1

Introduction

In everyday life, we are often confronted with *online problems*. Informally, this means that we have to make decisions—often banal ones, but sometimes also ones with significant impact—without knowing the future. Imagine, for example, you are in the car, on the way into your long-awaited vacations; taking the fastest route, you know it would be possible to reach your vacation destination within 8 hours—if it were not for traffic. An hour ago, you decided to leave your planned route due to a congested road ahead, just to find yourself in a bumper-to-bumper traffic jam right after that. In the end, you arrive at your destination, totally exhausted, after a 12-hour drive. It seems that every time you made the decision to alter your planned route or to stick to your current one, you made a bad choice. Being confronted with choices, not knowing what consequences each possible decision will eventually have, is a frustrating daily routine. There are various other examples for situations in real life in which we are forced to make decisions without knowing the future; ranging from rather insignificant ones, such as choosing appropriate clothing for a hike without knowing how the weather is going to develop, to choices with a great impact on our financial situation, such as deciding in which stock to invest.

Online computation is the field of computer science that deals with the formalization of such online problems and the development and analysis of algorithms to solve them. An algorithm that is supposed to solve some online problem receives its input piecewise and has to choose how to proceed with each piece of the input immediately, without any information about the future input and without the possibility to revise its decisions. Such an algorithm is called an *online algorithm*. The quality of an online algorithm A is traditionally measured in terms of its *competitive ratio*, which relates the quality of the solutions computed by A to the quality of the solutions computed by an optimal *offline algorithm* that knows the whole input in advance. Since we desire algorithms that also perform reasonably

well when given a worst-case input, we usually assume that the input given to the algorithm is chosen by a malicious *adversary* whose goal is to maximize the algorithm's competitive ratio, whereas the algorithm's goal is to minimize it.

For many problems in real life, we sometimes wish we could get information from some source of unlimited knowledge. We would like to know, for example, which route will be least congested, or what stock is going to increase soon. And, as a matter of fact, for all examples mentioned above, great efforts have been made to be able to make predictions about the future; there are the weather forecast, navigation systems with built-in algorithms for bypassing traffic jams, and attempts to predict fluctuation in the stock market. In online computation, this concept also exists, in the form of an omniscient *oracle* with unlimited computing power and full knowledge about the input instance at hand. This oracle can provide the online algorithm with *advice bits* to reveal crucial information about the input instance and thus improve the quality of the solution computed by the algorithm. The field of *advice complexity theory* deals with the question of how many advice bits are necessary and sufficient to compute solutions of a certain quality.

An especially interesting and challenging task is to prove lower bounds on the number of advice bits. One tool has proven to be extremely helpful in this regard, namely the *string guessing problem*. This problem is a very generic online problem, maybe even the most generic online problem, and it can be discovered in many other online problems. The input being an unknown string of length n, an online algorithm for the string guessing problem is asked to guess this input string, letter by letter. The task of the algorithm is to guess as many letters correctly as possible. Although this problem is extremely elementary, even very elaborate online problems can be interpreted in one way or the other as guessing letters of an unknown input string. This circumstance can be exploited to specify a *reduction* from the string guessing problem to a given online problem. The concept of reductions is often applied in computer science to prove that some given problem P is not easier to solve than another problem Q, transfering already known hardness results for Q to hardness results for P. This thesis has a strong focus on the string guessing problem, and on constructing reductions to obtain lower bounds on the advice complexity of other online problems.

1.1 This Dissertation

The remainder of this chapter serves to introduce the mathematical foundations that we will need throughout this thesis in a formal way. Apart from fixing some mathematical concepts and notation in Section 1.2, we give formal descriptions of

the concepts of online computation (Section 1.3), online compuation with advice (Section 1.4), and the string guessing problem (Section 1.5).

Each of the following four chapters deals, in one way or the other, with the string guessing problem and how it can be used to infer results concerning the advice complexity of other online problems. Chapter 2 covers the k-*server problem*, Chapter 3 the *disjoint path allocation problem*, and Chapter 4 addresses two related problems, the *graph exploration* and the *graph searching problem*. In Chapter 5, we introduce a more powerful adversary that is able to choose random bits, and analyze the string guessing problem thoroughly in this new model. We show that, also in this model, the string guessing problem can be used to transfer results for this problem to other online problems.

