ALG an *online algorithm with advice*.
- For every input, the oracle writes some so-called *advice bits* on the advice tape.
- The adversary knows both ALG and the oracle; in particular, it knows which advice the oracle writes on the tape, given a specific input.

In the classical model, the adversary inspects ALG and then constructs an input that causes ALG to perform as badly as possible. Now, there is a third party, which is essentially an all-knowing counselor working for the algorithm. Note that the third bullet point suggests that, in the model of computing with advice, we have an extremely powerful adversary. However, if we take a closer look, this is not the case; it is sufficient if the adversary merely knows ALG, and, as a consequence, an upper bound $b(n)$ on the number of advice bits that the algorithm reads for a given input length n. For any n and $b(n)$, the adversary can simply simulate ALG on every possible advice string ϕ of length $b(n)$ and therefore find the best advice. It can then choose an instance I' of length n of the given online problem Π such that

$$I' := \arg\max_I \left\{ \min_\phi \left\{ \frac{\text{gain}(\text{OPT}(I))}{\text{gain}(\text{ALG}^\phi(I))} \right\} \right\}$$

if Π is a maximization problem, or

$$I' := \arg\max_I \left\{ \min_\phi \left\{ \frac{\text{cost}(\text{ALG}^\phi(I))}{\text{cost}(\text{OPT}(I))} \right\} \right\}$$

if Π is a minimization problem. If possible, the adversary will additionally try to make sure that, for the set of instances it constructs in the above way, the optimal cost (gain, respectively) increases unboundedly with the input length, such that a lower bound on the non-strict competitive ratio is obtained.

The above formula reminds us of the minimax theorem from Section 2.4. However, both the adversary and the algorithm pick pure strategies. The important thing is that no matter which strategy the adversary decides to use, ALG will *always* pick a best of its strategies as a response. Now let us describe the steps that are made in the model of computing with advice.

Step 1. The adversary constructs an input I of length n such that the competitive ratio of ALG using the advice tape is maximized; the adversary knows the number $b(n)$ of advice bits ALG reads at most.

Step 2. After that, the oracle inspects I and writes an advice string ϕ on the advice tape which depends on I.

Step 3. ALG reads the input I and computes an output O while using the advice tape; ALG reads at most a prefix of length $b(n)$ from the tape.

Step 4. If ALG obtains a competitive ratio of at most c, we say that ALG is c-competitive with *advice complexity* $b(n)$ or that ALG needs at most $b(n)$ advice bits to be c-competitive.

A crucial property of this model is that the advice tape has infinite length. This is important to prevent any situations in which information may be encoded into the length of the advice. On first sight, it seems redundant since the oracle and ALG are designed in such a way that the online algorithm never uses more than $b(n)$ advice bits in total anyway; but if we take a closer look, without this property, we could design an online algorithm with advice and an oracle that work as follows. Suppose that, for some input of length n, the oracle writes $b(n) = n$ advice bits on the tape. At the beginning, the online algorithm reads all the $b(n)$ bits until the end, and thus knows the length of the input. In a way, it gets this knowledge for free since it is only implicitly communicated by the advice length but not by its content. Of course, it is perfectly fine if the oracle writes the input length on the advice tape explicitly. The difference is that, in this case, this information is part of the advice and therefore accounted for.

Moreover, the tape is accessed sequentially (similar to the random tape of a randomized algorithm).

Example 3.1. Summarizing what we just learned, we can state that there is an optimal online algorithm with advice for the ski rental problem which uses 1 bit of advice. An oracle first reads the whole input and computes whether there are more than k days with good weather. If there are, it writes a 1 at the first position of the advice tape; else it writes a 0 at this position. In the first time step, the corresponding algorithm reads the first bit. If it is 1, the algorithm buys the skis at the first day with good weather; otherwise it rents them at every such day. Clearly, this algorithm always has the smallest cost possible. ♢

Now we are going to formally define online algorithms with advice. Following the preceding discussion, it seems to make sense to have a definition analogous to Definition 2.1 for randomized online algorithms.

Definition 3.1 (Online Algorithm with Advice). 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 *with advice* for Π computes the output $\text{ALG}^{\phi}(I) = (y_1, y_2, \ldots, y_n)$, where y_i depends on $\phi, x_1, x_2, \ldots, x_i$ and $y_1, y_2, \ldots, y_{i-1}$; ϕ denotes a *binary advice string*.

The essential difference between randomized online computation and online computation with advice comes into play when we define the competitive ratio for online algorithms with advice.

Definition 3.2 (Competitive Ratio with Advice). Let Π be an online problem, let ALG be a consistent online algorithm with advice for Π, and let OPT be an optimal offline algorithm for Π. For $c \geq 1$, ALG is *c-competitive with advice complexity* $b(n)$ for Π if there is a non-negative constant α such that, for every instance $I \in \mathcal{I}$, there is an advice string ϕ such that

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

if Π is a maximization problem, or

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

if Π is a minimization problem, and ALG uses at most the first $b(n)$ bits of ϕ. If the above inequality holds for $\alpha = 0$, ALG is called *strictly c-competitive with advice complexity* $b(n)$; ALG is called *optimal with advice complexity* $b(n)$ if it is strictly 1-competitive with advice complexity $b(n)$. The *competitive ratio* of an online algorithm ALG with advice complexity $b(n)$ is defined as

$$c_{\text{ALG}} := \inf\{c \geq 1 \mid \text{ALG is } c\text{-competitive for } \Pi \text{ with advice complexity } b(n)\} .$$

For such an online algorithm ALG, we thus require that, for every input, there is some advice string that allows ALG to obtain the given competitive ratio; we do not care whether this particular advice string is extremely bad for all other instances. This is the crucial difference when comparing this model to that of randomized online computation. The oracle deduces the advice string from the concrete input and does not use any randomness in the process. Still, the models share a common property, namely a binary tape that allows us to treat them as a collection of deterministic algorithms. We can thus formulate an analogous statement to Observation 2.2 (see also Exercise 2.2).

Observation 3.1. *Every online algorithm* ALG *with advice that uses at most* $b(n)$ *advice bits for inputs of length* n *can be viewed as a set* $\text{strat}(\text{ALG}, n) = \{A_1, A_2, \ldots, A_{2^{b(n)}}\}$ *of* $2^{b(n)}$ *deterministic online algorithms on inputs of length* n*, from which* ALG *always chooses one with the best performance for the given instance.*

For every given randomized online algorithm RAND, we can design an online algorithm ALG with advice that uses its advice tape the same way RAND uses its random tape. Accompanying ALG, we create an oracle that, for every instance, writes a "best" string on the advice tape; following this idea, we can state the following observation.

Observation 3.2. *For any online problem* Π*, the following two implications hold.*

(i) If there is a randomized online algorithm for Π *that is c-competitive in expectation and uses at most* $b(n)$ *random bits, then there is also an online algorithm with advice for* Π *that is c-competitive and that uses at most* $b(n)$ *advice bits.*

(ii) Conversely, if there is provably no online algorithm with advice that is c-competitive while using at most $b(n)$ *advice bits for* Π*, then there is also no randomized online algorithm for* Π *that is c-competitive in expectation and that uses at most* $b(n)$ *random bits.*

In Section 3.5, we will revisit the relation between advice and randomization and show some non-trivial connections.

3.2 Self-Delimiting Encoding of Strings

In this section, we focus on how to concretely encode information onto the advice tape, and especially on one particular problem that arises when there are multiple pieces of information that need to be delimited when using a binary alphabet.

Let d be some natural number. We know that we can encode 2^d different numbers in binary with d bits. To encode an arbitrary natural number m in binary, we need $\lceil \log_2(m+1) \rceil$ bits. If m is always at least 1, we only need $\lceil \log_2 m \rceil$ bits; this can be done by writing $m-1$ on the advice tape in binary. In the following considerations, we always assume that this is true for m.

It gets more difficult if we think about the special kind of resource we are dealing with in this setting. In particular, as already discussed in the previous section, we are facing the fact that the advice tape we are using has an infinite length; behind the actual advice, there is an infinite undefined suffix. The alphabet that the oracle uses to write on the advice tape is binary, and thus we do not have any *delimiter* to mark where the encoding of some binary substring (encoding, for instance, the length of the input) ends. Moreover, in general, we cannot use any special sequence of bits like, for instance, "111" as a delimiter since the same sequence might also be part of the advice (see Exercise 3.3).

