Olympiads in Informatics, 2011, Vol. 5, 58–70

Teaching the Concept of Online Algorithms

Dennis KOMM

Department of Computer Science
ETH Zurich, Switzerland
e-mail: dennis.komm@inf.ethz.ch

Abstract. The term *algorithm* is well-defined: It is the formal description of a strategy, which always has to produce a solution for a specific problem. It is probably the most basic concept of computer science and accordingly both the creation and implementation of algorithms are among the most important things students have to learn when dealing with the subject. The situation is usually as follows. Given an input x for some problem P, we are interested in designing an algorithm A that reads x, performs calculations depending on x and creates some output $A(x)$ of some predefined form. Conversely, in many practical applications, this model is actually not accurate. Here, at one specific point in time, only parts of the input are known to the algorithm whereas parts of the output are already needed. We call an algorithm handling such a situation an *online algorithm*. Among the numerous examples for such situations are parts of any operating system, or routing and scheduling problems.

To the best of our knowledge, teaching the concept of online scenarios and algorithms to secondary school students only received little attention in the past. However, our experiences show that it catches the students' attention when taught in a comprehensive and motivated way. In this paper, we want to propose a strategy of how to introduce online algorithms by explaining the major ideas to students and we further want to share our experiences.

Key words: teaching, secondary school, online algorithms, competitive analysis.

1. Introduction

We expect the reader to be familiar with the basic definitions and notations of *online algorithms* and *competitive analysis* (introduced by Sleator and Tarjan (1985)). For further reading, we point to the standard literature (Borodin and El-Yaniv, 1998; Hromkovič, 2006; Irani and Karlin, 1997; Sleator and Tarjan, 1985). The material is suited to be presented in a lesson using slides that takes approximately two hours.

Our main goal is to explain secondary school students the concept of online scenarios and to make sure they understand that these situations frequently occur in the real world, for instance, when users interact with an operating system. Our aim is to furthermore use online scenarios as a framework to introduce the students to the following ideas, techniques, and concepts.

1. Worst-case analysis, that is, "an algorithm is only as good as it performs on the hardest instances".
2. This analysis can be performed by means of an imaginary adversary[1] who tries to harm the algorithm as much as possible.
3. Randomization is a powerful tool in algorithm design.
4. It is important get a good intutition of the problem at hand to be able to make statements about it (to prove theorems).

In the presentation, we try to omit formal definitions and proofs whenever possible. Furthermore, we want to keep the notation as simple and straightforward as we possibly can. According to the amount of knowledge the students already possess, it might be a good idea to go into more detail at some points.

Please note that all results presented here have already been known. Especially those about *job shop scheduling with unit length tasks*, which we have chosen as a means to communicate the basic concepts introduced above, are due to Hromkovič *et al.* (2009), Hromkovič (2006, 2009).

2. A Sample Lesson

We now propose a concrete way of how to teach the concepts and ideas of online algorithms. We gave a talk with this material to a group of Swiss secondary school students and received very good feedback from both their teachers and the students themselves.

2.1. *Introducing Online Problems*

We first show the class something familiar by describing the principle of operation of an algorithm. Even if they never thought about a formal definition, they agree that the high-level scheme may be depicted as shown in Fig. 1. In our case, this is consistent with what the students experienced in class until this point, no matter how advanced their programming skills are. An algorithm A reads an input x, modifies it, and outputs a solution $A(x)$ afterwards. As a next step, we introduce a problem we want to solve by means of such an algorithm. We do not even need to talk about computer programs at this point, but only about an unambiguous and clear strategy a human can follow.

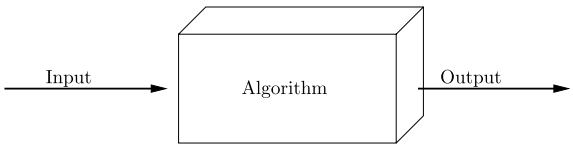


Fig. 1. The classical view of an algorithm.

[1] Here, we consider so-called *oblivious adversaries*.