Several problems analyzed in this dissertation have been proposed by or developed in collaboration with my colleagues. In particular, Juraj Hromkovič pointed me to all problems that are investigated in this thesis. First ideas for the lower bounds presented in Chapter 2 were developed during a workshop in Montserrat; the technical details were developed autonomously afterwards. Most of the results in Chapter 3 have been developed together with Heidi Gebauer, Dennis Komm, Rastislav Královič, and Richard Královič, and those in Chapter 4 in collaboration with Dennis Komm, Rastislav Královič, and Richard Královič. The model of the probabilistic adversary from Chapter 5 has been proposed by Juraj Hromkovič; all results and technical details therein were found and elaborated in independent work by the current author.

1.2 Mathematical Foundations

In this section, we present a short overview of the most important mathematical concepts and notation being used throughout this thesis. However, we broach every subject only briefly, mainly to fix our notation. For a more general introduction to online algorithms, see the textbook of Borodin and El-Yaniv [BEY98]; the concept of advice complexity is discussed in detail by Komm [Kom12].

1.2.1 Sets

A *set* is a collection of objects. These objects are called *elements* of the set and are usually required to be pairwise distinct. We write, for example, $\{0, 1\}$ for a set containing the two elements 0 and 1. The *cardinality* or *size* of a set S is the number of elements contained in S and denoted by $|S|$. The *empty set* is denoted by $\emptyset$. For each set S, we denote by $\mathcal{P}(S)$ the *power set* of S, defined as

$$\mathcal{P}(S) = \{R \mid R \subseteq S\}.$$

Whenever the order in which the elements of a set are listed matters, we talk about *ordered sets*. For ordered sets, we drop the requirement of all elements being pairwise distinct and allow multiple occurrences of the same element. To distinguish ordered sets from unordered ones, we use parentheses instead of braces to denote the former; for example, we write $(0, 1)$ instead of $\{0, 1\}$. An ordered set is also called a *sequence* or a *tuple*. For sequences, often the term *length* is used instead of size or cardinality. Tuples of size n are also called n-*tuples*; furthermore, tuples of size 2 are called *pairs*.

Throughout this thesis, we use the standard notation for the ordered sets of integers and real, rational, and natural numbers. For the set of *real numbers*, we use the symbol $\mathbb{R}$. We denote the set of *rational numbers* by $\mathbb{Q}$ and the set of *integers* by $\mathbb{Z}$. The set of *natural numbers* is denoted by $\mathbb{N}$. We often need to constrain our considerations to numbers that do not exceed or fall below a certain threshold. In such cases, we sometimes add a superscript to the set symbol to indicate this threshold. For example, in this notation, the set of negative real numbers can be denoted by $\mathbb{R}^{<0}$, and we have $\mathbb{N} = \mathbb{Z}^{\geq 0}$. Concerning the latter statement, though, the literature is not completely consistent. Although in this thesis we usually assume that 0 is included in the set of natural numbers, in some literature it is not (hence, $\mathbb{N} = \mathbb{Z}^{\geq 1}$). Therefore, whenever we are talking about $\mathbb{N}$ and want to make completely clear whether 0 is to be included in our considerations or not, we also make use of this superscript notation and write either $\mathbb{N}^{\geq 0}$ or $\mathbb{N}^{\geq 1}$.

1.2.2 Alphabets and Strings

An *alphabet* is a nonempty finite set of *letters*, and is usually denoted by Σ throughout this thesis. Often we consider the *binary alphabet* $\Sigma_2 = \{0, 1\}$. The letters 0 and 1 in Σ_2 are called *bits*. A *string* over an alphabet Σ is a sequence $r = (r_1, \ldots, r_n)$ of letters from Σ, for some natural number $n \in \mathbb{N}^{\geq 0}$, and if the letter r_i is contained in Σ_2, for each i with $1 \leq i \leq n$, we call r a *binary string* or *bit string*. If $n = 0$, we say that r is the *empty string*, which we denote by ε. Instead of writing $r = (r_1, \ldots, r_n)$, we also use $r = r_1 \ldots r_n$ as a shorthand notation. A string $r' = r_1 \ldots r_m$ with $m \leq n$ is called a *prefix* of r. Comparably, a string $r' = r_m \ldots r_n$ with $m \geq 1$ is called a *suffix* of r.

1.2.3 Functions and Constants

For any subset S' of an ordered set S, we define the *minimum of* S', denoted by $\min(S')$, to be an element x of S' such that $x \leq y$ for all elements $y \in S'$; analogously, we define the *maximum of* S', denoted by $\max(S')$, to be an element x of S' such that $x \geq y$ for all elements $y \in S'$.