What can we do about this? To answer this question, *self-delimiting encodings* come into play. The idea is to augment the advice with some *control bits* that allow the algorithm to decode the advice itself. Again, let m be a positive number that we want to encode. The idea is to tell the algorithm how many bits (from the infinite advice tape) belong to the binary representation of $m-1$.

First, we need at most $\lceil \log_2 m \rceil$ bits to encode $m-1$ on the tape. Then, we can use an additional $\lceil \log_2 m \rceil$ bits to tell the algorithm which bits belong to the string of length $\lceil \log_2 m \rceil$ as follows. We write the binary representation of $m-1$ on odd positions of the advice tape. On even positions, we write a 1 if the next bit still belongs to the binary representation of $m-1$, and a 0 otherwise. Thus, if $b_1 b_2 \dots b_{\lceil \log_2 m \rceil}$ is the binary representation of $m-1$, the content of the advice tape starts with

$$b_1\, 1\, b_2\, 1 \dots b_{\lceil \log_2 m \rceil - 1}\, 1\, b_{\lceil \log_2 m \rceil}\, 0\ .$$

As a consequence, we need to use $2\lceil \log_2 m \rceil$ advice bits instead of $\lceil \log_2 m \rceil$ bits; we call this a *self-delimiting encoding* of m.

With another simple idea, we can improve this approach and use a smaller number of bits. For small values of m, we use the encoding

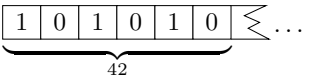
If m is at least 3, at the beginning of the advice tape, we tell the algorithm how many bits are used to encode $m-1$, that is, we write the number $\lceil \log_2 m \rceil$. This can be done using at most $\lceil \log_2(\lceil \log_2 m \rceil) \rceil$ additional bits since m is at least 3 and thus $\lceil \log_2 m \rceil$ is at least 2; hence, we can write $\lceil \log_2 m \rceil - 1$ on the tape. Now we are left with marking where these first $\lceil \log_2(\lceil \log_2 m \rceil) \rceil$ bits end and the binary representation of $m-1$ starts. To this end, we can use exactly the same idea as above and thus we need at most

$$2\lceil \log_2(\lceil \log_2 m \rceil) \rceil + \lceil \log_2 m \rceil$$

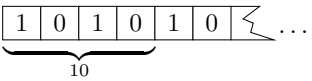
bits to encode m in a self-delimiting way. Note that, this way, the string always starts with a 1; hence, the cases where m is 1 or 2 can be distinguished from the case that m is at least 3 without any further information. The algorithm can now start reading until it encounters a 0 at an even position. Then, it computes the number of bits it needs to read afterwards to obtain m. We trade the multiplicative constant 2 for an additive term that is asymptotically smaller. Sample encodings are shown in the following table.

m	$m-1$	$\lceil \log_2 m \rceil - 1$	$\lceil \log_2(\lceil \log_2 m \rceil) \rceil$	self-delimiting string
1	0	–	–	0 0
2	1	–	–	0 1
3	2	1	1	1 0 1 0
4	3	1	1	1 0 1 1
5	4	2	2	1 1 0 0 1 0 0
6	5	2	2	1 1 0 0 1 0 1
7	6	2	2	1 1 0 0 1 1 0
8	7	2	2	1 1 0 0 1 1 1
9	8	3	2	1 1 1 0 1 0 0 0
10	9	3	2	1 1 1 0 1 0 0 1
⋮	⋮	⋮	⋮	⋮
256	255	7	3	1 1 1 1 1 0 1 1 1 1 1 1 1 1
				$2\lceil \log_2(\lceil \log_2 m \rceil) \rceil$ bits; $\lceil \log_2 m \rceil$ bits
⋮	⋮	⋮	⋮	⋮

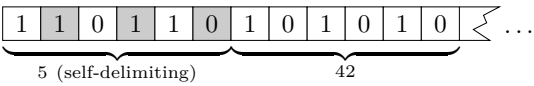
Example 3.2. Suppose we want to encode the number 43 onto the advice tape; moreover, it is known that the encoded number is not 0. Thus, we write 42 on the tape. If we simply encode it in binary, we get a prefix



of the advice string; but if we only see this string, we cannot at all decode it in a unique way. It could, for instance, also be interpreted as



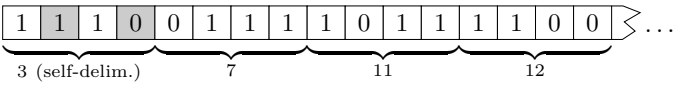
by an online algorithm assuming that the last two depicted bits already belong to the undefined part of the tape. Now let us make use of self-delimiting encodings. We realize that we need $\lceil \log_2 42 \rceil = 6$ bits to encode 42 in binary. Therefore, we first write the number 5 in a self-delimiting way on the advice tape (using $2\lceil \log_2 5 \rceil = 6$ additional bits) followed by the number 42 and obtain a prefix



of the tape content. Now an online algorithm with advice is able to decode the advice without further knowledge. ♢

At times, we want to encode multiple numbers, and we know that they have some common upper bound.

Example 3.3. Suppose we want to encode the numbers 8, 12, and 13. Furthermore, we know that there are three numbers in total and none of them are 0 or larger than 16, that is, they can all be written in binary using at most four bits. We can encode them as



and again, an algorithm is able to unambiguously decode the string. ♢

We will use self-delimiting strings on several occasions. One application is immediate, namely this enables us to let some online algorithm know the input length n. Intuitively, this already might be a great advantage in some applications. The input length is never 0. Furthermore, we ignore the special cases that n is 1 or 2 to keep the formulas simple. Therefore, to encode n in a self-delimiting way, we usually need an additional

$$2\lceil \log_2(\lceil \log_2 n \rceil) \rceil + \lceil \log_2 n \rceil$$

advice bits.

Exercise 3.1. Describe alternative ways to obtain self-delimiting encodings of a natural number m that use roughly $2\lceil \log_2 m \rceil$ bits. Again, assume that m is at least 1.

Exercise 3.2. Improve the upper bound on how many bits must be used to encode a number m in a self-delimiting way by iterating the above strategy. Informally discuss the limitations of this approach.

Exercise 3.3. Consider the following idea. No single bit is interpreted as a letter, but three consecutive bits, and the information is encoded using this larger alphabet. The sequence "111" marks the end of the current string. Argue why this strategy is not superior to the one we introduced above; here, we are interested in the case that the length of the string we want to encode tends to infinity.

Of course, there are situations where we do not need a self-delimiting encoding of the advice. Similarly to barely random algorithms, we can design online algorithms with advice that read a constant number of advice bits, which is fixed from the beginning; an extreme case was the simple algorithm for the ski rental problem from Example 3.1. Moreover, we could think of algorithms that read a fixed number of advice bits in every time step. In this case, the algorithm knows how much advice to use as it realizes when the input ends.

3.3 Proving Lower Bounds

Proving the existence of some object is usually a lot simpler than proving its non-existence. To show that an object with some given property (for instance, an online algorithm with advice that uses a certain number of advice bits) exists, it suffices to construct such an object; therefore, such proofs are usually *constructive*. Showing that such an object does not exist may generally be more difficult. What we need to do is to prove that all possible objects do not have the given property. In our case, how do we prove that there is no online algorithm with advice that reads at most a given number of advice bits and achieves a given output quality? For such hardness results, we often use combinatorial arguments that, on a high level, work as follows.

1. For infinitely many $n \in \mathbb{N}^+$, construct sets of instances of length n that have unique and pairwise different optimal solutions.
2. If an algorithm has to make different decisions for, say, two different instances in some time step T_i with $1 \leq i \leq n$, the common prefix of length i of these instances is the same. In other words, a deterministic online algorithm cannot tell which of the two instances it is dealing with until after T_i.
3. From Observation 3.1, we know that an online algorithm with advice that uses $b(n)$ advice bits can be seen as picking one from $2^{b(n)}$ deterministic algorithms. Thus, for each of the instances, one of these algorithms is chosen. If the set of instances of length n is larger than $2^{b(n)}$, some instances must be processed by the same deterministic algorithm; we do not know which, but we can still

argue why this implies that this particular algorithm cannot be optimal on all of these instances (or is even unable to achieve some particular competitive ratio).