Suppose we sit in front of a monitoring screen in a police central and want to route a number of k police cars through a city. At any time point, an emergency call might come in. Using GPS, we know the position of every car at this point and we are searching for a policy to decide which police car to radio and send to the corresponding crime scene.

Actually, this problem is the well-known k-server problem in disguise (Borodin and El-Yaniv, 1998), introduced by Manasse *et al.* (1990). To simplify the discussion, we assume that the police men are busy for a while after arriving at the crime scene and that we, therefore, are not allowed to move the same car twice. It is obvious to the students that this is a problem of practical relevance. Next, we do not talk about any problems that might appear in this scenario, but just propose a sample instance as shown in Fig. 2. There are three police cars stationed at the depicted positions. At 12 o'clock, an emergency call comes in. The crime spot is located between the positions of car 1 and car 2. It does not really seem to matter at this point which of the two cars we send. We therefore decide to take car 2. However, ten minutes later, car 2 arrives at the crime scene, but just one minute after that, a second crime is reported and it is located at the old position of car 2.

We can now clearly state that it would have been a better strategy to send car 1 instead of car 2 and we address the key point of online computation by directly asking the students why we made a bad decision. The answer is both trivial and important: "Because we did not know this before."

By introducing the problem of online scenarios in the above way, it is immediately clear that the restriction of not knowing the whole input in advance is actually very natural. We then point out that algorithm engineers have to deal with this problem in many applications. Some examples are

- operating systems (e. g., the well-studied *paging* problem; Borodin and El-Yaniv, 1998),
- routing problems (as shown in the example), and
- scheduling problems (which will be discussed in the following).

Before we introduce the problem we deal with in the main part of the lesson, we need to supply the basic tools and models we need in what follows. Therefore, the first question we ask is

How do we measure how good an online algorithm performs?

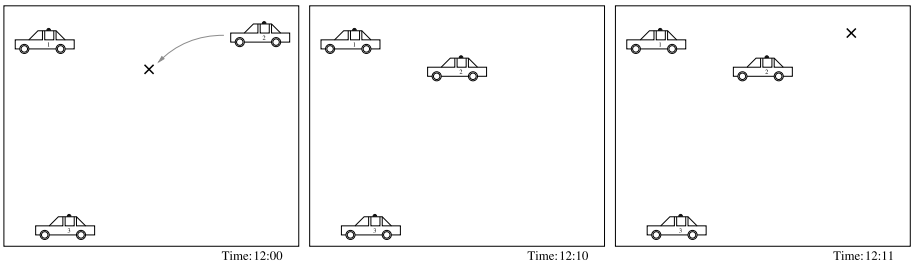


Fig. 2. An instance of the above problem to demonstrate the issues that come up when only knowing parts of the input.

By $\mathrm{cost}\,(A(x))$, we denote the costs of algorithm A on an input x. For the above problem, these costs are, e. g., the overall distance the cars have to drive (or, formulated differently, the overall time they need to arrive at a crime scene). Suppose that, at the end of the day, we browse through the log files of all emergency calls that appeared today. We now have the knowledge to compute a solution that is optimal with respect to the above measure. The costs of this solution are denoted by $\mathrm{Opt}\,(x)$. Compared to this solution, many of our decisions might have been bad, but we simply did not have another chance. We immediately had to send a police car when an emergency call came in. To evaluate our online strategy A on the input x, we want to look at the ratio

$$\frac{\mathrm{cost}\,(A(x))}{\mathrm{Opt}\,(x)},$$

and call this the competitive ratio of A on x. Furthermore, if we can guarantee that, for any possible instance x, we have

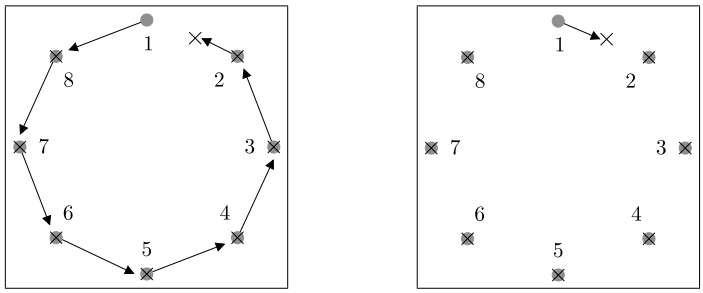
$$\frac{\mathrm{cost}\,(A(x))}{\mathrm{Opt}\,(x)} \leqslant d,$$

for some fixed d, we call A *d-competitive*. If the students already know the concept of approximation (offline) algorithms, they will surely recognize the analogy, but this is not important for understanding the concept in general. We now try to establish some intuition for *competitive analysis* in an informal way. The following question may be posed as an exercise to the students.