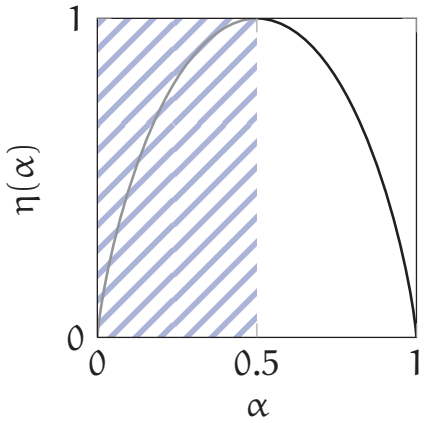

Figure 1.1. The binary entropy function $\eta(\alpha) = -\alpha\log(\alpha) - (1-\alpha)\log(1-\alpha)$. For our purposes, α is only considered in the range $1/2 \leq \alpha < 1$. Therefore, the other part of the graph is hatched.

For any real number $x \in \mathbb{R}$, we use $\lfloor x \rfloor$ to denote the largest integer y with $y \leq x$ and call it the *floor* of x. Accordingly, $\lceil x \rceil$ denotes the smallest integer y such that $y \geq x$ and is called the *ceiling* of x.

For any two real numbers $x \in \mathbb{R}$ and $y \in \mathbb{R}^{>0}$ with $y \neq 1$, we denote the *logarithm to base* y *of* x by $\log_y(x)$. In this thesis, logarithms are usually to base 2 if not stated otherwise. Hence, usually we take a pass on mentioning the base explicitly and just write $\log(x)$ instead of $\log_2(x)$. If it does not introduce any ambiguity, we also often omit the parentheses and write $\log x$ instead of $\log(x)$.

Furthermore, we will often encounter the so-called *entropy*, which is, originally, a measure of the information content of a given string and plays a great role in the field of coding theory (see, for example, Roth [Rot06]). For any real number $p \in \mathbb{R}$ with $0 \leq p \leq 1$ and every natural number $q \in \mathbb{N}^{\geq 2}$, the q-*ary entropy function of* p is defined as

$$\eta_q(p) = p\log_q(q-1) - p\log_q(p) - (1-p)\log_q(1-p),$$

where $0\log_q(0)$ is assumed to be 0. For the binary entropy function, which is the version we are usually considering, this yields

$$\eta_2(p) = p\log_2(p) - (1-p)\log_2(1-p).$$

A plot of the binary entropy function is shown in Figure 1.1. As for logarithms, we allow us to drop the subscript in the binary case and often write $\eta(p)$ instead of $\eta_2(p)$.

We follow the convention to give complexity measures in terms of orders of magnitude. To this end, we use the *Landau symbols* to group functions into classes according to their asymptotical growth. For any two functions $f\colon \mathbb{N}^{\geq 0} \to \mathbb{R}^{\geq 0}$ and $g\colon \mathbb{N}^{\geq 0} \to \mathbb{R}^{\geq 0}$, we say that f does not grow asymptotically faster than g, denoted by $f(n) \in O(g(n))$, if

$$\exists\, n_0, c > 0, \text{ such that } \forall\, n \geq n_0 : f(n) \leq c \cdot g(n),$$

and we say that g grows asymptotically faster than f, denoted by $f(n) \in o(g(n))$, if

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0.$$

Moreover, we use the notations

$$\begin{aligned} f(n) \in \Omega(g(n)) &\iff g(n) \in O(f(n)), \\ f(n) \in \omega(g(n)) &\iff g(n) \in o(f(n)), \text{ and} \\ f(n) \in \Theta(g(n)) &\iff g(n) \in O(f(n)) \cap \Omega(f(n)). \end{aligned}$$

In some contexts, we will come across *Euler's number*, a mathematical constant that we denote by e and which can be approximated by $e \approx 2.718$.