This idea behind this is formalized by so-called "partition trees." These are used to structure a set of instances according to common prefixes. In what follows, for every instance I of the given online problem, let $[I]_{n'}$ denote the prefix of length n' of I; likewise $[O]_{n'}$ denotes the prefix of length n' of a solution $O \in \mathrm{sol}(I)$. For every instance I, let $\mathrm{solOpt}(I) \subseteq \mathrm{sol}(I)$ denote the set of optimal solutions for I.

Definition 3.3 (Partition Tree). Let $\mathcal{I}$ be a set of instances of some online problem Π. A *partition tree* $\widehat{\mathcal{T}}$ of $\mathcal{I}$ is a tree with the following properties.

(i) Every vertex $\hat{v}$ of $\widehat{\mathcal{T}}$ is labeled by a set $\mathcal{I}_{\hat{v}} \subseteq \mathcal{I}$ of instances and a natural number $\rho_{\hat{v}}$ such that all instances in $\mathcal{I}_{\hat{v}}$ have a common prefix of length at least $\rho_{\hat{v}}$.

(ii) For every inner vertex $\hat{v}$ of $\widehat{\mathcal{T}}$, the set of instances of its children form a partition of the instances of $\mathcal{I}_{\hat{v}}$.

(iii) For the root $\hat{r}$, we have $\mathcal{I}_{\hat{r}} = \mathcal{I}$.

The set of instances $\mathcal{I}$ does not necessarily only include instances of the same length (see Theorem 8.13). The usual way to define partition trees, however, is to construct sets $\mathcal{I}^{(n)}$ of instances of length n for infinitely many n together with partition trees for every $\mathcal{I}^{(n)}$. The key to using partition trees to prove lower bounds on the advice complexity of optimal online algorithms with advice is formalized by the next lemma.

Lemma 3.1. *Let $\mathcal{I}$ be a set of instances of some online problem Π with a partition tree $\widehat{\mathcal{T}}$ of $\mathcal{I}$. Let $\hat{v}_1$ and $\hat{v}_2$ be two vertices from $\widehat{\mathcal{T}}$ such that neither one is an ancestor of the other one, let $I_1 \in \mathcal{I}_{\hat{v}_1}$ and $I_2 \in \mathcal{I}_{\hat{v}_2}$ be any two instances of Π, and let $\hat{v}$ be the lowest common ancestor of both $\hat{v}_1$ and $\hat{v}_2$. If*

$$[O_1]_{\rho_{\hat{v}}} \neq [O_2]_{\rho_{\hat{v}}} ,$$

for every $O_1 \in \mathrm{solOpt}(I_1)$ and $O_2 \in \mathrm{solOpt}(I_2)$, then every optimal online algorithm with advice has to use different advice strings for I_1 and I_2.

Proof. Since $\hat{v}$ is an ancestor of both $\hat{v}_1$ and $\hat{v}_2$, we have both $I_1 \in \mathcal{I}_{\hat{v}}$ and $I_2 \in \mathcal{I}_{\hat{v}}$, for all $I_1 \in \mathcal{I}_{\hat{v}_1}$ and $I_2 \in \mathcal{I}_{\hat{v}_2}$. Due to Definition 3.3, we have $[I_1]_{\rho_{\hat{v}}} = [I_2]_{\rho_{\hat{v}}}$, but due to the assumption of the lemma, $[O_1]_{\rho_{\hat{v}}} \neq [O_2]_{\rho_{\hat{v}}}$ for every $O_1 \in \mathrm{solOpt}(I_1)$ and $O_2 \in \mathrm{solOpt}(I_2)$. In other words, the instances have the same prefix of length $\rho_{\hat{v}}$, but their optimal solutions differ in the first $\rho_{\hat{v}}$ answers.

Now let Alg be any optimal online algorithm with advice for Π, and assume that Alg reads the same advice for I_1 and I_2, which means that it chooses the same

deterministic algorithm A for both instances. Since I_1 and I_2 have the same prefix of length $\rho_{\hat{v}}$, that is, they look identical to A up to time step $T_{\rho_{\hat{v}}}$, A produces the same output on this prefix. However, by the assumption of the lemma, some of the first $\rho_{\hat{v}}$ optimal answers must be different for I_1 and I_2, and therefore A and thus ALG cannot compute optimal solutions for both of them. □

Finally, we can use Lemma 3.1 to prove the following theorem.

Theorem 3.1. *Let $\mathcal{I}$ be a set of instances of some online problem Π with a partition tree $\widehat{\mathcal{T}}$ of $\mathcal{I}$ with w leaves, such that the conditions of Lemma 3.1 are satisfied. Then every optimal online algorithm with advice for Π has to use at least $\log_2 w$ advice bits.*

Proof. It follows from Lemma 3.1 that, under the given conditions, every optimal online algorithm with advice needs to use two different advice strings for any two instances that correspond to different vertices in $\widehat{\mathcal{T}}$ with neither one being an ancestor of the other. Thus, such an algorithm needs to use a different advice string for every leaf. Since, when reading at most b bits, there are 2^b different advice strings, it follows that $2^b \geq w$ must be satisfied, and thus $b \geq \log_2 w$. □

In order to keep our arguments simple, we will usually try to construct the set $\mathcal{I}$ such that all instances have unique optimal solutions that are only optimal for this one instance. Moreover, the leaves of the partition tree are such that they only contain (a set with) a single instance each. In many of the subsequent lower-bound proofs, we will not explicitly construct partition trees, but incorporate the above ideas in our direct arguments. Learning about this general idea is important if one is to see the bigger picture of what is happening. Sometimes, however, there will be alternative proofs that explicitly use partition trees.

The arguments for lower bounds can also be used in another way. Suppose that we can show that every deterministic online algorithm can only be optimal (c-competitive, respectively) for, say, at most δn instances of length n of some online problem Π. The best case for an online algorithm ALG with advice is met if all these sets of instances are disjoint. Suppose ALG uses at most $b(n)$ advice bits for inputs of length n. If we are able, for infinitely many n, to construct a set of instances of Π of size $\mu(n)$ with $\mu(n) > 2^{b(n)} \cdot \delta n$, then we know that ALG cannot be optimal (c-competitive, respectively).

3.4 The Advice Complexity of Paging

We are now ready to study the advice complexity of the paging problem, which we used before to illustrate the concepts of deterministic and randomized online computation.

3.4.1 Optimality

We start by describing three different approaches to design optimal online algorithms with advice that have linear advice complexity.

Example 3.4. We design a simple online algorithm PLin1 with advice and an oracle for paging that work as follows. The oracle inspects the input I, which consists of a sequence of pages that have indices between 1 and m (where m is the number of pages in total). The most straightforward strategy would be to communicate the whole instance I by encoding the indices. Note that, for any I with $|I| = n$, there are obviously m^n different instances. Clearly, if PLin1 knows the complete instance in advance, it can be optimal. A number between 1 and m can be encoded with $\lceil \log_2 m \rceil$ bits, thus we need a total of $n\lceil \log_2 m \rceil$ advice bits for this strategy.

However, we are not done yet. PLin1 needs to compute an optimal solution in advance; but it does not know the length of the input and the advice tape has infinite length. Thus, we can use self-delimiting strings as described in Section 3.2. PLin1 knows the number of different pages m in advance and it can thus compute $\lceil \log_2 m \rceil$; so the oracle "only" needs to tell the algorithm the concrete input length n. As we know from Section 3.2, writing it down in a self-delimiting way can be done with $2\lceil \log_2(\lceil \log_2 n \rceil) \rceil + \lceil \log_2 n \rceil$ bits.