> *As we have already seen, the above problem seems to be somewhat hard for online algorithms, but how bad can it really be?*

A possible idea to give an answer to this question is shown in Fig. 3. Suppose the police cars are positioned in a circle. We label them clockwise from 1 to k starting at the car on top. The first request (emergency call) is exactly between the two cars 1 and 2. Suppose we again choose to send car 2. After that, the next request appears and it is located just at car 2's starting point. The nearest car is car 3, but right after we told the corresponding driver to move there, another crime is committed at the starting point of this car. The requests continue in this fashion until the kth request is made at the starting point of car k (Fig. 3a). On the other hand, if we would have known the sequence of requests in advance, we would have moved car 1 instead of 2 at the beginning. Afterwards, no other movement would have been necessary (Fig. 3b).

Summing up, until this point, we have shown that the classical definition of algorithms as shown in Fig. 1 is not accurate for online scenarios due to our inability to predict the future. However, it is crucial to note that the basic properties of algorithms (which separate them from arbitrary programs) are *not* violated: Online algorithms halt on any finite input (which might be a prefix of a potentially infinite instance) and they create an output of some well-defined form.



(a) The computed solution. (b) The optimal solution.

Fig. 3. A hard instance for the above problem. For the ease of presentation, police cars are marked by grey circles, requests are marked by black daggers.

2.2. *Worst-Case Analysis by Means of an Adversary*

As a next step, we want to elaborate the above idea of constructing worst-case input instances to talk about an online algorithm's performance. To do this, we ask the following question.

> *How do we know that an online algorithm A does not have a specific performance, i. e., is not able to achieve a certain competitive ratio?*

Observe that, in the above informal definition of competitiveness, we clearly stated that an online algorithm has to have a competitive ratio of at most d on *all* possible instances to be called d-competitive. We had good experiences with explaining the students the following analogy.

> *Suppose I claim to have built a robot that beats every human player in tennis. Of course, you do not believe me, but I insist on it. I am convinced that my robot is better than anyone who challenges him. To disprove the claim, you might ask every single inhabitant of earth to play a match against the robot and you are positive that, eventually, someone will be found who wins. However, with almost seven billion people living on earth, it might take quite a while until someone is found who does not lose. What could you do to speed things up?*

It does not take very long until the first student proposes: "We just let your robot play against Roger Federer." This is just the answer we expected[2]. We want to do the exact same thing with our algorithm A. We think of the best adversary there might possibly be. If he always has a strategy that makes A perform bad with respect to some measurement (that is, if he is able to construct a hard input instance where A can provably never be d-competitive), it directly follows that A is never better in general (is clearly not d-competitive), because we simply cannot guarantee that this input never appears in practice no matter how unlikely it is.

Accordingly, we introduce the adversary ADV that has the following properties:

- ADV knows A and

[2]Surely, there are equivalent or even better examples, but this one suits Switzerland.

- thus is able to predict any of its steps;
- ADV tries to make A perform as bad as possible.

Observe that we have actually requested the students to act as ADV for the police car problem when they were asked to construct an instance as shown in Fig. 3.

2.3. *Job Shop Scheduling with 2 Unit Length Tasks*

We are now ready to describe the problem we want to investigate in the following. The ideas how to present the material are taken from the book "Algorithmic Adventures" by Hromkovič (2009). Here, we want to use the things we have taught until this point to show how randomization helps to deal with the shortcomings of online computation. To this end, it is sufficient to informally define the problem as follows.