1.2.4 Combinatorics

The *factorial* of n, i. e., the product of all positive natural numbers from 1 to n, is denoted by $n!$, for any natural number $n \in \mathbb{N}^{\geq 0}$, where $0!$ is assumed to be 1. For natural numbers $n, k \in \mathbb{N}^{\geq 0}$, the *binomial coefficient* $\binom{n}{k}$ indicates the number of possibilities to choose k elements out of a set containing n elements; it can be calculated as

$$\binom{n}{k} = \frac{n!}{k! \cdot (n-k)!}.$$

1.2.5 Probability Theory

At some point in this thesis, we will add a certain random element to the game between the online algorithm, the adversary, and the oracle. More precisely, we will allow the adversary to "toss a coin" (if necessary several times) and choose the instance given to the algorithm as its input depending on the outcome of these coin tosses. Such random elements with an uncertain outcome are called *experiments*, and each possible outcome is called an *elementary event*. The set S of all elementary events of an experiment is the *sample space*, and an *event* is a subset of the sample space and thus an element of the power set $\mathcal{P}(S)$ of S. In this thesis, we only encounter discrete probabilistic models, in which the sample space is a finite set. To assign a probability to each event, we use a function $\Pr\colon \mathcal{P}(S) \to [0, 1]$. This function is called a *probability distribution over* S and has to fulfill the following constraints.

(a) $\Pr(\{s\}) \geq 0$ for every elementary event $\{s\} \subseteq S$,

(b) $\Pr(S) = 1$, and

(c) $\Pr(A \cup B) = \Pr(A) + \Pr(B)$ for all events $A, B \subseteq S$ with $A \cap B = \emptyset$.

The pair $(S, \Pr)$ forms a so-called *probability space*. If the function Pr is such that each elementary event $\{s\} \subseteq S$ occurs with the same probability, i. e., if

$$\Pr(\{s\}) = \frac{1}{|S|} \quad \text{for all } \{s\} \subseteq S,$$

then Pr is called the *uniform distribution*.

A *random variable* in the probability space $(S, \Pr)$ is a function $X\colon S \to \mathbb{R}$ assigning a real number to every elementary event from the sample space. The probability that the random variable X attains a certain exact value y is given by the *probability mass function* $f_X\colon \mathbb{R} \to [0, 1]$, defined by

$$f_X(y) = \Pr(X = y) = \Pr\Big(\big\{\{s\} \subseteq S \mid X(s) = y\big\}\Big).$$

The probability mass function characterizes the *probability distribution of* X, which is formalized by a function $d_X\colon \mathbb{R} \to [0, 1]$, defined as

$$d_X(y) = \Pr(X \leq y) = \sum_{\substack{z \leq y \\ z \in D_X}} \Pr(X = z),$$

where $D_X \coloneqq \{y \in \mathbb{R} \mid \exists\, s \in S \text{ such that } X(s) = y\}$ is the co-domain of X. For a discrete probability space $(S, \Pr)$ and a random variable X in $(S, \Pr)$, the *expected value of* X is defined as

$$\mathbb{E}[X] = \sum_{y \in D_X} y \cdot \Pr(X = y).$$

A detailed introduction to randomized computation and probability theory is given by, e. g., Hromkovič [Hro05].

1.2.6 Graphs

In every one of the subsequent chapters, we will deal with certain classes of graphs. A *graph* is a pair $G = (V, E)$, where $V = \{v_0, \ldots, v_{n-1}\}$ is a set of *vertices*, some of which are connected by *edges*. The set of edges is given by $E \subseteq \{(v_i, v_j) \mid 0 \leq i, j \leq n - 1\}$. Throughout this thesis, we constrain ourselves to graphs that do not contain any loops, i. e., edges of the form (v_i, v_i).

Graphs can either be weighted or unweighted. In an *edge-weighted* or just *weighted graph*, each edge is assigned a *cost* or *weight* according to a *weight function* $\omega\colon E \to \mathbb{R}$. In an *unweighted graph*, such a weight function does not exist, and we usually assume every edge to have a weight of 1.

Both weighted and unweighted graphs can either be directed or undirected. In a *directed graph*, each edge (v_i, v_j) has an orientation, with v_i being the *startpoint* and v_j being the *endpoint* of (v_i, v_j). To v_i, the edge (v_i, v_j) is an *outgoing edge* and to v_j, it is an *incoming edge*. The *outdegree* of v_i is the number of outgoing edges

of v_i, and the *indegree* of v_i is the number of incoming edges of v_i. A sequence of pairwise distinct vertices $U := (u_0, u_1, \ldots, u_\ell)$ with $u_i \in V$, for $0 \leq i \leq \ell$, is called a *path from* u_0 *to* u_ℓ if, for every pair (u_i, u_{i+1}) with $0 \leq i \leq n-1$, there is an edge $(u_i, u_{i+1}) \in E$. The *length of the path* is the sum of the edge weights of all these edges (u_i, u_{i+1}). In the unweighted case, this coincides with the number of edges on the path U, and hence, the length of this path is ℓ. We say that U is a *shortest path from* u_0 *to* u_ℓ if, among all paths from u_0 to u_ℓ, the path U has minimal length. If there exists a path from u_0 to u_ℓ, we also say that u_ℓ is *reachable* from u_0.