PLin1 now proceeds as follows. It starts reading the advice tape until it finds a 0 at an even position. After that, it computes $\lceil \log_2 n \rceil$ from the first $\lceil \log_2(\lceil \log_2 n \rceil) \rceil$ bits it found at odd positions. Then it reads the next $\lceil \log_2 n \rceil$ bits and computes n. Now knowing n and m, it reads the next $n\lceil \log_2 m \rceil$ advice bits and interprets them as a sequence of length n of numbers between 1 and m. For this instance, it computes an optimal solution and acts according to it. All in all, we have thus created an optimal online algorithm with advice that uses

$$2\lceil \log_2(\lceil \log_2 n \rceil) \rceil + \lceil \log_2 n \rceil + n\lceil \log_2 m \rceil$$

advice bits. ♢

This was probably the easiest approach one could come up with. However, in general, we think of m as a very large constant, especially with respect to the cache size k. With an approach that is almost as simple as the one from Example 3.4, we now design an online algorithm with advice that has an advice complexity that does not depend on m at all. The only thing that is required is a little more work for the oracle.

Example 3.5. How about not encoding the input, but the optimal output? Again, we design an online algorithm PLin2 with advice and an oracle. For a given instance I, the oracle computes an optimal solution $\textsc{Opt}(I)$ where Opt is some arbitrary but fixed optimal algorithm. This solution is uniquely defined by a sequence of length at most n of numbers between 1 and k. Each number simply represents the position of the cache cell that Opt uses on a page fault. Thus, in every time step where the requested page is not in PLin2's cache, it reads the next $\lceil \log_2 k \rceil$ bits from the

advice tape and removes the corresponding page. Clearly, both algorithms compute the same solution. Moreover, since no algorithm can make more than n page faults in total, PLIN2 never uses more than $n\lceil \log_2 k \rceil$ advice bits.

There is a nice detail about this strategy. Since the oracle already computed the optimal solution for us, we do not need to communicate n to the algorithm. PLIN2 can just read exactly $\lceil \log_2 k \rceil$ bits in every time step where it causes a page fault, and k is known in advance. ♢

Both PLIN1 and PLIN2 are optimal online algorithms for paging with linear advice complexity. However, their advice complexities differ in the multiplicative constant. We now prove that it is even possible to be optimal without using any multiplicative constant.

Theorem 3.2. *There is an optimal online algorithm* PLIN3 *with advice for paging that uses at most $n + k$ advice bits.*

Proof. Let OPT be an optimal offline algorithm for paging. We call a page in the cache of OPT *active* if it is requested once more before OPT removes it from the cache. PLIN3 is designed such that it also has every active page in its cache in the corresponding time step. To this end, the algorithm has a flag for every cache cell that marks the page it contains as either active or *passive*. Note that passive pages do not necessarily correspond to the pages in OPT's cache that are not active.

For every request that causes a page fault, PLIN3 removes an arbitrary page that is passive. So, if a page p is requested that causes a page fault for PLIN3, this cannot be an active page as PLIN3 has all active pages in its cache in every time step. Furthermore, p cannot be a passive page that is in OPT's cache at this point in time since this immediately contradicts the definition of passive pages. Thus, p also causes OPT to make a page fault in this time step. OPT now removes a page p' that is not active. In this case, there is always a passive page in PLIN3's cache, which may be different from p'. It follows that PLIN3 does not cause more page faults than OPT; but then PLIN3 must be optimal as well.

Now let us bound the number of advice bits from above. For every request, PLIN3 reads a bit from the advice tape that indicates whether the requested page is active or passive (if the page is already in the cache, its flag is updated). For PLIN3 to be optimal, the k pages that are in the cache at the beginning need to be marked active or passive before the input is processed. It follows that PLIN3 uses $n + k$ advice bits in total. □

Especially the difference between Example 3.5 and Theorem 3.2 gives us an idea about what *advice complexity* is all about. In the former case, we basically encode a complete optimal solution. Thus, PLIN2 really knows exactly what it has to do when a page fault occurs, that is, which page must be replaced. But this full knowledge is not necessary; what needs to be "extracted" from this information is just which pages may be removed and which must not be removed. Which concrete page is then chosen from the ones that are allowed to be removed is not important in computing an optimal solution.

Exercise 3.4. Prove that there is a 1-competitive online algorithm with advice for paging that uses at most n advice bits.

Exercise 3.5. Suppose that we change the definition of paging such that the cache of any online algorithm is empty at the beginning. Does this affect the upper bound of Theorem 3.2? What happens if the optimal algorithm starts with a different cache content?

Exercise 3.6. Prove that if $m = k + 1$, then there is an optimal online algorithm PLIN4 with advice for paging that uses $\lceil n/k \rceil \cdot \lceil \log_2 k \rceil$ advice bits.

Next, we complement the upper bound with some lower bounds. In this context, this means that we need to show that there is no optimal online algorithm with advice that uses fewer than a given number of advice bits. As mentioned in the previous section, such a proof is in many cases harder than the above constructive proofs. We now use one of the approaches described to give a linear lower bound on the advice complexity of any optimal online algorithm with advice for paging.

Theorem 3.3. *Every optimal online algorithm with advice for paging has to use at least* $(\log_2 k/k)n$ *advice bits if the total number of pages* m *may depend on* n*.*

Proof. Let n be a multiple of k. We construct a set $\mathcal{I}^{(n)}$ that contains instances of length n of the following form. Every instance is again divided into N phases; each phase consists of exactly k requests for different pages. We also design an optimal algorithm OPT that replaces exactly one page in every phase. Any algorithm that diverges from OPT at some point cannot be optimal, as we show in the following.

Let $p_{j_1}, p_{j_2}, \ldots, p_{j_k}$ denote the pages that are in the cache of OPT at the beginning of some phase P_j with $1 \leq j \leq N$. In P_j, first a page $\overline{p}_j$ is requested that is different from all these pages and that was never requested before. This causes a page fault for any demand paging algorithm. Next, $k - 1$ of the k pages that were in OPT's cache at the beginning of P_j are requested. This means that there is some page $p'_j \in \{p_{j_1}, p_{j_2}, \ldots, p_{j_k}\}$ that OPT can replace in the first time step $T_{(j-1)k+1}$ of P_j without causing an additional page fault during P_j. The important point is that, due to the new page $\overline{p}_j$, every demand paging algorithm must make one page fault in every phase. If, in some phase, a second page fault is caused, this cannot be compensated afterwards.

Now we show that the optimal solution for any given instance from $\mathcal{I}^{(n)}$ is unique. For a contradiction, suppose there is a different optimal solution. Thus, in some phase, the two corresponding solutions for the first time replace different pages in the first time step of this phase (if both solutions replace the same page, they both do not make page faults in the remainder of this phase). However, this immediately implies that one of them makes two page faults during this phase and therefore cannot be optimal. Moreover, any two different instances have different optimal solutions, because they need to replace different pages at least once to make only one page fault in each phase.

Next, we calculate how many instances there are in total for inputs of length n. There are $N = n/k$ phases in total. In every phase P_j, exactly one page p'_j, which

is one of the k pages $p_{j_1}, p_{j_2}, \ldots, p_{j_k}$, is not requested. Since the page $\overline{p}_j$ that is requested first is the same for every instance, there are k different possibilities for a request sequence in one phase (the order of the other pages does not matter as it does not influence the optimal solution). It follows that

$$|\mathcal{I}^{(n)}| = k^{n/k} ,$$

and as a result of the observations we just made, each instance has a unique optimal solution that is only optimal for this particular instance.

Thus, every online algorithm Alg with advice that reads fewer than

$$\log_2\bigl(k^{n/k}\bigr) = \frac{n}{k} \cdot \log_2 k$$

advice bits uses one deterministic strategy $A \in \mathrm{strat}(\textsc{Alg}, n)$ for two different instances from $\mathcal{I}^{(n)}$. Let these two instances be I_1 and I_2. There is a phase P_i in which for the first time two different pages $p'_{i,1}$ and $p'_{i,2}$ for I_1 and I_2 are replaced by $\overline{p}_i$ in the corresponding optimal solutions $\textsc{Opt}(I_1)$ and $\textsc{Opt}(I_2)$, respectively. However, the prefixes of I_1 and I_2 that include the request $x_{(i-1)k+1} = \overline{p}_i$ are identical, and thus A replaces the same page for both instances; it immediately follows that A causes one additional page fault for one of the two instances. As a result, A cannot be optimal for both I_1 and I_2; and therefore Alg cannot be optimal for them as well. □

Exercise 3.7. Give an alternative proof of Theorem 3.3 using partition trees (see Definition 3.3).