> *Suppose we are dealing with a factory that consists of m distinct machines, each of which is designed to perform a single distinct assignment, such that all machines are different from each other. Moreover, there are two so-called jobs Job_1 and Job_2 that each consist of m different tasks. Each of these tasks needs exactly one machine and the $(i+1)$th task may not start to be processed before task i is finished. As long as Job_1 and Job_2 request different machines at the same time, the work can be parallelized. However, if they request the same machine, one of the two has to be delayed. Our aim is to construct an online algorithm that minimizes these delays.*

If we label the machines from 1 to m, an example input is

$$Job_1 = (1, 2, 3, 4, 5, 6, 7, 8),$$
$$Job_2 = (3, 1, 2, 4, 6, 5, 8, 7),$$

which means that Job_2 first requests to perform a task on machine 3, after it is finished, it wants to perform a task on machine 1, and so on. To simplify the problem, we suppose that each task exactly needs one time unit (thus talking about unit length tasks). It is important to state that each job needs each machine *exactly* once, and therefore, the above sequences are permutations of the numbers from 1 to m.

A feasible solution S to this instance is

$$\text{Schedule}\,(Job_1) = (1, 2, 3, 4, 5, 6, /, 7, 8, /, /),$$
$$\text{Schedule}\,(Job_2) = (3, 1, 2, /, 4, /, 6, 5, /, 8, 7),$$

which means that the first three tasks of both jobs are served greedily (in parallel), but after that, both jobs request machine 4. Job_2 is delayed while Job_1 is served. In the next time step, Job_1 performs its fifth task on machine 5 and Job_2 performs its fourth task on machine 4 which is now free. This schedule induces a delay of 3.

In an online scenario, as we consider it, for each job, the $(j + 1)$th request is only known after the jth task is finished, i. e., while we have not processed task j, we have no idea which machine task $j + 1$ wants to use.

2.4. *A Graphical Representation*

This section covers an important lesson. Before we are able to describe results about the problem in a coherent way, we want to introduce a more comprehensible presentation of input instances to get a grip of the problem as shown in Fig. 4.

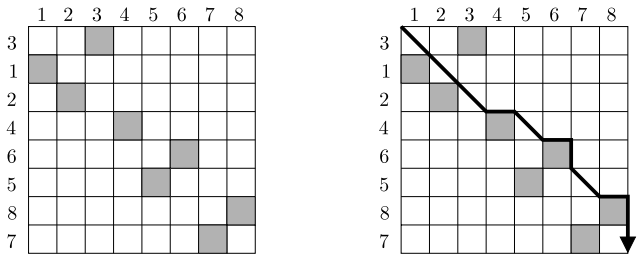
This description was introduced by Brucker (1988) and it is frequently used in the literature (Hromkovič *et al.*, 2009; Hromkovič, 2006; Hromkovič, 2009). Consider a 2-dimensional grid of size $m \times m$. We label the x-axis with the numbers from 1 to m in the order given by Job_1. We do the same thing with the y-axis and Job_2. Every cell that has the same number on both axis is marked by a grey square and we refer to these cells as *obstacles*. A feasible solution is a path through this grid from the upper left corner to the bottom right corner. Whenever two tasks can be performed in parallel, a diagonal step may be made (of course, it is also allowed to make a horizontal or a vertical step). However, if the path arrives at the upper left corner of a grey square, one job has to be delayed, and we say the path *hits an obstacle*. In this case, only a horizontal step (Job_2 is delayed) or a vertical step (Job_1 is delayed) is possible (we have to bypass the obstacle). Figure 4b shows the path that represents the solution S. Let us denote the number of horizontal [vertical, diagonal] steps of any strategy by $\mathtt{h}\,[\mathtt{v}\,,\mathtt{d}\,]$. We can make the following observations.

- O1 There always is *exactly* one obstacle in every row and every column.
- O2 For any feasible solution, we have $\mathtt{v} = \mathtt{h}$.
- O3 Also, for such a path, $\mathtt{d} + \mathtt{v} = \mathtt{d} + \mathtt{h} = m$.
- O4 Obviously, at least m steps have to be made in total.
- O5 Finally, the costs of every feasible solution are $\mathtt{d} + \mathtt{h} + \mathtt{v} = m + \mathtt{h} = m + \mathtt{v}$.