In an *undirected graph*, all edges are undirected, meaning that the edge (v_i, v_j) is identical to the edge (v_j, v_i). Thus, edges are not pairs but unordered sets of size 2, and any undirected edge (v_i, v_j) is usually written as $\{v_i, v_j\}$. We say that both v_i and v_j are *endpoints* of the edge $\{v_i, v_j\}$. For each edge $\{v_i, v_j\} \in E$, the vertex v_j is said to be a *neighbor of* v_i or *adjacent to* v_i. For each vertex v_i, all edges containing v_i are called *incident to* v_i. The *degree* of v_i is defined as the number of its neighbors. In an undirected graph $G = (V, E)$, a *path between* u_0 *and* u_ℓ is a sequence of pairwise distinct vertices $(u_0, u_1, \ldots, u_\ell)$ such that, for each i with $0 \leq i \leq \ell - 1$, the two vertices u_i and u_{i+1} are adjacent to one another. As in the directed case, the *length of a path* is the sum of the weights and thus the number of edges in the unweighted case. A sequence $(u_0, u_1, \ldots, u_\ell, u_0)$ is called a *simple cycle* in G if $(u_0, u_1, \ldots, u_\ell)$ is a path with $\ell \geq 2$ and if the edge $\{u_0, u_\ell\}$ exists in E. If there is a path from v_i to v_j, for any pair of vertices $(v_i, v_j) \in V \times V$, the graph is called *connected*.

For any directed or undirected, weighted or unweighted graph $G = (V, E)$, the graph $G' = (V', E')$ is called a *subgraph of* G if $V' \subseteq V$ and $E' \subseteq E$. The subgraph G' is called *induced by* V' if E' contains all edges that are also contained in E, constrained to the vertex set V'. Hence, the *subgraph of* G *induced by* $V' \subseteq V$ is the graph $G' = (V', E')$ with $E' = \{(v_i, v_j) \mid v_i, v_j \in V' \wedge (v_i, v_j) \in E\}$.

Throughout this thesis, we will sometimes consider particular classes of graphs, namely paths, cycles, and trees. In the following, we will give brief descriptions for these three classes. Since we consider each of these classes in its respective undirected unweighted version, we constrain our descriptions to these restricted versions, without mentioning this explicitly from now on.

A *path graph*, for short also named *path* (not to confuse with a path within a graph as described above), is a graph $G = (V, E)$ with $V = \{v_0, \ldots, v_\ell\}$ and $E = \{\{v_i, v_{i+1}\} \mid 0 \leq i \leq \ell - 1\}$. The *length of the path* is the number of edges contained in E, which is ℓ. A *cycle graph* or *cycle* is a path graph as described above with an additional edge between v_ℓ and v_0. Hence, $G = (V, E)$ is a cycle graph if $V = \{v_0, \ldots, v_\ell\}$ and $E = \{\{v_i, v_{i+1}\} \mid 0 \leq i \leq \ell - 1\} \cup \{\{v_\ell, v_0\}\}$. The length of the cycle is the number of edges in E, which is $\ell + 1$. A graph $G = (V, E)$ is called a *tree* if G is connected and does not contain any simple cycles. In every tree, vertices of degree at most 1 are called *leaves*; all other vertices, i. e., those with a degree of at least 2, are called *inner vertices*. Sometimes, we choose one designated vertex of

a tree $G = (V, E)$ to be the *root* of G. In this case, G is said to be *rooted*. For each vertex $v_i \in V$, there is exactly one shortest path U from the root to v_i, and if this path has length ℓ, we say that v_i is on *level* ℓ of the tree. For each vertex $v_i \in V$ on level ℓ, every neighbor v_j of v_i on level $\ell + 1$ is called a *child of* v_i. For each such child v_j, the vertex v_i is the only adjacent vertex on level ℓ and is called the *parent of* v_j. All other children of v_i that are not v_j itself are called *siblings of* v_j. If the maximum level of any vertex in the tree is d, then d is called the *depth of the tree*.