The arguments used in the proof of Theorem 3.3 rely on the fact that m is unbounded, that is, the number of pages requested in total grows with the input length n. It is preferable to get rid of this undesired requirement while maintaining that every input has one unique optimal solution. A naive approach that simply uses the same idea as above with a constant number of pages does not seem very promising, as the following example suggests.

Example 3.6. Let $k = 5$, $m = 6$, and suppose that the caches of Alg and Opt are, as always, initialized with the first five pages, that is, we have

Opt: | p_1 | p_2 | p_3 | p_4 | p_5 | and Alg: | p_1 | p_2 | p_3 | p_4 | p_5 | .

Assume that we follow the same strategy as we used in the proof of Theorem 3.3. The first phase starts by requesting p_6 and four of the other pages. In our example, the instance I starts with p_6, p_3, p_4, p_5, p_2. After the first request, Opt replaces the page p_1 with p_6, and therefore causes one page fault. Now let us assume that Alg replaces p_2 instead, which leads to the situation

Opt: | p_6 | p_2 | p_3 | p_4 | p_5 | and Alg: | p_1 | p_6 | p_3 | p_4 | p_5 | .

The subsequent requests of P_1 do not cause a page fault for OPT, but clearly, ALG causes a page fault when p_2 is requested in T_5. ALG may replace any page with p_2; let us assume it chooses p_3, which leads to

OPT: $\boxed{p_6}\boxed{p_2}\boxed{p_3}\boxed{p_4}\boxed{p_5}$ and ALG: $\boxed{p_1}\boxed{p_6}\boxed{p_2}\boxed{p_4}\boxed{p_5}$

when P_1 is over.

Now P_2 starts by requesting the unique page that is not in the cache of OPT, that is, p_1; the next four requests could be p_2, p_4, p_5, p_6. In this case, OPT replaces p_3 with p_1 in T_6, which causes one page fault. However, ALG does not induce any page fault in P_2. Therefore, after P_2, both algorithms made two page faults in total and we have

OPT: $\boxed{p_6}\boxed{p_2}\boxed{p_1}\boxed{p_4}\boxed{p_5}$ and ALG: $\boxed{p_1}\boxed{p_6}\boxed{p_2}\boxed{p_4}\boxed{p_5}$,

that is, both caches have the same content. ♢

We need to enlarge the phases so that there is still only one unique optimal solution for any instance we construct. Then, it is possible to give a proof for $m = k + 1$ that still shows a linear lower bound (with a constant that is two times worse).

Example 3.7. Again, let $k = 5$ and $m = 6$. Since the straightforward approach of Example 3.6 does not work, we now repeat each phase a second time right after the k different pages of this phase were requested. In this context, we speak of the first and second "iteration" of the phase. So this time, the instance starts with a phase P_1, which is, for instance, given by

$$(\underbrace{p_6, p_2, p_3, p_4, p_5}_{\text{iteration 1}}, \underbrace{p_6, p_2, p_3, p_4, p_5}_{\text{iteration 2}}) .$$

OPT again replaces p_1 with p_6 in T_1 and has cost 1 in P_1. If ALG again decides to replace another page with p_6 instead, for instance, p_5, this leads to a second page fault in the first iteration, because p_5 is requested again. We distinguish two cases depending on what ALG does when p_5 is requested during the first iteration of P_1.

Case 1. Assume that ALG replaces p_1 with p_5. The cache content is then

$\boxed{p_2}\boxed{p_6}\boxed{p_3}\boxed{p_4}\boxed{p_5}$,

which corresponds to the cache content of OPT after P_1. However, ALG made one additional page fault so far, and enters the next phase without any advantage compared to OPT.

Case 2. Assume that ALG replaces a page with p_5 such that its cache remains different from that of OPT, for instance, ALG removes p_4. This leads to

$\boxed{p_1}\boxed{p_6}\boxed{p_3}\boxed{p_5}\boxed{p_2}$,

which implies a third page fault in the second iteration of P_1 when p_4 is requested. Again, ALG may replace a page that leads to a cache content different from OPT,

and it may be the case that ALG has cost 0 in the next phase P_2 as a consequence. However, we can show that this does not give ALG an advantage with respect to the whole instance since it now caused two more page faults than OPT.

In both cases, ALG is worse than OPT. ♢

We generalize this idea to prove the following theorem.

Theorem 3.4. *Every optimal online algorithm with advice for paging has to use at least* $(\log_2 k/(2k))n$ *advice bits.*

Proof. Let there be $m = k+1$ pages in total; let n be a multiple of $2k$. We construct a set $\mathcal{I}^{(n)}$ of instances in the following way. Again, every input $I \in \mathcal{I}^{(n)}$ is divided into N phases, this time of length $2k$ each. Every phase P_j with $1 \leq j \leq N$ starts by requesting page $\overline{p}_j$, which is currently not in the cache of OPT. Then, as in the proof of Theorem 3.3, $k-1$ pages are requested that are all in the cache of OPT when P_j begins. These k different pages are then requested in the same order one more time. As in Example 3.7, we refer to these two sequences of k requests as the first and second iteration, respectively.

First, we prove that OPT is both optimal and unique. To this end, we show that, for all $I \in \mathcal{I}^{(n)}$, any solution that deviates from $\text{OPT}(I)$ is worse than OPT on I. Let ALG be some algorithm such that $\text{ALG}(I)$ and $\text{OPT}(I)$ differ; as before, we assume that ALG is a demand paging algorithm. Let P_j with $1 \leq j \leq N$ be the first phase in which ALG replaces a different page than OPT. This must happen at the beginning of P_j, that is, in time step $T_{(j-1)2k+1}$, because, if both algorithms replace the same page in such a time step, they act identically in the rest of the phase (since they both do not cause additional page faults during this phase).

As P_j is the first phase in which the algorithms differ, they have the same cache content at the beginning of P_j, and thus requesting $\overline{p}_j$ causes a page fault for both ALG and OPT. Since OPT removes the unique page p'_j that is not requested during P_j (note that $p'_j = \overline{p}_{j+1}$, for $1 \leq j \leq N-1$), ALG causes one additional page fault in this first iteration of P_j. This happens when the page $p''_j \neq p'_j$ is requested, which ALG replaced with $\overline{p}_j$ at the beginning. If $j = N$, it immediately follows that ALG is worse than OPT. Thus, in what follows, we assume that $1 \leq j \leq N-1$. We now distinguish two cases depending on ALG's action when p''_j is requested.

Case 1. If ALG replaces p'_j with p''_j, then the two algorithms end phase P_j with the same cache content, but ALG caused an additional page fault.

Case 2. If ALG replaces some page $p'''_j \neq p'_j$ with p''_j, then it will have another page fault in the second iteration, when p'''_j is requested again. On this request, ALG can again replace p'_j with p'''_j, which again leads to a cache content identical to that of OPT. If ALG chooses another page to replace with p'''_j, the two algorithms enter P_{j+1} with different cache contents.

If $j+1 = N$, we are again done. So suppose $j+1 < N$; then there is a (possibly empty) sequence of phases such that the cache content of ALG differs from that

of Opt at the end of each phase. For any such phase, Alg makes at least one page fault (since exactly the pages in Opt's cache were requested), and thus the distance between the two algorithms stays the same. If this is true for all remaining phases, we are again done. Conversely, suppose there is a phase $P_{j'}$ in which Alg causes no page fault. Then, it still caused one more page fault than Opt, but has the same cache content as Opt at the end of $P_{j'}$.

In any case, either the input ends with Alg having a larger cost than Opt, or both algorithms end up in a situation in which they start the next phase with the same pages in their caches, but Alg made at least one more page fault than Opt. Thus, we can apply the above arguments inductively for the remainder of the input. Since Alg is worse than Opt on I, it follows that the solution computed by Opt is indeed unique and optimal.

By the same reasoning as in the proof of Theorem 3.3, an optimal online algorithm with advice needs to use two different advice strings for any two instances from $\mathcal{I}^{(n)}$. Consequently, it needs to use

$$\log_2\left(k^{n/(2k)}\right) = \frac{n}{2k} \cdot \log_2 k$$

advice bits. □

In conclusion, a linear number of advice bits is both necessary and sufficient to compute an optimal output for any paging instance.