It is immediate that we aim at designing an online algorithm A that calculates a path of minimal length, i. e., makes as few non-diagonal steps as possible.

2.5. *An Adversary that Creates Hard Input Instances*

Using the above graphical representation, we now want to show how an adversary can create a bad input instance for any online algorithm.



(a) An example instance. (b) A feasible solution S.

Fig. 4. A graphical representation.

There exists an adversary ADV *that, by putting obstacles onto the grid in a clever way, can make sure that every second step of a path any online algorithm calculates, is not a diagonal one.*

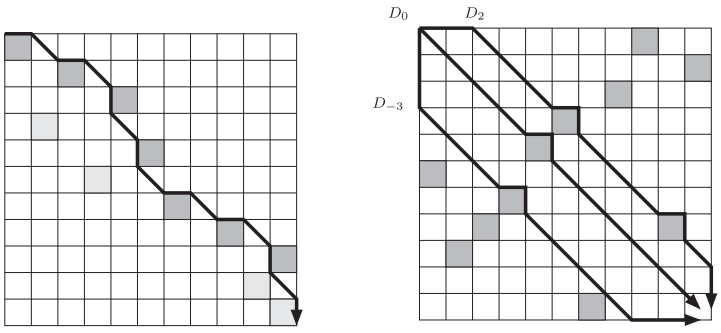
The idea of the proof is as simple as it gets. We do not prove it formally, but only show the students ADV 's strategy on an example which is straightforward to generalize. It is important to emphasize that, for a formal proof, we are not allowed to fix the algorithm in any way, but we would have to consider *all* algorithms. The following strategy might be presented on the blackboard (Fig. 5a). At first, ADV puts an obstacle to the the upper left cell by making both jobs request machine 1 at the start.

Note that the only restriction ADV has to obey in the following is given by Observation O1, that is, ADV is allowed to merely place *one* obstacle in every row and every column of the grid (otherwise, one job would request the same machine twice, which is forbidden by the problem definition). ADV 's strategy is as follows: Whenever A performs a diagonal step, the corresponding path enters a cell for which no task of both Job_1 and Job_2 has been assigned yet. ADV may then just place an obstacle by saying that both jobs now ask for the same machine with the smallest index that has not been used until this point. The next step of A must be either horizontal or vertical. By Observation O1, A may then perform a diagonal step thus enabling ADV to place another obstacle by the above strategy. If the path arrives at the right or bottom border of the grid, ADV may place the remaining obstacles in an arbitrary fashion.

Since there are at least m steps in total (Observation O4) and every second of them is not diagonal, it follows that there at least $m/2$ non-diagonal steps in the sum. Consequently, using Observation O2, the solution makes at least $m/4$ horizontal and $m/4$ vertical steps. The costs of this solution are therefore, by Observation O5, at least

$$\text{cost}\,(A(x)) \geqslant m + \frac{m}{4}.$$

Hromkovič *et al.* even proved a stronger claim (Hromkovič *et al.*, 2009). However, for our needs the above bound is fine and we omit the proof for the sake of not getting too formal.



(a) An instance created by ADV. (b) The strategies D_0, D_2, and D_{-3}.

Fig. 5. An example of how ADV can make sure that every second step of any path computed by A is not diagonal, and three strategies from $\mathcal{D}$.

So far, we have shown a lower bound on the costs of any online algorithm. But this is not sufficient to argue that the competitive ratio of any such algorithm is bad. It remains to show that there also always exists an optimal (offline) solution that has lower costs. To this end, we introduce the following set of strategies. For every $i \in \{1, \dots, \sqrt{m}\}$, the strategy D_i makes i horizontal steps at the beginning. Afterwards, it makes a diagonal step whenever possible. If it hits an obstacle, it bypasses it by making a horizontal step immediately followed by a vertical step. Eventually, D_i hits the right border. It then makes i vertical steps to reach the lower right corner. Analogously, for $i \in \{-\sqrt{m}, \dots, -1\}$, the strategy D_i makes i vertical steps at the beginning and i horizontal steps at the end, acting as above in between. Moreover, the strategy D_0 does not make any non-diagonal steps at the beginning or at the end unless it hits an obstacle. We now set

$$\mathcal{D} = \{D_i \mid i \in \{-\sqrt{m}, \dots, 0, \dots, \sqrt{m}\}\}.$$

Obviously, we are dealing with $2\sqrt{m}+1$ strategies in total, sample strategies are shown in Fig. 5b. In the following, for the ease of presentation, we assume that $\sqrt{m}$ is a natural number[3]. We are now interested in answering the following question.