Sometimes, we consider q-ary trees, for some $q \in \mathbb{N}^{\geq 2}$. In the literature, a q-*ary tree of depth* d is often defined as a rooted tree in which all inner vertices have at most q children and the maximum level among all vertices is d. For our purposes, we choose a more restrictive definition and define a q-*ary tree of depth* d to be a rooted tree in which all inner vertices have exactly q children and all leaves are on the same level d. In our case, each q-ary tree has exactly q^d leaves. For $q = 2$, we call such a q-ary tree a *binary tree*. Hence, in a binary tree of depth d, each inner vertex has 2 children, and the number of leaves is 2^d.

1.3 Online Computation

The classical scenario of online computation can be viewed as a game between an online algorithm and an adversary. The game played is determined by the given *online optimization problem*. The goal of the *adversary* is to construct a problem instance that is as hard as possible for the online algorithm. The aim of the *online algorithm*, also called *online strategy*, is to compute a good solution on the input instance generated by the adversary. The *input instance* (also *input sequence* or just *input*) is given to the algorithm as a sequence $I = (x_1, \ldots, x_n)$ of *requests*, exactly one request in each *round*. Hence, the number of rounds corresponds to the length of the input sequence, which we usually denote by n. The online algorithm has to respond immediately to each request given in round i, i. e., before the next request arrives, with an irrevocable *output* y_i. The output sequence of an online algorithm A on an input I is then $(y_1, \ldots, y_n)$, and we denote it by $A(I)$.

To be able to compare the quality of online algorithms and their computed solutions, we assign a value to each solution according to its quality. Depending on the nature of the optimization problem, the algorithm is to achieve values either as small or as large as possible. In the former case, the optimization problem is called an *online minimization problem*; in the latter case, an *online maximization problem*. For minimization problems, the function that serves to assign a value to each solution is usually called a *cost function*; for maximization problems, we call this function a *gain function* accordingly. The value of a particular solution is then called the *cost* or the *gain* of this solution, respectively. For any instance I, a solution $A(I)$ computed by an algorithm A is *optimal* if it has minimum cost or maximum

gain, respectively, among all solutions computed on I by all possible algorithms. Then we say that A is *optimal on* I. An algorithm that is optimal on every possible input instance is called *optimal* and usually denoted by Opt throughout this thesis. Obviously, without knowing the whole input instance in advance, it is not possible for an online algorithm to be optimal in general; with this lack of knowledge, the online algorithm A might make a decision in some round i that turns out to be suboptimal later, when a larger part of the input sequence is known. Thus, receiving the input sequentially is a huge drawback compared to receiving it completely before the start of the computation, as so-called *offline algorithms* do. Hence, we are interested in the quality of the given online algorithm, which is usually measured by means of the *competitive ratio*, a measure of "how close to optimal" the algorithm is. This means that we compare the online algorithm to an optimal offline algorithm with unbounded memory and computing power. For any instance I and any online algorithm A, the solution $A(I)$ computed by A on I is c-*competitive* if there is a constant α independent of I such that

$$\mathrm{cost}(A(I)) \leq c \cdot \mathrm{cost}(\mathrm{Opt}(I)) + \alpha \tag{1.1}$$

for a minimization problem, and

$$\mathrm{gain}(\mathrm{Opt}(I)) \leq c \cdot \mathrm{gain}(A(I)) + \alpha \tag{1.2}$$

for a maximization problem. An online algorithm A is c-competitive if (1.1) or (1.2), respectively, holds for any possible input instance; hence, if A computes a c-competitive solution on any possible input instance I. Thus, A has a *competitive ratio* of at most c if there is a constant α such that, for any instance I, the solution computed by A on I is c-competitive. We say that an online algorithm as well as a solution is *strictly* c-*competitive* if the corresponding inequality holds for $\alpha \leq 0$. An optimal algorithm is strictly 1-competitive. (Diverging from some examples in the literature, we use two different formulas for minimization and maximization problems, making sure that the competitive ratio is always at least 1.)

In this way, we can analyze the competitive ratio of a given online algorithm or investigate what is the best achievable competitive ratio of any online algorithm for a given online optimization problem. Since it has been introduced in 1985 by Sleator and Tarjan [ST85], the competitive ratio has developed to the most relevant measure of the quality of online algorithms. Being a worst-case measurement, the competitive ratio in the game between an online algorithm and an adversary has proven to be very helpful in analyzing the hardness of online problems [BEY98].

In this game, the online algorithm can be strengthened by allowing it to use random bits. In this case, we are talking about a *randomized online algorithm*. If the adversary already knows these random bits before it has to construct its hard input instance, the randomization is obviously utterly useless. Therefore, usually