3.4.2 Small Competitive Ratio

We continue with studying how much a small amount of advice can help when dealing with the paging problem. In particular, suppose an online algorithm is only allowed to read a constant number of advice bits that does not depend on the input length n. Actually, we already know that we can achieve quite a lot with a constant number of advice bits. We designed a barely random algorithm for paging in Section 2.6 that is $(3b + 2(k+1)/2^b)$-competitive in expectation when using b random bits; in the first section of this chapter (see Observation 3.2(i)), we have seen that this enables us to construct an online algorithm with advice that uses b advice bits and is as good. We can therefore derive the following theorem.

Theorem 3.5. *There is an online algorithm with advice for paging that uses b advice bits, where $2^b < k$, and is strictly*

$$\left(3b + \frac{2(k+1)}{2^b}\right)\text{-competitive.}$$

Proof. This is a direct consequence of Theorem 2.12 and Observation 3.2. □

Next, we complement this upper bound with a lower bound for a constant number of advice bits which is very close to it.

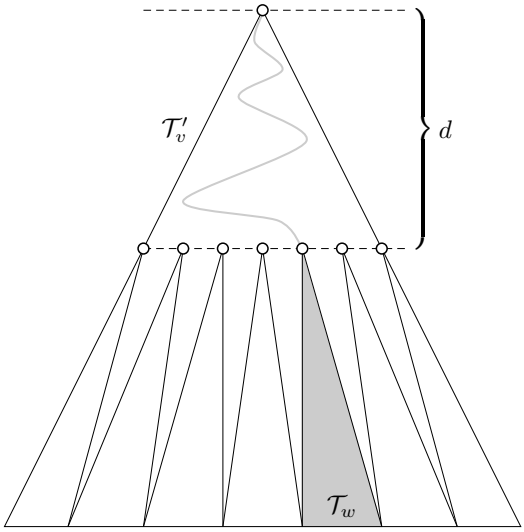


Figure 3.1. The tree $\mathcal{T}_v$ that represents some instances from $\mathcal{I}^{(n)}$.

Theorem 3.6. *Let $\varepsilon > 0$. No online algorithm with advice for paging that uses b advice bits, where $2^b < k$, is*

$$\left(\frac{k}{2^b} - \varepsilon\right)\text{-competitive.}$$

Proof. Let $\varepsilon > 0$, and let ALG be some online algorithm with advice for paging that uses a constant number b of advice bits. For the proof, it is again sufficient to consider instances with a total of $m = k + 1$ pages. Let n be a multiple of k. We now construct a set $\mathcal{I}^{(n)}$ of instances of length n that we can represent by a special kind of tree. All instances start by requesting the page p_{k+1}, which is not in the cache. Now we arrange all instances in a k-ary tree $\mathcal{T}$ that has n levels, that is, $\mathcal{T}$ is of height $n - 1$. The leaves represent the complete instances from $\mathcal{I}^{(n)}$ of length n. In general, every vertex v corresponds to a prefix of exactly those instances that are leaves in the subtree rooted at v. Every inner vertex in the tree has exactly k children, which represent the k possible pages that can be requested in the following time step. Of course, the same page is never requested twice in two consecutive time steps. It follows that the root of the tree corresponds to the first request p_{k+1}, which is a prefix of every instance that is represented by the tree.

For every instance, ALG chooses one out of 2^b online algorithms from strat(ALG); thus, every instance from $\mathcal{I}^{(n)}$ is processed by one of these algorithms. We are now going to color the leaves of $\mathcal{T}$ depending on which algorithm processes them; so every leaf gets a color between 1 and 2^b.

Let $d := \lfloor n/2^b \rfloor$. For every vertex v, we denote the subtree of $\mathcal{T}$ that has the root v by $\mathcal{T}_v$. We now show that the following is true for every v.

> *If $\mathcal{T}_v$ has at least $d \cdot i$ levels and all leaves of $\mathcal{T}_v$ are colored with at most i colors, then there is an instance in $\mathcal{I}^{(n)}$ (represented by a leaf of $\mathcal{T}_v$) for which* Alg *causes at least d page faults.* (3.1)

We prove the claim by induction on the number of colors i.

Base Case. Let $i = 1$, and let $\mathcal{T}_v$ be a tree with at least d levels whose leaves are colored with one color. This is equivalent to the situation where Alg uses the same algorithm from strat(Alg) for all instances represented by $\mathcal{T}_v$, and therefore works fully deterministically. Then there is an instance in $\mathcal{T}_v$ such that Alg causes exactly one page fault on every level of $\mathcal{T}_v$; thus, it makes d page faults in total, which covers the base case.

Induction Hypothesis. The claim holds for $i - 1$.

Induction Step. Let $i > 1$. We cut T_v after d levels yielding a tree $\mathcal{T}'_v$. Every leaf w of $\mathcal{T}'_v$ is the root of a subtree $\mathcal{T}_w$ of $\mathcal{T}$ with at least $d(i-1)$ levels; see Figure 3.1. We now distinguish two cases depending on the number of colors that are used in the trees $\mathcal{T}_w$.

Case 1. If there is a tree $\mathcal{T}_w$ whose leaves are colored with at most $i - 1$ different colors, then, by the induction hypothesis, it follows that Alg causes d page faults on some instance that is represented by a leaf of $\mathcal{T}_w$. Obviously, then there is also such an instance that is represented by a leaf of $\mathcal{T}_v$.

Case 2. Conversely, if such a tree does not exist, since all leaves of $\mathcal{T}_v$ are colored with i colors, we know that there is a color z such that every subtree $\mathcal{T}_w$ has a leaf that is colored with z. If we take these leaves from every subtree, the corresponding instances are again processed by the same algorithm, that is, the same advice is used for each of them.

Due to the construction of $\mathcal{T}$, the request sequences that lead to the corresponding trees $\mathcal{T}_w$ are all possible request sequences of length d (where the same page is never requested in two consecutive time steps). Hence, there is an instance such that Alg causes a page fault on all levels of $\mathcal{T}'_v$; therefore, it makes d page faults in total.

Now we can use (3.1) for $i = 2^b$ and conclude that Alg causes at least

$$\left\lfloor \frac{n}{2^b} \right\rfloor$$

page faults on inputs that are represented by a tree with $\lfloor n/2^b \rfloor \cdot 2^b \leq n$ levels, and thus have length at most n. On the other hand, we know that there is an optimal algorithm that makes a page fault at most every k requests. Since n is a multiple of k, it follows that the optimal cost for instances of length n is at most n/k.

To finish the proof, we plug these two bounds into Definition 3.2, that is, the definition of the competitive ratio, yielding

$$\frac{n}{2^b} - 1 \leq \left\lfloor \frac{n}{2^b} \right\rfloor \leq c \cdot \frac{n}{k} + \alpha \,,$$

which is why the competitive ratio of ALG can be bounded from below by

$$c \geq \frac{k}{2^b} - \frac{(\alpha+1)k}{n} \,,$$

which is larger than $k/2^b - \varepsilon$ for infinitely many n. □

3.5 Advice and Randomization

If we follow our intuition, advice bits seem to be a lot more powerful than random bits. After all, we compare a situation where we always pick a best strategy for the given instance to a situation where we pick strategies with a fixed distribution; in essence, we compare "the best" with "the average." It is therefore natural to ask whether there exists a scenario in which it is possible to save some bits if they are supplied by an oracle and not a random source. In what follows, we give a positive answer to this question. More specifically, we show that if there is some randomized online algorithm RAND for some online minimization problem Π, then there is also some online algorithm with advice that is almost as good while using a number of advice bits which (and this is the interesting part) does not depend on the number of random bits RAND uses. However, the bound does depend on the number of possible instances of Π of the given length. The proof uses some ideas that are similar to the proof of Yao's principle, which we have introduced in Sections 2.3 and 2.4.