What can we claim about the smallest costs from the set of these solutions?

Recall that we call the number of horizontal steps (h) the delay of a solution. Our idea is to first calculate the average delay. Let Obs_i denote the number of obstacles strategy D_i hits on its way through the grid. We conclude that

$$\text{Delay of strategy } D_i = \mathtt{h} = i + \mathrm{Obs}_i,$$

because D_i makes exactly one horizontal move for every obstacle it hits. We sum the costs for all strategies and get

$$\text{Total delay} = \sum_{i=-\sqrt{m}}^{\sqrt{m}} \big(i + \mathrm{Obs}_i\big) = \sum_{i=-\sqrt{m}}^{\sqrt{m}} i + \sum_{i=-\sqrt{m}}^{\sqrt{m}} \mathrm{Obs}_i.$$

Observe that we know that there are, in total, exactly m obstacles in the grid for any instance (Observation O1). It follows that the sum of all obstacles that are hit cannot be larger than m.

$$\begin{aligned}\text{Total delay} &\leqslant \sum_{i=-\sqrt{m}}^{\sqrt{m}} i + m = 2 \cdot \sum_{i=1}^{\sqrt{m}} i + m \\ &= 2\frac{\sqrt{m}\,(\sqrt{m}+1)}{2} + m = \sqrt{m}(\sqrt{m}+1) + m \\ &= \sqrt{m}\,(\sqrt{m}+1) + \sqrt{m} \cdot \sqrt{m} = \sqrt{m}\,(2\sqrt{m}+1).\end{aligned}$$

[3]The following proof works without this restriction, but we think that it is easier to follow this way.

If we now divide this number by the number of all strategies, we get

$$\text{Average delay } \leqslant \frac{\sqrt{m}\,(2\sqrt{m}+1)}{2\sqrt{m}+1} = \sqrt{m}.$$

By Observation O5, it follows that the average costs are at most $m + \sqrt{m}$. Therefore, there has to exist a single strategy that has costs of at most $m + \sqrt{m}$.

Let us sum up what we have just learned. We showed the existence of an adversary that can make sure that any solution computed by any online algorithm has costs of at least $m + m/4$. On the other hand, we also know that there always exists a solution that has costs of at most $m + \sqrt{m}$. What does this mean for the competitive ratio of any online algorithm A for the problem job shop scheduling with 2 unit length tasks?

To answer this question, we simply plug these two values into the formula for competitiveness. There exists an instance $\overline{x}$ such that, for any online algorithm A, we have

$$\frac{\text{cost}\,(A(\overline{x}))}{\text{Opt}\,(\overline{x})} \geqslant \frac{m+m/4}{m+\sqrt{m}} = \frac{5}{4}\left(\frac{m}{m+\sqrt{m}}\right) = \frac{5}{4}\left(\frac{1}{1+\frac{1}{\sqrt{m}}}\right).$$

As computer scientists, we are interested in how this function behaves with respect to an increasing input m. This means that we ask

What does this expression converge to as m grows?

Obviously, it converges to $5/4$ and we may therefore state that there exist inputs on which the solution computed by A is almost 25% worse than the optimal solution.

2.6. *Using Randomization Increases Performance*

We now search for a way that overcomes the above drawback and introduce the concept of randomized algorithms which are allowed to base some of their computations on random decisions. To be precise and formally correct, in the following, we would have to talk about the expected competitive ratio of the investigated randomized online algorithm. As a matter of fact, it is known that it tends to 1 with m tending to infinity (Hromkovič *et al.*, 2009), but we do not want to introduce the formal definitions of random variables and expected values which are needed for the proof. Therefore, we use a different approach as suggested by Hromkovič (2009).