Theorem 3.7. *Let Π be an online minimization problem with $\mu(n)$ different instances of length n. Suppose there is a randomized online algorithm for Π that is c-competitive in expectation. Then, for any $\varepsilon > 0$, there is a $(1+\varepsilon)c$-competitive online algorithm with advice for Π that uses at most*

$$2\lceil \log_2(\lceil \log_2 n \rceil) \rceil + \lceil \log_2 n \rceil + \log_2\left(\left\lfloor \frac{\log_2(\mu(n))}{\log_2(1+\varepsilon)} \right\rfloor + 1 \right)$$

advice bits.

Proof. Let RAND be a randomized online algorithm for Π that uses $b(n)$ random bits for any input length n. Due to Observation 2.2, this is equivalent to choosing uniformly at random a deterministic strategy from a set $\mathrm{strat}(\text{RAND}, n) = \{A_1, A_2, \ldots, A_{2^{b(n)}}\}$. We design an online algorithm ALG with advice for Π in the following way. Since RAND is c-competitive in expectation, according to Definition 2.2, there is a constant α such that, for every instance I of Π, we have

$$\mathbb{E}[\mathrm{cost}(\text{RAND}(I))] \leq c \cdot \mathrm{cost}(\text{OPT}(I)) + \alpha$$

or, equivalently,

$$\frac{\mathbb{E}[\text{cost}(\textsc{Rand}(I))] - \alpha}{\text{cost}(\textsc{Opt}(I))} \leq c\,.$$

Now, for each deterministic strategy A_j and each instance I_i of length n, for $1 \leq j \leq 2^{b(n)}$ and $1 \leq i \leq \mu(n)$, we set

$$c_{i,j} := \frac{\text{cost}(A_j(I_i)) - \alpha}{\text{cost}(\textsc{Opt}(I_i))}\,;$$

recall that $c_{i,j}$ is called the *performance* of A_j on I_i. As a next step, we construct a $(\mu(n)\times 2^{b(n)})$-matrix $\mathcal{M}$ that we fill with these entries similarly to Section 2.4.

	A_1	A_2	A_3	$\dots$
I_1	$c_{1,1}$	$c_{1,2}$	$c_{1,3}$	$\dots$
I_2	$c_{2,1}$	$c_{2,2}$	$c_{2,3}$	
I_3	$c_{3,1}$	$c_{3,2}$	$c_{3,3}$	
$\vdots$	$\vdots$			$\ddots$

As a result, the entry in the ith row and the jth column gives the performance of Rand on the input I_i if Rand chooses the deterministic strategy A_j. The central idea of the proof is to show that we are able to cleverly choose a small number of columns of $\mathcal{M}$ such that the performances of the corresponding deterministic strategies are good for many instances, and the chosen strategies cover all input instances. We collect these deterministic algorithms in a set $\mathcal{A}$, and Alg gets as advice the index of the algorithm from $\mathcal{A}$ that should be used for the input at hand (and some additional information we will describe later); the "index" can, for instance, refer to the canonical order of the binary random strings to which the algorithms correspond.

One row i of $\mathcal{M}$ corresponds to exactly one input I_i. Thus, by the definition of $c_{i,j}$ and the expected competitive ratio of Rand, for every i with $1 \leq i \leq \mu(n)$, we get

$$\begin{aligned}
\frac{1}{2^{b(n)}}\sum_{j=1}^{2^{b(n)}} c_{i,j} &= \frac{1}{2^{b(n)}}\sum_{j=1}^{2^{b(n)}} \frac{\text{cost}(A_j(I_i)) - \alpha}{\text{cost}(\textsc{Opt}(I_i))} \\
&= \frac{\frac{1}{2^{b(n)}}\sum_{j=1}^{2^{b(n)}} (\text{cost}(A_j(I_i)) - \alpha)}{\text{cost}(\textsc{Opt}(I_i))} \\
&= \frac{\mathbb{E}[\text{cost}(\textsc{Rand}(I_i))] - \alpha}{\text{cost}(\textsc{Opt}(I_i))} \\
&\leq c
\end{aligned}$$

or, equivalently,

$$\sum_{j=1}^{2^{b(n)}} c_{i,j} \leq c \cdot 2^{b(n)}\,.$$

For the sum of all entries in all cells of $\mathcal{M}$, we get

$$\sum_{i=1}^{\mu(n)} \sum_{j=1}^{2^{b(n)}} c_{i,j} \leq \sum_{i=1}^{\mu(n)} c \cdot 2^{b(n)} \leq c \cdot 2^{b(n)} \cdot \mu(n) \; .$$

Since there are $2^{b(n)}$ columns in $\mathcal{M}$, there is one column (deterministic strategy) j' such that

$$\sum_{i=1}^{\mu(n)} c_{i,j'} \leq c \cdot \mu(n) \; . \tag{3.2}$$

The corresponding online algorithm $A_{j'}$ is then included in $\mathcal{A}$ and it is used for any instance I_i, for which $c_{i,j'} \leq (1+\varepsilon)c$. Let $s = s(j')$ denote the number of these instances. In what follows, we want to estimate the size of s, that is, for how many instances ALG can use $A_{j'}$. Clearly, the performance of $A_{j'}$ is worse than $(1+\varepsilon)c$ on $\mu(n) - s$ instances.

Summing up, this gives a total of more than $(\mu(n)-s)(1+\varepsilon)c$ for the corresponding rows and we have

$$\sum_{i=1}^{\mu(n)} c_{i,j'} > (\mu(n) - s)(1+\varepsilon)c \; .$$

Together with (3.2), it follows that $(\mu(n) - s)(1+\varepsilon)c < \mu(n)c$ and therefore

$$s > \left(\frac{\varepsilon}{1+\varepsilon}\right)\mu(n) \; ,$$

which means we can use the deterministic strategy $A_{j'}$ for a fraction $\varepsilon/(1+\varepsilon)$ of the instances as we know that on these its performance is not worse than $(1+\varepsilon)c$.

After $A_{j'}$ is put into the set $\mathcal{A}$, we remove the column j' from $\mathcal{M}$ together with all rows that correspond to inputs on which $A_{j'}$ achieves a sufficiently good performance. There remain

$$\left(1 - \frac{\varepsilon}{1+\varepsilon}\right)\mu(n) = \left(\frac{1}{1+\varepsilon}\right)\mu(n)$$

rows for which we need to find another algorithm from strat(RAND, n). For every remaining row, the removed entry in column j' was larger than c. It follows that, after removing this column, the average over all entries of the remaining rows is still not larger than c. Therefore, we can repeat the aforementioned procedure with the remaining $1/(1+\varepsilon)\mu(n)$ rows of $\mathcal{M}$. This way, we find another deterministic online algorithm $A_{j''}$, which has a sufficiently good performance on a fraction $\varepsilon/(1+\varepsilon)$ of the remaining instances.

Now we compute how often we have to iterate this procedure until we have found an algorithm for every input; this means we want to find a natural number r such that

$$\left(\frac{1}{1+\varepsilon}\right)^r \mu(n) < 1 \, .$$

We get

$$\left(\frac{1}{1+\varepsilon}\right)^r < \frac{1}{\mu(n)} \iff (1+\varepsilon)^r > \mu(n) \iff r > \log_{1+\varepsilon}(\mu(n)) \, ,$$

which means that we have to make at most

$$\left\lfloor \frac{\log_2(\mu(n))}{\log_2(1+\varepsilon)} \right\rfloor + 1$$

iterations, that is, we need that many deterministic algorithms from $\mathrm{strat}(\textsc{Rand}, n)$. This immediately gives an upper bound on the size of $\mathcal{A}$.

Finally, we calculate the number of advice bits needed for this approach.

1. First, Alg needs to know the input length n, which can be encoded on the advice tape using $\lceil \log_2 n \rceil$ bits. However, this must be done in a self-delimiting fashion (as described in Section 3.2), summing up to a total of $2\lceil \log_2(\lceil \log_2 n \rceil) \rceil + \lceil \log_2 n \rceil$ advice bits.
2. Knowing n, Alg constructs $\mathcal{M}$ by simulating the randomized online algorithm Rand on every possible input of length n and every possible "random" string of length $b(n)$. Then, Alg constructs $\mathcal{A}$ and enumerates all algorithms from $\mathcal{A}$ in, for instance, canonical order. After reading another
$$\log_2\left(\left\lfloor \frac{\log_2(\mu(n))}{\log_2(1+\varepsilon)} \right\rfloor + 1\right)$$
advice bits, Alg can pick one algorithm from $\mathcal{A}$, which is then simulated for the input at hand.

It follows that the performance of Alg on any instance is at most $(1+\varepsilon)c$ and Alg uses as much advice as claimed by the theorem. □

Exercise 3.8. Explain where the argumentation in the proof of Theorem 3.7 does not work if we use Observation 2.1 instead of Observation 2.2, and specifically assume that $\ell(n) < 2^{b(n)}$?

The following example illustrates how the deterministic strategies are chosen in the proof of Theorem 3.7.

Example 3.8. Suppose we are given a randomized online algorithm RAND that uses three random bits for the given input length n; thus, we have $\text{strat}(\text{RAND}, n) = \{A_1, A_2, \ldots, A_8\}$. As described above, the online algorithm with advice knows n, computes $b(n)$, and finally simulates RAND on every input of length n and every "random" string of length $b(n)$.

Moreover, assume there are nine inputs of length n, and that RAND is 3-competitive in expectation. Then ALG obtains the following matrix $\mathcal{M}$.

	A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8	Average
I_1	5	4	4	2	3	1	4	1	3
I_2	3	1	1	3	5	5	2	4	3
I_3	4	6	4	1	2	4	1	2	3
I_4	1	1	5	5	4	2	3	3	3
I_5	2	2	4	2	5	2	2	5	3
I_6	1	5	1	8	1	2	4	2	3
I_7	2	1	4	1	4	3	5	4	3
I_8	1	3	1	7	2	2	3	5	3
I_9	3	3	6	2	1	4	1	4	3

As marked in $\mathcal{M}$, the deterministic strategy A_2 has a performance that is better than $(1+\varepsilon)\cdot 3$ (even 3 in this simple example) on the six inputs I_2, I_4, I_5, I_7, I_8, and I_9. A_2 is included in $\mathcal{A}$, and the second column is removed from $\mathcal{M}$ together with the rows that correspond to the above inputs. This results in the following matrix with decreased average values for every row.

	A_1	A_3	A_4	A_5	A_6	A_7	A_8	Average
I_1	5	4	2	3	1	4	1	2.86
I_2								
I_3	4	4	1	2	4	1	2	2.58
I_4								
I_5								
I_6	1	1	8	1	2	4	2	2.72
I_7								
I_8								
I_9								

Finally, there are even two algorithms, namely A_5 and A_8, that perform well for all remaining instances. Thus, $\mathcal{A}$ has size 2. ♢

At this point, one might wonder whether the factor of $1+\varepsilon$ is unavoidable, or whether it is possible to design an online algorithm with advice from any randomized online algorithm that is as good. The answer is that the latter is not possible, as the following online problem shows.

Example 3.9. Consider the following online minimization problem. The input $I = (x_1, x_2, \ldots, x_n)$ starts with the request $x_1 = 0$. All other requests are bits, that

is, $x_i \in \{0,1\}$ with $2 \leq i \leq n$; as answers, also bits must be given, that is, $y_i \in \{0,1\}$ with $1 \leq i \leq n$.

Now, if $y_i = x_{i+1}$ for all i with $1 \leq i \leq n-1$, then the total cost is 1, otherwise, it is n (the last answer y_n is ignored). In other words, an optimal algorithm has cost 1, and any other solution has cost n. Obviously, a best randomized online algorithm chooses every answer such that it is 0 or 1 with probability 1/2 each. This algorithm uses $n-1$ random bits and its expected competitive ratio is

$$\frac{\frac{2^{n-1}-1}{2^{n-1}} \cdot n + \frac{1}{2^{n-1}} \cdot 1}{1} = n - \frac{n+1}{2^{n-1}} \,.$$

On the other hand, no online algorithm Alg with advice that uses fewer than $n-1$ advice bits is better than n-competitive. This is due to the fact that there are at most 2^{n-2} deterministic strategies Alg chooses from if it uses at most $n-2$ advice bits. Therefore, there are at least two different instances that get the same advice string. Let these two instances be I_1 and I_2, and let $A \in \text{strat}(\text{Alg})$ be the deterministic algorithm that is chosen for both of them. I_1 and I_2 differ for the first time in time step T_i with $2 \leq i \leq n$; but they have the same prefix of length $i-1$. Therefore, A outputs the same bit in T_{i-1}, and consequently has cost n on one of the two instances. ♢

In Chapter 7, we will return to such problems where bits need to be guessed; however, there we will consider different cost functions. It follows that there exist online problems for which an online algorithm with advice that is equally good as a best randomized online algorithm in expectation needs as many advice bits as the latter uses random bits.

We have now introduced the three models of online computation that we will study in the following chapters. It will turn out that the relationship between them varies heavily with the problem we are considering.

3.6 Historical and Bibliographical Notes

The advice complexity of online problems was introduced by Dobrev et al. [53] in 2008 as a new measurement for online algorithms addressing the aforementioned pessimistic view of competitive analysis. In particular, the authors investigated paging and the problem of *differentiated services*; see, for instance, Lotker and Patt-Shamir [112]. Originally, two different "modes of operation" were proposed and studied, which allow different ways of communication between the oracle and the online algorithm. The following description is taken from Komm [97].

- *The helper model.* Here, we think of an oracle that oversees Alg's actions during runtime. The oracle may interact with the algorithm by giving some bits of advice in every time step; this is done without a request for help by Alg. The crucial observation is that the advice may be empty, which can also carry some piece of information and thus may be exploited by the algorithm.

- *The answerer model.* The second model is more restrictive in some sense as the algorithm Alg has to explicitly ask the oracle for advice in some time step. The oracle then has to respond with some advice string, that is, it is not allowed to send an empty string, but still may encode some extra information into the length of its answer.

In both models, it is assumed that Alg knows the length of the input in advance. A more detailed and formal description is given by Dobrev et al. [53, 54].

The problem with the above model (and both modes of operation) is that some information can be encoded into the lengths of the advice strings. Moreover, it is desirable to stick to a scenario where the input length of the current instance is not known in advance, because this is exactly one of the properties that define the nature of computing online. Of course, it is possible that the input length is communicated to the algorithm, but we demand that this is accounted for in a clean way and it is not possible to do such a thing implicitly.

Addressing these issues, two different refined models were suggested in 2009. Emek et al. [58] proposed a model in which the number of advice bits supplied is fixed in every time step. Note that it is therefore impossible to study sublinear advice as we did, for example, in Theorem 3.5. Emek et al. applied this model to metrical task systems and the k-server problem (which we will investigate in Chapter 4). Böckenhauer et al. [30] and Hromkovič et al. [82] proposed the model that is used throughout this book. Hromkovič et al. suggested using this approach to quantify the information content of the given online problem. Böckenhauer et al. first applied it to paging (obtaining some of the results given in Section 3.4), the disjoint path allocation problem (which will be described in Subsection 7.4.3), and the job shop scheduling problem (which we will study in Chapter 5); for more details, we refer to the technical report [31].

It is noteworthy that this model is equivalent to a variant of the answerer model where the input length is unknown and the online algorithm with advice specifies the number of bits it wants to get as an answer (it may ask multiple times in one time step); the oracle is not allowed to answer with any other number of advice bits (in particular, it is also not allowed to give an empty answer).

The self-delimiting encoding of binary strings that we introduced in Section 3.2 is strongly related to *Elias coding*, which was developed by Elias [57].

The concept of partition trees was implicitly used in many publications. It was first formalized (see Definition 3.3, Lemma 3.1, and Theorem 3.1) by Barhum et al. [16]; see also Steffen [135].

The problem-independent construction of an online algorithm with advice from a randomized online algorithm for online minimization problems was shown by Böckenhauer et al. [24, 29]. An analogous theorem for online maximization problems was later proven by Selečéniová [129].

Emek et al. [58, 60] proved a lower bound on the advice complexity of an online algorithm that also uses randomness for metrical task systems. Böckenhauer et al. [25] introduced the so-called boxes problem to further study the collaboration between

advice and randomization. More non-trivial connections between randomization and advice were observed by Komm [97], Mikkelsen [117], and Böckenhauer et al. [23].