Let us first get back to the model of ADV .

- ADV knows A,
- but he does not know A's random decisions in advance
- therefore, ADV has to cope with a lot of "different" algorithms at once.

Indeed, if ADV knows the source code of A, it does not really help him to know that there exists a code block which shows some instructions that merely say "Choose one out of d possible strategies, each with the same probability". Which one will actually be chosen will only be known at runtime. Therefore, this is consistent with our intuition about the adversary.

If an algorithm A is not allowed to use randomness and we show the existence of an adversary that is able to construct a bad input instance as we have shown in the previous section, A will *always* perform bad on this input. But actually, the property that makes this input "hard to deal with" might be very natural and thus appear frequently in practical applications. For randomized algorithms, we want to make statements of the form "the algorithm only performs bad on x with a very low probability", but in most of the cases its costs are low. Only very rarely, its costs are high for some fixed x and therefore, x is not "bad" in the above sense.

We now consider the randomized algorithm R that has the following strategy.

R chooses one out of the strategies from $\mathcal{D}$ and follows it. Here, any possible strategy gets chosen with the same probability.

Since there are exactly $2\sqrt{m}+1$ strategies in total to choose from, we have

$$\text{Probability to choose strategy } D_i = \frac{1}{2\sqrt{m}+1},$$

for every $i \in \{-\sqrt{m}, \ldots, 0, \ldots, \sqrt{m}\}$. For our next step, we need the following observation.

Let d be the average value of n natural numbers. Then at least half of these numbers have a value less than $2d$.

Informally, we can just draw the number line from 0 to $2d$ on the blackboard. Even if $n/2-1$ numbers out of the n numbers have a value of 0 (are as small as possible), not all remaining numbers can have a value of $2d$ or d is not the average (Fig. 6). Alternatively: to formally prove the claim, the students must understand the concept of an *indirect proof*. In this case, towards contradiction, suppose the claim is not true and at least $n/2+1$ numbers have a value of at least $2d$. It follows that

$$\begin{aligned} d &\geqslant \frac{(\frac{n}{2}+1)\cdot 2d + (\frac{n}{2}-1)\cdot 0}{n} = \frac{\frac{2dn}{2}+2d+0}{n} \\ &= \frac{dn}{n} + \frac{2d}{n} = d + \frac{2d}{n} > d, \end{aligned}$$

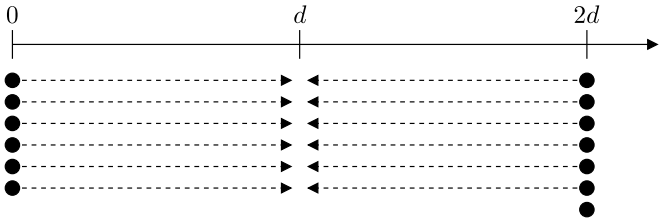


Fig. 6. An example to prove the claim. We can pair every element of value 0 with an element of value $2d$. Obviously, their average is d, but at the end, there will be at least one element left with a value of $2d$. Therefore, the average of all numbers cannot be d. Clearly, if we increase any of the values, the claim gets only more obvious.

which is a contradiction in itself. It is important to see that this statement can be generalized in a straightforward way.

Again, let d be the average value of n natural numbers. Then at least a $(1 - \frac{1}{c})$th of these numbers have a value of less than $c \cdot d$ for any natural number c.

The corresponding proof follows from the fact that

$$d \geqslant \frac{(\frac{n}{c} + 1) \cdot cd + (1 - \frac{n}{c} - 1) \cdot 0}{n} = \frac{\frac{cdn}{c} + cd + 0}{n}$$
$$= \frac{dn}{n} + \frac{cd}{n} = d + \frac{cd}{n} > d.$$

In the previous section, we have already seen that the average delay of the strategies in $\mathcal{D}$ is at most $\sqrt{m}$. Applying the observation above, we get that, e. g.,

at most half of the solutions have a delay greater than $2\sqrt{m}$ and at most a tenth has has a delay greater than $10\sqrt{m}$.

Furthermore, we know that any solution has costs of at least m (Observation O4). Therefore, at least a $9/10$th of the strategies have a competitive ratio of at most

$$\frac{m + 10\sqrt{m}}{m} = 1 + \frac{10}{\sqrt{m}},$$

which tends to 1 with an increasing m. This means that the competitive ratio of R tends to 1 with probability at least $9/10$.

The competitive ratio of R is compared to the one of A in Fig. 7. Actually, we can

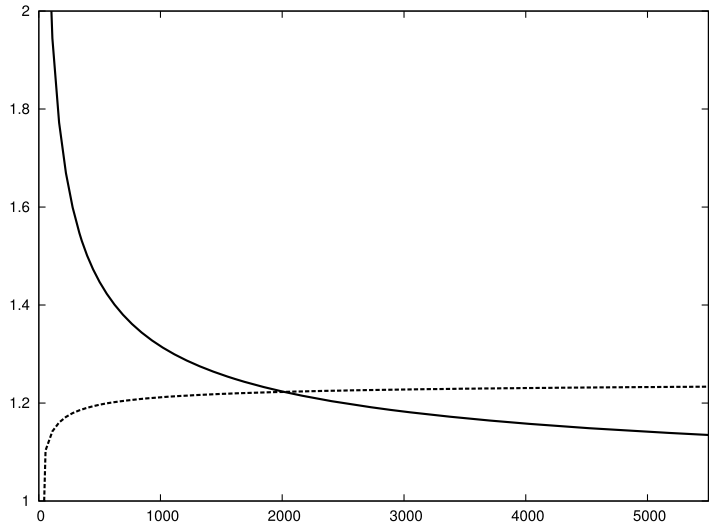


Fig. 7. Comparison of both competitive ratios, where the solid line shows the competitive ratio of R and the dashed line shows the ratio of A.

make the same statement for any probability that is constant with respect to m. Thus, we can also claim that the competitive ratio of R tends to 1 with probability $999/1000$. The only difference is that in this case the constant of the upper bound is 1000 instead of 10.

3. Conclusion

In this paper, we proposed a concrete way how to teach secondary school students the basic concepts of online computation, worst-case analysis via an imaginary adversary, and the power of randomization. To do so in a comprehensible way, we omitted formal definitions and proofs whenever possible. We started with an easy example and slowly increased the technical difficulties by trying to involve the audience as much as possible.

At the end, we showed a rather deep result which is new to the majority of secondary school students, although it seems straightforward to us as computer scientists: Allowing an algorithm to make random decisions helps it to deal with an unknown future.

References

Borodin, A., El-Yaniv, R. (1998). *Online Computation and Competitive Analysis*. Cambridge University Press, New York.

Brucker, P. (1988). An efficient algorithm for the job-shop problem with two jobs. *Computing*, 40(4), 353–359.

Hromkovič, J. (2006). *Design and Analysis of Randomized Algorithms: Introduction to Design Paradigms*. Springer-Verlag.

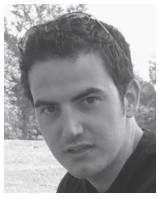
Hromkovič, J. (2009). *Algorithmic Adventures*. Springer-Verlag.

Hromkovič, J., Mömke, T., Steinhöfel, K., Widmayer, P. (2007). Job shop scheduling with unit length tasks: bounds and algorithms. *Algorithmic Operations Research*, 2(1), 1–14.

Irani, S., Karlin, A.R. (1997). *On Online Computation*. In: *Approximation Algorithms for NP-Hard Problems*, PWS Publishing Company, 521–564.

Manasse, M.S., McGeoch, L.A., Sleator, D.D. (1990). Competitive algorithms for server problems. *Journal of Algorithms*, 11(2), 208–230.

Sleator, D.D., Tarjan, R.E. (1985). Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2), 202–208.



D. Komm (1982), PhD student at the chair of Information Technology and Education, Department of Computer Science at ETH Zurich, studied computer science at RWTH Aachen University. His research interests focus on algorithmics and the advice complexity of online problems.